Universitat Oberta de Catalunya

uoc.edu

Desarrollo de un agente mediante Deep Q-Learning en un entorno de juegos de plataformas

Álvaro Buedo Risueño
Máster Universitario en Ingeniería computacional y Matemáticas
Inteligencia artificial

Consultor: Samir Kanaan Izquierdo
Profesor responsable de la asignatura: Carles Ventura Royo

Fecha de entrega: 06/2020

Licencias alternativas (elegir alguna de las siguientes y sustituir la de la página anterior)

A) Creative Commons:

Esta obra está sujeta a una licencia de Reconocimiento-NoComercial-SinObraDerivada 3.0 España de Creative Commons

Esta obra está sujeta a una licencia de Reconocimiento-NoComercial-CompartirIgual 3.0 España de Creative Commons

Esta obra está sujeta a una licencia de Reconocimiento-NoComercial 3.0 España de Creative Commons

Esta obra está sujeta a una licencia de Reconocimiento-SinObraDerivada 3.0 España de Creative Commons

Esta obra está sujeta a una licencia de Reconocimiento-CompartirIgual 3.0 España de Creative Commons

Esta obra está sujeta a una licencia de Reconocimiento 3.0 España de Creative Commons

B) GNU Free Documentation License (GNU FDL)

C) Copyright

FICHA DEL TRABAJO FINAL

Título del trabajo:	*Desarrollo de un agente mediante Deep Q-Learning en un entorno de juegos de plataformas*
Nombre del autor:	*Álvaro Buedo Risueño*
Nombre del consultor/a:	*Samir Kanaan Izquierdo*
Nombre del PRA:	*Carles Ventura Royo*
Fecha de entrega (mm/aaaa):	06/2020
Titulación:	*Máster universitario en Ingeniería computacional y Matemñáticas*
Área del Trabajo Final:	*El nombre de la asignatura de TF*
Idioma del trabajo:	*Español*
Palabras clave	*Deep Learning, Redes neuronales, aprendizaje por refuerzo*

Resumen del Trabajo (máximo 250 palabras): *Con la finalidad, contexto de aplicación, metodología, resultados i conclusiones del trabajo.*

Hoy en día cada vez es más frecuente la interacción entre las personas y los sistemas informáticos, por lo que las técnicas de aprendizaje máquina cada vez son más relevantes y necesarias en nuestras vidas ya que pueden tener aplicación en casi cualquier ámbito.

Este proyecto de investigación consiste en la aplicación de la técnica de aprendizaje por refuerzo profundo al juego de plataformas “Super Mario Bros”.

Concretamente, en el proyecto se desarrolla la implementación del algoritmo Q-Learning, uno de los algoritmos más famosos dentro de la técnica del aprendizaje por refuerzo. Además del aprendizaje por refuerzo, debido a la

complejidad del problema, se han introducido redes neuronales convolucionales, desarrollando así lo que se conoce como aprendizaje por refuerzo profundo (DQN) Con el uso de este algoritmo se pretende lograr que el personaje del videojuego sea capaz de superar los niveles que se le planteen en base a la experiencia obtenida a lo largo de su exploración de dichos niveles.

Para implementar este algoritmo se ha utilizado el lenguaje de programación Python, que es el más potente para el desarrollo de sistemas de iunteligencia artificial, con sus librerías TensorFlow y Keras para la implementación de las redes neuronales

A lo largo de la memoria se expone tanto el análisis, diseño e implementación del sistema, como la presentación de los resultados obtenidos y su interpretación. Para finalizar el trabajo se describe la serie de conclusiones obtenidas junto con el planteamiento del posible trabajo futuro a modo de ampliación y mejora del mismo.

Abstract (in English, 250 words or less):

Nowadays, the interaction between humans and computer systems is turning into a greater and greater tendency. Thus, the significance of machine learning techniques in our lives is constantly increasing, since they could be applied in almost every sphere.

In the present work we employ Deep Reinforcement Learning technique in the platform game Super Mario Bros. In particular, the research consists in implementing the Q-Learning algorithm, one of the most relevant Deep Reinforcement Learning technique algorithms.

Furthermore, due to the problem complexity, convolutional neural networks have been introduced, what leads to Deep Q-Learning (DQN). By using this algorithm we try to achieve that the videogame character is able to pass the levels which are set out to him by means of the experiences he has acquired throughout his exploration of those levels.

In order to apply this algorithm, Python programming language has been used, which is the most powerful one to develop artificial intelligence systems, with its

libraries TensorFlow and Keras for the neural networks implementation.

Along this work, we expound the analysis, the design and the system implementation, in addition to the results and their interpretation. Finally, a series of conclusions are described as well as a future research proposal with the aim of increasing and improving the present one.

Índice

Lista de figuras

Lista de tablas

Lista de ecuaciones

Capítulo 1 Introducción

El primer capítulo de esta memoria estará dedicado en su totalidad a la introducción y presentación del trabajo. El contenido de este está enfocado en presentar el tema que se va a tratar y su contexto, la motivación personal que me ha hecho desarrollar un trabajo sobre este tema, describir cuáles son los objetivos específicos y generales que se pretenden conseguir con su desarrollo y la metodología seguida junto con su planificación temporal. De manera adicional se incluye un resumen de los productos obtenidos a lo largo de su desarrollo junto, con una breve explicación de cada uno de los capítulos posteriores.

1.1 Contexto y justificación del Trabajo

Hoy en día las técnicas de *machine learning* son cada vez más importantes y utilizadas en el sector empresarial en lo que automatización se refiere. *Machine learning* consiste en la recogida y análisis de datos para elaborar un modelo predictivo descubriendo patrones o comportamientos en un fenómeno determinado. En la actualidad, donde cada vez hay mayores interacciones entre sistemas y usuarios, las técnicas de *machine learning* son cada vez más importantes.

Hay muchos sectores que se pueden beneficiar del uso de estas técnicas, sobre todo donde se genera un gran volumen de datos y donde exista un conjunto de acciones repetitivas que puedan ser optimizadas mediante estos algoritmos. Pueden suponer reducción de costos y aumento de la eficiencia y eficacia de los sistemas. Una de las aplicaciones más interesantes en el aprendizaje máquina es el aprendizaje de refuerzo (RL). La idea principal se basa en que un agente intenta aprender, interactuando con un entorno y utilizando la experiencia adquirida, para alcanzar de manera óptima algún objetivo dado en forma de recompensas acumulativas. Aprender a controlar los agentes provenientes de entradas sensoriales complejas tales como la visión y el habla es uno de principales desafíos y objetivos del aprendizaje de refuerzo. La mayoría de las aplicaciones exitosas de RL que operan en estos dominios se han basado en características hechas a mano combinadas con funciones de valor lineal o

representaciones políticas. Como es obvio, el rendimiento de estos sistemas depende en gran medida de la calidad de la representación de las características.

Tras el descubrimiento del Aprendizaje Profundo (Deep Learning), el aprendizaje por refuerzo profundo (Deep Reinforcement Learning) se ha hecho muy popular debido a su éxito. Con este tipo de técnicas de aprendizaje se ha conseguido superar a los humanos en los juegos de Atari (2013) y a los campeones de GO con DeepMind Alpha Go Zero (2016).

Ilustración 1.1. Capturas de 5 juegos de Atari

Esto se debe a que es una técnica muy potente para espacios de estado de altas dimensiones donde los algoritmos de iteración de Valor y Q son inútiles y los algoritmos de aproximación son más adecuados, haciendo así más fácil la aproximación de las funciones y políticas de valor a partir de datos de entrada muy complejos. Los videojuegos representan simulaciones perfectas para tratar los problemas del mundo real por lo que podemos, entonces, utilizarlos para estudiar y probar el rendimiento de los nuevos enfoques.

El aprendizaje profundo a lo largo de los últimos años ha hecho enormes avances, donde podemos destacar la extracción de características de datos sensoriales complejos (imágenes en nuestro caso), sin embargo, se enfrenta a muchos desafíos. En primer lugar, las aplicaciones que aplican algoritmos de esta índole creadas hasta la fecha han requerido grandes cantidades de datos y memoria. Por otra parte, los algoritmos RL deben aprender una señal de recompensa que con frecuencia es muy ruidosa.

1.1.1 Motivación

Tras cursar la asignatura de Minería de Datos en mis estudios de grado me he interesado bastante en el ámbito de la inteligencia artificial, ya que me parece un tema muy interesante, nuevo y explotable con capacidades aún por descubrir. Como Trabajo de Fin de Grado realicé una implementación de los algoritmos de aprendizaje

por refuerzo SARSA y Q-Learning aplicados al ajedrez y me gustaría continuar con la investigación y el aprendizaje sobre el tema a través del desarrollo de este Trabajo de Fin de Máster. En esta ocasión he seleccionado los juegos de plataformas de la consola Super Nintendo, más en concreto Super Mario Bros, de tal forma que me pueda abstraer de alguna manera a mi infancia y combinar el ámbito al que quiero dedicarme en un futuro (Inteligencia Artificial) junto con lo que tantas horas de entretenimiento me ha regalado cuando era un niño.

1.2 Objetivos del Trabajo

En este epígrafe se van a describir los requisitos que se tienen que cumplir una vez que el proyecto esté finalizado.

El objetivo principal del desarrollo de este proyecto es desarrollar un algoritmo para aplicar la técnica de aprendizaje automático Deep Q-Learning a un videojuego de plataformas y, así, conseguir que el personaje del videojuego aprenda una correspondencia entre el estado actual del juego, las posibles acciones y las recompensas obtenidas que le hagan capaz de superar los niveles de forma autónoma. Para lograr este objetivo han de cumplirse los siguientes objetivos secundarios:

- Entender el problema a tratar mediante un análisis y estudio del área de aprendizaje por refuerzo.
- Analizar y explicar las diferentes tareas que se realizarán, junto con su planificación temporal
- Aprender a utilizar la herramienta de Python OpenA IGym
- Lograr un algoritmo escalable que se pueda aplicar, no solo a diferentes niveles dentro de un videojuego, sino a videojuegos similares, aunque tengan diferentes características
- Elaborar una comparativa de rendimiento y eficacia al utilizar diferentes formas de recoger la información de los estados del videojuego que hagan al personaje aprender.
- Entender el problema a tratar mediante un análisis y estudio del área de aprendizaje por refuerzo

- Investigar trabajos similares
- Realizar un estado del arte en el que se explica todo lo relacionado con el aprendizaje profundo
- Recoger la información visual del videojuego mediante código
- Reducir dimensionalidad de imágenes
- Entender e implementar el algoritmo Q-Learning
- Entender los diferentes modelos de redes neuronales y ser capaz de realizar su implementación
- Ser capaz de analizar y entender los resultados obtenidos en cada experimento realizado al final del proyecto
- Lograr un algoritmo escalable a diferentes niveles e incluso a diferentes juegos
- Presentar un conjunto de conclusiones obtenidas tras la realización del TFM junto con el posible trabajo futuro en base a los conocimientos obtenidos durante su desarrollo

1.3 Enfoque y método seguido

La inteligencia artificial y, más en concreto, el ámbito del aprendizaje máquina es un campo de investigación cuya aparición es muy reciente, lo cual nos puede llevar a pensar que no hay suficiente información sobre el tema. Esto, en realidad, no es así, existe gran cantidad de libros, documentación, artículos, trabajos y proyectos en los que se explican o utilizan las técnicas de aprendizaje máquina y aprendizaje profundo. Como hay gran cantidad de información, no hay ningún problema en recopilar fuentes que nos permitan documentarnos antes de comenzar el proyecto.

El punto de partida del proyecto será la formación, es decir, leer y aprender sobre qué significa el aprendizaje profundo y cómo se utiliza. Otro punto de apoyo muy importante para el comienzo del proyecto es el estudio de trabajos similares, lo cual puede ayudar a entender cómo se enfoca un proyecto de estas características. Para recopilar la información es muy recomendable utilizar buscadores académicos como Dialnet, Link o Google Scholar, no solo los buscadores tradicionales como Google o Bing.

Una vez que se ha entendido el marco del problema y se sabe cómo enfocar el proyecto, el siguiente paso es pensar y buscar qué herramientas son las que nos van

a permitir desarrollar el trabajo. Como punto de apoyo tenemos la asignatura de Inteligencia Artificial que, junto con las recomendaciones del tutor, nos ha hecho tomar la decisión de utilizar el lenguaje de programación Python junto con la biblioteca de código abierto de redes neuronales Keras sobre TensorFlow.

Tras la selección de las tecnologías presentes en el proyecto se procederá al desarrollo del mismo, el cual consistirá en la implementación de un algoritmo con varios modelos de redes neuronales diferentes sobre los que se van a realizar diversos experimentos. A partir de estos experimentos realizados se obtendrán una serie de resultados a interpretar y comentar, a partir de cuales se van a obtener las conclusiones finales del proyecto.

Todo este largo proceso será acompañado durante los meses de desarrollo con la elaboración de una memoria explicativa que, junto con el código y una presentación, será la entrega final del proyecto.

1.4 Planificación del Trabajo

Uno de los puntos más importantes para el desarrollo de cualquier proyecto es la división en tareas del mismo y la estimación temporal de cada una de las tareas. Para esta planificación hay que tener en cuenta la disponibilidad laboral, que será de 2:30 horas de lunes a viernes, 4 horas los sábados y 4 horas los domingos a partir del 2 de diciembre de 2019. La estimación se ha planteado de tal manera ya que hay que compaginar el proyecto con el transcurso del resto de asignaturas del máster y el trabajo, cuya jornada es de lunes a viernes de 8 a 13 horas.

Las tareas principales, junto con la división de cada una de ellas en subtareas y su estimación temporal las podemos encontrar en la siguiente tabla.

	TAREAS	TIEMPO
1	**ESTADO DEL ARTE**	
1.1	Recopilación bibliográfica	6
1.2	Búsqueda y estudio de trabajos similares	5
2	**ANÁLISIS**	
2.1	Captación de requisitos	1
2.1	Estudio y elección de técnica de aprendizaje	5
2.3	Investigación y estudios de trabajos similares	6
3	**DISEÑO**	
3.1	Casos de uso	5
3.2	Diagrama de clases	3
4	**IMPLEMENTACIÓN**	
4.1	Implementación de agente	20
4.2	Implementación de arquitecturas de modelos	25
4.3	Implementación Q-Learning	22
5	**PRUEBAS**	
5.1	Ejecución de pruebas	20
5.2	Estudio y razonamiento de resultados obtenidos	19
6	**DEFENSA**	
6.1	Redacción de memoria	125
6.2	Desarrollo de presentación	5
6.3	Defensa de TFM	10
	Días TOTALES	140

Tabla 1.1. Planificación de tareas

Podemos observar que la estimación temporal del proyecto, incluida la defensa final, es de 140 horas. De un modo más claro, podemos presentar estas tareas en un diagrama de Gantt como el siguiente:

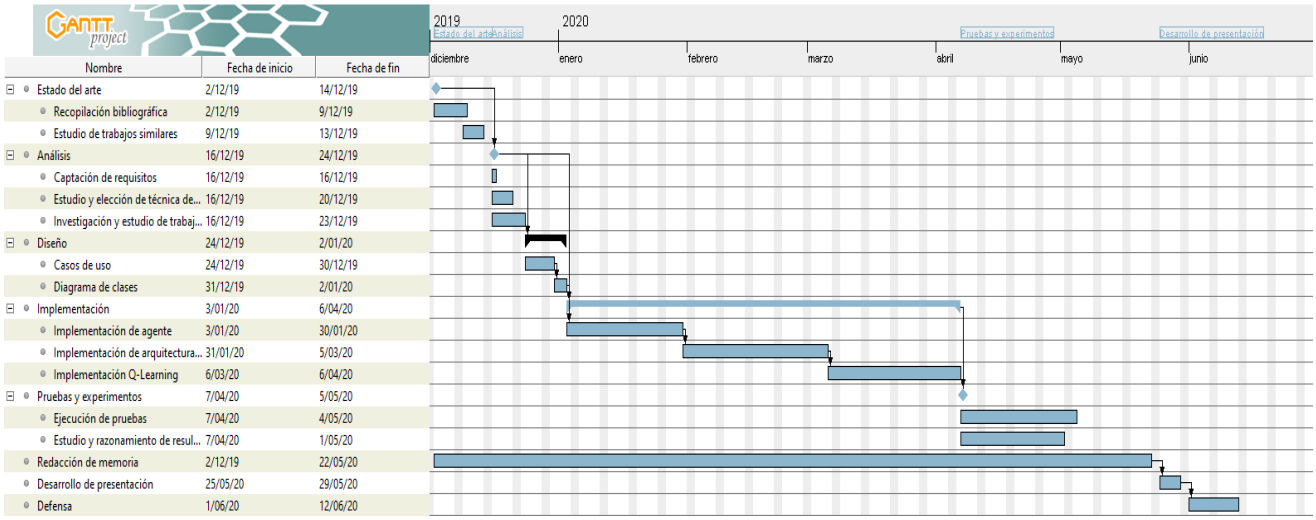

Ilustración 1.2. Diagrama de Gantt

1.5 Breve sumario de productos obtenidos

Tras el desarrollo de este TFM, los productos obtenidos serán los siguientes: código fuente, memoria, presentación y vídeo de resultados. La memoria se trata del documento presente, en la cual se describe toda la información del proyecto junto con los conocimientos adquiridos durante el desarrollo del mismo. Además de la memoria se entregará una carpeta que contendrá el código fuente junto con un vídeo que muestra algunos resultados obtenidos durante el proceso de aprendizaje del algoritmo desarrollado. Para finalizar, en días posteriores a la entrega de estos archivos será entregada una breve presentación que será utilizada para la defensa pública del TFM ante el tribunal.

La estructura del código fuente y los resultados obtenidos se van a explicar más en detalle en próximos capítulos presentes en esta memoria

1.6 Breve descripción de los otros capítulos de la memoria

A continuación, se va a presentar un breve resumen del contenido que podremos encontrarnos dentro del documento y cómo se está estructurado.

Este primer capítulo se trata de la presentación del TFM, en el que están incluidos los epígrafes obligatorios comunes para todos los proyectos desarrollados con la plantilla facilitada por la universidad. A grandes rasgos, la utilidad de este capítulo es poner el en contexto el proyecto, exponer sus objetivos, la planificación temporal y la metodología empleada para el desarrollo del mismo

En el capítulo 2 nos centraremos en el estado del arte del proyecto. En este apartado se analiza de una forma global el área del aprendizaje por refuerzo y las redes neuronales. Una vez entendido cómo funcionan estas metodologías es cuando

se puede empezar a desarrollar un proyecto de esta índole, por eso el estado del arte es uno de los puntos más relevantes del trabajo.

El tercer capítulo es el encargado de presentar las tecnologías utilizadas para desarrollar los conocimientos obtenidos durante la recopilación bibliográfica y el desarrollo del estado del arte. Se utilizará el lenguaje de programación Python con la librería de redes neuronales Keras sobre TensorFlow.

En el cuarto capítulo se explica todo el proceso de desarrollo del proyecto. Primero de todo se explica qué videojuego se utilizará para aplicarle el algoritmo a implementar, en este caso Super Mario Bros. A continuación, se describe todo el proceso natural del desarrollo de un proyecto: análisis, diseño e implementación. En el apartado de análisis identificaremos los diferentes casos de uso englobados por el sistema, junto con una explicación detallada de los mismos. En la fase de diseño se define el diagrama de clases. Para finalizar el capítulo se describe todo el proceso de implementación del algoritmo, con una explicación detallada de las funciones y modelos de redes neuronales desarrollados.

