



TRABALLO FIN DE GRAO
GRAO EN ENXEÑARÍA INFORMÁTICA
MENCIÓN EN COMPUTACIÓN



Implementación de aprendizaje por refuerzo en robots con patas para aprender a caminar

Estudiante: Jorge Rivadulla Brey
Dirección: Martin Naya Varela
Dirección: Francisco Javier Bellas Bouza

A Coruña, noviembre de 2023.

Para todos aquellos que, aún cuando parecía imposible, confiaron en que lo conseguiría.

Agradecimientos

En primer lugar me gustaría agradecer a mis tutores por permitirme conocer y profundizar en unos campos tan interesantes como son la robótica y el uso de inteligencia artificial para el aprendizaje.

También me gustaría agradecer a mis compañeros de facultad y amigos sin los cuales estos años no hubieran sido lo mismo. Gracias a su apoyo y ayuda es que hoy me encuentro aquí.

No puedo olvidarme de agradecer también a mi familia, que siempre me ha apoyado en lo bueno y en lo malo.

Por último quiero agradecer a dos personas que sin duda son las que, junto a mí, más han sufrido este TFG. Gracias Iago y Bea por apoyarme y más importante aguantarme durante todo este tiempo, escuchando casi todos los días los avances o la falta de ellos que había en mi TFG, sin vuestra infinita paciencia esto no hubiera sido posible.

Resumen

Este trabajo de fin de grado se enmarca dentro del campo de la inteligencia artificial y de la robótica. El grupo integrado de ingeniería de la UDC (GII) ha llevado a cabo diferentes investigaciones en este campo con la intención de comprobar el funcionamiento de algoritmos evolutivos a la hora de obtener la morfología óptima para que los robots con patas aprendan a caminar. A partir de esos experimentos, y usando las mismas morfologías de los robots, nace el interés por conocer como serían los resultados usando, en este caso, aprendizaje por refuerzo. Para ello, se utiliza el simulador CoppeliaSim donde se encuentran ya implementadas diferentes morfologías entre las que destaca la del robot de cuatro patas, por su estabilidad. El objetivo final de este proyecto es el de investigar diferentes tipos de aprendizaje por refuerzo, decidir cuál es el de mayor interés e implementarlo para conseguir que dicho robot sea capaz de aprender a caminar de forma autónoma.

Abstract

This final degree project is part of the field of artificial intelligence and robotics. The integrated engineering group of the UDC (GII) has carried out several experiments in this field with the intention of verifying the operation of evolutionary algorithms when it comes to obtaining the optimal morphology for robots with legs to learn to walk. From these experiments, and using the same morphologies of the robots, interest was born in knowing what the results would be like using, in this case, reinforcement learning. For this, the CoppeliaSim simulator is used where different morphologies are already implemented, among which the four-legged robot stands out, due to its stability. The final objective of this project is to investigate different types of reinforcement learning, decide which is of greatest interest and implement it to ensure that said robot is capable of learning to walk autonomously.

Palabras clave:

- Aprendizaje por refuerzo
- Qlearning
- Robots cuadrúpedos
- Tasa de aprendizaje
- E-greedy
- CoppeliaSim

Keywords:

- reinforcement learning
- Qlearning
- quadruped robots
- Learning rate
- E-greedy
- CoppeliaSim

Índice general

1	Introducción	1
1.1	Motivación	1
1.2	Objetivos	4
2	Estudios previos	5
2.1	Aprendizaje por refuerzo	5
2.1.1	Explotación vs exploración	6
2.1.2	On-policy	7
2.1.3	Off-Policy	7
2.2	Qlearnig	9
2.2.1	Tasa de aprendizaje	11
2.2.2	Tasa de descuento	11
2.2.3	E-greedy.	12
2.2.4	Algoritmo	12
2.3	Proximal Policy Optimization	13
2.3.1	PPO-Penalty	14
2.3.2	PPO-Clip	14
3	Fundamentos tecnológicos	18
3.1	Máquina virtualBox	18
3.2	CoppeliaSim	18
3.3	Python y librerías	18
3.4	CESGA	20
3.5	Singularity	20
4	Metodología	21
4.1	CoppeliaSim	21
4.2	Programar robot	22

4.3	CESGA	22
4.4	Comprobación de resultados	22
4.5	Redacción de memoria	22
4.6	Almacenamiento de resultados	22
5	Desarrollo	24
5.1	CoppeliaSim	24
5.2	Código de aprendizaje de robot Cuadscene	26
5.2.1	Primera iteración: Funciones básicas	26
5.2.2	Segunda iteración: Cuadscene2	28
5.2.3	Tercera iteración: Cuadscene5 y Cuadscene7	31
5.2.4	Cuarta iteración: Cuadscene8	35
5.2.5	Quinta iteración: Cuadscene11	36
5.2.6	Sexta iteración: Cuadscene14	37
5.3	Aprendizaje en el CESGA	38
6	Resultados	40
6.1	Cuadscene5	40
6.2	Cuadscene7	41
6.3	Cuadscene8	42
6.4	Cuadscene11, código sin crecimiento	44
6.4.1	Ángulo cuarenta y cinco grados	46
6.5	Cuadscene14, código con crecimiento	47
6.5.1	Estado inicial	48
6.5.2	Piernas en treinta grados	49
6.5.3	Comparación de resultados	51
7	Conclusiones	54
7.1	Conocimientos adquiridos	55
7.2	Trabajo a futuro	56
Lista de acrónimos		58
Glosario		59
Bibliografía		60

Índice de figuras

1.1	Esquema básico de un algoritmo de aprendizaje por refuerzo	3
2.1	Ejemplo de policy de comportamiento (fase de exploración) y policy de objetivo (fase de explotación)	8
2.2	Explicación de la fórmula del Qlearning.	9
2.3	Ejemplo de matrizQ	10
2.4	Pseudocódigo de el algoritmo E-greedy.	12
2.5	Pseudocódigo del Qlearning.	13
2.6	Pseudocódigo del algoritmo PPO-Clip.	15
2.7	Política de actualización de PPO-Clip	15
2.8	Valor de L.	15
2.9	Valor de L.	16
3.1	Ejemplo robot 4 patas.	19
3.2	Ejemplo de un robot de dos patas en CoppeliaSim mostrando también el menú de joints.	19
4.1	Diagrama de la realización del TFG.	21
4.2	Imagen del repositorio de Github	23
5.1	Bublebot dentro de un laberinto para probar el funcionamiento de los scripts .	24
5.2	Robot coche	25
5.3	Robot coche dentro del laberinto	25
5.4	Diagrama de flujo que representa el funcionamiento del algoritmo	27
5.5	Ejemplo de matrizQ de nuestro experimento	29
5.6	Función que muestra el refuerzo.	36
5.7	Ejemplo tabla de resultados matrizQ.	37
5.8	Aspecto del repositorio donde almacenamos el código dentro del servidor. .	38

5.9	Aspecto de la carpeta de scripts.	39
5.10	Comandos usados para lanzar un experimento.	39
6.1	Ejemplo de resultado, se ven los valores que utiliza el código.	40
6.2	Ejemplo de valores que nos devuelve el código en cada iteración.	43
6.3	Imagen robot desplazándose con el experimento 25.	44
6.4	Imagen robot con las patas en posición inicial.	45
6.5	Gráfica refuerzo hasta 4500 episodios.	45
6.6	Gráfica del refuerzo en cada episodio tras el aprendizaje.	46
6.7	Imagen robot con las patas con cuarenta y cinco grados de inclinación.	46
6.8	Gráfica del refuerzo en cada episodio durante el aprendizaje en cuarenta y cinco grados.	47
6.9	Gráfica del refuerzo en cada episodio tras el aprendizaje para cuadrúpedo en ángulo cuarenta y cinco grados.	48
6.10	Imagen de un robot que se ha quedado atascado, uno de los posibles errores que ocurren por que el cuadrúpedo se cae hacia un lado, en este caso hacia atrás	48
6.11	Gráfica del refuerzo en cada episodio tras el aprendizaje para cuadrúpedo con variable crecimiento en ángulo inicial 1.	49
6.12	Gráfica del refuerzo en cada episodio tras el aprendizaje para cuadrúpedo con variable crecimiento en ángulo inicial 2.	49
6.13	Imagen del robot con las patas con treinta grados de inclinación.	50
6.14	Gráfica del refuerzo en cada episodio tras el aprendizaje para cuadrúpedo con variable crecimiento en ángulo de treinta grados.	50
6.15	Gráfica del desplazamiento del código de 45 grados en cincuenta episodios. . .	51
6.16	Gráfica del desplazamiento del código de 45 grados en doscientos cincuenta episodios.	52
6.17	Gráfica del desplazamiento del código de 30 grados en cincuenta episodios. .	52
6.18	Gráfica del desplazamiento del código de 30 grados en doscientos cincuenta episodios.	53

Índice de tablas

3.1 Relación de librerías utilizadas	20
5.1 Relación entre array Arm[] y los joints a partir del Cuadscene5	29
5.2 Relación entre array Arm[] y los joints a partir del Cuadscene7	33

Capítulo 1

Introducción

En este primer capítulo exploraremos las motivaciones para la realización de este Trabajo Fin de Grado y los objetivos que se han tratado de cumplir

1.1 Motivación

Es de sobra conocido que vivimos en una época de continuo desarrollo e investigación de nuevas tecnologías, una gran parte de estas tienen relación directa o indirecta con la ingeniería informática y sus campos. Uno de los que más se ha desarrollado en estos últimos años, y que tiene un mayor potencial de crecimiento, es el relacionado con la inteligencia artificial. Este campo se podría definir como la capacidad, a partir de distintos tipos de algoritmos, de conseguir que los ordenadores sean capaces de aprender a realizar tareas simples o complejas por su cuenta [1].

Además, recientemente el campo de la IA a sufrido un gran auge con la aparición en escena de programas como ChatGPT de la empresa OpenAI, todo lo relacionado con este campo ha llegado al gran público, haciendo que muchos proyectos derivados de él reciban una renovada atención de la que no disponían antes, además de crear nuevos puestos de trabajo relacionados con ella [2].

La investigación y el desarrollo de las inteligencias artificiales tocan muchos campos, desde el relacionado con programas de visión artificial, como podría ser la creación de un algoritmo capaz de detectar ciertos objetos en imágenes, pasando por la creación de NPC dentro del mundo de los videojuegos que sean capaces de comportarse casi como humanos imitando nuestros comportamientos e incluso aprendiendo de nosotros. Como estos campos hay muchos más, dentro de los cuales, resulta de especial interés la robótica, ya que su impacto a nivel práctico es de gran relevancia.

Pese a que existen diferentes definiciones sobre la IA, en su artículo, [3] John McCarthy la describe de la siguiente manera: *“Es la ciencia y la ingeniería de la fabricación de máquinas*

inteligentes, especialmente programas informáticos inteligentes. Está relacionada con la tarea similar de usar computadoras para entender la inteligencia humana, pero la IA no tiene que limitarse a métodos que son biológicamente observables”. Otra definición más actual sería la dada por Russel y Norvig [4]. Según ellos existen cuatro tipos de inteligencia artificial: Los sistemas que piensan como humanos, aquellos que actúan como humanos, los que piensan de manera racional y los que actúan de manera racional [5].

Podemos definir un robot inteligente como un sistema autónomo compuesto por varios componentes, los sensores que reciben la información del entorno y los actuadores que serán los encargados de realizar las tareas con las que ha sido programado. Podemos tener robots que se encarguen de tareas en varios ámbitos, como la medicina o la educación. Estos robots son cada vez más una parte de nuestra vida cotidiana, conforme avanzamos en el estudio y desarrollo de nuevos modelos, van realizando tareas cada vez más complejas.

En este Trabajo de Fin de Grado nos vamos a centrar en el aprendizaje de robots con patas para que sean capaces de desplazarse por su cuenta. Este tipo de aprendizaje se puede realizar de varias maneras, pero para simplificar vamos a dividir el aprendizaje en dos grandes grupos, el primero será el que conglomera a los algoritmos evolutivos y el segundo al aprendizaje por refuerzo [6].

Si nos referimos a los algoritmos evolutivos [7], estos se inspiran en la evolución biológica de las especies, aplicados a la resolución de problemas complejos dentro del ámbito de la optimización. En lo referido al caso que nos atañe, el de los robots que se desplazan usando patas, los algoritmos evolutivos se pueden aplicar en el diseño automático de un sistema de control capaz de solucionar problemas de diversos tipos, desde simple del movimiento de sus extremidades, hasta la superación de obstáculos en diferentes terrenos.

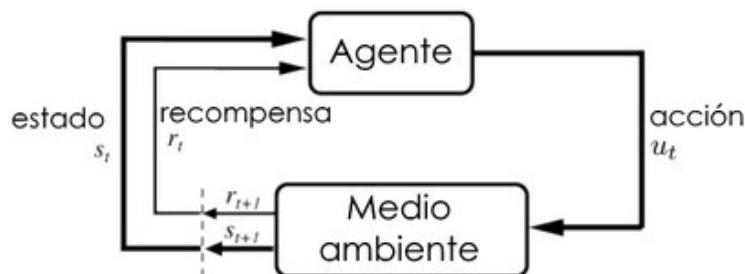
El control de estos robots es complejo debido al funcionamiento mecánico de sus patas, el uso de algoritmos evolutivos nos permite encontrar respuestas flexibles a los problemas que pueden ir apareciendo. En muchas ocasiones estas respuestas tienden a diferir de las que nosotros, los programadores, habíamos pensado, pero proporcionan soluciones adaptadas al entorno de forma automática.

Estos algoritmos basan su funcionamiento en la generación y posterior evolución de una población de individuos que representan diferentes estrategias y movimientos que puede realizar el robot. El algoritmo funciona obteniendo las soluciones de la población original de robots ante el problema y, usando ciertos criterios de evaluación estipulados por el programador, se decide cuáles soluciones y, por lo tanto, cuáles individuos, son los que han encontrado una mejor estrategia para resolver el problema que se la ha propuesto. Estos individuos se cruzan entre sí generando nuevos sujetos que volverán a obtener nuevas soluciones que podrán heredar de sus predecesores las características más beneficiosas para el aprendizaje del robot.

De esta manera, estos algoritmos son capaces de probar una gran cantidad de estrategias y movimientos, haciendo así que los resultados finales nos permitan que el robot con patas pueda superar los problemas que se le plantea. Además, esta metodología permite que el robot aprenda a superar problemas como el terreno variable que son cambiantes, ajustando sus movimientos, según los datos que recibe por sus sensores.

En la robótica orientada a robots con patas, esta clase de algoritmos ha demostrado ser bastante eficiente a la hora de obtener los mejores patrones de movimiento, solucionando los problemas del estilo de la sincronización entre patas, el equilibrio o la navegación por diferentes terrenos. Generando robots capaces de moverse de manera eficiente y estable.

El otro grupo de algoritmos, el cual engloba los que utilizaremos en el transcurso de este Trabajo Fin de Grado (TFG), son los denominados algoritmos de aprendizaje por refuerzo [8]. Estos algoritmos basan su funcionamiento en la premisa de que cada acción que el robot realiza recibe una recompensa, positiva o negativa, dependiendo de los resultados que produce, tal y como se puede apreciar en la figura 1.1 (página 3). A partir de estas recompensas, el robot va perfilando cuál es la sucesión de acciones que más le conviene hacer para optimizar la obtención de recompensas.



[Figure source: Sutton & Barto, 1998]

Figura 1.1: Esquema básico de un algoritmo de aprendizaje por refuerzo

Esta metodología es utilizada con muy buenos resultados a la hora de entrenar a robots con patas, ya que nos permite que aprenda y almacena cuál es la mejor acción posible a realizar para cada una de las distintas situaciones en las que se puede llegar a encontrar. Cabe decir que una ventaja que también posee sobre los algoritmos evolutivos es la capacidad de ser reentrenado en cualquier momento, permitiendo que un mismo individuo vaya desarrollando cada vez más sus capacidades. No entraremos en detalle del funcionamiento de estos algoritmos, ya que eso lo haremos en el apartado de Metodologías.

1.2 Objetivos

Desarrollar e implementar un sistema basado en aprendizaje por refuerzo capaz de enseñar a robots con patas a caminar de forma autónoma. Para esto estudiaremos los distintos tipos de aprendizaje por refuerzo y después decidiremos cuál de ellos utilizar para ser aplicado en robots con cuatro patas.

Además, se deberá crear un repositorio en el que almacenar todos los resultados de esta experimentación además de los códigos que los producen.

Capítulo 2

Estudios previos

En este apartado explicaremos más sobre el aprendizaje por refuerzo y dos de sus algoritmos, [Qlearning](#) y [Proximal Policy Optimization](#), los cuales fueron los que investigamos para decidir cuál de los dos implementar para entrenar a nuestros robots.

2.1 Aprendizaje por refuerzo

El aprendizaje por refuerzo [9] es un paradigma básico dentro del campo del aprendizaje por automático, y se utiliza aquí a la hora de entrenar robots con morfologías que basan su movimiento en el uso de patas. Basa su fórmula de aprendizaje en un "prueba y error", es decir, busca premiar a los movimientos que producen una acción que se entiende como positiva cuando cumple los objetivos determinados por el diseñador. Un ejemplo de esto sería un robot que trata de seguir una linea negra, si tras realizar una acción se encuentra sobre la linea negra se entiende que cumple el objetivo y se le premia. Con esto podemos decir que no solo aprende de los de los resultados correctos, sino que también de los errores. Mirando algo más en profundidad, podemos dividir el aprendizaje por refuerzo en seis fases diferenciadas, las cuales serían [10]:

1. Observación del entorno: En esta primera fase buscamos determinar las características del entorno que pueden ser importantes a la hora del desplazamiento de nuestro robot. Al mismo tiempo tenemos que definir los estados que pueden producirse al realizar las acciones.
2. Toma de decisiones: Ahora que conocemos el entorno y nuestros estados debemos decidir qué acción tomar. Esta acción nos hará pasar del estado en el que nos encontramos a otro de los posibles. Para la elección de qué acción es la mejor tomar debemos seguir una estrategia denominada policy. Podemos definir una policy como una estrategia o un conjunto de reglas cuyo objetivo es el de guiar las acciones de nuestro agente. Es

necesario usar una policy ya que esta será la encargada de guiar al agente en la toma de decisiones además de para que aprenda a realizar acciones con las que obtenga la máxima recompensa posible . Estas estrategias pueden ser muy variadas, ya que dentro del aprendizaje por refuerzo hay muchos tipos de algoritmos distintos.

