Aprendizaje por refuerzo profundo aplicado a juegos sencillos

Trabajo de Fin de Grado
Curso 2018–2019

Autores
Ricardo Arranz Janeiro
Lidia Concepción Echeverría
Juan Ramón Del Caño Vega
Francisco Ponce Belmonte
Juan Luis Romero Sánchez

Director
Antonio A. Sánchez Ruiz-Granados

Grado en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid

Aprendizaje por refuerzo profundo aplicado a juegos sencillos

Trabajo de Fin de Grado en Ingeniería Informática
Departamento de Ingeniería de Software e Inteligencia Artificial

Autores
Ricardo Arranz Janeiro
Lidia Concepción Echeverría
Juan Ramón Del Caño Vega
Francisco Ponce Belmonte
Juan Luis Romero Sánchez

Director
Antonio A. Sánchez Ruiz-Granados

Convocatoria: *Junio 2019*
Calificación:

Grado en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid

27 de mayo de 2019

Dedicatoria

A todos los profesores y compañeros de la facultad, por ayudarnos a aprender.

Agradecimientos

A Antonio, por guiarnos durante tantos meses y no salir de nuestra cabeza durante la realización de este TFG.

A todas nuestras familias y amigos, por darnos los ánimos para no desistir en momentos que las redes no conseguían aprender.

Y sobre todo, a Freire, por su silla.

Resumen

En este proyecto estudiaremos el campo del aprendizaje por refuerzo profundo, con el objetivo de lograr una aplicación estable en problemas clásicos de control. Para lograrlo investigaremos sus bases: el aprendizaje por refuerzo y las redes neuronales, comprobando cuáles son sus puntos fuertes y débiles. Después combinaremos lo aprendido para, progresivamente, mejorar el rendimiento y la estabilidad de nuestros agentes.

En busca de una mayor comprensión de su funcionamiento, todas las implementaciones de los agentes y algoritmos serán hechas por nosotros mismos. Todo ello será puesto a prueba a través del conocido sistema OpenAI Gym.

Todo el código fuente referente a este proyecto puede encontrarse en:

`https://github.com/delcanovega/TFG-DRL`

Palabras clave

- Aprendizaje por refuerzo
- Q-Learning
- Proceso de Márkov
- Redes neuronales
- Aprendizaje por refuerzo profundo
- DeepMind
- OpenAI

Abstract

In this project we will study the Deep Reinforcement Learning field in order to achieve an stable application for classic control problems. To do this we will investigate its fundamentals: Reinforcement Learning and Neural Networks, learning which are their strengths and weaknesses. Finally, we will merge both to progressivly improve our agent's performance and stability.

In order to gain a better insight we will personally implement the agents and algorithms. All of this will be tested through the popular framework OpenAI Gym.

This project's source code can be found in the repository:

`https://github.com/delcanovega/TFG-DRL`

Keywords

- Reinforcement Learning
- Q-Learning
- Markov decision process
- Neural Networks
- Deep Reinforcement Learning
- DeepMind
- OpenAI

Índice

Índice de figuras

Índice de tablas

CAPÍTULO 1

Introducción

"Los jóvenes deben enseñarse a sí mismos, entrenarse a sí mismos, con infinita paciencia, intentarlo una y otra y otra vez hasta que salga bien"
— William Faulkner

1.1. Motivación

La **inteligencia artificial** es una de las ramas de la computación que más interés ha generado, tanto entre expertos de la materia como en otro tipo de público, más interesado en la parte lúdica de este concepto. Dotar a una máquina de la capacidad de realizar funciones asociadas sólo al intelecto humano ha sido siempre considerado ciencia ficción. Y sin embargo esta misma idea ha supuesto un enorme avance tecnológico en los últimos años.

Para conocer mejor este concepto desde sus orígenes, es necesario hacer referencia al **test de Turing**, propuesto por Alan Turing [39]. Éste consiste en realizar una serie de preguntas a un ente y el test se considerará superado si el interrogador no es capaz de discernir si las respuestas provienen de una máquina o una persona.

Intentar definir la inteligencia artificial nos lleva directamente a conceptos como el *proceso del pensamiento* o el *razonamiento*, los cuales terminan por conducir a otros más complejos como es el *comportamiento*. A partir de estas ideas podemos encontrar otras definiciones, clasificadas en la matriz 1.1.

En base a estas clasificaciones podríamos diferenciar dos corrientes de

Pensar de forma humana	**Pensar de forma racional**
"El nuevo y excitante esfuerzo por hacer que las máquinas piensen… *máquinas con mente*, en un sentido completo y literal." (Haugeland, 1985)	"El estudio de las facultades mentales a través del uso de modelos computacionales" (Charniak and McDermott, 1985)
"[La automatización de] actividades que asociamos al pensamiento humano, actividades como la toma de decisiones, resolución de problemas, aprendizaje…" (Bellman, 1978)	"El estudio de las computaciones que hacen posible percibir, razonar, y actuar" (Winston, 1992)
Actuar de forma humana	**Actuar de forma racional**
"El arte de crear máquinas que realizan funciones que requieren inteligencia cuando son realizadas por personas" (Kurzweil, 1990)	"Inteligencia Computacional es el estudio del diseño de agentes inteligentes" (Poole *et al.*, 1998)
"El estudio de cómo hacer que máquinas realicen acciones en las que, de momento, las personas son mejores" (Rich and Knight, 1991)	"IA… se preocupa del comportamiento inteligente en artefactos" (Nilsson, 1998)

Figura 1.1: Definiciones de inteligencia artificial, Russell y Norvig [31]

interpretación:

- Una visión empírica (columna izquierda) con el ser humano como centro de la investigación. Involucra principalmente observaciones e hipótesis sobre cómo debería comportarse un humano. Esta vertiente tiene cierta relevancia a día de hoy, sobre todo gracias al auge de proyectos como el coche autónomo. Estos agentes autónomos, puestos en una situación límite, podrían verse obligados a decidir entre dos opciones que pongan en peligro vidas humanas. En este caso, deberíamos basar nuestra respuesta en qué elegiría un conductor real.
- Una visión racionalista (columna derecha), que implica una combinación de matemáticas e ingeniería. En esta vertiente se engloban proyectos como los asistentes de voz o los robots de Boston Dynamics. Ninguno de ellos necesita valorar las órdenes e información que se les provee a un nivel "humano". Los asistentes no necesitan ser asertivos, sólo necesitan ser capaces de procesar la información de forma correcta para actuar consecuentemente, mientras que los robots sólo deben

> ser capaces de aprender a adaptarse a cualquier terreno con el fin de realizar la función que se les ha encomendado.

Multitud de expertos han abordado ambos acercamientos de distintas formas. Nosotros buscaremos que nuestro *agente* tome las "decisiones correctas" en función del conocimiento que posea. En particular buscaremos que un agente sin ningún conocimiento previo sea capaz de aprender a realizar tareas sencillas mediante la interacción constante con el entorno, quien le proporcionará nuevas experiencias de las que extraer conocimiento.

Esta forma de aprendizaje es un campo de la inteligencia artificial llamado **aprendizaje por refuerzo**. También investigaremos una combinación del mismo con **redes neuronales**, resultando en el llamado **aprendizaje por refuerzo profundo**. La popularidad de estas técnicas no ha parado de crecer en los últimos años. Koray Kavukcuoglu, director de investigación en Deepmind, explica su potencial de la siguiente forma:

> El aprendizaje por refuerzo es un sistema muy general para aprender a tomar decisiones secuenciales. Por otra parte, el aprendizaje profundo es el mejor conjunto de algoritmos disponibles para aprender representaciones. Combinar estos dos modelos diferentes es la mejor opción que tenemos disponible para lograr buenas representaciones de estados en tareas complejas, no sólo para resolver juegos sencillos si no también complicados problemas reales.

En definitiva, el potencial y las posibilidades de esta metodología convierten al aprendizaje por refuerzo profundo en un campo muy interesante, que puede que nos lleve un paso más cerca al mundo de la inteligencia artificial general.

1.2. Objetivos

1. Comprender en qué es el aprendizaje por refuerzo distinto a otros métodos de aprendizaje automático, y en qué situaciones puede ser usado.
2. Estudiar los fundamentos del aprendizaje por refuerzo, entendiendo sus características, componentes y limitaciones.
3. Poner a prueba lo aprendido con simulaciones prácticas, en las que implementemos algoritmos de aprendizaje por refuerzo y estudiemos sus resultados.
4. Adentrarnos en el campo del aprendizaje profundo, donde veremos los fundamentos de las redes neuronales.

5. Estudiar cómo es posible combinar las redes neuronales y el aprendizaje por refuerzo con el objetivo de sortear las limitaciones de ambos. Será un camino en el que, paso a paso, encontraremos soluciones a los problemas que surjan hasta lograr un modelo estable de aprendizaje por refuerzo profundo.

6. En cada hito del camino evaluaremos los resultados obtenidos, para comprobar que las soluciones mejoran en rendimiento y estabilidad.

1.3. Estructura de la memoria

Nuestro trabajo intercala capítulos teóricos con aplicaciones prácticas de lo visto en dichos capítulos, resultando en dos bloques diferenciables: el primero sobre aprendizaje por refuerzo y el segundo sobre aprendizaje por refuerzo profundo.

- **Capítulo 1, Introducción.** Motivación y objetivos de nuestro proyecto.
- **Capítulo 2, Aprendizaje por Refuerzo.** El capítulo comienza con una comparación del aprendizaje por refuerzo con otros métodos de aprendizaje automático. Después, se proporciona toda la base teórica necesaria para comprender el aprendizaje por refuerzo. Para terminar, explicaremos en profundidad Q-Learning, el algoritmo que utilizaremos en nuestras pruebas.
- **Capítulo 3, Q-Learning en acción.** Introduciremos OpenAI Gym, la herramienta utilizada durante nuestras pruebas. Aplicaremos los conocimientos de aprendizaje por refuerzo en CartPole, un problema que nos permitirá evaluar resultados y limitaciones.
- **Capítulo 4, Redes Neuronales y Q-Learning.** Proporcionaremos el marco teórico necesario para comprender las redes neuronales. Después explicaremos cómo es posible aplicarlas a técnicas de aprendizaje por refuerzo, obteniendo las DQN.
- **Capítulo 5, DQNs en acción.** Volveremos a resolver CartPole y nos enfrentaremos a MountainCar, un problema que presenta nuevos retos para nuestro agente.
- **Capítulo 6, Conclusiones.** Síntesis de todo lo aprendido durante el camino. Puntos clave de las distintas etapas, limitaciones y oportunidades de cara al futuro.

CHAPTER 1

Introduction

"The young man or the young woman must possess or teach himself, train himself, in infinite patience, which is to try and to try and to try until it comes right"
— William Faulkner

1.1. Motivation

Artificial Intelligence is one of the computation fields that most interest has generated, both among experts and general public, who is more interested in the leisure side of it. Been able to grant a machine the hability to reason and perform functions only associated with the human intellect has always been considered Sci-Fi. And nonetheless this very idea has experienced a huge technological leap in the last years.

To better understand this concept from its roots, we need to reference the **Turing Test**, formulated by Alan Turing [39], which consists in performing a series of questions to a machine. The test is considered passed if the evaluator cannot discern if the answers were given by a human or a machine.

Trying to define Artificial Intelligence quickly leads us to concepts like *thought process* or *reasoning*, which end up driving to more complex ones like *behaviour*. From this ideas we can find other definitions, classified in the matrix 1.1.

According to these definitions we can differentiate two schools of thought:

- An empirical approach (left column) with the human being as the focus

Thinking Humanly	**Thinking Rationally**
"The exciting new effort to make computers think ... *machines with minds*, in the full and literal sense." (Haugeland, 1985)	"The study of mental faculties through the use of computational models." (Charniak and McDermott, 1985)
"[The automation of] activities that we associate with human thinking, activities such as decision-making, problem solving, learning ..." (Bellman, 1978)	"The study of the computations that make it possible to perceive, reason, and act." (Winston, 1992)
Acting Humanly	**Acting Rationally**
"The art of creating machines that perform functions that require intelligence when performed by people." (Kurzweil, 1990)	"Computational Intelligence is the study of the design of intelligent agents." (Poole *et al.*, 1998)
"The study of how to make computers do things at which, at the moment, people are better." (Rich and Knight, 1991)	"AI ... is concerned with intelligent behavior in artifacts." (Nilsson, 1998)

Figure 1.1: Artificial Intelligence definitions, Russell y Norvig [31]

of the investigation. This involves observations and hypotheses about human behavior. One example of this booming category would be an autonomous car system, which in some extreme situations might have to take a really difficult decission involving the risk of human lives. In this case, we need to ask ourselves: What would a real driver do in that situation?

- A rationalist approach (right column) implies a combination of mathematics and engineering. Proyects as voice assistants or the Boston Dynamics robots belong to this category. None of them need to actually understand orders and information in a "human" level. Assistants do not need to be assertive, they just need to process the information correctly and act accordingly. In case of the robots, they need to learn how to adapt to several environments, just to accomplish their functions.

Lots of experts have studied both approaches in different ways. Our goal will be making our *agent* able to take the "right choices" with its available knowledge. Being more specific, we will try to make an agent witout any previous knowledge able to learn simple tasks through constant interaction with the environment. This environment will provide new experiencies to the agent from which it may extract knowledge.

This learning approach is an Artificial Intelligence's field called Reinforcement Learning. We will also investigate Neural Networks, resulting in the

so called Deep Reinforcement Learning. The popularity of this techniques has grow a lot during this last years. The words of Koray Kavukcuoglu, the director of research at Deepmind, describe very well its potential:

> Reinforcement Learning is a very general framework for learning sequential decision making tasks. And Deep Learning, on the other hand, is of course the best set of algorithms we have to learn representations. And combinations of these two different models is the best answer so far we have in terms of learning very good state representations of very challenging tasks that are not just for solving toy domains but actually to solve challenging real world problems

Ultimately, the potential of this methodology makes Deep Reinforcement Learning a really interesting topic, which might bring us one step closer to General Artificial Intelligence.

1.2. Goals

1. Understand why is Reinforcement Learning different from other Machine Learning methods, and in which situations it can be applied.
2. Study the fundamentals of Reinforcement Learning, understanding its components, implementations and limits.
3. Test what we have learned with practical simulations, on which we will implement Reinforcement Learning algorithms, and study the results.
4. Dive into the Deep Learning field, where we will see the fundamentals of Neural Networks.
5. Learn how is it possible to combine Neural Networks with Reinforcement Learning techniques, trying to avoid the limitations of both. It will be a journey where, step by step, we will find solutions to the problems that arise, until we come with a stable Deep Reinforcement Learning solution.
6. After every milestone we will evaluate the results, contrasting if our solutions improve stability and performance.

1.3. Structure of the memory

Our project intercalates theorical chapters with practical applications of what we have seen, resulting in two big blocks: One about Reinforcement

Learning and other about Deep Reinforcement Learning.

- **Chapter 1, Introduction.** Motivation and goals of our proyect.
- **Chapter 2, Reinforcement Learning.** The chapter begins with a comparison between different Machine Learning techniques. Afterwards, the needed Reinforcement Learning theorical background is provided. Finally, we will thoroughly explain Q-Learning, one of the most commonly used Reinforcement Learning algorithms.
- **Chapter 3, Q-Learning in action.** We will introduce OpenAI Gym, the framework used in our tests. Later we will apply the acquired Reinforcement Learning knowledge into CartPole, an environment that will allow us to measure results and experience Reinforcement Learning's limitations.
- **Chapter 4, Neural Networks and Q-Learning.** We will give the necessary theorical background to understand Neural Networks, followed by how is possible to combine them with Reinforcement Learning techniques, obtaining the so called Deep Q-Networks.
- **Chapter 5, DQNs in action.** We will solve CartPole again, this time applying the new learned approaches. Then we will face MountainCar, a new and challenging environment for our agent.
- **Chapter 6, Conlusions.** Summary of everything we have achieved so far. Lessons learned, highlights and future work.

CAPÍTULO 2

Aprendizaje por Refuerzo

"El comportamiento es modelado y mantenido por sus consecuencias"
— B. F. Skinner

2.1. Aprendiendo a aprender

Una de las funciones humanas de las que necesitaremos dotar a nuestra máquina en busca de este *rendimiento ideal* es el **aprendizaje**.

El campo del **aprendizaje automático** (o *Machine Learning*) se encarga de esta tarea, a través de la generalización y la búsqueda de patrones en experiencias pasadas. En función del tipo de *realimentación* obtenido existen diferentes técnicas de aprendizaje, diseñadas para distintos casos y objetivos:

- **Aprendizaje supervisado** (*Supervised Learning*): el agente observa una serie de ejemplos de entradas y salidas, aprendiendo una función que es capaz de asignar a una entrada su salida correspondiente. Se llama **supervisado** porque esta serie de ejemplos, llamada conjunto de entrenamiento, debe estar correctamente clasificada desde un primer momento. Podría decirse que el agente aprende en base a estas experiencias, hasta que llegado un punto es capaz de clasificar una entrada completamente nueva, de modo que la exactitud de esta clasificación dependerá del entrenamiento recibido.
- **Aprendizaje no supervisado** (*Unsupervised Learning*): a diferencia del supervisado, en el aprendizaje no supervisado los ejemplos no cuen-

tan con una etiqueta que los clasifica inequívocamente. En su lugar el agente busca patrones en el conjunto de entrada, intentando extraer características comunes de sus elementos. Uno de los usos más comunes del aprendizaje no supervisado es la agrupación o **clustering**: la unión de ejemplos como grupos o *clústeres* que comparten características comunes. Por ejemplo, la agrupación de películas con características similares, de modo que si a un usuario le resultan interesantes varias de un mismo *clúster*, es muy probable que también disfrute de otros elementos de esa misma agrupación.

- **Aprendizaje por refuerzo** (*Reinforcement Learning*): el agente aprende a partir de una serie de refuerzos, recompensas si son positivos y penalizaciones en caso de ser negativos. Por ejemplo, cuando una mascota cumple una orden y recibe una galleta como recompensa, o un músico desafina en directo y recibe un abucheo por parte del público. Además, el agente puede "pensar" a largo plazo, de forma que su comportamiento le permita conseguir una recompensa mayor en un futuro, así como adaptarse a entornos totalmente nuevos para él. Esta técnica de aprendizaje será el punto de partida de nuestro proyecto y explicaremos su funcionamiento a lo largo de este capítulo.

2.1.1. ¿Por qué aprendizaje por refuerzo?

Como muchos otros campos pertenecientes a la rama de inteligencia artificial, el aprendizaje por refuerzo [40] se inspira en estudios sobre el comportamiento en humanos y animales. De esta forma toma su base en la psicología conductista [33], en la que un algoritmo decide si una acción tomada ha sido positiva o negativa, y la refuerza consecuentemente.

Pero antes de profundizar en ello, analicemos por qué es el aprendizaje por refuerzo una opción interesante y por qué será la que utilicemos en nuestro proyecto.

Echemos un vistazo a las técnicas de aprendizaje vistas en la sección anterior. ¿Qué es interesante de cada una de ellas? Empecemos por el *aprendizaje no supervisado*: este es quizá el caso más sencillo, ya que lo único que necesitaremos es algo de lo que vivimos rodeados, **información**. En internet disponemos de grandes cantidades de ella. Podemos servirnos de las APIs de servicios de música, series, entretenimiento, organismos públicos... En la mayoría de los casos, únicamente tendremos que normalizar y limpiar la información antes de alimentarla a nuestro algoritmo. Sin embargo, el *aprendizaje supervisado* necesita una entrada de datos normalizada. A la hora de presentar nueva información necesitamos que esté correctamente clasificada y, aunque en internet hay conjuntos de entrenamiento de gran calidad y variedad, podría darse el caso de que ninguno se adapte a nuestras necesidades.

En esta situación podríamos construir uno nosotros mismos, pero la tarea sería larga y tediosa.

El caso del *aprendizaje por refuerzo* es distinto. El agente no necesita grandes cantidades de casos de prueba, sino un entorno con el que interactuar por sí mismo, de forma que los casos de prueba se generen de forma automática. La manera más rápida de hacernos con un entorno en el que dejar a nuestro agente y verlo actuar es, claramente, simulándolo. Una simulación es la forma más sencilla de controlar las condiciones y el progreso de nuestro experimento, además de quitarnos otros problemas que podrían darse en un entorno real: variables que no se tuvieron en cuenta, cambios inesperados en dicho entorno, o cualquier tipo de error no controlado... El mundo real, en general.

Un dominio muy interesante sobre el que realizar simulaciones de aprendizaje por refuerzo son los videojuegos [30]. Durante los últimos años han tenido bastante éxito debido a varios motivos; entre ellos, porque son entornos controlados y reproducibles. De esta forma, no habrá ninguna diferencia en nuestras pruebas más allá de las propias acciones que realice nuestro agente sobre la simulación, disponiendo así de unos resultados fiables y fácilmente comparables. Además, los videojuegos resultan ser un dominio accesible, de diversa complejidad y ya preparados para nuestro tipo de agente: muchos de ellos disponen de un sistema de puntuación que sirve perfectamente como refuerzo, ya sea por recolección de objetos, número de enemigos derrotados, tiempo... Nuestro agente podrá utilizar fácilmente sus experiencias pasadas para decidir cuál es la mejor decisión a tomar en cada nueva partida.

