UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Implementación y comparación de diferentes algoritmos de Machine Learning para el aprendizaje y resolución del videojuego Super Mario World (SNES)

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Autor: Álvaro Rodríguez Sánchez

Tutor: Carlos David Martínez Hinarejos

Curso 2020-2021

Resum

L'objectiu del TFG proposat és desenvolupar i implementar dos algorismes diferents basats en *machine learning* en el videojoc *Super Mario World* de la consola *Super Nintendo Entertaiment System* (SNES), utilitzant per a això la llibreria *gym-retro* que ens ofereix *OpenAI* en el llenguatge de programació *Python*. El primer algorisme que serà utilitzat serà un algorisme evolutiu en què s'anirà aprenent mitjançant una *fitness function* i la selecció de les millors mostres per crear el millor jugador possible; aquest algorisme serà implementat mitjançant *NEAT-Python*. El segon algorisme utilitzat serà un algorisme d'aprenentatge per reforç, en concret *Deep Q-Learning*, que quan s'ha aplicat a jocs de poca complexitat ha obtingut resultats prometedors. Finalment, es passarà a la comparació dels resultats obtinguts després d'usar sengles algorismes, podent així demostrar quin dels dos algorismes resulta més interessant per a la resolució d'un joc de plataformes.

Paraules clau: *Machine Learning*, algorisme evolutiu, aprenentatge per reforç, *OpenAI*, videojoc, *Mario Bros*, *Nintendo*, xarxes neurals, *DeepQ*.

Resumen

El objetivo del TFG propuesto es desarrollar e implementar dos algoritmos distintos basados en *machine learning* en el videojuego *Super Mario World* de la consola *Super Nintendo Entertaiment System* (SNES), utilizando para ello la biblioteca *gym-retro* que nos ofrece *OpenAI* en el lenguaje de programación *Python*. El primer algoritmo que será utilizado será un algoritmo evolutivo en el que se irá aprendiendo mediante una *fitness function* y la selección de las mejores muestras para crear el mejor jugador posible; este algoritmo será llevado a cabo mediante *NEAT-Python*. El segundo algoritmo utilizado será un algoritmo de aprendizaje por refuerzo, en concreto *Deep Q-Learning*, que al ser aplicado a juegos de poca complejidad ha obtenido resultados prometedores. Finalmente se pasará a la comparación de los resultados obtenidos tras usar sendos algoritmos, pudiendo así demostrar cuál de los dos algoritmos resulta más interesante para la resolución de un juego de plataformas.

Palabras clave: *Machine Learning*, algoritmo evolutivo, apredizaje por refuerzo, *OpenAI*, videojuego, *Mario Bros*, *Nintendo*, redes neuronales, *DeepQ*.

Abstract

The objective of this end-of-degree project is to develop and implement two different algorithms based on Machine Learning for solving the *Super Mario World* video-game on the *Super Nintendo Entertainment System* (SNES) console, using the *gym-retro* library offered by *OpenAI* in the *Python* programming language. The first algorithm to be used Evolutionary Algorithm that will learn through a fitness function and the selection of the best samples to create the best possible player; this algorithm will be carried out using *NEAT-Python*. The second algorithm to be used will be a Reinforcement Learning algorithm, specifically *Deep Q-Learning*, which when it is applied to low-complexity games it has obtained promising results. Finally, we will proceed to the comparison of the results obtained after using both algorithms, thus being able to show which of the two algorithms is more interesting for the resolution of a platform game.

Key words: Machine Learning, evolutionary algorithm, Reinforcement Learning, *OpenAI*, videogame, *Mario Bros*, *Nintendo*, *Neural Networks*, *DeepQ*.

