Universidad Complutense de Madrid

Facultad de Informática

Máster Universitario en Ingeniería de Sistemas y Control

TRABAJO DE FIN DE MÁSTER

Navegación y exploración con Reinforcement Learning

David Serret Mayer

Tutores: Eva Besada Portas, Jose Antonio López Orozco

Madrid, Septiembre de 2021

Dedicado a mi hermano Albert.

Resumen

En este proyecto se va a investigar el desarrollo de algoritmos de exploración de mapas a través de aprendizaje por refuerzo. En esta última década se han producido grandes avances en las técnicas de aprendizaje por refuerzo basado en redes neuronales. A lo largo de este proyecto, estudiaremos un problema común como puede ser el de exploración de entornos mediante vehículos autónomos, lo traduciremos a un entorno virtual en el que sea posible realizar un gran número de simulaciones y finalmente expondremos los resultados frente a algoritmos ya consolidados. El proyecto está motivado por la necesidad de resolver un problema real como es la navegación en drones acuáticos para la inspección de cuerpos de agua. Este proyecto se engloba dentro una línea de investigación abierta por el grupo de Investigación de Ingeniería de Sistemas, Control, Automatización y Robótica de la Universidad Complutense de Madrid en la cual se está desarrollando un drone acuático autónomo para la medición de concentración de contaminantes en lagos.

Se han implementado varios algoritmos pertenecientes al estado del arte como son DQN, Double-DQN, A2C o REINFORCE, junto con otras herramientas relevantes en estas técnicas, como puede ser el Prioritized Experience Replay y se han desarrollado varios entornos con distinta dificultad para la evaluación de estos algoritmos.

Los resultados que se presentan como conclusiones, determinan que pese a que estas técnicas todavía no son capaces de resolver este problema con la eficiencia de algoritmos ya consolidados, ofrecen algunos resultados positivos. Y lo que es más importante, existen partes con un potencial de mejora en el futuro que quedarán identificadas en el apartado de conclusiones. Dado que el problema se ha resuelto también con dichos algoritmos consolidados, en este trabajo podemos ofrecer comparaciones entre algoritmos clásicos para la planificación y optimización de trayectorias y los que están basados en aprendizaje por refuerzo.

A grandes rasgos, los resultados finales ofrecen un algoritmo con un rendimiento que es un 50 % peor con respecto a la solución ideal y un 4 % peor con respecto a algoritmos sencillos puramente avaros. Cabe notar que la comparativa entre la solución ideal y la solución desarrollada no es justa, dado que en la solución ideal se tiene acceso a toda la información del entorno (totalmente observable), mientras que en la solución desarrollada no se puede observar el estado completamente. La segunda comparativa con el algoritmo avaro, es una comparativa justa dado que la observabilidad es parcial y semejante en los dos casos.

Por otro lado, se han obtenido resultados significativos en otros entornos secundarios, como puede ser el entorno *snake*, en el cual se ha conseguido una tasa de resolución del 40 % y mostrando unos comportamientos realmente avanzados, que muestran que el agente ha ejercido un trabajo de planificación.

Palabras Clave:

Aprendizaje por Refuerzo, DQN, A2C, Deep Learning, Redes neuronales, Planificación de Trayectorias, Exploración, Q-learning

Índice

Figuras

Tablas

Tabla de abreviaturas

A2C ..... Advantage Actor Critic

A3C ..... Asynchronous Advantage Actor Critic

API ..... Application Programming Interface, Interfaz de programación de aplicaciones

CNN ..... Convolutional Neural Network

DDPG ..... Deep Deterministic Policy Gradient

DL ..... Deep Learning

DQN ..... Deep Q Network

FIFO ..... First in First Out

IA ..... Inteligencia Artificial

ISCAR ..... Grupo de Investigación de Ingeniería de Sistemas, Control, Automatización y Robótica.

MDP ..... Markov Decision Process, Proceso de Decisión de Markov

PER ..... Prioritized Experience Replay

POMDP ..... Partially Observable Markov Decision Process, Proceso de Decisión de Markov parcialmente Observable

RL ..... Reinforcement Learning

SVM ..... Support Vector Machine

TD ..... Temporal Difference, Diferencia temporal

TFM ..... Trabajo de Fin de Máster

1. Introducción

En este proyecto se investigarán algoritmos de exploración y navegación de robots basados en técnicas de aprendizaje por refuerzo con redes neuronales profundas. El problema intenta ofrecer una alternativa a los algoritmos de navegación y planificación de trayectorias en uno de los proyectos propios del equipo de investigación. Este proyecto consiste de un drone acuático autónomo que se utilizará para la medición de concentración de bacterias de lagos y cuerpos de agua de forma automática.

1.1. Motivación original del problema

El grupo de investigación de Ingeniería de Sistemas, Control, Automática y Robótica (ISCAR) de la Universidad Complutense de Madrid está intentando encontrar una solución para el algoritmo de navegación y planificación para un drone autónomo acuático en forma de embarcación diseñado con el fin de estudiar y analizar concentraciones de contaminantes y elementos nocivos en estanques y grandes cuerpos de agua. El robot ya diseñado, funciona de forma adecuada y funcional. El ISCAR se ha propuesto rediseñar los algoritmos de planificación de trayectorias y control reactivo [5][6]. Para ello se han investigado diversas líneas algorítmicas y se ha abierto una para tener en cuenta implementaciones basadas en algoritmos de aprendizaje por refuerzo. Es en este momento cuando el autor toma las riendas de este proyecto de fin de máster para analizar el estado del arte en aprendizaje por refuerzo para aplicarlo a este tipo de entornos.

1.2. Motivación de redes neuronales

En este trabajo se evaluará si las técnicas modernas de aprendizaje por refuerzo mediante redes neuronales pueden ser de utilidad para resolver este tipo de problemas.

La resolución de problemas mediante redes neuronales y la inteligencia artificial ha sufrido una revolución en las últimas dos décadas. El incremento en la capacidad computacional, la disponibilidad de grandes cantidades de datos, el nuevo software disponible y los avances en investigación han permitido solucionar problemas complejos con asombrosa precisión y elegancia. Esto nos obliga a revisitar los problemas pasados ya resueltos con con otras técnicas y a estudiarlos con estos nuevos enfoques.

Aunque este problema puede ser resuelto mediante técnicas de optimización de trayectorias ya conocidas y estudiadas con mayor o menor facilidad, el objetivo de este proyecto es evaluar si utilizando concretamente las nuevas técnicas de aprendizaje por refuerzo es posible obtener algoritmos viables. De hecho existen algunos trabajos que intentan resolver problemas muy parecidos empleando técnicas de aprendizaje por refuerzo [7] [8]. En estos trabajos se emplea un entorno similar al que se usará en este trabajo, aunque el desarrollo de este proyecto ha sido independiente y paralelo a estos artículos. Se comenzará

desarrollando los algoritmos de aprendizaje por refuerzo que se consideren más adecuados. Una vez desarrollados estos algoritmos, se compararán frente a las técnicas ya conocidas de optimización de trayectorias. Para ello, se desarrollarán y adaptarán las versiones pertinentes de algoritmos de planificación al problema para poder hacer comparaciones y se plantearán estas comparaciones como parte de los resultados.

1.2.1. Aprendizaje por refuerzo o Reinforcement Learning

El aprendizaje por refuerzo es un método de resolución de problemas bajo un planteamiento común: un agente aprende el funcionamiento de un entorno desconocido a través de la interacción con el mismo. El mecanismo por el que el agente interactúa con el entorno es a través de acciones que se generan en función del estado del entorno y de las recompensas que obtiene el agente en dicho entorno. [9, p. 1]. El objetivo de estos algoritmos de aprendizaje por refuerzo es la maximización de la recompensa obtenida a lo largo de una experiencia en dicho entorno.

Más adelante se definirá formalmente en que consiste el aprendizaje por refuerzo, pero esta definición nos ayuda a entender que es lo que se esta buscando en este proyecto. Podríamos poner de ejemplo un ratón de laboratorio al que se le ha entrenado para resolver un laberinto. El ratón no entiende como funciona el laberinto al principio, ni el motivo por el que en el centro del laberinto puede encontrar queso (una recompensa). Pero si se permite al ratón (*el agente*) interactuar suficiente tiempo con el laberinto (*el entorno*) entenderá que efectuando ciertos movimientos por el laberinto (*acciones*) conseguirá encontrar el queso o *recompensa* con cierta facilidad.

1.2.2. Aplicabilidad del Refuerzo Positivo a la exploración de una masa de agua

Siguiendo la estructura del aprendizaje por refuerzo, vamos a replantear el problema de exploración mediante un drone acuático para la inspección de zonas contaminadas con el mismo sujeto y predicado. Es relativamente fácil de hacer, dado que en los **problemas de exploración** ya existe esta separación clara entre entorno, agente, acciones y unos objetivos que se pueden traducir en forma de recompensas. Se va a construir un entorno que consistirá en un mapa y un agente que pueda explorarlo. En nuestro caso el entorno es el cuerpo de agua que se está intentando estudiar y el agente el drone acuático que lo explora. Y la forma a través de las que el drone interactúa con el entorno son las acciones que le hacen desplazarse y le permiten obtener recompensas de algún tipo.

Una parte importante del proyecto será definir un entorno que sea apto para la simulación y cuyos resultados puedan ser aplicables a un entorno real, así como definir un sistema de acciones y recompensas que permitan que el aprendizaje sea viable.

Dado que la base sobre la que se asienta el problema es la *exploración*, tendremos que definir objetivos que favorezcan este tipo de comportamiento y modelar el problema

virtualmente para que el agente pueda realizar experiencias con las que aprender.

En el desarrollo de este proyecto dispondremos de una ventaja adicional y es que al tratarse de un problema que ya tiene soluciones previas, podremos comparar el funcionamiento de los algoritmos desarrollados. Por ejemplo, más adelante podremos ver que el problema de exploración se puede transformar a forma de grafo y resolver a través de una variante del algoritmo del viajante, que con una gran complejidad algorítmica, nos permite encontrar una solución ideal. Esta comparación nos ayudará también a entender las diferencias entre los algoritmos ya establecidos y sus diferencias fundamentales: fiabilidad, precisión, complejidad y dificultad de implementación.

1.2.3. Estudio de viabilidad

Como ya se ha comentado brevemente en esta introducción, en los procesos de aprendizaje por refuerzo se aprende a partir de la interacción con un modelo, a través de la exploración del entorno y la experimentación sobre él. Esto nos permitirá definir problemas más o menos complejos a través de la transformación del problema original en modelos de diferente complejidad. Esto es una **ventaja** con respecto a los algoritmos tradicionales, en el sentido que el desarrollo de un algoritmo *clásico* normalmente se basa en la información previamente conocida de un modelo, mientras que hay algoritmos de aprendizaje reforzado que son libres de modelo (o como se suele determinar en la literatura model-free [9, p. 8]). Esto tiene repercusiones que discutiremos más adelante, pero en resumen por ahora lo que nos interesa mencionar es que podremos crear algoritmos que son capaces de resolver problemas complejos a través de la definición de modelos complejos. Dado que a través de la experimentación con el entorno el algoritmo aprende, el problema puede ser resuelto con una política óptima de una forma que sería mucho más complejo resolver de con otro método. Y este aspecto, al menos en la teoría, supone una gran ventaja para estas familias de algoritmos y es por ello que a lo largo de este proyecto se realizarán esfuerzos tanto para comparar dichas familias con algoritmos convencionales y por extender el problema a variaciones donde los algoritmos tradicionales, basados en modelo, también sufren deficiencias.

Por poner un ejemplo concreto, una variación en la que se experimentará en las últimas fases del proyecto, es la conversión de un problema de exploración completamente observable (modelado como un Proceso de Decisión de Markov o MDP) a un problema parcialmente observable (modelado como un POMDP o Proceso de Markov Parcialmente Observable). En el caso del POMDP se utilizará un entorno que no es completamente visible en todo momento por el agente, algo que tiene sentido en tareas de exploración. Por ejemplo, un agente que solamente es capaz de observar el entorno más cercano a su posición, de la misma forma que un robot solamente es capaz de observar a través de sus sensores la sección más cercana del mapa. Mientras que modelar estos cambios en

algoritmos clásicos es posible y se hace, los algoritmos de aprendizaje por refuerzo serán fundamentalmente iguales (un entorno, un agente, acciones, estados y recompensas) y no requerirán cambios específicos con respecto a las variantes con entornos totalmente visibles.

1.3. Objetivos

Más concretamente, los objetivos de este proyecto de fin de máster son:

- Formular el problema original que se planteó para este proyecto, la navegación y exploración de un drone acuático en un cuerpo de agua, en un modelo válido para la investigación. Este proceso estará estrechamente condicionado por las limitaciones y resultados obtenidos en el desarrollo de los algoritmos de aprendizaje por refuerzo.
- El desarrollo e implementación de los algoritmos de aprendizaje por refuerzo del estado del arte para resolver todos los entornos planteados para este proyecto.
- Una vez obtenidos resultados comparables con el rendimiento esperado, utilizarlos en la aplicación específica de nuestro problema y compararlos con los algoritmos clásicos

1.4. Organización de la memoria

Esta memoria está dividida en 5 capítulos, el primero de todos es la introducción que acabamos de ofrecer al lector. En esta introducción se ha expuesto la razón de ser de este trabajo, las técnicas con las que se intentará resolver y su aplicabilidad.

El segundo capítulo aborda el estado del arte de todas las técnicas disponibles para la resolución del problema de este trabajo. Comenzaremos por una introducción histórica a la programación dinámica, continuaremos exponiendo los trabajos más relevantes a lo largo de la última década y finalmente también se mencionarán algunos trabajos que son relevantes para este proyecto o que se han empleado de alguna manera.

En el tercer capítulo se explicarán los componentes que se han desarrollado para este trabajo de fin de máster. Tendremos la oportunidad de explicar las decisiones que se han tomado en el diseño de entornos y algoritmos, las herramientas que se han utilizado y la motivación de todas estas decisiones.

En el cuarto capítulo se presentarán los resultados. Los resultados de este proyecto serán una comparativa entre diferentes tipos de algoritmos. En este capítulo se explicará en profundidad que métricas se van a utilizar, la razón de ser de estas métricas y finalmente como entender e interpretar los valores de estos resultados.

Finalmente, en el último capítulo se recogen las conclusiones más relevantes con respecto a los resultados obtenidos, estos se comentarán y valorarán y se intentará responder

a la cuestión sobre si los algoritmos de aprendizaje por refuerzo son válidos para el problema que hemos planteado. En estas conclusiones se presentarán también líneas futuras de investigación.

2. Estado del Arte y Fundamentos Teóricos

El objetivo de este apartado es dar a entender las bases teóricas sobre las que se apoya la tecnología y los algoritmos desarrollados en este trabajo de fin de máster. En primer lugar, se ofrece una breve introducción a las técnicas clásicas de aprendizaje por refuerzo y siguiendo algunos de los artículos más importantes de las últimas décadas llegaremos hasta el estado del arte actual. En este proceso entenderemos la motivación del aprendizaje por refuerzo, las carencias que presenta y cómo se han intentado resolver desde el nacimiento de esta familia de técnicas hasta la actualidad.

2.1. Aprendizaje por refuerzo

El aprendizaje por refuerzo es una técnica que nace simultáneamente en distintas ramas de investigación ante la necesidad de resolver problemas que provienen de distintos campos
.

La primera de estas ramas proviene de la familia de la Inteligencia Artificial clásica. En este campo se intentan resolver problemas a través de ensayo y error de forma similar a los algoritmos genéticos [9, p. 18]. La segunda de estas ramas es el control óptimo en los que comenzó la investigación sobre los años 50 [10, p 86]. En este campo se utiliza como estimador de función a través de la programación dinámica.

2.1.1. Programación dinámica

La programación dinámica es un campo estrechamente relacionado con la solución de problemas de decisión a través de ensayo y error. Los procesos se formulan como un vector de estados que se puede resolver a través de la toma de una o más decisiones en una o más etapas. En su fundamento, el objetivo es descomponer problemas en subproblemas de mayor facilidad de forma repetida hasta encontrar una solución trivial del subproblema. Una vez resueltos los subproblemas de menor nivel podemos recomponer las soluciones de problemas de mayor nivel a partir de las soluciones obtenidas de forma recursiva hasta obtener una solución global [10, p. 6].

Este requisito se fundamenta en el principio de optimalidad, también expuesto por Bellman. Este principio afirma que una política en un algoritmo de programación solamente será óptima si para cualquier estado inicial y decisión inicial, las decisiones posteriores tomadas por dicha política constituyen una política óptima con respecto al estado inicial [10, Capítulo 3].

2.1.2. La ecuación de Bellman

Richard Bellman propone, siguiendo la metodología de la programación dinámica, la expansión de las teorías de Jacobi y Hamilton y presenta la ecuación de Bellman como

función objetivo.

$$V(x_0) = \max_{a_0} \{F(x_0, a_0) + \beta V(x_1)\} \tag{1}$$

donde $V(x_0)$ es el *valor* del estado inicial x_0, $F(x_0, a_i)$ es la recompensa obtenida al tomar la decisión a_i en el estado x_0, β es una constante tal que $0 \leq \beta \leq 1$ y $V(x_1)$ es el valor del estado x_1 que se obtiene tras ejercer la acción a_i sobre el estado x_0. Se trata de una ecuación recursiva, en la que para calcular el valor del estado inicial es necesario calcular el valor de los estados consiguientes alcanzables a través de las decisiones (a_i).

2.2. Q-Learning

Q-Learning es un algoritmo de aprendizaje por refuerzo diseñado para la resolución de Cadenas (Finitas) de Markov propuesto por Watkins [11], que consiste en asignar valores de *calidad* $Q(s, a)$ a los pares estado-acción. Esta técnica tiene propiedades interesantes, una de ellas es que con un tiempo de ejecución infinito *en teoría* es capaz de encontrar la política óptima en todos los casos.

2.2.1. Ejecución de la política óptima

La política óptima es simple, para cada estado s el agente debe tomar la acción a cuyo valor $Q(s, a)$, que está determinado por un par estado-acción, sea mayor.

2.2.2. Entrenamiento de la política óptima

El entrenamiento es un proceso iterativo que combina la exploración del entorno a través de acciones aleatorias y de acciones tomadas con la ejecución de la política aprendida que van haciendo converger los valores Q a su valor óptimo. En cada paso se actualizan los valores $Q(s, a)$ de la siguiente forma.

$$Q(s_t, a_t) \leftarrow (1 - \alpha) \cdot Q(s_t, a_t) + \alpha(R + \gamma \cdot max_{ai}(Q(s_{t+1}, a_i))) \tag{2}$$

En esta ecuación actualizamos $Q(s_t, a_t)$ el valor de la expresión de la derecha teniendo en cuenta los siguientes parámetros:

- α representa el ratio de aprendizaje o cuánto del valor del siguiente estado se tiene en cuenta para la actualización.
- γ es el descuento con respecto a los valores Q en $t + 1$.
- R es la recompensa obtenida al pasar de $s_t \rightarrow s_{t+1}$.
- $max_{ai}(Q(s_{t+1}, a_i))$ representa el máximo valor Q obtenible para cualquier acción desde el nuevo estado s_{t+1}.