El quinto capítulo será donde se expondrán todas las pruebas realizadas y los experimentos, junto con una explicación, interpretación y razonamiento de los resultados obtenidos en cada una de las pruebas.

En el sexto capítulo se desarrollarán todas las conclusiones que hemos obtenido a partir del cumplimiento, o no, de los objetivos establecidos en el inicio del proyecto.

El séptimo capítulo consiste en el glosario de la terminología y acrónimos utilizados en la memoria.

El octavo capítulo es donde se presentan las referencias bibliográficas consultadas durante el desarrollo del proyecto

Capítulo 2 Estado del arte

En este apartado del proyecto se expone la información general para comprender el trabajo llevado a cabo. Este capítulo tiene como objetivo introducirnos al aprendizaje automático y, más en específico, aprendizaje por refuerzo profundo, para comprender su funcionamiento y métodos y cómo aplicar lo aprendido a los juegos de plataformas Python OpenAI Gym.

Se va a dividir en diferentes apartados donde, en cada uno de ellos, se tratará uno de los siguientes aspectos:

- Aprendizaje automático
- Aprendizaje por refuerzo
- Redes neuronales y aprendizaje por refuerzo profundo
- OpenAI Gym
- Marco del trabajo

A continuación, vamos a desarrollar cada uno de los apartados mencionados en más profundidad.

2.1 Introducción al aprendizaje automático

En este primer epígrafe se va a definir el aprendizaje automático para poder entender el contexto en el que está basado el proyecto. Se presentará de forma breve su historia.

2.1.1 Contexto

Uno de los campos de investigación que más ha evolucionado en los últimos años dentro de la informática ha sido la Inteligencia Artificial. Ahora bien, ¿a qué se denomina Inteligencia Artificial?

Una definición amplia es la que se nos presenta en la Encyclopedia Of Artificial Intelligence (Stuart C**.,** 1992):

"*La Inteligencia Artificial es un campo de la ciencia y la ingeniería que se ocupa de la comprensión desde el punto de vista informático, de lo que denomina comúnmente comportamiento inteligente- También se ocupa de la creación de artefactos que exhiben este comportamiento*".

2.1.2 Definición

Para hablar de aprendizaje máquina, el primer nombre que cabe destacar es el de Arthur Lee Samuel, uno de los pioneros en el campo de los juegos informáticos y la inteligencia artificial. Este definió, en 1959, el Aprendizaje Automático como "*Un campo de estudio que otorga a los ordenadores la capacidad de aprender sin estar programados explícitamente*".

Kevin P. Murphy en su libro *Machine learning, a probabilistic perspective* (2012) aporta una definición bastante clara y completa. Para él, el aprendizaje automático o aprendizaje máquina es "*Una serie de métodos que pueden detectar, automáticamente, patrones en un conjunto de datos concretos y entonces usar dichos patrones para predecir datos futuros, o también, llevar a cabo cualquier otro tipo de toma de decisiones bajo incertidumbre.*"

Estas dos, son buenas definiciones, hay que destacar la de Tom Mitchell, otra persona con gran reconocimiento en el ámbito del aprendizaje máquina. En 1998 propuso una de las definiciones más completas: "*Un programa que es capaz de aprender de cierta experiencia E, con respecto a una tarea T con respecto a una medida de rendimiento R, siempre que el sistema mejore el rendimiento R en la tarea T siguiendo la experiencia E*". A partir de este conjunto de diferentes definiciones se puede deducir que el aprendizaje automático está basado en crear máquinas o dispositivos que sean capaces de adquirir conocimiento de algún ámbito concreto de forma autónoma para, entonces, poder desarrollar un comportamiento inteligente de manera que no se les indiquen a dichas máquinas de forma exacta el comportamiento o las pautas a seguir, como ocurriría en la programación clásica.

En el libro “Machine Learning, An artificial intelligence approach” (1983) podemos encontrar de una forma más concreta los objetivos del Aprendizaje Automático, son los siguientes:

- **Estudios orientados en tareas:** El análisis y desarrollo de los sistemas de aprendizaje para mejorar el rendimiento en un predeterminado conjunto de tareas (también conocido como “*engine approach*”).
- **Simulación cognitiva:** La investigación y la simulación por ordenador de los procesos de aprendizaje humano.
- **Análisis teórico:** La exploración teórica del espacio de posibles métodos de aprendizaje y algoritmos sin importar el dominio de las aplicaciones.

2.1.3 Tipos

En el campo del aprendizaje automático existen distintos tipos de técnicas y algoritmos. Haciendo una clasificación muy genérica, se puede dividir el aprendizaje automático en dos tipos:

- **Aprendizaje no supervisado**: En este tipo de aprendizaje no existe un conocimiento previo del problema o el sistema que estemos tratando, lo que quiere decir que se conocen los valores de entrada, pero no hay información de que salidas deberíamos obtener a partir de dichas entradas. Estos algoritmos buscan algún tipo de estructura en el conjunto de datos para realizar los entrenamientos, cómo encontrar qué ejemplos son similares a otros y agruparlos en “*clusters*”.
- **Aprendizaje supervisado:** En este caso sí se dispone de un conocimiento a priori de datos de entrenamiento que se utilizan a modo de referencia o ejemplos. Para cada ejemplo se tiene el dato de entrada y la salida o etiqueta correcta, de manera que por cada acción realizada se hará una comparativa con la salida obtenida y la salida que se debería obtener y así realizar el entrenamiento.

Además de estos dos tipos existe uno adicional que será en el que se basa este proyecto, el aprendizaje por refuerzo. Este método se considera por muchos autores y

expertos como un tipo de aprendizaje no supervisado ya que a priori no disponemos del conjunto de valores de salida para determinar si una acción es la correcta dependiendo del estado en que nos encontremos. Por otro lado, existen autores que defienden el aprendizaje por refuerzo como un aprendizaje supervisado debido a que es necesario determinar, de manera previa al aprendizaje, unas medidas de satisfacción conocidas como recompensas o rewards, que indican al sistema como de buena o mala es la acción elegida en un instante concreto, en función de estado.

A partir de estas diferentes formas de pensar de los expertos en el tema, de una manera aproximada, se podría diferenciar el aprendizaje por refuerzo como un método de aprendizaje que no es ni supervisado ni no supervisado, sino que se encuentra en un punto intermedio. Más claro y conciso, podemos deducir lo siguiente:

	APRENDIZAJE NO SUPERVISADO	**APRENDIZAJE SUPERVISADO**	**APRENDIZAJE POR REFUERZO**
ENTRADA	SÍ	SÍ	SÍ
SALIDA	NO	SÍ	RECOMPENSAS

Tabla 2.1. Comparativa de tipos de aprendizaje

2.2 Redes Neuronales

Las redes neuronales artificiales han sido creadas a partir de las redes neuronales biológicas y su objetivo es intentar hacer que los ordenadores aprendan de una forma similar a la forma de aprender de los humanos simulando el comportamiento del cerebro. El cerebro humano tiene neuronas interconectadas con dendritas que reciben impulsos o entradas para, en función de las entradas recibidas, producir una señal de salida eléctrica a través de su axón. El funcionamiento de las redes neuronales no se fundamenta en resolver problemas complejos siguiendo una secuencia ordenada de pasos, se basa en la evolución de un algoritmo inspirado en el cerebro humano, constituyendo de esta forma una parte muy importante dentro del ámbito de la inteligencia artificial. Existen problemas que son complicados para los

humanos (por ejemplo, problemas aritméticos muy complejos) y, a su vez, existen problemas muy sencillos para los humanos, pero muy difíciles para las máquinas (por ejemplo, reconocer a una persona en una foto). Las redes neuronales intentan resolver problemas que serían muy sencillos para las personas, pero complejos de llevar a cabo por los ordenadores.

Una red neuronal artificial está formada por los siguientes componentes esenciales:

- Conjunto de nodos o neuronas que sirven para procesar la información.
- Conexiones entre las neuronas de cada una de sus capas.
- Una variable de estado de activación.
- Regla de propagación de las señales de salida.
- Regla de activación, combinación y modificación.
- Una función de salida por cada una de las neuronas de la red.

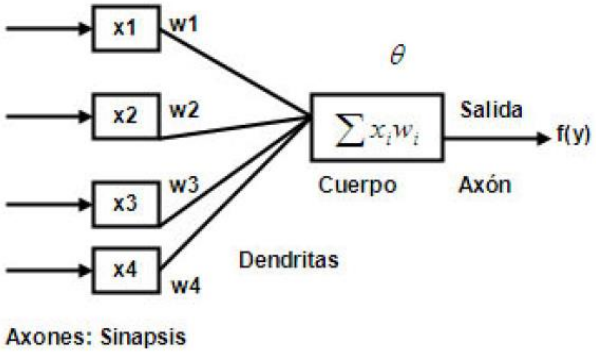

Ilustración 2.1. Ejemplo de Neurona Artificial

2.2.1 Single Perceptrón

Antes de entrar en detalles se va a explicar el modelo más importante dentro de las Redes Neuronales, el perceptrón. Esta fue la primera red neuronal artificial, creada por Rosenblatt, quien la describió por primera vez en 1958.

El perceptrón es la forma más sencilla de una red neuronal artificial utilizada para la clasificación de patrones binarios que son linealmente separables y, por

consiguiente, separables sin dificultad por un hiperplano. Los clasificadores binarios son funciones capaces de decidir si una entrada de datos, representada por un vector numérico, pertenece a una clase o a otra. A grandes rasgos el perceptrón se fundamenta en una única neurona la cual recibe tantas entradas como componentes tenga el vector numérico, a las cuales se les asigna un peso *w* y una conexión a la red.

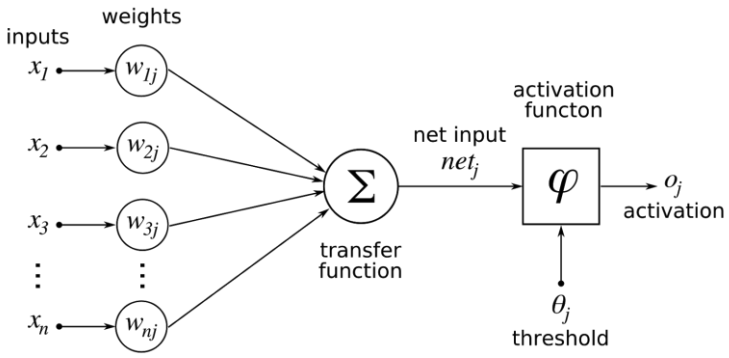

Ilustración 2.2. Estructura de perceptrón

La red neuronal (*transfer function*) calcula su salida mediante una combinación lineal de sus entradas aplicándoles su correspondiente peso asignado a cada una, junto con un sesgo *w0* aplicado desde el exterior. La suma resultante se aplica a una función de activación dada por la ecuación 2.1, donde la entrada de dicha función es cada *x* (con *x0* = 1) representa una entrada de la red a la que se le aplica su peso asociado *w y*, además el sesgo *w0.*

$$y = F[\sum_{i=1}^{p} w_i x_i + w_0] = F[\sum_{i=1}^{p} w_i x_i] = F[W^T X]$$

Ecuación 2.1. Función de activación en perceptrón

Los pesos de conexión se ajustan de forma global para optimizar el rendimiento de la red a través de una función de coste.

2.2.2 Funciones de activación

Las funciones de activación reciben las señales de entrada a la que se lo aplican los pesos asociados para, a continuación, calcular una salida y realizar una clasificación de los datos de entrada. Los subapartados presentan a continuación explican las diferentes funciones de activación que están relacionadas con el aprendizaje automático (John F. y Stefan C., 2001).

2.2.2.1 Función Sigmoide

La función Sigmoide es la función de activación más antigua y popular dentro del mundo de las redes neuronales. Han sido la base de la mayoría de redes neuronales utilizadas durante décadas, pero hoy en día han decaído en uso, eso es debido a que las redes de muchas capas presentan una *desaparición de gradiente,* lo que puede saturar la red neuronal ralentizando el proceso o, incluso, hacerla incapaz de aprender de forma exacta. A pesar de estos problemas, estas funciones se siguen utilizando en casos concretos de tareas de clasificación. Está definida mediante la ecuación

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Ecuación 2.2. Función sigmoide

Como podemos ver en la ilustración 2.3, las salidas de esta función están acotadas entre 0 y -1

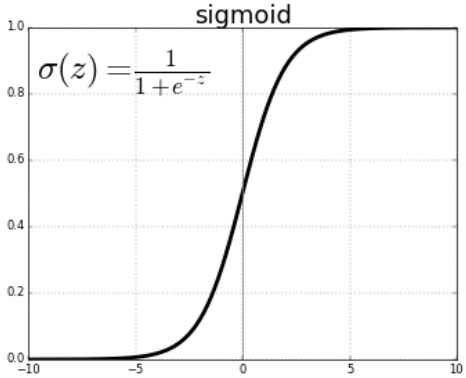

Ilustración 2.3. Función de activación sigmoide

A modo de resumen tenemos las siguientes características:

- Converge lento
- Satura el gradiente
- Acotada entre 0 y 1
- Buen rendimiento en su última capa

2.2.2.2 Función Tanh

La función de activación Tangente Hiperbólica es muy similar a la función Sigmoide, pero presenta dos diferencias importantes. La primera reside en los límites, la función Tanh se encuentra acotada entre -1 y 1, y la segunda es que la función se encuentra centrada en el cero, de esta forma podemos clasificar decidiendo entre una opción o la contraria. La ecuación matemática que representa esta función es:

$$f(x) = \frac{2}{1 + e^{-2x}} - 1$$

Ecuación 2.3. Función de activación Tanh

Las diferencias con la función sigmoide las podemos observar en la ilustración 2.4.

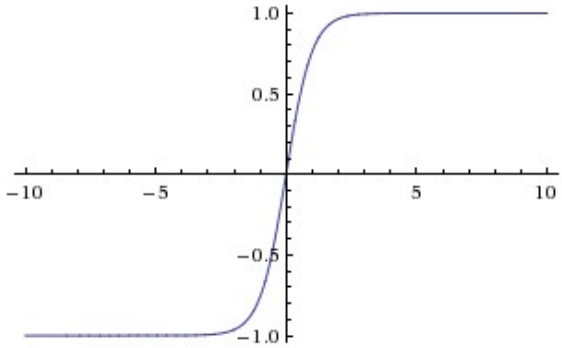

Ilustración 2.4. Función de activación Tanh

Sus características, muy similares a la función explicada antes, son las siguientes:

- Satura el gradiente
- Converge muy despacio.
- Acotada entre -1 y 1
- Clasifica entre opciones opuestas (positivo/negativo)
- Eficaz en redes neuronales donde existe recurrencia
- Está centrada en 0
- Más eficiente de calcular que Sigmoide

2.2.2.3 Función ReLU

La función de activación de la unidad lineal rectificada, también conocida como ReLU devuelve como salida la entrada recibida anulando los negativos, de manera que, si llega un número negativo como entrada, la salida será 0. Este tipo de activación hace a las redes neuronales converger mucho más rápido. La ecuación 2.4, junto con la ilustración 2.5 describen de forma gráfica el comportamiento de esta activación

$$f(x) = \max(0, x) = \begin{cases} 0 \; for \; x < 0 \\ x \; for \; x \geq 0 \end{cases}$$

Ecuación 2.4. Función de activación ReLU

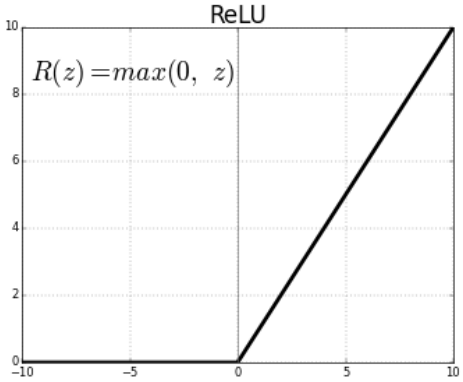

Ilustración 2.5. Función de activación ReLU

Las características de ReLU son:

- Sólo se activa con entradas positivas
- No está acotada entre ningún valor
- Funciona muy bien si recibe imágenes como entrada
- Elimina el problema de degradación del gradiente, lo que permite una propagación eficiente en redes profundas

2.2.2.4 Función Softmax

ReLU, sigmoide y Tangente Hiperbólica sin funciones de activación muy recomendadas y utilizadas en las redes neuronales, sin embargo, si se quiere tratar un problema de clasificación, presentan algunos inconvenientes. Para solventar estos problemas se utiliza la función de activación Softmax. La suma de las salidas de esta función es 1, por lo que esta salida equivale a una distribución categórica de probabilidades. Se representa a mediante la siguiente ecuación:

$$f(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}$$

Ecuación 2.5. Función de activación Softmax

Algunas de las características de la activación Softmax son:

- Se encuentra acotada entre 0 y 1.
- La representación obtenida mediante esta salida es en forma de probabilidades
- Utilizada para normalizar
- Alto rendimiento en las capas finales

2.2.3 Función de coste

Antes, en el epígrafe 2.2.1 ha sido mencionado con brevedad el concepto de función de coste, el cual es una de las formas de medir la precisión de la red neuronal entrenada. La función de coste es una métrica de evaluación de rendimiento de los modelos de redes neuronales, muy importante para la fase de entrenamiento. Esta precisión representa el porcentaje de clasificaciones o predicciones realizadas de forma precisa por la red en relación a una salida conocida. Existen varias funciones de coste, como por ejemplo las siguientes (Simon Haykin, 1993):

- *Sum squared error*: Se trata de la función más común dentro de los modelos de redes neuronales, cuya fórmula es

$$C = \frac{(y - y')^2}{2}$$

Ecuación 2.6. Sum squared error

- Función de coste de entropía cruzada, cuya expresión es

$$C = -\frac{1}{n}\sum_x [y \ln y' + (1-y)\ln(1-y')]$$

Ecuación 2.7. Función de coste de entropía cruzada

La variable n se corresponde con el total de datos del conjunto de entrenamiento. La entropía cruzada es positiva y tiende hacia cero a medida que la neurona devuelve resultados más precisos para cada una de las entradas.

2.2.4 Algoritmo de entrenamiento

La estimación de los pesos de las redes neuronales se consigue gracias a algoritmos iterativos basados en el gradiente. Estos se corresponden con algoritmos de adaptación, aprendizaje o entrenamiento que tratarán de optimizar la función de coste en términos de peso, es decir, conseguir calcular unos pesos que hagan a la red minimizar el valor de dicha función (Simon Haykin, 1993). Este algoritmo suele ser denominado, también, algoritmo de retro propagación (*back-propagation*). El proceso de entrenamiento de dicho algoritmo se divide en dos fases:

- **Fase de avance (*forward phase*)**. En esta primera fase del entrenamiento, los pesos de la red se transmiten y la señal de entrada se propaga a través de la misma, capa por capa, hasta alcanzar la salida. Todos los cambios producidos en esta fase están directamente relacionados con las funciones de activación utilizadas en cada capa y las salidas de las neuronas

- **Fase de retroceso (*backward phase*)**. En la fase de retroceso se produce una señal de error tras comparar la salida de la red con la salida esperada o real. Esta señal de error de propaga a través de la red, capa a capa, en sentido inverso. A lo largo de esta propagación se realiza un ajuste de los pesos de la red.