Podemos ver el funcionamiento del aprendizaje por refuerzo en el pseudocódigo 2.1:

Listing 2.1: Pseudocódigo Aprendizaje por Refuerzo

```
Initialize state

Repeat (for each step of the episode):
    Select action
    Perform action; Observe reward and next-state
    state ← next-state
```

Pero hay mucho más. En las próximas secciones describiremos los distintos componentes de nuestros problemas y las técnicas usadas sobre los diferentes dominios para su resolución, además de cuestiones más complejas: cómo definir la interacción entre agente y entorno, cómo representar el entorno de una forma eficiente o cómo discernir si una acción tomada ha sido buena o mala.

2.2. Elementos del Aprendizaje por Refuerzo

A lo largo de este capítulo el término "agente" ha aparecido en repetidas ocasiones. Un **agente** no es más que el sujeto de nuestro experimento. Si nos encontrásemos en el mundo real, un agente podría ser un ratón intentando salir de un laberinto. En nuestro contexto software, llamamos agente a un programa capaz de interactuar con el **entorno**, aprender de él y **tomar decisiones**.

El **entorno** es el otro elemento fundamental del aprendizaje por refuerzo. Es lo que proporciona información al agente, quien la usará para tomar una decisión. Esta decisión en forma de acción provocará cambios en el entorno, los cuales servirán para que éste proporcione la recompensa consecuente al agente.

Más allá del agente y el entorno, existen cuatro subelementos en un sistema de aprendizaje por refuerzo: una *política*, una *recompensa*, una *función de utilidad* y un *modelo* del entorno.

La **política** define la forma en la que el agente afronta un determinado problema. Por ejemplo, si nos encontrásemos a un enemigo en un videojuego, una política válida sería enfrentarnos a él. Huir podría ser otra política perfectamente válida. Generalizando esto, la política es una función que asocia el estado en que se encuentra el entorno y la acción que tomará el agente.

La **recompensa** es la forma por la cual el agente obtiene realimentación del entorno. Después de cada acción tomada, el agente obtiene un valor numérico que le ayude a saber si dicha acción fue buena o mala. A través de la recompensa se modela y modifica, de cara a futuras acciones, la política del agente, pudiendo éste aprender a maximizar dicha recompensa a lo largo de una simulación y aprender a jugar mejor.

Pero la recompensa no es suficiente para lograr un comportamiento óptimo por parte del agente. Es aquí donde entra en juego la **función de utilidad**. Si nos paramos a pensarlo, existen acciones que no nos proporcionan una gran recompensa por si solas, pero facilitan el camino a otras que generan una recompensa aún mayor. En una partida de *Tetris* podríamos apilar piezas a un lado mientras esperamos la ficha en forma de barra vertical, completando así cuatro líneas en un solo movimiento y obteniendo muchos más puntos que si hubiésemos completado una única línea cuatro veces seguidas. Desde un punto de vista más teórico, podemos entender la utilidad como la recompensa máxima a la que se podría llegar desde un estado. Es decir, una recompensa a largo plazo.

Dependiendo del problema que afrontemos tendremos que ajustar la importancia que damos a la recompensa y a la utilidad, para que nuestro agente sea capaz de lograr buenos resultados.

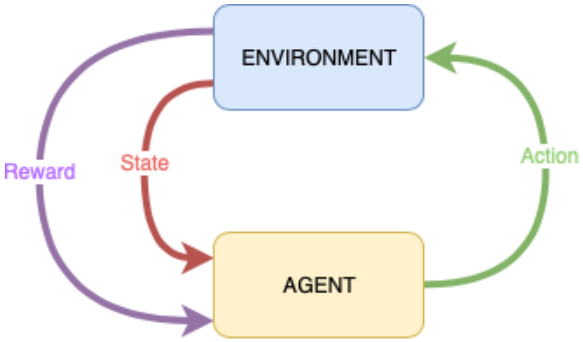

Figura 2.1: Modelo de interacción del aprendizaje por refuerzo

Llegamos al último elemento por definir, el **modelo**. El modelo es una representación del entorno que el agente construye y mediante la cual es capaz de aprender a optimizar la política, así como a predecir las transiciones y recompensas obtenidas. No todos los agentes usan un modelo, también existen aquellos *libres de modelo* en los cuales el agente depende únicamente de la prueba y error.

Podemos apreciar la relación de estos elementos en la figura 2.1.

2.2.1. Proceso de decisión de Markov

El proceso de decisión de Markov [28], al cual nos referiremos como MDP, llamado así por el matemático ruso Andrei Markov, es un modelo matemático mediante el cual podemos modelar la resolución de cierta clase de problemas relacionados con la toma de decisiones. Para comprender mejor cómo funciona, definiremos los elementos que componen nuestro MDP:

- **S**: conjunto de estados finitos en los que nos podemos encontrar en un determinado momento.
- **A**: conjunto de acciones que pueden ser tomadas.
- **Modelo de transición**: función que define el estado futuro (**s'**) basándose en el estado actual (**s**) y la acción tomada en dicho estado (**a**). Es representado mediante la función **P(s, a, s')**. El modelo de transición puede ser de dos tipos:
  - Determinista: ejecutar una acción en un estado determinado siempre lleva al mismo estado siguiente.
  - Probabilista: representa la probabilidad de que un estado siguiente sea alcanzado si se toma una acción en un estado.

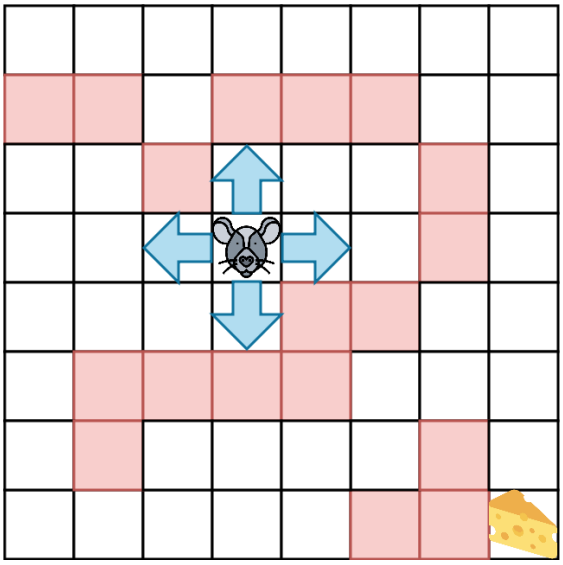

Figura 2.2: MDP como laberinto

- **Recompensa**: "puntuación" positiva o negativa que se obtiene al tomar una acción en un determinado estado. Se representa mediante la función **R(r | s, a)**.

Podemos relacionar todos estos elementos de modo que para un estado actual (**s**) se toma determinada acción (**a**), llevándonos así a un estado siguiente (**s'**) definido en el modelo de transición, y obteniendo una recompensa (**r**) asociada a la toma de dicha decisión en ese estado. Cabe destacar que el MDP relaciona estos elementos de forma que, para obtener los resultados futuros, sólo se toma en cuenta el estado inmediatamente anterior junto la última acción escogida; es decir, no se guarda memoria de las transiciones anteriores.

Pongamos el ejemplo de un laberinto, como podemos ver representado en la figura 2.2. Nuestro conjunto de **estados** serían todas las posibles posiciones dentro del laberinto donde podemos situarnos y las **acciones** serían las direcciones en las que nos podemos mover en cada posición del laberinto. Nuestro **modelo de transición** estaría definido de modo que si nos encontramos en una posición $[x, y]$ y decidimos tomar la acción de movernos hacia arriba, nuestra siguiente posición (estado) sería $[x, y+1]$. Finalmente podríamos definir la **recompensa** como una puntuación de $+100$ si conseguimos llegar a la salida, 0 si conseguimos movernos a una posición buena dentro del laberinto o -1 si nos chocamos contra una pared.

El hecho de que un agente tome una acción y haga cambiar el estado del entorno lo llamamos **paso**. Un paso está compuesto por el estado en el que

el agente se encuentre, la acción tomada y la recompensa obtenida:

$$paso = (estado, accion, recompensa)$$

Los pasos se suceden hasta que el entorno llega a un estado de "finalizado", de forma que todos los pasos recorridos hasta dicho estado componen un **episodio**. Un entorno puede llegar a su estado "finalizado" si perdemos el juego o, si por el contrario, lo ganamos. Podemos diferenciar 2 tipos de entornos y sus diferentes objetivos para conseguir ganar:

- **Entornos finitos**: finalizan cuando el agente consigue alcanzar un estado objetivo o si el número de pasos que componen cada episodio hasta alcanzarlo es limitado. Un ejemplo de entorno finito sería el laberinto antes planteado, en el cual existe el objetivo de llegar a la salida.
- **Entornos infinitos**: son aquellos entornos que carecen de un estado objetivo. De este modo, el objetivo que se plantea en este caso es el de mantenerse de forma infinita en un estado bueno. Dicho de una forma mucho más simple, el agente debe aprender a no perder.

Pero, ¿cómo sabemos o cómo decidimos qué acción es mejor tomar en cada estado? Éste es el objetivo de nuestro agente, el de encontrar una política óptima que nos ayude a tomar buenas decisiones acerca de qué acciones tomar en cada momento basándonos en el estado actual y conseguir maximizar nuestra "recompensa futura" (*future return*).

Entendemos como **recompensa futura** la recompensa a largo plazo que el agente puede acumular siguiendo su política a partir del estado actual. De este modo, el objetivo de nuestro agente es optimizar su política para buscar obtener una mayor recompensa a la larga, aunque eso conlleve obtener recompensas inmediatas más pequeñas. Este concepto se conoce como **recompensas retrasadas**, ya que no las obtendremos instantáneamente tras ejecutar una acción, sino después de la sucesión de una serie de acciones.

Gracias a esto conseguimos que nuestra política aprenda a pensar, no sólo en qué acción tomar en este momento, sino en qué acción tomar después de la misma. De esta manera, un inconveniente que podemos encontrar es que nuestra política piense demasiado a largo plazo y le lleve un elevado número de pasos encontrar una recompensa que merezca la pena, de modo que nuestra recompensa futura diverge. Para evitar que esto suceda, introduciremos los factores de descuento.

Un **factor de descuento** es un pequeño valor que se descuenta de la recompensa con el fin de evitar que, como acabamos de ver, nuestra política tome demasiados episodios antes de encontrar una recompensa positiva, de forma que se equilibre la búsqueda de recompensas cercanas en el tiempo y se

maximice la recompensa total, del mismo modo que evite que la recompensa futura tienda a infinito en los entornos infinitos. Teniendo esto en cuenta, podemos definir nuestra función de recompensa como:

$$R_t = \sum_{k=0}^{T} \gamma^t r_{t+k+1}$$

El factor de descuento, γ, representa cuánto se descuenta de la recompensa en cada paso, y su valor siempre se encuentra entre 0 y 1, de forma que valores más altos corresponden a una penalización menor. Los factores de descuento más comúnmente usados se encuentran entre 0,97 y 0,99.

Esta búsqueda del balance entre las recompensas cercanas y futuras está muy relacionado con el problema de **Exploración vs. Explotación**. Nuestro agente contará con un hiperparámetro *exploración*, el cual determina con qué frecuencia el agente decide tomar una acción aleatoria (exploración) en lugar de aquella que le proporcione el valor más prometedor (explotación), con la esperanza de encontrar un nuevo estado en el que nunca haya estado y que pueda resultar más beneficioso. En otras palabras, el agente decide si sacar provecho de lo ya aprendido o arriesgarse e intentar aprender algo mejor.

Siguiendo con esta idea, nuestra estrategia para equilibrar el dilema de la explotación vs. exploración se basa en el llamado **ϵ-Greedy** [38]. ϵ-Greedy es una estrategia que implica tomar una decisión en cada paso para tomar la acción registrada por el agente con una mayor recompensa o tomar una acción al azar. La probabilidad de que el agente tome una acción aleatoria se rige por el parámeto epsilon (ϵ). Podemos hacer un acercamiento de esta idea en el código 2.2.

De esta manera, se busca obtener un balance entre la exploración y explotación, con el objetivo de evitar que se explore demasiado, lo que conllevaría que nuestro agente no optimizase lo suficiente su política, ni que explote demasiado, lo que significaría caer en un mínimo local.

Para ayudarnos a conseguir este balanceo entre exploración y explotación, lo que haremos será reducir dinámicamente el valor de epsilon, de manera que al principio explore más opciones y después explote los conocimientos adquiridos para terminar convergiendo a una política concreta.

Listing 2.2: Pseudocódigo ϵ-Greedy

```
if random() < epsilon:
    # explore
    Choose a random action

else:
    # exploit
    Choose best action for current state
```

2.3. Q-Learning

Hasta ahora hemos hablado de múltiples conceptos y elementos: agente, entorno, recompensa, política... Llega el momento de aplicarlos de manera conjunta para conseguir una técnica de aprendizaje tangible. **Q-Learning** es una técnica de Aprendizaje por Refuerzo que tiene como objetivo enseñar al agente qué acción es mejor tomar en cada circunstancia. Se trata de un algoritmo *off-policy* con *diferenciación temporal* que busca encontrar una función acción-utilidad que nos dirá cómo de bueno es ejecutar una acción es un determinado estado. Los algoritmos *off-policy* son capaces de encontrar una política óptima independientemente de la politica utilizada por el agente para elegir acciones, siempre que pase por todos los estados suficientes veces [27].

El algoritmo de Q-Learning cuenta con una tabla-Q con los estados posibles contemplados a partir del MDP, en la que se van almacenando las sumas de las posibles recompensas futuras. También conocidas como **valores-Q**, se predicen o actualizan usando el valor-Q del estado futuro s' y la acción a' que más utilidad produzca. Siendo $Q(s, a)$ el valor de aplicar una acción a en el estado s, los valores devueltos por nuestra función-Q serán la utilidad máxima alcanzable de forma:

$$U(s) = \max_{a} Q(s, a)$$

Con **diferenciación temporal** hacemos referencia al hecho de que, en busca de un aprendizaje correcto, nuestro agente tendrá que tener en cuenta las recompensas actualizadas obtenidas en un momento futuro para actualilzar los valores-Q registrados anteriormente. Reformulando esto, el agente deberá actualizar las estimaciones obtenidas en un momento $Q(s, a)$ con el valor actualizado en pasos siguientes $Q(s + k, a)$. Para poder hacerlo, debemos encontrar un equilibrio entre las recompensas inmediatas y las futuras. Como podemos imaginar, potenciar la importancia de las recompensas directas puede perjudicar el valor de las futuras.

Existen varias fórmulas para calcular la utilidad de los estados. Uno de los métodos más utilizados es a través de la **Ecuación de Bellman** [4], que nos indica que la máxima recompensa futura de tomar una acción es la suma de la recompensa actual y la máxima recompensa futura del siguiente episodio, permitiéndonos así obtener una relación entre valores-Q.

$$Q^*(s_t, a_t) = r_t + \gamma \; max_{a'} \; Q^*(s_{t+1}, a')$$

De este modo, podemos calcular el valor-Q para un par estado-acción, del mismo modo que con cada nuevo episodio podremos actualizar valores-Q previamente calculados con nueva información obtenida.

Q-Learning cuenta con tres hiperparámetros, dos de los cuales ya hemos visto: la *exploración* y el *factor de descuento*, introducido al final de la sección anterior. El tercer hiperparámetro en cuestión corresponde a la **tasa de aprendizaje**. Este hiperparámetro, que toma valores entre 0 y 1, nos indica cuánta nueva información actualizamos sobre los valores-Q previamente calculados. De este modo, si nuestra tasa de aprendizaje es igual a 0, con cada nuevo episodio no se actualizarán valores-Q con nueva información, y si es igual a 1, los valores previos se sobreescribirán por completo con la nueva información obtenida.

Podemos relacionar esta tasa de aprendizaje (α) y la ecuación antes propuesta del siguiente modo:

$$Q^*(s_t, a_t) = (1 - \alpha)Q(s_t, a_t) + \alpha[r(s_t, a_t) + \gamma \; max_{a'} \; Q^*(s_{t+1}, a')]$$

Combinando todos estos factores, podemos definir el pseudocódigo del funcioniamiento de Q-Learning como apreciamos en el Listing 2.3, de manera que relacionemos todos los hiperparámetros y conceptos vistos hasta ahora. Cabe destacar el booleano `end` que nos indica si el episodio ha terminado o no y la variable `next_max` que se corresponde al valor-Q máximo en el estado-siguiente.

Listing 2.3: Pseudocódigo Q-Learning

```
import numpy as np

# Initialize Q-table, states and actions
states  = {1, .., n_s}
actions = {1, .., n_a}
q_table = np.zeros((states, actions))

for i in range(episodes):
    Initialize state ∈ states

    while not end:
        Choose an action ∈ actions

        Perform action
        Observe next_state, reward and end

        # Update Q-table
        Q'(state,action) =
            (1 - LEARNING_RATE) * Q(state,action) +
            LEARNING_RATE * (reward +
            DISCOUNT_FACTOR * next_max)

        state = next_state
```

Llegados a este punto, podemos observar cómo se crea la tabla-Q que relacionará los pares estado-acción con su correspondiente valor-Q. En consecuencia, nos encontramos con que esta tabla crece alarmantemente rápido al tener que almacenar todas las combinaciones estado-acción. El número de posibles estados puede ser inabarcable incluso para problemas con un pequeño grado de complejidad, haciendo excesivamente costoso el hecho de visitar todas las experiencias recogidas, así como de actualizarlas, y convirtiendo el problema en algo inmanejable desde un punto de vista computacional.

CAPÍTULO 3

Q-Learning en acción

"La vida no es un problema por resolver, sino una realidad que experimentar"
— Søren Kierkegaard

3.1. Lenguajes y herramientas utilizadas

La elección de Python [15] como lenguaje para nuestras pruebas se debe a múltiples motivos. Para empezar se trata de un lenguaje de programación que permite empezar a trabajar muy rápidamente, no se necesitan grandes entornos de desarrollo y su gestión de paquetes es rápida y sencilla, por lo que en apenas unos minutos es posible tener un proyecto configurado y listo para trabajar en cualquier ordenador.

Además, durante los últimos años Python ha sido la opción por excelencia para el aprendizaje automático. Esto se traduce en una gran cantidad de recursos disponibles, ya sea en forma de frameworks, librerías de utilidades, documentación o una gran comunidad activa dispuesta a resolver cualquier duda.

La distribución elegida de Python ha sido Anaconda [2], la cual incluye multitud de librerías básicas para trabajar en aprendizaje automático. Además de proporcionar facilidades extra, como es poder mantener distintas instalaciones de Python en entornos aislados, con la seguridad de que los paquetes o configuraciones para otros proyectos no se verán alterados.

Algunas de las librerías que hemos utilizado han sido NumPy y Mat-

plotlib. Numpy [23] dispone de arrays multidimensionales y multitud de funciones para trabajar con ellos. Se tratan de implementaciones de muy alto rendimiento en C, utilizadas a través de la comodidad y simpleza de Python. Matplotlib [17] es una biblioteca para la generación de gráficas a partir de datos almacenados en arrays. Generar gráficas sencillas como las que mostraremos más adelante es rápido e intuitivo, pero la librería es lo suficientemente compleja como para poder generar elementos mucho más avanzados.

Más adelante también utilizaremos Keras. Keras [11] es una biblioteca de alto nivel para aprendizaje profundo. Es capaz de funcionar con varios *backends*, como TensorFlow [1], Theano [36] o CNTK [32]. Su principal ventaja es que permite trabajar de forma rápida y amigable a los usuarios novicios, pero al mismo tiempo posee la suficiente flexibilidad para aquellos usuarios avanzados que necesiten mayor control sobre las implementaciones.

3.2. OpenAI

OpenAI es una iniciativa cuyo objetivo es asegurar que la **inteligencia artificial general** (AGI) beneficie a toda la humanidad. En su web se encuentra el siguiente manifiesto [24]:

> La misión de OpenAI es asegurar que la inteligencia artificial general - refiriéndose a sistemas altamente autónomos que superen el rendimiento humano en el trabajo más valioso económicamente - beneficie a toda la humanidad. Intentaremos desarrollar AGI de una forma segura y beneficiosa, pero también consideraremos nuestra misión completa si nuestro trabajo ayuda a otros a conseguir esta meta.

En busca de este objetivo, OpenAI sigue tres caminos:

- **Investigación y desarrollo:** sus contribuciones al aprendizaje automático como estado del arte durante los últimos años no han sido pocas: desde nuevos algoritmos de aprendizaje por refuerzo [13] hasta demostraciones de cómo una inteligencia artificial puede sobrepasar a jugadores expertos [25].
- **Recursos académicos** puestos a disposición por OpenAI [26] para todo aquél que quiera aprender sobre el aprendizaje por refuerzo profundo.
- **Herramientas y plataformas** mediante las cuales facilitar la labor de investigación y pruebas del aprendizaje automático.

Es nuestra investigación nos serviremos de una de estas plataformas, **OpenAI Gym**, para experimentar y descubrir los puntos fuertes y débiles de las distintas técnicas y algoritmos de aprendizaje que utilizaremos.