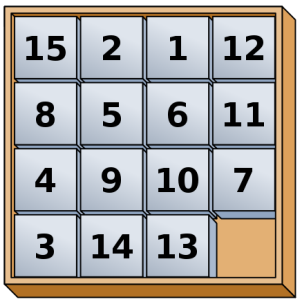


Fig. 1: Tablero del juego del 15

El proceso repite una traza con cierto rango de aleatoriedad y actualiza los valores hasta alcanzar un estado terminal. A partir de ahí el proceso se repite con un nuevo estado inicial.

2.2.3. Tabla de valores Q

Esta tabla almacena todos los valores $Q(s, a)$ que forman los pares de acción estado. Durante el entrenamiento se utiliza para actualizar los valores iterativamente y en la explotación de la solución se utiliza para seleccionar en cada estado la acción con mayor valor Q. La explotación del algoritmo es la forma en que se emplea el algoritmo una vez se ha terminado el entrenamiento. Pero cabe notar que durante el entrenamiento también se explota la solución parcialmente entrenada para muestrear experiencias no aleatorias. El algoritmo Q-Learning se apoya tanto en el teorema de optimalidad de Bellman como en la ecuación de Bellman.

2.3. El objetivo de las redes neuronales profundas

Llegados a este punto, podemos preguntarnos qué ocurre cuándo nos encontramos un problema que tiene una dimensionalidad excesivamente alta en el contexto del aprendizaje por refuerzo. Por ejemplo, el *taken* o juego del 15 consiste en ordenar 15 fichas numeradas en un tablero con 16 casillas deslizando las fichas en el espacio libre tal y como se muestra en la figura 1. La combinación de estos estados es 16! o 20922789888000 y en un problema de Q-learning por ejemplo podemos llegar a tener el triple de pares estado-acción.

Otro ejemplo interesante ocurre si tenemos que el vector de estados en un entorno es una variable continua, que puede tomar valores infinitos, como por ejemplo la lectura del sensor inercial de un drone. La utilidad de las técnicas clásicas de Q-Learning quedaría reducida a *discretizar* variables continuas asociadas a cada estado hasta formar un vector de estados, dejando al implementador la cuestión de como dividir dichos estados.

Es aquí donde entra en juego una de las principales ventajas que nos ofrecen las redes neuronales, que una vez entrenadas funcionan como aproximador de funciones o regresores.

Este tipo de modelos matemáticos nos permiten:

- Abstraer la definición de un modelo de regresión o una función de aproximación.
- Inferir información sobre datos *nunca antes vistos* fuera del entrenamiento.

Dado que existe un conjunto grande de problemas en los que no va a ser posible computar todos los valores $Q(s, a)$ en la búsqueda de una solución iterativa, o incluso problemas en los que el conjunto de estados sea demasiado grande para visitar en un tiempo razonable, utilizar redes neuronales artificiales es de una utilidad muy valiosa tal y como veremos a continuación.

2.4. Deep Q learning - DQN

El Deep Q Learning es uno de los trabajos más importantes en el replanteamiento las técnicas de aprendizaje por refuerzo y abre un nuevo campo en la aplicación de algoritmos de aprendizaje por refuerzo mediante redes neuronales profundas. Con este algoritmo se consiguen rendimientos *humanos* en algunos problemas planteados sobre pequeños juegos, como el Pong, el BreakOut o el Pinball [1]. Estos entornos están perfectamente planteados como un problema sin modelo, ya que la información que recibe el agente es la misma imagen que un humano podría recibir y las acciones que se esperan del agente están *mapeadas* directamente a los botones con los que un humano iteractuaría con estos juegos. Esto es importante, porque se relega a la red neuronal el entendimiento del entorno a través de la extracción de la información a partir de la imagen. Esto es algo que será recurrente en este trabajo y que el resto de algoritmos de aprendizaje por refuerzo también realizarán.

En la presentación original del trabajo [1] se reformula el algoritmo de Q-learning, sustituyendo uno de sus componentes esenciales, la tabla de valores Q, por un estimador con redes neuronales. Mientras que previamente se almacenaba un valor Q para cada par de estado-acción del problema, ahora se usa una red neuronal profunda que tomando el estado como entrada, que estima el valor Q de cada una de las acciones posibles. El objetivo en el entrenamiento es minimizar el error esperado en la ecuación de Bellman, pero esta vez tomando los valores Q de la red neuronal.

En la figura 2 podemos ver un diagrama representando la arquitectura de la red neuronal que se usó en el artículo original [1]. Tiene una estructura que se emplea con bastante frecuencia cuando se trabaja con imágenes y redes neuronales profundas. La entrada es una imagen RGB del estado del entorno. Las primeras capas alternan capas de convolución con capas de activación tipo RELU. Estas primeras capas son las responsables de extraer características generales de la imagen. Son de gran utilidad, ya que interpretan la información de forma local. Según avanzan las convoluciónes la información obtenida se

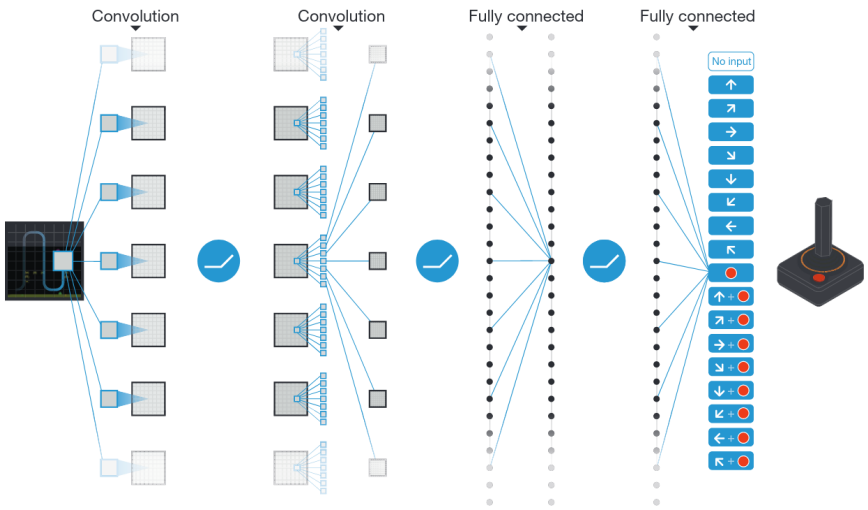


Fig. 2: Arquitectura de la red en el artículo original de DQN [1]

convierte en más abstracta y global, ya que las convoluciones tienen acceso a una superficie mayor de información (y a la vez más profunda). Finalmente, la información extraída por las capas se interpreta a través de varias capas lineales, que a su vez también están alternadas por capas de activación RELU. La salida de la red tiene el mismo tamaño que el espacio de acciones y el valor que retorna es el valor Q de cada una de las acciones para el estado (imagen) de entrada.

Ahora tomamos los valores Q estimados por la red y el objetivo en el entrenamiento de la red será minimizar el error obtenido entre el valor Q que se estima y el que se obtiene. Es un proceso iterativo e impreciso por dos razones:

- En primer lugar, el valor que se estima tiene en cuenta el valor del estado predecesor. De la misma forma que en las primeras etapas del Q-learning, si un valor Q tiene un valor que es *injustamente* bajo, los estados adyacentes seguirán actualizando sus valores con respecto a ese valor y por lo tanto tampoco conseguirán una estimación correcta de su valor Q.
- Dado que el agente solamente explora en la dirección de la acción seleccionada, los valores Q solamente se actualizan en la dirección en la que se ha tomado la acción y en la que se ha podido apreciar la recompensa.

2.4.1. Double Deep Q learning - DDQN

En los algoritmos de aprendizaje por refuerzo basados en Q-learning se produce una sobreestimación que se introduce en el sistema por el uso del operador máximo en la función de actualización de valores Q [12, p. 6].

$$Q(s_t, a_t) \leftarrow (1 - \alpha) \cdot Q(s_t, a_t) + \alpha \cdot (R + \gamma \cdot max_{ai}(Q(s_{t+1}, a_i))) \tag{3}$$

Esto se produce en parte por el efecto que tiene usar estimaciones de valores Q futuras, las cuales tienen una cierta incertidumbre asociada y en entornos ruidosos pueden ser incorrectas. En el artículo [2] se ofrece un algoritmo llamado Double Deep Q Learning para compensar los errores de estimación que se pueden producir en el Q learning simple. El funcionamiento es similar a DQN, pero en vez de entrenar y mantener una sola tabla de valores Q, se entrenan dos tablas con experiencias no relacionadas (en el artículo original se recomienda utilizar una muestra aleatoria de un conjunto grande de experiencias). Con esas dos tablas de valores Q, la actualización de valores Q se efectúa de manera similar pero se extrae la acción máxima de una tabla y el valor de Q de dicha acción de la otra tabla. Es decir, en la primera tabla se extrae la acción de valor Q máximo para el estado s_{t+1} (por ejemplo circular en dirección norte o la acción número 4) y en la segunda tabla se extrae el valor Q de ese par estado acción independientemente de si es máximo o no (es decir, el *valor* Q de la acción de ir al norte o la acción número 4). En la primera tabla se toma la acción máxima y en la segunda tabla el valor Q de dicha acción.

Para actualizar la tabla A utilizaríamos la expresión:

$$\begin{aligned} a^* &= \arg\max_{a_i} Q^B\left(s_{t+1}, a_i\right) \\ Q^A(s_t, a_t) &\leftarrow (1-\alpha)Q^A(s_t, a_t) + \alpha(r + \gamma Q^A(s_{t+1}, a^*)) \end{aligned} \tag{4}$$

Y de forma homóloga para la tabla B:

$$\begin{aligned} a^* &= \arg\max_{a_i} Q^A\left(s_{t+1}, a_i\right) \\ Q^B(s_t, a_t) &\leftarrow (1-\alpha)Q^B(s_t, a_t) + \alpha(r + \gamma Q^B(s_{t+1}, a^*)) \end{aligned} \tag{5}$$

En [12, p. 6] también se demuestra que lo errores de sobreestimación se reducen fuertemente e incluso se llega a estimar a la baja en algunos casos.

De la misma forma que DQN se sustituyen las tablas de valores Q o de pares de valor estado-acción por una red neuronal profunda que actúa como estimador de estos valores, el funcionamiento recomendado en este artículo es mantener una copia de los *pesos* de la red que está siendo entrenada [2, p. 1] para luego actualizarlos periódicamente. Esta copia de la red *desactualizada* se puede usar como la segunda tabla en DDQN. En la práctica, esta técnica se puede ver implementada de otras formas, o bien a través del entrenamiento de una red completamente nueva, mediante la forma propuesta en el artículo o con sustituyendo el paso de actualización con en una combinación lineal de las red entrenada y la copia de la última red.

$$Wb' \leftarrow (1-\alpha)Wa + \alpha Wb \tag{6}$$

Donde Wb' son los pesos actualizados de la red B y Wa, Wb son los pesos de las dos redes neuronales (Q_A, Q_B).

Estas arquitecturas de redes permiten reducir el error de estimación Q según el artículo. En la práctica permiten obtener resultados mejores de forma fácilmente corroborable en prácticamente todos los parámetros medibles como podemos ver en la figura 3. Es por esto que esta es una de las *extensiones* de DQN más populares entre los investigadores.

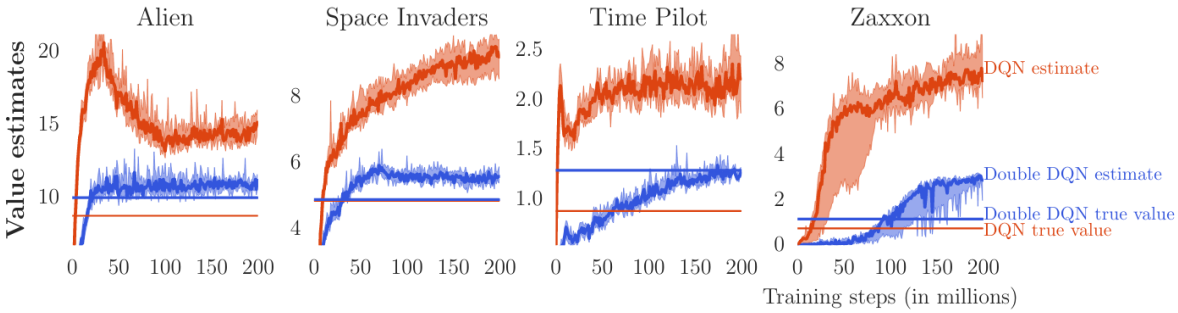
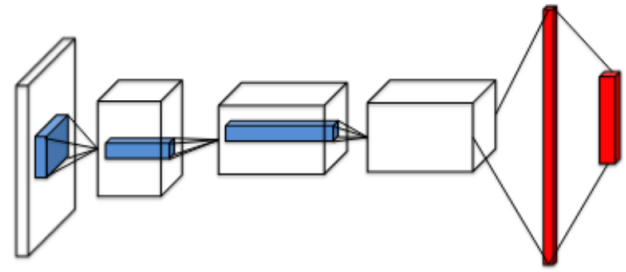


Fig. 3: Reducción de la sobreestimación de los valores Q en DDQN. Imágenes obtenidas de [2]

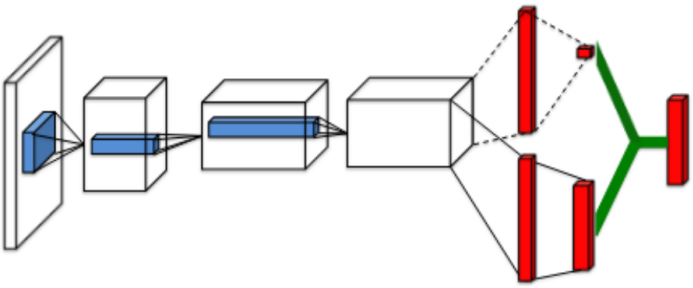
2.4.2. Dueling Deep Q learning, Dueling-DQN

Este algoritmo es una variante del algoritmo DQN que ha tenido mucha aceptación por la simplicidad de implementación y las mejoras de rendimiento que se obtienen. Es más, normalmente cuando se trabaja con algoritmos de DQN se suelen implementar con alguna variación de Dueling-DQN. El algoritmo sigue exactamente el mismo procedimiento de inferencia y de entrenamiento al de DQN normal. La única variación que existe es que la red neuronal está dividida en dos partes, una estima la *bondad* del estado y la otra estima la calidad de la acción.

Tal y como podemos ver en la figura 4, la extracción de información es común para las dos redes en las capas de convolución. Después estas capas se separan en dos segmentos diferentes de la red formando dos grupos de *fully connected*. Uno de los grupos se utiliza para calcular el valor del estado $V(s)$ y tiene una salida de dimensión 1 y el otro grupo para estimar la calidad de las acciones, y tiene una salida que es del tamaño del espacio de acciones. Finalmente, estas dos capas se combinan como si el valor del estado fuera la media del valor Q obtenido y el valor de la acción la desviación sobre la media del valor Q. Por lo tanto, se suma el valor del estado a cada uno de los valores de la acción. Los resultados mejoran considerablemente el rendimiento del algoritmo DQN hasta en un 1000 % en algunos entornos y mejora prácticamente todos los entornos con respecto a DQN estándar [3, p.6].



DQN estándar



Dueling DQN

Fig. 4: Diferencias entre DQN y Dueling-DQN. Figura extraída de [3]

2.4.3. Averange DQN

Esta es otra variación con respecto a DQN introducida en 2017 [13, p.6]. Esta variante se plantea en pos de una solución a uno de los problemas más prominentes en el algoritmo DQN, la actualización iterativa del valor de la red. De la misma forma que se concluye en Double-DQN, el estimador máximo para el estado s_{t+1} es problemático porque conlleva problemas de estabilidad y sobreestimación.

La forma en la que se resuelve este problema en este artículo es tomar los valores Q del estado s_{t+1} como la media de los valores tomados por varias redes, o más bien, la misma red pero con copias de los pesos en diferentes puntos del entrenamiento.

$$Q(s_t, a_t) \leftarrow (1-\alpha) \cdot Q(s_t, a_t) + \alpha \cdot (R + \gamma \cdot \frac{\sum_{n=1}^{m} max_{ai} \cdot (Q_n(s_{t+1}, a_i))}{m} \tag{7}$$

Este algoritmo promete una reducción del error significativo de sobreestimación de los valores Q, además de unos incrementos del rendimiento generales buenos [13, p.6]. En la realidad, este artículo es el de menor relevancia con respecto a todos los mencionados anteriormente, pero el concepto de tomar una serie de copias de los pesos de la red neuronal durante el entrenamiento es algo que se usará más adelante en este proyecto y la inspiración de esta idea viene directamente influida por este artículo.

2.5. Almacenamiento de experiencias, Prioritized Experience Replay, PER

Los algoritmos basados en Q-learning y más concretamente los algoritmos basados en DQN son estrictamente *off-policy*. Esto significa que una experiencia formada por la tupla s_t, a_t, r_t, s_{t+1} se puede reaprovechar en cualquier punto del entrenamiento futuro. Normalmente, en los trabajos basados en DQN se implementan distintos tipos de bufferes de memoria que permiten acceder a las experiencias generadas a través de la interacción con el entorno para luego utilizarlas en la fase de entrenamiento. La forma más simple sería obtener una experiencia y automáticamente hacer un ajuste en la red neuronal/función de estimación. Este mecanismo es ineficiente y produciría unos gradientes inestables y sensibles al ruido. La otra forma más común de usar estas experiencias sería insertarlas en un buffer y después de N ciclos de exploración, realizar uno o varios entrenamientos con la información acumulada. Este método es más eficiente ya que permite entrenar la red con un **conjunto** de datos muestreados, evitando así el sobre-ajuste a los datos individuales. Finalmente una forma válida y eficiente de gestionar el entrenamiento es crear una cola de experiencias en la que cabe un número finito de experiencias N. En el momento en el que se inserta la experiencia N+1, la experiencia más antigua se elimina (cola FIFO o first-in, first-out). El entrenamiento se realiza muestreando aleatoriamente experiencias en la cola de experiencias y este es el método que se usó en las primeras versiones de DQN [1].

Las experiencias en DQN tienen mucha importancia, por lo que balancear el ratio exploración-explotación es importante para no tener un aprendizaje lento y aun así, y dependiendo del entorno sobre el que se este trabajando, muchas de las experiencias tienen poco valor para la red, dado que son puntos de información que ya están bien ajustados y en los que la predicción de la red es efectiva. Podríamos verlo por ejemplo en un videojuego de coches: en las rectas conducir es sencillo mientras que en las curvas es donde se marca la diferencia entre un agente experto y otro que no lo es. Y aun así, la memoria estará llena de experiencias pasadas en rectas, porque las rectas suponen la mayor parte de la interacciones. Esto es claramente ineficiente.