3. Ejecución de la decisión tomada: Una vez la policy de nuestro algoritmo ha decidido cuál es la mejor acción a tomar, llevamos a cabo esa acción. Al realizar este paso es importante vigilar como afecta esa acción tanto al entorno como a nuestro propio estado.
4. Asignación de la recompensa: En esta fase pasamos a evaluar la acción, dando como resultado un valor de recompensa que puede ser positivo o negativo, dependiendo de la acción y del estado en el que se encuentre. Un ejemplo de esto puede ser un robot que se encargue de seguir una línea negra. Si tras hacer un movimiento el robot se encuentra sobre la línea negra el refuerzo será positivo, de no encontrarse sobre ella será negativo.
5. Mejora de nuestra estrategia: Con los datos obtenidos por la asignación de la recompensa podemos decidir si la acción tomada para el estado en el que nos encontrábamos ha sido buena o mala. De ser una acción con un resultado positivo en el sistema nos interesa que en situaciones similares (Es decir, cuando volvamos a encontrarnos en el mismo estado) se repita esta acción, mientras que no nos interesará que vuelva a ocurrir si el resultado en nuestro sistema ha sido negativo. Con el objetivo de ir afinando el funcionamiento de nuestro sistema deberemos ir actualizando nuestra política basándonos en las recompensas obtenidas.
6. Proceso iterativo: Para obtener una estrategia óptima habrá que realizar este bucle de acciones varias veces, con la idea de que con cada recompensa, sea positiva o negativa, se vaya afinando la toma de decisiones de la política que estamos utilizando.

Tal y como acabamos de explicar, un algoritmo de aprendizaje por refuerzo contiene cuatro elementos básicos, un agente, una política, una recompensa y una función de valor. Además, es importante puntualizar la existencia de dos fases dentro del aprendizaje por refuerzo, denominadas fases de explotación y exploración.

2.1.1 Explotación vs exploración

Dentro del funcionamiento de los algoritmo de aprendizaje por refuerzo hay que diferenciar dos etapas. La primera es, para cada situación o estado, utilizar la mejor de las acciones, en nuestro caso movimientos. Conocemos esta etapa como explotación. Mientras tanto, la etapa en la que probamos las diferentes políticas de manera aleatoria para entrenar la matriz es la que se conoce como exploración. Es muy importante saber encontrar el equilibrio entre ambas etapas.

Ahora pasaremos a comentar los dos principales grupos de políticas [11].

2.1.2 On-policy

Los algoritmos comprendidos dentro de **On-Policy** son aquellos que evalúan y mejoran la misma policy que utilizan para seleccionar que acciones deben realizar en cada estado [12]. De esta manera intentarán evaluar y mejorar la misma política que el agente utiliza para la selección de acciones. Se puede decir que estos algoritmos usan la misma policy tanto para la fase de exploración como para la posterior fase de explotación. Algunos ejemplos de algoritmos On-Policy son Policy Iteration, Value Iteration, Monte Carlo.

Estas políticas han sido implementadas en varios campos, entre los que destacan el aprendizaje de robots de movimiento, la **Inteligencia Artificial (IA)** de los NPC de videojuegos o en sistemas de navegación de vehículos autónomos, donde estas políticas son importantes para ajustar el comportamiento que tiene el vehículo al tiempo que recibe estímulos del entorno, de esta manera se le entrena para que evite obstáculos y optimice su ruta.

2.1.3 Off-Policy

A diferencia de los algoritmos On-Policy, los **Off-Policy** tienen dos policies completamente distintas. La primera es la policy de destino, que se encarga de la fase de exploración del algoritmo. Su objetivo es evaluar y entrenar al agente para que decida cuál es la mejor acción para cada uno de los estados en los que se puede encontrar. La segunda policy se encargará de la parte de explotación del algoritmo, cuya finalidad es guiar al agente con los datos obtenidos en la fase anterior. En resumen, la diferencia es que tenemos policies distintas para cada una de las partes del algoritmo, tal y como se puede apreciar en la figura 2.1 (página 8). Algunos ejemplos de algoritmos de aprendizaje Off-Policy son Qlearning, Sarsa, etc.

Estos algoritmos suelen utilizarse en casos en los que se puede haber escogido los datos del entorno con anterioridad al inicio del aprendizaje, es decir, que tenemos datos de un aprendizaje previo que queremos reutilizar.

También suele usarse Off-Policy en casos donde la exploración en tiempo real es muy costosa, por lo que necesita de depender de datos anteriormente almacenados para un correcto aprendizaje del agente. Otras situaciones en las que pueden ser utilizados es en aprendizaje en línea, donde los datos de interacción en tiempo real son costosos o peligrosos de obtener, como entrenar a un vehículo autónomo para que se conduzca por zonas con una gran cantidad de tráfico, estos algoritmos nos ayudan a que el agente obtenga información de un banco de datos obtenido con anterioridad.

Algunos beneficios de los métodos de Off-Policy son los siguientes [12, 13, 14]:

1. Exploración continua: como un agente está aprendiendo otra política, puede usarse para

continuar la exploración mientras aprende la política óptima.

2. Aprendizaje por demostración [12]: A diferencia de otros algoritmos donde el agente aprende únicamente a base de prueba y error, el aprendizaje de la demostración se basa en que el agente aprenda también observando a otro agente o a un experto realizar las acciones.
3. Aprendizaje paralelo: El aprendizaje en paralelo se basa en que varios agentes interactúan con el entorno al mismo tiempo siguiendo las mismas policies. Los resultados obtenidos por todos ellos se almacenan de manera conjunta, haciendo que el aprendizaje sea más rápido y eficiente.
4. Permiten la reutilización de datos: Los algoritmos Off-Policy permiten volver a trabajar con bases de datos de información ya obtenida. Un ejemplo de esto puede ser la reutilización de una MatrizQ ya entrenada con anterioridad a la que le añadimos nuevos estados o acciones. Esto nos ahorra una gran cantidad de tiempo. Ya que obtener nuevos datos del entorno puede ser muy costoso
5. Mayor Eficiencia en la Exploración: Estos algoritmos dan más facilidades a la hora de explorar de una manera más eficiente los resultados de las acciones realizadas en diferentes estados, puesto que no están limitadas por las realizadas en tiempo real.

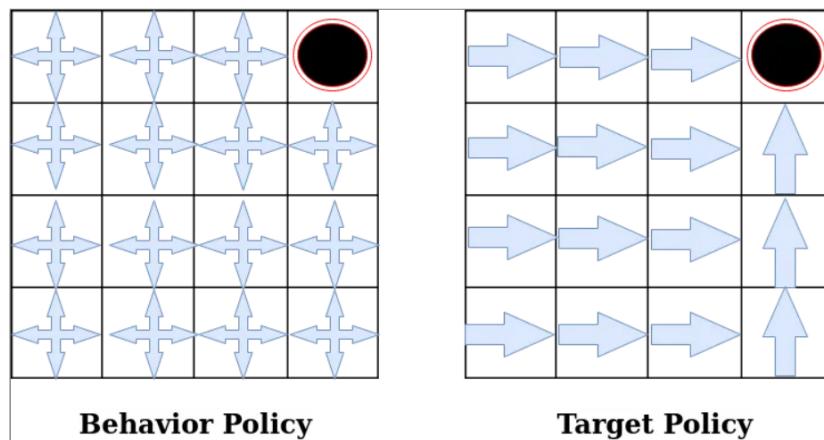


Figura 2.1: Ejemplo de policy de comportamiento (fase de exploración) y policy de objetivo (fase de explotación)

De esta manera, se puede decir que algoritmos "Off-Policy" pueden explorar con mayor eficacia el espacio de acciones, ya que no están limitados por las acciones tomadas en tiempo real. Pueden aprender de una variedad más amplia de situaciones y, por lo tanto, explorar estrategias más diversas.

Dentro de los algoritmos por refuerzo decidimos centrar nuestra investigación en dos de ellos, estos nos parecían ser los mejores y más útiles para cumplir nuestro objetivo de entrenar un robot con patas para que aprendiera a caminar. Estos algoritmos son Qlearni_g y Proximal Policy Optimization.

2.2 Qlearni_g

Qlearni_g es un algoritmo Off-Policy por lo que, tal y como hemos explicado antes, para dilucidar cuál es la política óptima para su correcto funcionamiento, usa una política distinta mientras el agente se encuentra aprendiendo que la que utilizará después para continuar con la estimación [9].

Con esto queremos decir que nuestro algoritmo tiene dos fases en las que el agente realiza distintos comportamientos. Primero nos encontramos con la fase de exploración, cuyo objetivo es probar las diferentes policies de modo que el agente aprenda cual es la mejor de las acciones para cada estado. Posteriormente procedemos con la fase de explotación, donde se comprobará en que estado se encuentra el agente y cuál es la acción que debe realizar.

El algoritmo de Qlearni_g se basa en el uso de una matriz a la que denominaremos matrizQ para la estimación de la policy óptima. La Q de Qlearni_g viene de la palabra inglesa Quality. El cálculo del nuevo valor de Q se realiza mediante la siguiente expresión: 2.2.

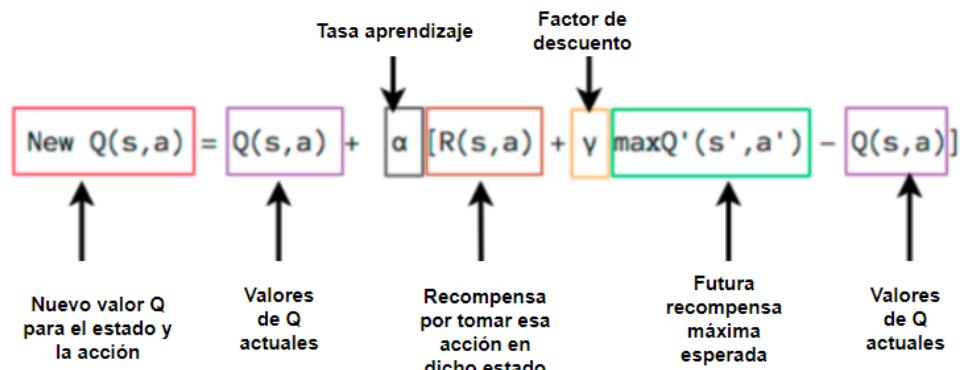


Figura 2.2: Explicación de la fórmula del Qlearni_g.

La matrizQ relaciona los estados (filas de la matriz) y las acciones (columnas de la matriz) con su valor esperado tal y como se ve en la figura 2.3. Con esto nos referimos a que, para cada estado, buscamos encontrar cual es la acción de mayor valor, que se obtienen realizando cada una de las acciones posibles varias veces. La decisión de cuales son los estados y acciones es tomada por los programadores. En un principio los valores de la matriz serán nulos, esto irá cambiando durante la fase de exploración.

Q	Acción1	Acción2	Acción3	Acción4
Estado1				
Estado2				
Estado3				
Estado4				

Figura 2.3: Ejemplo de matrizQ

El valor esperado del que hablamos antes es el equivalente a la recompensa que esperamos obtener al realizar una acción encontrándonos en determinado estado. Esas recompensas vienen determinadas por varios factores entre los que se encuentra el refuerzo, un valor que se obtiene de manera positiva si tras realizar la acción en ese estado se evalúa que el resultado es positivo, y negativo si por consiguiente el resultado es negativo.

En resumen, este algoritmo busca almacenar en la matrizQ los valores positivos , es decir, las recompensas, que se obtienen al realizar ciertas acciones evaluadas como positivas en esos estados y también los negativos, cuando una acción no es evaluada como positiva para cierto estado. La matriz se va actualizando cada vez que se realiza una acción, esto funciona de la siguiente manera:

Lo primero que se debe hacer es observar en que estado nos encontramos antes de realizar cualquier acción. Tras hacerlo realizamos una acción. La decisión de qué acción tomar puede determinarse de varias formas o con distintas estrategias de las que hablaremos más adelante. Tras realizar esta acción se obtiene la recompensa actual y las recompensas futuras dadas por el descuento. Con toda esta información que hemos obtenido, además de con ciertas variables de las que hablaremos ahora, se actualizan los valores de la matriz siguiendo la fórmula que nos muestra la figura 2.2 (página 9).

Tal y como vemos en la fórmula, el nuevo valor de la matrizQ en la posición limitada por el estado en el que nos encontrábamos antes de realizar la acción, y la acción que hemos realizado, es igual al valor que ya había más un valor alfa, del que hablaremos más adelante, que multiplica a todo lo siguiente, que es el valor del refuerzo para esa acción y ese estado más un valor gamma por el máximo del estado en el que nos encontramos tras hacer la acción menos el valor inicial.

Cabe destacar que uno de los valores más importantes de obtener y de los que más pueden hacer variar el resultado de los algoritmos es el refuerzo. Este valor del que ya hemos hablado antes será positivo si al haber realizado esa acción en ese estado valoremos que el resultado es positivo para los objetivos. Esa evaluación debe de ser estudiada y preparada antes de que

se empieza a entrenar, ya que pueden darse casos en los que el programador piense que un comportamiento no debería ocurrir de manera natural cuando, por el contrario, sí que ocurre. Esto produciría que se generasen valores de refuerzo falsos y se produjera un mal aprendizaje de la matriz.

Los dos factores de los que no hemos hablado que aún quedan en la función son el valor gamma, o tasa de descuento, y el valor alfa, o tasa de aprendizaje. Estos factores se conocen como hiperparámetros, diseñados por el programador, se encargan de regular el aprendizaje. Ambos parámetros siempre deben de tomar valores entre cero y uno [15].

2.2.1 Tasa de aprendizaje

En la fórmula de la Figura 2.2 (página 9) podemos observar que para actualizar el valor de la matriz Q hay dos sumandos, el primero sería el valor que ya había en esa posición y el producto en el que interviene la tasa de aprendizaje.

Esto nos permite intuir que el valor de la tasa de aprendizaje influye de gran manera en el valor final de la matriz Q , de modo que cuanto mayor sea la tasa de aprendizaje mayor será la velocidad con la que aumente de valor Q . De ahí el nombre de tasa de aprendizaje, ya que es la encargada de regular la velocidad de aprendizaje de la matriz.

Esto puede llevar a engaños, haciéndonos pensar que una tasa de aprendizaje con un valor muy alto va a ser siempre la mejor opción a la hora de que nuestra máquina aprenda lo más rápido posible, pero no es así. Pongamos de ejemplo una función de refuerzo con una variable de aprendizaje alta, esto nos hará llegar en un menor numero de pasos de tiempo al valor máximo que puede llegar a alcanzar la función de refuerzo, pero al hacerlo puede sobrepasar ese valor produciéndonos un resultado erróneo, haciéndonos muy difícil alcanzar nuestro objetivo.

Esto tampoco quiere decir que una tasa de aprendizaje baja sea la mejor opción, ya que tardaríamos demasiado en alcanzar ese valor máximo que nos interesa. Así que como en casi todos los ámbitos, lo mejor será tener un punto intermedio, un valor que sea suficientemente bajo como para no sobrepasar el objetivo, pero que sea lo suficientemente grande como para no necesitar demasiadas iteraciones para conseguir el resultado objetivo.

2.2.2 Tasa de descuento

Como se puede apreciar, si nos fijamos en la Figura 2.2 (página 9) la tasa de descuento multiplica a los valores futuros de la matriz. Entonces, cuanto más cercana a cero sea el valor de este factor, menos importancia tendrán los valores tras las siguientes acciones de la matriz al cálculo de la recompensa. Por el contrario, cuanto más se acerquen a uno más importancia tendrán estos valores a la hora de dictaminar el valor final de la recompensa.

Sabiendo esto podría parecer una buena decisión el usar siempre tasas de descuento con valores muy altos, de esta manera el agente no solo se centra en realizar la mejor acción para el estado actual, sino que también la acción que desembocarían en el mejor posible siguiente estado. Sin embargo, y como en el caso anterior, esto no es tan simple. Tal y como se explicó antes, cuanto mayor sea la tasa de descuento mayor será la influencia de los estados futuros en el actual, pero no en todos los estados las acciones tienen las mismas repercusiones y una tasa demasiado alta produciría que cierta información irrelevante corrompiera y acabaría influyendo en el valor del estado actual.

2.2.3 E-greedy.

Este algoritmo es el más comúnmente implementado junto con el Qlearning [9] a la hora de trabajar con las fases de exploración y explotación. Esto pasa por ser muy sencillo de implementar y de comprender. Para su funcionamiento vamos a añadir un nuevo hiperparámetro E que en nuestro caso inicializaremos a 1. Este hiperparámetro se encarga de controlar la probabilidad de que la acción que realice el algoritmo sea aleatoria. Al mismo tiempo tendremos una probabilidad $1-E$ de que la acción elegida sea la mejor posible para ese estado.

Conforme se van realizando pasos del experimento, el valor de E va disminuyendo con el objetivo de que tras realizar un número de pasos decididas por el programador pasemos a solo entrenar la matriz con las mejores acciones posibles. Podemos ver el funcionamiento del algoritmo en la siguiente figura 2.4(página 12).