3.2.1. OpenAI Gym

Gym es un conjunto de herramientas para desarrollar y comparar algoritmos de aprendizaje de refuerzo. A través de sus diferentes entornos los agentes son capaces de aprender cualquier cosa, desde caminar hasta jugar a juegos como Pong o Pinball.

Para ello la librería aporta al usuario una interfaz sencilla, a través de la cual se le provee un entorno de simulación. En nuestro caso, el algoritmo que proporcionaremos será el agente que interaccionará con el entorno para aprender de él e intentar lograr unos objetivos.

Listing 3.1: Creación y renderizado del entorno CartPole

```
import gym

env = gym.make('CartPole-v0')
env.reset()

for _ in range(1000):
    # Muestra el estado actual del entorno
    env.render()
    # Toma una accion aleatoria
    env.step(env.action_space.sample())

env.close()
```

Un ejemplo mínimo del uso de la librería puede verse en el fragmento de código 3.1. La operación `gym.make()` crea una instancia del entorno especificado. No debemos preocuparnos por las reglas que rigen su funcionamiento (leyes físicas, acciones disponibles, resultados desencadenados por esas acciones...), ya que todo ello se hace automáticamente. La instrucción `env.reset()` devuelve el entorno a su estado inicial. Más adelante encontramos un bucle, en el que dos acciones ocurren en cada iteración. Primero, el estado actual nos es mostrado mediante una interfaz gráfica a través de la función `env.render()`, como puede comprobarse en la imagen 3.1. A continuación una acción aleatoria es escogida del conjunto de acciones posibles mediante `env.action_space.sample()` para después ser tomada como decisión en `env.step(action)`.

Todos los entornos respetan la estructura definida en la sección 2.2.1, correspondiente a los procesos de decisión de Markov. Cada vez que interaccionamos con un entorno a través de `env.step()` cuatro elementos nos son

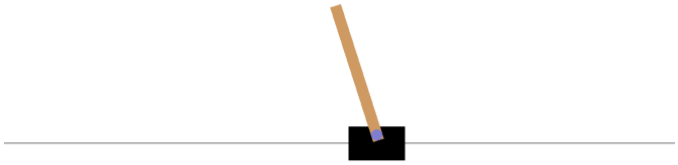

Figura 3.1: Entorno CartPole, Andrew G. Barto [3]

devueltos:

- **Observación** (objeto): consiste en un array que contiene la información necesaria para describir el estado actual.
- **Recompensa** (float): refuerzo para el agente proporcionado por el entorno.
- **Fin** (boolean): señal que indica si la simulación ha concluido. En la mayoría de los entornos puede deberse a dos motivos: que el agente haya logrado su objetivo o que haya fracasado.
- **Info** (diccionario): información de diagnóstico útil para la depuración del agente. Esta información no debe ser usada por el agente para aprender.

Nuestro objetivo consistirá en que el agente aprenda por sí mismo cuál es la acción que más le conviene tomar en cada situación, buscando maximizar la recompensa.

3.3. CartPole

CartPole es uno de los problemas más famosos, introductorios al aprendizaje automático. Fue introducido por primera vez en Andrew G. Barto [3] y su descripción en OpenAI es la siguiente:

> Un mástil está unido por una bisagra a un carro, el cual se mueve a lo largo de una pista sin rozamiento. El sistema es controlado

Num	Observación	Min	Max
0	Posición del carro	-4.8	4.8
1	Velocidad del carro	-Inf	Inf
2	Ángulo del poste	-24º	24º
3	Velocidad del poste	-Inf	Inf

Tabla 3.1: Observación del entorno para CartPole

> aplicando una fuerza de +1 o -1 al carro. El péndulo (mástil) comienza en posición vertical, el objetivo es prevenir que caiga. Una recompensa de +1 es otorgada por cada *timestep* que el mástil permanece erguido. El episodio finaliza cuando el mástil se encuentra desplazado más de 15 grados de su posición vertical, o el carro se mueve más de 2.4 unidades del centro.

Nuestro conjunto de acciones, como se indica en el enunciado, tiene dos elementos: aplicar fuerza hacia la derecha o aplicar fuerza hacia la izquierda. El agente deberá decidir cuál de estas dos acciones tomar y se lo comunicará al entorno. Éste responderá con los elementos descritos en la sección 3.2.1. Su contenido se especifica a continuación:

- **Observación** (objeto): los rangos de valores para el array pueden verse en la tabla 3.1.
- **Recompensa** (float): +1.0 mientras la simulación continúe.
- **Fin** (boolean): la señal cambiará a `True` si el agente no consigue mantener el mástil erguido (como se indica en el enunciado) o, por el contrario, si se mantiene en pie durante suficiente tiempo (por defecto 200 *timesteps*).

Como podemos ver a través de las observaciones, este es uno de esos problemas en los que el número de estados distintos es potencialmente infinito, debido a que estamos tratando con valores continuos. Por ello tendremos que encontrar una buena **función de discretización** que simplifique la representación del estado y nos permita trabajar con un número óptimo de estados. Por suerte para nosotros, disponer de un número tan limitado de acciones posibles disminuye la complejidad del problema.

3.3.1. Discretizando el estado

A la hora de discretizar una observación, una de las mejores formas de hacerlo es a través de una simplificación de la misma. Este acercamiento es

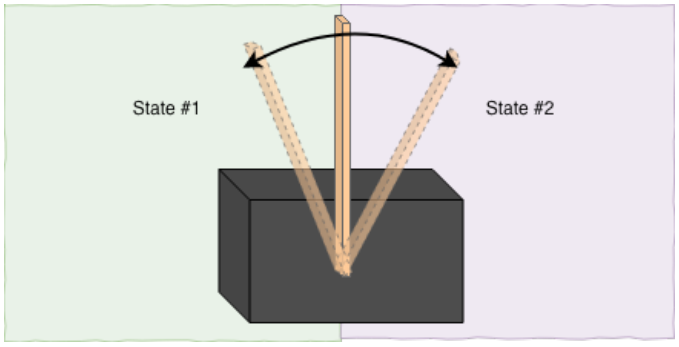

Figura 3.2: Simplificación de estados, fuente propia

una adaptación del *Constraint Relaxation* [29] que se aplica en muchos problemas de satisfacción de restricciones. De esta forma podemos comprender el problema desde un punto de partida mucho más sencillo, para después pulir la fórmula y tener en cuenta más variables. Además, podemos ir evaluando el rendimiento de nuestro agente a lo largo del camino, y así poder comprobar si las modificaciones que vamos añadiendo realmente mejoran su comportamiento o simplemente añaden ruido.

Centrémonos por un momento en el objetivo más básico del problema: mantener el mástil en pie. Podemos dividir el problema en dos estados muy simples, uno en el que el mástil está cayendo hacia la derecha y otro en el cae hacia la izquierda, como podemos ver en la Figura 3.2. Esto coincide con nuestras dos acciones disponibles y, de forma algo ingenua, podemos considerar esto un punto de partida válido.

Por supuesto, esta discretización es demasiado simple para resolver el problema de una forma eficiente; es necesario probar otras divisiones de estados y tener en cuenta las demás observaciones. Puesto que experimentar manualmente cuál de las cuatro observaciones es más descriptiva (y por tanto, hace que nuestro agente diferencie situaciones mejor), crearemos una función parametrizable, la cual decide en cuántos estados dividir cada observación. Así podemos decidir de una forma sencilla a qué observaciones damos más importancia, creando más o menos estados distintos en función a éstas, como podemos apreciar en la Figura 3.3.

Este aumento en el número de estados permite que nuestro agente sea capaz de especializarse más en situaciones concretas. Pero esta mejora viene con un precio a pagar: cuantos más estados distintos tengamos más ocuparán en memoria, llegando al punto de quedarnos sin ella; y más tiempo tardará en visitarlos todos los estados (en repetidas ocasiones) durante su etapa de entrenamiento. Esto quiere decir que nuestro agente tardará más en tener un buen rendimiento. Es importante entonces que evaluemos el problema al

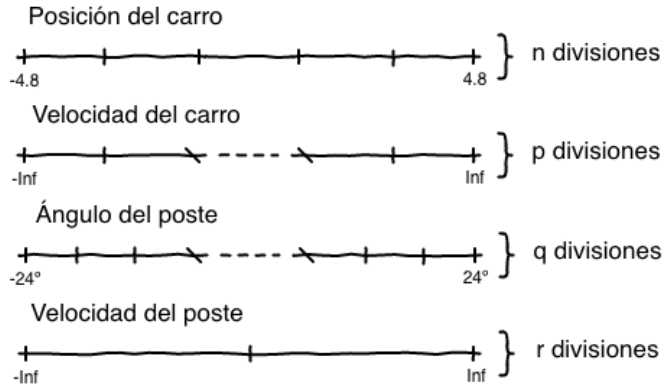

Figura 3.3: División de las observaciones, fuente propia

que nos enfrentamos para encontrar el equilibrio ideal y así poder ahorrar recursos de memoria y tiempo.

3.3.2. Resultados

> CartPole define como solucionado el problema al obtener una recompensa media de 195.0 durante 100 episodios consecutivos.

Para nuestra implementación usamos los siguientes hiperparámetros explicados en la sección 2.2.1 y configuración de estados (Listing 3.2). Es común establecer un mínimo de exploración para que, pese a encontrarnos en un estado muy avanzado de la simulación, siempre tengamos una pequeña probabilidad de ir a parar a un estado nunca visitado que pueda mejorar nuestro resultado.

Siguiendo con esta idea, después de cada episodio reduciremos el hiperparámetro de exploración, en nuestro caso multiplicándolo por `0.99`. De esta forma a medida que los episodios se suceden las decisiones tomadas por el agente se centran más en explotar los caminos ya conocidos en lugar de explorar otros nuevos, facilitando así la convergencia.

Listing 3.2: Hiperparámetros y configuración de estados

```
LEARNING_RATE    = 0.1    # Alpha
DISCOUNT_FACTOR  = 1.0    # Gamma
EXPLORATION      = 0.5    # Epsilon (initial)
MIN_EXPLORATION  = 0.01   # Epsilon (final)

# (cart_pos, cart_vel, pole_angle, pole_vel)
STATE_CONFIGURATION = (1, 1, 3, 6)
```

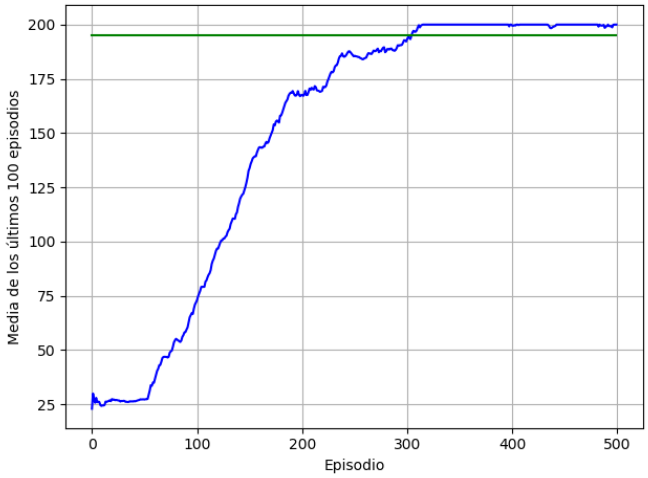

Figura 3.4: Visualización de los resultados del agente RL

Para realizar las mediciones, creamos una cola con una longitud máxima de 100 elementos. En ella añadimos el resultado de cada simulación, de forma que el resultado número 101 elimina el número 1. De esta forma, podemos calcular la media sobre la estructura de datos y comprobar si el problema se consideraría resuelto por OpenAI Gym. Además dibujaremos esta media de la estructura tras cada episodio en una gráfica, con el fin de que los resultados sean más fáciles de apreciar y analizar. Las gráficas resultantes tienen variaciones tras cada ejecución, pero el caso más común es el mostrado en la Figura 3.4.

En dicha gráfica se pueden apreciar tres etapas distintas en el rendimiento del agente:

- Episodios 1-50: durante las primeras simulaciones el agente aún ha de rellenar la tabla-Q. Se encuentra en situaciones nuevas ante las que aún no sabe cómo reaccionar. También hay que tener en cuenta que su rendimiento se calcula en base a la media con los primeros resultados, por lo que en la etapa final de esta fase aún es lastrado por esos primeros episodios.

- Episodios 50-300: suponen el despegue en rendimiento del agente. En ellos se empieza a liberar del lastre de las primeras simulaciones, además de que su tabla interna ya se encuentra en un estado bastante estable y es capaz de generar mejores resultados.

- A partir del episodio 300: el agente alcanza su meta y el algoritmo comienza a estabilizarse.

Como hemos indicado anteriormente, estos resultados han sido obtenidos con una configuración en la discretización de `(1, 1, 3, 6)`. La memoria del agente es una tabla multidimensional de estados, cada observación representa una dimensión y lo que se indica en la configuración es el tamaño de cada dimensión, provistas en el orden: posición del carro, velocidad del carro, ángulo del poste y velocidad del poste respectivamente.

Utilizar `'1'` para las observaciones de la velocidad y posición del carro quiere decir que ignoramos las mismas. Aunque esto pueda parecer contraproducente, durante nuestras simulaciones descubrimos que el principal motivo de finalización de un episodio era la caída del mástil. En la mayoría de las ocasiones el mástil cae mucho antes de que el carro alcance los límites laterales que veíamos en la figura 3.2. Esto nos permite ignorar las dos primeras observaciones (posición del carro y velocidad del carro), reduciendo notablemente la dimensionalidad del problema, con un resultado de tan solo $1 * 1 * 3 * 6 = 18$ estados. De esta forma, el agente aprende mucho antes y no consume tantos recursos.

3.3.2.1. Otras configuraciones

Para comprender mejor el comportamiento del agente, realizamos pruebas con otras configuraciones en la discretización del estado durante simulaciones con un mayor número de episodios. En concreto, probamos la discretización expuesta en la imagen 3.2, la cual es demasiado simple como para conseguir aprender. La memoria del agente es demasiado pequeña (tan sólo dos estados), por lo tanto cuando mejora su rendimiento una situación concreta lo hará a costa de empeorarlo para otras.

Otra configuración probada fue `(1, 2, 3, 8)`, la cuál es más detallada que la usada en 3.4. Por ello, cabe esperar que el agente tarde más en obtener buenos resultados, ya que al haber un mayor número de estados tardará más en visitarlos todos y reforzarlos consecuentemente. Por otra parte, cabría esperar una mayor estabilidad cuando el agente ha entrenado durante mucho tiempo. Los resultados de las pruebas pueden verse en la gráfica 3.5.

3.4. Los límites de la tabla-Q

A lo largo de este capítulo hemos introducido el aprendizaje por refuerzo como método de aprendizaje, las ventajas que tiene sobre otros métodos, sus mecánicas y elementos, su algoritmo de Q-Learning... Pero también nos hemos encontrado ciertos problemas.

El más importante es el que vimos durante la discretización del estado, cómo a medida que aumentan las dimensiones el número de estados se dispa-

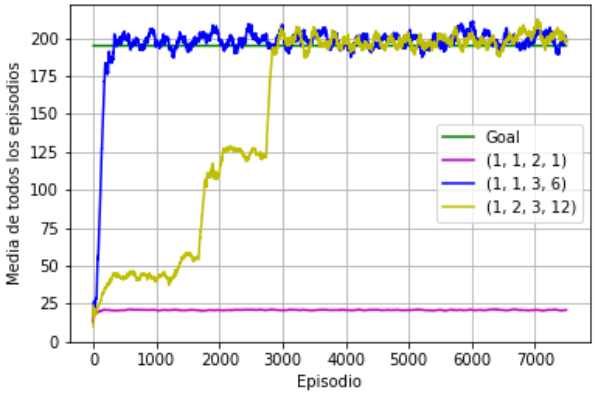

Figura 3.5: Distintas configuraciones de discretización

ra. Este problema es conocido como la **maldición de la dimensión** (*curse of dimensionality*), expresión que fue acuñada por Richard E. Bellman [5] [6] y que en el aprendizaje automático hace referencia al aumento exponencial en el tamaño de las tablas de estados en la memoria de los agentes, aunque también está presente en otros dominios como combinatoria, muestreo, optimización... El problema puede ser contenido hasta cierto punto a través de buenas aproximaciones en la discretización. No obstante, abordar un entorno finito con cierto grado de complejidad puede acabar resultando inviable.

Por ejemplo, en el juego de conecta 4 se dispone de un tablero de 6x7 casillas, las cuales puedes estar vacias, ocupadas por una ficha roja u ocupadas por una ficha amarilla, lo que resulta en una combinación de $3^{(7*6)} = 4531985219092$ estados diferentes [14], convirténdolo así en un problema demasiado grande para ser abarcado mediante el Q-Learning.

Otra solución ha surgido durante los últimos años y consiste en combinar el *aprendizaje por refuerzo* con técnicas de *aprendizaje profundo*, que estudiaremos en los próximos capítulos.

CAPÍTULO 4

Redes Neuronales y Q-Learning

"En otras actividades más allá del puro pensamiento lógico, nuestras mentes funcionan mucho más rápido que cualquier ordenador jamás concebido"
— Daniel Crevier

4.1. Redes Neuronales: definición y elementos

Una rama del aprendizaje automático en auge es la de **aprendizaje profundo** (o *Deep Learning*), englobada dentro de aprendizaje automático, que decidió diseñar estos nuevos modelos basándose en la estructura del cerebro humano, desarrollando de esta forma las primeras neuronas artificiales.

4.1.1. Neuronas

La neurona es el elemento básico de una red neuronal. Al igual que las existentes en el cerebro humano, estas neuronas son células de procesamiento; reciben varias entradas de datos y devuelven una única salida. Cada neurona contiene uno o varios pesos propios, en función del número de variables que reciba, que se relacionan con ellas de la siguiente forma:

$$Y = f(x\ w\ + b)$$

Echando un vistazo a la representación de la neurona en la imagen 4.1, distinguiremos las x como las variables de entrada y w los pesos que les

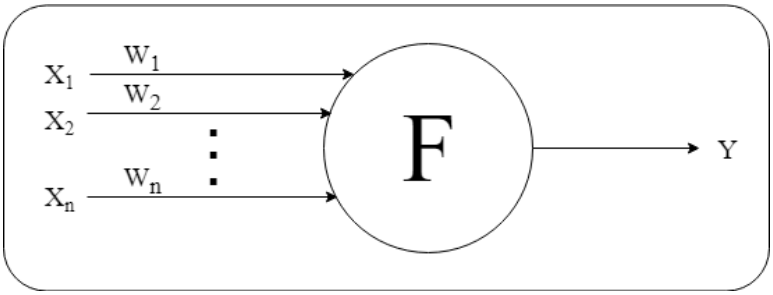

Figura 4.1: Esquema de una neurona artificial

asigna la propia neurona. La variable b es el *bias*, o desplazamiento, que se suma a la multiplicación entre las variables y sus respectivos pesos y f es la **función de activación**, que nos devolverá la salida Y de la neurona.

Sin embargo, dentro de estas neuronas existen varios tipos. Las que nos interesan y trataremos en profundidad son las especializadas en no linealidad, manteniendo ésta característica en su función de activación. A continuación, listaremos las más utilizadas y las que podremos observar en la gráfica 4.2.

- Sigmoide $f(z) = \frac{1}{(1+e^{-z})}$: esta función logit marca una diferenciación clara entre valores pequeños y grandes, agrupándolos en 0 o 1 respectivamente. Entre estos dos extremos, las salidas de las neuronas adoptarán una forma de S.
- Tanh $f(z) = tanh(z)$: este tipo de neurona tiene una salida muy parecida a las tipo sigmoide, manteniendo la forma de S, pero con la diferencia de que el marco está entre -1 y 1. Suele ser usada con más asiduidad debido a que está centrada en 0.
- ReLu $f(z) = max(0, z)$: la neurona de linealidad restringida o ReLu (Restricted Linear Unit) utiliza un distinto tipo de no linealidad. A pesar de su simplicidad, ha sido la función de activación utilizada para la resolución de diferentes problemas debido a su mayor facilidad de convergencia y facilidad computacional [9].