En el trabajo Schaul. Et al. [4] se presenta un sistema para muestrear de forma más eficiente las experiencias en DQN utilizando un estimador para determinar la *prioridad* de las experiencias en función de la diferencia temporal. La diferencia temporal es un estimador que según ellos mismos ya se ha usado para estas mismas funciones en otros trabajos.

Si revisamos la ecuación de Bellman aplicada a Q-learning y concretamente a DQN tenemos que:

$$Q(s_t, a_t) \leftarrow (1 - \alpha) \cdot Q(s_t, a_t) + \alpha \cdot (R + \gamma \cdot max_{ai}(Q(s_{t+1}, a_i))) \tag{8}$$

En un agente determinista completamente entrenado, el valor Q de un par estado acción

es en teoría exactamente al valor máximo del estado al que se produce transición s_{t+1} multiplicado por el factor de descuento más la recompensa obtenida en dicha transición. Es más, el objetivo de la minimización en DQN es precisamente minimizar esa diferencia temporal. Podríamos entenderlo como el error producido *a priori*, previamente al entrenamiento.

$$TD(0) = (R + \gamma \cdot max_{ai} Q(s_{t+1}, a_i)) - Q(s, a)) \tag{9}$$

O siguiendo la notación del artículo [4, p. 13]:

$$\delta_t := R_t + \gamma_t \max_a Q\left(S_t, a\right) - Q\left(s_{t-1}, a_{t-1}\right) \tag{10}$$

Concretamente, se utiliza el valor absoluto de la diferencia temporal, dado que un error negativo, sigue siendo un error en la estimación y tiene un efecto sobre el agente similar al de un error positivo.

Con este valor, cada vez que se muestree una experiencia podemos actualizar su importancia o *prioridad* en el buffer de experiencias. Las experiencias con un valor mayor serán tendrán una probabilidad mayor de ser muestreadas. Tras ser muestreadas con mayor frecuencia su importancia será reducida dado ya han sido aprendidas en el proceso.

Pese a que muestrear experiencias difíciles es ventajoso, puede hacer que las experiencias sean monótonas y el entrenamiento pierda validez precisamente por el motivo contrario: muchas experiencias desconocidas que producen ruido en el entrenamiento. Es por esto que los autores proponen suavizar el efecto de las experiencias con el objetivo de controlar el *bias* que puede producir la repetición excesiva de elementos con alta diferencia temporal. Para ello, en primer lugar un parámetro α que suaviza la probabilidad de que se dichas muestras. El parámetro está posicionado entre 0 (muestreo completamente uniforme), y 1 (probabilidad proporcional al TD).

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha} \tag{11}$$

Donde p_i es la probabilidad de un elemento de ser muestreado. Después se añade también un parámetro β para compensar en la fase de entrenamiento la *importancia*:

$$w_i = \left(\frac{1}{N} \cdot \frac{1}{P(i)}\right)^\beta \tag{12}$$

De tal forma que el paso del gradiente en la fase de entrenamiento se multiplica por w_i:

$$\Delta \leftarrow \Delta + w_j \cdot \delta_j \cdot \nabla_\theta Q\left(S_{j-1}, A_{j-1}\right) \tag{13}$$

Finalmente se proponen otros métodos para la estimación de estos valores w_i, como el

índice en una lista ordenada con todas las experiencias (de mayor a menor error temporal, donde el de más valor es 1 y el de menor valor es 0 y el resto de los valores del buffer tiene un valor intermedio) por el error temporal o simplemente proporcional al error temporal. En este trabajo, por simplicidad se implementará todo mediante el método proporcional.

Los resultados mejoran mucho el rendimiento con respecto a los artículos originales de DQN [1]. En la figura 5 podemos observar un incremento considerable en los resultados obtenidos por la misma arquitectura de red neuronal basada en DQN si se entrena con experiencias muestreadas con PER. El algoritmo que se usará en este proyecto es la variante *proporcional*, mientras que los resultados sin PER se corresponden con la etiqueta con la etiqueta *uniform DQN*.

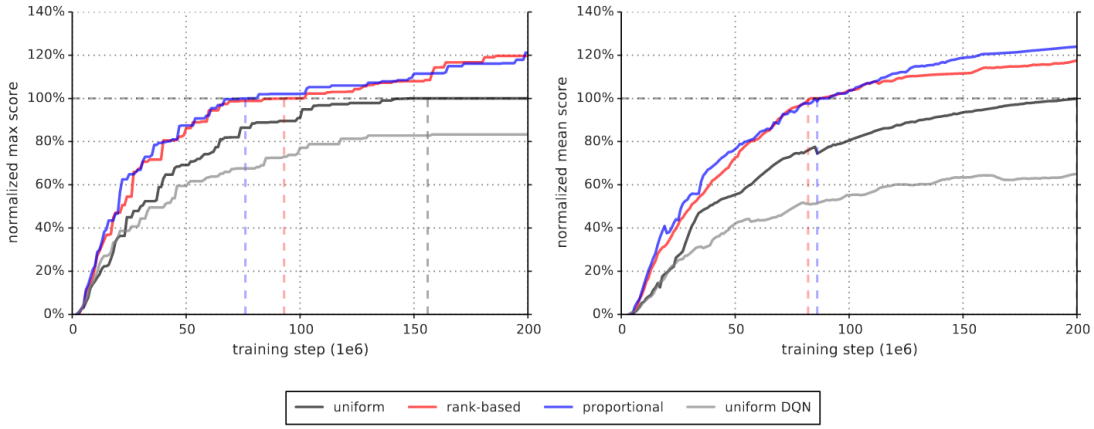


Fig. 5: Mejoras de rendimiento obtenidos a través de PER. Imagen extraída de [4]

2.6. Policy Gradient y métodos de reducción en la política

Bajo esta categoría se recogen varios algoritmos que comparten algunas características. Citamos directamente el artículo de Suton et al. [14, p 1]:

Rather than approximating a value function and using that to compute a determinis-tic policy, we approximate a stochastic policy directly using an independent function approximator with its own parameters. For example, the policy might be represented by a neural network whose input is a representation of the state, whose output is action selection probabilities, and whose weights are the policy parameters. Let 0 denote the vector of policy parameters and p the performance of the corresponding policy (e.g., the average reward per step). Then, in the policy gradient approach, the policy parameters are updated approximately proportional to the gradient

En definitiva, en vez de minimizar la *diferencia temporal* como hacíamos en Q-learning, optimizaremos directamente la red que se encarga de dirigir la política. Para ello, en cada

traza de experiencias, haremos que las recompensas R y los parámetros θ se aproximen al gradiente:

$$\Delta\theta \approx \alpha \frac{\partial R}{\partial \theta} \tag{14}$$

Las publicaciones que plantean estos métodos son relativamente antiguas. Veamos por ejemplo el funcionamiento del algoritmo REINFORCE, que es una de las variantes de algoritmos más populares de esta familia, siguiendo el capítulo 13 de Suton & Sarto [9]. En primer lugar, el objetivo es maximizar la esperanza de recompensa a través de todos los estados visitados:

De esta maximización podemos actualizar los pesos de la siguiente manera:

$$\theta \leftarrow \theta + \alpha G(t) \nabla_\theta \ln \pi_\theta \left(A_t \mid S_t\right) \tag{15}$$

En este caso, las $G(t)$ son recompensas compuestas, donde cada una de ellas es la recompensa recibida por la experiencia más todas las futuras multiplicadas por un factor γ, $\pi_\theta \left(A_t \mid S_t\right)$ la distribución de probabilidades en las acciones y ∇_θ las acciones seguidas.

Finalmente en el framework de ML (Tensorflow) se cambia el signo de la función objetivo para convertir el problema de maximización en uno de minimización y se modela con una función de coste de entropía cruzada.

En esta familia de técnicas, el resultado de la inferencia en la función a minimizar (y más adelante en la red neuronal) retorna una probabilidad de transición que se usa para determinar la explotación del algoritmo. Las probabilidades de transición se ejecutan de forma acorde para realizar la transición. Esto es un factor importante, ya que no es necesario emplear algoritmos de exploración como ϵ-greedy y aunque algunas publicaciones denotan este algoritmo como *off-policy*, no lo es. En el momento en el que se aplica un gradiente, es decir, se realiza un paso de optimización, las experiencias pasan a estar automáticamente en una distribución de probabilidades desactualizada y no se pueden utilizar más. En estos artículos se emplea el término *off-policy* para referirse a que se muestrean experiencias durante un episodio completo antes de minimizar, en vez de minimizar tras cada una de las experiencias individuales.

2.7. Advantage Actor Critic (A2C), Asyncronous Advantage Actor Critic (A3C)

Estas técnicas se presentaron en 2016 [15] y son algoritmos que han permitido avanzar con respecto a los algoritmos de Q learning. Fundamentalmente, emplean dos redes neuronales que trabajan conjuntamente evaluando, tanto el valor de los pares acción-estado (critic), como el valor de la política de las acciones (actor). Estas dos redes neuronales representan por tanto a los algoritmos de Policy Gradient por un lado y a los de Q-learning por el otro. Se trata de un algoritmo que es *on-policy* ya que la red neuronal basada en

Policy Gradients necesita de una *traza* de experiencias recientes, que son las relevantes con respecto al entrenamiento, mientras que la red de estimación de valor podría ser potencialmente off-policy.

Primero, se muestrea un conjunto de experiencias que se toman mediante la política que la red actora tenga en ese momento. La última capa de la red de actora está normalizada mediante una activación de tipo softmax (que nos garantiza que el módulo de esta capa será unitario). Muestrear con la red *actor* es tan sencillo como introducir el estado en la entrada de la red y recibir un vector que representa una probabilidad para cada una de las acciones disponibles. La forma de explorar el entorno es más similar a la que estamos acostumbrados a utilizar con Policy Gradient más que con DQN, lo cual es una ventaja en general, porque no es necesario balancear la exploración/explotación mediante técnicas como ϵ-greedy. Una vez realizado un episodio de experiencias completas, se componen las recompensas, es decir, el valor de cada una de las recompensas es la recompensa recibida más la recompensa en cada uno de los episodios siguientes por un factor de descuento:

$$R_i = r_i + \gamma R_{i+1} \tag{16}$$

donde i está dentro del rango de experiencias en un episodio, R denota la recompensa compuesta y r la recompensa recibida en el entorno.

Una vez compuesta la recompensa, podemos entrenar ambas redes con las experiencias acumuladas en el episodio. Mientras que la red crítica se entrena minimizando la diferencia temporal entre el valor calculado y la predicción de la red (de la misma forma que en DQN), la red actora estaría minimizando la entropía cruzada de la siguiente forma:

$$\mathrm{H}(y, x) = -\sum_x y \log p(x) \tag{17}$$

dónde y codificaría la acción tomada y $p(x)$ la probabilidad muestreada en la red actora. Pero estos valores estarían ponderados mediante el valor de la **ventaja** que ofrece el par estado-acción, es decir, la diferencia entre la recompensa compuesta y el valor de la predicción del crítico. Esto introduce la interacción entre el crítico y el actor.

$$H'(p, q) = -\sum_x adv(s) p(x) \log q(x) \tag{18}$$

Esta es la fuente fundamental de la fórmula:

$$d\theta \leftarrow d\theta + \nabla_{\theta'} \log \pi \left(a_i \mid s_i; \theta'\right) \left(R - V\left(s_i; \theta'_v\right)\right) \tag{19}$$

En último apartado, se compara el rendimiento de estos algoritmos con respecto al artículo original de DQN [1] y los resultados son, por lo general, mejores aunque hay ciertos entornos en los que ciertas extensiones de DQN todavía destacan [15, p. 14] [16,

p. 11]. En el caso de este proyecto, destacamos como ventaja principal la forma en la que se muestrean nuevas experiencias (a través de probabilidades de transición), como gran ventaja para la resolución de nuestro problema. Actuar con probabilidad permite explorar de forma más natural un entorno con una gran cantidad de soluciones posibles. Más adelante desarrollaremos esta idea.

3. Desarrollo del trabajo

En este apartado se expondrán todos los componentes que se han desarrollado para este trabajo de fin de máster. Intentaremos exponer el motivo y la razón de ser de las decisiones que se han tomado con respecto a los entornos, algoritmos desarrollados y las herramientas que se han utilizado para dichos desarrollos. El objetivo es que al terminar este capítulo, podamos pasar a exponer los resultados con las ideas y los conceptos claros para entender mejor cómo funcionan todos los componentes y presentar los resultados con claridad.

3.1. Simplificación original del problema

Para alcanzar de forma progresiva los objetivos de este trabajo vamos a simplificar el problema problema de la navegación en cuerpos de agua mediante drone en un entorno para resolver mediante aprendizaje por refuerzo. En este entorno preliminar vamos a imponer algunos requisitos que permitan mantener la esencia del problema original, pero que a la vez nos permitan plantearlo de forma más empírica:

- Un modelo sencillo que nos permita obtener resultados desde la primera iteración.
- Se va a dar más importancia a la interpretación de la información y a la toma de decisiones del algoritmo que al control del agente. Por lo tanto las ecuaciones de control del agente estarán extremadamente simplificadas con el fin de centrar los esfuerzos en la exploración.
- En primer lugar, vamos a plantear el problema de tal forma que tanto las acciones como los estados sean discretos. Si bien en un entorno real, tanto el mapa como el drone acuático pueden tener estados continuos, vamos a intentar discretizarlos de alguna manera para simplificar el aprendizaje. Es importante notar que existe la posibilidad en la mayoría de algoritmos de aprendizaje por refuerzo de tratar estados y acciones continuas, pero son procesos más complejos por lo general.
- Como ya es sabido, el entorno debe tener propiedades de MDP o POMDP. Esto tiene algunas implicaciones en la forma en la que se escribirá el problema y la forma en la que se evaluarán las recompensas.
- Dentro del proceso de decisión, consideraremos que nos enfrentamos a procesos de decisión de Markov completamente deterministas.
- Se intentará crear un sistema de objetivos y recompensas que simule el del problema original.
- Se utilizará una interfaz común y sistema de recompensas semejante a la que se utiliza en los entornos de OpenAI Gym [17].

Una vez desarrollado este entorno preliminar, se ha iterado sobre su definición para perfeccionar los resultados y reorientar los objetivos hacia los aspectos en los que un algoritmo de refuerzo positivo podría ser útil.

3.2. Entornos

A continuación, presentamos los entornos que se han desarrollado para este proyecto de fin de máster.

3.2.1. Entorno preliminar

El modelo por el que nos hemos decantado para la primera iteración del proyecto funciona de la siguiente forma:

- El entorno tiene un tamaño de tablero cuadrado de 8x8 casillas. Aunque solamente sean 64 posiciones para el agente, su complejidad es lo suficientemente elevada para analizar el funcionamiento de los algoritmos.
- El agente puede tomar 4 acciones que se corresponden con las 4 direcciones ortogonales del tablero. Si el agente realiza alguna acción que lo lleve fuera de los límites del tablero, no se producirá ninguna transición en el estado del sistema.
- Existen 5 objetivos que al inicializar el entorno se sitúan en casillas aleatorias.
- La simulación termina si el agente encuentra las recompensas o si no lo consigue tras una simulación con 100 pasos.
- El objetivo del problema es realizar las acciones necesarias para conseguir las recompensas en el tablero.
- El algoritmo representa el estado como una imagen de 8x8 y 2 canales de profundidad: uno para la representar la posición del agente y otro para la representar la posición de los objetivos.
- Las acciones se toman seleccionando un número entero en el rango [0,3] que representan las 4 acciones disponibles.
- La recompensa es 10 si el agente consigue el objetivo, -0.1 si el agente no lo encuentra y -1 si el agente intenta tomar una acción que lo llevaría al agente a salirse del tablero.

Hay que tener en cuenta cuál es el objetivo de esta evaluación: en primer lugar, este entorno que utilizaremos para asegurar la convergencia de la primera implementación de los algoritmos que usaremos más adelante. A priori no parece algo complejo, pero el problema se irá extendiendo hasta demostrar ser útil.

3.2.2. Entornos relevantes de OpenAI Gym

OpenAI Gym es una de las librerías de referencia en cuanto a aprendizaje por refuerzo [17]. Su adopción generalizada en los últimos años ha permitido a las partes interesadas en estas tecnologías compartir resultados en entornos y ensayos estándares.

En el caso de este proyecto, se han empleado varios de los entornos como demostradores de la primera implementación de los algoritmos:

- Cartpole. El típico problema del péndulo invertido, es uno de los entornos más sencillos y se ha empleado principalmente para comprobar que los algoritmos están bien escritos y diseñados.
- Lunar Lander. Consiste en aterrizar una *nave espacial* en dos dimensiones controlando el motor. Es un algoritmo con una recompensa que tiene un mayor retardo, es más dinámico que el problema Cartpole y se considera más difícil. Aun así el espacio de estados es de un tamaño reducido.
- Bipedal Walker. Consiste en controlar los actuadores de un robot de tal forma que pueda caminar. El robot tiene información sobre su entorno y su posición y el problema se resuelve en dos dimensiones, tal y como se muestra en la imagen c) en la figura 6. Es incluso más difícil, tiene salidas discretas y un espacio de acciones algo mayor.
- Breakout. El objetivo del entorno consiste en romper todos los ladrillos sin permitir que la pelota caiga al suelo, las únicas opciones de control son mover el agente a la derecha o a la izquierda. Podemos ver la forma del entorno en la imagen d) de la figura 6. Usa una imagen como entrada para la red neuronal lo cual es de nuestro interés también dado que en nuestra aplicación final este también es el caso. Tener un entorno sobre el que poder testear el sistema de extracción de información es importante.

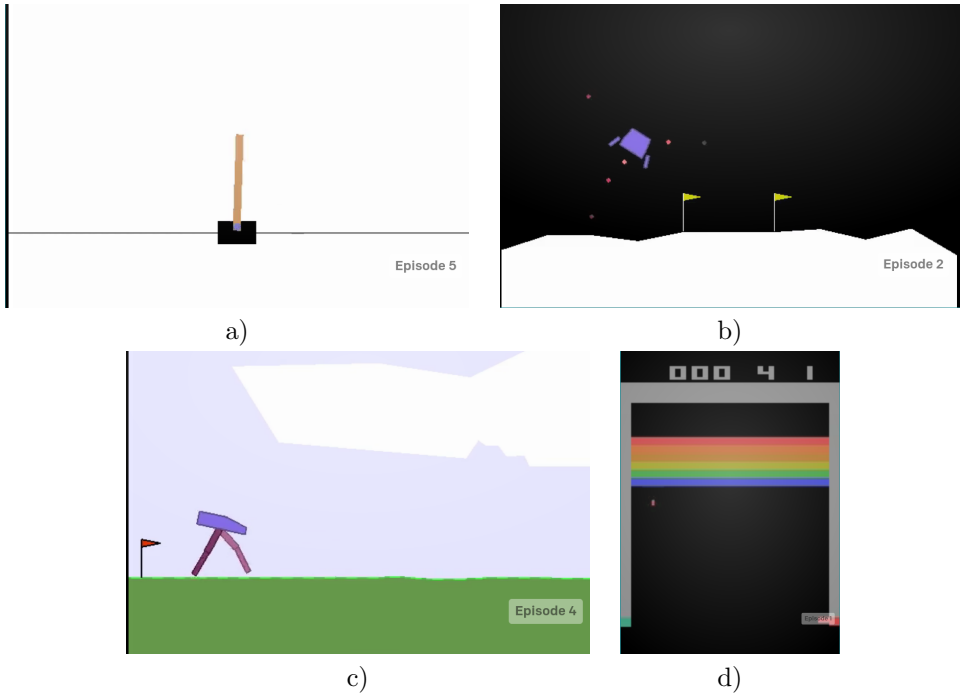


Fig. 6: Interfaz gráfica de algunos de los entornos de OpenAI Gym

En la figura 6 podemos ver imágenes que se muestran al renderizar los entornos de OpenAI Gym. El renderizado ayuda al usuario a entender el funcionamiento del entorno, pero no es representativo del espacio de estados del entorno. El orden por el que aparecen es el mismo por el que se han nombrado previamente.