Índice general

Índice de figuras

Índice de tablas

Índice de algoritmos

Siglas

AG Algoritmos Genéticos

DL Deep Learning

DNN Deep Neural Network

DQN Deep Q Network

EA Evolutionary Algorithms

EE Estrategias Evolutivas

IA Inteligencia Artificial

LSMPE Least-Squares Methods for Policy Evaluation

LSPI Least-Squares Policy Iteration

LSTD Least-Squares Temporal Differencing

MaOP Many-objective Problems

MDP Markov Decision Processes

ML Machine Learning

MOP Multiobjective Problems

MP Método Porcentajes

MR Método Resta

NN Neural Networks

PE Programación Evolutiva

RL Reinforcement Learning

SMW Super Mario World

SNES Super Nintendo Entertaiment System

Agradecimientos

Me gustaría transmitir mi más sincero agradecimiento a todos aquellas personas que me han ayudado a lo largo de este trabajo y han colaborado en esta investigación. En primer lugar, a mi tutor Carlos David, tanto por apoyar mi propuesta de trabajo, como por consolidarla ayudándome en su planificación, organización y dudas surgidas a lo largo de su desarrollo. En segundo lugar, a mi familia, tanto mi padre José Manuel, como su mujer Ornela, como mis hermanos José Manuel, Aleksandar y Lucía, que siempre han estado dándome apoyo anímico a pesar de que algunos de ellos han estado residiendo en el extranjero durante mi periodo universitario. A mis amigos, con los que he superado las distintas asignaturas del grado, apoyándonos y ayudándonos mutuamente tanto en el ámbito de estudio como en el ámbito personal. También, expresar mi más sentido agradecimiento a la ETSINF, y por ende a la UPV, por trasladar mi expediente académico a Valencia y acogerme dentro de sus aulas, haciéndome sentir como en casa.

A todos ellos, mil gracias.

CAPÍTULO 1
Introducción

Como bien se sabe, hoy en día la Inteligencia Artificial (IA) es un campo que está en pleno auge y crecimiento. Pero, a diferencia de lo que se cree, el campo de la IA no es un concepto tan actual. Ya desde antes de mediados del siglo XX, figuras tan célebres como importantes en el campo de la computación e informática como Alan Turing o Isaac Asimov fueron pioneros en las bases de esta nueva idea, pues le dieron tanto un enfoque teórico - Test de Turing - como moral y ético - Las tres leyes de la robótica de Isaac Asimov -. Pero no fue hasta el año 1956 cuando John McCarthy introdujo por primera vez este concepto en la conferencia de Dartmouth [1].

A pesar de que no sea un término tan actual como pensamos, no ha sido hasta principios de la década de los 90 y principios de siglo XXI cuando se ha empezado a desarrollar e investigar de forma masiva alrededor de este campo. Esto es debido a que tanto la capacidad computacional como la accesibilidad a un computador antes de la década de los 90 era muy escasa, dificultando por lo tanto el crecimiento en esta área. Tal y como se muestra en la Figura 1.1, ha habido un total de tres *boom* a lo largo de la historia de la IA, siendo éste último el más grande debido a la fácil accesibilidad a un computador y a la gran capacidad de cómputo de la que disponemos actualmente.

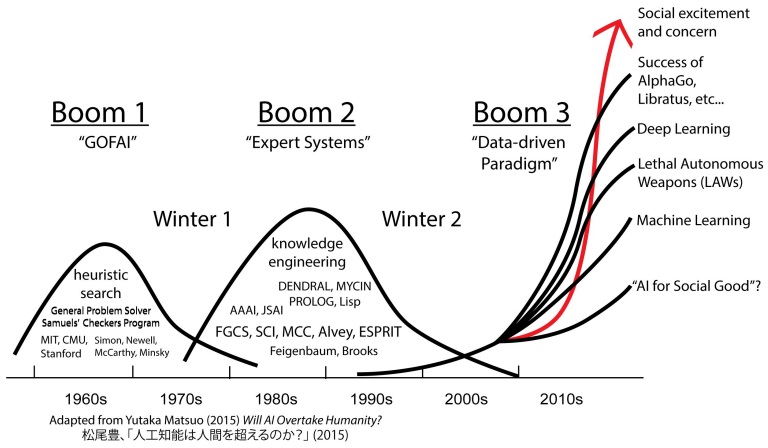

Figura 1.1: Crecimiento en investigación de la IA.

El ser humano siempre ha sentido curiosidad por la posibilidad de diseñar y crear un sistema computacional capaz de simular el cerebro humano. La idea de que un computador pueda pensar, razonar, decidir y realizar tareas tal y como hacen los humanos es una idea deslumbrante, y más cuando un computador es capaz de almacenar y computar una gran cantidad de datos. Como se verá en el Capítulo 3, hay diferentes formas de aplicar la IA dependiendo de la finalidad que se busque.