4.1.2. Estructura y funcionamiento

A pesar de que una neurona podría resolver cualquier problema linealmente separable, nos resultaría muy difícil resolver problemas de aprendizaje automático más complejos a través de una sola de ellas. Para ello, de nuevo basándose en el modelo de un cerebro humano, las neuronas se encuentran agrupadas por capas, formando así una red neuronal. Cada capa consta de

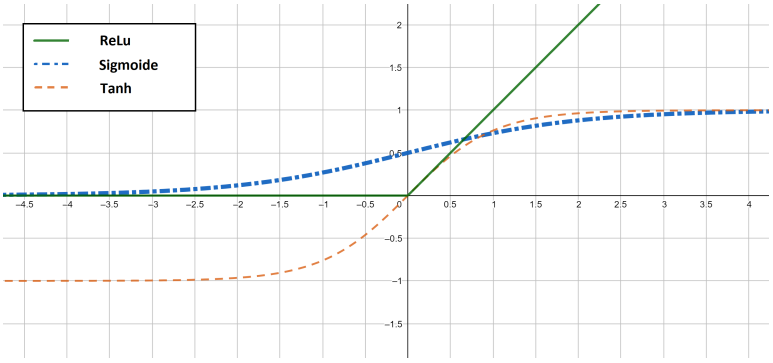

Figura 4.2: Comparación entre funciones Sigmoide, Tanh y ReLu

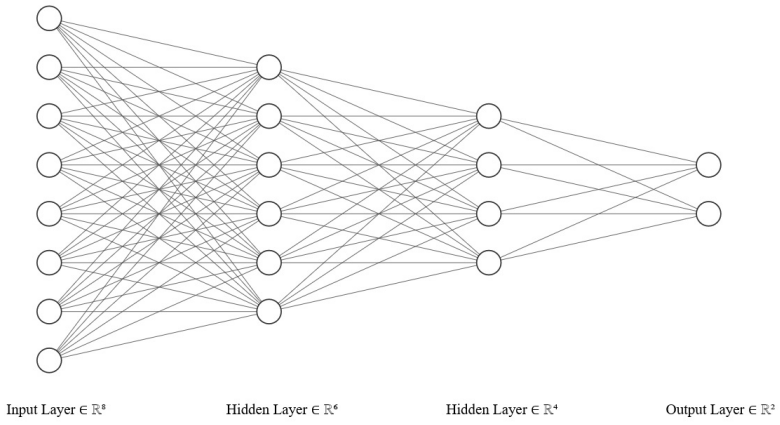

Figura 4.3: Esquema de estructura de una red neuronal profunda

un número finito de neuronas del mismo tipo, recibiendo todas los mismos tipos y cantidades de datos devueltos por las neuronas de la capa anterior pero, como hemos mencionado antes, devolviendo resultados diferentes cada una.

Las redes neuronales están compuestas generalmente por varias capas, de forma que las salidas de la anterior forman las entradas de la siguiente, tal y como podemos observar en la imagen 4.3. Entre ellas, podemos distinguir la capa de entrada, que es la que recibe los datos a analizar, la capa de salida, que es la que devuelve los resultados en el formato que nos interesa en función del problema, y las capas intermedias u ocultas. Éstas son las encargadas de abstraer todos los datos recibidos por la capa anterior para que sea más fácil procesarlos y analizarlos.

Las redes con múltiples capas intermedias son conocidas como **redes**

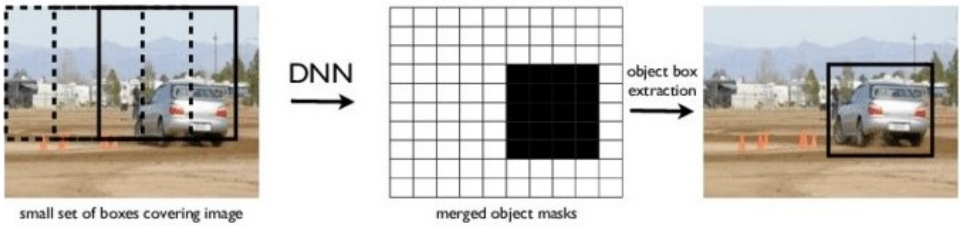

Figura 4.4: Ejemplo de uso de una DNN [35]

neuronales profundas, o *Deep Neural Networks* (DNNs). El ejemplo más conocido para el que podemos utilizar estas redes es el de reconocimiento de objetos en imágenes. Existen múltiples formas de utilizar las DNNs en este ámbito, así que utilizaremos el ejemplo que podemos ver en la imagen 4.4.

La capa de entrada comienza recibiendo un conjunto de píxeles de distintos colores. Las capas intermedias se encargan de distinguir el objeto, ya sea diferenciando los píxeles según sus colores y su proximidad, distinguiendo siluetas o bien reconociendo patrones que puedan distinguir al coche del resto de la fotografía. Sin conocer como tal la forma exacta en que se ha resuelto el problema, podríamos intuir que con una sola capa intermedia nos sería imposible, o realmente difícil, resolver este problema, ya que el proceso requiere varios pasos entre neuronas (por ejemplo, reconocer los faros, las ventanas, la matrícula...). Observando el esquema, la capa de salida sería la encargada de marcar las regiones en las que el coche es visible, o más bien, en las que existe el objeto coche. En otros casos, las capas intermedias también podrían ser utilizadas para disminuir las dimensiones del problema, reduciendo el número de píxeles de una imagen desde una magnitud de miles a una de decenas.

4.1.2.1. Descenso de gradiente y retropropagación

Como ya hemos visto en este mismo capítulo, la neurona dispone de unos pesos para asignar a sus entradas y así producir una salida que nos resulte útil. Sin embargo, el proceso de asignar un valor óptimo a esos pesos no resulta para nada trivial, así que necesitamos un método que nos ayude a encontrar una configuración óptima de todas las neuronas de nuestra red.

El método del **descenso de gradiente** (o *gradient descent*) [8, cap. 2] es un algoritmo de optimización que permite converger hacia el valor mínimo de una función de manera iterativa. En este caso, la función a minimizar es la de error (*loss*), que calcula la diferencia entre la salida estimada (y) y la salida real (t), consiguiendo mejores resultados. Para identificar el mínimo de la función, el método del descenso del gradiente calcula la derivada parcial

respecto a cada parámetro (x) en el punto de evaluación. La derivada indica el valor y la dirección en la que la función crece más rápido (por eso en el entrenamiento mueves los pesos en dirección contraria). No obstante, uno de los problemas que surgen en este punto es que éste puede ser tanto un mínimo local como global, siendo posible que nunca lleguemos a unos valores óptimos.

El resultado de la derivada se resta a cada uno de los parámetros, multiplicando finalmente por la **tasa de aprendizaje** (α). Aunque hemos hablado en capítulos anteriores sobre ella, aplicada a este caso indica lo rápido que converge el algoritmo hasta un mínimo local, manteniendo un valor entre 0 y 1. Es importante ver que la tasa influye mucho a la hora de utilizar el descenso de gradiente y es necesario escoger un valor adecuado para evitar problemas.

El descenso de gradiente resulta útil a la hora de asignar pesos a una neurona, pero para aplicarlo a una red neuronal completa hay que dar un paso más allá. Es por ello que existe un algoritmo conocido como **retropropagación** (o *backpropagation*) [8, cap. 2]. Al igual que en el descenso de gradiente, compara la salida real de la red neuronal con la salida deseada y minimiza su error. El resultado se propaga hacia todas las neuronas de la capa anterior que han contribuido directamente a crear la salida de la red. No obstante, cada una de ellas recibe una fracción de la señal total del error, según la contribución realizada. Este proceso se repite capa por capa, hasta la de entrada, ya que cada una de las capas tiene su propia función de activación y no serviría de nada aplicarles a todas las mismas modificaciones.

4.1.2.2. Optimizadores

Antes hemos hablado de la tasa de aprendizaje, pero no hemos llegado a ver qué supone un menor o mayor valor. Concretamente, la tasa influye sobre todo dependiendo del tamaño del problema: mientras que un ratio pequeño puede aproximarse mejor al error mínimo, en problemas de gran magnitud puede resultar demasiado lento. Con un ratio de aprendizaje demasiado grande, ocurre exactamente lo contrario: aprenderá más rápidamente pero puede resultar muy difícil converger en un mínimo error local.

Para evitar que ocurrieran este tipo de problemas, se estudiaron múltiples algoritmos de optimización, de forma que dicho ratio se pudiera modificar dinámicamente durante el entrenamiento. A continuación explicaremos los optimizadores más utilizados [41]:

- **AdaGrad**: intenta adaptar la tasa de aprendizaje global a partir de la acumulación del historial de gradientes. Para ser específicos, mantiene un seguimiento de la tasa de cada parámetro, de forma que éste sea

inversamente proporcional a la raíz de la suma de los cuadrados del historial de gradiente de todos los estados.

- **RMSProp**: este algoritmo utiliza el de descenso de gradiente con momento que, en resumen, se trata de un descenso de gradiente donde el tamaño de los pasos para reducir el error se modifica dinámicamente, de forma que se pueda converger en un error mínimo sin problemas. Combinando este algoritmo con el historial de gradientes de AdaGrad, obtenemos el optimizador RMSProp.

- **Adam**: mientras que RMSProp utiliza la media de los gradientes que guarda Adagrad (primer momento), Adam utiliza la media de la varianza (segundo momento). Hay que tener en cuenta que, ya que los momentos están inicializados a cero, es necesario un factor de corrección para obtener el valor real de ellos. Adam será el optimizador que utilizaremos, por ser más eficiente y poder adaptarse a la mayoría de problemas [8, cap. 4].

4.1.3. Problemas de clasificación

Los problemas de clasificación consisten, a grandes rasgos, en asignar un dato recibido a una categoría de un conjunto ya establecido. El ejemplo más claro de esta categoría sería proveer de imágenes de perros y gatos a una red neuronal, y que sea ésta la que decida de qué animal se trata en cada caso. Modificando el comportamiento del ojo humano, la red recibiría dichas imágenes como un conjunto de píxeles y a partir ellos, crearía contornos y buscaría patrones para distinguir entre un animal u otro.

A pesar de que no utilizaremos directamente este tipo de problemas en futuras secciones, este ejercicio nos ha servido para entrar en contacto con Keras y comprobar su funcionamiento a la hora de crear redes neuronales. Para este ejemplo hemos elegido un conjunto de datos conocido: reconocimiento de cifras escritas a mano [20]. Éste consiste en un conjunto de 4500 imágenes de diferentes cifras escritas a mano de distintas formas, como se ve en la imagen 4.5, incluyendo la respuesta correcta para comprobar los resultados de nuestro clasificador.

Para este caso, hemos establecido una red neuronal de tres capas, la de entrada, la de salida y una capa intermedia u oculta para que la red realice los cálculos necesarios. Hemos utilizado un optimizador Adam, de forma que el descenso de gradiente se realice de forma más efectiva y los valores de los pesos en la red se reajusten más rápidamente. Además, puede regular automáticamente el valor de la tasa de aprendizaje, comenzando en un valor de 0.001.

Tanto la capa de entrada como la oculta, son de tipo *ReLu*. La capa de

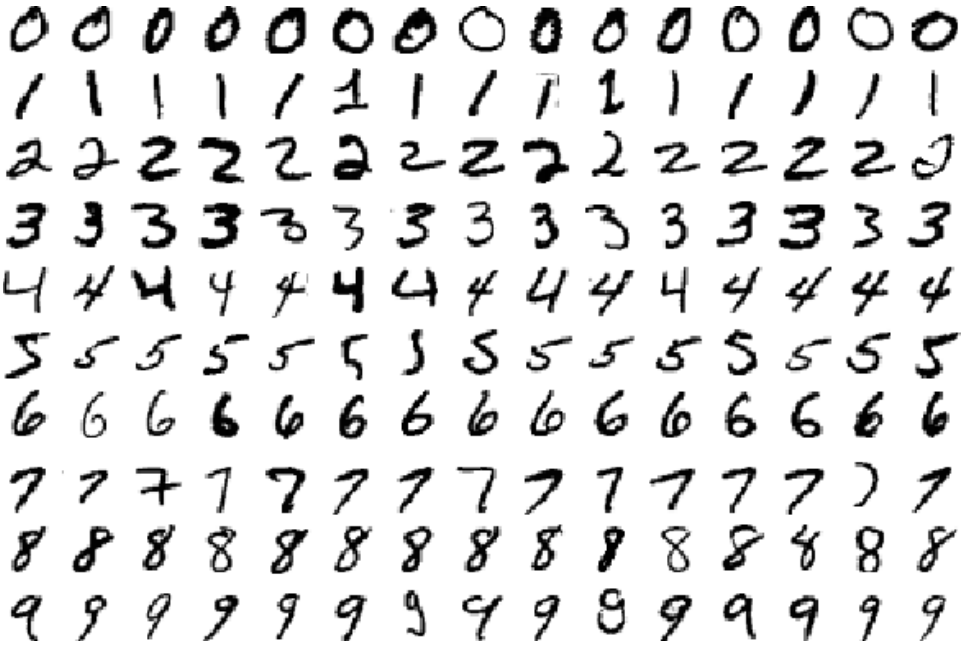

Figura 4.5: Imagen del conjunto de datos de cifras escritas a mano de MNIST [19]

entrada se encarga de procesar los 400 píxeles que representan la imagen de una cifra escrita a mano. A modo de salida, hemos utilizado un tipo especial de capa llamado *softmax*. En ella, se da la condición de que la suma de todas las salidas sea igual a 1. De esta forma, las probabilidades que reflejan las distintas salidas se encuentran normalizadas y son excluyentes entre sí, dejando que haya una probabilidad que destaque por encima de las demás en su conjunto; ésa será nuestra predicción.

Además, para comprobar que nuestra red neuronal entrena y aprende correctamente, hemos utilizado validación cruzada en K-iteraciones (**K-Fold cross-validation**). La técnica de validación cruzada consiste en dividir en varios grupos el conjunto de datos proporcionado y entrenar varias veces el modelo sobre ellos, alternando la función de dichos grupos entre entrenamiento y prueba. El algoritmo de *K-Fold* nos permite barajar el conjunto de datos y dividirlo en k grupos, todos destinados a entrenamiento, a excepción de uno que se reserva para la prueba. De esta forma, podemos comprobar las distintas formas en las que ha aprendido, en función de los grupos formados durante la aplicación del K-Fold.

Como podemos ver en la imagen 4.6, las distintas agrupaciones presentan una precisión distinta entre sí, por lo que nos quedaremos con la más óptima como resultado final. Siendo un resultado tan preciso podemos concluir que

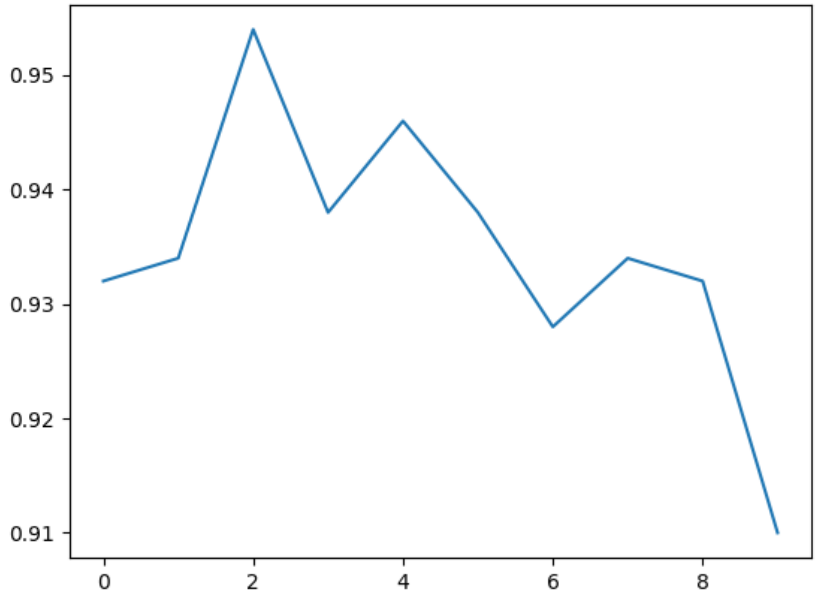

Figura 4.6: Comparación de la precisión de cada una de las iteracciones del k-fold

la técnica es claramente efectiva. Todo esto nos ha servido para aprender el uso adecuado de las redes y la importancia de los datos a tratar, tanto los de entrenamiento como los de prueba, lecciones que nos serán de gran utilidad para el tipo de problema que realmente nos interesa: regresión.

4.1.4. Problemas de regresión

Los problemas de regresión se distinguen de los de clasificación en que la respuesta de la red neuronal no pertenece a un grupo, sino que se intenta predecir un valor real. La forma de entender este tipo de problemas sería tomando como ejemplo una red neuronal que sea capaz de predecir el precio de una casa en función de su tamaño, número de habitaciones, localización, etc.

Para este ejemplo, hemos utilizado el conjunto de datos **Boston Housing**, incluido en uno de los paquetes de Keras [10]. Éste consta de 13 características distintas de un número determinado de viviendas (506 casos), a partir de las cuales se puede averiguar su precio.

Esta vez los datos son muy diversos y los atributos pueden representar

tanto unidades en escalas de miles, como de unos o ceros, para representar si una característica se da o no. Por ello, antes de crear la red neuronal, es necesario estandarizar los valores mediante la función *StandardScaler* de *sklearn*. De esta forma, todos los atributos estarán normalizados y la red podrá trabajar con ellos de forma más eficiente.

La red neuronal se compone de tres capas, igual que en el ejemplo visto de clasificación, utilizando el optimizador Adam. De nuevo, las capas de entrada y la oculta, serán de tipo **ReLU**, con un tamaño de 13 y 26 neuronas cada una. La capa de salida consta de una única neurona, de tipo **ReLU**, de forma que nos devolverá directamente el precio predicho por la red neuronal. La fórmula de diferencia de gradiente para este problema quedaría expresada como la siguiente:

$$\Delta w_k = \sum_i \alpha \; x_k^{(i)} |t^{(i)} - y^{(i)}|$$

Así que con un error final tan bajo podemos confirmar que la red predice correctamente los precios. Además, para asegurarnos de la fiabilidad de la red neuronal y comprobar que no se ha dado una situación de **sobreentrenamiento** (*overfitting*), en el conjunto de datos disponemos dos partes separadas, una para entrenamiento y otra para la prueba de validación, en una proporción de 80 % frente a un 20 %, respectivamente. De esta forma, mientras entrenamos la red con la primera, se puede comprobar la diferencia de errores con la segunda. Como podemos apreciar en la imagen 4.7, los resultados en la validación son algo peores que en el entrenamiento, lo cual indica que no estamos cometiendo sobreentrenamiento y que la red aprende correctamente.

En los casos de regresión haremos uso del **error cuadrático medio** (o *ECM*), el más utilizado de los múltiples métodos de cálculo de error existentes. En nuestro caso, $Ypred$ se refiere al valor calculado por la red neuronal mientras que $Yreal$ es el valor real del precio de las casas. El cálculo del *ECM* nos indicará la diferencia entre ambos, pudiendo así comprobar lo cerca que han estado las predicciones de la red neuronal de los valores reales.

$$ECM = 1/n \sum_{i=1}^{n} (Ypred_i - Yreal_i)^2$$

4.2. Redes Neuronales y Q-learning

Al hablar de Aprendizaje por Refuerzo anteriormente, nos centramos en el método de Q-Learning. Observamos que las soluciones planteadas eran válidas, pero el mayor problema que presentaba se debía al tamaño de las

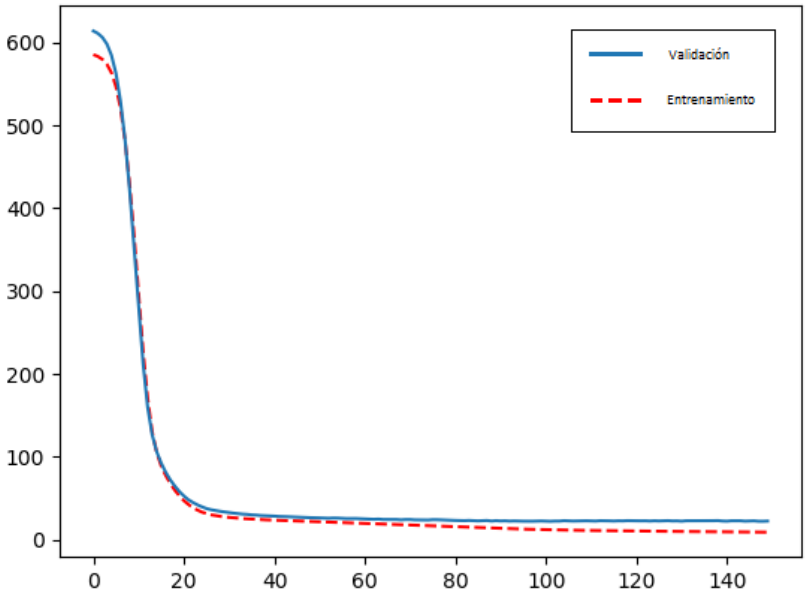

Figura 4.7: Disminución del error sobre el entrenamiento respecto a la validación

tabla-Q: ésta terminaba con tamaños intratables incluso con problemas de complejidad baja.