3.2.3. Entorno *Snake*

En la fase intermedia del proyecto se ha creado un entorno adicional que consiste en el juego arcade *Snake*:

- El entorno tiene un tamaño de tablero cuadrado de 5x5 casillas.
- El agente puede tomar 3 acciones que se corresponden con las 3 de las 4 direcciones ortogonales dado que no está permitido dar media vuelta. Es decir, el agente puede girar 90 grados a ambos lados o continuar recto.
- Existen objetivos que van apareciendo de uno en uno en lugares aleatorios tras haber conseguido el anterior, es decir, solamente hay un objetivo en todo momento en el tablero. Al conseguir un objetivo el agente *crece* una unidad y tras su cabeza la estela de su cuerpo es una unidad más larga.
- La simulación termina si el agente colisiona con las paredes del tablero o con su propio cuerpo. El juego termina inevitablemente cuando el agente ha comido tantas recompensas que no cabe en el tablero.

- En el entorno se busca conseguir el máximo número de objetivos en el menor tiempo (o número de acciones) posible.
- El algoritmo representa el estado en una imagen tan ancha y alta como el tablero, con 4 canales de profundidad. El primer canal representa la posición de la cabeza del agente. El segundo canal representa la posición de todos los elementos del cuerpo. El tercer canal representa las posiciones de los objetivos. Finalmente, el cuarto canal representa la dirección del agente, marcando (de forma poco eficiente) la dirección en la que está viajando el agente (algo necesario desde las condiciones de observabilidad del MDP).
- Las acciones se codifican con un número entero [0,3] representando las 4 direcciones cardinales. En caso de que el agente tome la acción de girar 180^{o}, que es una acción no permitida, seguirá en la dirección previa.
- La recompensa es 2 si el agente consigue uno de los objetivos del tablero, -0.1 si la no consigue recompensa y -1 si el agente colisiona con el límite del tablero o consigo mismo.

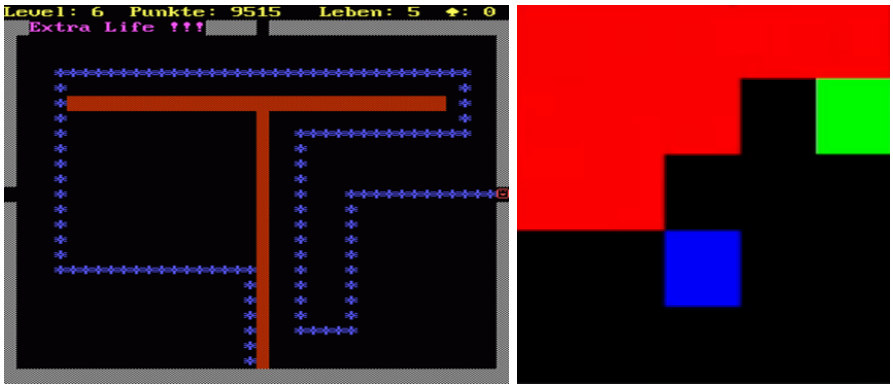


Fig. 7: Dos entornos de Snake tal y como se presentan a un usuario.

En la figura 7 vemos a la izquierda el juego original, a la derecha el que se ha diseñado para el entrenamiento de este entorno con una interfaz semejante a la de OpenAI Gym. En verde se representa la cabeza de la serpiente, en rojo se muestra el cuerpo y finalmente las recompensas están señalizadas en azul. En el entrenamiento, el agente recibe una matriz o tensor de 5x5x4 como datos de entrada.

3.2.4. Entorno Objetivo

Este es el entorno que se utilizará para la evaluación final de este trabajo. Se trata de un entorno que cumple todos los requisitos que se marcaron al inicio de este proyecto como fundamentales para la exploración del mapa. Contiene elementos de dificultad propios del

problema y algunos adicionales que harán que la resolución a través de aprendizaje por refuerzo tenga más sentido.

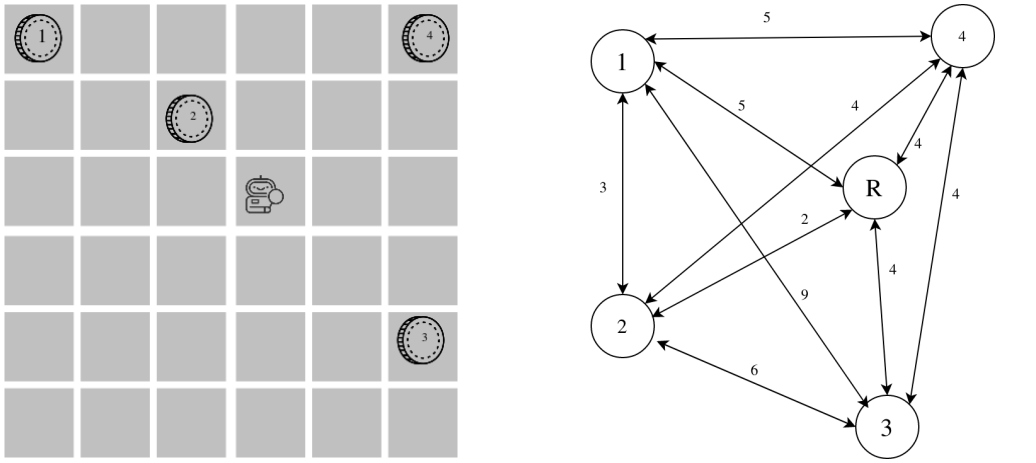
Estos elementos son principalmente incertidumbres en el entorno, los cuales hacen que modelar el problema a través de un algoritmo de exploración clásico sea mucho más difícil de justificar, aunque es verdad que este tipo de implementaciones siguen siendo viables. Más concretamente, se trata de negar la visibilidad completa del mapa y limitarla a las casillas más cercanas al agente explorador. Si el entorno mapa se compone de un tablero cuadrado de N casillas, el agente solamente podrá saber el estado de una casilla (si existe una recompensa o no) si visita alguna de sus casillas adyacentes.

Se procede a definir por lo tanto el entorno:

- El entorno tiene un tamaño de tablero cuadrado de 12x12 casillas.
- El drone acuático o agente puede tomar 4 acciones que se corresponden con las 4 direcciones ortogonales.
- Existen 8 localizaciones en el mapa que conviene medir y que aparecen en posiciones aleatorias. La posición inicial del agente también es aleatoria.
- La simulación termina cuando el agente consigue descubrir y alcanzar la posición de todas las localizaciones a medir o, en su defecto, si no lo ha conseguido tras 200 pasos o acciones.
- El objetivo del problema es conseguir el máximo número de medidas útiles en el menor tiempo (o número de episodios) posible.
- El entorno representa el estado con una imagen tan alta y ancha como el mapa, de 3 canales de profundidad: En el primer canal se aprecia el estado del agente, en el segundo canal se representan las localizaciones a medir que no han sido obtenidas y en el tercer canal se representa si la casilla ha sido observada. Como se ha estipulado anteriormente, el estado de una casilla es conocido si se ha visitado alguna de las casillas adyacentes (en cualquiera de las 8 direcciones ortogonales y diagonales) pero para tomar la medición no basta con observar, hay que visitar la casilla. En el segundo canal no se muestran las recompensas por obtener si la casilla no se encuentra en estado *descubierto* es decir, el tercer canal enmascara al segundo.
- Las acciones se codifican con un entero en el rango [0,3] representando las 4 acciones disponibles.
- La recompensa es 2 si el agente consigue una recompensa, -0.1 si la casilla visitada no contiene recompensa. Finalmente, si el agente toma una acción que le llevaría a colisionar con el límite del tablero, recibe una recompensa de -1 y no se produce ninguna transición en el estado.

3.2.5. Transformación del entorno objetivo a el problema del viajante.

Si suponemos que el entorno final es totalmente observable (aunque no lo sea), podemos apreciar un patrón: en el problema existen 9 nodos de interés, la posición del drone y la posición de las recompensas. Estos puntos de interés están su vez separados por un número de casillas uniforme y para efectuar una transición entre estos puntos es necesario un coste uniforme, que podríamos definir como **distancia**.



a) Problema presentado en forma de cuadrícula b) Problema presentado en forma de grafo

Fig. 8: Conversión del problema de exploración a uno de grafos.

La semejanza se presta sola y la comparación también. El problema en el que se tiene que visitar todos los nodos de un grafo con el menor coste (o distancia o tiempo) se conoce como el **problema del viajante**. Este es un problema ampliamente estudiado y conocido, en el que se puede obtener la solución ideal en determinadas situaciones (si el coste computacional lo permite) y, en la mayoría de los casos, una solución *muy cercana a la solución óptima* puede ser conseguida a través de heurística. En el caso de este problema, se trata de una variante del problema del viajante en la que no es necesario volver al punto de partida.

En la figura 8 podemos ver un ejemplo de esta analogía, a la izquierda tenemos un ejemplo simplificado y totalmente observable del *entorno objetivo*, definido con un tablero cuadrado de 6x6 casillas y 4 objetivos. A la derecha encontramos el mismo entorno representado como un grafo, en el que solamente mantenemos los puntos de interés y marcamos la distancia (o coste de transición) como la norma L1 o distancia Manhattan.

Existen dos observaciones que tenemos que tener en cuenta para resolver el problema:

- El problema del viajante es un problema computacionalmente muy complejo cuya resolución ideal como heurística requiere de algoritmos con complejidades altamente polinómicas en el mejor de los casos.

- Estos problemas parten de un entorno conocido y representado de forma eficiente en forma de grafos o matrices de adyacencia. En el caso de nuestro problema, lo único que se obtiene es una *caja negra* en forma de imagen en la que la mayor parte de la información no es útil y una interfaz de acciones, cuyo funcionamiento es desconocido. Con esto queremos decir que el paso de mapa a grafo que hemos realizado nosotros es algo que un algoritmo de aprendizaje por refuerzo no podría imitar por la profundidad y complejidad del problema.

Por este motivo tenemos que poner en valor al algoritmo de aprendizaje por refuerzo como lo que es, una técnica que sirve para resolver problemas sin modelo conocido y que puede sintetizar políticas a través de la interacción. Además, en el problema final el entorno no es completamente visible, solamente se hace visible una casilla una vez se ha visitado una casilla cercana a ésta. Modelar esto sería muy complicado a través de algoritmos como TSP ya que habría que modelar la exploración en el grafo.

Cuando usemos el entorno objetivo, vamos a comparar los resultados obtenidos mediante el aprendizaje por refuerzo contra las soluciones ideales del problema del viajante con observabilidad total. Es una comparación injusta, por lo que acabamos de exponer y además, porque el entorno tendrá acceso incluso a menos información, dado que tendrá que explorar primero. Aun así, el objetivo de esta comparación es tener un punto de referencia para comparar los resultados, en vez de medir el rendimiento como una puntuación arbitraria que se ha asignado a todas las acciones.

3.3. Estrategia de planteamiento de algoritmos

Este proyecto está rodeado de mucha incertidumbre:

- En primer lugar, no se conocía al principio del proyecto la dificultad que supondría la resolución de un problema así, aunque en el estado de la cuestión hemos podido ver problemas y planteamientos parecidos. La estrategia para resolver este problema la acabamos de exponer, se han definido una amplia serie de entornos que varían en dificultad. E incluso sobre algunos de ellos se tiene información de rendimiento por haber sido ya resueltos numerosas veces en otros proyectos.
- En segundo lugar, existe un amplio abanico de algoritmos de Aprendizaje por Refuerzo. E incluso dentro de cada una de las variantes existen muchas posibilidades de optimización y de mejoras incrementales.

Para evitar la gran combinación que se generaría al testear todos los algoritmos frente a todos los entornos, se ha diseñado un plan de desarrollo para evitar sobrecargar este trabajo con algoritmos que demuestran bajo rendimiento.

Primero, se implementará un test de cada una de las familias basado en entornos conocidos y ejemplos disponibles. Primero contra los entornos de OpenAI Gym para

comprobar que el rendimiento de los algoritmos desarrollados es similar a otros proyectos que usen estos entornos. Y después se comparará también con alguno de los entornos simplificados enfocados a nuestro problema.

Esta será la primera fase de pruebas y su objetivo analizar el funcionamiento de los diferentes algoritmos que se han implementado en este proyecto y seleccionar uno o una familia para proseguir con el trabajo. Esta selección reducida nos permitirá ajustar mejor los parámetros, la implementación y el rendimiento para conseguir mejores resultados.

En la segunda fase, se compararán las mejoras presentadas, propias e importadas de otros proyectos similares con el fin de evaluar qué efecto tienen sobre nuestro problema. En esta fase, las comparaciones serán de cada una de estas optimizaciones contra una línea *base* de algoritmo. Se emplearán los entornos implementados específicamente para este proyecto: el entorno preliminar y el entorno snake.

En la última fase, compararemos la mejor variante del algoritmo (algoritmo final) de la fase anterior frente a algoritmos clásicos. Esto será para ofrecer al lector una idea del rendimiento frente a otros algoritmos. Estos algoritmos clásicos serán TSP (que proporciona la solución ideal) y otras políticas que sean o no capaces de observar todo el estado basadas en políticas puramente avaras. Las políticas avaras se componen de reglas sencillas como por ejemplo: ir a la casilla objetivo más cercana o explorar la zona de valor más cercana.

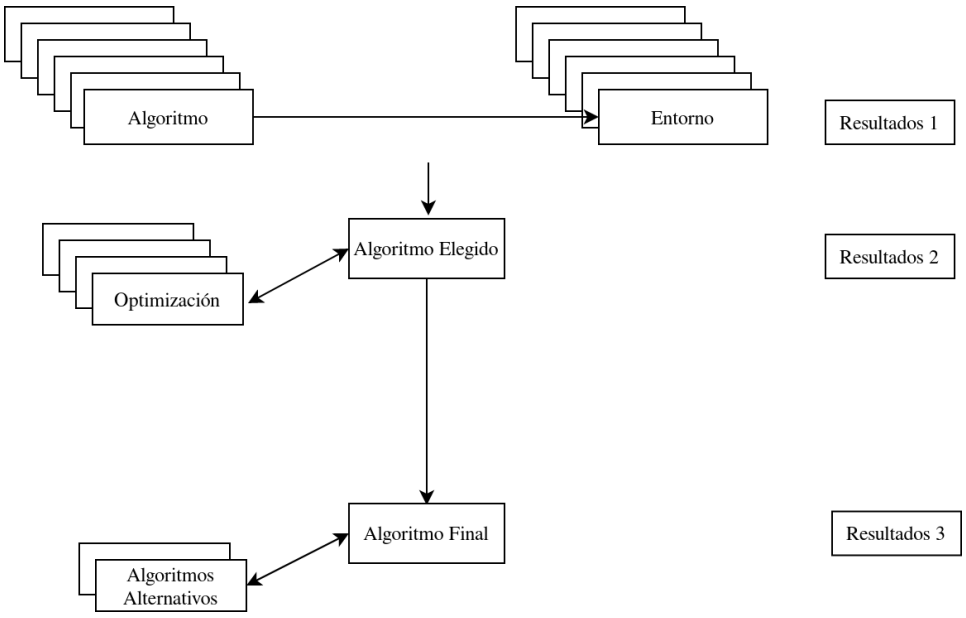


Fig. 9: Diagrama con la estrategia de desarrollo y análisis de resultados.

3.4. Implementación de los algoritmos y los entornos

Todos los algoritmos se han implementado usando Python3, mediante el la herramienta de ML Keras y acompañado por Tensorflow. Como se han desarrollado varios algoritmos,

se ha utilizado una interfaz común para todos ellos para abstraer con mayor facilidad las diferentes implementaciones de cada uno. Todo el código de los agentes tiene una interfaz común:

- Un constructor por donde se introduce metainformación sobre el entorno, tamaño del espacio de estados y de acciones, y algún hiperparámetro (como el paso de aprendizaje).
- Una función para determinar la próxima acción. Esta función determina una acción a partir de la inferencia en la red neuronal del agente. Cada una de las implementaciones varía ligeramente por la forma de los algoritmos. Por ejemplo, en A2C se muestrea una acción aleatoria teniendo en cuenta el vector de probabilidades, en Averange-DQN se muestrea en primer lugar una copia aleatoria del buffer de redes y después la acción máxima del vector de valores Q obtenido de dicha copia.
- Una función para almacenar los valores de una transición. Dependiendo si el algoritmo es *on* u *off-policy* estos valores se guardan ordenados, en un sum tree o en un buffer normal.
- Una función para realizar un ciclo de entrenamiento. Dentro de esta función también se realizan otras tareas que deberían acompañar al entrenamiento, como por ejemplo, hacer copias de la red en Average-DQN, borrar buffers en algoritmos on-policy, almacenar datos de rendimiento, procesar la información de las tuplas de transición para adecuarla al entrenamiento (véase por ejemplo el proceso de inferencia en DQN para extraer el valor máximo de s_{t+1}) o ajustar hiperparámetros de aprendizaje.

Los entornos desarrollados siguen el mismo modelo de interfaz que OpenAI Gym:

- Una función para actuar. El agente toma dicha acción en el entorno y como consecuencia se produce una transición. La función tiene como retorno este nuevo estado s_{t+1}.
- Una función para resetear el entorno (y comenzar una nueva simulación).
- Una función para representar gráficamente el estado del entorno.

Finalmente, para renderizar los entornos se ha utilizado una librería diseñada para videojuegos llamada PyGame, mientras que para la mayoría de los cálculos se ha utilizado Numpy, una librería diseñada para la computación matemática de matrices de alto rendimiento. Para resolver el problema TSP se ha empleado una librería llamada python-tsp.