2.2.5 Ajuste de pesos

Para llevar a cabo el ajuste de pesos que se produce durante la fase de retroceso del periodo de entrenamiento es necesario aplicar unas técnicas específicas. Estas técnicas son conocidas como **Optimizadores.** Los optimizadores son utilizados para minimizar la salida de la función de coste. La elección entre un optimizador u otro puede tener un fuerte impacto en los resultados, sobre todo en el tiempo de entrenamiento de las redes neuronales. Podemos destacar los más importantes:

- **Descenso de gradiente (GD)** (Simon Haykin, 1993): Se trata del optimizador de primer orden más común, cuyo fundamento es buscar mínimos de forma gradual siguiendo la siguiente expresión matemática donde α es la ratio de aprendizaje:

$$w(t+1) = w(t) - \alpha \frac{\delta E}{\delta w}$$

Ecuación 2.8. Expresión matemática de GD

  Es muy común que, en lugar de utilizar de forma continuada la misma ratio de aprendizaje en todos los pasos, este se vaya modificando. Si bien es el optimizador más común, presenta un problema importante debido a su lenta convergencia, lo cual sucede porque calcula el gradiente de todo el conjunto de datos al mismo tiempo. Otro problema añadido es la compatibilidad, es decir, si se incluyen nuevos datos en el conjunto será necesario repetir todo el proceso de entrenamiento desde cero.
- **Descenso de gradiente estocástico (SGD)** (Simon Haykin, 1993): Se trata de una variante de GD, solo que mucho más rápido debido a que calcula el descenso de gradiente a diferentes subconjuntos en lugar de utilizar el conjunto de datos completo
- **Algoritmo de gradiente adaptativo (Adagrad)** (Diederik P. y Jimmy, 2014) Se trata de una modificación de GD en el que se utilizan diferentes ratios de aprendizaje. Las ratios de aprendizaje son actualizadas en cada iteración, adaptándose a los componentes, y por cada parámetro almacenamos la suma de los cuadrados y el gradiente

histórico de los mismos. Tomando G como el gradiente histórico, tenemos la siguiente representación matemática del optimizador:

$$\theta^k = \theta_{k-1} - \frac{\alpha}{sqrt(G^{k-1}}.\nabla J(\theta_{k-1})$$

Ecuación 2.9. Expresión de Adagrad

- **Estimación del momento adaptativo (Adam)** (Diederik P. y Jimmy, 2014)**:** Adam es una variante de Adagrad en el que, para actualizar la ratio de aprendizaje, se calcula una combinación lineal entre el gradiente y el incremento anterior. Para garantizar las actualizaciones de la ratio de aprendizaje para cada elemento tiene en cuenta los gradientes aparecidos recientemente. Es un optimizador eficiente, sencillo y consume poca memoria, por ello será el elegido para el desarrollo del proyecto

2.3 Aprendizaje profundo

Una de las técnicas más potentes dentro del ámbito del aprendizaje automático es el Aprendizaje Profundo, también conocido en inglés como *Deep Learning.*

El aprendizaje profundo es una forma de aprendizaje automático que permite a los ordenadores aprender de las experiencias recibidas para así aprender y entender el mundo en términos de jerarquía de conceptos. Como un ordenador obtiene los conocimientos mediante la experiencia no es necesario que un usuario humano especifique todos los conocimientos que estos algoritmos requieren, aquí reside su gran potencial. La jerarquía de conceptos hace referencia a que un ordenador es capaz de aprender conceptos complicados a partir de otros conceptos más sencillos. Esta jerarquía dispondría de una gran cantidad de capas de profundidad. A grandes rasgos lo que esta técnica intenta es simular el funcionamiento del cerebro humano. (Ian, Yoshua y Aaron, 2016)

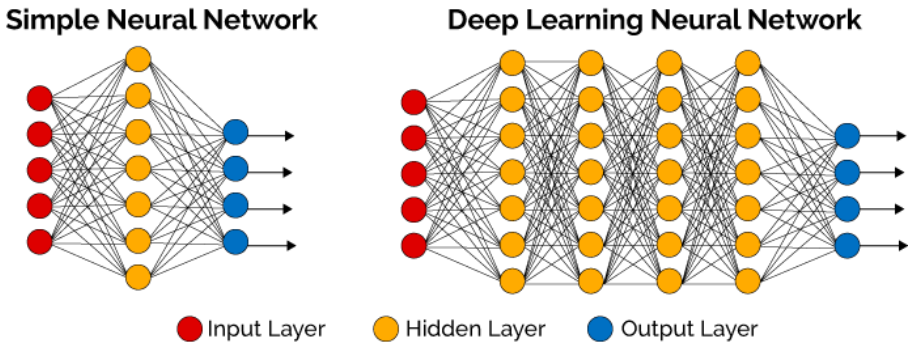

Ilustración 2.6. Comparativa red neuronal y red neuronal profunda

A modo de comparativa tenemos la ilustración 2.6 mostrada arriba, en la que se muestra una red neuronal normal junto con una red neuronal profunda de izquierda a derecha respectivamente. Podemos observar en esta imagen la clara diferencia que existe entre los dos tipos de redes neuronales, básicamente una red neuronal profunda (DNN) no es otra cosa que una red neuronal artificial con múltiples capas ocultas.

El auge de *Deep Learning* se dio a finales de la década de 2000, cuando hasta entonces no existía una forma fiable y potente de entrenar redes neuronales con muchas capas (muy profundas), es decir, las redes hasta el momento utilizadas eran demasiado superficiales y solo aprovechaban las primeras capas. Tras los avances en hardware aparecidos a partir de la segunda década de los años 2000, y más concretamente en GPU y capacidad de almacenamiento, los problemas que presentaban los modelos neuronales existentes desaparecieron, hasta tal punto que hoy en día podemos entrenar modelos complejos en ordenadores portátiles personales.

.

2.3.1 Convolutional Neural Networks

Las redes neuronales convolucionales (CNN) (Simon, 1993) son un tipo de redes neuronales artificiales que simulan la visión del cerebro humano, es decir, sus entradas son imágenes o mapas de características. Reciben este nombre realizan una operación matemática conocida como convolución, en lugar de la multiplicación de matrices general, en una de sus capas como mínimo. La convolución no es más que aplicarles un filtro a las imágenes para reducir su tamaño, por consiguiente, que su uso es muy adecuado para el reconocimiento y clasificación de imágenes. En la

ilustración 2.7 podemos ver el comportamiento de una capa que utiliza el operador de convolución.

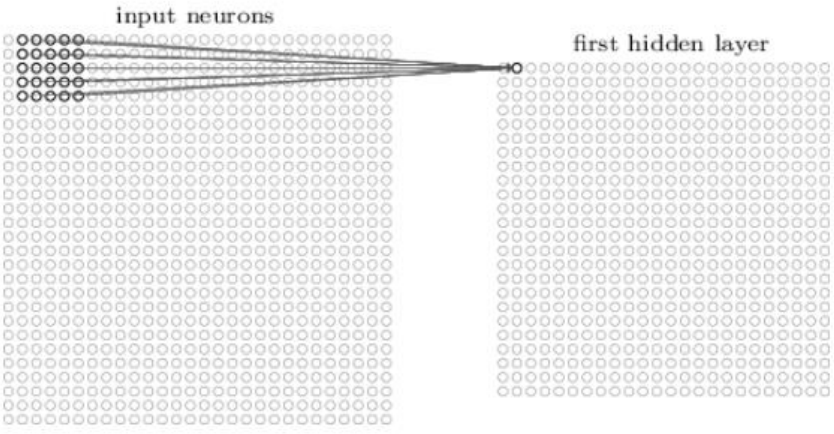

Ilustración 2.7. Capa de red convolucional

Existen diferentes tipos de capas en una red neuronal convolucional, entre las que podemos destacar la capa convolucional, capas de agrupación (pooling layers) y capas completamente conectadas (fully connected layers).

2.3.1.1 Capas convolucionales

Las capas convolucionales son la base de CNN, y realizan la gran mayoría de los cálculos matemáticos. Como se ha mencionado anteriormente en este epígrafe, la operación principal es la convolución, cuyo fundamento es extraer características específicas de las imágenes. Con la convolución se consigue mantener la relación espacial entre los pixeles reduciéndolos a cuadrados más pequeños de características similares, por tanto, un punto a importante a tener en cuenta en esta arquitectura es el filtro o *kernel* utilizado para desarrollar la convolución.

El fundamento consiste en pasar el filtro a través de la imagen de entrada, dividiéndola en matrices de tal forma que se calculan unas nuevas matrices de menor tamaño. Esta matriz resultante es conocida como mapa de características. Esta es una técnica muy utilizada para detectar bordes y enfocar la imagen.

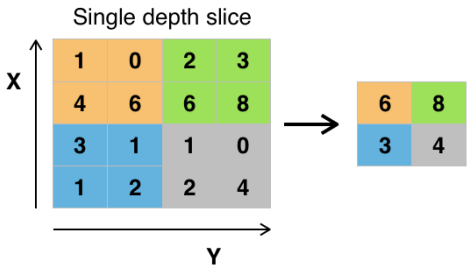

Ilustración 2.8. Ejemplo de max pooling

En función de número de neuronas utilizadas existen 3 hiper parámetros necesarios de ajustar (Ian, Yoshua y Aaron, 2016):

- El número de iteraciones que se utilizan para la convolución, también conocido como profundidad. Depende del tamaño de la imagen, pero lo más común es realizar 64 iteraciones.
- El número de píxeles que vamos a agrupar cuando se aplique el filtrado
- Relleno-cero (*Zero padding*), que se aplica a los bordes para realizar correctamente la convolución. Este parámetro ayuda a tener un control del tamaño de los volúmenes de salida.

En lo que se refiere a las conexiones, en estas arquitecturas cada neurona está conectada a una única región de la imagen de entrada. Esta región local es un hiper parámetro conocido como campo receptivo (*receptive field*), que es igual al tamaño del filtro.

2.3.1.2 Capas de agrupación

Las capas de agrupación son implementadas generalmente justo después de la capa convolucional y son muy recomendables para evitar el sobre entrenamiento. Su funcionamiento consiste en realizar un muestreo que reduce las dimensiones los mapas de características obtenidos de la red convolucional, de manera que se queda con la información más relevante. Existen varias técnicas de agrupación, tales como agrupación máxima o agrupación en la media. Estas técnicas sustituyen agrupaciones de píxeles de las imágenes por un resumen estadístico de los píxeles cercanos. La más utilizada es la agrupación máxima, llamada así porque toma el valor máximo resultado de aplicar el filtro, como hemos podido ver en la ilustración 2.8.

Al igual que ocurre en las capas convolucionales, el principal objetivo de las capas de agrupación es reducir de manera progresiva las dimensiones de las imágenes de entrada para, así, hacerlas más sencillas de tratar.

2.3.1.3 Capas completamente conectadas

Las capas completamente conectadas son un tipo de capas en las que cada neurona de una capa está conectada con todas las neuronas de las capas de la siguiente. Estas usan comúnmente la función de activación Softmax.

A modo de resumen de este epígrafe, en la ilustración 2.9 se presenta una posible estructura de las redes neuronales convolucionales capa a capa.

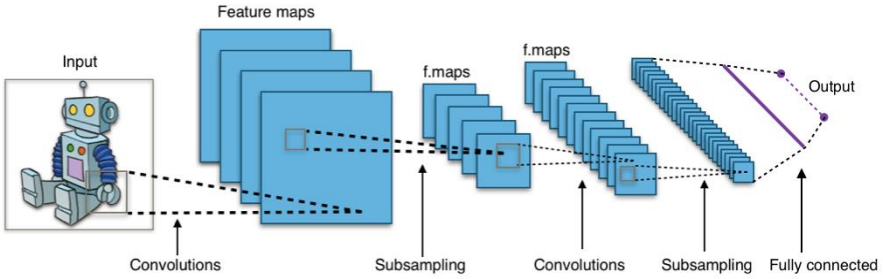

Ilustración 2.9. Resumen CNN

2.4 Aprendizaje por refuerzo

En este epígrafe se va a definir la técnica de aprendizaje por refuerzo para que se pueda entender, de una forma general, el trabajo realizado. Vamos a conocer un poco de historia, los elementos que lo forman y los tipos de algoritmos que pueden ser utilizados.

2.4.1 Introducción

La idea de que nosotros aprendemos en nuestro día a día a partir de la interacción con nuestro propio entorno es probablemente lo primero que se nos viene a la mente cuanto pensamos en el concepto de aprendizaje. Cuando un niño pequeño

juega, saluda con la mano o mira a su alrededor, no tiene un profesor explícito como tal, pero sí que tiene una conexión sensorio-motriz con su entorno. Ejercitar esa conexión produce una enormidad de información sobre causa y efecto, las consecuencias de las acciones y qué hacer para lograr objetivos. A lo largo de nuestras vidas, estas interacciones son indudablemente una gran fuente de conocimiento sobre nuestro medio ambiente y sobre nosotros mismos. Si estamos aprendiendo a conducir o manteniendo una conversación, somos extremadamente conscientes de cómo nuestro entorno responde a lo que nosotros hacemos. Lo que se busca es aplicar este método de aprendizaje a las máquinas y ordenadores, de forma que estos aprendan a encontrar un equilibrio entre la exploración del entorno que estemos tratando (mediante ensayo y error) y la explotación de los comportamientos aprendidos. El concepto de equilibrio se refiere a que se debe explotar los conocimientos aprendidos, pero también es menester explorar para poder realizar las mejores acciones posibles en un futuro.

Barto y Sutton, en su libro *Reinforcement Learning: An Introduction* (2017) explican que el aprendizaje por refuerzo es "*aprender a largo plazo qué hacer, de tal forma que se consiga la máxima recompensa numérica posible. Al agente que aprende no se le dice qué acción debe realizar, sino que debe descubrir qué acción te va a proporcionar la mejor recompensa mediante un entrenamiento. Las acciones puede que no afecten sólo a la recompensa inmediata, sino que pueden afectar a la toma de decisiones futuras.*"

2.4.1.1 Agente

En términos de Aprendizaje por Refuerzo, los agentes son los softwares informáticos que toman las decisiones inteligentes. Los agentes deben ser capaces de predecir qué está ocurriendo en un entorno de manera autónoma. Los pasos básicos de un agente son (Abhishek y Manisha, 2017):

- El agente toma una decisión tras recibir información del entorno
- La decisión que el agente toma se resulta en una acción a realizar
- La acción elegida debe ser la mejor, es decir, la más óptima

2.4.1.2 Entornos

Los entornos en el espacio del Aprendizaje por Refuerzo se componen de ciertas características que determinan el impacto que este tiene sobre un agente, es decir, los agentes deben aprender y adaptarse en base al entorno en el que se encuentren. Estos entornos pueden ser 2D o 3D y los más importantes son los siguientes (Abhishek y Manisha, 2017):

- **Determinista:** Si se puede inferir y predecir lo que sucederá con un determinado escenario en el futuro, decimos que el entorno es determinista. En entornos deterministas los problemas de aprendizaje por refuerzo se resuelven con más facilidad porque no se depende del proceso de toma de decisiones para cambiar de estado.
- **Observable:** Si podemos decir que el entorno que nos rodea es totalmente observable, tenemos un escenario perfecto para implementar el Aprendizaje de Refuerzo. Un ejemplo de perfecta observabilidad es un juego de ajedrez. Un ejemplo de observabilidad parcial es un juego de póquer, donde algunas de las cartas son desconocidas para cualquier jugador.
- **Discreto o continuo:** Si hay más de una opción para la transición al siguiente estado, es un escenario continuo. Cuando hay un número limitado de opciones, se trata de un escenario discreto.
- **Individual o multi agente:** Las soluciones en el aprendizaje de refuerzo pueden ser de tipo agente individual o de tipo multi agente. Si se trata de un problema complejo se utiliza una solución multi agente. Estos problemas complejos pueden tener diferentes entornos no deterministas donde el agente puede realizar diferentes acciones. Son entornos no deterministas porque cuando los agentes interactúan puede haber más de una opción para realizar una transición al siguiente estado, por lo que existe cierta ambigüedad en la toma de decisiones.

2.4.2 Flujo

El modelo de aprendizaje por refuerzo lo mostramos en la siguiente ilustración.

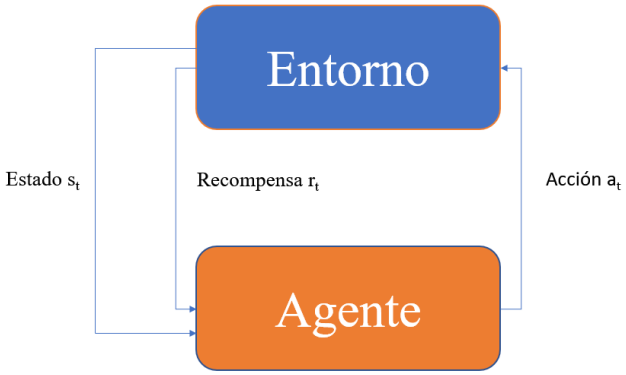

Ilustración 2.10. Arquitectura básica de un sistema de aprendizaje por refuerzo

A raíz de este sencillo esquema podemos deducir que los problemas de aprendizaje por refuerzo poseen principalmente un entorno y un agente, cuya comunicación se besa en los siguientes componentes:

- Estado: Este representa la situación o configuración en la que se encuentra el entorno en un instante t concreto.
- Acción: Esta señal indica que acción es tomada por el agente en función del estado en el que se encuentre.
- Recompensa: Señal automática que se recibe tras realizar la acción elegida y que nos indica cómo de buena es dicha acción en relación con el estado en que se encontraba el sistema.

La dinámica de estos algoritmos se basa en la siguiente estructura:

- El entorno envía al agente información del estado en el instante t en que se encuentran, de un conjunto de estados S posible.
- El agente selecciona una acción dentro de todo el conjunto de acciones A posibles.
- El entorno recibe la señal de la acción
- El entorno envía la señal de recompensa al agente

- Cambiamos del estado st al estado st+1 siempre y cuando st no sea terminal. Si así lo fuera, se considera que ha terminado un episodio (en el aprendizaje se pueden realizar miles de episodios).

Más allá del agente y el entorno, es posible identificar otros cuatro elementos más en los sistemas de aprendizaje por refuerzo: una política, una señal de recompensa, una función de valor, y, opcionalmente, un modelo del entorno.

2.4.3 Política

Una política (π) concreta cómo se va a comportar en agente durante el aprendizaje en un momento concreto. Se corresponde con lo que en psicología se conoce como un conjunto de agrupaciones estímulo-respuesta.
La política se utiliza para hacer un mapeo de un conjunto de estados recibidos del ambiente a un grupo de acciones A: $\pi: S \rightarrow A$.

Las políticas pueden ser desde una tabla de entradas hasta un proceso complejo de búsqueda. Es el núcleo de un sistema de aprendizaje por refuerzo el objetivo de estos sistemas es aprender una política óptima que maximice el valor esperado de la función de recompensa en todos los estados. La política determina cómo va a ser el comportamiento del sistema.

2.4.4 Señal de recompensa

Una señal de recompensa (Kevin Murphy, 2012) define el objetivo a alcanzar dentro del problema en cuestión. En cada instante de tiempo, el entorno envía al agente una única señal numérica llamada recompensa. El agente buscará en todo momento que esa recompensa que recibe sea la máxima posible, ya que esta es la que define como de buena es la acción elegida en cada instante. La señal de recompensa es la base primaria para alterar la política; si una acción elegida por la política supone una recompensa baja, entonces la política debe cambiar para seleccionar alguna de las acciones restantes en esa misma situación en el futuro. En

general, las señales de recompensa se podrían considerar como funciones estocásticas del estado del entorno y las acciones escogidas.

En cada estado se hace un mapeo a la recompensa r indicando cómo de deseables son los estados.

$$R : S \to \Re, \text{ o bien } R : S \times A \to \Re.$$

Ilustración 2.11. Mapeo de recompensa

Las recompensas recibidas en cada instante se acumulan en una secuencia que se conoce como secuencia de retorno. Este valor de retorno es el que obtiene el agente al estar en un estado St y a partir del mismo seguir una política concreta. Existen diferentes modelos de recompensas (Aviv, Dotan y Ron, 2012) que varían en función de las tareas a las que nos enfrentemos.