Algoritmo 2: Selección de acciones con ϵ -greedy

```

1 Seleccionar un número aleatorio  $r$  entre 0 y 1
2 Si  $r < \epsilon$ :
3     Seleccionar una acción aleatoria  $a$ 
4 En otro caso:
5     Seleccionar la acción  $a$  con mayor valor esperado
         $a = \arg \max_a Q(s, a)$ 
6 Devolver  $a$ 
```

Figura 2.4: Pseudocódigo de el algoritmo E-greedy.

2.2.4 Algoritmo

Para finalizar de hablar de Qlearning me parece interesante el mostrar un pseudocódigo de este algoritmo como el que se ve en la figura 2.5 (página 13).

Vemos que en el algoritmo se diferencian episodios y pasos, siendo los pasos el realizar una acción, con todos lo que ello conlleva y ya hemos explicado, mientras que un episodio

sería la realización de todos los pasos necesarios hasta llegar a nuestro objetivo, que puede ser, por poner un ejemplo, alcanzar una casilla determinada. Tras realizar un episodio se volvería al estado inicial, se reinicia el agente, volviendo a realizar todos los pasos anteriores.

Algoritmo 1: Q-Learning

```

1   Inicializar  $Q$  arbitrariamente (generalmente a 0)
2   Por cada episodio repetir:
3       Inicializar el estado  $s$ 
4       Por cada paso en el episodio repetir:
5           Según la política de selección (por ejemplo,
               $\epsilon$ -greedy) seleccionar una acción  $a$ 
6           Ejecutar la acción  $a$ , observar la recompensa  $r$ 
              y el nuevo estado  $s'$ 
7           Actualizar  $Q$ :
8                $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_a(Q(s', a) - Q(s, a))]$ 
9           Actualizar  $s \leftarrow s'$ 
10      Hasta que  $s$  sea un estado final

```

Figura 2.5: Pseudocódigo del Qlearning.

2.3 Proximal Policy Optimization

Este algoritmo [16], también denominado Proximal Policy Optimization (PPO), fue diseñado por Schulman y es el sucesor del algoritmo Trust Region Polizy Optimization (TRPO) , con el objetivo de obtener la misma eficiencia que su predecesor pero con menos cálculos, y es un Policy Gradient Method [17]. Además, cabe destacar que a diferencia del algoritmo de Qlearning se trata de un algoritmo On-policy con todo lo que ello conlleva [18]. Estos métodos son muy populares en los aprendizajes por refuerzo. Se basa en el uso de un gradiente ascendente cuyo objetivo es seguir las políticas buscando la mayor recompensa posible. Aun así, es bastante importante de destacar que acarrean ciertos problemas con las optimizaciones de primero orden, cuyas derivadas tienden a no ser muy precisas en áreas curvas. Esto puede generar ciertos excesos de confianza en el algoritmo que estropeen el desarrollo de nuestro aprendizaje.

El algoritmo de TRPO se trata de un método iterativo cuyo objetivo es el de obtener las máximas recompensas esperadas. Esto lo hace modificando la política mediante la búsqueda de una función que va a realizar el trabajo de límite inferior de la deseada y actualizando en su dirección ascendente. Para esto utiliza, tal y como hemos explicado antes, un paso de gradiente ascendente con todo lo que ello conlleva.

Utilizando este método, el diseñador busca actualizar la política a seguir de una manera gradual, con el objetivo de evitar los problemas a los que nos referimos con anterioridad, y de esa manera nos aseguramos de que cada paso que realizamos es el ideal. Aun así, este algoritmo no está exento de problemas, entre los que destaca la necesidad de calcular la derivada de segundo orden y, por extensión, su inversa, dos operaciones con un coste computacional muy alto.

El algoritmo de PPO busca encontrar el equilibrio entre la facilidad de implementación, la complejidad de muestra y la facilidad de ajuste [17]. Esto lo conseguimos actualizando tras cada iteración la función de recompensa y disminuyendo su valor, usando únicamente para ello la primera derivada.

Este algoritmo tiene dos variantes, PPO-Penalty y PPO-Clip que pasaremos a explicar ahora [17].

2.3.1 PPO-Penalty

Esta variante resuelve una aproximación actualizada con restricciones de Kullback-Leiber (KL) como TRPO. Estas son un componente que se utiliza con el objetivo de controlar cuando se permite que se cambie de una política nueva durante la actualización en comparación con la política anterior. Con esto evitamos que se produzca una actualización que podrías estropear el aprendizaje.

Podemos definir la Divergencia de Kullback-Leiber como una medida que se encarga de cuantificar la diferencia entre dos distribuciones de probabilidad. En el contexto en el que estamos trabajando es utilizada como una forma de limitar cuanto permitimos que una nueva política se aleje de la política anterior. Con esto nos aseguramos de que, tal y como explicamos antes, los cambios de una política a otra no sean demasiado bruscos.

Si ponemos como ejemplo el uso de la divergencia de Kullback-Leibler en el algoritmo de PPO lo que realizamos es una relación de probabilidad entre las dos políticas, comparando con un valor Clip establecido. Si la relación está dentro de la horquilla del Clip, la actualización se realizará sin ningún cambio. Si no es así, la política sufrirá ciertas restricciones.

2.3.2 PPO-Clip

En cambio, esta variante no tiene un término de divergencia Kullback-Leibler (KL) y tampoco tiene ninguna restricción. A diferencia de la anterior, esta se basa en ir retocando la función objetivos con la idea de que, al eliminar los incentivos, la nueva política se vaya alejando de la política anterior. Las principales ecuaciones para el funcionamiento de PPO- Clip son las siguientes 2.6 , 2.7:

El valor de L se obtiene tal y como se ve en la figura 2.8 (página 15). Donde épsilon es un hiperparámetro que dice aproximadamente qué tan lejos se permite que la nueva política sé

Algorithm 1 PPO-Clip

-
- 1: Input: initial policy parameters θ_0 , initial value function parameters ϕ_0
 - 2: **for** $k = 0, 1, 2, \dots$ **do**
 - 3: Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
 - 4: Compute rewards-to-go \hat{R}_t .
 - 5: Compute advantage estimates, \hat{A}_t (using any method of advantage estimation) based on the current value function V_{ϕ_k} .
 - 6: Update the policy by maximizing the PPO-Clip objective:

$$\theta_{k+1} = \arg \max_{\theta} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^T \min \left(\frac{\pi_{\theta}(a_t | s_t)}{\pi_{\theta_k}(a_t | s_t)} A^{\pi_{\theta_k}}(s_t, a_t), g(\epsilon, A^{\pi_{\theta_k}}(s_t, a_t)) \right),$$

typically via stochastic gradient ascent with Adam.

- 7: Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg \min_{\phi} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^T \left(V_{\phi}(s_t) - \hat{R}_t \right)^2,$$

typically via some gradient descent algorithm.

- 8: **end for**
-

Figura 2.6: Pseudocódigo del algoritmo PPO-Clip.

$$\theta_{k+1} = \arg \max_{\theta} \mathbb{E}_{s, a \sim \pi_{\theta_k}} [L(s, a, \theta_k, \theta)],$$

Figura 2.7: Política de actualización de PPO-Clip

una al eje de la anterior.

$$L(s, a, \theta_k, \theta) = \min \left(\frac{\pi_{\theta}(a | s)}{\pi_{\theta_k}(a | s)} A^{\pi_{\theta_k}}(s, a), \text{clip} \left(\frac{\pi_{\theta}(a | s)}{\pi_{\theta_k}(a | s)}, 1 - \epsilon, 1 + \epsilon \right) A^{\pi_{\theta_k}}(s, a) \right),$$

Figura 2.8: Valor de L.

La fórmula mostrada por la figura 2.8 (página 15) es bastante compleja y puede llevar a errores y malentendidos a la hora de ser implementada, por eso existe otra fórmula más sencilla que podemos observar en la figura 2.9 (página 16).

Tal y como comentamos antes, PPO es un algoritmo On-Policy. Esto quiere decir que realiza la exploración (fase de aprendizaje) usando acciones de su última política de explotación. El número de acciones aleatorias que realiza durante el aprendizaje depende en o igual medida de las condiciones iniciales del entorno como del propio aprendizaje. Durante el transcurso de este la política va volviéndose cada vez menos aleatoria, ya que la regla de actualización lo motiva a repetir las acciones que ya le han dado buenas recompensas. Esto puede provocar

$$L(s, a, \theta_k, \theta) = \min \left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s, a), g(\epsilon, A^{\pi_{\theta_k}}(s, a)) \right),$$

$$g(\epsilon, A) = \begin{cases} (1 + \epsilon)A & A \geq 0 \\ (1 - \epsilon)A & A < 0. \end{cases}$$

Figura 2.9: Valor de L.

que la política queda atrapada en óptimos locales [18]. Algunos de los usos que han tenido los algoritmos de Proximal Policy Optimization son los siguientes:

1. Aprendizaje de Robots [19]: en este caso hablamos del aprendizaje de un brazo robótico virtual de 7 juntas modelado con un sistema real. Cuyo objetivo es el de alcanzar ciertas posiciones objetivo sin llegar a necesitar la supervisión de un ser humano
2. Aparcamiento de vehículos [20]: Entrenar a vehículos inteligentes en un entorno no controlado para que sean capaces de aparcar, evitando accidente y posibles incidentes por culpa del tráfico.
3. Aplicación en Juegos [21]: Es un algoritmo muy útil a la hora de entrenar a inteligencias artificiales para que sean capaces de ofrecer un desafío a los jugadores a los que se enfrentan en juegos de mesa. Sabiendo esto también podemos deducir que sería muy útil para entrenar a otro tipo de inteligencias artificiales dentro de los videojuegos, como pueden ser las de los NPC.

Tras todo esto comprendemos que los algoritmos de PPO destacan dentro de los algoritmos de aprendizaje por refuerzo por su estabilidad durante el aprendizaje, su buen rendimiento en políticas continuas y discretas.

Tras estudiar estos dos algoritmos encontramos varios trabajos en campos similares realizados con ellos. En el caso de PPO encontramos un proyecto en el que trabajaban con un problema muy similar [22]. Por la parte del algoritmo de Qlearning encontramos otro cuya conclusión final era que dicho algoritmo puede utilizarse para entrenar a robots cuadrúpedos [23].

Con toda esta información decidimos que la mejor forma de introducirnos a la hora de solucionar problemas como este era utilizando el algoritmo de Qlearning. Principalmente esta decisión fue tomada porque nos era conocido su funcionamiento además de que, al ser un algoritmo Off-Policy, nos parecía que las características de este podían solucionar de manera

más eficiente el problema. Aunque por encima de todo esto cabe destacar que es un algoritmo sencillo de entender por lo cual vimos que era la mejor manera de entrar a trabajar en aprendizaje por refuerzo.

Capítulo 3

Fundamentos tecnológicos

En este capítulo hablaremos de qué herramientas y programas hemos utilizado a la hora de trabajar con nuestro robot, implementar nuestro código y lanzar nuestros experimentos.

3.1 Máquina virtualBox

Se decidió que este proyecto se llevaría a cabo utilizando una versión específica de Linux, la 22.04. Esto significaba que tenía la necesidad de o hacerle una partición del ordenador o descargar una máquina virtual con este sistema para poder trabajar. Por mis buenas experiencias con VirtualBox nos acabamos decidiendo por la segunda opción y el proyecto se llevó a cabo en una máquina virtual.

3.2 CoppeliaSim

Recordemos que el objetivo de este proyecto es el de enseñar a caminar a un robot con patas. Para ello una parte muy importante es la de obtener un robot o en nuestro caso trabajar con un simulador. CoppeliaSim [24] permite que los robots ahí creados sean controlados desde un código en Python con mucha facilidad, además que permite editar los robots de forma como y añadirles scripts para su correcto comportamiento. Nuestro proyecto lo realizamos trabajando con la versión 4.4.0 para Ubuntu 22.04. Ejemplos de robots con los que trabajaremos son los que se pueden ver en las figuras 3.1 y 3.2

3.3 Python y librerías

Para programar los controladores que utilizariámos para los diferentes robots decidimos trabajar en usando como entorno VScode y como lenguaje Python. Las principales librerías que llegamos a utilizar serán Numpy, para trabajar con matrices, las librerías Re y Time, para

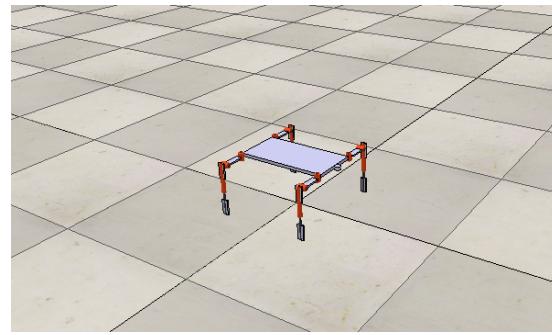


Figura 3.1: Ejemplo robot 4 patas.

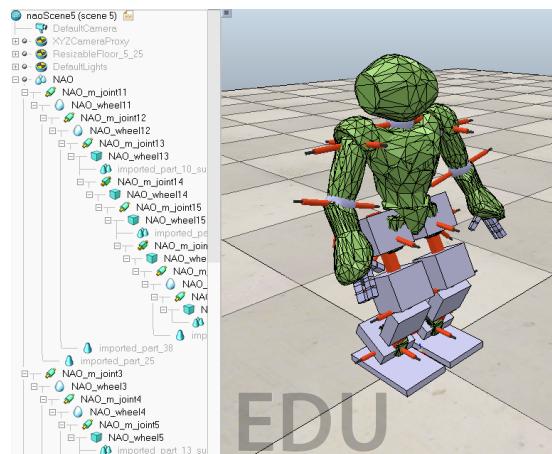


Figura 3.2: Ejemplo de un robot de dos patas en CoppeliaSim mostrando también el menú de joints.

los tiempos de simulación, ZmqRemoteappi sería la que utilizamos para enlazar el código con el robot y así que realizará las acciones que le pedíamos. Math y Random las utilizamos para los cálculos matemáticos y para poder trabajar con elemento aleatorios. Por último, las librerías Pickle y os serían de gran importancia a la hora de almacenar los datos obtenidos durante los aprendizajes.

Librería	Utilidad
<code>time</code>	Permite controlar la duracion del experimento
<code>zmqRemoteApi</code>	Permite controlar la morfología el Coppelia desde python
<code>math</code>	Permite trabajar con ecuaciones matemáticas
<code>random</code>	Permite generar número aleatorios
<code>numpy</code>	Permite trabajar con matrices
<code>pickle</code>	Permite guardar matrices en documentos
<code>os</code>	Permite guardar datos en documentos

Tabla 3.1: Relación de librerías utilizadas

3.4 CESGA

Para entrenar a robots con patas, el algoritmo deber realizar aprendizajes durante una gran cantidad de tiempo y con una potencia computacional superior a la de la mayoría de ordenadores. Sabiendo esto se le pidió permiso a la facultad para crearle una cuenta de usuario en el [Centro de Supercomputación de Galicia \(CESGA\)](#) ?? al alumno para que allí pudiera lanzar los experimentos sin problemas. Llevó un tiempo obtener el permiso y transportar la máquina virtual y los códigos al servidor, además de que es necesario el uso de ciertos scripts para lanzar allí los códigos, pero de esta manera se pudieron lanzar una gran cantidad de experimentos de manera simultánea.

3.5 Singularity

Para poder lanzar nuestros experimentos desde nuestro ordenador dentro del ordenador del CESGA trabajamos con Singularity. Esta aplicación nos permite crear además de ejecutar contenedores ligeros donde vamos a encapsular el sistema operativo que utilizamos, además de las aplicaciones como CoppeliaSim. Una ventaja que tiene Singularity con respecto a otras aplicaciones como Docker es que este se encuentra optimizado para que pueda funcionar en supercomputadoras.

Capítulo 4

Metodología

En este capítulo comentaremos la planificación que llevamos a cabo para este proyecto y como lo dividimos en diferentes fases.

Para llevar a cabo este TFG decidimos organizarnos con una metodología iterativa, por la cual se le iban proponiendo al programador diferentes problemas para que fuera solucionando de manera procedural. Empezando por los más sencillos y pasando a los más complejos que nos dieron la capacidad de afrontar el problema principal. Para organizar estas diferentes fases se estuvieron llevando a cabo reuniones semanales, en un principio de manera presencial, en las que el tutor repasaba los avances realizados y proponía nuevas tareas. Aun así podemos separar el proyecto en diferentes fases diferenciadas por el tipo de trabajo que se realizó en ellas. [4.1](#)

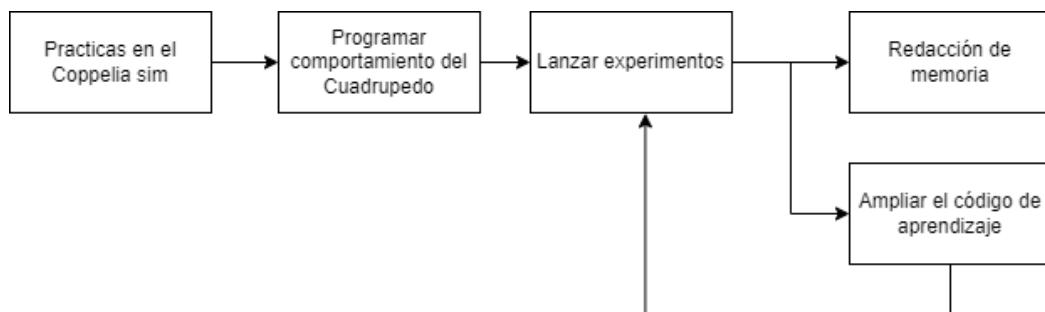


Figura 4.1: Diagrama de la realización del TFG.

4.1 CoppeliaSim

Durante esta primera fase del proyecto el objetivo fue el de acostumbrarse al uso del simulador de CoppeliaSim. Primero montando el programador los propios robots y trabajando con los scripts para acostumbrarse a ellos y después creando unos primeros controladores

básicos para desplazar el robot.