3.5. Espacio de estados *amplio*, los problemas de exploración que se han observado.

En el proceso de entrenamiento, el buen ajuste entre exploración y explotación es muy importante. Tal y como hemos visto en las fases iniciales de este documento, REINFORCE y A2C usan probabilidades para muestrear la exploración. Expresado de otra manera, el resultado de la inferencia sobre las redes es una distribución de probabilidades para cada una de las acciones a tomar. Por otro lado, los algoritmos basados en DQN dan como resultado de la inferencia el valor Q de cada una de las acciones. Típicamente en los algoritmos basados en Q learning se utiliza una tasa de exploración que determina si se muestrearán experiencias de forma aleatoria o infiriendo sobre los conocimientos aprendidos previamente [1]. Esta forma de exploración es necesaria pero extremadamente ineficiente para nuestro caso, veamos dos casos:

Caso 1, sobre el entorno Snake

En las primeras fases de este juego, el agente tiene una longitud muy corta y unos objetivos muy sencillos. A medida que el agente toma experiencia, la planificación se complica, ya que si la serpiente colisiona consigo misma o si realiza una acción demasiado avara terminará en unas circunstancias que pese a que podrá sobrevivir unos cuantos movimientos más, se quedará sin espacio para maniobrar. Esto significa que tras un conjunto pequeño de episodios exitosos, una acción obtenida aleatoriamente tendrá un efecto muy negativo en la política seguida y supondrá, probablemente, la terminación inmediata de la simulación. Hasta que la tasa de exploración no se reduzca significativamente no será posible explorar las fases más profundas del entorno.

Caso 2, entorno objetivo

En el caso de los entornos de *exploración* (tanto el preliminar como el final), encontramos que existen muchas formas de resolver un mismo problema. Para entenderlo, expongamos el problema sobre el esquema que se usó para TSM.

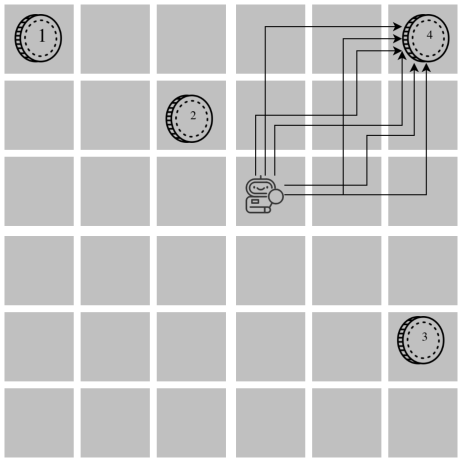


Fig. 10: Todas las trayectorias válidas del robot desde su posición a la recompensa 4.

Todas las trayectorias que podemos apreciar en la figura 10 son perfectamente válidas y óptimas para transicionar de una posición hasta la que es objetivo en la siguiente. En las primeras fases del entrenamiento, el factor de exploración es muy alto, y se toman muchas acciones que son simplemente erróneas y esto permite al agente aprender el funcionamiento básico del entorno. A medida que el entrenamiento avanza, y se espera del algoritmo algún tipo de rendimiento que permita llegar a las últimas fases del episodio, baja la tasa de exploración y comienza a explotarse más. El problema surge cuando el algoritmo *converge* erróneamente en un conjunto de estados cíclico que hace que la exploración se detenga y la memoria de experiencias se llene de experiencias repetidas y de muy poco valor añadido para el aprendizaje.

En definitiva, el aprendizaje es ineficiente porque en un muestreo de experiencias en el ciclo de aprendizaje, las experiencias se suelen muestrear de forma aleatoria y el buffer suele terminar con experiencias repetidas con muy poco valor. Al fin y al cabo, una red neuronal es una función estimadora para el valor Q de cada uno de los pares acción del entorno. Si esta memoria acaba con muchos puntos que ya están bien encajados o estimados, o están repetidos, o ambos, el entrenamiento se hace ineficiente e inviable.

3.6. Exploración mediante average DQN

El algoritmo ϵ-greedy decide si toma una decisión aleatoria o una decisión tomada con la tabla Q o red DQN que se esté entrenando. Al principio del entrenamiento, la probabilidad de tomar acciones aleatorias es 1 dado que no se conoce absolutamente nada del entorno y a medida que el entrenamiento va siendo efectivo, la probabilidad decae hasta llegar a un número muy bajo, cercano a 0. Reducir este número a 0 nos permite tomar más acciones según las políticas del agente y de esta forma se estudia el entorno con más profundidad. Por profundidad entendemos estados del entorno que solamente son

alcanzables tras una secuencia larga de acciones que solamente se pueden alcanzar tras seguir una política exitosa.

El caso 2 del apartado 3.5 hace que sea muy sencillo que se produzcan errores de entrenamiento en los entornos *Grid world* como los que estamos intentando resolver. El problema es que incluso en fases muy avanzadas del entrenamiento, el agente entra en trazas de estados cíclicas.

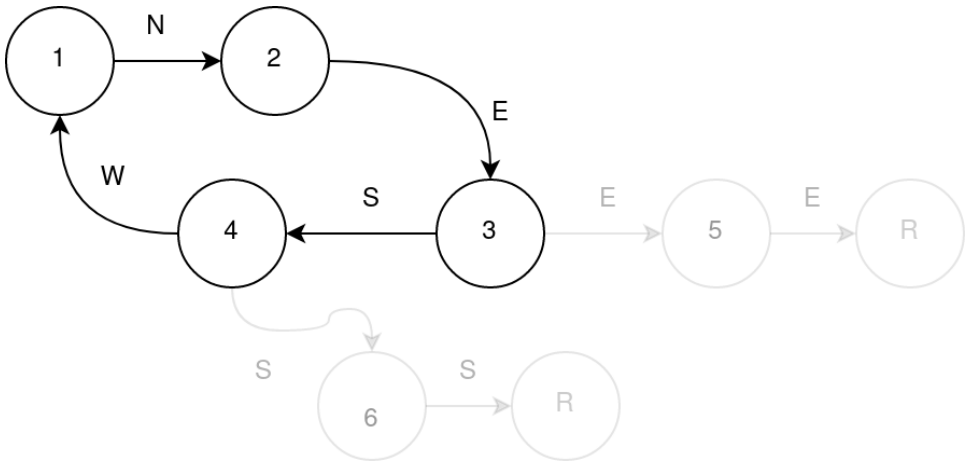
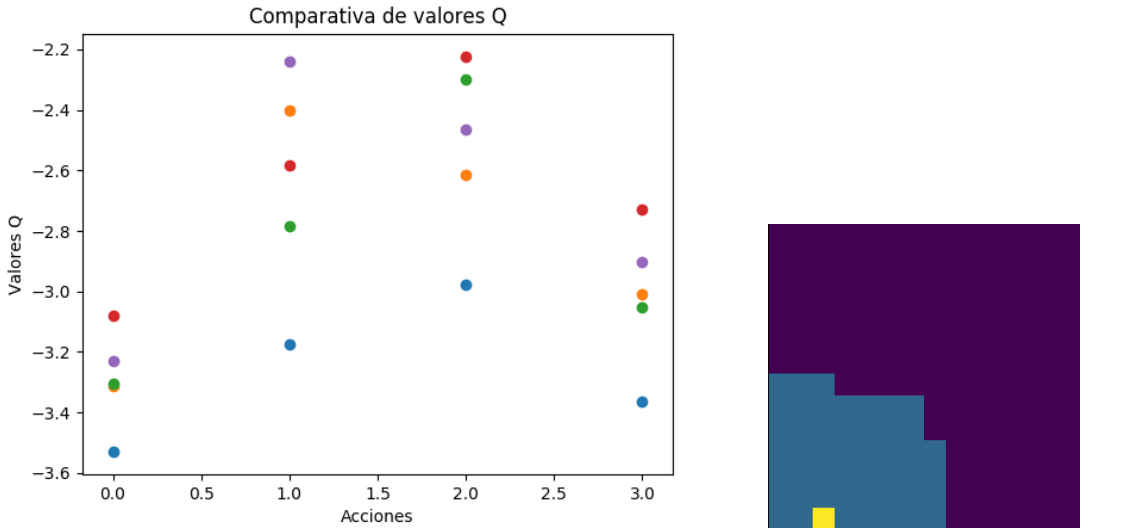


Fig. 11: Traza incorrecta seguida por el agente.

En la figura 11 podemos ver el problema al que nos referimos: el agente, por culpa de un entrenamiento incompleto o deficiente, decide que la mejor acción en el estado 3 es ir en dirección sur, presuntamente en busca de la recompensa que obtendrá tras el estado 6. Una vez llegado al estado 4, decide que volver al estado 1 es la mejor opción continua hacia el oeste y a partir de ahí se repiten de nuevo las transiciones 1 2 y 3, quedando el agente atrapado en un lazo. Estos lazos son fáciles de romper con ϵ-greedy, una acción aleatoria llevará al agente a un estado distinto a los que están en el lazo, sin embargo es muy poco eficiente que este fenómeno ocurre en prácticamente todas las fases del entrenamiento.

La solución que se ha usado en este proyecto es la siguiente: Siguiendo la filosofía de Averange-DQN [13], vamos a almacenar una copia de los pesos de la red neuronal en forma de buffer circular. En el momento de tomar acciones, se sigue el mismo procedimiento que con ϵ-greedy pero cuando sea el momento de muestrear una acción tomada por el agente (no de las aleatorias), seleccionaremos una acción empleando los pesos de una de las redes del buffer circular. En los casos donde los valores Q son muy similares, existe una probabilidad alta de que las copias tomen una política diferente. Al seleccionar políticas diferentes se rompen estos lazos y el agente puede seguir explorando sin tener que recurrir a acciones aleatorias.



a) Valores Q en el estado

b) Representación visual del estado

Fig. 12: Valores Q tomados en un mismo estado por un conjunto de 5 copias de redes.

En la imagen a) de la figura 12 podemos ver los valores Q en el estado que se muestra en la imagen b) para cada una de las 5 copias de pesos en el buffer. Las acciones están representadas en el eje de abscisas con los valores 0,1,2,3 se corresponden con sur, este, oeste y norte y los valores Q se representan en el eje de ordenadas. Cada uno de los colores representa una de las 5 copias del buffer. Se puede ver claramente que las redes están por lo general de acuerdo en el valor Q de las acciones, pero entre las que tienen un resultado similar los valores Q varían lo suficiente para que se produzcan elecciones distintas de política. Por ejemplo siguiendo las políticas de las copias roja, verde y azul, el robot circulará hacia el norte, mientras que si toma la política de la morada o amarilla circulará hacia el oeste. Ambas políticas son perfectamente válidas, ya que como podemos ver en el estado, representado en la imagen b) de la figura 12, la dirección del sur produce una colisión, y por ello todas las copias están de acuerdo en que es una opción con un valor Q bajo, la dirección oeste lleva al agente (en amarillo) a una zona ya explorada (azul) sin objetivos (rojo, en esta imagen no se puede ver ningún objetivo dado que todavía están por descubrir) y por otro lado las opciones del norte y este llevan a una zona sin explorar y con potenciales objetivos.

Esta modificación del algoritmo que proponemos sería el equivalente a añadir algo de ruido a los valores Q con el fin de producir algo de aleatoriedad. La estrategia de mantener varias copias en memoria, pese a que incrementa el uso de memoria en el entrenamiento, no tiene ningún coste adicional. La inferencia es igualmente rápida y solamente que se efectúa en una de las redes. Por otra parte, el entrenamiento solamente se produce en una de las redes también debido a que entre todas las copias existe una *maestra* que es la que se está entrenando en todo momento y es la más actualizada. El resto de las

redes son copias de la maestra que se efectúan en periodos constantes (cada 150 ciclos de entrenamiento). Además de ser eficiente respecto a la computación, también lo es con respecto a la decisión. Dado que para la explotación del entorno, se usa una copia de las redes relativamente actualizada sin tener que añadir hiperparámetros como deberíamos hacer para añadir ruido.

Esto en definitiva rompe el ciclo que hemos observado en la figura 11 y permite una exploración más estable.

3.7. Implementación de Priority Experience Replay.

Ya hemos podido entender el fundamento teórico sobre el que se apoya la técnica del Priority Experience Replay y por qué es una herramienta tan potente para algoritmos que pueden tener un aprendizaje *Off-Policy.* Desafortunadamente, no ha sido posible encontrar una herramienta que se ajuste a los requisitos del proyecto para poder utilizarla directamente, así que ha sido necesario implementar una. Esto nos ha dado una buena oportunidad para controlar mejor el aprendizaje y expandir de forma personalizable la herramienta.

Dejando de lado los fundamentos teóricos, consideramos que es interesante observar la implementación especifica del trabajo que presentan Schaul et al [4]. Por tratarse de un árbol binario, el número de nodos debe ser 2^N, los nodos que no se usan se pueden marcar con probabilidad nula para hacerlos virtualmente inalcanzables.

La implementación inicial ha consistido en dos estructuras. En primer lugar un buffer circular que almacena 2^N tuplas de experiencias. Una tupla de experiencias a su vez esta formada por 5 elementos: un estado, la acción que se tomó en ese estado, el estado al que se transicionó como consecuencia de esa acción, la recompensa que se obtuvo y si el estado que se alcanzó fue terminal.

Por otro lado, tenemos el mencionado Sum-tree. Éste está también implementado como un *montículo* en una lista con $2^(N+1)-1$ elementos. Los $2^(N-1)$ primeros valores del montículo son las ramas del árbol y se componen todas de dos hijos. El valor de cada uno de los valores de las ramas equivale a la suma de los valores de sus hijos. Por otro lado, los N últimos valores de la lista montículo son las hojas y cada una de estas se corresponde con uno de los índices del buffer circular.

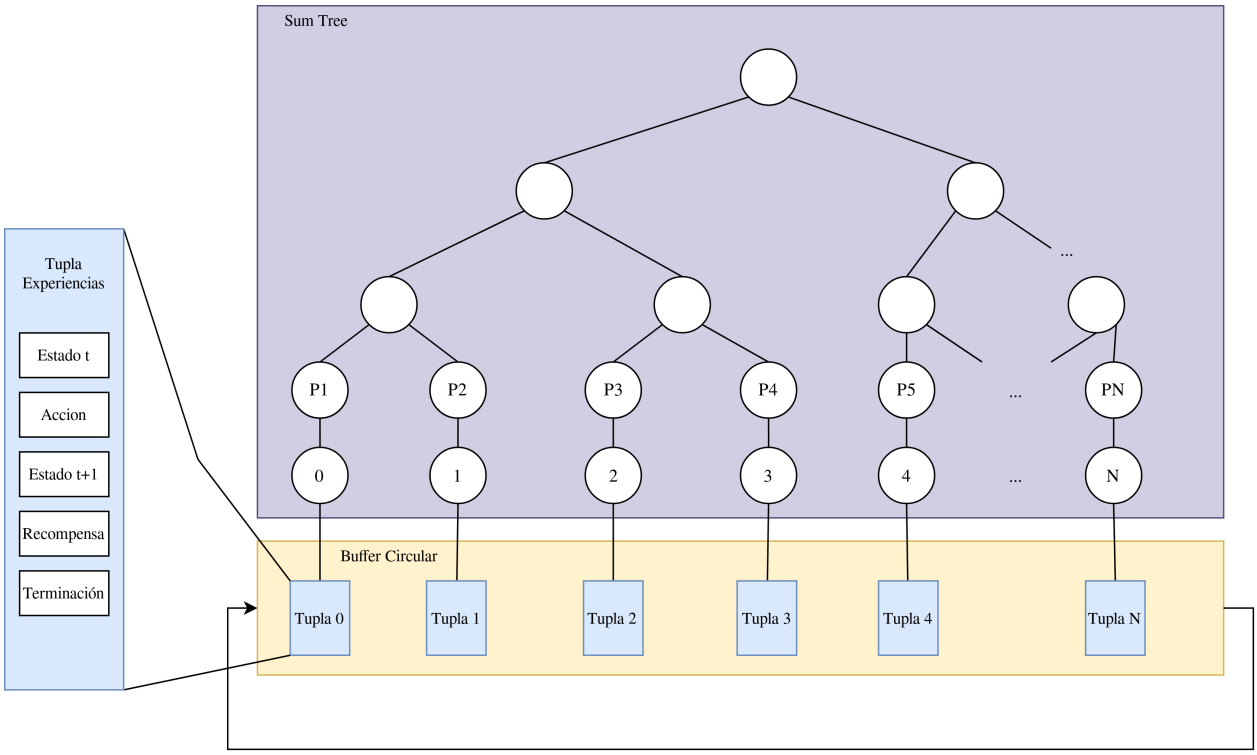


Fig. 13: Funcionamiento del sum-tree.

La estructura permite interactuar con los datos de tres formas:

- Insertando nuevos datos en la cola por primera vez. El programa tiene un valor de inserción por defecto al que se le da a la experiencia al ser insertada, normalmente suele ser un valor alto para asegurar que la experiencia se repita al menos una vez. Cada inserción tiene una complejidad computacional $log(N)$.
- Extrayendo datos. En el aprendizaje por refuerzo (y en ML en general) se suelen tomar paquetes de experiencias que se usan simultáneamente en un ciclo de aprendizaje con el fin de estabilizar este proceso. Esta interfaz permite muestrear experiencias en paquetes. En ésta se dan dos retornos, un vector con las experiencias muestreadas por importancia y otro vector con los índices a los que se corresponden dichas muestras en el árbol. Ésta es una operación que tiene una complejidad computacional $Mlog(N)$, donde M es el número de experiencias en el paquete y N es el número de experiencias en el buffer.
- Una vez se han evaluado las muestras, se procede a la parte si no más importante, a la parte más distintiva de todo este proceso, con la diferencia temporal obtenida, podemos saber el *valor o importancia* que tienen cada una de las muestras por el error que se ha producido al predecir. Estas nuevas importancias se actualizan con los índices previamente obtenidos al muestrear las experiencias y su importancia se

actualiza dentro del árbol. Ésta es una operación con un complejidad computacional $Mlog(N)$.

Finalmente se ha añadido un componente adicional desarrollado específicamente para este proyecto.

Cuando las hojas del árbol están completamente ocupadas y se añade una nueva experiencia, en la implementación propuesta en el artículo original de PER se elimina la más antigua, como debería ser en un buffer circular. Pero de la misma forma que a la hora de muestrear experiencias se toman las más importantes, podríamos emplear el mismo criterio para eliminarlas, es decir, eliminar las de menor importancia.

Por ello, de forma análoga al árbol de importancia, se ha creado un árbol de *irrelevancia* que contiene las mismas probabilidades de forma invertida (incluyendo un factor para evitar divisiones nulas):

$$P_i^{inv} = \frac{1}{P_i + \epsilon} \tag{20}$$

donde P_i es el la probabilidad de muestrear un nodo, ϵ es una constante con un valor pequeño y P_i^{inv} es la probabilidad en el árbol de irrelevancia. Por lo tanto, con esta metodología, en vez de sustituir el valor más antiguo por uno nuevo, se muestrea en el árbol inverso un valor de baja importancia y se sustituye por éste.

Finalmente, para suavizar el desmuestreo del árbol de irrelevancia, también tomaremos métricas usando la raíz cuadrada de la diferencia temporal.

$$P_i^{inv} = \sqrt{\frac{1}{P_i + \epsilon}} \tag{21}$$

Este proceso de suavizado está fundamentado en la técnica del artículo original de Prioritized Experience Replay en la que se suavizan los datos para evitar la monotonía de información, pero con el sum tree de irrelevancias. La hipótesis es que si no se aplica esta técnica, las experiencias con una diferencia temporal menor serán eliminadas del PER demasiado rápido y terminaremos con un buffer de experiencias igualmente monótono, compuesto de experiencias con una diferencia temporal alta.