El primer tipo de modelo son las tareas episódicas u horizonte finito, cuya característica principal es tener un principio y fin determinado en forma de episodio, es decir, tienen un punto terminal. El retorno se define de la siguiente manera:

$$E\left[\sum_{t=0}^{h} r_t\right]$$

Ilustración 2.12. Retorno tareas episódicas

Donde ɣ es la tasa de descuento. Es un parámetro cuyo rango de valores está definido entre 0 y 1 para evaluar o darle un “peso” a cada recompensa que se reciba. Cuanto mayor sea el valor de esta tasa de descuento mayor será la influencia que tendrán las recompensas futuras y cuanto más pequeño sea, más se preocupará el agente por las recompensas inmediatas. Lo que se pretende es maximizar la recompensa total esperada.

$$\lim_{h\to\infty} E\left[\frac{1}{h}\sum_{t=0}^{h} r_t\right]$$

Ilustración 2.13. Retorno recompensa promedio

2.4.5 Modelo

Un modelo, como indican Sutton y Barto en su libro (2017), se utiliza para imitar el comportamiento del ambiente. Esto quiere decir que, por ejemplo, dado un estado y una acción concreta, el modelo será capaz de predecir el siguiente estado junto con la recompensa resultante. Así, entendemos que los modelos se usan para planear, es decir, considerar estado y acciones actuales, junto con posibles estados y acciones futuras antes de que sean experimentadas realmente para decidir un curso de acciones. Existen técnicas para la resolución de problemas que se basan en no tener modelo alguno, lo que se conoce como libre de modelo. Otra técnica consiste en definir previamente un modelo del entorno, que será utilizado con posterioridad para ayudar a elegir las mejores acciones. Hoy en día también existen algoritmos que pretenden utilizar ambos enfoques, dependiendo del conocimiento que haya acumulado el agente y el estado en el que este se encuentre.

2.4.6 Procesos de Decisión de Markov

En los problemas de aprendizaje por refuerzo el agente se basa en el estado del ambiente para realizar la toma de decisiones, por lo que el estado tiene un papel muy importante y la información que se introduce en el mismo es fundamental. Introducir mala información o información incorrecta supondrá un aprendizaje fallido (Martin L., 2012).

Es posible definir un proceso de decisión de Markov mediante la lista {S, A, f, R}.

- S → Conjunto de estados posibles dentro del sistema
- A → Conjunto de acciones posibles a realizar en cada uno de los estados
- f → Se trata del modelo, también conocido como función de transacción. En la mayoría de los casos f es desconocido y se asume que es estocástico.
- R → Es la función de recompensa

La metodología del MDP o Proceso de decisión de Markov es la siguiente: el agente en cuestión, en un estado st perteneciente al conjunto de estados S lleva a cabo una acción at del conjunto A. El realizar esa acción, el agente recibirá una recompensa con un valor esperado devuelto por la función R y a continuación se

pasará a un nuevo estado st+1 en función del modelo o función de transacción. Esta metodología se repetirá hasta que alcancemos un estado del conjunto S que sea terminal.

$$s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} s_3 \ldots$$

Ilustración 2.14. Proceso de decisión de Markov

El punto de partida de los MDP es que las acciones no dependen de acciones pasadas, sino del estado actual (propiedad de Markov), por lo que es necesario que dicho estado "resuma" de alguna forma lo que ha ocurrido en el pasado. No importa las acciones que te han llevado al estado en el que te encuentres en ese momento, pero dicho estado debe ser información suficiente para la toma de decisiones futura. Si los estados cumplen esta condición enviaran una señal que resume toda la información relevante conocida como señal Markoviana.

2.4.7 Función Q

En el aprendizaje por refuerzo está basado en la existencia de la tabla o función Q, que es el lugar en el que se almacena toda la información relacionada con los estados y las acciones realizadas en los mismos. Esta función recibe como parámetros de entrada el estado actual y la acción realizada en dicho estado, y calcula el valor recibido por el hecho de realizar dicha acción en el estado. El objetivo del aprendizaje por refuerzo es que, para cada estado, el algoritmo sea capaz de seleccionar la acción cuyo valor en la función Q sea el máximo o el más óptimo. La función Q se actualiza de la siguiente forma:

$$Q'(s,a)=Q(s,a)+\alpha[R(s,a)+\gamma \max Q'(s',a')-Q(s,a)]$$

Ecuación 2.10. Actualización de función Q

Donde:

- S es el estado actual
- a es la acción realizada en el estado actual
- s' es el estado siguiente tras realizar la acción
- a' es la acción que se tomará en el estado siguiente que da como resultado el valor máximo de Q en el estado s'

2.4.8 Función de valor

La familia de algoritmos de RL conocida como Algoritmos basados en la función de valor se basan en la aproximación de las funciones de valor estimando cómo de bueno es para el agente estar en un estado concreto o, también, la función de valor nos indica qué acciones van a ser buenas a largo plazo. También existe otra familia importante en RL llamada de gradiente de políticas que no se basan en optimizar la función Q. Nosotros desarrollaremos un algoritmo basado en esa primera familia, los basados en función de valor.

En pocas palabras, el valor de un estado St es el valor esperado de las recompensas que un agente puede esperar acumular en el futuro, a partir del estado en cuestión. Si al estar en un estado se recibe una recompensa baja, pero dicho estado está seguido de estados que garantizan recompensas buenas, este primer estado tendrá un valor alto a pesar de las malas recompensas. El objetivo de los aprendizajes es encontrar la función de valor óptima posible.

El valor de un estado s, siguiendo una política concreta π, se denota con V π(s) y se trata del refuerzo que esperamos obtener si nos dejamos llevar por esa política de selección de acciones π desde el estado actual s hasta el estado terminal o infinito.

$$V^{\pi}(s) = E_{\pi}\{\sum_{k=0}^{\infty} \gamma^{k} r_{t+k+1} | s_t = s\}$$

Ecuación 2.11. Ecuación de función de valor

En la que Eπ {} define el valor esperado dado que el agente sigue la política π. Esta función es conocida como función de valor-estado ("*state-value function*"). Además de esta, existe la función Q(s,a) llamada función de valor-acción ("*action-value function*"), que devuelve el valor de ejecutar una acción a desde un estado s siguiendo la política π. Su ecuación es:

$$Q^{\pi}(s,a) = E_{\pi}\{R_t | s_t = s, a_t = a\} = E_{\pi}\{\sum_{k=0}^{\infty} \gamma^{k} r_{t+k+1} | s_t = s, a_t = a\}$$

Ecuación 2.12. Función de valor-acción

Como bien puede observarse, en las dos ecuaciones está el parámetro γ que, como ya se ha mencionado previamente en este documento, se trata del factor de

descuento de las acciones futuras. Una política es considerada mejor o igual que otra siempre que la recompensa que se espera obtener es mayor que la segunda para cada uno de los estados. Uno de los principales objetivos es encontrar la política óptima, ya que siempre hay políticas que son mejores o iguales que el resto. Todas ellas tienen como similitud una función de valor-estado y otra de valor-acción, definidas de la siguiente forma:

$$V^*(s) = \max_\pi V^\pi(s), \forall s \in \mathcal{S}$$

Ecuación 2.13. Ecuación de función de valor-estado de política óptima

$$Q^*(s,a) = \max_\pi Q^\pi(s,a), \forall s \in \mathcal{S}, \forall a \in \mathcal{A}$$

Ecuación 2.14. Ecuación de función de valor-acción de política óptima

2.4.9 Métodos TD

El aprendizaje por diferencia temporal (Sutton y Barto, 2017) es una de las aportaciones más importantes de las técnicas de aprendizaje por refuerzo. Se trata de una combinación de los métodos de Monte Carlo y los métodos de programación dinámica. Esta combinación se basa en aprender directamente basándose en la experiencia, sin modelos previos (Monte Carlo) y realizan aproximaciones a partir de estimaciones que se aprendieron previamente (programación dinámica).

Dentro de los métodos basados en las diferencias temporales vamos a definir dos: Q_Learning y SARSA.

- ***Q-Learning***

Se trata de uno de los algoritmos más populares de aprendizaje por refuerzo. Como explican Sutton y Barto (), se trata de un algoritmo off-policy definido por la siguiente función de actualización de Q.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\Big]$$

Ecuación 2.15. Q-Learning

En este caso, la función de acción-valor aprendida, Q, se aproxima directamente a Q*, la función de acción-valor óptima, independientemente de la política de exploración que se esté siguiendo, en nuestro caso ε-Greedy, por eso es un método off-policy.

Dicha política tiene efecto sobre el algoritmo, ya que determina que par acción-estado es visitado y actualizado en cada momento.

inicializar $Q(s, a)$ arbitrariamente
para cada episodio de entrenamiento **hacer**
| inicializar s
| **repetir** para cada paso del episodio
| | escoger a desde s usando la política derivada de Q
| | realizar la acción a, observar r, s'
| | $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$
| | $s \leftarrow s'$
| **hasta** s es terminal
fin

Ilustración 2.15. Algoritmo Q-Learning

- ***SARSA***

  El otro método basado en diferencias temporales es el algoritmo SARSA (Simon, 1993). Este algoritmo se basa en buscar la política óptima a partir de la situación en la que se encuentre en un instante concreto por lo que es independiente de la política utilizada por el agente para la toma de decisiones, lo que lo convierte en un algoritmo on-policy.

  El primer paso de este algoritmo es estimar una función de acción-valor en lugar de una función estado-valor. En particular, para un método on-policy debemos estimar el valor de la función de acción-valor Q para la política actual y para el resto de los estados y acciones. Cabe recordar que un episodio consiste en una secuencia de estados y pares estado-acción como se muestra en la siguiente ilustración (Kevin P**.,** 2012):

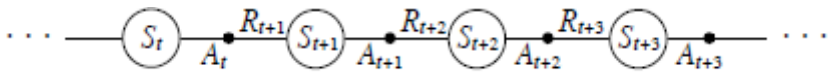

Ilustración 2.16. Iteración algoritmo SARSA

Matemáticamente, este algoritmo se representa de la siguiente manera:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\Big]$$

Ecuación 2.16. Función de actualización de Q del algoritmo SARSA

Esta actualización de la función se realiza para cada transición para un estado no terminal St. Si el estado St+1 es terminal, entonces Q (St+1, At+1) es definido como cero.

inicializar $Q(s, a)$ arbitrariamente
para cada episodio de entrenamiento **hacer**
| inicializar s
| escoger a desde s usando la política derivada de Q
| **repetir** para cada paso del episodio
| | realizar la acción a, observar r, s'
| | escoger a' desde s' usando la política derivada de Q
| | $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma Q(s', a') - Q(s, a)]$
| | $s \leftarrow s'$; $a \leftarrow a'$
| **hasta** s es terminal
fin

Ilustración 2.17. Algoritmo SARSA

El escoger una acción a' desde s' es lo que diferencia a este algoritmo del algoritmo Q-Learning.

2.5 Deep Q Learning (DQN)

Una vez explicados los conceptos de aprendizaje profundo y aprendizaje por refuerzo podemos introducir el concepto de *Deep Reinforcement Learning* (Maxim, 2018).

Deep Reinforcement Learning, en español conocido como Aprendizaje por Refuerzo Profundo es un término que hace referencia a la técnica que combina la teoría del aprendizaje por refuerzo con la utilización de redes neuronales. El uso de redes neuronales para modelar una política o una función Q abre la posibilidad de aplicar algoritmos de aprendizaje por refuerzo a tareas complejas, como por ejemplo los algoritmos que reciben imágenes como entrada. Los videojuegos son el ejemplo clásico donde los píxeles en bruto de la pantalla se proporcionan como representación del estado y, en consecuencia, como entrada de una política o una función Q.

$$Q^*_{t+1}(s,a) = \begin{cases} Q^*_t(s,a) + \alpha_t \left[r_{t+1} + \gamma \max_{a'} Q^*_t(s_{t+1}, a') - Q^*_t(s,a) \right] & \text{if } s = s_t, a = a_t \\ Q^*_t(s,a) & \text{else} \end{cases}$$

Ecuación 2.17. Función de actualización de Q del algoritmo Q-Learning

La principal idea residente en Deep Q-Learning es adaptar los algoritmos de diferencias temporales de tal forma que actualizar el valor de Q sea equivalente a un descenso de gradiente para entrenar una red neuronal y resolver cierta tarea de regresión. El funcionamiento residente en la función Q es el siguiente:

1. Inicializar los parámetros para Q(s,a), la red neuronal con pesos aleatorios y la memoria de almacenamiento de experiencias vacía.
2. Con probabilidad ty, seleccionar una acción aleatoria a, o, si así lo requiere la política de selección de acciones, tomar la acción óptima.
3. Ejecutar una acción *a* y observar la recompensa *r* y el siguiente estado *s'*
4. Almacenar la transición (s, a, r, s′) en la memoria de experiencias
5. Recoger un subconjunto aleatorio de un cierto tamaño de la memoria de experiencias
6. Calcular el valor de la función Q para cada elemento del subconjunto recogido en el paso anterior
7. Calcular función de coste
8. Actualizar Q (s, a) utilizando descenso de gradiente para minimizar el valor de la función de coste.
9. Repetir desde el paso 2 hasta la convergencia

Capítulo 3 Tecnologías presentes

En este capítulo se presenta un breve resumen de las tecnologías que han hecho posibles el desarrollo de este proyecto.

3.1 Python

El lenguaje de programación seleccionado para llevar a cabo el trabajo ha sido Python. Este es un lenguaje de programación de alto nivel, interpretado, dinámico y multiplataforma enfocado en desarrollar una sintaxis que priorice la legibilidad de su código.

Para el desarrollo de software de aprendizaje máquina Python es muy recomendable y, de hecho, es el más popular hoy en día desde su lanzamiento en 1991. Los motivos por los que Python es muy recomendado para este tipo de proyectos son varios, entre los que podemos destacar como el principal la legibilidad del código; es sencillo, consistente y elegante, además de disponer de una gran facilidad de implementación. Otro punto a favor de Python la velocidad de iteración de datos. Python es también de fuente abierta, lo que quiere decir que es gratuito, además dispone de una gran comunidad científica con acceso a su código fuente, de tal manera que se contribuye a que exista una continua mejora de este.

A modo de resumen hay que destacar que Python posee tanta popularidad por dos aspectos en concreto, el primero es la facilidad de aprendizaje de la que presume por la claridad de su sintaxis y segundo es su poder y capacidad de cómputo.

3.2 Tecnologías de aprendizaje máquina

Para desarrollar un proyecto de cualquier ámbito del aprendizaje máquina con Python necesitaremos crear un entorno virtual en el que se pueda desarrollar el código con comodidad. Para esto se utilizará el entorno de desarrollo Spyder en Anaconda.

Anaconda Permite programar entornos inalterables en lo que poder instalar las librerías necesarias. A continuación, vamos a explicar las librerías de aprendizaje máquina presentes en el proyecto.

3.2.1 Matplotlib

La librería Matplotlib está programada para generar imágenes y gráficas a partir de una serie de datos almacenados en *arrays* o listas de Python cuya extensión matemática es NumPy.

3.2.2 TensorFlow

TensorFlow es una herramienta creada por Google, cuyo motor está escrito en C y C++, utilizada para desarrollar y entrenar redes neuronales basadas en el descifrado de patrones y correlaciones. Su funcionamiento se basa en realizar cálculos numéricos mediante diagramas de flujo de datos. Lo que hace más fuerte a TensorFlow en relación a otras tecnologías es la gran variedad de herramientas de visualización del aprendizaje, además de tener bajos tiempos de compilación y soporte para sistemas multi GPU. Puede ser ejecutado en casi cualquier dispositivo tales como móviles, tablets o sistemas de cientos de computadoras.

Existen muchas APIs ya incluidas en TensorFlow que facilitan la comprensión del comportamiento de las redes neuronales implementadas. Una herramienta destacada es TensorBoard, que consiste en un conjunto de herramientas de visualización para ayudar al entendimiento, depuración y optimización de los gráficos generados por TensorFlow. Además, se pueden trazar diferentes métricas para mostrar información adicional. Otro aspecto importante de esta herramienta, es la visualización de estas métricas en tiempo real.

Hoy en día es una de las bibliotecas más populares para desarrollar algoritmos de Aprendizaje Profundo, debido a dos razones principales: la facilidad y rapidez con la que estos modelos pueden ser desarrollados usando Python; y la gran cantidad de documentación disponible.

3.2.3 Keras

Keras es una API de alto nivel de redes neuronales escrita en Python capaz de ejecutarse sobre TensorFlow. Fue desarrollada con el objetivo de permitir experimentaciones rápidas. Poder pasar de la hipótesis o idea al resultado final empleando el menor tiempo posible es crucial para el desarrollo de una buena experimentación. Las Principales propiedades de Keras son las siguientes:

- Facilidad de uso: Keras es un API diseñada para los seres humanos, no para las máquinas, por lo que se centra en reducir la carga cognitiva. Se trata de una API muy simple y consistente que minimiza el número de acciones que el usuario necesita para los casos de uso comunes y proporciona información clara y procesable de los errores que dichos usuarios puedan tener durante el desarrollo de un programa.
- Modularidad: Todos los modelos de aprendizaje profundo están compuestos por diferentes elementos tales como capas, funciones de activación, optimizadores, o funciones de coste. Keras proporciona todos estos elementos en forma de módulos, con el fin de utilizarlos cuando se requieran. Los usuarios pueden combinar estos elementos para crear modelos más complejos de manera sencilla.
- Derivado del modularidad se puede afirmar que es de fácil extensión.
- Como el núcleo de Keras está escrito en Python, todos los modelos son implementados en este mismo lenguaje, lo que resulta un proceso bastante simple.

La idea principal de este trabajo es utilizar la API Keras con TensorFlow de *backend* para implementar los modelos de aprendizaje profundo.

Capítulo 4 Trabajo realizado

En este epígrafe se va a explicar todo el trabajo llevado a cabo durante el periodo de desarrollo del proyecto. Primero presentaremos el juego seleccionado y cómo ponerlo en funcionamiento con el código; a continuación, se explicará la interpretación y la extracción de características del juego para poder implementar el algoritmo de aprendizaje, como por ejemplo la definición de estado y las recompensas; y para finalizar se explicará en detalle la implementación del algoritmo de aprendizaje por refuerzo profundo y los diferentes modelos de redes neuronales experimentados.

4.1 Juego: Super Mario Bros

Lanzado en sus inicios para la consola NES en 1985, Super Mario Bros es un juego de plataformas de desplazamiento lateral donde el jugador controla al personaje (Mario) y su objetivo es llegar al final de los niveles mientras el personaje sortea los obstáculos y evita los enemigos que aparecen a su paso. Los niveles son lineales y atravesados por Mario de izquierda a derecha. El objetivo principal de este proyecto es que el personaje del juego sea capaz de superar los nivele sin que un usuario externo interactúe con él.