Otro concepto que lleva revolucionando tanto la sociedad como el ocio desde finales del siglo XX es el mundo de los videojuegos. Desde la creación de uno de los primeros juegos como el de *Tennis for Two* en 1958 por William Higginbotham hasta los videojuegos creados en nuestros días, ha habido una gran evolución al respecto tanto en los gráficos y en las físicas como la forma de jugarlos. Se puede ver cómo se transita de un videojuego que se sirve de un programa para el cálculo de trayectorias y un osciloscopio -videojuego anteriormente mencionado- a juegos que están desarrollados mediante un motor gráfico específico, emuladores/simuladores de luz, motor de físicas muy realista, periféricos adaptados, etc. (Figura 1.2). Al igual que le ocurrió al desarrollo e investigación de la IA, la evolución en los videojuegos ha sido exponencial, pues a medida que se mejoraba el *hardware*, se permitía llevarlo hasta su máximo exponente, haciendo juegos con una complejidad mayor.

Figura 1.2: Recreación de *Tennis for Two* (izquierda) y *AO tennis 2* (derecha).

Actualmente los videojuegos, aparte de ofrecer diversión, se han convertido también en un gran mundo competitivo, conocido actualmente como *esports*, donde al igual que en otros deportes de competición (como por ejemplo el fútbol o el tenis), se lleva a cabo un plan de entrenamiento, y existe un ambiente de competición contra otros jugadores, moviendo todo ello grandes cantidades de dinero.

Estos hechos son los que motivan la realización de este TFG: implementar una IA que, a partir de lo que recibe un agente en el entorno de un videojuego, sea capaz de aprender y tomar sus propias decisiones para alcanzar su objetivo.

1.1 Motivación

Este TFG ha sido motivado por varios aspectos. El primero de ellos es el entusiasmo personal de ser capaz de crear un algoritmo, en el caso de este trabajo dos algoritmos, a partir del cual simplemente con la recolección de datos en pantalla y de definidas unas acciones de movimiento, aprenda para superar niveles de un videojuego. El segundo de los aspectos es un tanto subjetivo y personal, pues el videojuego sobre el que vamos a desarrollar estos algoritmos es el *Super Mario World (SMW)* de la consola *Super Nintendo Entertaiment System (SNES)* de la compañía japonesa de entretenimiento *Nintendo*, puesto que fue uno de los primeros videojuegos que jugué en mi infancia. Finalmente, y no por ello menos importante, el último aspecto está relacionado con el ámbito profesional, ya que considero este trabajo como una gran forma de poner en práctica los conocimientos que he ido adquiriendo a lo largo de la carrera y ser así una carta de presentación a la hora de aspirar a un puesto de desarrollador de IA en cualquier ámbito.

1.2 Objetivos

Con este trabajo se pretende investigar y profundizar en las diferentes técnicas que nos ofrece la IA, concretamente el *Machine Learning*, para alcanzar nuestra meta de manera fluida dentro del entorno dinámico que nos ofrece un videojuego, llegando así al final de la pantalla.

Los objetivos principales que se van a abordar son:

1. Investigación sobre las técnicas y algoritmos que van a ser usados, profundizando en el *Reinforcement Learning (RL)* y *Evolutionary Algorithms (EA)* (aprendizaje por refuerzo y algoritmos evolutivos en castellano respectivamente).
2. Crear y entender la topología de la red neuronal que vamos a diseñar e implementar para el algoritmo basado en *Reinforcement Learning*.
3. Estructurado e implementación de un algoritmo basado en EA.
4. Aprendizaje, adaptación y desarrollo requeridos para combinar un videojuego y los algoritmos anteriormente citados para crear un entorno de aprendizaje automático.
5. Comparación de los dos algoritmos en cuanto a eficacia y eficiencia.

1.3 Estructura de la memoria

Esta memoria se divide en distintos capítulos; en cada uno de ellos se hará énfasis en un determinado ámbito.

El segundo capítulo expone el estado del arte, de forma que se hablará de lo último respecto a las aplicaciones del aprendizaje por refuerzo y de los algoritmos evolutivos en el mundo de los videojuegos.

El tercer capítulo explica cuáles son las distintas categorías dentro de la IA, describiendo así los conceptos de *Machine Learning*, *Deep Learning* y Algoritmos Evolutivos. Además, se encasillarán los dos algoritmos que se van a programar en este trabajo dentro de estas categorías descritas.

El cuarto capítulo introduce la biblioteca *gym-retro*, así como las funciones imprescindibles para controlar las variables y los datos de un videojuego; por este motivo, además, expondremos el nivel del videojuego *Super Mario World* en el que vamos a trabajar. También se hablará sobre la herramienta que ofrece *OpenAI* para mostrar los datos en RAM del videojuego para sacar los valores pertinentes que necesitemos.