Buscando la forma de arreglar este problema, descubrimos la posibilidad de realizar aproximaciones a la función-Q a través del uso de una red neuronal, de forma que no fuera necesario recorrer todos los pares estado-valor constantemente. Es así como surgieron las Redes-Q Profundas (*Deep Q-Networks* o **DQNs**, como nos referiremos a ellas).4.8

Las transiciones creadas a partir de las DQNs se pueden representar como un vector de valores de la siguiente forma:

$$Q(s, \cdot\,; T)$$

Donde T son los parámetros de la red y $s, \cdot$ son todos los estados, especificando s como el estado actual. Esto significa que la red es capaz de calcular múltiples valores-Q de forma dinámica para todas las acciones posibles en un estado, sin abandonar la representación que suponía la tabla-Q. Además, permite generalizar estados no visitados pero sí muy parecidos a otros que ya ha visitado, ya que la red ya tendrá calculadas las decisiones para éstos

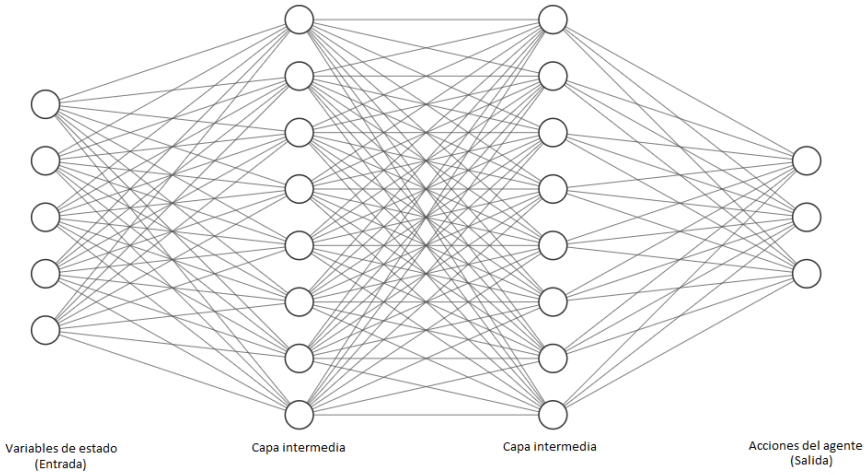

Figura 4.8: Ejemplo simplificado de la red de una DQN

también.

Podemos ver el funcionamiento de las DQNs en el pseudo-código 4.1, sin embargo esta es una versión bastante simplificada y para conseguir un verdadero éxito habrá que probar otras técnicas a mayores que veremos en el siguiente capítulo.

Listing 4.1: Pseudocódigo DQN

```
for i in range(EPISODES):
  while not done:
    # Choose between a random choice (exploration)
    # or the best valued action (exploitation)
    action = agent.predict_action(state)

    # Perform the action
    next_state, reward, done = env.step(action)
    # Remember the result
    agent.update_model(state, action, reward, next_state, done)

    # Apply the Bellman's equation with a set of
    # randomly selected memories (experience replay)
    agent.replay()

    state = next_state
    # Decrease exploration ratio
    agent.update_hyperparameters()
```

CAPÍTULO 5

DQNs en acción

"La vida sólo puede entenderse hacia atrás, pero debe vivirse hacia adelante"
— Søren Kierkegaard

5.1. CartPole

El primer problema que trataremos al trabajar con aprendizaje por refuerzo profundo se corresponde con el de CartPole, ya descrito en el apartado 3.3, en el que jugamos con un carro cuyo objetivo es moverse a derecha o izquierda para evitar que el poste vertical que hay sobre él se caiga. Todas las técnicas aplicadas para resolverlo han sido implementadas desde cero y su código se encuentra en el repositorio de nuestro proyecto.

5.1.1. Reemplazar la tabla-Q: `SimpleAgent`

Como primer paso para la resolución del problema, sustituimos la tabla-Q y su consecuente discretización de estados por una red neuronal como la de la figura 5.1, abandonando por fin los problemas y limitaciones que habíamos observado anteriormente. La red recibe como entrada el **estado** actual (posición del carro, velocidad del carro, posición del péndulo y velocidad del péndulo) y cuenta con dos capas ocultas tipo *ReLu*, de 24 neuronas. La última capa, de tipo *Linear* produce dos salidas que son los valores Q aproximados por la red neuronal, asociados a cada una de las acciones disponibles: ejercer fuerza hacia la izquierda o hacia la derecha. Después de ejecutar cada

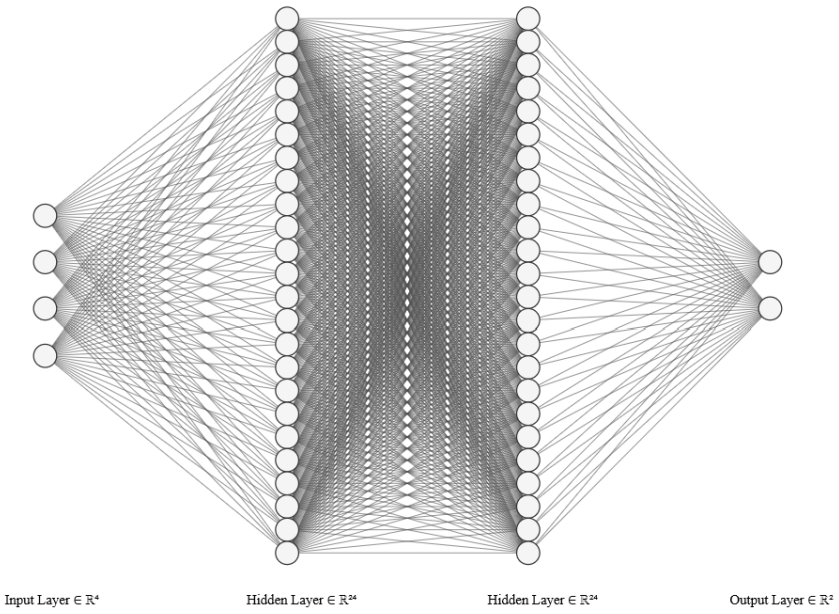

Figura 5.1: Estructura de la red neuronal en `SimpleAgent`

movimiento, se aplica la ecuación de Bellman para entrenar la red e intentar que el agente aprenda a jugar. El optimizador que utilizamos es el Adam, debido a su popular uso en este tipo de problemas.

En el fragmento de código 5.1 podemos ver que la implementación del algoritmo es muy similar a la usada en 2.3, con unas modificaciones que describiremos a continuación.

Listing 5.1: Pseudocódigo SimpleAgent

```
state = env.reset()

while not done:
  action = agent.predict_action(state)
  next, reward, done = env.step(action)
  agent.learn(state, action, reward, next, done)
  state = next
```

Para **predecir acciones** utilizamos la red neuronal, a la cual introducimos el estado actual para que nos devuelva los resultados de ambas acciones, de los que nos quedaremos con el mejor. Una vez ejecutada la acción disponemos del estado siguiente y la recompensa, lo cual podemos usar para **entrenar** la red neuronal de nuevo.

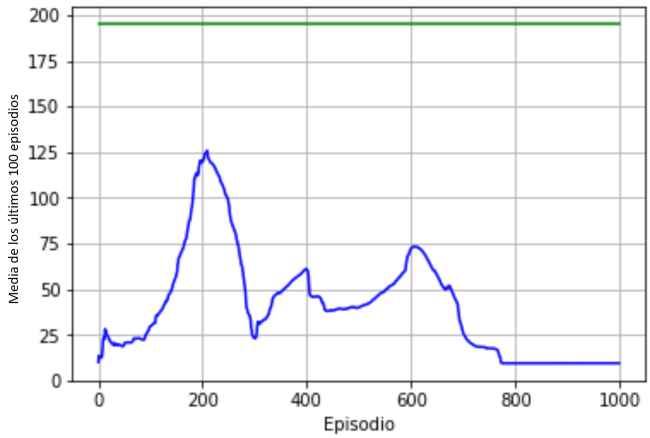

Figura 5.2: Resultados del `SimpleAgent`

Como cabía esperar con una modificación tan ingenua, los resultados no son buenos, tal y como podemos apreciar en la imagen 5.2. El agente no logra aprender de una forma estable. Si miramos atrás, en nuestra solución con una tabla-Q, el único estado que es modificado tras un paso de simulación es aquel en el que nos encontramos. Por otra parte, cuando trabajamos con una red neuronal esto no es posible. Entrenar una red neuronal implica que toda la red se modificará, buscando satisfacer el nuevo ejemplo de entrenamiento proporcionado [8, cap. 2]. En otras palabras, el agente mejora su rendimiento en el estado actual, pero al mismo tiempo empeora en todos los demás.

Los hiperparámetros usados también son una ligera modificación de los usados en la solución de aprendizaje por refuerzo. Como puede verse en 5.2, el mayor cambio es en la tasa de aprendizaje, la cual debe ser mucho menor para intentar evitar grandes modificaciones en la red.

Listing 5.2: Hiperparámetros

```
LEARNING_RATE = 0.0001    # Alpha
DISCOUNT_FACTOR = 0.95    # Gamma
EXPLORATION = 0.5         # Epsilon (initial)
MIN_EXPLORATION = 0.01    # Epsilon (final)
```

5.1.2. Añadir memoria al agente: `BatchAgent`

Una forma de conseguir estabilidad es añadir una memoria adicional al agente, en la que se almacenen las decisiones tomadas con los resultados obtenidos, de forma que a la hora de entrenar la red se puedan usar varias experiencias en lugar de sólo una. En nuestra memoria guardamos todos los

datos necesarios para la perfecta representación de lo sucedido en pasos de ejecución anteriores, lo que se traduce en: **estado-actual**, **acción-tomada**, **recompensa**, **estado-siguiente** y **finalizado**.

Listing 5.3: Pseudocódigo función Replay de BatchAgent

```
# Create a iterator to go through the memory
# starting in the last position
iter = reversed(memory)

for _ in range(BATCH_SIZE):
    state, action, reward, next_state, done = next(iter)

    if done:
        q_update = -reward
    else:
        q_update = bellman(reward, next_state)

    target = model.predict(state)
    target[action] = q_update

    model.fit(state, target)
```

Tras la ejecución de cada acción, y con la información proporcionada por el entorno en ese momento, guardamos todos estos valores en la memoria para ser utilizados posteriormente. De esta forma contamos con un conjunto (*batch*) de los últimos N pasos guardados, para que podamos entrenar la red neuronal con un mayor número de casos de prueba, intentando evitar así la inestabilidad del `SimpleAgent`.

El problema es que los pasos tomados dentro de estos *batch* están fuertemente cohesionados entre sí, ya que al ser consecutivos la información aportada por cada uno de ellos es muy similar. La memoria de estados consecutivos provee de más información a la red pero las similitudes entre ellos provocan que el agente aprenda muy lentamente, de forma insuficiente para la resolución del problema, como se puede ver en la imagen 5.3.

5.1.3. Romper la cohesión de la memoria: `RandomBatchAgent`

Para solucionar el problema de la cohesión en el `BatchAgent` tendremos que, en lugar de utilizar las últimas N experiencias, elegir experiencias pasadas suficientemente diferentes para que el entrenamiento de la red no esté sesgado hacia un tipo de situación concreta. Tan sólo necesitaremos que la memoria que habíamos utilizado antes sea lo bastante grande como para albergar experiencias de varias partidas. Con este repertorio ya preparado, la mejor forma de romper la cohesión de los datos consiste en seleccionar elementos aleatorios dentro del mismo, a diferencia de tomar una muestra

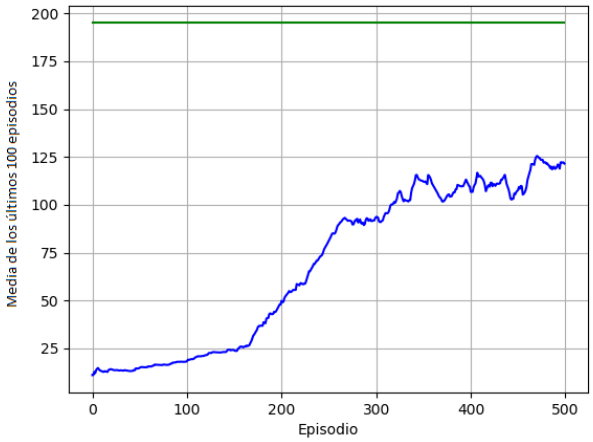

Figura 5.3: Resultados del `BatchAgent`

de elementos seguidos como habíamos hecho anteriormente. Con esta solución, el agente debería tener una muestra más amplia, variada y significativa respecto a los posibles estados en los que se puede encontrar y así rendirá mejor en la mayoría de las situaciones.

Listing 5.4: Pseudocódigo RandomBatchAgent

```
# We get a random sample of past experiences
batch = random.sample(self.memory, BATCH_SIZE)

for experience in batch:
  state, action, reward, next_state, done = experience

  # We calculate the new Q-value
  if done:
    q_update = -reward
  else:
    q_update = bellman(reward, next_state)

  # We update the Q-value of our action...
  target = model.predict(state)
  target[action] = q_update

  # ...and try to approximate it
  model.fit(state, target)
```

El resultado es notoriamente mejor, como muestra la imagen 5.4. El agente consigue resolver el problema de forma rápida, además de conseguir puntuaciones mucho más altas en general. Sin embargo, entre unas simulaciones

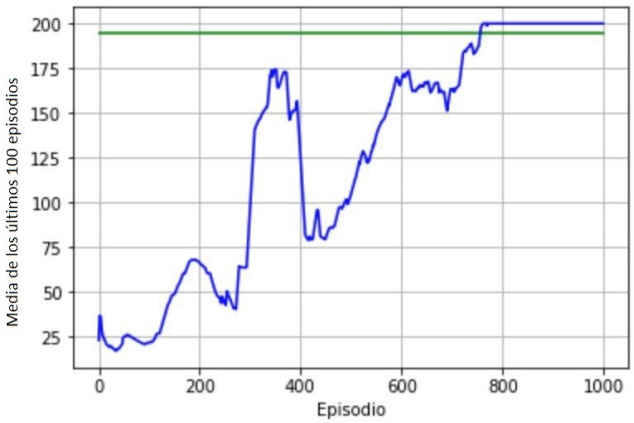

Figura 5.4: Resultados del `RandomBatchAgent`

y otras los resultados son muy dispares, desde encontrar rápidamente y reforzar un camino "bueno" hasta lograr la resolución del problema, hasta no encontrar nunca un camino válido. En éste último caso puede deberse a que las muestras tomadas de la memoria no son lo suficiente variadas y el estado de la red neuronal se degrada progresivamente.

El problema del volumen de datos estaba solucionado, ya que en varias ocasiones el agente había conseguido aprender una solución, y en las que no, al menos daba la impresión de que iba por buen camino. No obstante, la volatilidad de los resultados nos da a entender que el modelo propuesto aún no estaba completo.

5.1.4. Estabilizar la red: `DoubleAgent`

Con el fin de en estabilizar el aprendizaje, intentando evitar esos escollos que el agente puede experimentar en algunas ejecuciones, surge por parte de *DeepMind* el concepto de la **Target Q-Network** [16]. Éste implica la utilización de una segunda red objetivo (*target*), como copia de la red principal de predicción. A través de la fórmula siguiente, podemos ver que la red objetivo (Y) sólo se actualiza cada cierto número de pasos (t como paso actual), tomando el mejor resultado de la red de predicción junto con la siguiente recompensa (R), ahorrándose los cambios innecesarios en los pesos que ésta sufre constantemente y consiguiendo así más estabilidad en los resultados.

$$Y_t^Q \equiv R_{t+1} + \gamma max_a Q(S_{t+1}, a; \theta_t)$$

Como derivación de este método existe otro conocido como **Double Q-Network**, en esta, se cuenta con dos redes que llamaremos objetivo (*target*)

y agente (*agent*). La red objetivo se utiliza para guardar los estados con los que tomar las decisiones en cuanto a los movimientos a ejecutar, mientras que la red de aprendizaje, agente, se encarga de guardar los valores de la ejecución actualizados. De la misma forma que antes, cada cierto número de pasos (t como paso actual) ambas redes son comparadas y se guarda la que haya dado mejores resultados como objetivo (Y), junto con su siguiente recompensa (R), para la toma de decisiones, y se utiliza la otra para el aprendizaje.

$$Y_t^Q = R_{t+1} + \gamma Q(S_{t+1}, argmax_a Q(S_{t+1}, a; \theta_t); \theta_t)$$

El principal inconveniente que mostraba nuestro agente hasta el momento se debía a la gran diferencia de resultados entre las ejecuciones, dando unos resultados que era necesario estabilizar. Tomando como referencia estas soluciones, optamos por el uso de dos redes diferentes.

Listing 5.5: Pseudocódigo DoubleAgent

```
state = env.restart()

while not done:
  action = target.predict_action(state)
  next, reward, done = env.step(action)

  agent.remember(state, action, reward, next, done)
  # Takes the N random memories and trains with them
  agent.learn_by_replay()

  state = next

# If the agent's performance surpasses the target's
if agent.performance() > target.performance():
  # the agent becomes the target to improve his solution
  target = agent
```

Tras la implementación de este método, comprobamos que existe una mayor consistencia en los datos entre unas simulaciones y otras, obteniendo resultados menos dispares que con el entrenamiento por conjunto aleatorio, como podemos observar en la imagen 5.5.

Gracias a este método se han logrado grandes hitos en el campo del aprendizaje por refuerzo profundo. Por ejemplo, hacer que una red neuronal sea capaz de jugar a juegos de Atari a nivel profesional utilizando tan solo los píxeles como entrada [21]

Por otro lado, este método no es perfecto, en la imagen 5.5 se puede ver una pequeña caída al final de la ejecución. Inicialmente achacamos este comportamiento a una desafortunada conjunción probabilística. Pero tras in-

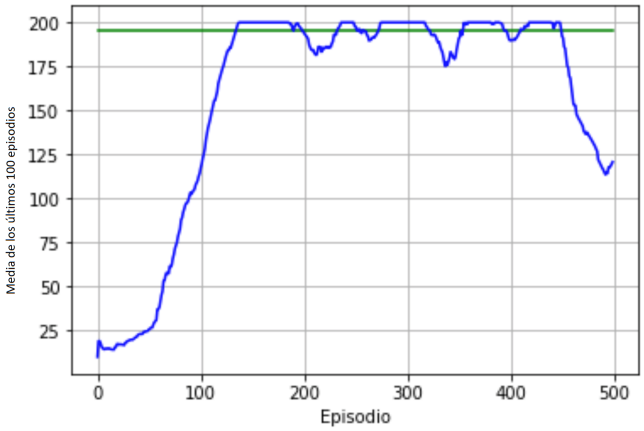

Figura 5.5: Resultados del `DoubleNetworkAgent`

vestigarlo, descubrimos que se trataba de un fenómeno mucho más cotidiano: el causante era el propio método de Deep Q-Learning. Se ha observado cómo algoritmos basados en Q-Learning que buscan aproximar la función Q con modelos no lineales, como una red neuronal, sufren de esta inestabilidad. A este problema nos enfrentaremos más a fondo en el siguiente capítulo, pero en el apartado de CartPole nos damos por satisfechos.

5.2. MountainCar

Con el fin de comprobar la versatilidad de nuestro agente, lo pondremos a prueba en un entorno distinto. En este caso será el problema de MountainCar, descrito por primera vez por A. Moore [22], y con el siguiente enunciado en OpenAI:

> Un coche se encuentra en una pista unidimensional, posicionado entre dos "montañas". El objetivo es alcanzar la cima de la montaña derecha; sin embargo, el motor del coche no es lo bastante potente como para alcanzar la cima de la montaña con un único impulso. Por ello, la única forma de lograrlo es conducir hacia delante y hacia atrás aumentando el impulso.

5.2.1. Especificación del problema

- **Estado:** el estado de este problema nos viene dado a través de la posición respecto al eje de abscisas y la velocidad del coche.

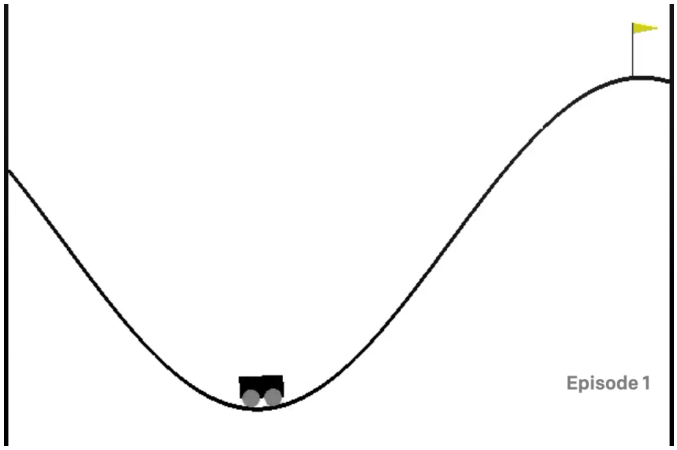

Figura 5.6: Entorno de simulación MountainCar, Brockman et al. [7]

- **Recompensa:** la recompensa recibida será de `-1` por cada paso en el que el coche no alcance la meta.
- **Terminación:** el episodio terminará con éxito cuando el coche alcance la meta, o fracasará si no lo logra tras 200 pasos.
- **Acciones:** dispondremos de tres acciones disponibles, impulsarse hacia la derecha, impulsarse hacia la izquierda y no hacer nada. En cada situación el agente deberá elegir la opción que más impulso le ayude a acumular.

A primera vista puede parecer que es un problema muy parecido a CartPole; no obstante, tiene detalles que lo convierten en un problema especialmente interesante. En particular, su función de recompensa: recibir `-1` en cada paso, combinado con el hecho de que la simulación termine tras sólo 200 pasos si no se alcanza la meta, hace que prácticamente todas las simulaciones terminen con un resultado de -200. Esto elimina el factor de progresión del que disponíamos en CartPole, donde el agente generalmente iba obteniendo recompensas pequeñas, pero cada vez mayores.

Para resolver este problema partiremos de la versión más estable de nuestro agente desarrollado en 5.1.