Ilustración 4.1. Captura del primer nivel de Super Mario Bros

4.1.1 Uso

Gracias a el conjunto de herramientas de Python, OpenAI Gym, es posible ejecutar Super Mario en un entorno local, para ello hay que tener instalada con antelación en nuestra máquina la versión de Python 3.5 como mínimo y las librerías pip3 y OpenCV2. Para instalar el entorno sólo será necesario ejecutar el siguiente comando en la consola de python:

pip3 install gym gym-super-mario-bros

4.1.2 Python

Para ejecutar el juego en un programa de Python, primero de todo hay que importar el paquete *gym_super_mario_bros.* Tras esta importación será posible crear un entorno de Super Mario Bros. Por defecto, este entorno utiliza un conjunto discreto de 256 acciones. Para reducir esta gran cantidad de acciones, *gym_super_mario_bros* proporciona tres listas diferentes con un número bastante más reducido.

```
from nes_py.wrappers import JoypadSpace
import gym_super_mario_bros
from gym_super_mario_bros.actions import SIMPLE_MOVEMENT
env = gym_super_mario_bros.make('SuperMarioBros-v0')
env = JoypadSpace(env, SIMPLE_MOVEMENT)

done = True
for step in range(5000):
    if done:
        state = env.reset()
    state, reward, done, info = env.step(env.action_space.sample())
    env.render()

env.close()
```

Ilustración 4.2. Ejemplo de importación y ejecución del entorno Super Mario

Como podemos ver en la ilustración 4.2, el entorno creado llama al método *step*, que recibe una acción como parámetro y avanza el juego al siguiente paso (step). Este método devuelve 4 variables: *state, reward, done, info.*

4.1.3 Estado

El estado que se recibe tras ejecutar la función step es un array de dimensiones [256,240,3], que representa un frame concreto del juego en formato RGB. Si ejecutamos en la función env.render() podemos visualizar por pantalla el contenido del array estado. Esto será lo que utilizaremos para comprobar de forma visual que el agente ha aprendido una vez que el proceso de entrenamiento ha terminado. Si llamamos de manera continuada al método render durante el entrenamiento ralentizará el proceso demasiado, lo cual es insostenible.

4.1.4 Espacio de acciones

El repositorio desde el cual descargamos el juego permite al usuario especificar un conjunto personalizado de accione que el personaje puede realizar, o incluso podemos crearlo nosotros mismos. Cada conjunto de acciones posee un nivel de complejidad distinto. El hecho de elegir un espacio de acciones sencillo hará que el personaje aprenda más fácil y rápido, pero le impide realizar movimientos complejos que podrían ser necesarios para superar niveles complicados. Si Mario tiene dificultades en un nivel en concreto, es recomendable reducir la complejidad del espacio de acciones. Hasta la fecha, la lista de conjuntos de acciones disponibles en este juego son las siguientes:

Right only

Mario sólo puede moverse hacia la derecha. Este conjunto de acciones simplifica el proceso de entrenamiento, pero limita la posibilidad de llevar a cabo acciones más complejas. Los siguientes botones son los soportados

- Ninguna
- Right
- Derecha + A
- Derecha + B
- Derecha + A + B

Simple movement

Como añadido a la posibilidad de moverse a la derecha, correr y saltar, con este conjunto de acciones Mario puede desplazarse hacia la izquierda. Los botones soportados son:

- Ninguna
- Derecha
- Derecha + A
- Derecha + B
- Derecha + A + B
- A
- Izquierda

Complex movement

Este conjunto de acciones permite al personaje realizar todas las posibles acciones dentro del juego. Esta opción debería ser la escogida por defecto para realizar una exploración exhaustiva de los niveles, pero es muy probable que incremente la complejidad de aprender el nivel. Permite a Mario entrar en los tubos. Las combinaciones de botones soportadas son las siguientes:

- Ninguna
- Derecha
- Derecha + A
- Derecha + B
- Derecha + A + B
- A
- Izquierda
- Izquierda + A
- Izquierda + B
- Izquierda + A + B
- Abajo
- Arriba

4.1.5 Wrappers

Mediante **Wrappers**, importado desde el emulador **nes-py,** podemos definir el comportamiento del entorno, ya que de esta forma seleccionamos el conjunto de acciones disponibles dentro del mismo. La forma de utilizarlo en nuestro código python es la siguiente:

from nes_py.wrappers import JoypadSpace

env = JoypadSpace(Lista de acciones)

La lista de acciones que se pueden utilizar se pasa como parámetro al llamar a la función JoypadSpace y se representan mediante un número entero entre 0 y el número de acciones -1.

4.1.6 Función de recompensa

La función de recompensa se fundamenta en que el objetivo del juego es moverse hacia la derecha para alcanzar el final del nivel (incrementar la posición en el eje X del personaje), tan rápido como sea posible, sin morir. Para diseñar la función de recompensas se requieren 3 variables, que son las siguientes:

1. ***v*:** La diferencia de la posición en x del personaje en dos estados consecutivos. Puede entenderse como la velocidad en cada instante.
    - Se puede considerar como la velocidad en un instante concreto
    - $v = x1 - x0$
        - *x0* es la posición en x antes de realizar una acción
        - *x1* es la posición en x tras realizar una acción
    - moverse hacia la derecha implica $\Leftrightarrow v > 0$
    - moverse hacia la izquierda implica $\Leftrightarrow v < 0$
    - sin cambios en la posición en x implica $\Leftrightarrow v = 0$
2. ***c***: es la diferencia del reloj del juego entre fotogramas
    - La penalización evita que el personaje se quede en el mismo sitio
    - $c = c0 - c1$
        - *c0* es el reloj leído antes de la acción
        - *c1* es el reloj leído después de la acción
    - no cambios en el reloj $\Leftrightarrow c = 0$
    - cambio en el reloj $\Leftrightarrow c < 0$
3. ***d*:** es la penalización que recibe el agente por morir
    - esta penalización sirve para que el agente evite las acciones que le hagan morir
    - vivo $\Leftrightarrow d = 0$
    - muerto $\Leftrightarrow d = -15$

$$r = v + c + d$$

Tras esta explicación se puede asegurar que las recompensas, entonces, se encuentran definidas en el intervalo *[-15, 15]*.

4.1.7 Diccionario

El entorno Super Mario Bros dispone de una función llamada step, la cual recibe como parámetro un número entero que representa una de las acciones de la lista que hayamos definido mediante **Wrappers.** Esta función hace que el entorno avance al siguiente paso (step), devolviendo un diccionario con las siguientes claves:

- coins: El número de monedas recogidas(int)
- flag_get: True si mario ha recogido o no la bandera (boolean)
- life: El número de vidas (int)
- score: puntuación acumulativa otenida
- stage: El escenario en el que se encuentra
- status: El estado de mario ({*'small', 'tall', 'fireball'*})
- time: Tiempo transcurrido
- world: Mundo actual (int de 1 a 8)
- x_pos: Posición en X en el escenario
- x_pos_screen: Posición en x en la pantalla
- y_pos: Posición en y en el escenario y pantalla

4.2 Desarrollo

En este episodio se van a tratar los puntos principales que marcarán el desarrollo del proyecto. Estos se dividen en tres: análisis, diseño e implementación. A continuación, vamos a desarrollar cada uno más en detalle.

4.2.1 Análisis

En este primer punto del capítulo vamos a tratar el comportamiento estático y dinámico del sistema. A continuación, en los siguientes subapartados se desarrollarán los casos de uso, el diagrama de secuencia y de actividad.

4.2.1.1 Casos de uso

En este apartado vamos a describir en detalle los casos de uso de nuestro proyecto. Antes de nada, dentro de los casos de uso existen actores que son los usuarios que interactúan con el sistema. Los casos de uso pueden definirse como:

"*Un caso de uso especifica un conjunto de secuencias de acciones, incluyendo variantes, que el sistema puede ejecutar y que produce un resultado observable de valor para un actor en particular.*"

Se trata de una descripción de un grupo de secuencias de acciones donde cada secuencia define un posible comportamiento del sistema, añadiendo valor a los usuarios o actores de este. En este sistema sólo existe un usuario o actor. En la ilustración 4.3 se pueden observar los casos de uso que nuestro sistema posee para cumplir los requisitos estipulados en el capítulo.

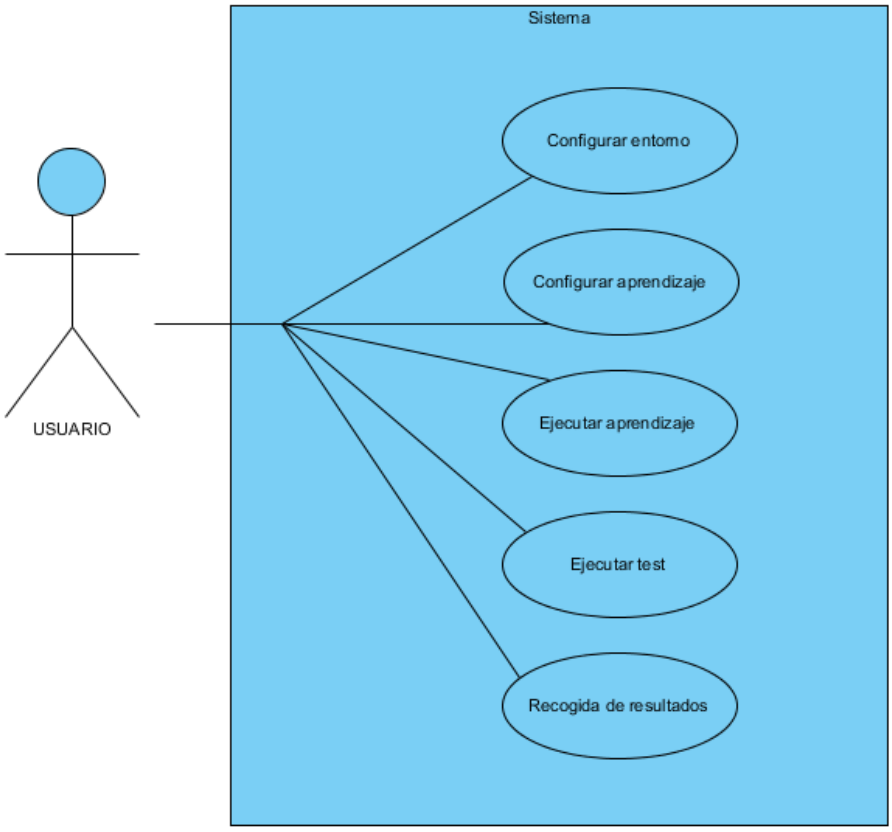

Ilustración 4.3. Casos de uso

Una vez identificados los casos de uso en la ilustración anterior, vamos a comentarlos de forma detallada a continuación.

Configurar entorno	
Actor	Usuario
Descripción	El usuario se encarga de elegir el nivel que va a representar el entorno
Precondición	Tener importado los paquetes super_mario_gym y nes_py
Postcondición	Juego iniciado sin fallos
Flujo normal	Actor: 1. El usuario importa los paquetes necesarios 2. El usuario introduce el nivel
Flujo alternativo	1. El usuario introduce un nivel que no existe

Tabla 4.1. Caso de uso: Configurar entorno

Configurar aprendizaje	
Actor	Usuario
Descripción	Configurar el modelo y los hiper parámetros para realizar el aprendizaje
Precondición	Tener importada la librería keras
Postcondición	Modelo e hiper parámetros bien definidos
Flujo normal	Actor: 1. El usuario define el modelo y los hiper parámetros del algoritmo
Flujo alternativo	Ninguno

Tabla 4.2. Caso de uso: Configurar aprendizaje

Ejecutar aprendizaje	
Actor	Usuario
Descripción	Se ejecuta el periodo de aprendizaje
Precondición	Modelo e hiper parámetros bien definidos
Postcondición	Modelo e hiper parámetros bien definidos
Flujo normal	Actor: 1. Define el número de partidas que se van a jugar Sistema: 1. Ejecuta el aprendizaje
Flujo alternativo	Actor: 1. Introduce un número de partidas menor o igual que cero Sistema: 1. Error de ejecución, el programa se cierra

Tabla 4.3. Caso de uso: Ejecutar aprendizaje

Ejecutar test	
Actor	Usuario
Descripción	El sistema pone en práctica lo aprendido jugando partidas prediciendo las mejores acciones
Precondición	Aprendizaje realizado
Postcondición	Ninguna
Flujo normal	Actor: 1. El sistema pregunta al usuario si quiere realizar un test para observar lo aprendido Sistema: 1. Usuario introduce por teclado en consola que quiere realizar test
Flujo alternativo	Actor: 1. Usuario decide no realizar el test

Tabla 4.4. Caso de uso: Ejecutar aprendizaje

4.2.2 Diseño

Una vez que se han definido y explicados los casos de uso, vamos a proceder a realizar el diseño de nuestro proyecto paso a paso. Se va a presentar un diagrama de clases para poder tener una visión más clara y general de las clases implementadas y la estructura global.

4.2.2.1 Diagrama de clases

En la ilustración 4.4, que aparece a continuación, podemos encontrar el diagrama de clases que define este proyecto, donde se presentan los elementos más importantes del sistema y la relación existente entre cada uno de ellos. Las clases presentes en este proyecto son las siguientes:

- Estado: Se trata de la clase del sistema que está relacionada con todo lo que tiene que ver con la información de la visual. En esta clase se ejecuta el juego en cuestión y se trata la información de manera adecuada para poder poner en marcha el aprendizaje.

- Modelo: Clase principal en el sistema en la que se crea el modelo de la red neuronal que vamos a entrenar. Esta clase es el cerebro del sistema, por lo que es la más delicada y compleja de implementar. el foco principal del trabajo es encontrar el mejor modelo posible que nos permita lograr un algoritmo óptimo.
- Q-Learning: Clase que contiene toda la información relacionada con el aprendizaje por refuerzo. En ella se encuentran los hiper parámetros necesarios de ajustar para desarrollar el proceso de aprendizaje.

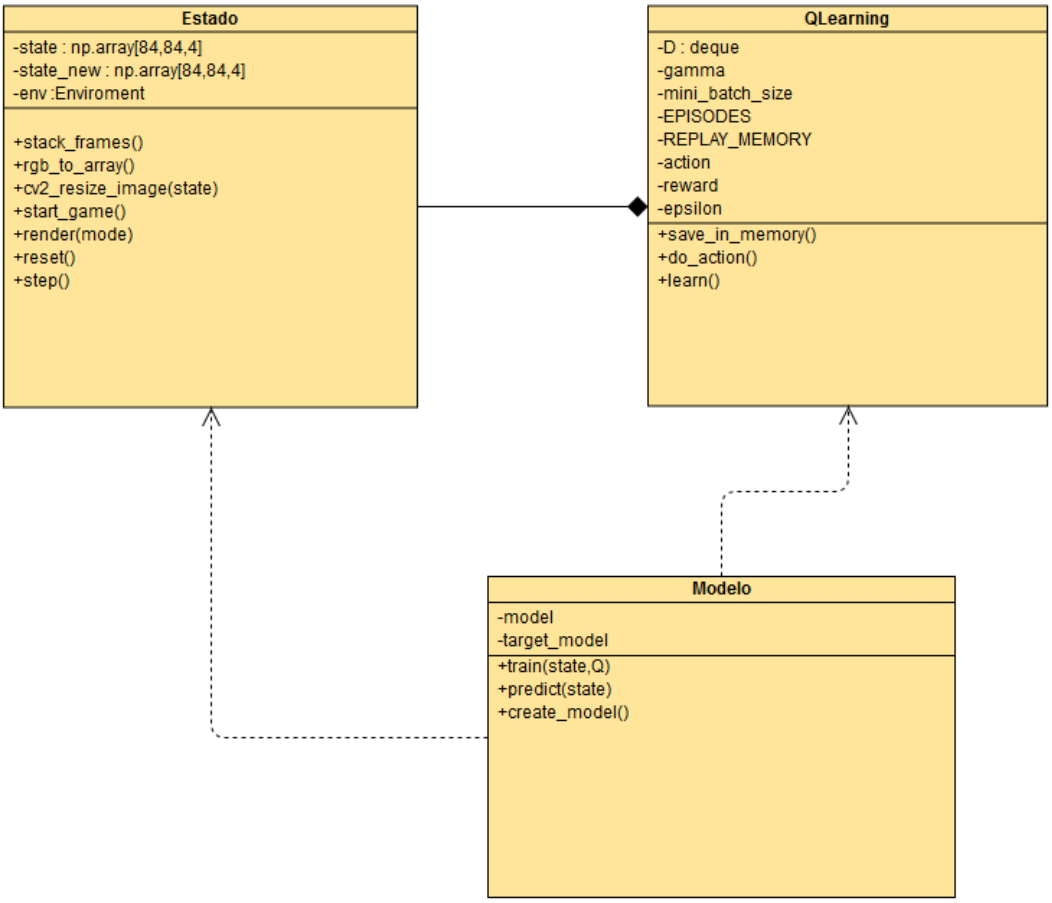

Ilustración 4.4. Diagrama de clases

4.2.3 Implementación

Una vez que se hemos elegido el juego sobre el que vamos a trabajar, lo hemos estudiado, entendido y sabemos cómo obtener todas las características del mismo, podemos comenzar con la implementación de nuestro algoritmo. En este epígrafe se va a explicar cómo está codificado el sistema a partir de los aspectos tratados en el diseño y análisis de los apartados previos. A continuación, vamos a detallar varios de los aspectos más destacados del código para comprender el funcionamiento del sistema.

4.2.3.1 Estado

Para comenzar, vamos a explicar toda la implementación relacionada con el estado con el mayor detalle posible. Esta clase es la encargada de representar el juego y recoger la información del mismo en forma de imágenes que serán tratadas, ya sea para mostrarlas por pantalla y visualizar el juego o para convertirlas en el estado que recibe la red neuronal que estamos entrenando.

4.2.3.2 Entorno

El punto de partida del desarrollo del proyecto es seleccionar el entorno de un videojuego en concreto. Como ya se ha explicado en epígrafes anteriores, nuestro entorno seleccionado es el videojuego **Super Mario Bros.** El entorno lo representamos con la variable **env**, la cual se inicializa llamando a la función **make** del paquete importado **gym_super_mario** como podemos ver a continuación.

```
env = gym_super_mario_bros.make('SuperMarioBros-1-1-v0')
env = JoypadSpace(env, RIGHT_ONLY)
```

Hay que destacar como añadido la llamada a la función JoypadSpace, donde definimos el conjunto de acciones disponibles dentro de nuestro entorno.

De la variable **env** podemos utilizar varias funciones que nos serán de utilidad, las cuales, son las siguientes:

- **env.reset():** Función encargada de reiniciar el juego para que el nivel comience desde cero.
- **env.step(action):** Función que recibe como parámetro un número entero entre 0 y el número de acciones -1 y avanza hasta el estado siguiente. La salida de

esta función es un conjunto de 4 variables '*observation, reward, done, info'*. Resepectivamente, cada unoa de estas variables se corresponden con el siguiente estado, la recompensa recibida tras ejecutar la acción, información de si el juego ha terminado o no y un diccionario. Un ejemplo de diccionario podría ser el siguiente:

{'stage': 1, 'world': 1, 'x_pos_screen': 89, 'flag_get': False, 'time': 326, 'status': 'small', 'life': 2, 'coins': 0, 'y_pos': 93, 'x_pos': 722, 'score': 100}

- **env.render(tipo de render):** Función encargada de la representación visual del videojuego. Como parámetro recibe el tipo de render, que puede ser '*human'* (mostraría por pantalla el estado del juego en este instante)

Ilustración 4.5. Imagen de Super Mario Bros

Otro modo de render es '*rgb_array*', que devuelve un array de 3 dimensiones [256,240,3], que representa la imagen del videojuego de forma numérica.