4.2 Programar robot

La siguiente fase y una de las que más tiempo llevo fue la de programar el robot para que cumpliera el objetivo de este proyecto. Para ello trabajamos con un robot de cuatro patas la gran mayoría del tiempo realizando diferentes implementaciones del código. Con cada una de ellas íbamos aprendiendo más y nos íbamos acercando más al objetivo con la idea de conseguir que el cuadrupedo comenzase a desplazarse hacia adelante de manera fluida.

4.3 CESGA

En cuanto tuvimos un primer código lo suficientemente desarrollado como para poder empezar a dar resultados, obtuvimos el acceso al CESGA y subimos todos los archivos a él. Para ello también tuvimos que obtener unos scripts que serían necesarios para la ejecución del código dentro del superordenador. Por suerte obtuvimos unos scripts casi funcionales de manos de nuestro tutor a los que solo les tuvimos que hacer unas pequeñas correcciones para que funcionaran en nuestro código.

4.4 Comprobación de resultados

En cuanto todo estuvo montado, el siguiente paso fue el de lanzar los experimentos en el CESGA con el objetivo de comprobar si nos daban los resultados deseados o, por el contrario, el código no funcionaba como se esperaba. Mientras íbamos recibiendo los resultados fuimos corrigiendo los problemas que iban surgiendo con la idea de optimizar el código.

4.5 Redacción de memoria

Tras realizar todo esto y obtener unos cuantos resultados de los experimentos, pasamos a ir redactando la memoria de todo lo hecho hasta ese momento, este paso se realizó al tiempo que se seguían realizando nuevas implementaciones del código y lanzando nuevos experimentos.

4.6 Almacenamiento de resultados

Se creo un repositorio en Github donde almacenamos la información sobre el trabajo tutelado. La idea es subir a este repositorio los códigos implementados además de los resultados y las gráficas producidas. También se tiene pensado ir subiendo a este repositorio los

vídeos que certifique el correcto desplazamiento del cuadrúpedo. El link al repositorio es el siguiente: [Implementación aprendizaje por refuerzo en robots con patas para aprender a caminar 4.2](#). La dirección de dicho repositorio sería el siguiente: <https://github.com/jorgeriva/TFG-Implementacion-de-aprendizaje-por-refuerzo-en-robots-con-patas-para-aprender-a-caminar>

[TFG-Implementaci-n-de-aprendizaje-por-refuerzo-en-robots-con-patas-para-aprender-a-caminar.](#) / TFG / 

jorgeriva Add files via upload	
Name	Last commit message
...	
Codigo	Add files via upload
Graficas	Add files via upload
Resultados/REsultadosCorrectos	Add files via upload

Figura 4.2: Imagen del repositorio de Github

Capítulo 5

Desarrollo

En este capítulo comentaremos de manera detallada cada una de las fases del desarrollo de nuestro proyecto, comentando las decisiones que realizamos y el porqué de ellas. Seguiremos un esquema similar al del capítulo anterior.

5.1 CoppeliaSim

Tal y como comentamos en el capítulo anterior, la primera fase de nuestro proyecto fue la instalación de CoppeliaSim y la realización de varias tutoriales para acostumbrarnos a trabajar con este programa. Como íbamos a trabajar con [External Controller](#), es decir, que para controlar el robot no usaríamos los scripts propios de Coppelia, sino que trabajaríamos desde Python, también realizamos varias prácticas sobre esto.

El primer tutorial, y el que a mi forma de ver fue más importante, trataba de la construcción de un Bubblebot, es decir, un robot con forma de esfera que se desplazaba usando par de ruedas, como se ve en la figura 5.1 (página 24). Este robot lo introducimos dentro de un laberinto con el objetivo e comprobar como funcionaba su movimiento y tratar de comprender como funcionaban los joints.

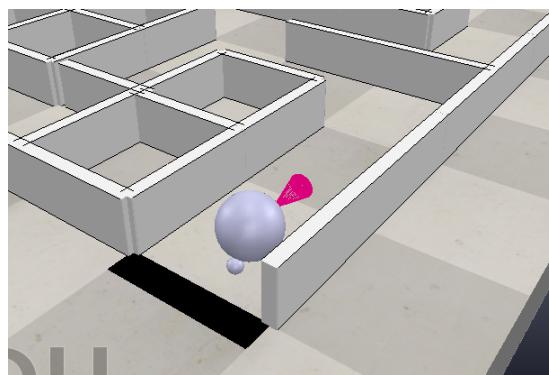


Figura 5.1: Bubblebot dentro de un laberinto para probar el funcionamiento de los scripts

Tras realizar esto decidimos montar otro robot al que denominaremos coche y que se puede observar en la figura 5.2 (página 25). A este robot le añadimos unos sensores laterales y frontales con el objetivo de detectar cuando se acercaba a una pared y si a sus lados tenía o no camino libre. Tras esto introducimos el robot dentro de un laberinto con una única salida para ver si éramos capaces de hacer que saliera de él sin problema como se ve en a figura 5.3 (página 25). Esta vez, en vez de usar los scripts para controlar su movimiento, implementamos un sencillo código en Python que toma las decisiones de sí seguir recto si no detectaba ninguna pared y, cuando la detectaba, dependía de los sensores laterales para saber hacia qué lado salir.

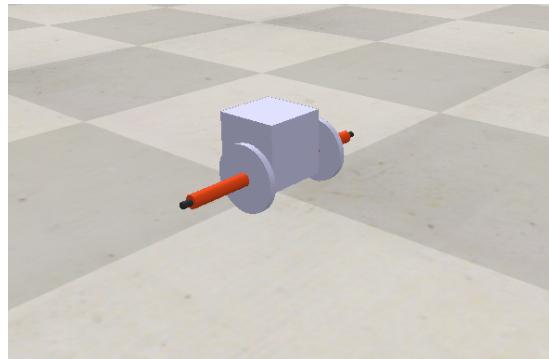


Figura 5.2: Robot coche

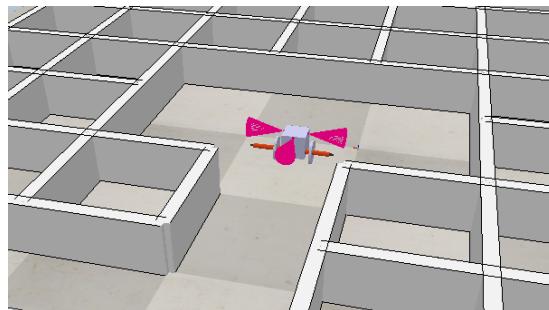


Figura 5.3: Robot coche dentro del laberinto

Estas prácticas en CoppeliaSim se pueden ver como un aprendizaje para obtener los conocimientos básicos antes de que pudiera empezar a programar el código que resolviera el problema propuesto. Lo más importante que se obtuvo de este aprendizaje, además de una familiarización con Coppelia, fue el entendimiento de como funcionaba los joints y un primer acercamiento a controlar desde Python lo que pasaba dentro de ese programa.

5.2 Código de aprendizaje de robot Cuadscene

Para empezar a hablar del desarrollo del código creo que es importante recordar primero cuál es el objetivo de esta práctica. Nuestro objetivo es enseñar a caminar a un robot cuadrúpedo usando aprendizaje por refuerzo. En este caso hemos decidido empezar aplicando Qlearning.

Durante el desarrollo de la práctica hemos ido implementando varias versiones de un mismo código al que nos referiremos como Cuadscene.py. Este código será el encargado de generarnos la matrizQ que, tal y como explicamos en los capítulos anteriores, será la que utilizaremos para definir para saber cuál es la acción correcta a realizar en cada uno de los estados.

Estas versiones van corrigiendo ciertos errores que hemos ido encontrando, desde una mala comprensión del algoritmo de Qlearning hasta una simplificación del problema para buscar obtener mejores resultados.

De todas ellas, la más importante y con las que nos encontramos trabajando ahora es con la cuarta iteración, que es una simplificación de la tercera. Podemos ver la base del funcionamiento de este algoritmo con el siguiente diagrama de flujo [5.4](#) (página 27)

Con este aprendizaje trataremos de obtener una matrizQ cuyos estados sean las diferentes posibles posiciones de las patas, y sus acciones sean los posibles movimientos. Una posible matrizQ sería la siguiente: [5.5](#) (página 29)

5.2.1 Primera iteración: Funciones básicas

La primera aproximación que realizamos para resolver este problema es un código cuyo único objetivo es hacer que el robot realice los movimientos que nosotros deseamos. Con el objetivo de familiarizarnos con las librerías que vamos a utilizar además de practicar el control del robot desde Python. Estos movimientos serían el desplazamiento de las patas hacia adelante y hacia atrás. En este código implementamos las funciones mover adelante y mover atrás que debían mover la pata que nosotros escogíramos cuarenta y cinco grados hacia delante el mover adelante, y volver a la posición original el mover atrás. Esto lo acabaríamos cambiando por una única función, mover aleatorio. Esta función escoge de manera aleatoria que movimientos hacer de una lista de posibilidades que irán variando en cada una de las versiones futuras del código.

En un principio tomamos como decisión de diseño el que solo pudiera realizar esos dos movimientos como manera de acotar el número de posibles acciones. Un aprendizaje totalmente libre permitiría que los joints se movieran en cualquier ángulo, pero esto es deficiente a la hora de implementar el código, ya que se generan demasiadas variables.

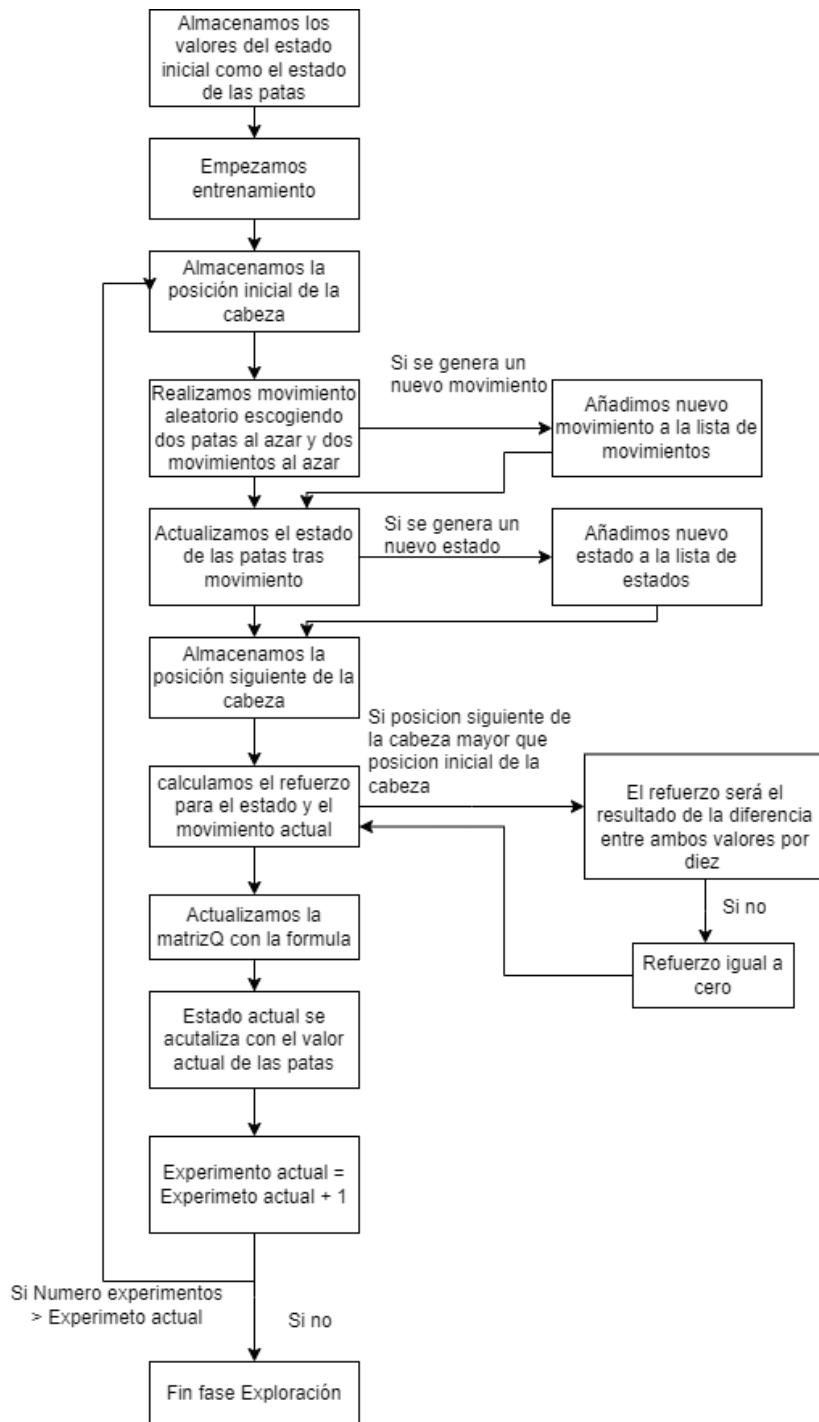


Figura 5.4: Diagrama de flujo que representa el funcionamiento del algoritmo

5.2.2 Segunda iteración: Cuadscene2

Tras esta primera aproximación de aprendizaje generamos un código que aunque entrenaba al robot no cumplía todas las normas del Qlearning, con esto queremos decir que tenía un único estado para cualquiera que fuera la posición de sus patas. A este nos referiremos en un principio como Cuadscene2, este código no siguió las normas del Qlearning al pie de la letra, por eso lo acabamos desechando pero es la base de los siguientes:

La idea que teníamos para esta primera aproximación a un aprendizaje era que el robot realizase dieciséis movimientos aleatorios desde su posición original y después volviera a esta. Haciéndolo así podríamos encontrar cuál era la mejor sucesión de movimientos, en la matriz, definidos como acciones, teniendo un único estado. Decidimos que fueran dieciséis los movimientos, ya que el número de joints era ocho, y así podíamos llegar a permitir que se llegaran a mover todas las patas dos veces.

Empezamos generando las matrices que van a ser necesarias posteriormente para el funcionamiento del código, esta serían una matriz Q de tamaño indefinido, puesto que irá aumentando de tamaño cada vez que añadimos un nuevo conjunto posibles desplazamientos. La matriz desplazamientos que será necesaria para ir almacenando estos distintos desplazamientos como array y la matriz visitas, matriz Q que usamos únicamente en este algoritmo para calcular alfa, necesaria a la hora de aplicar el Qlearning. Aquí se encuentra el problema en relación al algoritmo de Qlearning ya que solo estamos permitiendo que tenga un estado, por esta razón posteriormente implementamos otros códigos.

Tras hacer esto obtenemos el control de los ocho joints del robot y metemos estos joints dentro de un array para poder acceder en ellos utilizando solamente el array al que denominaremos como $arm[]$. De esta manera podemos asociar un número a cada uno de estos joints sabiendo por ejemplo que el Joint 1 es la pierna delantera y el Joint 2 es la rodilla de esa pierna. La relación entre los joints y los valores de array es la que se puede ver en la tabla 5.1 (página 29). También definimos dos variables, aprendizaje e imprimir, que utilizaremos más adelante durante el transcurso de la programación del código. La primera la utilizaremos dándole el valor true hasta que se realice un número de episodios predefinidos. Los cuales son una variable global a la que llamamos Número experimentos, durante los primeros experimentos que lanzamos el valor de esta variable rondaba los quince mil. La segunda la definimos false hasta que el bucle de episodios se complete, tras esto imprimiremos en un nuevo documento los valores obtuvimos, estos valores serán las tres matrices de las que hablamos antes, tras realizar esto se pondrá false y no habrá que volver utilizar nunca esa parte del código.

Ahora cada una de estas iteraciones del bucle principal realizan los siguientes pasos: Lo primero es obtener la posición de la cabeza del robot, tras esto realizamos dieciséis movimientos. Tras realizar esos 16 movimientos los almacenamos ordenados en un array llamado movimientos hechos, ese array lo almacenamos dentro de otro llamado lista de movimientos. Antes

Acciones Estados \ Acciones Estados	[1,2,2,0]	[0,1,2,0]	[0,1,0,2]	[1,1,1,2]
[0,0,0,0]	1.23	1.63	2.9	0.46
[0,2,0,0]	0.7	0.0	0.005	1.54
[1,2,0,0]	0.35	1.29	2.8	0.7
[2,2,0,0]	2.02	1.11	2.65	0.5

Figura 5.5: Ejemplo de matrizQ de nuestro experimento

Arm[]	joint del Cuadrúpedo
arm[0]	arm0
arm[1]	leg0
arm[2]	arm3
arm[3]	leg3
arm[4]	arm1
arm[5]	leg1
arm[6]	arm4
arm[7]	leg4

Tabla 5.1: Relación entre array Arm[] y los joints a partir del Cuadscene5

de hacerlo realizamos la siguiente comprobación: comprobamos que esa lista de movimientos en ese orden no se encuentre ya dentro del array y de así ser lo que hacemos es no añadirla y almacenar para esta iteración en qué posición de la lista de movimientos hechos se encuentra. Tras hacer esto ampliamos la matriz añadiéndole una nueva columna si hemos ampliado el array de la lista de movimientos

Tras hacer los movimientos el siguiente paso es comprobar en qué posición se encuentra la cabeza del robot, de esta manera sabemos si ha avanzado mucho, poco, nada, o ha retrocedido. Recordar que el objetivo del experimento es encontrar cuál es la mejor sucesión de movimientos que hará que el robot vaya avanzando hacia delante.