El quinto capítulo muestra cómo se ha desarrollado el primer algoritmo (basado en algoritmos evolutivos), las librerías utilizadas para tal fin y los resultados obtenidos.

El sexto capítulo consta del desarrollo del segundo algoritmo (basado en *Reinforcement Learning*), las librerías utilizadas y el estudio de los resultados obtenidos.

El séptimo capítulo compara los resultados obtenidos con sendos algoritmos y concluye cuál de ellos ha resultado más eficiente y eficaz, además de posibles mejoras que se podrían tener en cuenta para un futuro.

El octavo capítulo cierra el presente trabajo con la extrapolación de estos algoritmos a otros entornos y una conclusión final.

CAPÍTULO 2
Estado del arte

A lo largo de este capítulo se va a exponer el estado del arte de los EA como del RL, tanto en el mundo de la investigación, como en el mundo de la ingeniería como en el de los videojuegos.

2.1 Estado del arte en el mundo de la investigación

Desde el comienzo de las ideas del RL y del EA, éstas han ido evolucionando y ramificándose en conceptos e ideas cada vez más robustas y complejas. Todo ello gracias a la investigación, donde, se han utilizado como base los estudios realizados por otros investigadores, para de esta forma mejorar ese nuevo estudio.

2.1.1. Mejora en la evaluación de la política dentro del *Reinforcement Learning*

Como ya se comentará en el Capítulo 6, el concepto de política es importante dentro del RL, puesto que ésta es un método por el que, en función del estado actual en el que se encuentra el agente, se le va a proponer realizar una acción u otra para maximizar una puntuación asociada al agente, llamada *reward*. El objetivo del RL es pues, encontrar una política que optimice el *reward*. La iteración de políticas funciona evaluando y mejorando iterativamente las políticas; la mejora de ésta es estrictamente mejor que la anterior. En el Algorithm 2.1 se muestra la forma general en la que se evalúan iterativamente estas políticas; estos conceptos se verán en más profundidad en el Capítulo 6. En el Algorithm

Algorithm 2.1 Iteración para la búsqueda de la política óptima

1: **input** política inicial π_0
2: $k \leftarrow 0$
3: **repeat**
4: encontrar Q^{π_k}
5: $\pi_{k+1}(s) \leftarrow argmax_{a \in A} Q^{\pi_k}(s, a)$
6: $k \leftarrow k + 1$
7: **until** $\pi_k = \pi_{k-1}$
8: **output** $\pi^* = \pi_k, Q^* = Q^{\pi_k}$
9: **return** =0

2.1, las líneas 4 y 5 muestran la evaluación de la política y la mejora de la política, respectivamente. A lo largo de las distintas investigaciones se han implementado diferentes formas para evaluar la política; generalmente usando ecuaciones lineales. Recientemente

se investigó sobre el uso de ecuaciones cuadráticas en lugar de lineales para la evaluación de la política; un ejemplo de ello es el *Least-Squares Methods for Policy Evaluation (LSMPE)* [2], conocido en castellano como el método de mínimos cuadrados para la evaluación de la política. Este método en sí tiene buenos resultados en la evaluación de la política, pero en ciertas ocasiones también se encuentra con dificultades en la obtención de buenos resultados en la evaluación, por lo que a partir de él se derivaron otros dos métodos (también cuadráticos), que solucionan dichas dificultades:

- *Least-Squares Temporal Differencing (LSTD)*.
- *Least-Squares Policy Iteration (LSPI)*.

2.1.2. *Many-Objective Evolutionary Algorithms*

Actualmente, los *Multiobjective Problems (MOP)* [3], problemas con múltiples objetivos en castellano, son muy recurrentes cuando los objetivos de éstos son minimizaciones. Cuando existen al menos cuatro objetivos, a estos MOP se les categoriza como *Many-objective Problems (MaOP)*; éstos consisten en problemas que se aplican a problemas del mundo real, como puede ser en ingeniería del diseño, en control del tráfico aéreo, en optimización de conducción automática de automóviles, planificación del suministro de agua, etc. Recientemente, la investigación en los MaOP ha estado muy presente en campos como la IA y la computación evolutiva, trayendo consigo la creación de 238 artículos con más de 8000 citas relacionadas con los MaOP. Un MaOP puede ser expresando como se puede se muestra en las ecuaciones (2.1) y (2.2).

$$\text{minimizar F}(x)=(\text{f}_1(x), f_2(x), ..., f_m(x))^T \tag{2.1}$$

$$\text{sujeto a } x \in \Omega \tag{2.2}$$

Donde:

- $\boldsymbol{x}$ es el vector de decisión del espacio de decisiones Ω.
- $\boldsymbol{m}$ es el numero de objetivos (donde m >3)

El objetivo de la optimización multiobjetivo es aproximar el frente de Pareto en el espacio objetivo para que no se pueda lograr ninguna mejora adicional en ningún objetivo sin perjudicar a los demás.