4.2.3.3 Representación del estado

Cómo se representa el estado es una de las partes más importantes y delicadas de seleccionar a la hora de implementar el algoritmo. El estado es la entrada de nuestra red neuronal, por lo que una mala elección del mismo hará que la red no aprenda correctamente o, incluso, no aprenda nada. Se han implementado varias formas de representar el estado, unas con mejores resultados que otras, que son explicadas a continuación

- **1ª interpretación**

En una primera instancia del problema, se ha desarrollado una implementación del estado muy sencilla, la cual consiste en una matriz de 2 entradas en la que se almacena la posición en el eje X del personaje dentro del nivel, la posición en el eje Y,

el tiempo que lleva transcurrido en el instante en el que se recoja la información y la posición de la pantalla. Todas estas variables son extraídas del diccionario *info* mencionado antes. Esta se trata de una representación muy escueta, que, si bien es un factor experimental, no es suficiente porque es carente de información. No sabemos cuál es la posición de los enemigos o de los obstáculos, por ejemplo.

- **2ª interpretación**

La manera más conveniente, aunque también la más costosa, de representar un estado es con su imagen. De la imagen podemos extraer toda la información que necesitamos ya que no requiere de variables de posición o de tiempo. Para obtener la imagen del juego sólo hace falta llamar al método *render* en modo *rgb_array.* Este método nos devuelve un array de dimensiones [256,240,3], que representa la imagen de un frame del juego en escala RGB. Esta es la mejor representación posible, pero también la más costosa, por lo que conviene pasarla por una serie de filtros para reducir su tamaño y, por consiguiente, acelerar la velocidad de cómputo.

A la hora de implementar nuestra red neuronal, para el caso que estamos planteando, no es necesario que la imagen sea a color, por lo que podemos convertirla de escala RGB a escala de grises y no perderíamos información relevante. Convertir una imagen RGB en una imagen en escala de grises es muy sencillo. El valor de cada píxel en una escala de grises representa la intensidad de la luz, que puede ser calculada con la siguiente fórmula:

$$Y = 0.2162*R+0.7152*G+0.0722*B$$

Aquí, R, G, B son los canales rojo, verde y azul de la imagen, respectivamente. Para aplicar esta fórmula utilizamos la siguiente función en nuestro código:

```
def rgb_to_gray(im):
    return np.dot(im, [0.2126, 0.7152, 0.0722])
```

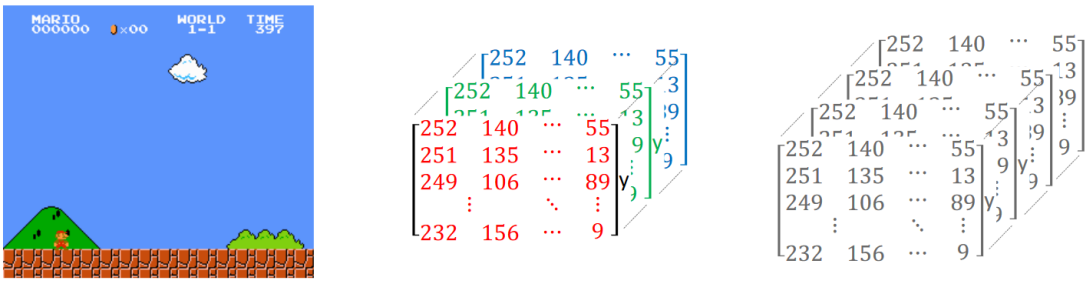

Ilustración 4.6. Ejemplo reducción RGB a gris

Convertir la imagen a escala de grises es una buena reducción, pero no es suficiente. En este juego, como en la mayoría de los juegos de Atari, la parte de arriba de la pantalla muestra los puntos obtenidos, el tiempo transcurrido y demás información irrelevante que solo genera ruido. Como esta información no es útil para superar el nivel, puede ser eliminada sin problemas. Para ello se reduce el tamaño de la imagen a [84,84], en la que la parte de arriba ha sido recortada. Para aplicar este filtro disponemos de la función **cv2_resize_images**, que recibe como parámetros la imagen, las dimensiones de la imagen filtrada, el método que se aplicará para recortarla (*crop* en este caso) y *offset* (cuánto va a ser recortado).

```
def cv2_resize_image(image, resized_shape=(84, 84), method='crop', crop_offset=8):
    height, width = image.shape
    resized_height, resized_width = resized_shape
    if method == 'crop':
        resized = cv2.resize(image, (resized_width, resized_height), interpolation = cv2.INTER_LINEAR)
        resized_crop = resized[15:resized_height, :]
        resized2 = cv2.resize(resized_crop, (resized_width, resized_height), interpolation = cv2.INTER_LINEAR)
        return np.asarray(resized2)
    elif method == 'scale':
        return np.asarray(cv2.resize(image,(resized_width,resized_height),interpolation=cv2.INTER_LINEAR),dtype = np.uint8)
    else:
        raise ValueError("Unrecognized image resize method")
```

El resultado tras aplicar todos los filtros en una imagen es el siguiente:

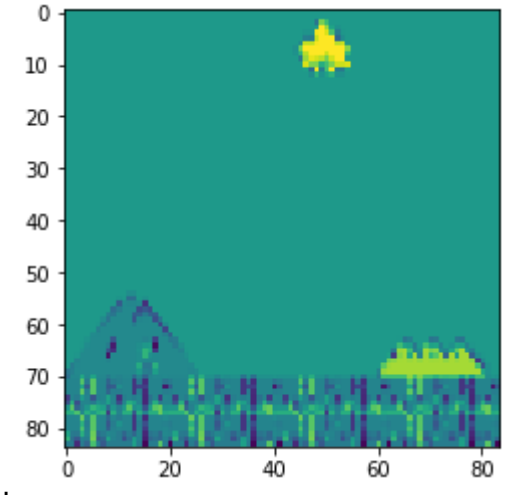

Ilustración 4.7. Imagen con filtros aplicados

Utilizar una imagen como entrada de una red neuronal convolucional es posible que no sea buena información. En lugar de usar sólo la pantalla del juego actual como entrada del modelo, es común usar múltiples frames apilados como entrada de la CNN. Al hacerlo, el modelo puede procesar cambios y "movimientos" en la pantalla entre frames consecutivos, lo que no sería posible cuando se utiliza un solo frame del juego.

4.2.3.4 Modelo

En este apartado todo el proceso seguido para el desarrollo de los modelos de aprendizaje por refuerzo profundo es explicado en detalle. El paso inicial es definir la arquitectura del modelo, entendiendo por arquitectura el número de capas, el número de nodos por capa y cómo se interconectan entre cada una de ellas.

Es importante tener en cuenta que no hay una fórmula concreta a seguir para construir una red neuronal que garantice un buen comportamiento. Cada problema requiere una red neuronal diferente. Como se ha mencionado en capítulos anteriores, la librería **Keras** sobre **TensorFlow** ha sido la escogida para implementar todos los modelos.

Primera arquitectura

La primera arquitectura implementada se basa en un modelo muy simplista en lo que a redes neuronales se refiere. La principal idea detrás de esta arquitectura es plantear un experimento en el que la información recogida del entorno sea la menos posible para acelerar el proceso de entrenamiento. Con experiencias en trabajos anteriores, tenía la certeza que trabajar con imágenes es la mejor opción, pero también es mucho más costosa. Esta arquitectura representa una red neuronal simple que recibe como parámetros de entrada a las posiciones del agente en los ejes de coordenadas, el tiempo transcurrido y la posición de desplazamiento de la pantalla. Se pretende que con esta poca información el agente sepa, dependiendo del tiempo y las posiciones, donde hay un objeto obstáculo para sortearlo. Del entorno no podemos obtener información de la posición de los enemigos ni de los obstáculos, por lo que deja la entrada de la red neuronal con una información muy escueta.

La arquitectura de la red neuronal secuencial es la siguiente:

- Entrada: Array de 4 elementos numéricos;
- Primera capa: Dense con 10 unidades rectificadoras y función de activación ReLU.
- Segunda capa: Dense con 10 unidades rectificadoras y función de activación ReLU.
- Tercera capa: Dense con 5 unidades rectificadoras y función de activación ReLU.
- Cuarta capa: Dense con 5 unidades rectificadoras y función de activación ReLU.
- Output: Valores para las 6 acciones posibles.

A modo de resumen y para favorecer su comprensión, tenemos el siguiente grafo que representa la red neuronal implementada.

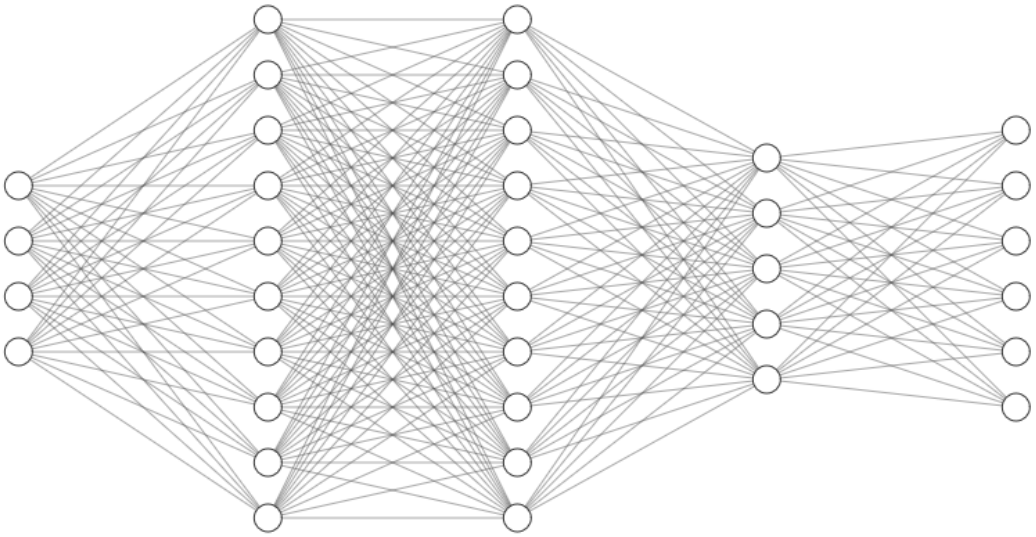

Ilustración 4.8. Representación gráfica de primera red neuronal

Mediante código, la forma de implementar este modelo es la siguiente:

```
self.model= Sequential()
self.model.add(Dense(10, input_shape=(4,1), init='uniform', activation='relu'))
self.model.add(Flatten())
self.model.add(Dense(10, activation='relu'))
self.model.add(Dense(5, activation='relu'))
self.model.add(Dense(env.action_space.n, activation='softmax'))    # Tantas salidas como accciones
self.model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

Segunda arquitectura

Para a segunda se produce un cambio bastante sustancial, dejamos de utilizar posiciones y tiempos como entradas de la red neuronal y nos centramos en un problema de reconocimiento de imágenes, lo que hace que pasemos a utilizar una red neuronal convolucional.

- Entrada: 4 frames de dimensiones 84×84 en escala de grises;
- Primera capa (convolucional): 64 8×8 kernel con *stride* de 4 píxeles y la función de activación ReLU.
- Segunda capa (convolucional): 64 4×4 kernel con un *stride* de 2 píxeles, y la función de activación ReLU.
- Tercera capa (completamente conectadas): 512 unidades rectificadoras;
- Output: Valores parar las 6 posibles acciones.

El esquema que resume el modelo seleccionado es el siguiente:

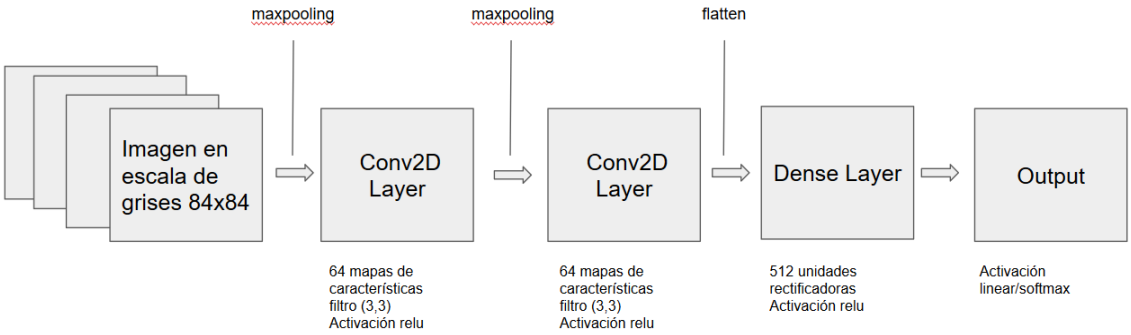

Ilustración 4.9. Representación gráfica de segunda red neuronal

La manera de implementar en el código del proyecto es la siguiente:

```
model = Sequential()
model.add(Conv2D(32, kernel_size=(8,8),
                 activation='relu', input_shape=(84,84,4)))
model.add(MaxPool2D())
model.add(Conv2D(64, kernel_size=(4,4), activation='relu'))
model.add(MaxPool2D())
model.add(Conv2D(64, kernel_size=(3,3), activation='relu'))
model.add(Flatten())
model.add(Dense(512, activation='relu'))
model.add(Dense(env.action_space.n, init='uniform',activation='linear'))
adam = Adam(learning_rate=0.00001)
model.compile(loss='mse', optimizer=adam, metrics=['accuracy'])
```

Los frames utilizados son procesados como se ha explicado antes, después son convertidos a escala de grises y se reduce su dimensionalidad a 84x84. Una vez que pasan este procesado, se apilan en el método **stack_frames** presentado a continuación.

```
def stack_frames(stacked_frames, state,frame, new_episode, stack_size=4):
    if new_episode == True:
        stacked_frames = deque([np.zeros((84,84), dtype=np.int) for i in range(stack_size)], maxlen=4)
        for i in range(4):
            stacked_frames.append(frame)
        stacked_state = np.stack(stacked_frames, axis=2)
    else:
        stacked_frames.append(frame)
        stacked_state = np.stack(stacked_frames, axis=2)
    return stacked_state, stacked_frames
```

Tercera arquitectura

El modelo de una red neuronal profunda tiende a sobreestimar los valores Q de las acciones potenciales en un estado dado. No causaría ningún problema si todas las acciones fueran sobreestimadas de la misma forma, pero el caso es que una vez que una acción específica se sobreestima, es más probable que sea elegida en la siguiente iteración, lo que hace muy difícil para el agente explorar el entorno de manera uniforme y encontrar la política correcta. Esto se debe a la fórmula para obtener el valor del *target*:

$$y_i := \mathbb{E}_{a' \sim \pi}\left[r + \gamma \max_{a'} Q\left(s', a'; \theta_{i-1}\right) \right]$$

Ecuación 4.1. Target

En esta ecuación, podemos ver se toma el valor máximo conocido en ese instante de tiempo. Esto puede hacer que nuestra red priorice dicho valor en situaciones que sean muy altos pero irreales. Para solventar este problema existe un truco bastante simple. Vamos a disociar la elección de la acción de la generación del valor Q. Para ello vamos a tener dos redes separadas. Usaremos nuestra red primaria para seleccionar una acción y una red objetivo (*target*) para generar un valor Q para esa acción. Con el fin de sincronizar nuestras redes, vamos a copiar pesos de la red primaria a la red objetivo cada *n* pasos de entrenamiento.

La arquitectura utilizada para ambas redes neuronales profundas es la siguiente:

- Entrada: 4 frames de dimensiones 84×84 en escala de grises;
- Primera capa (convolucional): 64 8×8 kernel con *stride* de 4 píxeles y la función de activación ReLU.
- Segunda capa (convolucional): 64 4×4 kernel con un *stride* de 2 píxeles, y la función de activación ReLU.
- Tercera capa (completamente conectadas): 512 unidades rectificadoras;
- Output: Valores parar las 6 posibles acciones.

Como podemos ver en la siguiente imagen, los dos modelos (inicial y target) son creados la misma forma:

```
model = Sequential()
model.add(Conv2D(32, kernel_size=(8,8),
                 activation='relu', input_shape=(84,84,4)))
model.add(MaxPool2D())
model.add(Conv2D(64, kernel_size=(4,4), activation='relu'))
model.add(MaxPool2D())
model.add(Conv2D(64, kernel_size=(3,3), activation='relu'))
model.add(Flatten())
model.add(Dense(512, activation='relu'))
model.add(Dense(env.action_space.n, init='uniform',activation='linear'))
adam = Adam(learning_rate=0.00001)
model.compile(loss='mse', optimizer=adam, metrics=['accuracy'])


target_model = Sequential()
target_model.add(Conv2D(32, kernel_size=(8,8),
                 activation='relu', input_shape=(84,84,4)))
target_model.add(MaxPool2D())
target_model.add(Conv2D(64, kernel_size=(4,4), activation='relu'))
target_model.add(MaxPool2D())
target_model.add(Conv2D(64, kernel_size=(3,3), activation='relu'))
target_model.add(Flatten())
target_model.add(Dense(512, activation='relu'))
target_model.add(Dense(env.action_space.n, init='uniform',activation='linear'))
adam = Adam(learning_rate=0.00001)
target_model.compile(loss='mse', optimizer=adam, metrics=['accuracy'])
```

4.2.4 Q-Learning

En este apartado vamos a explicar los detalles de la clase Q-Learning. Esta clase es la parte del código encargada de gestionar en aprendizaje, es decir, es donde se realiza la toma de decisiones (elegir acciones), se recogen las recompensas y se actualiza la función de valor en función de las recompensas recibidas.

En esta clase tenemos las siguientes variables, que son elegidas por el usuario antes de ejecutar el programa:

- **EPISODES**: cuantos episodios hay que ejecutar
- **gamma**: el factor de descuento que se aplica a las recompensas
- **epsilon:** el parámetro de exploración, disminuirá con el tiempo.
- **action:** número entero mayor o igual que 0 que representa la acción a realizar
- **reward:** número entero entre -15 y 15 que representa la recompensa recibida tras realizar una acción.
- **total_reward:** variable incremental en la que se van sumando las recompensas recibidas durante un episodio
- **D**: lista para guardar las experiencias
- **memory_size:** tamaño de la memoria
- **mini_batch_size:** cuántas experiencias se cogen de la memoria D cada vez que actualizamos la red neuronal

4.2.4.1 Selección de acciones

En cada iteración de nuestro algoritmo, el agente tiene que tomar la decisión de qué acción debe realizar. El proceso de toma de decisiones consiste en realizar una exploración aleatoria del entorno, que poco a poco deja de ser aleatoria. Nosotros utilizamos la política llamada epsilon greedy, que se basa en la probabilidad de elegir una acción aleatoria, o la mejor acción posible en base a los valores de del modelo que se está entrenando, siguiendo una función de probabilidad exponencial negativa. A medida que el número de partidas va aumentando, el agente tiene más información de cómo avanzar en el nivel y por tanto debe ser cada vez más probable hacer la mejor acción posible y que el algoritmo converja. Para ello utilizamos una función de una forma similar a la siguiente:

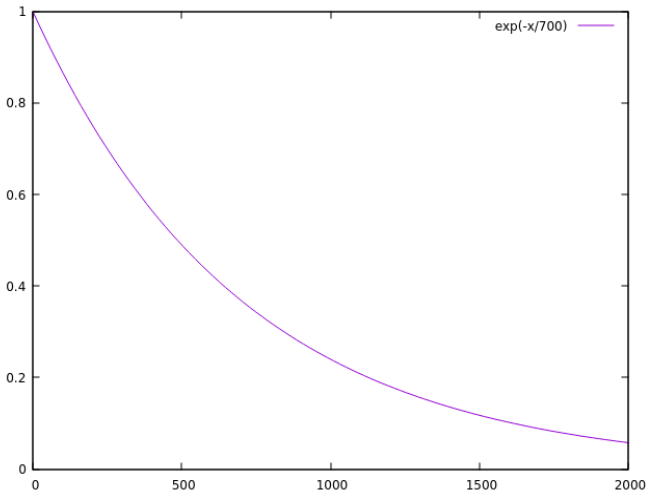

Ilustración 4.10. Gráfica E-Greedy

En el periodo de entrenamiento, aunque la posibilidad de realizar una exploración aleatoria disminuya, siempre tiene que haber una pequeña probabilidad para dicha exploración. Para ello se define un valor mínimo, el cual nuestro parámetro *épsilon* puede tomar.

Intuitivamente, es útil tener un valor muy alto (máximo 1) durante las etapas iniciales del proceso de entrenamiento para que el agente