Nuestro siguiente paso es calcular el refuerzo que va a obtener la matrizQ con esa acción. Esto lo hacemos de la siguiente manera, tenemos una función que recoge el valor de la posición de la cabeza al inicio de la iteración y el valor de la posición de la cabeza tras realizar los movimientos, si la posición tras realizar movimientos es mayor que la de antes de realizar los movimientos el refuerzo se calculará con una simple ecuación de segundo grado, de no ser así el refuerzo será cero. Podemos ver la ecuación en la figura

$$\text{refuerzo} = 10 * (\text{pos_siguiente} - \text{pos_actual}) \quad (5.1)$$

Antes de aplicar el algoritmo de Qlearning necesitamos saber el valor de alfa, es decir, el valor de la tasa de aprendizaje. Como realizamos varios experimentos con este código, el valor de alfa fue variando. El primero fue el de la siguiente fórmula y después trabajamos con una variable global.

$$\text{refuerzo} = 1.0 / (1 + \text{Visitas}[\text{pos_list}][\text{estadoactual}]) \quad (5.2)$$

Tras realizar todos estos pasos intermedios por fin aplicamos el algoritmo de Qlearning para obtener el valor de la matrizQ en esa posición . Aquí es donde mi algoritmo de la matrizQ se diferencia un poco del original. En el algoritmo original tenemos varios estados y varias acciones distintas. El objetivo del algoritmo es ver para cada estado cual es la acción que nos proporciona un mejor resultado. En este primer intento de implementar un código solo trabajábamos con un estado, que es la posición del robot en ese momento, y las acciones serían los distintos conjuntos de dieciséis movimientos. Al contar con un solo estado nos quedaba una matriz con una única fila y tantas columnas como posibles combinaciones de movimientos haya.

$$\begin{aligned}
 Matrizq[textpos_list] &= \\
 (1 - alfa) * Matrizq[textpos_list] + alfa * \\
 (refuerzo + textFACTOR_DESCUENTO * Maximo)
 \end{aligned}$$

Tras esto actualizamos el valor de visitas, y sumamos uno al número de episodios con el objetivo de llegar al número definido, la variable global. Por ahora en la mayoría de mis experimentos he tratado de realizar 10.000 o 15.000 episodios, pero en la mayoría de los casos tras 12 horas no se sobrepasan las 7.500.

Viendo que esto pasaba y no acaba de realizar el experimento, cree cuatro funciones que se iban actualizando en cada iteración, imprimiendo y generando cuatro documentos que almacenaban tanto el array de visitas como el de matrizQ como la lista de movimientos hechos y el mejor movimiento de todos ellos, es decir, el mejor movimiento para el único estado que tenemos en la primera implementación del algoritmo y en el otro caso el mejor movimiento para cada uno de los diferentes estados. También retoque el código original para que en vez de empezar creando unas matrices nuevas pudiera empezar leyendo las antiguas y así seguir trabajando con los datos que ya habíamos obtenido

Si se alcanzara el número de experimentos planificado, alcanzaríamos el punto en el que utilizaríamos la variable Imprimir para imprimir cuatro archivos con un nombre similar a los antes mencionados, pero terminados con la palabra final. Un ejemplo de esto sería que al lanzar el experimento se imprimiría un documento llamado matrizQ y si se llega al final se imprimiría otro llamado matrizQfinal. Teniendo en cuenta que queríamos lanzar varios experimentos al mismo tiempo decidí implementar una función que revisa que no haya ningún documento con este nombre en la carpeta, y de haberlo le añade un número de la misma manera que funcionaba el añadir la palabra final.

Comentaremos los resultados de este código en el capítulo de resultados, pero se puede adelantar que no funcionó de la manera que esperábamos, por lo que realizamos dos implementaciones nuevas, siguiendo con más cuidado la teoría del Qlearning. Estas las denominaremos Cuadscen5.py y Cuadscene7.py entre ellas y la original hubo más implementaciones pero fallidas.

5.2.3 Tercera iteración: Cuadscene5 y Cuadscene7

Estas dos son muy similares entre sí, así que comentaremos los detalles comunes entre ellas primero y después pasaremos a lo que las diferencia.

Cuadscene5

La primera diferencia con respecto al anterior código, y una de las más importantes, es que en estas implementaciones dependeríamos de los diferentes estados en los que se encontraba el robot para diseñar el algoritmo, siguiendo al pie de la letra el algoritmo de Qlearning. Para hacer esto editamos la función que estábamos usando para mover el robot con la idea de que nos devolviera en qué estado se encontraba tras realizar el movimiento. El estado cero sería en la posición cero y el estado 1 sería si la pata del cuadrúpedo había avanzado hasta cuarenta y cinco grados. Además de esto creamos un nuevo array al que denominamos como lista estados en el que almacenábamos el array con los estados de las patas. Para hacer esto seguimos el mismo método que con el array arm[] creando un array estados[] al que le íbamos actualizando los valores cada vez que realizábamos un movimiento. La lista de estados de este código almacena los doscientos cincuenta y seis posibles estados que hay, cada uno compuesto por los valores que tienen las patas en ese punto. Tendrán el valor cero si se encuentra en cero grados y el valor uno si se encuentran en cuarenta y cinco grados. Por su parte, la lista de movimientos contará en este caso con cuatro mil noventa y seis, siendo cada uno de ellos un array de ocho elementos, siendo los impares la pata que se mueve y los pares el estado al que se mueve la pata.

El que hubiera tantos posibles movimientos fue una de las razones por las que desechamos este código e implementamos el Cuadscene7.

Otro cambio que se realizó en este código fue que la matriz pasó a ser de un tamaño predefinido, ya que se podía calcular con facilidad el número de estados y el número de acciones que podría haber en el experimento. También variamos el número de movimientos que haría el robot, pasando de dieciséis a cuatro.

Con todo esto realizamos los cambios necesarios para el correcto funcionamiento del código, incluidos cambios en la función de la matrizQ . Por último editamos los valores que almacenábamos, en este caso solo sería necesario guardar en documentos los valores de la matrizQ, de la lista de estados y de la lista de movimientos. Tras estos cambios nos quedó la siguiente ecuación:

$$\begin{aligned} \text{Matrizq}[estadoreal][textpos_list] = \\ (1 - \alpha) * \text{Matrizq}[estadoreal][textpos_list] + \alpha * \\ (\text{refuerzo} + \text{FACTO}_\text{DESCUENTO} * \text{Maximo} - \\ \text{Matrizq}[estadoreal][textpos_list]) \end{aligned}$$

Tras realizar todo esto y lanzar varios experimentos, vimos que los resultados no eran los esperados ya que el robot no se desplazaba. Se produjo un problema por el cual el algorit-

mo detectaba que el mejor movimiento para que el robot avanzase era uno que mantenía el robot quieto. Para tratar de solucionar este problema buscamos simplificarlo realizando los siguientes cambios.

Cuadscene7

La primera decisión que tomamos fue la de trabajar únicamente con los joints de las piernas, sin usar los de las rodillas, la relación entre los valores del array arm[] y los joints se ve en la siguiente tabla 5.2 (página 33).

Arm[]	joint del Cuadrúpedo
<i>arm[0]</i>	arm0
<i>arm[1]</i>	arm3
<i>arm[2]</i>	arm1
<i>arm[3]</i>	arm4

Tabla 5.2: Relación entre array Arm[] y los joints a partir del Cuadscene7

La Lista de estados de este código almacena los ochenta y un posibles estados que hay. Este número de estados se obtiene de que un estado se encuentra definido por un array que de cuatro elementos y cada uno de esos elementos puede ser un número entre cero y dos. Cada estado esta compuesto por los valores que tienen las patas en ese punto, siendo cuatro patas con tres posibles valores cada una. Tendrán el valor cero si se encuentra en cero grados, el valor uno si se encuentran en cuarenta y cinco grados y el valor dos si se encuentran en menos cuarenta y cinco grados. Por su parte, la Lista de movimientos contará en este caso con ciento cuarenta y cuatro elementos. Este número de estados se obtiene de que un movimiento se encuentra definido por un array que de cuatro elementos, dos de esos elementos pueden tomar valores entre cero y tres y dos de esos valores entre cero y dos, dando un total de ciento cuarenta y cuatro posibilidades. Estos valores definen para los impares la pata qué se mueve y para los pares el estado al que se mueve la pata.

Por la primera aproximación que habíamos hecho, en la que solo buscamos hacer que el cuadrúpedo se desplazara sin usar aprendizaje, sabíamos que el robot era capaz de caminar usando movimientos con ese formato. Además, si conseguíamos que en esta versión en la que ya contabamos con aprendizaje, funcionase, podríamos añadirle a posteriori los otros cuatro joints y mas movimientos para que sea más natural el movimiento.

Con todo esto implementado se necesitaba un código para comprobar que los aprendizajes en el CESGA funcionaban. Este código recibe la lista de movimientos, la lista de estados y la matrizQ. Con estos datos el robot Cuadscene comprueba en que estado está, con esa infor-

mación busca la posición de ese estado en la lista de estados y posteriormente busca el valor de los mejores movimientos para ese estado. Realiza los movimientos mientras actualiza el estado del robot, y así repetidamente. De esta manera deberíamos de conseguir que el robot avanzase hacia delante.

A continuación se muestra un ejemplo del pseudocódigo utilizado en la parte de entrenar la matrizQ.

```

1 #include math
2 #include random
3 #include numpy
4 #include os
5 #include Pickle
6
7 creamos tres variables globales, tasa aprendizaje, factor descuento
    y numero experimentos
8 Definimos funciones auxiliares como obtenerMaximo, getRefuerzo o
    Movimiento
9
10 int main() {
11
12     Inicializamos Q a cero, creando una matriz de 81 filas y 144
        columnas
13     Creamos un array vacio para almacenar los estados
14     Creamos un array vacio para almacenar los movimientos
15     Creamos un array con los joints
16     Creamos un array con el estado actual
17     Añadimos el estado actual a la lista de estados
18     Creamos una variable estado actual a 0
19
20     mientras la variable entrenar este a True:
21         Para varible inicial hasta número de experimentos
22             Tomamos la posición de la cabeza del robot
23             Relizamos los movimientos
24             Actualizamos el estado del robot
25             Si el movimiento no esta en la lista de Movimientos:
26                 Añadimos los movimientos a la lista de movimientos
27             Si el estado no esta en la lista de estados:
28                 Añadimos el estado a la lista del estados
29                 actualizamos el valor de estado siguiente con la
                    posición del estado en la lista estados
30                 Obtenemos la posición de la cabeza del robot
31                 Obtenemos el valor del refuerzo
32                 Obtenemos el valor del máximo posible futuro refuerzo
33                 Actualizamos el valor de la matrizQ
34                 Actualizamos la posición, estado actual=estado siguiente

```

```

35
36      Almacenamos en tres documentos los valores de la
37      matrizQ de la Lista estados y de Lista movimientos
38      Actualizamos el valor de entrenar a False
39
40      #Tras esto vendría la parte del código cuyo objetivo es
        comprobar el funcionamiento del robot tras el aprendizaje pero
        tambien tenemos un código propio que se encarga de hacer eso.

```

Tras lanzar varios experimentos seguía ocurriendonos el mismo problema, no eramos capaces de que el robot se desplazara y seguía realizando movimientos que lo mantenían quieto. Se nos ocurrió que un posible culpable de este problema era que no estabamos implementando un algoritmo que definiera los límites entre la exploración y la explotación, así que implementamos un nuevo código que si lo tuviera,

5.2.4 Cuarta iteración: Cuadscene8

En esta implementación del código realizamos un par de variaciones al código Cuadscene7.

Hasta ahora nuestro código tenía predefinido un número de episodios a realizar. Mientras durase el bucle definido por ese número de episodios nos encontrábamos en la etapa de exploración y al acabar nos queda una matriz entrenada para la explotación. Esto no es del todo eficiente, por lo que decidimos implementar el algoritmo E-greedy de la siguiente forma.

Declaramos el hiperparámetro E con valor 1 para controlar la probabilidad de realizar acciones aleatorias. Para cada iteración del bucle de control usamos una función random para obtener un valor entre cero y uno, si el número es mayor que E la acción a realizar será la que tiene un mayor valor dentro de la matriz para ese estado en particular. De no ser así se realizará una acción aleatoria.

Al final de cada iteración del bucle vamos reduciendo el valor de E con el objetivo de ir disminuyendo poco a poco la posibilidad de que realice una acción aleatoria y que, por el contrario, vaya aumentando la posibilidad de que la acción a realizar sea la que tiene un mayor valor en la matrizQ para ese estado.

La última incorporación realizada a este código es dentro del calculo del refuerzo en el que añadimos un diferencial en el calculo del refuerzo que funciona de la siguiente manera 5.6 (página 36):

Como se ve en ella hemos añadido un diferencial de 0.008 para comprobar el avance de la figura. Ahora la posición de la cabeza después del movimiento no solo debe ser mayor que la posición antes, sino que estamos eliminando los errores que surgían antes. Con errores me refiero a los casos en los que se detectaba un movimiento cuando realmente esa acción no

```

def get_refuerzo(pos_actual,pos_siguiente):
    print(pos_siguiente, "pos siguiente")
    print(pos_actual,"pos actual")
    if((pos_actual+0.008)>= pos_siguiente):
        return 0
    if ((pos_actual+0.008)<pos_siguiente):

        refuerzo= 10*(pos_siguiente - pos_actual)
        print("Este es mi refuerzo ", refuerzo)
        return refuerzo

```

Figura 5.6: Función que muestra el refuerzo.

producía ninguno. Esto se comenta con más detenimiento en el capítulo de resultados.

5.2.5 Quinta iteración: Cuadscene11

Con la anterior iteración conseguimos por primera vez que la morfología realizase algún tipo de movimiento, pero no uno que pudiéramos decir que era correcto, viendo esto decidimos implementar el siguiente código en el que, basándonos en el anterior, buscamos subsanar los problemas que nos daba.

Este código funciona de manera que cada una de las acciones que conforman la matriz Q está definida por dos movimientos. Esto se puede ver en el ejemplo de matriz Q que tenemos justo debajo, donde en la parte de acciones vemos que cada recuadro tiene cuatro valores, el primero y el tercero nos indican cuales son las patas que estamos moviendo y el segundo y el cuarto valor nos indican hacia qué posición se mueven, cero significa hacia la posición original, uno cuarenta y cinco grados hacia delante y dos cuarenta y cinco hacia atrás . [5.7](#) (página 37):

La otra parte de la matrizQ es la que define los estados de ella. Esta estará definida por una lista de arrays de cuatro valores que sirven para definir en qué estado se encuentran las patas. Cero significa en la posición original, uno cuarenta y cinco grados hacia delante y dos cuarenta y cinco hacia atrás. Esto nos da un total de ochenta y un posibles estados y ciento cuarenta y cuatro posibles movimientos. Este número de posibles movimientos se obtiene de que un movimiento se encuentra definido por un array que de cuatro elementos, dos de esos elementos pueden tomar valores entre cero y tres y dos de esos valores entre cero y dos, dando un total de ciento cuarenta y cuatro posibilidades. El numero de estados se calcula de la misma manera, ya que cada estado es un array de tamaño cuatro, y cada uno de sus elementos puede tener tres valores distintos.

Con el objetivo de buscar que el cuadrúpedo se desplazará definimos la tasa de aprendizaje con un valor de 0.75 y el factor de descuento a 0.9 . Escogimos un valor tan alto para el factor de descuento porque cuanto este se acerca más a uno mayor será la importancia de los futuros movimientos a la hora de premiar una decisión del cuadrúpedo.

Acciones Estados	[1,2,2,0]	[0,1,2,0]	[0,1,0,2]	[1,1,1,2]
[0,0,0,0]	1.23	1.63	2.9	0.46
[0,2,0,0]	0.7	0.0	0.005	1.54
[1,2,0,0]	0.35	1.29	2.8	0.7
[2,2,0,0]	2.02	1.11	2.65	0.5

Figura 5.7: Ejemplo tabla de resultados matrizQ.

Este código nos brindo resultados en los que el cuadrúpedo se desplazaba hacia delante de manera más o menos correcta. Esos resultados se pueden apreciar dentro del apartado de resultados. Como se obtuvieron estos resultados el código lo hemos decidido lanzar con tres morfologías distintas. La primera es la original, con la que llevamos trabajando durante el resto de versiones. Las otras dos morfologías tienen las patas en angulos de treinta y cuarenta y cinco grados.

5.2.6 Sexta iteración: Cuadscene14

Después de obtener estos resultados decidimos continuar con la posibilidad de que las piernas del robot crecieran, haciendo que durante el transcurso de los episodios cada vez que fueran aumentando su tamaño hasta llegar al punto en el que se encontraba en los experimentos anteriores, es decir, partíamos de la posición 0 a la posición 0.075. Esto los lanzamos con las tres morfologías de las que hemos hablado antes. Sospechamos que al aprender mientras va creciendo el problema que ocurría con la morfología de treinta grados para el código anterior podría subsanarse. Nuestro objetivo era comprobar si el aprendizaje funcionaba de una manera similar o distinta si el robot crecía a cuando ya se encontraba en su posición final desde el principio. Esto nos produjo resultados positivos que se muestran en el capítulo de resultados.

5.3 Aprendizaje en el CESGA

Tras obtener el primer código que creíamos que podría devolvernos resultados favorables empezamos a lanzar los experimentos dentro del CESGA, pero ello subimos una copia de la máquina virtual que estábamos utilizando al servidor usando singularity. Además, almacenamos una copia de los robots con los que íbamos a utilizar para poder lanzar así los experimentos.

Dentro de nuestro repositorio, en el directorio Home creamos dos directorios, el primero EscenasTfg, donde almacenamos todo el trabajo y los scripts, y después el directorio Mover que utilizamos para transportar código desde mi ordenador hacia el servidor y viceversa.