3.8. Arquitectura de las redes neuronales

Los procedimientos de entrenamiento de cada uno de los algoritmos varían dependiendo de si son *off* u *on policy*. Con el objetivo de hacer comparaciones justas entre el rendimiento de todos los algoritmos y pese a que todos tienen peculiaridades que hacen que no sea posible evaluarlos con la misma red, vamos a utilizar una red con un componente inicial común, que estará formado por unas capas de convolución sobre la imagen que representa el estado del algoritmo para extraer información espacial. Después de aplanar

los pesos de la última convolución, se introducen una serie de capas densas que permiten clasificar y extraer los valores necesarios de la información. Es una arquitectura empleada habitualmente para interpretar información a partir de una imagen [18][19]

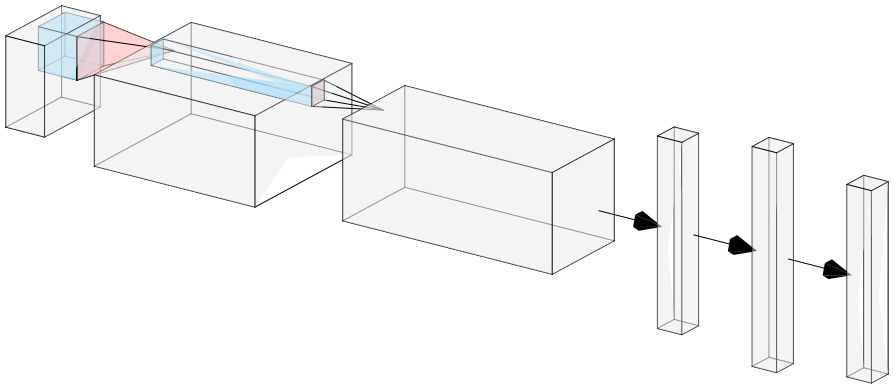


Fig. 14: Representación visual aproximada de la arquitectura común de las redes neuronales.

Lectores familiarizados con el uso de redes neuronales podrán observar que esta red neuronal común de extracción de información, tiene unas capas de convolución poco comunes y estarán en lo cierto. La imagen de entrada que representa el estado a lo largo de este proyecto oscila en tamaños de 5x5 píxeles hasta un máximo de 14x14. Los sistemas típicos de redes convolucionales CNN tienen convoluciones más profundas, con un ancho mayor y algunas otras propiedades (como Max Pooling) que esta red no tiene. Todo esto se debe a que el tamaño de imagen es muy pequeño para poder trabajar con convoluciones típicas, así que se han adaptado parcialmente, reduciendo el tamaño y profundidad y limitando o simplemente no usando las *strides*. Esto nos permite terminar con una capa de convolución similar a la de una imagen de mayor resolución y extrapolar el conocimiento aplicado a imágenes de alta resolución a nuestros algoritmos.

Tras la capa de extracción de datos, común a todos los algoritmos, cada algoritmo añade una o varias capas para adaptar los resultados de la capa inicial a la salida de la red. En el caso de los algoritmos basados en DQN se utiliza una sola salida densa con el tamaño del espacio de acciones ya que por cada entrada a la función de estados vamos a tener una salida con N valores con el valor Q de cada una de las acciones. En la figura 14 podemos ver un ejemplo de como será la red. Se compone de 3 capas de convolución con una profundidad de 3 y una profundidad de 8, 16 y 32. Finalmente después los pesos se aplanan y se procesan por un clasificador de 3 niveles de profundidad. Estos niveles de fully connected tienen unos tamaños de 128, 256 y 128.

Por otro lado en el algoritmo basado en A2C vamos a tener dos redes distintas usando dos cabezas idénticas, (en la arquitectura, al menos, los valores de los pesos serán diferentes dado que se someterán a dos procesos de entrenamiento distintos y paralelos). Una tendrá una salida con el tamaño del espacio de acciones pasada por una capa softmax de normalización para el actor y la otra tendrá sola salida, la de valor, para el crítico.

El algoritmo REINFORCE, basado en técnicas de Policy Gradient va a emplear una red con la misma entrada y con una salida similar a la del actor de A2C. Sin embargo, el proceso de entrenamiento será algo distinto.

En la exploración y explotación también es necesario tener en cuenta que para Q learning es necesario usar ϵ-greedy para explorar, mientras que para A2C o Policy Gradient se explora con la misma política del agente.

Finalmente, es necesario aclarar que los ritmos de descenso del gradiente han sido ajustados para cada uno de los algoritmos, debido a que algunos son más sensibles al ruido y es necesario entrenarlos más despacio.

Tal y como se ha mencionado, todos los algoritmos que se presentan en estos ensayos han sido implementados, ajustados y testeados en los entornos de OpenAI Gym hasta demostrar rendimientos similares en entornos conocidos. En documentación de la librería OpenAI Gym está documentado tanto el rendimiento a partir del cual un entorno se considera resuelto como el número de episodios que diferentes algoritmos tardan en conseguir dicho rendimiento. Dado que es posible verificar esta información, hemos ajustado los algoritmos de tal forma que se ha podido resolver y verificar en primer lugar que se han implementado de forma correcta. Ha sido una tarea de gran utilidad dado que trabajar con imágenes hace que el entrenamiento sea más sensible a errores por el hecho de que el espacio de estados crece considerablemente. Los procedimientos de entrenamiento de cada uno de los algoritmos varían dependiendo de si son *off u on policy*. Con el objetivo de hacer comparaciones justas entre el rendimiento de todos los algoritmos y pese a que todos tienen peculiaridades que hacen que no sea posible evaluarlos con la misma red, vamos a usar una red con un componente inicial común, que estará formado por unas capas de convolución sobre la imagen que representa el estado del algoritmo para extraer información espacial. Después de aplanar los pesos de la última convolución se introducen una serie de capas densas que permiten clasificar y extraer los valores necesarios de la información. Es una arquitectura utilizada habitualmente para interpretar información a partir de una imagen [18][19]

Algoritmo	**Ratio aprendizaje** (OpenAI)	**Ratio aprendizaje** (Entornos Propios)	**Tasa decaimiento** exploración/explotación
DQN_N	2e-4	1e-5	0.99999
DQN	2e-4	1e-5	0.99999
A2C	2e-5 (crítico), 0.5e-5(actor)	1e-5 (crítico), 0.2e-5(actor)	-
REINFORCE	2e-6	1e-6	-

Cuadro 1: Comparativa de parámetros elegidos en cada uno de los algoritmos.

En la tabla podemos apreciar los parámetros más relevantes que se han usado para el entrenamiento de los algoritmos en los resultados que se presentarán a continuación. Estos parámetros se han fijado teniendo en cuenta tasas de aprendizaje de problemas similares y a través de la experimentación. Observando las oscilaciones en la función de

pérdida que se obtiene tras el entrenamiento, se puede apreciar si el ratio de aprendizaje está bien ajustado. Es un proceso que requiere varios intentos hasta encontrar parámetros bien ajustados.

3.9. Algoritmos tradicionales

En la etapa final del proyecto se quiere comparar los algoritmos de aprendizaje por refuerzo frente a otros más convencionales. Ya hemos comentado en el apartado 3.2.5 que mediante la transformación al algoritmo del viajante podemos obtener la solución ideal del problema. También ha quedado claro que en el momento que el estado no es completamente observable, es decir, hay que explorar, no es tan sencillo usar este tipo de técnicas. Esto es lo que nos ha motivado al desarrollo de algoritmos adicionales. Los algoritmos que proponemos se han desarrollado pensando en cubrir todos los casos posibles para una resolución clásica del problema. Dos de los algoritmos disponen de observabilidad parcial y los otros dos tienen observabilidad total. Excepto el algoritmo TSP, son todos muy sencillos y están compuestos de reglas muy básicas. Esto nos ha permitido ofrecer al lector comparativas objetivas sin desviar mucho el desarrollo del aprendizaje por refuerzo. A continuación presentamos los algoritmos:

- El algoritmo puramente greedy, con un espacio de estados totalmente observable cuya política es simplemente ir a la casilla con objetivo más cercana hasta terminar el juego.
- El algoritmo ideal con espacio de estados perfectamente observable. Esta solución se basa en la transformación a el algoritmo del viajante representado en grafos tal y como hemos comentado en el apartado 3.2.5.
- Un algoritmo también greedy pero con el mismo espacio de estados que el problema que estamos intentando resolver. La política de este algoritmo es ir a la casilla de valor más cercana si existe alguna observable. En caso de que ninguna casilla de valor sea observable la política del agente será ir a casilla no explorada más cercana.
- Agente que toma acciones completamente aleatorias.

El lector podrá observar que en el caso de los dos primeros algoritmos que se proponen no se trata de una comparativa justa, ya que estos algoritmos disponen de una observabilidad perfecta sobre el entorno, mientras que tanto el algoritmo basado en DQN como el tercero de los algoritmos solamente pueden observar la parte explorada del entorno. De todas formas este es un ejercicio interesante especialmente en el caso del TSP, dado que nos permite conocer la solución ideal y poner en valor la eficacia de los otros algoritmos en términos globales.

4. Resultados

4.1. Evaluación de rendimiento de las variantes Prioritized Experience Replay

En primer lugar vamos a mostrar los resultados obtenidos respecto a las optimizaciones que se producen como consecuencia de utilizar el Prioritized Experience Replay. Los elementos que se han testeado son tanto el borrado en forma de buffer como el borrado de datos en forma de PER inverso y finalmente un buffer continuo con muestreo uniforme. El tamaño del buffer ha sido para todos los casos de 2^{17} experiencias. Se han hecho experimentos con diferentes valores de probabilidad constante ϵ, con valores que oscilan entre 0.1 y 100.

Estos son los resultados obtenidos en este ensayo:

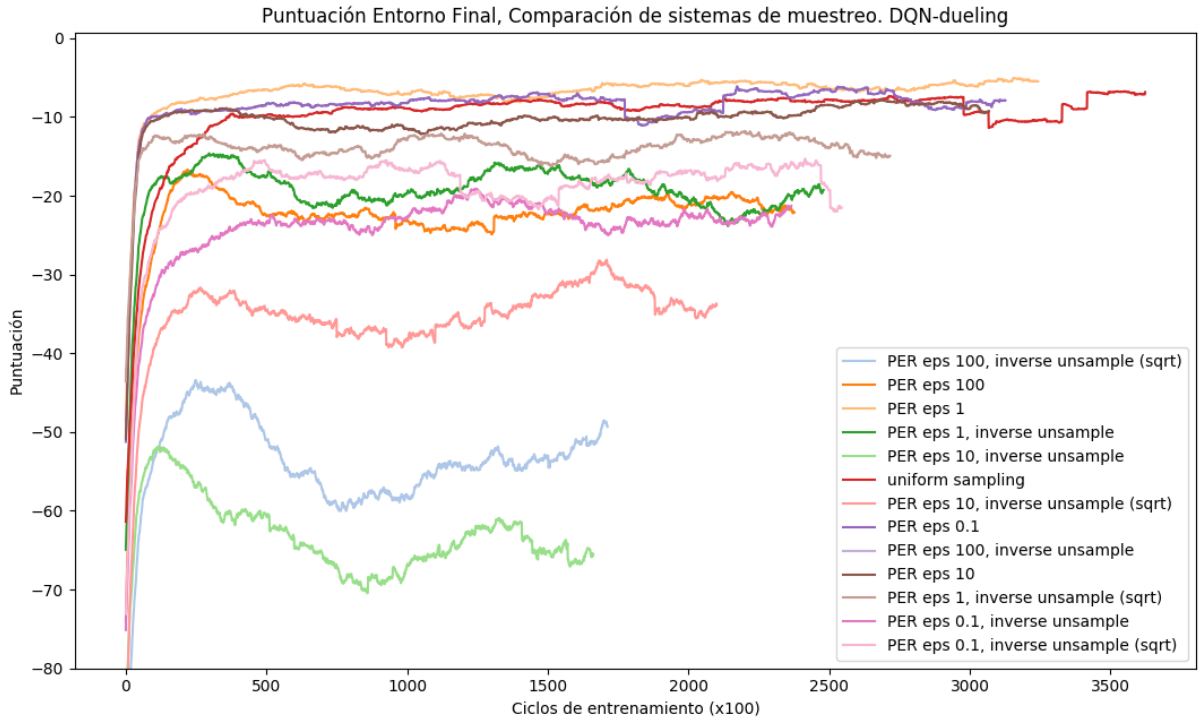


Fig. 15: Resultados obtenidos en DQN con todas las variantes del algoritmo de muestreo.

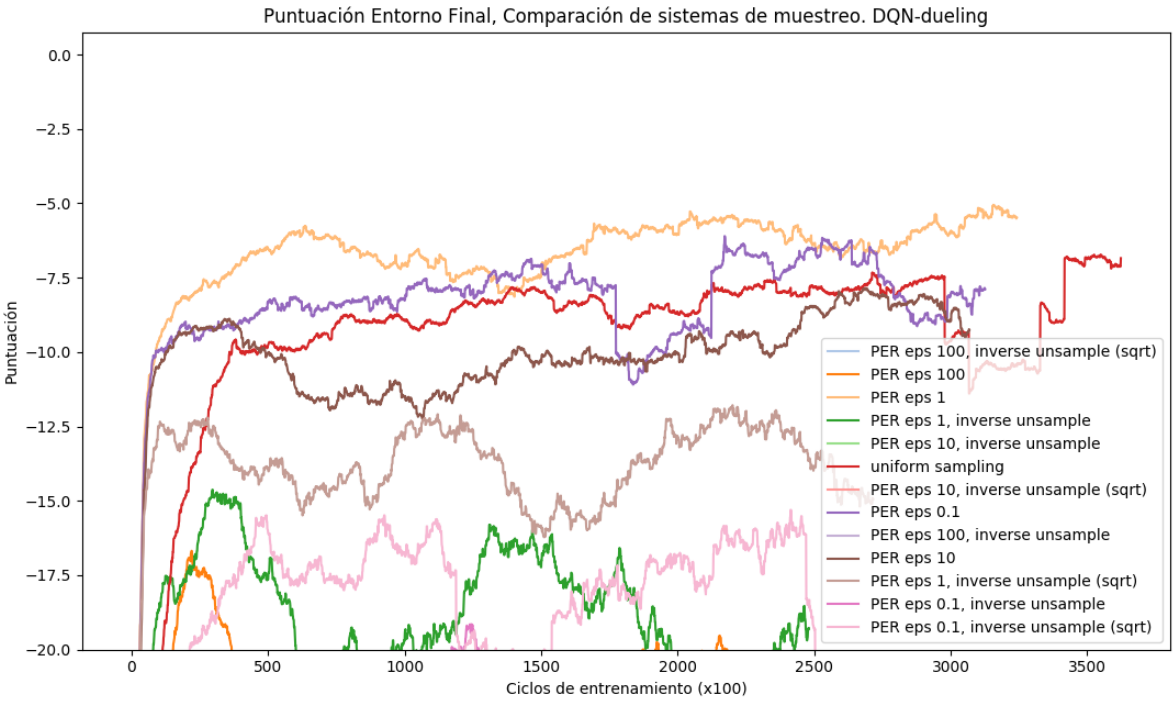


Fig. 16: Resultados obtenidos en DQN con todas las variantes del algoritmo de muestreo, sección de puntuación máxima.

Podemos ver una mejora leve en el progreso del entrenamiento empleando la técnica de Prioritized Experience Replay. En prácticamente todos los casos usar PER garantiza un aprendizaje más rápido y con un rendimiento mayor si se ajustan bien los hiperparámetros. Sin embargo los algoritmos de eliminación de experiencias a través del árbol inverso no han ofrecido ninguna ventaja con respecto a el buffer circular estándar.

En las figuras 15 y 16 podemos ver la puntuación media conseguida en el entorno preliminar. Todas las redes tienen la misma arquitectura y se han entrenado durante el mismo tiempo. Las conclusiones que podemos sacar de este estudio es que PER ayuda a aprender más rápido en las primeras etapas del entrenamiento, pero después es necesario ajustar de forma fina los parámetros de muestreo para conseguir un rendimiento superior al del muestreo uniforme.

Todos los algoritmos basados en DQN utilizarán el PER con $\epsilon = 1$ en el resto del proyecto por haber demostrado un rendimiento superior.

En las figuras podemos ver el rendimiento suavizado de todos los algoritmos corridos, donde los marcados como *unsample* pertenecen a las variaciones de sustitución por importancia, el marcado como *uniform* consiste en un muestreo aleatorio. Los marcados como *sqrt* suavizan el desmuestreo con la siguiente fórmula, siguiendo el criterio de la fórmula 20.

4.2. Evaluación de algoritmos, entorno Snake

El entorno Snake es uno de los dos entornos preliminares que se ha usado para la evaluación de este problema y se han implementado 5 algoritmos diferentes para ello.

Se han empleado dos métricas para valorar los resultados en este apartado. La primera es la puntuación acumulada a lo largo de un episodio siguiendo los criterios que se marcaron en el apartado 3.2.3, una puntuación más alta indica un rendimiento superior. La segunda métrica que vamos a utilizar es el porcentaje de compleción de los episodios. Esta métrica se usará dado que se considera que la compleción de este entorno es significativa en sí. Esta métrica se debe interpretar como el porcentaje de episodios que se han resuelto completamente entre los últimos 100 simulados. Por ejemplo, un porcentaje de resolución de 35 % se debe interpretar como en que los últimos 100 episodios se han resuelto completamente 35. En ambas métricas, un valor superior indica un rendimiento superior.

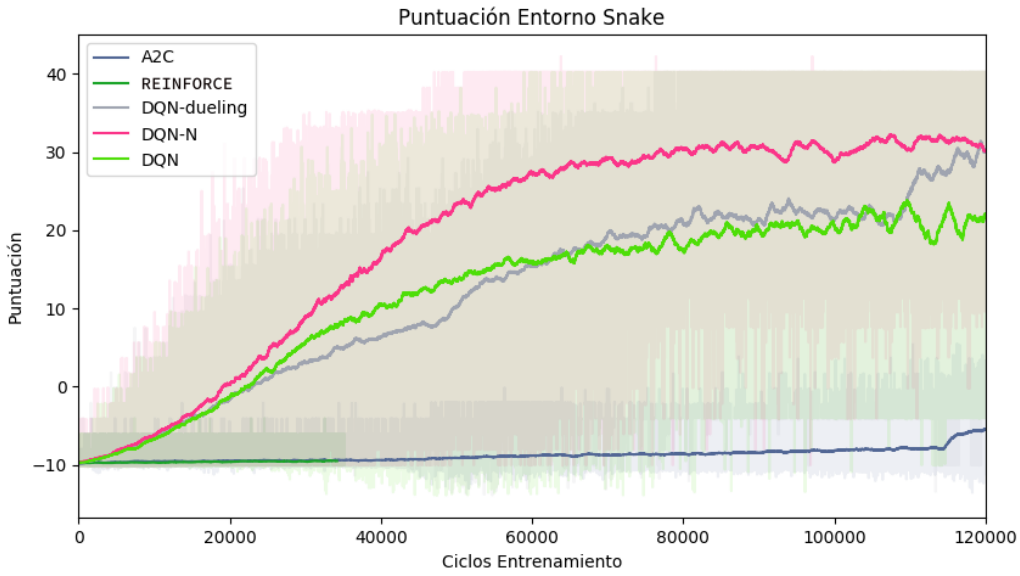


Fig. 17: Resultados de la recompensa obtenida por los diferentes algoritmos en el entorno Snake.