```
epsilon = np.exp(-episode/700)
if epsilon < 0.1:
    epsilon = 0.1

if np.random.rand() <= epsilon:
    action = np.random.randint(0, env.action_space.n, size=1)[0] #Seleccion de accion aleatoria
else:
    Q = target_model.predict(np.expand_dims(state, axis=0))         # Predicciones de red neuronal
    action = np.argmax(Q)
```

Épsilon es un hiper parámetro que puede ser ajustado a un buen valor en base a los experimentos realizados. Un valor más alto significa que las acciones del agente serán aleatorias y un valor más bajo significa que el agente utilizará lo que ya sabe sobre el entorno y no intentará explorar (acción aleatoria).

4.2.4.2 Memoria

Es muy posible que requiera demasiado tiempo que un agente mejore y sea capaz de resolver el problema planteado. En este apartado vamos a presentar el concepto de memoria, que se utiliza para mejorar el rendimiento del agente.

En la mayoría de los entornos, la información recibida por el agente no es independiente ni está distribuida de forma idéntica. Lo que esto significa es que la información que el agente recibe está fuertemente correlacionada con la información anterior que había recibido y la siguiente que recibirá. Esto es comprensible ya que, normalmente, los problemas que el agente resuelve en entornos de aprendizaje de refuerzo típicos son secuenciales. Se ha demostrado que las redes neuronales convergen mejor si las muestras recibidas son independientes y distribuidas de forma idéntica.

El uso de la memoria para almacenar la experiencia permite, entonces, reutilizar la experiencia pasada del agente. Las actualizaciones de la red neuronal, especialmente con menores tasas de aprendizaje, requieren varios pasos de retro propagación y optimización para converger en buenos valores. La reutilización de los datos de la experiencia pasada, especialmente en mini lotes para actualizar la red neuronal, ayuda de una forma muy notable a la convergencia de la red.

Implementación de la memoria de experiencias

Una vez introducido el concepto de memoria de experiencias, vamos a explicar en qué se basa su implementación. Para ponernos en contexto, cabe remarcar el concepto de experiencia. En aprendizaje por refuerzo, donde los problemas se representan utilizando Procesos de Decisión de Markov (MDP), ya explicado en el estado del arte, resulta eficiente representar una experiencia reciba como una estructura de datos que almacena la información observada en un instante de tiempo *t,* la acción realizada a partir de la información observada, la recompensa recibida por realizar esa acción y la siguiente observación (o estado) hacia la cual el entorno se dirige en función de la acción. También es conveniente incluir el valor Booleano

“done”, que representa si la siguiente observación que ha sido recibida es un estado terminal o no.

Para almacenar esta información, en Python, utilizaremos un objeto de tipo **deque ().** Es una cola iterable que permite almacenar estructuras de datos mediante la función **append().**

```
D = deque(maxlen=REPLAY_MEMORY)  # Memoria para almacenar la informacion en cada instante
```

Cuando la memoria está llena, si se intenta introducir nuevos valores, de manera automática se desencola el valor más antiguo. De esta forma no se pierde información relevante y mantenemos la memoria actualizada en base a las últimas experiencias recibidas.

```
D.append((state, action_aux, reward_aux, state_new , done)) # 'Memoria' accion y efecto
```

Ahora podemos pasar a implementar el tratamiento de la memoria. Para averiguar qué métodos necesitamos implementar en la clase de memoria de experiencias, hay qué pensar cómo se va a utilizar dicha memoria.

Primero, queremos ser capaces de almacenar nuevas experiencias que el agente recoge. Luego, queremos tomar muestras o recuperar experiencias en lotes de la memoria cuando queramos actualizar la Q-función.

Como ya hemos mencionado, esta clase dispone del método **append()** para almacenar información, así que, lo que necesitaremos es un método que pueda recoger un pequeño lote de experiencias para ser tratadas. En nuestro proyecto tenemos el siguiente código, que cumple los requisitos que se acaban de mencionar:

```
if(len(D) > mb_size):
    minibatch = random.sample(D, mb_size)
    for mb in minibatch:
        state = mb[0]
        action = mb[1]
        reward = mb[2]
        state_new = mb[3]
        done = mb[4]
        Q = model.predict(np.expand_dims(state, axis=0))
        Q_Next = target_model.predict(np.expand_dims(state_new, axis=0))

        target_aux = np.zeros((1,env.action_space.n),float)
        if done:
            Q[0,action]= reward
        else:
            Q[0,action] = reward + gamma * np.max(Q_Next)