Dentro del directorio EscenasTfg estaban nuestros códigos guardados, los robots y los resultados, además del acceso a la carpeta de scripts donde tendremos almacenados los archivos necesarios para lanzar los aprendizajes 5.8.

```
[ulccojrb@login210-18 Escenas_TFG]$ ls
Lista_estados16.npy    cuadScene5.py.save    resultados
Lista_movimientos16.npy cuadScene6.py        resultados10
Mover                  cuadScene7.py        resultados11
cuadScene.py            matrizQ16.npy      resultados15
cuadScene.ttt           matrizQfinal.pickle resultados5
cuadScene2.py           matrizQfinal2.pickle resultados7
cuadScene3.py           matrizQfinal3.pickle resultadosactualizables
cuadScene4.py           matrizQfinal4.pickle scripts
cuadScene5.py           naoScene5.ttt       zmqRemoteApi
```

Figura 5.8: Aspecto del repositorio donde almacenamos el código dentro del servidor.

Dentro de la carpeta contamos con tres scripts 5.9, el primero el runbasescript se encarga de lanzar, los otros dos, que se encargan de lanzar el código y activar la aplicación de Coppelia para que generar un simulador utilizando la morfología ya creada. Además, por cada experimento lanzado se crea un slurm, un documento de texto que nos muestra los errores que han hecho que el código se detenga, o si ha funcionado lo que se va mostrando, en el caso de este experimento utilizamos los slurms para realizar ciertas comprobaciones sobre el código que habíamos subido.

Para lanzar uno de estos experimentos hay que usar el comando que se muestra en la siguiente figura 5.10 (página 39). Además, también se puede usar el comando squeue para ver el estado del experimento y el scancel para cancelarlo si llegara a ser necesario.

Cuando obteníamos resultados que podían ser favorables, el código era devuelto a mi ordenador, donde utilizábamos los documentos que almacenaban la matriz, la lista de estados y la lista de movimientos para comprobar si estos nuevos parámetros obtenidos conseguían que el robot fuera capaz de desplazarse.

```
[ulccojrb@login210-18 scripts]$ ls
python_script.sh    slurm-3406271.out  slurm-3538180.out  slurm-3592932.out
run_baseScript.sh   slurm-3406276.out  slurm-3538249.out  slurm-3593009.out
run_coppelia.sh    slurm-3406290.out  slurm-3538290.out  slurm-3593147.out
slurm-3404252.out  slurm-3407563.out  slurm-3549491.out  slurm-3593229.out
slurm-3404307.out  slurm-3407567.out  slurm-3560634.out  slurm-3593247.out
slurm-3404658.out  slurm-3407617.out  slurm-3566657.out  slurm-3593282.out
slurm-3404742.out  slurm-3407747.out  slurm-3560701.out  slurm-3610151.out
slurm-3404768.out  slurm-3409045.out  slurm-3567719.out  slurm-3610261.out
slurm-3404770.out  slurm-3436042.out  slurm-3585254.out  slurm-3610320.out
slurm-3404771.out  slurm-3436361.out  slurm-3592395.out  slurm-3610354.out
slurm-3404841.out  slurm-3436419.out  slurm-3592445.out  slurm-3610404.out
slurm-3404875.out  slurm-3436486.out  slurm-3592466.out  slurm-3610655.out
slurm-3404953.out  slurm-3436870.out  slurm-3592519.out  slurm-3618592.out
slurm-3406246.out  slurm-3512356.out  slurm-3592666.out  slurm-3618614.out
slurm-3406254.out  slurm-3531533.out  slurm-3592777.out  slurm-3618651.out
```

Figura 5.9: Aspecto de la carpeta de scripts.

```
[ulccojrb@login210-18 scripts]$ sbatch run_baseScript.sh 23000
sbatch: slurm_job_submit: setting QoS to Partition QoS value: clk_medium
Submitted batch job 3789444
[ulccojrb@login210-18 scripts]$ squeue
      JOBID      PARTITION      NAME      USER ST      TIME  NODES NODELIST(REASON)
      3789444        medium  run_base  ulccojrb PD      0:00      1 (Priority)
[ulccojrb@login210-18 scripts]$ █
```

Figura 5.10: Comandos usados para lanzar un experimento.

Capítulo 6

Resultados

En este capítulo comentaremos los resultados obtenidos con el aprendizaje. Comentaremos los datos obtenidos por los códigos a partir del Cuadscene5, ya que fue el primero en el que trabajamos con una buena implementación del algoritmo de Qlearning. Iremos comentando los problemas que nos fueron dando, los códigos y los resultados.

6.1 Cuadscene5

Recordamos que este código fue el primero en el que utilizamos la teoría de la matrizQ de manera correcta. También es un dato en su funcionamiento entender que trabajábamos con ocho joints, los equivalentes a piernas y rodillas. Aun así, no conseguimos que el robot se desplazara de manera correcta, ya que tras realizar un par de movimientos hacia delante se quedaba instalado en una situación anómala. Por alguna razón se situaba en un estado cuyo mejor movimiento no movió ninguna de las patas, tal y como se puede ver en la siguiente imagen .

```
este es el brazo que se mueve 2
esto es lo que se mueve 45
Este es el estado en el que estamos [0. 0. 1. 2.]
este es el numero del estado 1
este es el valor en la matriz 0.0
este es el numero del movimiento 0
Esto es el movimiento [3. 2. 2. 1.]
```

Figura 6.1: Ejemplo de resultado, se ven los valores que utiliza el código.

Según esto, al mover la pierna tres a la posición dos y la pierna dos a la una es el movimiento que genera un mayor desplazamiento hacia delante. Esto obviamente es incorrecto porque las piernas ya se encuentran en esa posición, por lo que no se genera ningún movimiento. Tras realizar una serie de fallidos decidimos pasar a un nuevo código más simple

desde el que lanzaríamos varios experimentos en los que habría cambios buscando así acotar el posible error.

Los experimentos aquí realizados nos sirvieron para corregir ciertos errores de programación que no habíamos encontrado antes.

6.2 Cuadscene7

Recordamos que este código se diferencia del anterior en que únicamente trabajaremos con los joints de las piernas. Hemos usado este código para ver si el problema se encontraba dentro de los valores que le dábamos a las variables tasa de aprendizaje y factor de descuento, o si tenían que ver con la fórmula que usábamos para definir la matrizQ.

Como con estos experimentos comenzamos a esperar un resultado positivo, empezamos a numerarlos y almacenar la información que los diferenciaba de los otros. Aun así, ninguno de estos experimentos fue capaz de darnos una pista con la que saber como solucionar que el cuadrúpedo no se moviera. Tal y como se muestra en la lista de experimentos, fuimos desechando diferentes posibilidades, desde un error en algunas de las fórmulas que estábamos utilizando hasta la utilización de tasas de aprendizaje o de penalización incorrectas, pasando por el número de experimentos a realizar.

- Experimento 1: Usamos la ecuación de matrizQ original, cambios en el refuerzo, pasamos de una ecuación en la que elevábamos al cuadrado los valores de posición siguiente y posición actual por una en la que únicamente los restábamos
- Experimento 2: Usamos la ecuación de matrizQ original, cambios en el refuerzo. Vimos que esto generaba refuerzos ínfimos, por lo que decidimos multiplicar el resultado por diez.
- Experimento 3: Usamos la ecuación de matrizQ original, 1 millón de episodios
- Experimento 4: Usamos la ecuación de matrizQ original, 10 millones de episodios.
- Experimento 5: Usamos la ecuación de matrizQ original, 10 millones de episodios y cambiamos un valor del cálculo del refuerzo, pasado a multiplicar el valor de refuerzo que se obtenía antes por diez para aumentar su tamaño.
- Experimento 6: Usamos la ecuación de matrizQ original, pero en vez de usar el alfa calculado usamos la tasa de aprendizaje, 1 millón de episodios.
- Experimento 7: Usamos la ecuación de matrizQ original, pero en vez de usar el alfa calculado usamos la tasa de aprendizaje, 10 millones de episodios.

Durante los siguientes experimentos estuvimos realizando pequeños cambios en el refuerzo con la idea de encontrar una ecuación que nos diera buenos valores.

- Experimento 13: Usamos el algoritmo con ciertos cambios sobre el original. Con un refuerzo que ya da buenos resultados. Este algoritmo no es más que únicamente sumarle a la matriz Q el valor del refuerzo en esa posición, olvidándonos de la tasa de aprendizaje y de penalización. Hacíamos esto con la idea de comprobar si esto solucionaba algo y el problema se encontraba únicamente en el refuerzo.
- Experimento 14: Usamos el algoritmo original con el refuerzo anterior.
- Experimento 15: Usamos el algoritmo creado por mí con el refuerzo anterior y hacemos que tras cada movimiento hagamos un pequeño parón, añadiendo dos `client.step()` entre movimientos, para que así podamos tomar mejor los resultados. Hacemos esto porque sospechábamos que tomaba el valor de la cabeza después de hacer el movimiento antes de que acabase de hacerse el propio movimiento.
- Experimento 16: Usamos el algoritmo original con un buen refuerzo y hacemos que tras cada movimiento hagamos un pequeño parón para que así podamos tomar mejor los resultados.
- Experimento 23: Usamos el algoritmo que sabemos que es correcto con una tasa de aprendizaje de 0.9 y un diferencial de 0.1. Nos referimos a un diferencial como una variable que le sumamos a la posición actual de la cabeza y que debe de haber superado la posición siguiente para obtener un refuerzo positivo.
- Experimento 24: usamos el algoritmo que sabemos que es correcto con una tasa de aprendizaje de 0.9 y un diferencial de 0.08.

Al finalizar los experimentos no fuimos capaces de detectar el origen del error, así que decidimos pasar a un nuevo código en el que incluimos el algoritmo de greedy y varios cambios más.

6.3 Cuadscene8

El código de esta implementación se diferencia de las anteriores porque le añadimos un algoritmo de greedy para definir el paso de exploración a explotación. Hicimos varios experimentos a los que denominamos experimentosE en los que íbamos corrigiendo ciertos errores que nos surgieron. Entre ellos cabe destacar que ciertas funciones que utiliza la librería NumPy para generar números aleatorios entre cero y uno no funcionaban en el CESGA. Por lo

que tuvimos que usar la función `numpy.random.random()` para conseguir generar un numero aleatorio entre cero y uno.

El código generaba los siguientes valores:

```
.....  
este es el estado actual [0, 1, 2, 1]  
0.1293321018523701 esta es la y  
0.9999420000000016 esta es la epsilon  
usamos aleatorio  
[1. 2.] comp  
[1. 2. 3. 0.] comp  
estos son los movimientos que hay 29  
0.19657474756240845 pos siguiente  
0.19260887801647186 pos actual  
este es el estado siguiente [0, 2, 2, 0]  
este es el numero del estado 19  
este es el movimiento [1. 2. 3. 0.]  
este es el numero del movimiento 28  
Este es el valor de la matriz q 0.0  
Esto es el experimento 29
```

Figura 6.2: Ejemplo de valores que nos devuelve el código en cada iteración.

Siendo la `y` que se ve en la imagen el número aleatorio que debe ser mayor que épsilon en esa iteración para que en vez de usar un movimiento aleatorio usemos el mejor movimiento posible para ese estado.

Tras corregir ciertos errores volvimos al mismo problema que los códigos anteriores que esperamos que en los siguientes experimentos se solucionen.

- Experimento 21: Usamos el algoritmo que sabemos que es correcto con una tasa de aprendizaje de 0.9 y un diferencial de 1. Con el algoritmo que es correcto nos referimos a la fórmula de actualizar la matrizQ que mostramos en una imagen al principio de la memoria. Este experimento nos devuelve una matrizQ vacía ya que no hay movimientos del tamaño de ese diferencial.
- Experimento 22: Usamos el algoritmo que sabemos que es correcto con una tasa de aprendizaje de 0.9 y un diferencial de 0.8. Este experimento nos devuelve una matrizQ vacía ya que no hay movimientos del tamaño de ese diferencial.
- Experimento 25: Usamos el algoritmo que sabemos que es correcto con una tasa de aprendizaje de 0.9 y un diferencial de 0.008. Tras haber lanzado dos experimentos con diferencial, vimos que este tenía un valor demasiado alto para funcionar. Decidimos bajar el diferencial hasta 0.008 y obtuvimos que el robot comenzaba a desplazarse hacia delante, pero con la particularidad de que se giraba hacia la izquierda. Esto podía ser por dos razones, la primera es que también hiciera falta acotar con otro diferencial el desplazamiento lateral que puede llegar a realizar el robot. La segunda es que parte de los problemas del desplazamiento ocurren por culpa de que el ordenador en el que se está

realizando la comprobación produce que el CoppeliSim vaya a trompicones. Sospechamos de esta posibilidad dado que la segunda vez que comprobamos el desplazamiento este fue mucho más fluido y el robot se desplazó hacia delante, hasta que volvió a ir a trompicones y empezó a girar a la izquierda.

Tras volver a lanzar este experimento conseguimos que avanzase en linea recta como se ve en la figura 6.3 (página 44) pero con la extraña particularidad de que se desplaza en dirección contraria a lo que habíamos pensado, manteniendo ese primer giro hacia la izquierda que ya ocurría antes.

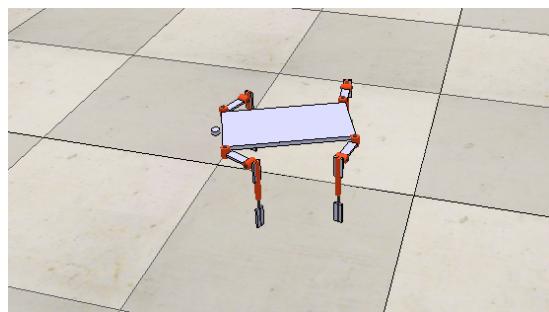


Figura 6.3: Imagen robot desplazándose con el experimento 25.

Tras este experimento decidimos lanzar dos mas con diferenciales similares para ver que resultados nos daban, estos sería 0.006 y 0.01.

Con el experimento 27, cuyo diferencial era 0.006 volvimos a caer en el problema que teníamos al principio, es decir el robot detecta movimiento en acciones en las que no lo hay por lo que cuando pasamos a la fase de explotación este no se desplaza hacia ningún lado.

El de 0.001 al que denominamos como experimento número 28 nos generó otro problema. En esta ocasión el diferencial era tan alto que nos generaba una matriz Q con todos los refuerzo nulos ya que la función de recompensa siempre nos devolverá cero.

6.4 Cuadscene11, código sin crecimiento

Con este código conseguimos que el cuadrúpedo con las patas en posición inicial se desplazara de manera continua hacia delante 6.4 (página 45). Además de poder ver dentro del Coppelia como se desplaza el cuadrúpedo contamos con dos gráficas que nos muestran los resultados obtenidos. La primera 6.5 (página 45) nos muestra los valores de los refuerzos tomados dentro del entrenamiento, aquí se muestran dos casos. El primero es del código haciendo una prueba sobre cuatro mil quinientos episodios. Al ser pocos casos podemos observar que los valores van en aumento hasta llegar a los cuatro mil episodios, al llegar ahí el valor

de épsilon es cero por lo que la fase de aprendizaje ya ha terminado y solo tomara las mejores decisiones por eso se estabiliza en unos valores.

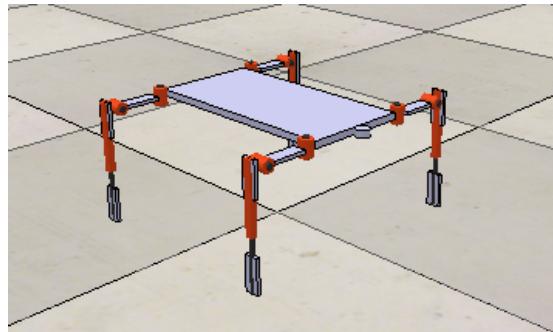


Figura 6.4: Imagen robot con las patas en posición inicial.

Estos valores no son los máximos, teorizamos que esto ocurre porque esos valores máximos del refuerzo ocurrían en casos extremos a los que costaría demasiado llegar para que realizarlos fuera eficiente. 6.5 (página 45).

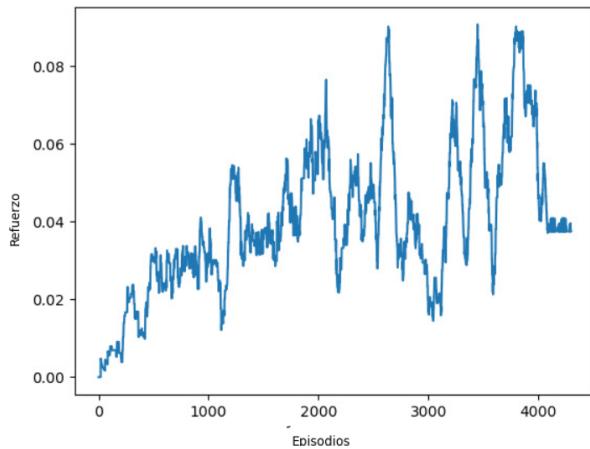


Figura 6.5: Gráfica refuerzo hasta 4500 episodios.

Esta otra gráfica 6.6 (página 46) muestra los valores de refuerzo de la matriz Q tras la fase de aprendizaje, es decir, los valores que tiene cuando se desplaza hacia delante partiendo desde la posición inicial. Cómo ve los primeros movimientos tienen un valor bajo y lo que busca es llegar a un estado desde el cual poder hacer una sucesión cíclica de movimientos que haga que el cuadrúpedo se desplace. En este caso se desplaza usando solo dos patas.