En la figura 17 podemos observar la recompensa obtenida en el entrenamiento en entorno Snake durante 100000 ciclos de entrenamiento y 1500000 episodios. En cada gráfica la envolvente son los valores reales obtenidos mientras que las gráficas sólidas son una media corrida de los últimos 100 elementos de la misma información.

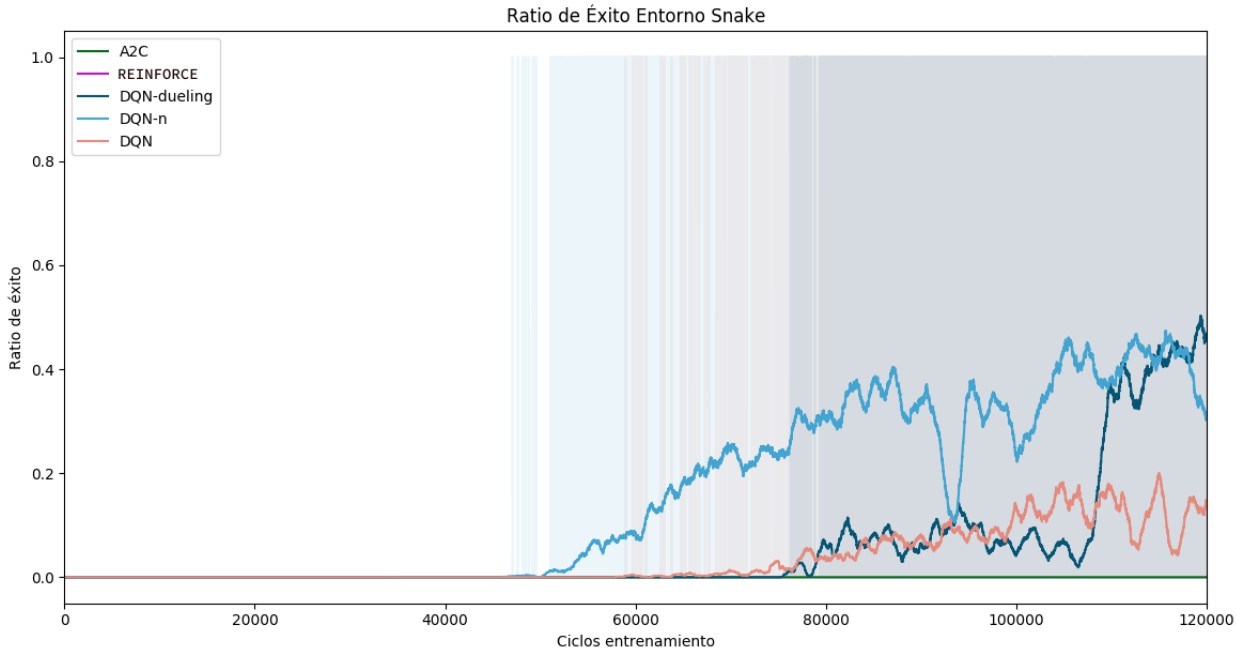


Fig. 18: Porcentaje de éxito de los diferentes algoritmos en el entorno Snake.

La gráfica 18 muestra el porcentaje de episodios en los que se completó el juego, es decir, el agente obtuvo todas las recompensas posibles, 24, sin ninguna colisión y que por lo tanto ya no puede seguir con el *juego* ya que el cuerpo de la serpiente no puede seguir creciendo.

En los algoritmos basados en DQN se ha medido la puntuación mientras se estaba aplicando la política de exploración, es decir el algoritmo ϵ-greedy aplica durante las primeras iteraciones. Esto se debe tener en cuenta a la hora de la visualización de los resultados dado que las primeras iteraciones del entrenamiento son principalmente el producto de la toma aleatoria de decisiones. En los algoritmos que actúan mediante probabilidades como REINFORCE o A2C no es necesario porque la exploración es automática y por lo tanto esto no ocurre. Pese a esto, las curvas de aprendizaje de los algoritmo DQN tienen un crecimiento más rápido.

Los resultados son muy favorables a todos los algoritmos DQN. El algoritmo REINFORCE no llega a converger de una forma significativa, mientras que A2C llega a puntos de compensa relativamente alta, pero no consigue mejorar excesivamente. Por otro lado todas las variantes de DQN que se han incluido en este ensayo consiguen la resolución completa del entorno al menos en algún episodio y según avanza el entrenamiento podemos apreciar cómo todos los algoritmos consiguen al menos un porcentaje significativo de éxito llegando hasta el 50 % de resolución.

La figura 18 muestra una conclusión muy interesante de los resultados del entrenamiento. El porcentaje de éxito ya es significativo. Según el agente va acumulando recompensas,

el cuerpo de la serpiente se hace más largo y es más complicado maniobrar. Si no se tienen en cuenta estos factores estratégicos, conseguir recompensas altas es muy difícil.

Si este comportamiento esta produciéndose por parte del agente, es algo que no se puede visualizar en las figuras 17 y 18 en las que solamente se muestran los resultados. Por esto procedemos a mostrar algunos ejemplos interesantes de comportamientos *inteligentes* que se pueden apreciar tras la fase de entrenamiento con el algoritmo Average-DQN.

En primer lugar, uno de los puntos más interesantes es la gratificación retardada de la que hemos hablado en apartados anteriores. El agente evita tomar una decisión con gratificación inmediata para tomar una que no ofrece la recompensa de forma inmediata pero que de forma global permite un rendimiento mayor.

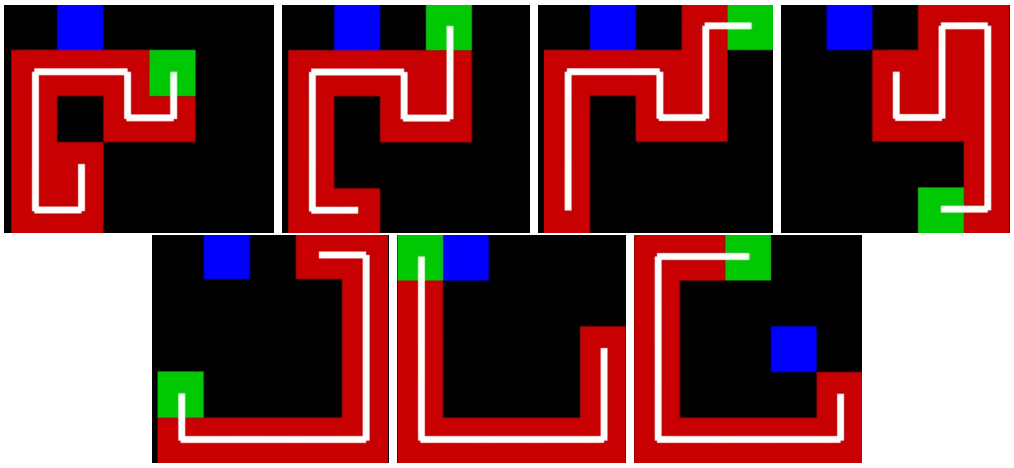


Fig. 19: Secuencia de desarrollo, ejemplo 1

En la figura 19, de izquierda a derecha se puede observar una secuencia en la que se aprecia el comportamiento antes mencionado. La cabeza de la serpiente se visualiza en verde y el cuerpo en rojo. La línea blanca nos ayuda a visualizar con más facilidad el cuerpo de la serpiente, finalmente las recompensas están representadas en azul.

En la primera y segunda imagen podemos ver que la recompensa está solamente a dos movimientos pero el agente decide no capturarla, debido a que supondría quedarse atascado 2 movimientos después. En el resto de las figuras de la secuencia podemos ver cómo el agente da una vuelta completa por el tablero hasta posicionar el cuerpo de tal forma que capturar la recompensa que antes había evitado no resulte en una colisión.

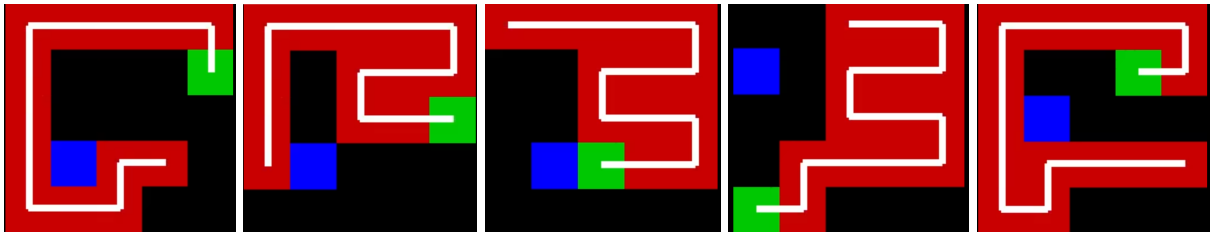


Fig. 20: Secuencia de desarrollo, ejemplo 2

En la figura 20, de nuevo podemos ver que el agente posiciona el cuerpo en las dos primeras imágenes de esta serie, rechazando la recompensa que está a un cuadrado de distancia para posponer su captura 4 movimientos. En la tercera imagen de la figura, podemos ver como captura dicha recompensa. Después podemos ver en las imágenes 4 y 5 que posponer la recompensa en esos 4 movimientos permite al agente dar la vuelta sin quedarse atascado, un conjunto de movimientos que tienen en cuenta factores que ocurren mucho más adelante.

4.3. Evaluación de algoritmos, entorno preliminar.

En el entorno preliminar de exploración, los resultados presentan un comportamiento similar pero con ciertos matices. En primer lugar, el rendimiento de todos los algoritmos en general es menor, o por lo menos así se percibe.

La métrica que se ha empleado para valorar los resultados de este entorno es también la puntuación acumulada a lo largo de un episodio siguiendo siguiendo los criterios expuestos en el apartado 3.2.1. Dado que la única forma de obtener puntuaciones positivas es obtener objetivos, y como hasta que no se completen todos los objetivos no terminará el episodio, una puntuación más alta indica un rendimiento superior.

Los entornos DQN son más propensos a crear *lazos* cerrados (siguiendo el criterio de transición con valor Q máximo) no óptimos y por lo tanto el agente se queda atascado en el ciclo hasta que el entorno termina de forma forzosa por límite de pasos. Esto es un comportamiento muy poco deseado, ya que llena la cola de experiencias de valores repetidos, que además ocurre en puntos avanzados del entrenamiento (si ocurriera en la primera fase se saldría del ciclo mediante una decisión aleatoria tomada en la exploración). Como ya hemos expuesto antes, esto ocurre por dos motivos, existe un número elevado de soluciones válidas para cada una de las recompensas que el agente tiene que explorar y un entrenamiento insuficiente puede producir que ocurran estas situaciones.

La diferencia de rendimiento entre A2C y la familia de algoritmos DQN no está tan acentuada en este entorno tal y como podemos ver en las figuras 21 y 22. Esta diferencia se debe a diferentes razones, la primera es que el agente A2C no tiene el problema de los lazos cíclicos, dado que las probabilidades en decisiones inciertas tienden a repartirse de

forma uniforme y los ciclos se rompen de manera diferente. Además, al no ser un entorno donde una mala decisión se castiga con la terminación del juego, la selección de acciones a través de probabilidad es ligeramente más permisiva, en este entorno incluso ventajoso por las diferentes soluciones posibles. De todas formas, el rendimiento de las variantes DQN sigue siendo superior.

Una estadística que quizás no queda reflejada en estas figuras, es la dificultad para conseguir un entrenamiento estable en A2C mediante los algoritmos que se han programado. Los algoritmos DQN son mucho más permisivos en cuanto a los hiperparámetros que se pueden usar. Mientras que, nuestra experiencia al menos, es que los algoritmos basados en Policy Gradient o A2C son mucho más sensibles a hiperparámetros mal ajustados. En la gráfica, en todo caso, solamente se observa el mejor resultado obtenido para cada una de las familias.

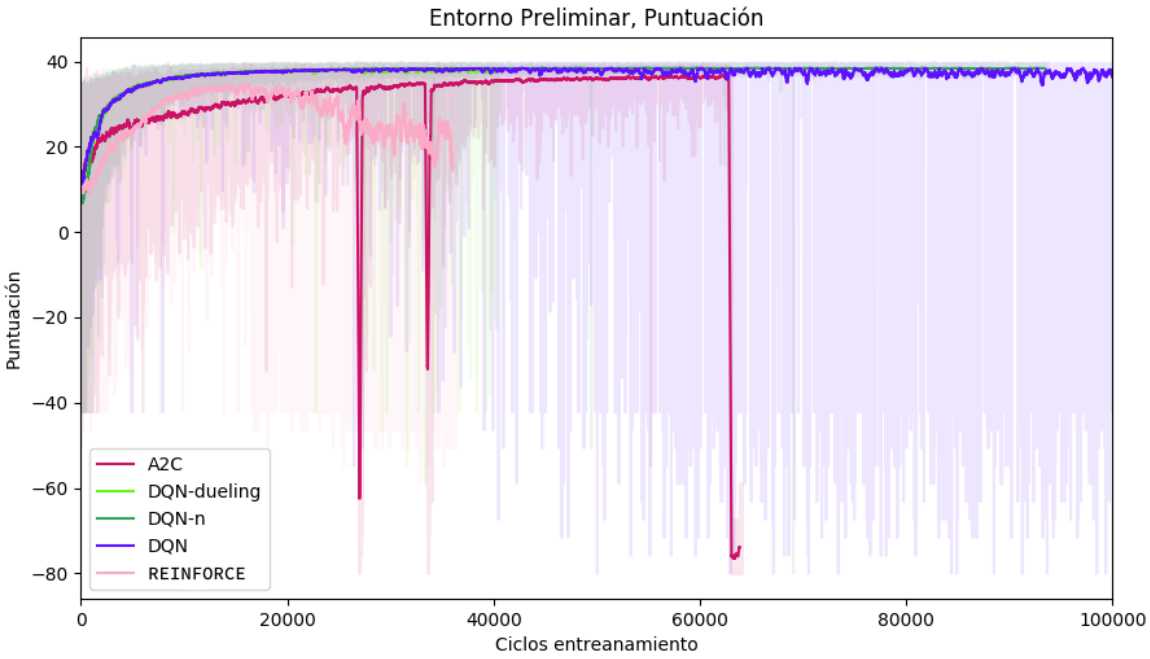


Fig. 21: Comparación de algoritmos por máxima recompensa acumulada en el entorno preliminar

4.4. Evaluación de algoritmos, entorno final.

Con las conclusiones claras sobre el rendimiento de las distintas familias de los algoritmos en los entornos simplificados, procedemos a desarrollar el algoritmo final mediante el cual intentaremos resolver el entorno final. Dado que el rendimiento de los algoritmos basados en DQN fue mucho más alto en las pruebas preliminares, la versión final será una variación de DQN que además use Prioritized Experience Replay para la repetición de experiencias.

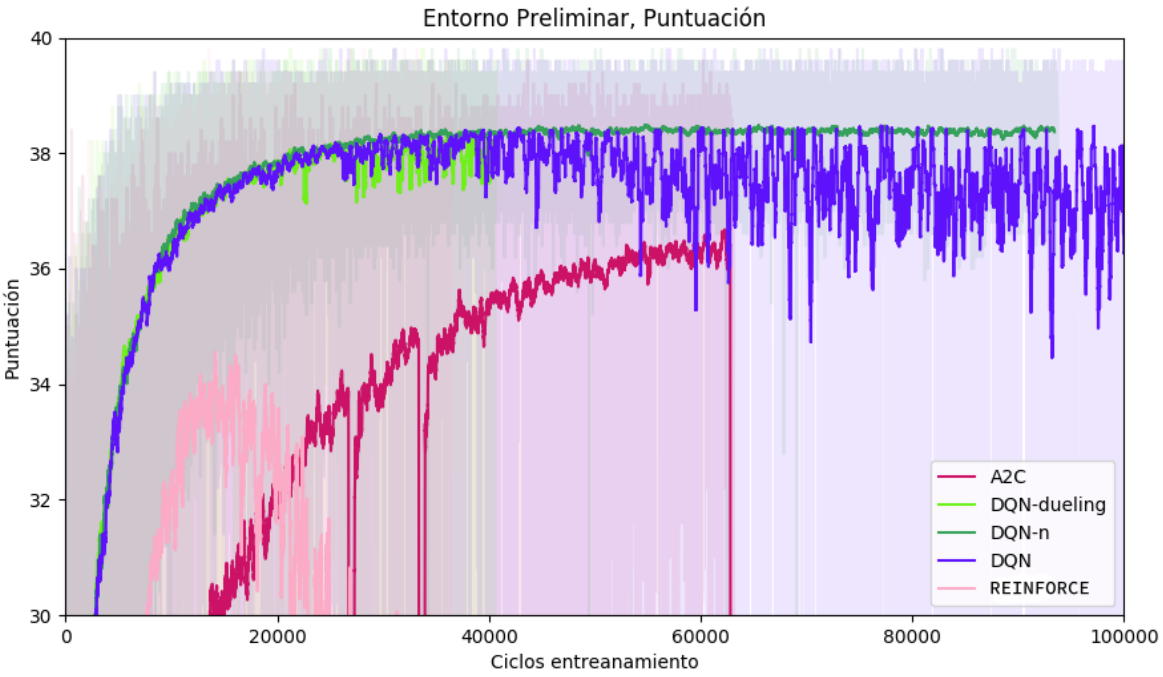


Fig. 22: Comparación de algoritmos por máxima recompensa acumulada en el entorno preliminar, parte alta de la gráfica.

La métrica para la evaluación de estos algoritmos en este caso es diferente. Vamos a utilizar la media de acciones necesarias para resolver completamente el problema. Por lo tanto, un número menor de acciones necesarias para resolver el algoritmo indica un rendimiento superior. Se ha elegido una métrica diferente para esta sección para poder interpretar los algoritmos clásicos con mayor facilidad, porque es la métrica principal que usan. Por otro lado, el algoritmo basado en aprendizaje por refuerzo puede emplear esta métrica también, contando el número de acciones que el agente tiene que emplear para solucionar el entorno.