Como se explica en el Capítulo 5, los EA se basan en una población que no necesitan supuestos particulares como la continuidad o diferenciabilidad. Es por ello por lo que son ideales para tratar con los MOP. En las últimas décadas, los investigadores han propuesto muchos *Multiobjective Evolutionary Algorithms (MOEA)*, que se pueden clasificar en cuatro clases:

- Basado en Pareto.
- Basado en agregación.
- Basado en indicadores.
- Basado en preferencias.

Sin embargo, cuando se trata de un MaOP, es más probable que los MOEA tradicionales no converjan al frente de Pareto, ya que el problema del deterioro se vuelve más prevalente con un número creciente de objetivos. Es por ello por lo que los investigadores han diseñado varios algoritmos para superar estos obstáculos en función del tipo de MOEA (Figura 2.1).

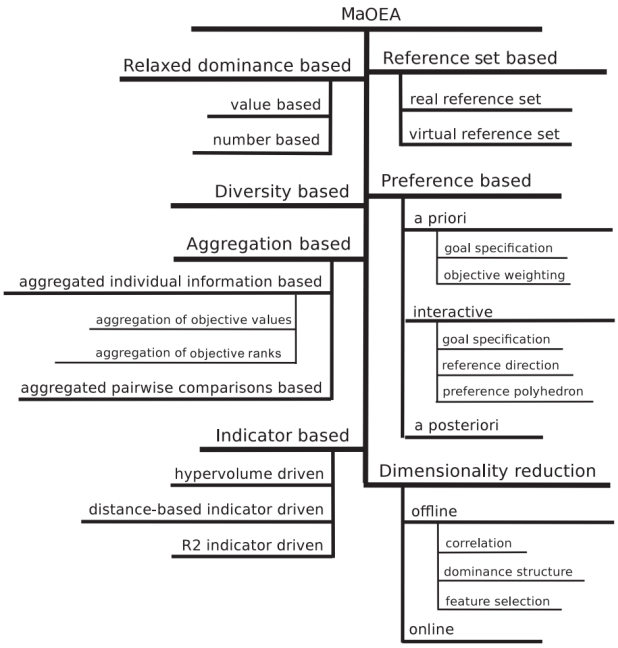

Figura 2.1: Clasificación de los MOEA.

2.2 Estado del arte en el mundo de la ingeniería

Como bien es sabido, hoy en día el término IA se encuentra presente de una forma u otra en la mayoría de los sistemas que contengan un gran conjunto de datos y que deban ser optimizados o clasificados. A continuación se exponen aplicaciones de IA, en concreto en RL y EA, en ámbitos de ingeniería en el momento actual.

2.2.1. Transformando la manufacturación mediante *Reinforcement Learning*

A día de hoy, las *startups Covariant*, *Ocado's Kindred* y *Bright Machines* [4] están utilizando el RL para cambiar la forma en que se controlan las máquinas en las fábricas y almacenes, resolviendo desafíos de gran complejidad, como hacer que los robots detecten y recojan objetos de diferentes tamaños y formas de contenedores.

Por otro lado, *X Development*, organización fundada por *Google*, en el año 2016 anunció sus "granjas de brazos robóticos", que consistían en espacios llenos de brazos robóticos que estaban aprendiendo a agarrar elementos de distintos tamaños y formas. Además, éstos se encargaban de enseñar a otros brazos cómo hacer lo mismo que acababan de aprender ellos. Para ello, al igual que en el caso de las *startups*, se aplicó un algoritmo basado en RL que, a partir de sus bases (explicadas en el Capítulo 6), conseguían que estos brazos robóticos fuesen aprendiendo de manera satisfactoria y correcta.