Tras hacer esto decidimos continuar utilizando esta implementación pero con diferentes morfologías, es decir usamos el código que daba estos resultados pero con un robot en el cual su morfología se encontraba en ángulos de treinta grados y en ángulos de cuarenta y cinco

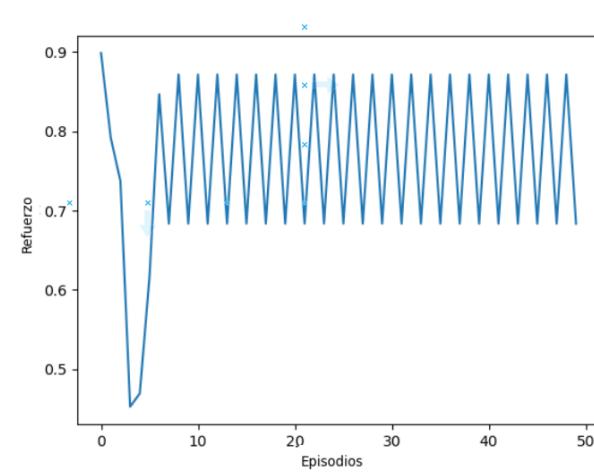


Figura 6.6: Gráfica del refuerzo en cada episodio tras el aprendizaje.

grados para comprobar funcionaba de manera correcta los resultados se muestran aquí.

6.4.1 Ángulo cuarenta y cinco grados

Con el cuadrúpedo con las patas con cuarenta y cinco grados de inclinación [6.7](#) (página [46](#)), los resultados fueron los siguientes: Hemos lanzado un pull de experimento de los que tres de ellos hacen el esfuerzo de desplazarse hacia delante pero sin realizar un buen movimiento. Además de esos hay un cuarto caso en el que el cuadrúpedo utiliza varias de sus patas para desplazarse de una manera que podemos entender como correcta hacia delante. De ese experimento sacamos los siguientes resultados:

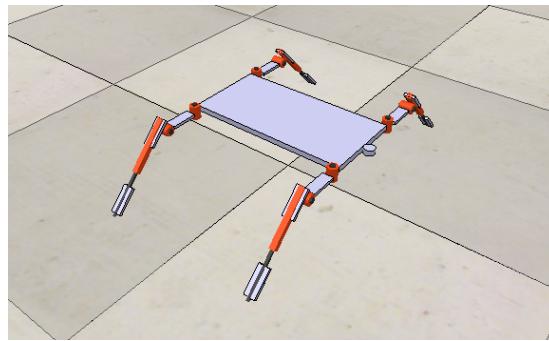


Figura 6.7: Imagen robot con las patas con cuarenta y cinco grados de inclinación.

La primera gráfica nos muestra el refuerzo que hay en la matriz Q en cada uno de los episodios que componen el aprendizaje del cuadrúpedo. Empieza con valores muy bajos y va aumentando hasta el punto que va a acabar estabilizándose. [6.8](#) (página [47](#)).

Esta otra gráfica muestra los valores de refuerzo de la matriz Q tras la fase de aprendizaje,

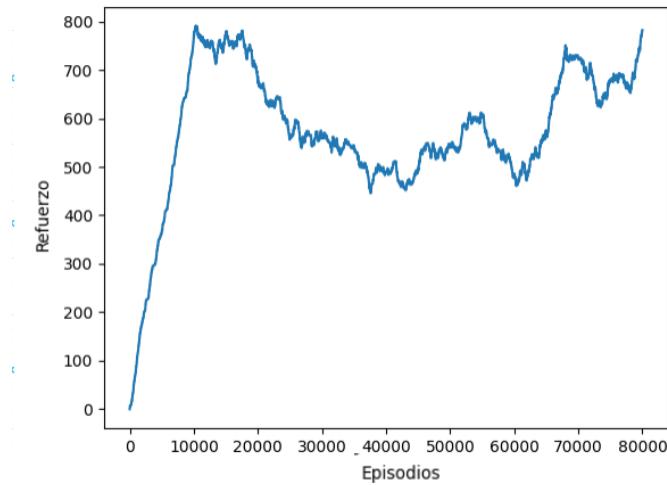


Figura 6.8: Gráfica del refuerzo en cada episodio durante el aprendizaje en cuarenta y cinco grados.

es decir, los valores que tiene cuando se desplaza hacia delante partiendo desde la posición inicial. Cómo ve los primeros movimientos tienen un valor bajo y lo que busca es llegar a un estado desde el cual poder hacer una sucesión cíclica de movimientos que haga que el cuadrúpedo se desplace. En este caso se desplaza usando las cuatro patas por lo que es más eficiente que el caso que comentamos primero. 6.9 (página 48).

Intentamos hacer lo mismo con una morfología en las que las patas se encontrasen colocadas en ángulos de treinta grados. En la mayoría de casos el robot comenzaba a realizar un desplazamiento correcto para acabar estancando y no siendo capaz de avanzar más por culpa de que sus patas chocaban contra el suelo y no eran capaces de realizar el movimiento que se le pedía. 6.10 (página 48).

6.5 Cuadscene14, código con crecimiento

Utilizamos el mismo código que en el Cuadscene11 pero aquí añadimos una variable que hacia que las patas del robot pasaran desde la posición cero hasta la posición 0.075, que es la posición en la que se encontraban en el código anterior. Lanzamos este código con tres variaciones, en ángulo inicial, con una desviación de treinta grados y con una desviación de cuarenta y cinco grados.

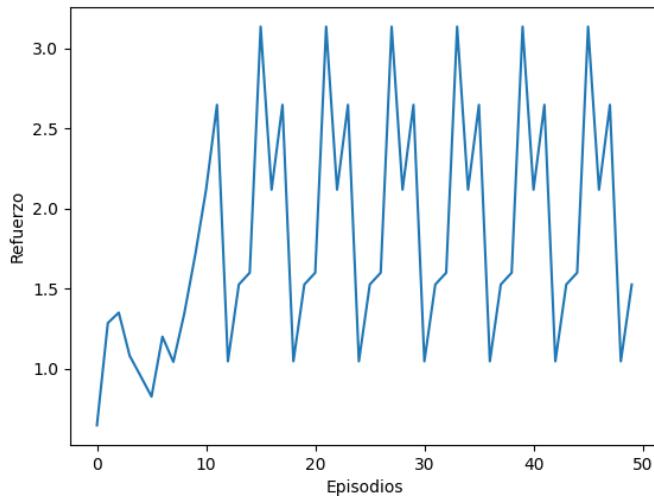


Figura 6.9: Gráfica del refuerzo en cada episodio tras el aprendizaje para cuadrúpedo en ángulo cuarenta y cinco grados.

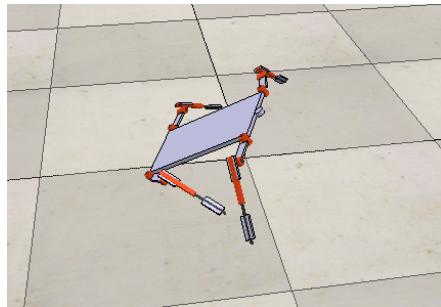


Figura 6.10: Imagen de un robot que se ha quedado atascado, uno de los posibles errores que ocurren por que el cuadrúpedo se cae hacia un lado, en este caso hacia atrás

6.5.1 Estado inicial

En relación al experimento en estado inicial con la variable de crecimiento obtuvimos varias soluciones en las que el cuadrúpedo se desplazaba, cabe destacar dos de ellas. La primera funcionaba de manera similar a la que produjo la gráfica de mover en estado inicial sin crecimiento. Es decir moviendo únicamente la pata delantera derecha y la pata trasera izquierda. [6.11](#) (página 49).

La otra utilizaba tres de sus patas para moverse hacia delante, avanzaba algo menos en cada grupo de movimientos pero su desplazamiento era mucho más continuo que el de el aprendizaje anterior: [6.12](#) (página 49).

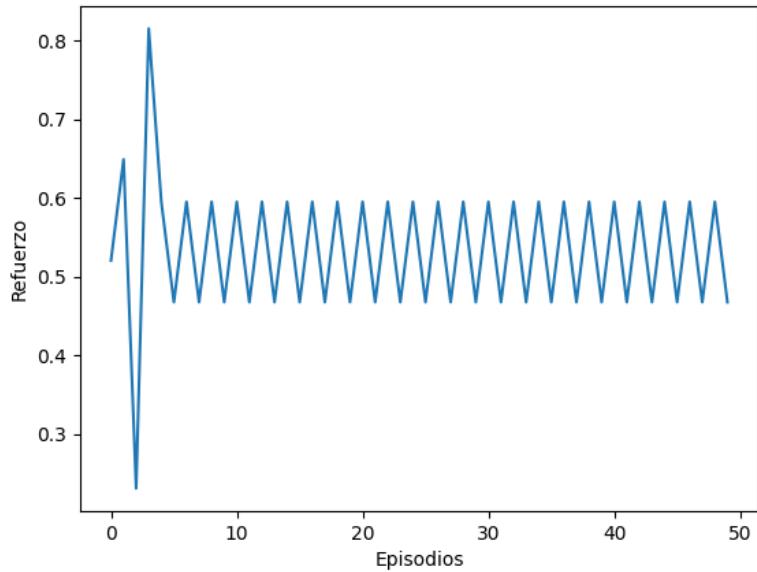


Figura 6.11: Gráfica del refuerzo en cada episodio tras el aprendizaje para cuadrúpedo con variable crecimiento en ángulo inicial 1.

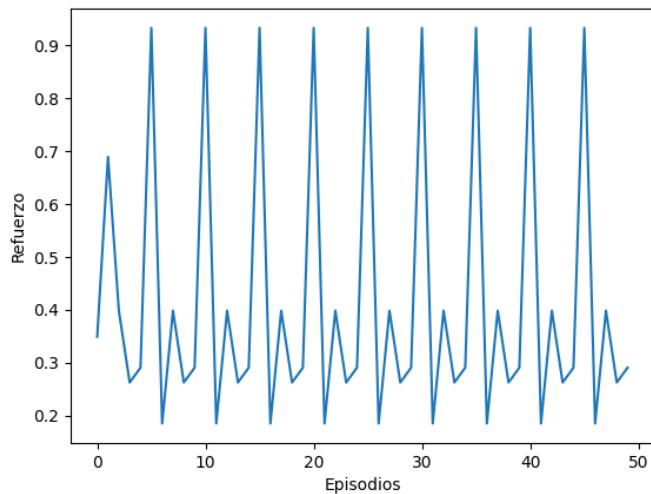


Figura 6.12: Gráfica del refuerzo en cada episodio tras el aprendizaje para cuadrúpedo con variable crecimiento en ángulo inicial 2.

6.5.2 Piernas en treinta grados

En el caso de que la morfología se encuentre con las patas en treinta grados de inclinación 6.13 (página 50), obtuvimos por primera vez resultados positivos entre los que destacamos el primer experimento, donde conseguimos que el cuadrúpedo se desplazara usando toda sus

patas y de manera bastante fluida aunque en ciertos casos se generaban parones entre movimientos. [6.14](#) (página 50).

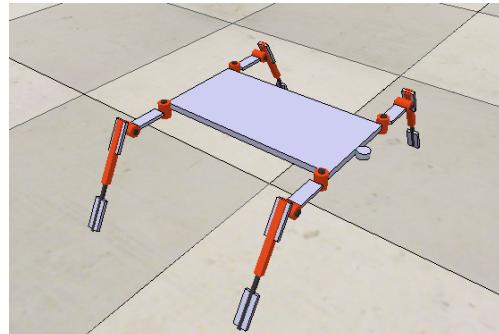


Figura 6.13: Imagen del robot con las patas con treinta grados de inclinación.

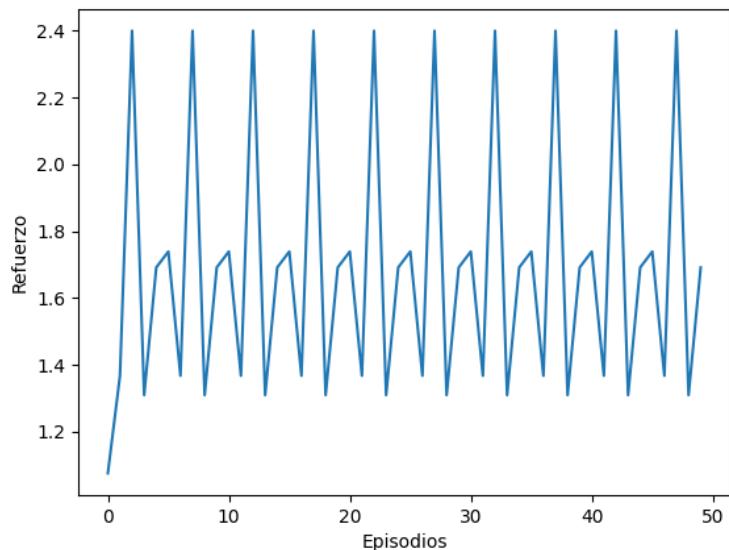


Figura 6.14: Gráfica del refuerzo en cada episodio tras el aprendizaje para cuadrúpedo con variable crecimiento en ángulo de treinta grados.

Este ha sido hasta ahora el experimento con un mejor desplazamiento junto con el de 45 grados sin crecimiento. Esto lo deducimos ya que son los que usan más de los miembros del cuadrúpedo, además de que son los más fluidos en movimiento y que menos tiempo pasan bloqueados entre movimientos.

En la morfología en la que el cuadrúpedo encuentra sus patas en 45 grados basándonos en el código que crece lanzamos en dos tiradas un pull de diez lanzamientos. De los cuales la gran mayoría de ellos empezaban a avanzar hacia delante hasta atascarse y quedarse estancados, sospechamos que esto ocurre por que durante la etapa de crecimiento en la que las patas se

encuentran recogidas se beneficia a ciertos movimientos dándoles un valor de refuerzo muy alto, estos movimientos posteriormente cuando el cuadrúpedo a crecido llevan a problemas ya que se atasca el movimiento, por eso en este caso nos han dado mejor resultados para la morfología de cuarenta y cinco grados cuando usamos el código que no crece.

6.5.3 Comparación de resultados

Tras estos experimentos se ha llegado a la conclusión que para el caso de la morfología normal si se usa el código base o el que va creciendo en función del tiempo el resultado es muy similar. Para la morfología de treinta grados nos ha dado mejor resultado el código que aplica crecimiento mientras que en el caso de la morfología de cuarenta y cinco grados es lo contrario y nos ha dado mejor resultado cuando no hay crecimiento.

Decidimos seguir haciendo comprobaciones entre ambos códigos, para ello escogimos varios de los resultados que hacían que el cuadrúpedo se desplazara hacia delante y decidimos comprobar cual de ellos, partiendo de que realizaban el mismo número de acciones, hacía avanzar más al cuadrúpedo, para ello definimos cada una de las acciones que hace, que son dos movimientos de las piernas, como episodios.

Para ello probamos comparar los dos mejores casos que obtuvimos, que serían la morfología que se encuentra en cuarenta y cinco grados sin que el cuadrúpedo creciera que nos dio estos resultados en cincuenta episodios [6.15](#) (página 51) y en doscientos cincuenta episodios: [6.16](#) (página 52).

Cabe destacar a favor de esta morfología que realizó un avance lento pero continuo, sin desviarse mucho hacia los lados y recuperando la posición en una ocasión en la que parecía que iba a perder el equilibrio.

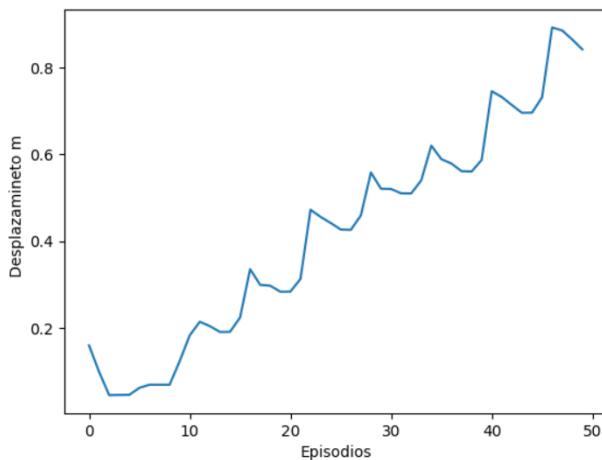


Figura 6.15: Gráfica del desplazamiento del código de 45 grados en cincuenta episodios.

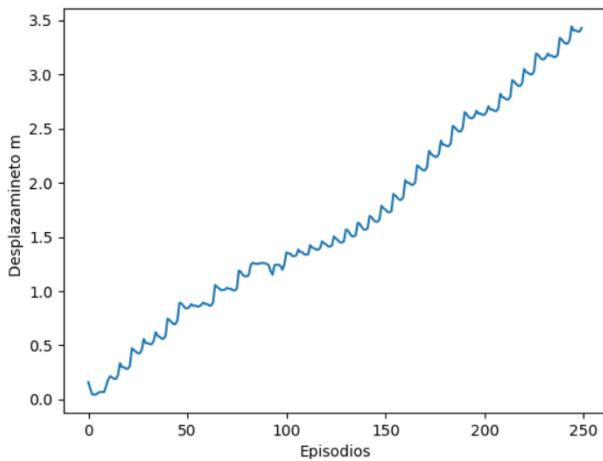


Figura 6.16: Gráfica del desplazamiento del código de 45 grados en doscientos cincuenta episodios.

La otra morfología que utilizamos es la del cuadrúpedo con las patas en un ángulo de treinta grados y habiendo aprendido con el código que crecía. Sus resultados son los siguientes para cincuenta episodios [6.17](#) (página 52) y en doscientos cincuenta episodios: [6.18](#) (página 53).

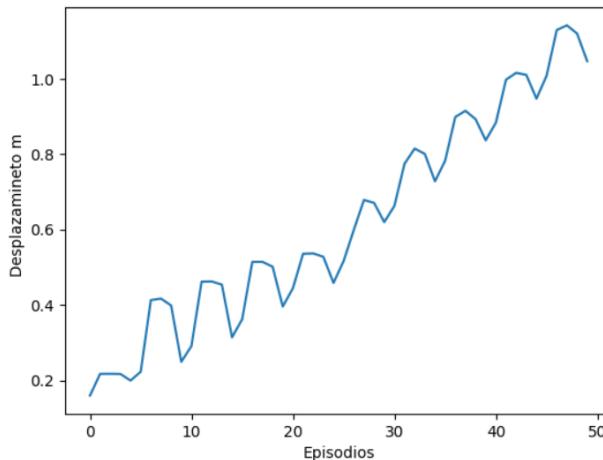


Figura 6.17: Gráfica del desplazamiento del código de 30 grados en cincuenta episodios.