5.2.2. Enfoques de resolución

Ejecutando el problema con el agente descrito en 5.1.4 y los parámetros usados en 5.2, obtenemos como resultado que el agente no aprende nada en absoluto. La topología interna de la red se mantiene igual, adaptando el

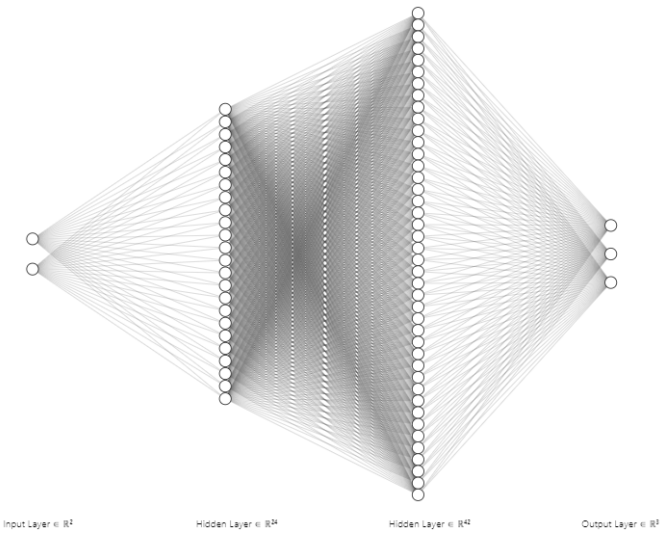

Figura 5.7: Topología MountainCar

número de entradas y salidas a las del problema; CartPole describía su estado a través de cuatro variables y tenía dos acciones disponibles, mientras que MountainCar dispone de dos variables y tres acciones que elegir.

Los resultados tan pobres obtenidos puede que sorprendan, viendo el rendimiento demostrado en el problema de CartPole, pero recordemos que el tamaño de las capas internas de la red se eligió para ese problema en concreto. Es de esperar que distintos escenarios necesiten redes de distinto tamaño para que sean capaces de interpretar las observaciones de manera efectiva. En definitiva, nuestra red actual no es lo bastante descriptiva para la complejidad del problema.

5.2.2.1. Nueva topología e hiperparámetros

Tras modificar al agente para usar la topología de red visible en la imagen 5.7, utilizar los siguientes hiperparámetros (Listing 5.6) y dejar al agente aprendiendo durante un largo entrenamiento, conseguimos que llegue a ser capaz de resolver el problema.

Listing 5.6: Hiperparámetros MountainCar

```
LEARNING_RATE = 0.0001   # Alpha
DISCOUNT_FACTOR = 0.95   # Gamma
EXPLORATION = 0.5        # Epsilon (initial)
MIN_EXPLORATION = 0.01   # Epsilon (final)
```

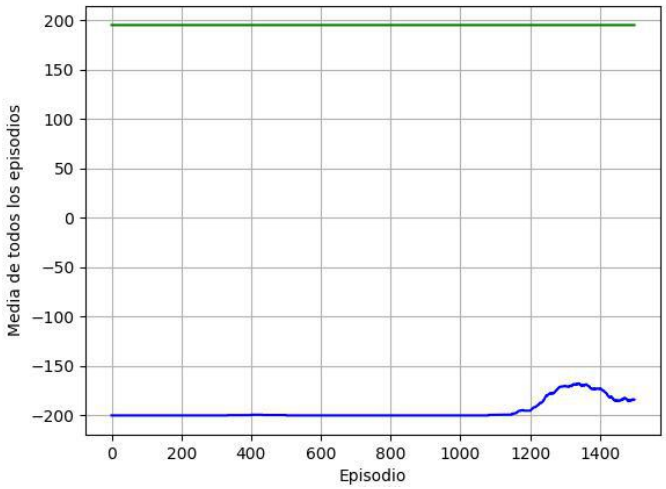

Figura 5.8: Resultados de la nueva topología

La gráfica 5.8 representa las puntuaciones obtenidas durante nuestra simulación. En este punto de desarrollo, la puntuación representa la inversa de los pasos que ha tardado el avatar en llegar a la meta, dado que la recompensa que da el entorno para el estado objetivo es 0 y -1 a cualquier otro, la puntuación -200 implica que la meta no se ha alcanzado y la puntuación -199 implica que ha llegado en el paso 200.

No obstante, creemos que este resultado tiene un gran margen de mejora, por lo que seguiremos explorando otras soluciones que permitan al agente aprender de una forma más eficiente.

5.2.2.2. Función de recompensa

El primer obstáculo a superar para acelerar la curva de aprendizaje, es la función de la recompensa. La recompensa proporcionada por el entorno no es lo bastante útil para el agente. Podríamos entenderlo como el siguiente ejemplo: al hacer un examen, sin importar las veces que se intente o las respuestas que se den, si no está todo contestado de forma correcta la nota obtenida siempre es cero. Viéndolo así, resulta casi imposible identificar qué puntos en particular es necesario reforzar.

Nuestra solución, basada en la propuesta en [37], pasa por sustituir la recompensa dada por el entorno, reemplazando ese resultado binario de *meta* o *no meta* por algo más descriptivo, que empuje al agente a mejorar su funcionamiento. Utilizando la **velocidad** o la **posición** del agente como recompensa, motiva al mismo a tomar las decisiones que lo llevan a moverse

Figura 5.9: Visualización de la recompensa modificada

más lejos, hasta que finalmente termine alcanzando la meta.

Fijándonos en la imagen 5.9, podemos ver que usando una recompensa modificada favorecemos que el coche se mueva e intente llegar cada vez más lejos, a diferencia de la recompensa original con que el coche solía quedarse estancado en la parte más baja de la curva.

A continuación, evaluaremos en mayor detalle cada una de las opciones disponibles a la hora de reemplazar la recompensa, para ver cuál se adaptará mejor al problema y nos proporcionará mejores resultados.

Posición

La posición es el primer candidato para utilizar como recompensa. Al fin y al cabo, si conseguimos que el coche avance lo suficiente, terminará llegando a la meta. Hicimos una serie de pruebas en las que directamente reemplazábamos el valor de la recompensa por el valor de la posición. Para ser capaces de apreciar en qué momento se alcanzaba la meta, le otorgamos una recompensa significativamente más grande cuando lo conseguía, con la esperanza de que esto reforzase mucho más ese comportamiento.

Como podemos ver en la gráfica 5.10, el rendimiento sigue siendo muy parecido al visto anteriormente en la imagen 5.8. La repentina subida de puntuación en torno al episodio 1400, se traduce en que el coche ha con-

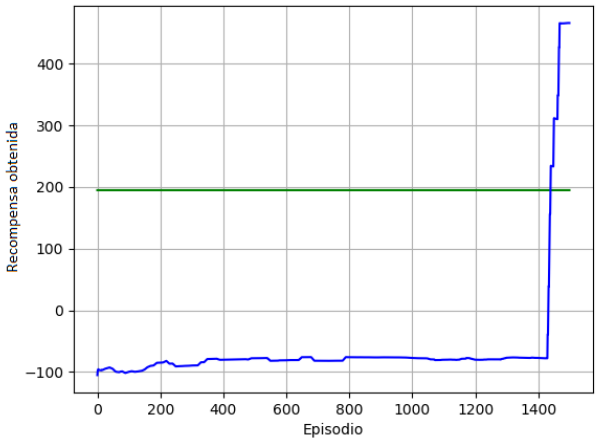

Figura 5.10: Visualización de la recompensa en función de la posición

seguido alcanzar la meta, la cima de la montaña derecha. A partir de los nuevos resultados podemos hacer las siguientes observaciones:

- Nuestra nueva recompensa sólo empuja al agente a intentar escalar la rampa de la derecha, ya que es la dirección que hace crecer el valor de la `posición`. No obstante, para conseguir ascender por la derecha también necesita tomar impulso desde la izquierda, lo que se corresponde con valores negativos de la posición. Como solución, deberemos considerar usar una posición definida con valores absolutos para conseguir un mejor resultado.
- Otro detalle observado es que la posición central no se corresponde con `0`, sino con `-0.5`. También tendremos que corregir esta desviación.

Teniendo en cuenta estos nuevos factores, los resultados obtenidos pueden verse en la gráfica 5.11. Claramente, conseguimos que el problema se resuelva mucho más rápido: 400-500 episodios frente a los 1400 anteriores. Sin embargo la estabilidad aún sigue lejos de parecerse a la vista en el problema de CartPole. Siguen existiendo situaciones en las que el agente consigue resolver el problema en algunas partidas, pero luego vuelve a fracasar en las siguientes.

Velocidad

Otra opción explorada es la de usar la velocidad como recompensa. Al fin y al cabo, cuanta más velocidad consiga acumular el coche, más alto llegará

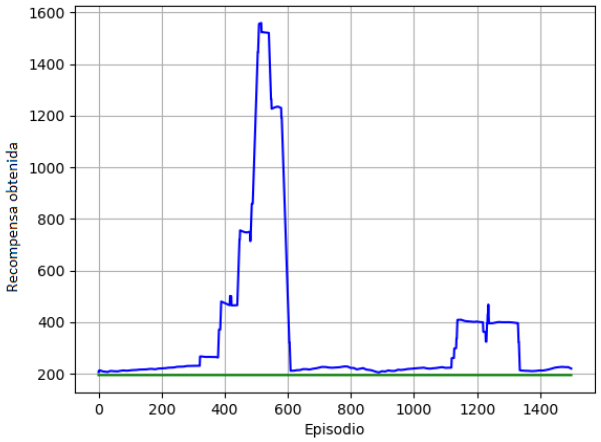

Figura 5.11: Resultados finales de la posición como recompensa

subiendo la rampa.

A diferencia de la posición, la velocidad devuelta por el entorno tiene unos valores muy pequeños, por lo que basar la recompensa en la velocidad recibida sería semejante a darle una recompensa constante. Cuando alcanzase la meta, sería necesario proporcionarle una notablemente mayor, lo que nos dejaría en una situación muy similar a la vista en 5.8. Por ello decidimos multiplicarla por 100, haciéndola más visible y relevante para el agente.

Como vemos en la imagen 5.12, nos encontramos con un comportamiento muy similar al visto usando la posición en valores absolutos. El agente consigue resolver el problema en varias ocasiones pero no se estabiliza.

Otras pruebas

Puesto que los resultados obtenidos por la posición y velocidad parecían haberse estancado, decidimos investigar otras piezas clave de nuestro modelo, como el optimizador. Empezamos a cuestionarnos si Adam [18] era nuestra mejor opción, e hicimos algunas pruebas con Adadelta [41].

Para poder comparar lo mejor posible los resultados, decidimos crear una serie de gráficas para cada ejemplo, de forma que no sólo se apreciara mejor la evolución general de los resultados, sino que se añadieran otros datos de interés, como la recompensa original del entorno, la velocidad máxima, o la posición máxima alcanzada. Así seríamos capaces de controlar en una sola simulación todas las variables.

Se puede ver claramente en los resultados de Adadelta (figura 5.14) con-

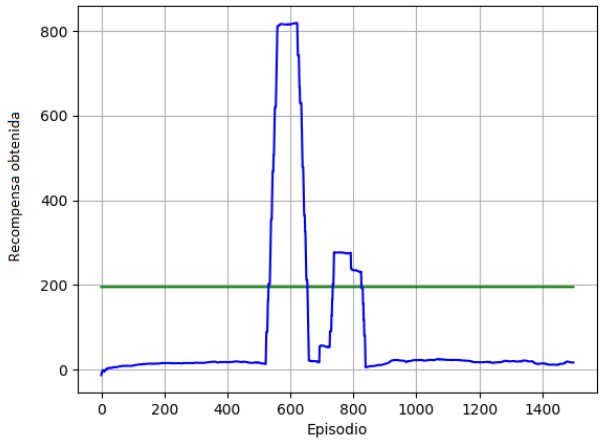

Figura 5.12: Resultados finales de la velocidad como recompensa

sigue alcanzar el objetivo varias iteraciones antes que Adam (figura 5.14), y ambas se mantienen casi totalmente estables durante los siguientes episodios.

La imagen cuenta con tres gráficas:

- La primera muestra la evolución general del sistema en función de los episodios. Se pueden ver los resultados de las dos redes neuronales, la azul para agente y la cian para el aprendiz. La línea roja representa la recompensa real que devuelve el entorno. La negra muestra la recompensa media de las cien ultimas partidas. Así tenemos una observación general y fácil de interpretar.

- La gráfica inferior izquierda muestra la posición máxima alcanzada durante la partida. Esto nos permite ver lo cerca que se ha quedado el agente de alcanzar la meta.

- La gráfica inferior derecha representa las puntuaciones obtenidas por ambas redes en una simulación al final de la ejecución, con el fin de comprobar si realmente han aprendido. Esta simulación se realiza utilizando la recompensa original, simplificando su interpretación.

Dado que seguíamos encontrando fluctuaciones en el aprendizaje de algunas ejecuciones y continuábamos en busca de una mayor estabilidad, decidimos añadir una tercera red neuronal que permitiera almacenar el mejor modelo encontrado hasta el momento, como una copia de seguridad, introduciendo así una especie de elitismo entre los episodios. El objetivo de dicho modelo es intervenir eventualmente para medir la progresión del aprendizaje

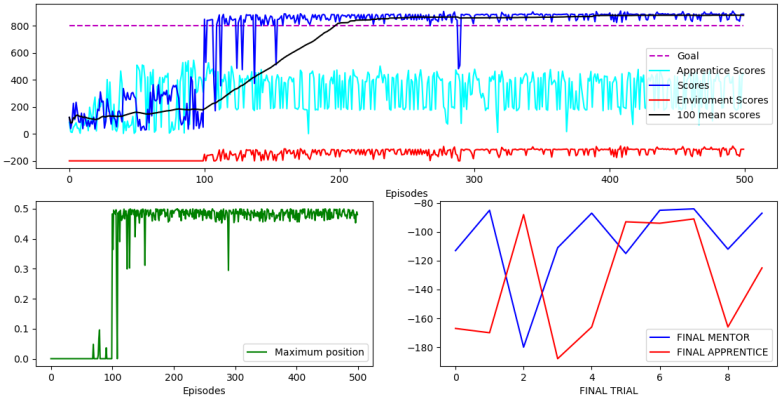

Figura 5.13: Resultados velocidad Adam

del modelo y, en caso de producirse un empeoramiento en los resultados, imponer la configuración de la copia de seguridad; en caso contrario se guarda la nueva mejor configuración.

En la imagen 5.15 se puede ver cómo, una vez alcanza el máximo, el modelo se mantiene sin divergir y sin apenas fluctuar, no dista demasiado con respecto a los últimos prototipos, que ya mantenían un alto nivel de convergencia, por lo que podemos concluir que este último añadido, no aporta demasiado al modelo.

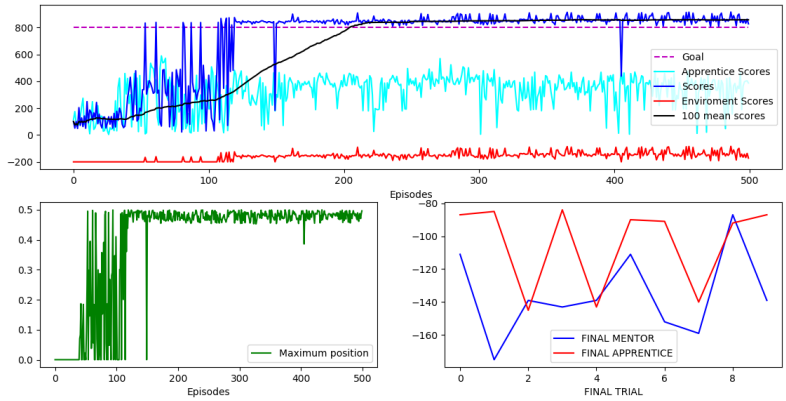

Figura 5.14: Resultados velocidad Adadelta

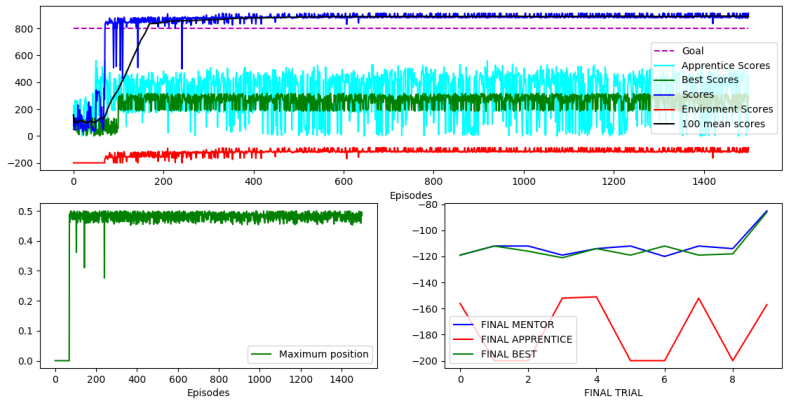

Figura 5.15: Resultado Final

CAPÍTULO 6

Conclusiones y trabajo futuro

"Somos lo que hacemos repetidamente. La excelencia, entonces, no es un acto, es un hábito"
— Aristóteles

6.1. Conclusiones

Este proyecto nos ha servido como un extenso estudio sobre los orígenes y la evolución de la que actualmente es una de las tecnologías más exploradas dentro de la inteligencia artificial: aprendizaje automático, concretamente la subcategoría de aprendizaje por refuerzo profundo.

Cuando empezamos el proyecto, sólo teníamos una leve idea de la complejidad que implicaba este término o del alcance de las utilidades que se pueden conseguir a través de esta tecnología. Pero por suerte, teníamos claro nuestro objetivo y nuestra investigación fue dirigida a conseguirlo: conseguir que una IA pudiera jugar por sí sola a un videojuego.

Comenzamos estudiando las bases del aprendizaje automático, tomando como punto de partida la técnica de aprendizaje por refuerzo. Ésta encajaba perfectamente en nuestro proyecto, ya que tratándose de videojuegos, era fácil encontrar recompensas en ese entorno. Investigando sobre ello, descubrimos el método de Q-Learning, que parecía ser el que mejor se adecuaba a nuestro caso. Nos permitía tomar decisiones en una representación fiable de cualquier entorno, ya que no era necesario establecer un modelo del mismo, así que la IA podría jugar con normalidad independientemente del juego.

Tras el estudio previo, era el momento de hacer nuestras primeras pruebas y aprender cómo utilizar el entorno. Empezamos con un juego sencillo, el CartPole disponible en la librería de OpenAI Gym. El entorno era sencillo y las acciones que debía tomar el agente, reducidas. Sin embargo, el modelo utilizado en Q-Learning nos podía dar problemas en entornos mayores, a pesar de que el enfoque era bueno en un principio: el tamaño que requería la representación de los estados posibles crecía de forma casi exponencial cada vez que se introducían nuevas variables en el entorno, o aumentaba el rango de valores de las existentes. Por ello, necesitaríamos ir un paso más allá.

No fue fácil acostumbrarse al uso del método de Q-Learning. El concepto resultaba curioso a la vez que simple, debido a su parecido con una máquina de estados. No obstante, fue durante la implementación cuando más comenzamos a notar la complejidad que implicaba su uso, incluso en un problema a primera vista sencillo, como resultó ser el de CartPole. El hecho de **plantear la representación del problema supuso un reto**, ya que tratábamos con valores continuos, y fueron necesarias varias configuraciones para la discretización de dichos valores, hasta ver con cuál trabajaba mejor el algoritmo.

Como primera toma de contacto con los elementos que conformarían nuestras futuras investigaciones, en este caso todo lo aplicado sobre el ejemplo del CartPole en Q-Learning, podemos decir que los resultados fueron satisfactorios, y no solo porque el algoritmo consiguiese predecir correctamente los movimientos en el juego. Nos sirvió para prepararnos para los errores que podríamos encontrarnos más adelante, además de comprobar el funcionamiento de este método y crear lo que sería una base teórica para saber cómo aplicarlo a otros entornos en un futuro.

Simultáneamente, comenzamos buscando otros caminos por los que desarrollar nuestro proyecto, optando finalmente por el uso de redes neuronales. Comprobamos con varios ejemplos que éstas eran compatibles con nuestro problema, llegando a la conclusión de que era un buen punto por el que avanzar. Explorando más allá de lo que ya conocíamos de ellas, encontramos la forma perfecta de combinar la idea y resultados del método de Q-Learning con la rapidez y comodidad que suponían las redes neuronales: las DQNs (o *Deep Q-Networks*).

A pesar de que los conceptos que implicaban la creación de las DQNs ya nos eran familiares, los resultados en un primer momento nos pillaron totalmente por sorpresa. El agente no aprendía correctamente, y aunque esperábamos peores resultados por ser nuestro primer intento, no nos habíamos planteado unos tan malos. Fue en este punto cuando la tarea de investigación tomó un papel vital para nosotros, obligándonos a todos a volcarnos en ella, hasta ver nuestros fallos y posibles soluciones a ellos.

Finalmente, gracias al uso de una memoria para que nuestro agente tam-

bién aprendiese de sus propios errores y a una segunda red neuronal para estabilizar sus resultados, conseguimos un correcto aprendizaje. A partir de estos elementos, entendimos que no sólo importaba el volumen de los datos de los que disponíamos, o el número de episodios que dejásemos al agente aprendiendo. **Lo importante era lo relevantes y representativos que debían ser éstos datos para que sirviera de algo aprender de ellos**. De la misma forma, comprendimos que no siempre que nuestro agente aprendía algo nuevo, tenía que ser bueno. La segunda red nos fue muy útil para "guiar.ªl agente por la mejor ruta de aprendizaje, consiguiendo buenos resultados en pocos episodios, en comparación con todas las pruebas anteriores.