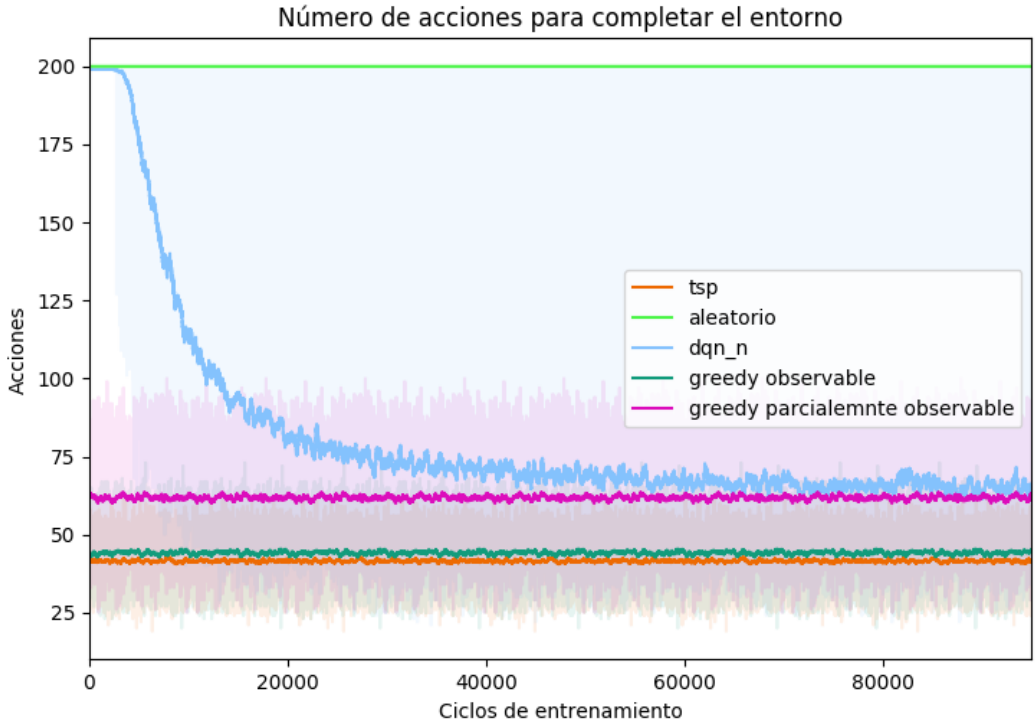


Fig. 23: Comparación de algoritmos por número de acciones en resolver.

La figura 23 muestra los resultados obtenidos por cada uno de los algoritmos en el entorno final. El algoritmo greedy con espacio de estados totalmente observable tiene un rendimiento similar al del algoritmo ideal. El agente que toma acciones aleatorias es prácticamente incapaz de resolver el entorno en ningún caso. En la gráfica podemos ver que todos los resultados son de 200 acciones que se corresponden con el límite de terminación. Los dos algoritmos con espacio de estados parcialmente observables tienen un rendimiento similar, el algoritmo greedy tiene un rendimiento mejor al que se obtiene con el algoritmo de aprendizaje por refuerzo. Las gráficas se han suavizado con una media móvil de 100 unidades para poder visualizar el progreso de forma más sencilla.

El algoritmo que se ha usado para estos resultados finales es una red con la arquitectura descrita en el apartado 3.8, combinado con el algoritmo Dueling DQN y muestreando acciones mediante el algoritmo Average-DQN descrito en el apartado 2.4.3.

A continuación se presenta una tabla con una comparativa de los resultados:

Algoritmo	**Puntuación media**	**% respecto a solución ideal**
Viajante	41.45	100
Greedy	44.13	106.46
Greedy (parcialmente observable)	61.44	148.22
Aprendizaje por refuerzo (parcialmente observable)	64.21	154.90

Cuadro 2: Comparativa de resultados obtenidos en el entorno final frente a la solución ideal con el problema del mercante.

Los resultados del algoritmo de aprendizaje por refuerzo son aproximadamente un 50 % peores a la solución ideal y un 4 % peor comparados con los del algoritmo greedy con estados parcialmente observables.

Esta última comparación, entre el algoritmo Greedy parcialmente observable y el algoritmo de aprendizaje por refuerzo, se puede interpretar como un resultado de rendimiento bajo. El algoritmo de reglas greedy tiene un rendimiento superior al de aprendizaje por refuerzo. En las conclusiones valoraremos e interpretaremos en profundidad el motivo de estos resultados.

5. Conclusiones

Los resultados de este proyecto son amplios y las decisiones que se han tomado a lo largo del mismo son numerosas. El proyecto comenzó de forma ambiciosa como la búsqueda de un reemplazo para un algoritmo de planificación de trayectorias en un proyecto con un robot físico. Y poco a poco se ha ido modificando el objetivo hasta simplificarlo en la búsqueda de un conjunto de problemas más sencillos, sobre los que se puedan aplicar con más facilidad las técnicas disponibles en la familia de algoritmos de aprendizaje por refuerzo:

5.1. Simplificación y sintetización del problema

Se han tomado muchas medidas para simplificar el problema original: la *discretización* del espacio de estados, la discretización del espacio de acciones, la conversión del mapa a un tablero con direcciones cardinales y la eliminación las ecuaciones del modelo del barco que se han reemplazado por acciones directas (por ejemplo: arriba significa arriba, no hay ni inercia, ni timón, ni motor). También se ha reducido considerablemente el espacio de estados, limitando el tablero y acomodándolo a un tamaño razonable con respecto a los entornos conocidos y similares.

Todas estas decisiones se han tomado con la intención de simplificar el problema original ante la imposibilidad de resolver con los algoritmos existentes el problema en su totalidad, pero manteniendo fundamentalmente las propiedades del problema original. De tal forma que, cuando en el futuro se intente portar al drone, sea relativamente sencillo, dado que un algoritmo funcional en el entorno propuesto, supone relativamente el mismo proceso con algunos pasos adicionales que añaden cierta complejidad. En definitiva, se ha intentado eliminar del problema todos los componentes *adicionales* y se ha terminado con un problema lo más similar a los entornos estándares a los utilizados por aprendizaje por refuerzo. Todos estas decisiones se han explicado y justificado a lo largo del proyecto y se han hecho, a la vista del autor, de forma correcta para el alcance del proyecto, pese a que lo alejan en cierta manera el objetivo inicial de este trabajo.

5.2. Resultados obtenidos

El abanico de algoritmos que se ha utilizado para este proyecto y de entornos distintos con los que se han validado los resultados, es grande, y consiste de entornos conocidos como los de OpenAI Gym, entornos preliminares de Snake y de exploración y finalmente el entorno final. En este apartado es donde se pueden apreciar con mayor claridad los puntos fuertes y débiles de este proyecto.

5.2.1. Estado del arte de los algoritmos

Todos los algoritmos se han implementado en primer lugar sobre entornos de OpenAI Gym. Lo que nos ha permitido comprobar si se obtenía el rendimiento esperado sobre cada uno de ellos y *preajustar* sus parámetros para que el aprendizaje fuese eficiente frente a rendimientos estandarizados, ya que se conoce el rendimiento esperado de dichos entornos. Esto nos ha permitido comenzar con unas implementaciones relativamente funcionales en los entornos que se han desarrollado para el proyecto y solamente ha sido necesario ajustar los algoritmos (en términos de ratio de aprendizaje o cambio de arquitectura en la red para un espacio de estados). Este apartado se menciona en la metodología del proyecto, pero no se ha presentado como resultado, dado que no es el objetivo, pero la conclusión de este punto es muy positiva. Para todos los algoritmos que se han presentado al lector a lo largo de este documento, existe una o varias implementaciones análogas que permiten corroborar los resultados contra entornos conocidos con un rendimiento óptimo, según lo estipulado por otros proyectos y algoritmos similares.

5.2.2. Implementaciones preliminares

Los dos entornos simplificados tienen varias notas positivas en las conclusiones a obtener, especialmente en el entorno Snake. El entorno Snake quizás sea uno de los grandes éxitos de este trabajo, pese a que no era el objetivo principal de éste. El rendimiento obtenido ha sido excepcional, llegando a obtener un algoritmo capaz de resolver el entorno en su totalidad con una tasa de éxito del 50 %, y como ya hemos visto, demostrando comportamientos realmente avanzados (por ejemplo evitar capturar recompensas que podrían suponer una pérdida en el largo plazo y merodear hasta encontrar la posición óptima para capturarlas sin dicha pérdida). Estos resultados son especialmente positivos si se tienen en cuenta en comparación con el resto. Atribuimos este buen comportamiento a varios motivos:

En primer lugar, las potenciales acciones que se pueden realizar respecto a este problema son limitadas y aunque al principio el entorno Snake y el entorno de exploración son muy similares, en el momento en que el cuerpo de la serpiente crece de tamaño, las decisiones que debe tomar el algoritmo en el entorno Snake se reducen considerablemente. Esto se debe a que la mayor parte de éstas terminan en una colisión. Esto permite que una vez la exploración termina, no se pierda mucho tiempo en estudiar soluciones *múltiples*. Además, la aparición de ciclos locales no óptimos es menos probable. Cuando la exploración se reduce, la probabilidad de que el algoritmo considere que la secuencia $A \rightarrow B \rightarrow C \rightarrow A$ sea la más óptima es prácticamente nula, dado que por la forma del cuerpo de la serpiente debería ser un ciclo extremadamente largo. Adicionalmente, si la secuencia no fuera óptima, seguramente acabaría con la terminación forzada de la experiencia (a través de una colisión).

Finalmente, cabe decir, que las comparaciones que se han hecho son más justas y similares a las que típicamente se utilizan con este tipo de algoritmos. La comparación se hace **frente a la expectativa de rendimiento humano**. El problema del Snake, especialmente con un tablero tan pequeño como el que hemos usado, es extremadamente sencillo y se pueden conseguir algoritmos capaces de encontrar una solución no óptima de forma trivial y con una fiabilidad del 100 % (véase por ejemplo, la construcción de un camino cerrado, es decir, un camino Hamiltoniano por el tablero, por el que la serpiente simplemente tiene que desplazarse y conseguir todas las recompensas). Por la dificultad relativa que supone el desarrollo de un algoritmo de esta forma, las expectativas están *correctamente* rebajadas y el resultado es una comparación más real con respecto al estado del arte.

5.2.3. Implementaciones finales

Los resultados sobre el entorno final son significativamente diferentes a los preliminares. Este segmento del proyecto es, con diferencia, al que más atención se ha prestado, en cuanto a implementación, horas de desarrollo, algoritmos testeados e iteraciones de desarrollo. Desde una etapa bastante preliminar del proyecto, se ha conocido la solución óptima del problema a través de la transformación al problema del viajante comentada en los apartados iniciales. Esto cambia el marco de referencia de las comparaciones, al comparar en todo momento el rendimiento del algoritmo a las expectativas de la solución ideal. De la misma forma, conocemos que la complejidad algorítmica de la solución ideal es $O(N!)$, es decir, que computacionalmente es un problema complejo, más todavía si lo intentamos resolver con algoritmos de aprendizaje por refuerzo con complejidad constante $O(1)$ o lineal con respecto al número máximo de pasos $O(M)$. Podríamos decir que la resolución ideal a través de TSP del problema requiere una complejidad computacional factorial. Mientras que al ser resuelta por aprendizaje por refuerzo es mucho menor, es una heurística que nos permite reducir el tiempo computacional con una inevitable caída en el rendimiento. Estas descripciones son solamente válidas para un espacio de estados completamente observable, algo que no es así en el entorno final, añadiendo un grado más de dificultad en la lectura de los resultados.

Finalmente, el algoritmo *puramente avaro* que se presenta en las comparaciones, también tiene un rendimiento superior. Esto no es aceptable dado que es un algoritmo extremadamente sencillo y tiene el mismo espacio de estados (parcialmente visible). Estos resultados son inequívocamente insatisfactorios, pero la razón de esto es comprensible:

En primer lugar, el algoritmo de aprendizaje por refuerzo no conoce el entorno al principio del entrenamiento, no conoce lo que hacen las acciones, las cuales representan los valores de los estados, ni conoce el objetivo, porque es un algoritmo libre de modelo. Una parte del periodo de entrenamiento se pierde en entender estas reglas. De la misma forma

ocurre en el entorno Snake. La diferencia es que sintetizar el modelo en reglas en caso de los entornos del explorador es **extremadamente sencillo**, el algoritmo *avaro* se compone de 20 líneas de código, esto es algo que no sería posible en el entorno Snake. Por otro lado, existen sutilezas en cómo se consiguen las recompensas en el entorno Snake con respecto al de exploración. El entorno Snake, una vez ya está en un estado avanzado, tiene un abanico de posibilidades relativamente pequeño, el resto resultan en la terminación del episodio por colisión. En el caso del entorno de explorador, es justamente al contrario, para cada una de las recompensas existe un abanico muy amplio de trayectorias, que son igualmente óptimas individualmente. Aún así, la diferencia entre un algoritmo con buen rendimiento y otro que no lo tiene, no es la capacidad de circular por estas trayectorias individualmente óptimas, si no que consiste en el orden en el que se capturan las recompensas a lo largo del episodio (véase el problema del viajante). Esto hace que en primer lugar se creen muchos ciclos no óptimos ($A \rightarrow B \rightarrow C \rightarrow A$), se pierda mucha capacidad de aprendizaje en entender las diferentes trayectorias óptimas.

En definitiva, tras el proceso de transformación del problema, hemos terminado con un problema que es *demasiado sencillo* de resolver con técnicas tradicionales y sigue siendo complicado para los algoritmos de aprendizaje por refuerzo. Nosotros argumentamos en todo caso, que no ha sido en la transformación donde se ha producido este fenómeno, es algo intrínseco al problema original y el objetivo de este proyecto ha sido comprobarlo.

Pese a todos estos comentarios, no se debe quitar peso a los resultados obtenidos: un rendimiento cercano al óptimo y similar o mejor al que obtendría una persona. Pongámoslo de la siguiente manera, muchos de los problemas de OpenAI Gym nos presenta para resolver, especialmente en la sección de control (por ejemplo CartPole o LunarLander) son problemas típicos de control, con un espacio de estados reducido ($X = x, y, \dot{x}, \dot{y}, \theta, \dot{\theta}$) que con un controlador PID o incluso un MPC quedarían resueltos de forma mucho más elegante y con una puntuación mucho más alta, pero no se efectúa de esta forma ya que estos entornos están diseñados para explorar algoritmos de aprendizaje por refuerzo. *Y este también es el caso que aplica para este proyecto.* Es por esto que tomamos los resultados como válidos y aceptables como conclusión de este trabajo.

5.2.4. Aplicabilidad de un algoritmo de aprendizaje por refuerzo en un entorno de exploración

No es definitivo y todavía queda margen para mejorar, pero las conclusiones que se obtienen de este proyecto es que usar un algoritmo de aprendizaje por refuerzo para el componente de cálculo de trayectorias, en un drone acuático, no es la forma más eficiente de conseguir resultados. Esto no significa que no exista posibilidad de mejora o que el aprendizaje por refuerzo no pueda ser empleado en un drone acuático

5.3. Acciones futuras y continuidad del proyecto

Aunque los resultados no ofrezcan el rendimiento esperado no se deben dar todas las posibilidades por perdidas en esta línea de investigación. En primer lugar, el campo del aprendizaje por refuerzo con redes neuronales está viviendo una expansión en este último lustro y la tendencia continuará al menos durante un tiempo. Será necesario revisitar el problema para ver los nuevos algoritmos y técnicas que se puedan ir presentando. Incluso dentro de las técnicas que ya están disponibles, se puede continuar con la investigación, una posibilidad es la implementación del algoritmo con redes con componentes LSTM (Long Short Term Memory) que permitan analizar trayectorias como series temporales. Este es solamente una entre las muchas técnicas posibles para implementar en este problema.

Dentro del proyecto, también debería ser posible el uso de redes neuronales que utilicen aprendizaje por refuerzo en otros componentes de *más bajo nivel*, como por ejemplo, el control de la trayectoria planificada por la planificación de trayectorias, la lectura e interpretación de sensores o el control de sistemas en general. Aunque esta línea de investigación trasladaría la dificultad del problema a otros campos como el modelado del sistema y se debería seguir utilizando algoritmos clásicos para la planificación de trayectorias.

En definitiva, las conclusiones que se han sacado y las técnicas investigadas, aunque no ideales, son prometedoras y tienen un gran potencial para investigaciones futuros.

Referencias

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, pp. 529–533, Feb. 2015.

[2] H. van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," *CoRR*, vol. abs/1509.06461, 2015.

[3] Z. Wang, T. Schaul, M. Hessel, H. van Hasselt, M. Lanctot, and N. de Freitas, "Dueling network architectures for deep reinforcement learning," 2016.

[4] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," 2015. cite arxiv:1511.05952Comment: Published at ICLR 2016.

[5] G. Carazo-Barbero, E. Besada-Portas, J. M. Girón-Sierra, and J. A. López-Orozco, "Ea-based asv trajectory planner for pollution detection in lentic waters," *Universidad Complutense de Madrid*, 2020.

[6] E. Besada-Portas, J. M. Girón-Sierra, J. Jiménez, and J. A. López-Orozco, "Data-driven exploration of lentic water bodies with asvs guided by gradient-free optimization/contour detection algorithms," *Proceedings of the 2021 Winter Simulation Conference*, 2021.

[7] S. Y. Luis, D. G. Reina, and S. L. T. Marín, "A multiagent deep reinforcement learning approach for path planning in autonomous surface vehicles: The ypacaraí lake patrolling case," *IEEE Access*, vol. 9, pp. 17084–17099, 2021.

[8] S. Y. Luis, D. G. Reina, and S. L. T. Marín, "A deep reinforcement learning approach for the patrolling problem of water resources through autonomous surface vehicles: The ypacarai lake case," *IEEE Access*, vol. 8, pp. 204076–204093, 2020.

[9] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[10] R. Bellman, *Dynamic Programming*. Dover Publications, 1957.

[11] C. J. C. H. Watkins and P. Dayan, "Q-learning," *Machine Learning*, vol. 8, pp. 279–292, May 1992.

[12] H. Hasselt, "Double q-learning," in *Advances in Neural Information Processing Systems* (J. Lafferty, C. Williams, J. Shawe-Taylor, R. Zemel, and A. Culotta, eds.), vol. 23, Curran Associates, Inc., 2010.

[13] oron anschel, nir baram, and nahum shimkin, "averaged-dqn: variance reduction and stabilization for deep reinforcement learning," 2017.

[14] R. S. Sutton, D. Mcallester, S. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *In Advances in Neural Information Processing Systems 12*, pp. 1057–1063, MIT Press, 2000.

[15] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," *CoRR*, vol. abs/1602.01783, 2016.

[16] A. Nair, P. Srinivasan, S. Blackwell, C. Alcicek, R. Fearon, A. D. Maria, V. Panneershelvam, M. Suleyman, C. Beattie, S. Petersen, S. Legg, V. Mnih, K. Kavukcuoglu, and D. Silver, "Massively parallel methods for deep reinforcement learning," *CoRR*, vol. abs/1507.04296, 2015.

[17] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016. cite arxiv:1606.01540.

[18] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *International Conference on Learning Representations*, 2015.

[19] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015.

6. Anexo

Todo el código relevante a este proyecto se encuentra publicado en el siguiente repositorio: `http://github.com/lasdasdas/tfm-rl`.