Se puede observar que entre cada movimiento que le hace avanzar el cuadrúpedo retrocede un poco de terreno, haciendo un leve movimiento hacia atrás. Esto ocurre porque su desplazamiento se asemeja a una acción de obtener impulso antes de dar un salto hacia adelante. También es importante destacar que este, a diferencia del anterior, se desplaza bastante a la izquierda mientras va avanzando, haciendo que el desplazamiento sea más diagonal que recto. Aun así en doscientos cincuenta episodios fue capaz de llegar al final del mapa del

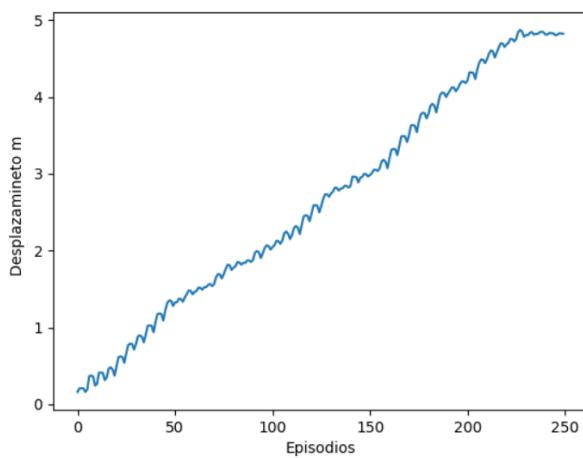


Figura 6.18: Gráfica del desplazamiento del código de 30 grados en doscientos cincuenta episodios.

Coppelia, lo cual es más que la opción anterior.

Comparando ambas se vio que la mejor de treinta grados se desplaza más que la mejor de cuarenta y cinco grados, aunque esta tiende a desviarse más y su desplazamiento es más irregular.

Capítulo 7

Conclusiones

En este último capítulo hablaremos de las conclusiones a las que hemos llegado gracias a los resultados obtenidos durante estos experimentos, además de comentar el conocimiento y competencias obtenidas durante la realización de este trabajo.

Durante la realización de este proyecto hemos trabajado con la idea de que robots con patas aprendieran a desplazarse hacia delante usando aprendizaje por refuerzo. Decidimos usar el algoritmo de Qlearning ya que nos permitía empezar trabajando en un caso muy específico y ‘posteriormente ir escalando la complejidad de la morfología del robot para así poder hacer que robots más complejos aprendieran. Nuestros códigos fueron probados sobre un cuadrúpedo con joints que le permitían mover las piernas y las rodillas. En nuestra primera aproximación tratamos de hacer que aprendiera a moverse en bucles de dieciséis movimientos, pudiendo mover cualquiera de esos joints. Como nuestro código no se encontraba lo suficientemente refinado no fuimos capaces de limitar bien la ecuación de refuerzo necesaria para el Qlearning y no conseguimos que el robot aprendiera a desplazarse.

Versión tras versión de nuestro algoritmo de aprendizaje fuimos tratando de enseñar a nuestro cuadrúpedo a caminar hasta que, en el código que denominamos como Cuadscene11, empezamos a obtener resultados satisfactorios. La morfología que usaba ese código nos permitía mover solo los joints de las patas y cada acción del algoritmo de Qlearning estaba asociada a dos movimientos cualesquiera de las patas. A partir de ese primer resultado fuimos escalando el código, añadiendo más posibles movimientos, cambiando el ángulo inicial en el que los joints de las rodillas se encontraban y añadiendo que la dimensión de las patas fuera creciendo durante el transcurso del aprendizaje, consiguiendo que el cuadrúpedo aprendiera a caminar de forma autónoma. Tras esto nos dispusimos a comparar de manera analítica los diferentes resultados positivos que habíamos obtenido y comparamos cuanto se habían desplazado en el mismo espacio de tiempo, viendo que aunque varios de ellos hacían avanzar al cuadrúpedo no todos conseguían alcanzar la misma distancia.

La utilización del Qlearning y su escalabilidad nos permitió que con muy pocos cambios

podamos ir añadiendo nuevos movimientos y joints a la morfología para poder aumentar la complejidad del aprendizaje. Lo único que se necesitaría sería poder aumentar el tiempo que pasen los experimentos en el CESGA, ya que cuanto más complejo sean los futuros experimentos más tiempo se necesitará.

Podemos decir que hemos alcanzado los objetivos propuestos al inicio del desarrollo de este TFG. Hemos conseguido implementar un algoritmo de aprendizaje capaz de enseñar a un robot con patas a desplazarse hacia delante. También hemos implementado un código capaz de que la morfología de nuestro cuadrúpedo fuera creciendo durante el aprendizaje y hemos comparado los resultados entre ambos algoritmos.

7.1 Conocimientos adquiridos

Este proyecto se encontraba planteado dentro del campo de la programación, en específico en el entrenamiento por refuerzo orientado a robots con patas. Es cierto que gracias a mi formación tenía conocimientos básicos sobre ambos temas, pero la realización de este trabajo me ha llevado a ampliar dichos conocimientos.

Dentro del campo de la robótica me sirvió para familiarizarme con el trabajo con robots en un simulador como puede ser el CoppeliaSim. Haciéndome ver que cualquier pequeño detalle pueda hacer variar su comportamiento. Además, también tuve la posibilidad de montar yo mis propios robots en el simulador, lo cual me ayudó a entender más el comportamiento físico de ellos.

El estudio que fue necesario a la hora de programar el código para entrenar el robot cuadrúpedo y redactar la memoria me ha permitido el entender en mucho mayor profundidad el funcionamiento de los algoritmos por refuerzo. No solo entendiendo su funcionamiento, sino también haciéndome ver su utilidad para diversos cambios y lo importante que puede ser el escoger bien cuál de ellos utilizar. También gracias a la programación de este código me he acostumbrado a trabajar con ciertas librerías de Python con las que tenía poca experiencia hasta el momento.

La necesidad de acceder al CESGA para lanzar los experimentos también me ha permitido aprender como es el trabajo de investigación. Haciéndome ver la importancia que es ir apuntando las bases de todos los experimentos que uno va lanzando porque llega un momento, cuando ya has lanzado más de cien experimentos, que si no tienes bien almacenada y ordenada esa información, es imposible saber las diferencia entre unos y otros. También fue educativo la instalación de todos los programas en el CESGA y la preparación de los scripts que lanzarían los códigos, fue algo que llevo más tiempo del debido, pero que gracias a haber sido capaz de solucionarlo me veo ahora con más capacidad para resolver ese tipo de problemas.

Pese a que este ha sido un trabajo que ha conseguido llevarme al límite en varias ocasiones

al no ser capaz de obtener los resultados deseados y no encontrar los errores que los producía, la satisfacción que se generaba al avanzar cada paso ha generado en mí un gran interés en la investigación, por lo que creo importante añadir también ese interés entre lo que me ha aportado la realización de este proyecto.

7.2 Trabajo a futuro

Lo primero sería conseguir depurar aun más el código para conseguir obtener un desplazamiento continuo hacia delante que sea mucho más fluido y natural. Al obtener este desplazamientos sería bastante importante añadir los cuatro joints de las rodillas de vuelta al código y ver si se desplaza, esto no debería de dar problemas si hemos conseguido que se desplace con las cuatro patas. Tras esto veo principalmente tres vías para seguir avanzando.

1. Sería interesante el tratar de implementar otro código para el desplazamientos del robot cuadrúpedo. Para ello creo que el algoritmo de Proximal Policy Optimization es una de las mejores opciones. Tras implementar ese algoritmos se podrían comprobar los resultados entre ambos para ver cual de los dos es más eficiente a la hora de entrenar a un robot cuadrúpedo en condiciones similares.
2. El código que utilizamos actualmente tiene una limitación a la hora de decidir los movimientos que hacen las patas, permitiendo que solo se muevan a tres posiciones, cuarenta y cinco grados, cero grados y menos cuarenta y cinco grados. Esto es una limitación realizada como una medida de diseño para poder delimitar el numero de movimientos posibles y el numero de estados posibles de las patas. Aunque también hemos empezado a implementar un código con un mayor numero de posibles movimientos creo que sería interesante ir ampliando cada vez más las posibilidades ya que un cuadrúpedo real no debería de estar limitado a unos pocos posibles movimientos. Podría ser interesante el tratar de implantar un código siguiendo el algoritmo de Qlearning, pero que no limitáramos el movimiento de las patas y este pudiera ser cualquiera entre los ángulos de noventa grados y menos noventa grados.
3. Una de estas vías sería la de pasar a trabajar con el robot bípedo cuya morfología también tenemos. Esta morfología es más problemática que la anterior ya que cuenta con varios joints de rodillas que rotan en distintas direcciones. También había que implementar una función dentro del código que detectara cuando el robot al tratar de avanzar pierde el equilibrio y se cae ya que, de ocurrir esto, habría que reiniciar el robot a la posición inicial. Para hacer esto ya contamos con una primera implementación del código con la que decidimos dejar de trabajar para poder centrarnos en el cuadrúpedo.

Apéndices

Listado de acrónimos

CESGA Centro de Supercomputación de Galicia. [20](#)

IA Inteligencia Artificial. [7](#)

KL Kullback-Leiber. [14](#)

NPC Non Playable Character. [1](#)

PPO Proximal Policy Optimization. [13](#)

TFG Trabajo Fin de Grado. [3](#)

TRPO Trust Region Polizy Optimization. [13](#)

Glosario

Clip Horquilla de valores sobre la que se realiza un experimento.. [14](#)

Divergencia de Kullback-Leiber Medida utilizada para comparar dos distribuciones probabilísticas y comprobar cuanto se diferencian una de la otra.. [14](#)

External Controller Funcionalidad dentro de nuestro código que se encarga de permitir controlar el robot de forma externa, es decir, usando un código de python en vez de scripts del CoppeliaSim.. [24](#)

MatrizQ Matriz que se utiliza en Qlearning para almacenar la información del refuerzo en cada estado para cada una de las acciones que lo componen.. [8](#)

Off-Policy Subdivisión de los algoritmos de entrenamiento por refuerzo que engloba a aquellos que tienen diferentes políticas para la fase de exploración y la de explotación.. [7](#)

On-Policy Subdivisión de los algoritmos de entrenamiento por refuerzo que engloba a aquellos que tienen la misma política tanto para la fase de exploración como la de explotación.. [7](#)

Policy Gradient Method Método de aprendizaje por refuerzo que entrena a un agente para que tome las mejores decisiones.. [13](#)

Proximal Policy Optimization Algoritmo de entrenamiento por refuerzo On-Policy.. [5, 9](#)

Qlearning Algoritmo de entrenamiento por refuerzo Off-Policy.. [5](#)

Bibliografía

- [1] “Qué es la inteligencia artificial.” 2020, consultado el 12 de noviembre de 2023. [En línea]. Disponible en: <https://planderecuperacion.gob.es/noticias/que-es-inteligencia-artificial-ia-prtr>
- [2] T. Fernandez, “Inteligencia artificial: ‘boom’ de nuevas profesiones.” 2023, consultado el 12 de noviembre de 2023. [En línea]. Disponible en: <https://www.expansion.com/expansion-empleo/profesiones/2023/07/30/64c6736d468aeb72148b45b1.html>
- [3] J. McCarthy, “What is artificial intelligence?” 2007, consultado el 12 de noviembre de 2023. [En línea]. Disponible en: <https://www-formal.stanford.edu/jmc/whatisai.pdf>
- [4] S. Russel and P. Norvig, “Artificial intelligence: A modern approach,4th us ed,” 2022, consultado el 12 de noviembre de 2023. [En línea]. Disponible en: <https://aima.cs.berkeley.edu/>
- [5] Algorithmia, “¿cómo funciona la inteligencia artificial? tipos, ejemplos y ventaja,” 2023, consultado el 12 de noviembre de 2023. [En línea]. Disponible en: <https://algoritmia8.com/2021/04/22/como-funciona-la-inteligencia-artificial-tipos-ejemplos-y-ventajas/#:~:text=Russell%20y%20Peter%20Norvig%20existen,su%20vez%2C%20ofrecer%C3%A1%20sus%20subcampos.>
- [6] B. Modasara, “Aprendizaje por refuerzo - ejemplo práctico y marco.” 2023, consultado el 12 de noviembre de 2023. [En línea]. Disponible en: <https://www.alexanderthamm.com/es/blog/refuerzo-aprendizaje-marco-y-ejemplo-de-aplicacion/>
- [7] C. C. R. José Ignacio Hidalgo Pérez, “Una revisión de los algoritmos evolutivos y sus aplicaciones.” 2004, consultado el 12 de noviembre de 2023. [En línea]. Disponible en: https://www.researchgate.net/publication/277274186_Una_revision_de_los_algoritmos_evolutivos_y_sus_aplicaciones

- [8] M. Merino, “Conceptos de inteligencia artificial: qué es el aprendizaje por refuerzo.” 2019, consultado el 12 de noviembre de 2023. [En línea]. Disponible en: <https://www.xataka.com/inteligencia-artificial/conceptos-inteligencia-artificial-que-aprendizaje-refuerzo>
- [9] R. R. C. D. Cobellis, “Comparación de algoritmos de aprendizaje por refuerzo basados en q-learnings.” 2021, consultado el 12 de noviembre de 2023. [En línea]. Disponible en: <https://riuma.uma.es/xmlui/handle/10630/23354>
- [10] C. Shyalika, “A beginners guide to q-learnings.” 2019, consultado el 12 de noviembre de 2023. [En línea]. Disponible en: <https://towardsdatascience.com/a-beginners-guide-to-q-learning-c3e2a30a653c>
- [11] R. sagar, “On-policy vs off-policy reinforcement learning: the differences.” 2020, consultado el 12 de noviembre de 2023. [En línea]. Disponible en: <https://analyticsindiamag.com/reinforcement-learning-policy/>
- [12] A. Suran, “On-policy v/s off-policy learning,” 2020, consultado el 12 de noviembre de 2023. [En línea]. Disponible en: <https://towardsdatascience.com/on-policy-v-s-off-policy-learning-75089916bc2f/>
- [13] M. S. Ausin, “Introducción al aprendizaje por refuerzo. parte 5: políticas de gradiente.” 2020, consultado el 12 de noviembre de 2023. [En línea]. Disponible en: <https://markelsanz14.medium.com/introducci%C3%83n-al-aprendizaje-por-refuerzo-parte-5-pol%C3%ADticas-de-gradiente-8e92725e9c8f>
- [14] L. experts, “What are the pros and cons of on-policy and off-policy learning in rl?” 2023, consultado el 12 de noviembre de 2023. [En línea]. Disponible en: <https://www.linkedin.com/advice/0/what-pros-cons-on-policy-off-policy-learning>
- [15] Wikipedia, “hyperparameter machine learning - wikipedia.” 2020, consultado el 12 de noviembre de 2023. [En línea]. Disponible en: <https://en.wikipedia.org/wiki/Hyperparameter>
- [16] A. L. Sanchez, “Trabajo de fin de master: aplicación de aprendizaje profundo por refuerzo a problemas de robotica area,” 2020, consultado el 12 de noviembre de 2023. [En línea]. Disponible en: https://oa.upm.es/68638/1/TFM_AITOR_LOPEZ_SANCHEZ.pdf
- [17] J. Hui, “Trust region policy optimization explained.” 2018, consultado el 12 de noviembre de 2023. [En línea]. Disponible en: <https://jonathan-hui.medium.com/rl-trust-region-policy-optimization-trpo-explained-a6ee04eeee9>
- [18] “Proximal policy optimization.” 2023, consultado el 12 de noviembre de 2023. [En línea]. Disponible en: <https://spinningup.openai.com/en/latest/algorithms/ppo.html>

- [19] A. F. Simón Tena, “Control de robots mediante algoritmos de aprendizaje por refuerzo profundo .” 2021, consultado el 12 de noviembre de 2023. [En línea]. Disponible en: <https://zaguan.unizar.es/record/112250#>
- [20] I. Cornejo, “Aplicación de aprendizaje por refuerzo para el estacionamiento automático de un automóvil en un ambiente simulado,” 2021, consultado el 12 de noviembre de 2023. [En línea]. Disponible en: <https://repositorio.ulima.edu.pe/handle/20.500.12724/17545/>
- [21] N. Angarita, “Aplicación de algoritmos de reinforcement learning en el juego colonos de catán.” 2020, consultado el 12 de noviembre de 2023. [En línea]. Disponible en: <https://repositorio.uniandes.edu.co/bitstream/handle/1992/68271/Proyecto%20de%20grado.pdf?sequence=3>
- [22] C. R. Illán, “Control de locomoción de robots cuadrupedos mediante reinforcement learning.” 2020, consultado el 12 de noviembre de 2023. [En línea]. Disponible en: <http://hdl.handle.net/10045/136355/>
- [23] C. E. D. L. S. J. D. C. Gerardo Franco Delgado, M. De La Rosa, “Movimientos en pierna robótica para el pateado de un balón a través es de reinforcement learning.” 2019, consultado el 12 de noviembre de 2023. [En línea]. Disponible en: https://www.rcc.cic.ipn.mx/2019_148_8/Movimientos%20en%20pierna%20robotica%20para%20el%20pateado%20de%20un%20balon%20a%20traves%20de%20reinforcement%20learning.pdf
- [24] “Coppelia manual.” 2023, consultado el 12 de noviembre de 2023. [En línea]. Disponible en: <https://www.coppeliarobotics.com/helpFiles/>