El último paso al que llegamos en nuestro proyecto fue a trasladar todo lo aprendido y lo conseguido en el CartPole a otro entorno para comprobar que el agente seguía aprendiendo correctamente. Esta vez se trataba de el juego de MountainCar, también disponible en la librería de OpenAI Gym.

El principal problema al que nos enfrentamos aquí fue a la velocidad de aprendizaje. Tras observar cómo se desenvolvía nuestro agente en el nuevo entorno, nos dimos cuenta de que **la recompensa no trasmitía una información útil para el aprendizaje** del modelo; o por lo menos, no hasta que se alcanzaba el objetivo final, lo cual implicaba un entrenamiento muy largo.

Fue por ello que decidimos centrarnos en cambiar la recompensa. Tras diferentes enfoques y pruebas para modificarla, conseguimos acelerar el aprendizaje de nuestro agente. Sorprendentemente, las soluciones más simples fueron las más efectivas y las que nos llevaron a concluir que habíamos conseguido desarrollar una IA capaz de jugar a videojuegos.

Tal vez en un primer momento la idea de hacer un trabajo de investigación, con una meta que cada vez pueda resultar más lejana, no suene muy atractiva. Sin embargo, para nosotros esto ha resultado mucho más provechoso que centrarnos en terminar un producto final. Este proyecto nos ha abierto una puerta hacia el aprendizaje por refuerzo profundo, las técnicas que lo componen y los avances que se siguen haciendo en este campo.

La investigación y el tiempo invertidos nos han llevado a comprender el trabajo que supone construir una IA de este tipo, incluso teniendo una pequeña base inicial en la que sostenerse. De la misma forma, hemos podido experimentar de primera mano las facilidades que supone su uso, sobre el nuestro y sobre otros proyectos. Además de nuevos conocimientos, hemos adquirido nuevas ideas y una base sobre la que construirlas.

El aprendizaje por refuerzo profundo es un campo que todavía está en crecimiento y del que nos queda mucho por aprender. Pero si algo podemos decir con seguridad es que su evolución todavía no ha acabado y que con el tiempo tendrá mucha más importancia de la que ya tiene a día de hoy. Y,

con suerte para nosotros, estaremos preparados para seguir su avance.

6.1.1. Lecciones aprendidas

- Lograr una buena representación del estado es una tarea complicada en entornos continuos.
- Puede usarse una red neuronal para aproximar una tabla-Q y así evitar lidiar con la representación del estado.
- A la hora de entrenar una red neuronal, se debe hacer con información representativa de todos los estados o situaciones posibles. Si no, el agente mejorará su rendimiento en algunas situaciones pero lo empeorará en muchas otras.
- Utilizar una segunda red neuronal facilita la convergencia, ya que el agente persigue un objetivo que permanece estable durante mayor tiempo.
- Una recompensa representativa es clave para el aprendizaje.

6.2. Cumplimiento de objetivos

Al principio de este documento, planteábamos unos objetivos que explorar durante la realización de este proyecto, los cuales nos han servido para organizar el trabajo y su exposición en el presente documento:

1. El primer objetivo era comprender qué es el aprendizaje por refuerzo y en qué se diferencia de otras ramas del aprendizaje automático. A lo largo del capítulo 2 vimos una explicación de dicho concepto, así como las ideas en las que se basaba y sus diferencias respecto a otras ramas del aprendizaje automático y por qué la elegimos como punto de partida. Siguiendo con esta idea, explicamos todos los elementos que lo componen y la forma que tienen de relacionarse.
2. Una vez visto cómo se relacionan todos los elementos del aprendizaje por refuerzo, decidimos poner dichos conceptos a prueba durante el capítulo 3, en el que explicamos la librería Gym de OpenAI. Elegimos el entorno de CartPole para llevar a cabo nuestros experimentos, logrando superar el objetivo planteado para dicho entorno, pudiendo experimentar los problemas que suponía la implementación de dicho algoritmo en referencia al aumento exponencial de la tabla-Q, así como la complejidad que implicaba mantenerla completamente actualizada, y conseguir el correcto aprendizaje de nuestro agente.

3. Otro objetivo era el de adentrarnos en el campo del aprendizaje profundo y los fundamentos de las redes neuronales. Lo tratamos a lo largo del capítulo 4, durante el cual explicamos los principios de las redes neuronales, su composición, estructura y funcionamiento. De este modo, jugamos con distintos problemas de clasificación y regresión que nos sirvieron para tener una primera toma de contacto que más adelante nos ayudaría para manejar problemas dentro del ámbito del aprendizaje por refuerzo profundo.

4. Nuestro último objetivo era el de combinar las redes neuronales y el aprendizaje por refuerzo, de forma que consiguiéramos obtener los beneficios de ambos, así como encontrar soluciones a los inconvenientes que tenían por separado, como vimos durante el capítulo 5. Estudiamos paso a paso cómo combinar los dos ámbitos, primero reemplazando la tabla-Q del aprendizaje por refuerzo por una red neuronal y viendo que no nos otorgaba un aprendizaje suficiente en nuestro agente. Tras ello introdujimos el concepto de reproductor de experiencias y más tarde el uso de dos redes, gracias a los cuales conseguimos estabilizar el aprendizaje y una convergencia de los resultados.

5. Cumplidos todos los objetivos y en vista a los resultados obtenidos, hemos conseguido ver las limitaciones que tiene este campo de estudio actualmente, el cual sigue en evolución y sobre el que queda mucho trabajo por hacer.

6.3. Trabajo futuro

El proyecto iba dirigido a enseñar a una IA a jugar y, aunque ésta ha aprendido de forma exitosa, apenas ha sido probada en un par de entornos con pocas variables. El camino de nuestra investigación podía ser el correcto, pero sin más avances, o pruebas en otros entornos, no podemos estar seguros de ello.

Dicho esto, en caso de poder continuar nuestro proyecto, nos centraríamos en continuar con más pruebas en distintos juegos. Tras comprobar que nuestra IA es adaptable, empezaríamos actualizándola para que resultase escalable a problemas con más acciones a elegir o un mayor volumen de variables, en caso de que no lo fuera ya. Un punto a considerar sería si el agente podría aplicarse directamente a problemas parecidos a los que ya conoce (*Transfer Learning*), ya entrenado en ellos, y explorar cómo adaptarlo para mejorar sus resultados.

A partir de estas modificaciones, tendríamos que considerar los posibles elementos nuevos que pudieran surgir, en función del problema o juego que tratásemos. El reto consistiría en adaptar el agente a elementos que se salen

de lo visto anteriormente y que pueden provocar que la resolución o victoria no dependa sólo del agente. El caso más común en el que podemos pensar es en entornos multijugador, donde el agente tenga que enfrentarse a otros jugadores, o agentes (entornos multiagente - INSERTAR REF AQUÍ).

Entre estos casos, también podríamos considerar los entornos con información incompleta: situaciones en las que no se pueda ver todo el mapa y en las que el agente tendría que actuar sin conocer el problema en su completitud. Los entornos probabilísticos, fáciles de ver en juegos de cartas, dados o cualquier *RPG* con encuentros y daños no deterministas, también serían algo a tener en cuenta. Además de tener una estrategia ya aprendida, el agente debería aprender a adaptarla o a optar por otra totalmente distinta en función de resultados que no siempre son los que puede esperar.

El trabajo más a largo plazo podría seguir muchos caminos. Por una parte se podría experimentar con otro tipo de algoritmos, buscando una mayor eficiencia. Prueba de ello es "Baselines", el repositorio de OpenAI que ya mencionamos [13]. Otra opción sería dar un paso más allá y empezar a interpretar directamente imágenes como entrada, para así no depender de observaciones en forma de variable como hemos hecho hasta ahora.

DeepMind ya demostró esto hace algunos años. En Mnih et al. [21] demostraron que su IA era capaz de aprender a jugar a una serie de juegos de Atari a nivel profesional, todo esto únicamente a través de los píxeles de cada imagen. Para lograr esto habría que especializarse en redes neuronales convolucionales, lo cual sería todo un reto. Ya no sólo por ser más complejas sino porque, al necesitar mayores recursos a la hora de entrenarse, el sistema debe ser mucho más preciso para ser viable.

Pero todo esfuerzo trae su recompensa: una vez se dispone de una IA capaz de reconocer imágenes, las posibilidades se disparan. Al no estar atado a unas observaciones provistas a través de variables, deja de ser necesario depender de frameworks o tediosas implementaciones manuales para realizar pruebas. Se podría trabajar a un nivel mucho más "real", en el sentido de que tal vez ya no sería necesario describir un problema a la perfección, con todas sus leyes físicas, para trabajar en él. Tal vez simplemente se necesitaría proveer al agente de una serie de imágenes para que fuese capaz de aprender sobre ellas, entendiendo comportamientos y patrones y aprendiendo de forma mucho más humana. No obstante, esta idea todavía queda algo lejana para nosotros.

CHAPTER 6

Conclusions and future work

"We are what we do repeatedly. Excellence, then, is not an act, it is a habit"
— Aristotle

6.1. Conclusions

This project has served us as an extensive study about the origins and evolution of one of the most explored technologies within the Artificial Inteligence: Machine Learning, concretely the Deep Learning field.

When we started this project, we just had a slight idea about this term's complexity involvements or the utilities's scope that can be reached with this technology. But fortunately, our objective was clear and our research was aimed for a single goal: To make it possible for an AI to play a videogame on its own.

We began by studying the basics of Machine Learning, taking as our starting point the technique of Reinforcement Learning. This one fitted perfectly our project, since when it comes to videogames, it is easy to find rewards in that environment. Investigating about it, we discovered the Q-Learning method, which seemed to be the best suited to our case. It allowed us to make decisions at every moment, maintaining a reliable representation of any possible environment, in a way that the AI could play normally regardless of the game.

After the previous study, it was time to do our first tests and learn how

to use the environment. We started with a simple game: CartPole, available in the OpenAI Gym library. The environment was simple and the actions to be taken by the agent were reduced. In spite of that, the model used in Q-Learning could give us a lot of problems in larger environments, even though the idea was good at first: The required size of the representation for all possible states grew almost exponentially every time we introduced new environment variables or we modified the existing ones. That is why we needed to go one step further.

It was not easy to get used to Q-Learning. The concept was interesting and also simple, due to its similarity with a state machine. However, it was during the implementation when we started to notice the complexity that its use implies, even in an easy problem as it was the CartPole one. The fact of **setting the problem representation was a challenge itself**, since we were treating with continuous values. Multiple configurations were needed for discretizing these values, to see which one was the best for the algorithm.

As a first contact with the elements that will form our future researchs, in this case, everything related to the CartPole example on Q-Learning, we can conclude that the results were satisfying and not only because our algorithm was finally able to predict correctly the moves in the game. It served us to be prepared for the kind of errors we would find later, besides learning this method's functioning and creating a theoretical base, in order to know how to apply it on future environments.

Almost simultaneously, we started looking for other ways to develop our project, finally opting for Neural Networks. We verified with several examples that these were compatible with our problem, concluding that it was a good point to advance. Exploring beyond what we already knew about them, we found the perfect way to combine the idea and results of Q-Learning with the speed and comfort that Neural Networks provide: DQNs, or Deep Q-Networks.

Even though the concepts involving the DQNs' origination were already familiar for us, the very first results caught us by surprise. Our agent was not learning correctly and although we were expecting worse results than the Q-Learning implementation, for being our first attempt, we were not prepared for that bad. It was at this point when the research task took a crucial role for us, compeling us to get involved in it, until we realized our mistakes and the possible solutions to them.

Finally, thanks to the use of an additional memory, for our agent to learn from its own mistakes, and a second network, for stabilizing its results, we achieved a correct learning. From this elements, we understood that it did not matter the quantity of the available data or the number of episodes that the agent was learning. **The essential was how significant and representative should our data be**, so that we could learn something

useful from them. In the same way, we understood that the fact of learning something new was not necessarily good for our agent. The second neural network was very helpful to "guide" the agent along the best learning way, accomplishing better results in fewer episodes, compared to previous tests.

The last step we reached in our project was to transfer everything we learned and accomplished in CartPole to another environment to check that the agent was still learning correctly. This time it was the MountainCar game, also available in the OpenAI Gym library.

The main problem we had to face was the learning rate. After observing our agent's progress in this new environment, we realized that **the reward was not giving us any helpful information** for the model's learning; or, at least, not until the game goal was accomplished, which implied a very long lasting training.

Because of this fact, we decide to focus on changing the reward. After many different approaches and tests to modify it, we finally achieved a faster learning for our agent. Surprisingly, the simplest solutions were also the most effective ones. These leads us to conclude that we had managed to develop an AI capable of playing.

Maybe at first sight, choosing a research project, with an increasingly distant aim, does not sound very attractive. Nevertheless, this has been result way more profitable for us than focusing ourselves in complete a final product. This project opened us a door towards Deep Reinforcement Learning, including its component techniques and the progresses that continue to be made in this field.

The research and time invested, has led us to understand the work involved in building an AI, even having a small initial base on which to stand. We have been able to experience personally the facilities of its use, applied to many projects, including ours. In addition to new knowledge, we have acquired new ideas and base on which build them.

Deep Reinforcement Learning is a still growing field, on which much remains to be learned. But if we can say something for sure is that its evolution is not over yet and that, in a not too distant future, it will gain much more importance than it already has. And luckily, we will be prepared to follow its progress.

6.1.1. Lessons learned

- Finding a good representation of the state is a hard task for continuous environments.
- A Neural Network can be used to approximate a Q-table and then avoid having to discretize the state.

- Training a Neural Network must be done with representative information of as much states as possible. Otherwise, the agent might improve its performance for a specific subtask at the cost of getting worse at others.
- Using a second Neural Network improves convergence, because the agent follows a more stable goal for longer periods.
- A representative reward is vital for the agent's learning.

6.2. Achievement of objectives

At the beginning of this document, we proposed some objectives to explore during the course of this project, which have helped us to organize the work and its presentation in this document:

1. The first objective was to understand what Reinforcement Learning is and how it differs from other branches of Machine Learning. Throughout the chapter 2 we saw an explanation of this concept, the ideas on which it was based and their differences from other branches of Machine Learning, aswell as why we chose it as a starting point. Continuing with this idea, we explain all the elements that compose it and the way in which they relate to each other.
2. Having seen how all the elements of Reinforcement Learning are related, we decided to test these concepts during the chapter 3, in which we explained the OpenAI Gym library. We chose the CartPole environment to carry out our experiments, managing to overtake the objective set for this environment and experimenting the problems involved in the implementation of this algorithm, in reference to the exponential increase of the Q-table and the complexity involved in keeping it completely updated, and achieving the correct learning of our agent.
3. Another objective was to get into the field of Deep Learning and the fundamentals of Neural Networks. We discussed it throughout the chapter 4, during which we explained the principles of Neural Networks, their composition, structure and behaviour. This way, we played with different problems of classification and regression that served us to have a first contact that later would help us to manage problems within the scope of Deep Reinforcement Learning.
4. Our last goal was to combine Neural Networks and Reinforcement Learning, in a way that we could get the benefits of both, as well as finding solutions to their different issues, as we saw during the chapter

5. We studied step by step how to combine the two scopes, first replacing the Q-table of Reinforcement Learning with a Neuronal Network, which we saw that it didn't give our agent enough learning skills. After that, we introduced the concept of experience replay and the use of two networks, thanks to which we managed to stabilize learning and convergence.

5. Having fulfilled all the objectives, and in view of the obtained results, we have been able to see the limitations of this field of study as of today, which is quickly evolving with new techniques every day. There definetly are lots of ways to follow if you are eager and interested in it.

6.3. Future work

The project was aimed for teaching an AI to play and, although it has learned successfully, it has hardly been tested in a couple of environments with few variables. The path of our research could be the right one, but without further progress, or testing in other environments, we cannot be sure of that.

That said, if we could continue our project, we would focus on continuing with more tests on different games. After verifying that our AI is adaptable, we would start by adding more options to the possible actions that the player can perform, updating it to make it scalable, in case it already wasn't.

From there, work could follow several paths. On one hand we could experiment with other types of algorithms, looking for a greater efficiency. Proof of this is "Baselines", the OpenAI repository that we have already mentioned [13]. Another option would be to go a step further and start interpreting images directly as input, so as not to depend on observations in the form of a variable as we have done so far.

DeepMind already demonstrated this a few years ago. In Mnih et al. [21] they proved that their AI was able to learn to play some Atari games at a professional level, all of this only through the pixels of each image. To achieve this we would have to specialize in Neural Convolutional Networks, which would be quite a challenge. Not only because they are more complex, but also because, since they need more resources when it comes to training, the system must be much more precise to be viable.

But every effort pays off: Once you have an AI capable of recognizing images, the possibilities soar. By not being tied to observations provided through variables, it is no longer necessary to rely on frameworks or tedious manual implementations to perform tests. One could work at a much more "real" level, in the sense that perhaps it would no longer be necessary to describe a problem perfectly, with all its physical laws, in order to work

on it. Perhaps it would simply be necessary to provide the agent with a series of images so that it would be able to learn about them, understanding behaviors and patterns and learning in a much more humane way. However, this idea is still a long way off for us.

CAPÍTULO 7

Aportación de los participantes

7.1. Ricardo Arranz Janeiro

7.1.1. Antecedentes

Antes de comenzar este proyecto, mi conocimiento sobre redes neuronales se limitaba al adquirido durante la asignatura de Inteligencia Artificial. En esta asignatura nos enseñaron los tipos básicos de redes neuronales, cuándo aplicar cada tipo de red y cómo interpretar gráficas resultantes. No llegamos a implementarlos en ningún proyecto, pero contaba con la base teórica.

Por otro lado, el concepto de aprendizaje por refuerzo me era prácticamente desconocido, surgió durante la asignatura de Programación Evolutiva, cuando unos compañeros hicieron un prototipo capaz de jugar al Super Mario Bros, y que mediante técnicas de programación evolutiva iba mejorando su árbol de decisión, en ese momento el profesor comentó que para este tipo problemas era más eficiente utilizar técnicas de Aprendizaje por refuerzo. Parecía algo interesante pero no volvimos a tratar el tema.

7.1.2. Aportación

Al ser el último en unirme al grupo, me encontré con que mis compañeros ya contaban con cierto conocimiento en la materia que íbamos a tratar, así que inicialmente me dediqué a ponerme al día. Estudié el libro *Fundamentals of Deep Learning* [8] y empecé a indagar en internet buscando artículos que

hablaran de esta tecnología.

Después, asistí a mis compañeros, Lidia Concepción y Francisco Ponce, en el desarrollo de redes neuronales, mi pequeña aportación consistió en ayudar con el ajuste de los hiperpámetros del ejercicio de regresión, para así acelerar el aprendizaje.

A estas alturas la primera fase de nuestro proyecto llegaba a su fin y teníamos que empezar con el desarrollo del prototipo de Aprendizaje por Refuerzo Profundo. Esta parte consistía en suplir el problema de almacenamiento del Q-Learning con una red neuronal.

El primer problema al que nos enfrentamos fue implementar CartPole, pero esta vez utilizando la técnica de *Deep Q-Learning*, Juan Ramón del Caño y Juan Luis Romero, que se habían encargado de la parte de aprendizaje por refuerzo, fueron los que más aportaron en esta parte. Aunque todos investigamos cómo solucionar los problemas que iban surgiendo, ellos fueron los que llevaron mayor carga de trabajo. Mientras tanto yo me centré en desarrollar las distintas versiones del *DoubleAgent*, que inicialmente, comenzó siendo dos agentes completamente separados y terminó siendo un solo agente relativamente escalable.

Más tarde me volqué en el desarrollo de los prototipos del problema de MountainCar. El prototipo inicial se basaba en la recompensa devuelta por el entorno, pero viendo los resultados comprendimos que no bastaba con las técnicas aprendidas durante el CartPole, teníamos que buscar alguna forma de acelerar el aprendizaje.

Tras una lluvia de ideas en el despacho de Antonio, comenzamos a explorar nuevas opciones. Por mi parte, me centré en la función de recompensa, creé distintas variantes en busca de un aprendizaje más rápido, pero todos fueron un fracaso al principio, no aprendían bien o se atascaban en máximos locales. Además, debido a la baja velocidad de aprendizaje necesitábamos muchos episodios para ver diferencia entre los prototipos, lo que se traduce en grandes tiempos de espera entre pruebas.

Poco a poco, con la implementación de nuevas técnicas, refinamiento y descarté de otras, y el conocimiento adquirido en sucesivos intentos, fui consiguiendo buenos resultados. Finalmente obtuvimos una solución capaz de alcanzar el objetivo en un periodo mucho menor al que obtuvimos al principio y con una robusta convergencia.

7.2. Lidia Concepción Echeverría

7.2.1. Antecedentes

La asignatura que me sirvió de base para entrar al proyecto fue la de Inteligencia Artificial. Si bien es cierto que ésta era más una introducción que un análisis en profundidad de toda la rama a la que se refiere, me llamaron la atención los temas dedicados a Aprendizaje Automático. Las nociones que aprendí entonces eran las de los tipos de Aprendizaje, diferenciando entre Supervisado y No Supervisado, además de conocer como concepto los algoritmos genéticos y las Redes Neuronales.