2.2.2. Creación de una antena de telecomunicaciones mediante Algoritmos Evolutivos

En el año 2006, la *NASA* utilizó un algoritmo evolutivo [5] para la creación de una antena de comunicación; ésta iba a utilizarse en una misión espacial con el nombre de *Space Technology 5 (ST5)*, cuyo objetivo era el de demostrar tecnologías innovadoras de potencial uso para futuras misiones. El objetivo principal de estas antenas era el de la comunicación con las estaciones terrestres desde el espacio.

Estas antenas tienen unos diseños un tanto peculiares e inusuales, pero son los que, dada una serie de variables a maximizar (relacionadas con ancho de banda para escuchar el máximo número de frecuencias posibles), fueron obtenidos por el algoritmo evolutivo (Figura 2.2), cuyo objetivo era maximizar y optimizar estas variables.

Figura 2.2: Antena creada por la *NASA* usando EA.

2.3 Estado del arte en el mundo de los videojuegos

La IA es un concepto que ha estado presente en el mundo de los videojuegos desde sus inicios [6], pues juegos como *Pong*, *Pacman* o *Space Invaders* - Figura 2.3 - ya disponían de sistemas inteligentes para combatir contra el jugador en la década de los 70 y 80. Posteriormente, en la época de los 90, aparecieron juegos como *Starcraft* y *Age of Empires*, donde la IA era capaz de crear tácticas en función de las decisiones que tomaba el jugador y gestionar recursos en función de ello.

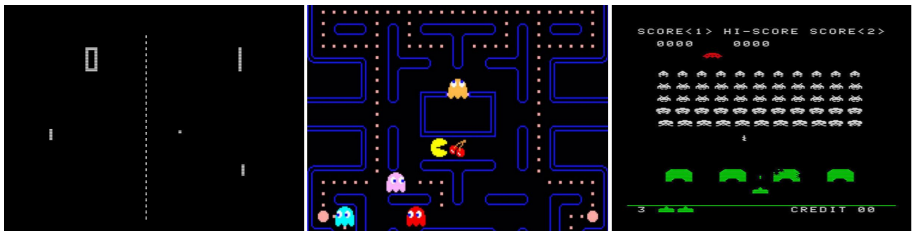

Figura 2.3: Videojuegos *Pong* (izquierda), *Pacman* (medio), *Space Invaders* (derecha).

A medida que han ido trascurriendo los años, los videojuegos han ido mejorando su realismo, gráficos y complejidad, y con ello también la IA que los envuelve, cuya finalidad básica es la de enfrentarse contra el jugador para hacerle mejorar y competir [7].

En este capítulo cabe destacar dos logros mediáticos en los que la IA ha conseguido superar al jugador humano. Los nombres de éstas son *AlphaGo* y *OpenAI Five*, que pasarán a ser descritas a continuación.

2.3.1. *Google DeepMind: AlphaGo*

AlphaGo consiste en un software desarrollado por la compañía de IA inglesa *DeepMind Technologies* cuyo objetivo es jugar al *Go* simulando cómo lo haría un humano; el *Go* es

un juego de tablero de estrategia de origen chino con más de 4000 años de antigüedad. Consiste en una cuadrícula de 19x19 casillas donde los dos jugadores colocan, por turnos, fichas blancas y negras en las intersecciones del tablero (Figura 2.4), correspondiendo el color blanco a un jugador y el negro a otro. El objetivo del juego es controlar más del 50 % del área de la cuadrícula, donde para controlar un área es necesario crear un perímetro usando fichas de un mismo color [8].

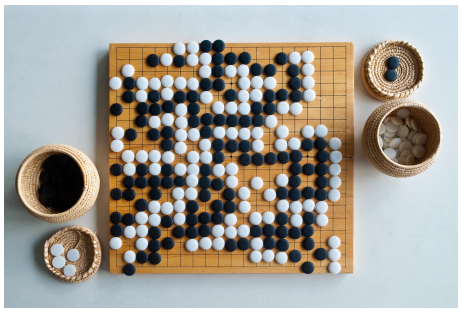

Figura 2.4: Tablero tradicional de *Go*.