        model.fit(np.expand_dims(state, axis=0), Q, epochs=1, verbose=0)
    target_model.set_weights(model.get_weights())
```

Capítulo 5 Pruebas y resultados

En este episodio del proyecto vamos a explicar las pruebas que se han realizado con el sistema implementado junto con los resultados obtenidos. Esto consiste en realizar diferentes experimentos en los que se utilicen modelos o configuraciones distintas, ya sea en los hiper parámetros de Q-Learning o en tipo y configuración de red neuronal utilizada. Las pruebas se basan en las recompensas acumuladas por cada episodio, que como ya hemos visto en epígrafes anteriores, están relacionadas directamente con la distancia recorrida por nuestro agente dentro del nivel. Esto quiere decir que, a mayor recompensa acumulada recibida en un episodio, más distancia se ha logrado recorrer y, por consiguiente, mejores resultados estamos obteniendo en el periodo de aprendizaje.

El motivo de centrarse en las recompensas/distancias para valorar si un aprendizaje ha sido bueno o no es porque, si bien es cierto que el agente debe ser capaz de aprender a superar un nivel completo, es posible que esto no sea así. Sin embargo, esto no quiere decir que no se haya conseguido aprender nada. Al ser un problema tan amplio y complejo, en el que los hiper parámetros y el tipo de modelo son muy delicados de configurar, junto con la gran cantidad de tiempo que requiere ejecutar una única vez el algoritmo (podemos hablar de días enteros), es posible que el agente sólo aprenda a avanzar hasta cierto punto del nivel, lo cual también se puede considerar como un buen resultado. Para desarrollar unas buenas pruebas, el primer paso es la correcta elección de los hiper parámetros que definen cada arquitectura.

A continuación, se van a presentar las pruebas de las cuales se han podido extraer los resultados más interesantes. Se describirán las configuraciones utilizadas junto con sus resultados a modo de gráfica junto con una interpretación de las mismas explicaciones de lo que sucede en cada una de ellas.

5.1 Prueba 1

La configuración utilizada para el desarrollo de esta primera prueba es la siguiente:

Parámetros del aprendizaje	
Partidas	2000
Peso de recompensas futuras - γ	0.9
Learning rate	0.00001
Tamaño memoria D	10000. Cuando llena su espacio, se resetea
Tamaño minibatch	32.
Epsilon max	1
Epsilon min	0.1
Recompensas	Las recompensas utilizadas son las que se reciben del entorno, no es necesario hacer modificaciones.
Lista de acciones	RIGHT_ONLY

Tabla 5.1. Configuración prueba 1

Modelo

El modelo utilizado para realizar esta prueba es el que se corresponde con la primera arquitectura explicada antes, sin capas convolucionales.

Resultados

La gráfica utilizada para explicar los resultados obtenidos es la siguiente, en la que se representan las recompensas acumuladas por cada episodio o nivel.

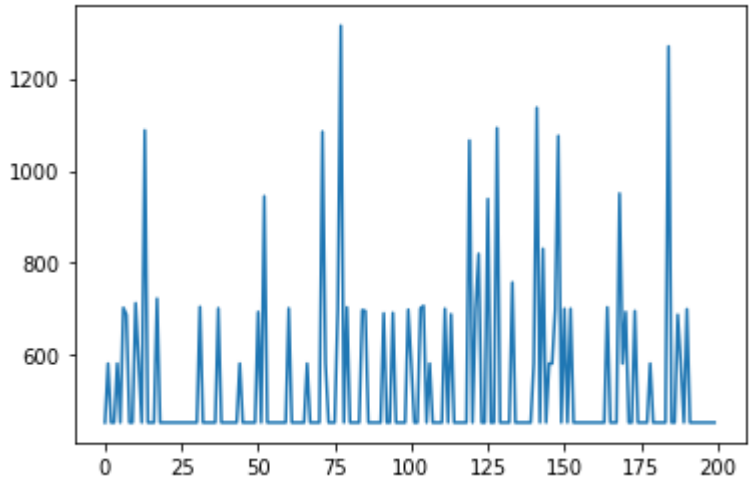

Ilustración 5.1. Gráfica de recompensas prueba 1

Viendo los resultados obtenidos podemos deducir que han sido muy malos. La información que recibe como parámetro la red neuronal era demasiado simple, no se tiene en cuenta las posiciones de los enemigos ni de los obstáculos. Otro problema que este modelo ofrece es la utilización de los datos de un único frame como entrada de la red neuronal, lo que imposibilita al agente a predecir trayectorias y posiciones de los enemigos, y por consiguiente reduce su capacidad de aprendizaje al mínimo. Por último, otro problema añadido es que la posición de los enemigos no es siempre la misma desde que comienza el nivel, sino que a medida que el personaje avanza en el eje X, los enemigos van apareciendo. Esto hace que, aunque nos entremos en episodios distintos con el tiempo y las posiciones iguales para ambos episodios, los enemigos puedan no estar en la misma posición. Todos estos motivos han hecho descartar la inclusión de estos resultados en el resumen de las pruebas.

5.1 Prueba 2

La configuración utilizada para el desarrollo de esta primera prueba es la siguiente:

Parámetros del aprendizaje	
Partidas	2000
Peso de recompensas futuras - γ	0.9
Learning rate	0.00001
Tamaño memoria D	10000. Cuando llena su espacio, se resetea
Tamaño minibatch	32.
Epsilon max	1
Epsilon min	0.1
Recompensas	Las recompensas utilizadas son las que se reciben del entorno, no es necesario hacer modificaciones.
Lista de acciones	RIGHT_ONLY

Tabla 5.2. Configuración prueba 2

Modelo

El modelo utilizado para realizar esta prueba es el que se corresponde con la segunda arquitectura explicada antes.

Resultados

A lo largo de esta ejecución, se mostraba el avance de cada episodio para poder hacer una valoración visual del mismo. Conforme los episodios iban avanzando, se obtenían mejores resultados. El agente poco a poco era capaz de avanzar más en el nivel. Muy a nuestro pesar, llegamos a un punto en el que esto cambió, como podemos ver en la siguiente gráfica de episodios y recompensas.

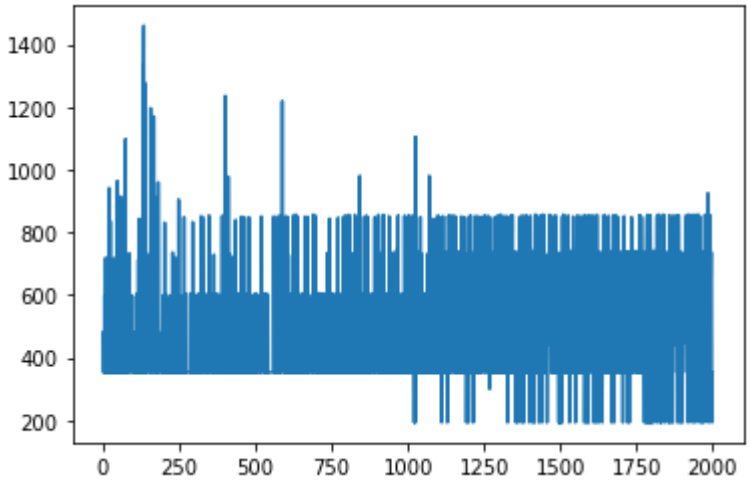

Ilustración 5.2. Gráfica de recompensas prueba 2

En esta tabla vemos como, al principio, el sistema evoluciona como se esperaba. Aprende poco a poco donde están los obstáculos y, por tanto, cada vez consigue una recompensa más alta (avanza más dentro del nivel), llegando incluso casi a superarlo.

Tras llegar a la mitad del aprendizaje la situación se invierte y el agente olvida todo lo que a aprendido. Si bien es cierto, de manera eventual, volverá a aprender a avanzar en el nivel y tendrá recompensas altas, pero a medida que la probabilidad de exploración se reduce, también se reduce el valor de estas. Podemos ver la evidencia de lo comentado en los episodios finales.

¿Por qué sucede esto si el agente llega a tener éxito? Bien, el problema radica en el reseteo de la memoria D. Esto se hacía con el fin de aumentar el rendimiento al no tener ocupada tanta memoria, pero, al contrario de lo esperado, el agente ha olvidado lo que ya sabía. Esta es una situación muy común en los sistemas de aprendizaje por refuerzo profundo conocido como olvido catastrófico

Este olvido y reaprendizaje parece ser una característica constante del algoritmo DQN. Tanto el DQN básico como el completo tienen este problema, llamado olvido catastrófico.

5.2 Prueba 3

Para la ejecución de esta primera prueba se ha eliminado el punto en el que la memoria se resetea cuando está llena. La configuración utilizada es la siguiente.

Parámetros de aprendizaje

Parámetros del aprendizaje	
Partidas	2000
Peso de recompensas futuras - γ	0.9
Learning rate	0.00001
Tamaño memoria D	1000. No se resetea, sino que se desencola cuando se introduce un valor nuevo y está lleno.
Tamaño minibatch	32
Epsilon max	1
Epsilon min	0.1
Recompensas	Las recompensas utilizadas son las que se reciben del entorno, no es necesario hacer modificaciones. Para el entrenamiento sólo se toman las positivas
Lista de acciones	RIGHT_ONLY

Tabla 5.3. Configuración prueba 3

Modelo

El modelo utilizado en esta prueba vuelve a corresponderse con la segunda arquitectura explicada antes.

Resultados

En esta ocasión sí podemos decir que se han obtenido buenos resultados. Durante la ejecución del algoritmo se ha hecho un análisis visual de lo que ocurría. Mediante este análisis se podía ver que cuantos más episodios llevaba ejecutados, más avanzaba en el nivel.

Podemos interpretar estos resultados analizando la siguiente gráfica de episodios/recompensas.

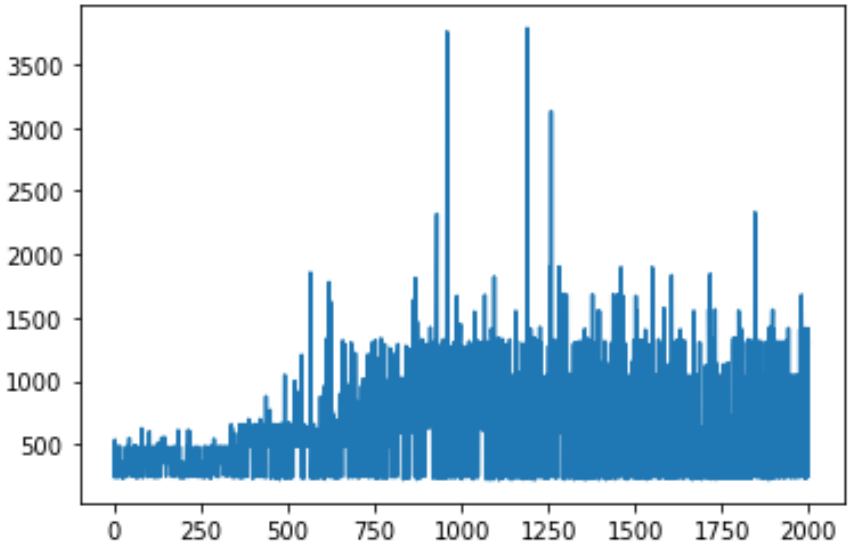

Ilustración 5.3. Gráfica de recompensas prueba 3

Tras la ejecución del algoritmo de aprendizaje, como bien se especifica en los casos de uso, se ha procedido a la visualización de lo aprendido. Se ha podido observar como el agente es capaz de avanzar a lo largo del nivel, lo que denota buenos resultados, pero no consigue superarlo. Esto puede deberse a falta de exploración o al uso de una red neuronal ineficiente.

5.3 Prueba 4

En esta tercera prueba se pretende mejorar la eficiencia de la prueba anterior, por lo que ahora utilizaremos la técnica de aprendizaje por refuerzo profundo doble. La configuración utilizada es la siguiente.

Parámetros de aprendizaje

Parámetros del aprendizaje	
Partidas	2000
Peso de recompensas futuras - γ	0.9
Learning rate	0.00001
Tamaño memoria D	20000. Igual que la prueba anterior, desencola si está lleno
Tamaño minibatch	64
Epsilon max	1
Epsilon min	0.1
Recompensas	Las recompensas utilizadas son las que se reciben del entorno, no es necesario hacer modificaciones.
Lista de acciones	RIGHT_ONLY

Tabla 5.4. Configuración prueba 4

Modelo

El modelo presente en este experimento es el que se corresponde con la tercera arquitectura explicada, es decir, la doble red convolucional.

Resultados

En la exploración visual realizada durante la ejecución, se han observado resultados similares a los del experimento 2, lo cual es buena señal. Tras el entrenamiento se nos presenta la siguiente gráfica de recompensas/episodios.

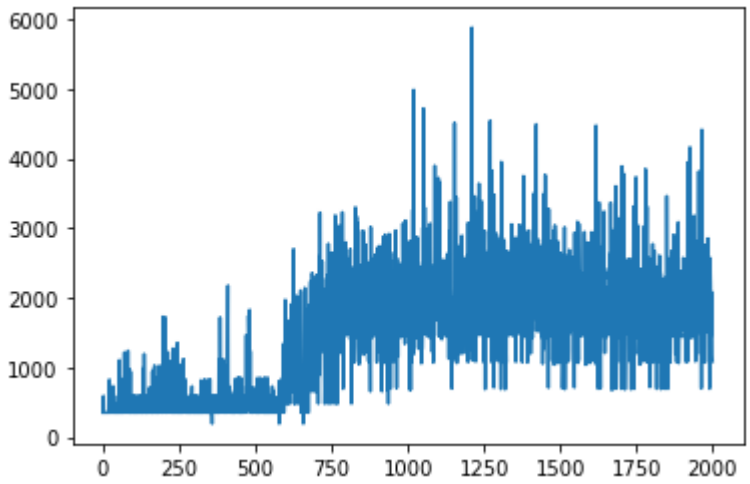

Ilustración 5.4. Gráfica de recompensas prueba 4

En esta tabla de resultados vemos como el algoritmo tiende a converger más rápido que en la prueba 2. A medida que el número de episodios avanza, el número de recompensas acumuladas tiende a ser mayor, aún con repuntes a la baja debidos al proceso de exploración. Estas recompensas poseen los máximos relativos y absoluto más altos de todas las pruebas realizadas, lo que quiere decir que se ha conseguido mejorar la eficiencia del algoritmo introduciendo la red neuronal *target.* El pico en la gráfica cuyo valor es de casi 6000 indica que el agente ha conseguido superar el nivel durante el entrenamiento, lo cual ha sucedido tras alrededor de unas1100 partidas

5.4 Prueba 5

En esta última prueba se ha realizado una modificación del modelo de la prueba 3. En esta ocasión se han utilizado filtros en las capas convolucionales de dimensiones 3x3. Con esta modificación se espera tener una convergencia más rápida ya que con un filtro pequeño la red neuronal puede encontrar características de las imágenes más en profundidad.

Parámetros de aprendizaje

Parámetros del aprendizaje	
Partidas	2000
Peso de recompensas futuras - γ	0.9
Learning rate	0.00001
Tamaño memoria D	20000. Igual que la prueba anterior, desencola si está lleno
Tamaño minibatch	64
Epsilon max	1
Epsilon min	0.1
Recompensas	Las recompensas utilizadas son las que se reciben del entorno, no es necesario hacer modificaciones.
Lista de acciones	RIGHT_ONLY

Tabla 5.5. Configuración prueba 5

Modelo

Como esta prueba es una modificación de la prueba anterior, se ha vuelto a realizar la doble red neuronal

Resultados

Como podemos ver en la gráfica de resultados, esta ocasión se han conseguido unos resultados más inestables. Si bien es cierto que la gráfica tiene tendencia ascendente, aunque los episodios avancen siempre hay partidas en la que la puntuación es baja. Los resultados no son estables, aunque el número de episodios avancen, siempre tenemos mínimos relativos.

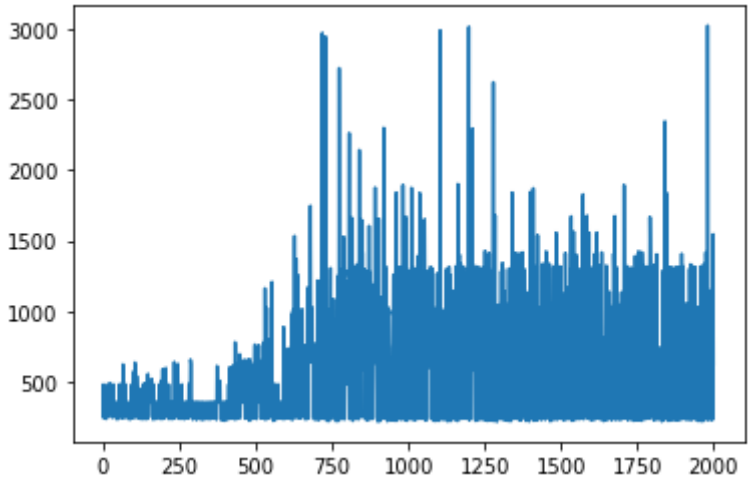

Ilustración 5.5. Gráfica de recompensas prueba 5

De esta gráfica podemos deducir que, aunque durante el aprendizaje el agente tiende a avanzar en el nivel, los resultados no son tan buenos como lo esperado tras los resultados de las pruebas anteriores.

5.5 Comparativa de resultados

A modo de resumen y para exponer los resultados de una forma más clara y concisa, se presenta a continuación una tabla comparativa de todas las pruebas realizadas a lo largo del proyecto.

Prueba	Modelo	¿Supera el nivel?	Episodio
1	Red neuronal sin capas convolucionales	No	No converge
2	Red neuronal con capas convolucionales	No (olvido catastrófico)	No converge
3	Una red neuronal	No	No converge
4	Doble red neuronal	Sí	1100 aprox.
5	Doble red neuronal	No	No converge

Tabla 5.6. Comparativa de resultados

Capítulo 6 Conclusiones

Este capítulo es el cierre de todo el trabajo desarrollado. Siempre es importante finalizar un trabajo con una serie de conclusiones donde se proyecten todos los conocimientos obtenidos durante todo el proceso.

El aprendizaje de refuerzo es un campo apasionante dentro del ámbito del aprendizaje máquina que ofrece una amplia gama de posibles aplicaciones en la ciencia y los negocios. Sin embargo, el entrenamiento de los agentes de aprendizaje de refuerzo es todavía bastante engorroso y a menudo requiere una tediosa puesta a punto de los hiper parámetros y la arquitectura de la red para que funcione bien. Esto hace que los proyectos de esta índole sean complicados de abordar y suponen un reto muy interesante para todo el que se enfrente a uno.

Al comienzo del proyecto se aclaró que el objetivo principal del mismo era el desarrollo de este proyecto es desarrollar un algoritmo para aplicar la técnica de aprendizaje automático Deep Q-Learning a un videojuego de plataformas y, así, conseguir que el personaje del videojuego aprenda una correspondencia entre el estado actual del juego, las posibles acciones y las recompensas obtenidas que le hagan capaz de superar los niveles de forma autónoma. Bajo mi punto de vista, este objetivo ha sido cumplido en mayor o menor medida. Para lograr el objetivo principal se han debido cumplir una serie de objetivos secundarios que fueron establecidos en el primer capítulo de esta memoria. Estos objetivos secundarios, junto con una breve conclusión obtenida de cada uno de ellos, son los siguientes:

- Entender el problema a tratar mediante un análisis y estudio del área de aprendizaje por refuerzo. Para cumplir este objetivo se ha hecho una búsqueda exhaustiva de información en libres y documentación que queda reflejada en el estado del arte de esta memoria.
- Investigar trabajos similares. Junto con el objetivo anterior, también se han realizado búsquedas detenidas sobre trabajos similares al nuestro que han ayudado a abordar el proyecto. Por ejemplo, ha servido como referencia para construir los modelos de las redes neuronales utilizados.
- Analizar y explicar las diferentes tareas que se realizará, junto con su planificación temporal. Uno de los principales y necesarios en el desarrollo de un proyecto es la planificación de las tareas que se van a desarrollar junto con

el tiempo que va a ocupar cada una de ellas. Es un punto delicado ya que una mala planificación puede suponer el fracaso. De cara a trabajos futuros servirá como experiencia para no errar en sus planificaciones y así evitar que se produzcan desviaciones temporales y retrasos.

- Aprender a utilizar la herramienta de Python OpenAI Gym. Este es el punto de partida del trabajo ya que con este framework es como se implementa el agente utilizado para entrenar en nuestro algoritmo por lo que es necesario entender cómo utilizarlo. Este sencillo objetivo ha sido cumplido a la perfección.
- Elaborar una comparativa de rendimiento y eficacia al utilizar diferentes formas de recoger la información de los estados del videojuego que hagan al personaje aprender.
- Recoger la información visual del videojuego mediante código. Para poder realizar el aprendizaje era necesario obtener una buena información sobre el entorno en el que el agente se está moviendo, por lo que en nuestro código se necesitaba extraer la mejor información posible de dicho entorno para representar el estado. Esta información son cada frame del juego, lo cual se ha conseguido sin muchas complicaciones.
- Reducir dimensionalidad de imágenes. En el capítulo donde se explica la implementación se ha explicado la necesidad de reducir la dimensionalidad de las imágenes y cómo se ha realizado, por lo que es otro objetivo complicado que se ha conseguido.
- Entender e implementar el algoritmo Q-Learning. Saber cómo funciona el aprendizaje por refuerzo es otro punto importante para el desarrollo del proyecto. No se puede realizar un algoritmo sin los conocimientos previos. Tras estudios y experiencias en trabajos pasados no ha sido muy complicado entenderlo, por lo que este objetivo se ha cumplido y el conocimiento objetivo puede ser de utilizad para trabajos futuros del mismo ámbito.
- Entender los diferentes modelos de redes neuronales y ser capaz de realizar su implementación. Lar redes neuronales son, casi seguro, la parte más compleja de este proyecto, no solo por el aspecto teórico, sino por lo delicadas que son. Una mala elección de hiper parámetros hará que una red nunca llegue a converger.
- Ser capaz de analizar y entender los resultados obtenidos en cada experimento realizado al final del proyecto. En esta memoria se ha presentado una lista de pruebas realizadas con tablas explicativas de cómo se han desarrollado dichas pruebas junto con los resultados interpretados y comentados de dichas pruebas.

- Lograr un algoritmo escalable a diferentes niveles e incluso a diferentes juegos. La falta de pruebas relacionadas con este objetivo no puede garantizar que haya sido cumplido en su totalidad, aunque cabe destacar que es muy probable que el algoritmo sea escalabre, al menos, a otros niveles del juego. Como comentaremos a continuación, esta tarea queda pendiente para un posible trabajo futuro.
- Presentar un conjunto de conclusiones obtenidas tras la realización del TFM junto con el posible trabajo futuro en base a los conocimientos obtenidos durante su desarrollo

Como la mayoría los objetivos secundarios se han cumplido, podemos afirmar que el objetivo principal del trabajo se ha conseguido.

A partir del trabajo que se ha llevado a cabo se puede reflexionar sobre qué modificaciones se podrían realizar en un trabajo futuro a modo de ampliación y mejora del sistema realizado y seguir aumentando conocimientos en el ámbito del aprendizaje por refuerzo profundo y las redes neuronales. La principal tarea que se queda pendiente para el futuro es aplicar el algoritmo a nuevos niveles del juego y, sobre todo, a nuevos juegos de plataformas cuyo funcionamiento sea similar al de Super Mario Bros, uno de los aspectos que hacen que un algoritmo sea de calidad es la escalabilidad. También es interesante el utilizar otro tipo de técnicas, como por ejemplo los algoritmos genéticos, los cuales proporcionan una gran fiabilidad para proyectos de este tipo.

Para finalizar este trabajo me gustaría remarcar que me ha proporcionado muchos conocimientos nuevos que me pueden servir de apoyo para dedicarme en un futuro a este campo de la ingeniería. Además, me ha ayudado a mejorar mi capacidad de recopilación bibliográfica y de síntesis de información, ya que mis conocimientos sobre el aprendizaje profundo al comienzo del proyecto eran nulos.

.

Capítulo 7 Glosario

Término	Definición
API	Siglas de Application Programming Interface, conjunto de reglas y códigos que las aplicaciones pueden utilizar para comunicarse entre ellas. Es una interfaz entre programas diferentes
Campo receptivo	Hiper parámetro de las capas convolucionales de las redes neuronales. Hace referencia a la parte de la imagen que es visible por un filtro en in instante concreto de tiempo. Se incrementa linealmente a medida que se apilan mas capas convolucionales y se reduce exponencialmente cuando se apilan capas convolucionales atroces.
Clusters	Forma de clasificación del aprendizaje no supervisado. Se trata de grupos de elementos que comparten una serie de características específicas.
CNN	Acrónimo de Convolutional Neural Network (Red Neuronal Convolucional)
DNN	Acrónimo de Deep Neural Network (Red Neuronal Profunda)
Gradiente	Variación de una magnitud en base a la distancia
Linealmente separable	Funciones que se pueden separar en el espacio por una línea recta, de tal forma que representan características diferentes.
Machine Learning	Disciplina científica del ámbito de la Inteligencia artificial mediante la cual se crean sistemas capaces de aprender automáticamente.
Mapa de características	Resultado de aplicar un filtro a una imagen que pasa a través de una capa convolucional
On-policy	Algoritmos que estiman el valor de la política a la vez que la usan para el control. El fin es mejorar la política que se usa para la toma de decisiones
Peso de conexión	Representan la conexión de las neuronas de una capa con todas las neuronas de la capa siguiente. Los valores de estos pesos se van actualizando de manera que se busca reducir el valor de la función de coste.
Q-Learning	Técnica del ámbito del aprendizaje por refuerzo utilizada en el aprendizaje

	automático.
Recompensa	Número entero recibido a modo de incentivo o premio cuando en un algoritmo de aprendizaje por refuerzo se realiza una acción. Indica lo buena que es la acción realizada
RL	Acrónimo de Reinforcement Learning (Aprendizaje por refuerzo)
SARSA	Técnica del ámbito del aprendizaje por refuerzo utilizada en el aprendizaje automático.

Capítulo 8 Bibliografía

Mnih, Volodymyr; Kavukcuoglu, Koray; Silver, David; Graves, Alex; Antonoglou, Ioannis; Daan, Wierstra; Riedmiller, Martin (19 de diciembre de 2013). "Background" *Playing atari with deep reinforcement learning* [artículo en línea].

David Silver; Aja Huang; Chris J. Maddison; Arthur Guez; Laurent Sifre; George van den Driessche; Julian Schrittwieser; Ioannis Antonoglou; Veda Panneershelvam; Marc Lanctot; Sander Dieleman; Dominik Grewe; John Nham; Nal Kalchbrenner; Ilya Sutskever; Timothy Lillicrap; Madeleine Leach; Koray Kavukcuoglu; Thore Graepel; Demis Hassabis (28 de enero de 2016). "Evaluating the playing strength of AlphaGo" *Mastering the game of Go with deep neural networks and tree search* [artículo en línea].

Pathak, Deepak; Agrawal, Pulkit; Efros, Alexei; Darrell, Trevor (15 de mayo de 2015). *"*Experimental Setup*" Curiosity-driven exploration by self-supervised prediction* [artículo en línea].

Shapiro, Stuart C (1992). *Encyclopedia of Artificial Intelligence*. Nueva York: John Wiley & Sons. 0471503053

Murphy, Kevin P (2012). Machine Learning: A Probabilistic Perspective. Cambridge: MIT Press. 9780262018029.

Michalski, R.S., Carbonell, J.G., Mitchell, T.M (1983). *Machine learning: An artificial intelligence approach* (1a. edición). Londres:Springer Publishing Company

Goodfellow, Ian; Bengio, Yoshua; Courville, Aaron (2016). "Deep Learning Research". En: *Deep Learning* ["Aprendizaje Profundo"] (1a. Edición). San Francisco:MIT Pres.

Haykin, Simon (1993). *Neural networks and Learning machines* (3a edición). Nueva York:Pearson

Rosenblatt, F. (1958). T*he perceptron: A probabilistic model for information storage and organization in the brain*. *Psychological Review, 65*(6), 386–408.

De Armas Teyra M (2009). *Fundamentos básicos de Inteligencia Artificial con Matlab.* (1a. Edición). EDUCOSTA

Nandy, Abhishek; Biswas, Manisha (2017). Reinforcement Learning With Open AI, TensorFlow and Keras Using Python (1a. Edición). Nueva York:Apress.

Kolen, John F.; Kremer,**Stefan C.** (2001), "Gradient Flow in Recurrent Nets: The Difficulty of Learning LongTerm Dependencies". *En A Field Guide to Dynamical Recurrent Networks*, IEEE, pp.237-243, doi: 10.1109/9780470544037.ch14.

Simon Haykin.Neural networks and Learning machines. Pearson, 1993.

Kingma, Diederik P.; Ba, Jimmy (22 de diciembre de 2014)."Algorithm". *Adam: A Method for Stochastic Optimization* [artículo en línea].

[**Ballard, Will** (2018). *Hands-On Deep Learning for Images with TensorFlow: Build intelligent computer vision applications using TensorFlow and Keras* (1a. edición). Birmingham: Packt Publishing

Lapan, Maxim (2018). *Deep Reinforcement Learning Hands-On: Apply modern RL methods, with deep Q-networks, value iteration, policy gradients, TRPO, AlphaGo Zero and more* (1a. edición). Birmingham: Packt Publishing

Saito, Sean; Ravichandiran, Sudharsan; Shanmugamani, Rajalingappaa (2019). "Playing Atari Games". En: *Deep Reinforcement Learning Hands-On: Apply modern RL methods, with deep Q-networks, value iteration, policy gradients, TRPO, AlphaGo Zero and more* (1a. edición). Birmingham: Packt Publishing

Standford university.CS231n: *Convolutional Neural Networks for Visual Recognition. Modulo 1, Modulo 2*

Richard S. Sutton, Andrew G. Barto (2017)**.** *Reinforcement Learning: An Introduction. Cambridge*:MIT Press

Murphy, Kevin P. Machine Learning (2012): *A Probabilistic Perspective. Cambridg*e MIT Press.

Tamar, Aviv; Di Castro, Dotan, Meir, Ron (2012). *Integrating a partial model into Model free reinforcement learning. Journal of Machine Learning Researc*h,13:1927–1966, 2012.

Puterman, Martin L (1994)**.** *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. New York: John Wiley & Sons. 0471619779.

Fernández Rebollo, F. (2002). *Aprendizaje por refuerzo en espacios continuos*. Tesis Doctoral. Universidad Carlos III de Madrid Escuela Politécnica Superior.

Martn Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro,Greg S. Corrado, Andy Davis, Je rey Dean, Matthieu Devin, Sanjay Ghemawat, **Ian Good-fellow, Andrew Harp, Geo rey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, LukaszKaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mane, Rajat Monga, Sherry Moore,Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever,Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viegas, OriolVinyals, Pete Warden,**

Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng (2015)**.***TensorFlow: Large-scale machine learning on heterogeneous systems*,.

Chollet, Francois (2017). *Deep Learning with Python.* San Diego: Manning Publications 978-1-61729-443-3

Capítulo 9 Enlaces consultados

A lo largo del transcurso proyecto, además de la bibliografía, se han consultado numerosos enlaces para recopilar información y adquirir conocimientos de ayuda y ampliación. Estos enlaces son los siguientes:

Using Reinforcement Learning to play Super Mario Bros on NES using TensorFlow

https://www.statworx.com/de/blog/using-reinforcement-learning-to-play-super-mario-bros-on-nes-using-tensorflow/

Super Mario Bros. Bot with Reinforcement Learning
https://medium.com/datadriveninvestor/super-mario-bros-reinforcement-learning-77d6615a805e

Gym-Super-Mario
https://github.com/ppaquette/gym-super-mario

Getting Mario Back into the Gym: Setting up Super Mario Bros in OpenAI's gym
https://becominghuman.ai/getting-mario-back-into-the-gym-setting-up-super-mario-bros-in-openais-gym-8e39a96c1e41

Keras
https://keras.io/

Teaching your computer to play Super Mario Bros
https://www.ehrenbrav.com/2016/08/teaching-your-computer-to-play-super-mario-bros-a-fork-of-the-google-deepmind-atari-machine-learning-project/

Deep Q-Learning to Play Mario
http://cs229.stanford.edu/proj2016/report/klein-autonomousmariowithdeepreinforcementlearning-report.pdf

An introduction to Deep Q-Learning: let's play Doom
https://www.freecodecamp.org/news/an-introduction-to-deep-q-learning-lets-play-doom-54d02d8017d8/

Colab Super Mario

https://colab.research.google.com/drive/1rj50M2J4iYqMorbhdAvMaLnKTaUMf7sb#scrollTo=6D-IfncyslMJ

My Journey into Deep Q-Learning with Keras and Gym
https://medium.com/@gtnjuvin/my-journey-into-deep-q-learning-with-keras-and-gym-3e779cc12762

Zurich R-USER Gruop. Reinforcement Learning Using R
https://zurich-r-user-group.github.io/slides/20190618_reinforcelearn.pdf

NN-SVG

http://alexlenail.me/NN-SVG/LeNet.html

Let's build a DQN: Simple implementation
https://tomroth.com.au/dqn-simple/

Gym Retro Documentation
https://readthedocs.org/projects/retro/downloads/pdf/latest/

API Funcional de Keras
https://torres.ai/api-funcional-de-keras/

OpenAI Gym for NES games + DQN with Keras to learn Mario Bros from raw pixels
https://naereen.github.io/gym-nes-mario-bros/

Deep Learningn básico con Keras

https://enmilocalfunciona.io/deep-learning-basico-con-keras-parte-1/

Clasificación de imágenes en Python

https://www.aprendemachinelearning.com/clasificacion-de-imagenes-en-python/

Deep Learning fácil con DeepCognition

https://planetachatbot.com/deep-learning-f%C3%A1cil-con-deepcognition-9af43b2319ba

Eduardo Morales, Hugo Jair Escalante. Curso aprendizaje por refuerzo

https://ccc.inaoep.mx/~emorales/Cursos/Aprendizaje2/Acetatos/refuerzo.pdf.

Redes neuronales
https://ml4a.github.io/ml4a/es/neural_networks/