Minería de Datos y Aprendizaje Automático fueron el siguiente paso, ambas asignaturas destinadas a aprender más sobre la rama con el mismo nombre que ésta última. Mientras que en Minería de Datos aprendí el uso de las librerías de Python para resolver directamente problemas y buscar mejoras en los resultados, en Aprendizaje Automático estudié el funcionamiento de esas librerías y los cálculos matemáticos que había tras los procesos que se llevaban a cabo para llegar finalmente a una resolución, ambas desde un enfoque práctico. Aprendizaje Automático fue la que más me sirvió para entender el funcionamiento de Redes Neuronales y los problemas aplicables a las mismas.

Respecto a las tecnologías utilizadas, apenas había manejado Python antes de cursar las dos asignaturas mencionadas anteriormente. Gracias a ellas conseguí una introducción para el uso de librerías como `numpy`, `pandas` y `scikit-learn`, en forma de *notebook*. Las prácticas en empresa que realicé este mismo curso tocaban temas relacionados con Aprendizaje No Supervisado, como técnicas de clustering, utilizando también Python. De esta forma, conseguí mantener una base sólida de este lenguaje de forma que no entorpeciera el trabajo durante el proyecto.

7.2.2. Aportación

Durante la primera parte de la investigación, me encargué junto a Francisco Ponce de lo relacionado con Redes Neuronales, ya que ambos estábamos cursando Aprendizaje Automático y teníamos más reciente ese tema. El libro que usamos de referencia para saber por dónde avanzar y ampliar mi investigación fue el *Fundamentals of Deep Learning* [8].

Lo primero era realizar algunas pruebas con los conocimientos que tenía sobre Redes Neuronales, con el fin de familiarizarnos con `Keras`, la librería que íbamos a utilizar a lo largo del proyecto. Para ello, desarrollamos una prueba sobre un ejemplo de clasificación conocido 4.1.3. Viendo que funcionaba sin problemas, realizamos una segunda prueba, esta vez con un problema

de regresión 4.1.4. Este tipo sería el que utilizaríamos posteriormente, por lo que debíamos asegurarnos de que podíamos manejarnos con redes destinadas a estos problemas.

Tras la segunda prueba, fue el momento de poner en común lo aprendido por nuestra parte respecto al grupo que se había dedicado a la parte de Q-Learning, y comenzar a discutir cómo avanzar al siguiente paso en nuestro proyecto: Aprendizaje por Refuerzo Profundo, aplicado concretamente al problema de CartPole. Cada uno intentó ver el problema desde un enfoque distinto, convergiendo finalmente en lo que sería el resultado final 5.1.4. Llegar a este punto conllevó una importante investigación por parte de todos, ya que las versiones anteriores no funcionaban correctamente o daban resultados poco deseables. La mayor carga de trabajo en este punto fue llevada por Juan Ramón del Caño y Juan Luis Romero; por mi parte, me dediqué a apoyar en la resolución de problemas y a investigar en busca de otras opciones.

Mientras el desarrollo del agente utilizando DQNs era trasladado al entorno de MountainCar, por parte de Ricardo Arranz, decidimos desarrollar y corregir las ideas que habíamos ido planteando para la memoria final. Me encargué del capítulo dedicado a explicar las Redes Neuronales y DQNs, además de añadir múltiples aportaciones y correcciones en los demás capítulos.

7.3. Juan Ramón del Caño Vega

7.3.1. Antecedentes

Antes de empezar el proyecto ya contaba con un conocimiento básico sobre el Aprendizaje por Refuerzo. En la asignatura que cursé de Inteligencia Artificial se le daba bastante importancia a este apartado, especialmente de forma práctica. En los laboratorios trabajamos con Q-Learning en un entorno Java. Se trataba de una simulación en la que teníamos que estabilizar una nave espacial con tres motores, no obstante tan sólo tuvimos que implementar las funciones de recompensa y discretización.

Respecto a Redes Neuronales, recuerdo que no entramos en profundidad. Se nos explicaron, pero ni llegamos a utilizarlas de forma práctica ni se consideraba materia de examen, por lo que quedaron bastante de lado. Lo mismo ocurrió con el Aprendizaje por Refuerzo Profundo, el cual se nos mencionó al final del curso junto con sus posibles usos en campos como el reconocimiento de imágenes.

También tenía experiencia en otras áreas del Aprendizaje Automático, tanto en Aprendizaje Supervisado como en No Supervisado, con los que he trabajado en las librerías `scikit-learn`, `pandas` o `numpy` de Python. Esto, a

pesar de no estar directamente relacionado con nuestro trabajo, me facilitó acostumbrarme a trabajar con `Keras`.

7.3.2. Aportación

Inicialmente me dediqué a la parte de Aprendizaje por Refuerzo. Puesto que ya tenía una buena base teórica pudimos empezar a hacer pruebas con bastante rapidez.

Empecé por programar el simulador para poder ejecutar CartPole. El objetivo era hacerlo de forma modular, principalmente por un motivo: encapsular toda la lógica del agente en una clase propia nos permitiría mantener el "bucle de ejecución" lo más limpio y simple posible, de esta forma se asemejaba mucho al pseudocódigo que veíamos en los libros, como puede verse en los fragmentos de código de la sección 5.1).

Una vez conseguido eso, sólo quedaba por implementar la lógica del agente. Quizá lo más complicado fue la función de discretización, explicada en el apartado 3.3.1. Una vez implementada la función parametrizada sólo fue cuestión de probar algunas configuraciones e hiperparámetros hasta dar con la solución que más nos gustase. Eso y corregir algún que otro bug, como que la ecuación de Bellman no sumase recompensas futuras si el agente se encontraba en un estado final, lo cual hacía que nuestro algoritmo divergiese.

A la hora de documentar este primer bloque recurrí al libro *Artificial Intelligence: A Moddern Approach* [31], el cual también fue mi libro de referencia durante la asignatura de Inteligencia Artificial y proporciona explicaciones bastante concisas de distintos campos. Aproveché este libro para escribir la introducción del proyecto y la primera parte de Aprendizaje por Refuerzo. Para los apartados más técnicos cambié a *Reinforcement Learning: An Introduction* [34], manual por excelencia del Aprendizaje por Refuerzo. No obstante fue Juan Luis Romero (quien también ayudó e hizo pruebas con CartPole) el encargado de rematar el capítulo con las secciones de Q-Learning y Markov Decision Process.

En este punto el resto del equipo ya había acabado de investigar Redes Neuronales (especialmente Lidia Concepción y Francisco Ponce), y nos preparábamos para empezar con el Aprendizaje por Refuerzo Profundo. Para ponerme al día con Redes Neuronales repetí uno de los ejemplos de clasificación que ellos ya habían hecho, MNIST [12], pero esta vez usando Jupyter Notebooks y el conjunto de datos propio de Keras.

Una vez hecho esto todos nos pusimos a volver a resolver CartPole utilizando las DQN que vimos en libros como *Fundamentals of Deep Learning* [8] y los artículos de DeepMind. Mientras el resto del equipo saltó directamente a las implementaciones vistas en los apartados 5.1.3 y 5.1.4, yo empecé

desde el punto 5.1. Estas implementaciones, a pesar de que sabíamos que no funcionarían demasiado bien, nos permitieron comprender el proceso de mejora del agente mucho mejor, y por supuesto a documentarlo mejor, en lo que también participé.

Finalmente, también dediqué tiempo al problema de MountainCar. Especialmente a darnos cuenta de por qué es un problema tan especial y cuáles eran los motivos por los que presentaba nuevos retos. Finalmente fue Ricardo Arranz quien se enfrentó con el problema hasta el final.

7.4. Francisco Ponce Belmonte

7.4.1. Antecedentes

Al principio tenía conocimientos bastante ligeros sobre temas como Aprendizaje por Refuerzo y Redes Neuronales, todos ellos lo aprendí en la asignatura de Inteligencia Artificial. Sin embargo, considerando esta base insuficiente, decidí cursar la optativa de Aprendizaje Automático, con el fin de ganar más conocimientos y soltura en el uso y funcionamiento de las Redes Neuronales.

Por otro lado, también cursé Minería de Datos. Aunque la asignatura no estaba directamente relacionada con el objetivo de este proyecto, sí que utilizaba algunas herramientas y principios que me resultaron útiles para empezar con mis aportaciones. Entre ellos, cabría destacar las librerías `scikit-learn`, `pandas` o `numpy` de Python.

7.4.2. Aportación

Mientras algunos de mis compañeros se encargaban de la base para Aprendizaje por Refuerzo, yo me centré en el desarrollo de la red neuronal junto a Lidia Concepción. Aunque ya teníamos unos conocimientos base, además de lo que íbamos aprendiendo paralelamente en Aprendizaje Automático, decidimos empezar desde abajo para ir analizando paso a paso las posibilidades de esta rama.

Debido a ello, comenzamos con el desarrollo de una red neuronal de clasificación 4.1.3 usando las herramientas con las que ya estábamos familiarizados, consiguiendo resultados rápidamente y sin ningún problema. Una vez más acostumbrados al funcionamiento de las Redes Neuronales, empezamos a preparar lo que realmente necesitaría para nuestro proyecto, una red neuronal de regresión 4.1.4. Para ello, dejamos atrás las herramientas conocidas y comenzamos a utilizar `Keras`. Éste ya poseía unos cuantos ejemplos que podía aprovechar, aparte de automatizar muchos de los procesos necesarios para la resolución del problema.

Para entonces, el resto del grupo ya había concluido con su parte y pudimos empezar a unir nuestros aportes para empezar con el Aprendizaje por Refuerzo Profundo. Por mi parte, empecé directamente con las implementaciones vistas en 5.1.3, en busca de obtener resultados que analizar. Sin embargo, ante la tesitura de que éramos muchos trabajando individualmente sobre el mismo problema, y que algunos de mis compañeros estaban consiguiendo mejores resultados, decidí centrarme más en la memoria del proyecto.

Durante la parte de Aprendizaje Profundo, desarrollada en el capítulo 4, comencé referenciando lo aprendido en las pruebas de los primeros meses tanto el problema de casificación visto en el punto 4.1.3, como el problema de regresión desarrollado en el punto 4.1.4, para luego basarme en el libro de *Fundamentals of Deep Learning*, especialmente los capitulos sobre el descenso de gradiente (capítulo 2) y sobre optimizadores (capítulo 4) del libro de Buduma y Locascio [8]. En ellos se tratan de manera profunda los fundamentos y mecánicas de Redes Neuronales y su aplicación y uso en el Aprendizaje por Refuerzo Profundo.

En última instancia, al igual que el resto de mis compañeros, revisé el trabajo completo tanto para búsqueda de errores como para una mayor comprensión de lo conseguido en el proyecto.

7.5. Juan Luis Romero Sánchez

7.5.1. Antecedentes

Hace un año, antes de embarcarme en la realización de este trabajo de investigación, no tenía las ideas claras sobre qué tema o idea sentar las bases del proyecto, por lo que decidí investigar en la lista de trabajos propuestos por los profesores y encontré en este la única temática que me llamaba la atención. Tras una primera toma de contacto con Antonio, donde me explicó la idea a desarrollar y que el grupo inicial ya estaba formado por Juan, Francisco y Lidia, me preguntó acerca de mis conocimientos en relación al tópico tratado. Me di cuenta que no había cursado ninguna asignatura en relación a Aprendizaje Automático, Redes Neuronales, Inteligencia Artificial ni nada por el estilo, ni siquiera había programado una sola línea de código en Python antes, por lo que tenía trabajo pendiente para verano antes de ponernos a trabajar todos en conjunto.

De este modo, y para no quedarme rezagado respecto a los conocimientos previos que otros compañeros ya tenían al haber cursado dichas asignaturas, durante el verano me puse a estudiar Python ya que sería el lenguaje de programación que utilizaríamos en toda la parte referente al código, así como

a leer artículos [21] [39] [30] y libros como el de Buduma y Locascio [8], recomendados por el profesor.

Toda la información que iba recogiendo me sonaba muy abstracta, pero sin darme cuenta, estaba creando una base que más tarde cuando comenzase a trabajar con el resto de compañeros cogería forma.

7.5.2. Aportación

Al principio hicimos una división en dos del grupo, con lo que unos empezarían a trabajar con la parte relacionada con las Redes Neuronales, y otra parte (en la que yo estuve) que trabajase en el Aprendizaje por Refuerzo. Decidí empezar a trabajar sobre este apartado ya que había leido algo más acerca de él, de modo que creí que podría aportar más, al mismo tiempo que no me sintiera tan perdido en la materia.

Si bien es cierto que fue Juan el que hizo todo el código sobre el que empezamos a trabajar, en un primer momento fui un poco más a remolque ya que él tenía una idea más concreta del tema, al haber cursado con anterioridad asignaturas que le ayudaban a ello, de modo que mi aportación se limitó a entender qué y cómo lo hacíamos. Con el paso de las semanas y alguna reunión con el profesor entre medias, logré reengancharme y entender mejor el tema que estábamos tratando, gracias a lo cual pude servir de más ayuda haciendo más pruebas con el código, probando y entendiendo más configuraciones, así como arreglando algún que otro bug que encontrábamos, como fue afinar la ecuación de Bellman para que actualizase las recompensas de una forma correcta.

En paralelo, el resto de compañeros estuvieron trabajando con problemas relacionados con las Redes Neuronales, para que consiguieran entenderlas y saber cómo trabajar con ellas, ya que más tarde tendríamos que unificar ambos conceptos para trabajar en conjunto. Una vez la parte de Aprendizaje por Refuerzo quedó bastante estable, a falta de pocos retoques para darla por terminada, empecé a revisar a documentación y los ejercicios en los que mis compañeros habían estado trabajando este tiempo.

Referente a la memoria, hasta este punto revisé lo que ya había escrito Juan, corrigiendo pequeñas cosas del capítulo 2 y completándolo con las secciones de Problemas de Decisión de Markov y Q-Learning, para las cuales me basé en el libro de Buduma y Locascio [8, cap. 9] y Watkins y Dayan [40].

Llegada la hora de empezar a trabajar con el Aprendizaje Profundo por Refuerzo, me salté la parte del `SimpleAgent` y el `BatchAgent` para comenzar directamente sobre el `RandomBatchAgent`, en el cual volqué mucho esfuerzo para hacer que funcionara como debería. Solventé la mayor parte de los pro-

blemas respecto al `Experience Replay` puesto que no lográbamos aplicar bien la ecuación de Bellman y de ese modo hacer que el agente aprendiese, hasta llegar al punto de tener un aprendizaje bastante estable, pero aún así con significativas diferencias entre ejecuciones. No lográbamos que convergiese del todo, hasta que Ricardo añadió la parte del `DoubleAgent`, sobre la cual trabajé haciendo pequeños retoques para conseguir que funcionase. Así mismo, hice una redacción inicial en lo referente a esta sección en la que había trabajado, ayudando posteriormente en su refinamiento.

Por último le llegó el turno al cambio de entorno y empezar a trabajar con el MountainCar, sobre el cual hice pruebas al principio, pero al ver que otros compañeros conseguían más y mejores avances, en especial Ricardo, decidí dejar en esas manos el problema y volcar mi esfuerzo en corregir y redactar partes de la memoria en base a las correcciones que Antonio nos fue indicando.

Bibliografía

[1] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y. y Zheng, X. TensorFlow: Large-scale machine learning on heterogeneous systems. 2015. Software available from tensorflow.org.

[2] Anaconda, I. Anaconda. `https://www.anaconda.com/why-anaconda/`, 2012.

[3] Andrew G. Barto, C. W. A., Richard S. Sutton. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Trans. Systems, Man, and Cybernetics*, vol. 13(5), páginas 834–846, 1983.

[4] Baird, L. Residual algorithms: Reinforcement learning with function approximation. página 10, 1995.

[5] Bellman, R., Bellman, R. y Corporation, R. *Dynamic Programming*. Rand Corporation research study. Princeton University Press, 1957.

[6] Bellman, R. y Collection, K. M. R. *Adaptive Control Processes: A Guided Tour*. Princeton Legacy Library. Princeton University Press, 1961.

[7] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J. y Zaremba, W. Openai gym. 2016.

[8] Buduma, N. y Locascio, N. *Fundamentals of Deep Learning: Designing Next-Generation Machine Intelligence Algorithms*. O'Reilly Media, Inc., 1st edición, 2017. ISBN 1491925612, 9781491925614.

[9] Chigozie Enyinna Nwankpa, Winifred Ijomah, Anthony Gachagan, and Stephen Marshall. Activation functions: Comparison of trends in practice and research for deep learning. (November), páginas 8–9, 2018.

[10] Chollet, F. et al. Boston housing price regression dataset. `https://keras.io/datasets/#boston-housing-price-regression-dataset`, 2015.

[11] Chollet, F. et al. Keras. `https://keras.io`, 2015.

[12] Chollet, F. et al. Mnist database of handwritten digits. `https://keras.io/datasets/#mnist-database-of-handwritten-digits`, 2015.

[13] Dhariwal, P., Hesse, C., Klimov, O., Nichol, A., Plappert, M., Radford, A., Schulman, J., Sidor, S., Wu, Y. y Zhokhov, P. Openai baselines. `https://github.com/openai/baselines`, 2017.

[14] Edelkamp, S. y Kissmann, P. Symbolic classification of general two-player games. En *Proceedings of the 31st Annual German Conference on Advances in Artificial Intelligence*, KI '08, páginas 185–192. Springer-Verlag, Berlin, Heidelberg, 2008. ISBN 978-3-540-85844-7.

[15] Foundation, P. S. Python. `https://www.python.org/about/`, 1991.

[16] Hasselt, H. V. Double q-learning. páginas 2613–2621, 2010.

[17] Hunter, J. D. Matplotlib. `https://matplotlib.org/`, 2003.

[18] Kingma, D. P. y Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[19] Lecun, Y., Bottou, L., Bengio, Y. y Haffner, P. Gradient-based learning applied to document recognition. En *Proceedings of the IEEE*, páginas 2278–2324. 1998.

[20] LeCun, Y. y Cortes, C. MNIST handwritten digit database. 2010.

[21] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D. y Riedmiller, M. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[22] Moore, A. W. Efficient memory-based learning for robot control. Informe técnico, University of Cambridge, 1990.

[23] Oliphant, T. Numpy. `https://www.numpy.org/`, 1995.

[24] OpenAI. Charter. `https://openai.com/charter`, 2018.

[25] OpenAI. Five. `https://blog.openai.com/openai-five/`, 2018.

[26] OpenAI. Spinningup. `https://spinningup.openai.com/en/latest/`, 2018.

[27] Poole, D. y Mackworth, A. *Artificial Intelligence: Foundations of Computational Agents*. Cambridge University Press, Cambridge, UK, 2 edición, 2017. ISBN 978-0-521-51900-7.

[28] Puterman, M. L. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley and Sons, Inc., New York, NY, USA, 1994. ISBN 0471619779.

[29] Rardin, R. *Optimization in Operations Research*. Prentice Hall, 1998. ISBN 9780023984150.

[30] Rodriguez, R. y Bontrager, P. Deep reinforcement learning for general video game. *Deep Reinforcement Learning*, 2018.

[31] Russell, S. y Norvig, P. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edición, 2009. ISBN 0136042597, 9780136042594.

[32] Seide, F. y Agarwal, A. Cntk: Microsoft's open-source deep-learning toolkit. En *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, páginas 2135–2135. ACM, New York, NY, USA, 2016. ISBN 978-1-4503-4232-2.

[33] Skinner, B. *Science and human behavior*. Macmillan, New York, NY, USA, 1953. ISBN 0029290406, 9780029290408.

[34] Sutton, R. S. y Barto, A. G. *Reinforcement Learning: An Introduction*. A Bradford Book, USA, 2018. ISBN 0262039249, 9780262039246.

[35] Szegedy, C., Toshev, A. y Erhan, D. Deep neural networks for object detection. En *Advances in Neural Information Processing Systems 26* (editado por C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani y K. Q. Weinberger), páginas 2553–2561. Curran Associates, Inc., 2013.

[36] Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, vol. abs/1605.02688, 2016.

[37] Tim. Solving mountain car with q-learning. `https://medium.com/@ts1829/solving-mountain-car-with-q-learning-b77bf71b1de2`, 2018.

[38] Tokic, Michel. Adaptive e-greedy exploration in reinforcement learning based on value differences. En *Proceedings of the 33rd Annual German Conference on Advances in Artificial Intelligence*, páginas 203–210. Springer-Verlag, Berlin, Heidelberg, 2010. ISBN 3-642-16110-3, 978-3-642-16110-0.

[39] Turing, A. M. Computing machinery and intelligence. *Mind*, vol. 59(October), páginas 433–460, 1950.

[40] Watkins, C. J. C. H. y Dayan, P. Q-learning. *Machine Learning*, vol. 8(3), páginas 279–292, 1992. ISSN 1573-0565.

[41] Wilson, Ashia C and Roelofs, Rebecca and Stern, Mitchell and Srebro, Nati and Recht, Benjamin. The marginal value of adaptive gradient methods in machine learning. páginas 4148–4158, 2017.