A simple vista el juego no parece excesivamente complejo, pues está calificado como un juego de complejidad media. Lo que hace este juego realmente interesante es el nivel de estrategia que implica para realizar jugadas certeras, convirtiéndose así en uno de los juegos de tablero más estratégicos del mundo. Es aquí donde entra en juego *AlphaGo*, software que fue desarrollado alrededor de 2014 para evaluar cómo de bueno era el uso de una *Neural Network* (red neuronal) que utiliza *Deep Learning* [9] (aprendizaje profundo) para competir en este tradicional juego. El algoritmo de *AlphaGo* utiliza una combinación de técnicas de *Deep Learning* y *Tree Search* (búsqueda en árbol) - usando en este caso Monte Carlo *Tree Search* [10] - que se combinan con un gran *dataset* (conjunto de datos) de partidas de humanos y ordenador. Inicialmente, las *Neural Networks* fueron entrenadas mediante la experiencia de alrededor de 30 millones de movimientos de jugadores expertos en *Go*. Una vez alcanzó cierto nivel de habilidad, pasó a desempeñar un gran número de partidas contra sí misma, utilizando esta vez *Reinforcement Learning* [11], donde, a partir de un sistema de *rewards* y *penalties*, mejoró su nivel aún más (Figura 2.5). Esta última técnica es una de las técnicas que es utilizada en este trabajo para hacer aprender a nuestro personaje Mario.

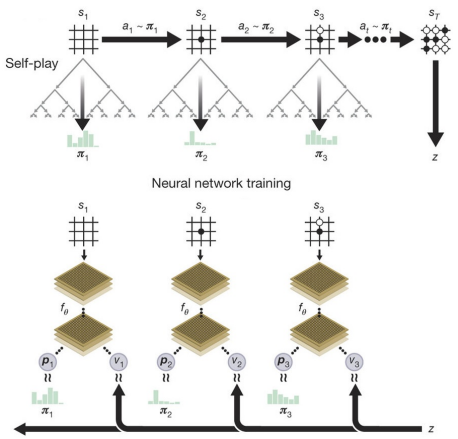

Figura 2.5: Entrenamiento de la *Neural Network* de *AlphaGo* y toma de decisiones jugando contra sí misma.

El momento cumbre de *AlphaGo* fue cuando tras 2 años de trabajo por parte del equipo consiguieron ganar 4 partidas de 5 al mejor jugador del mundo en aquel momento, Lee

Sedol. Esto dejó conmocionado a todo el mundo del *Go* y fue un acontecimiento mediático a nivel mundial, pues la gran mayoría de jugadores dentro de este gremio pensaban que la máquina perdería todas las partidas frente a Lee Sedol.

2.3.2. Diseño de estrucuras: *Car Evolution*

Car Evolution consiste en diseñar la estructura de un coche usando polígonos y ruedas para que sea capaz de circular por un terreno 2D, tal y como se puede observar en la Figura 2.5. Se trata de una aplicación programada en HTML5 en la que, por defecto, se crea una población de 20 individuos (en este caso coches). A medida que se va avanzando en el número de generaciones se cogen aquellos coches que han obtenido mayor distancia recorrida (que hará de *reward function*) de la generación anterior, creando así otros 20 nuevos individuos con un *mutation rate* del 5 % y un *mutation size* del 50 % [12] (conocido como ratio de mutación y tamaño de mutación respectivamente en castellano). Más adelante, en el Capítulo 5, se explicarán a fondo estos conceptos para su fácil entendimiento.

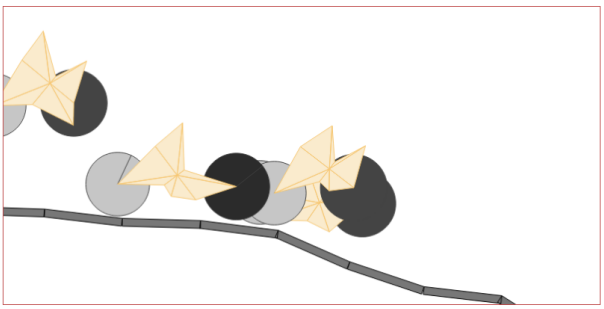

Figura 2.6: Ejemplo ejecución de *Car Evolution*, generación nº 20.

Este es un ejemplo muy representativo y gráfico del gran potencial que ofrecen los algoritmos evolutivos, pues en función de los mejores individuos de cada generación se van haciendo mutaciones genéticas con el genotipo de éstos para crear los mejores individuos posibles.

Además de éste, hay muchas más aplicaciones de los algoritmos evolutivos que se emplean en campos tan importantes y complejos como la medicina [13] y cálculo de estructuras en ingeniería civil [14].

CAPÍTULO 3
Introducción a la IA

Hoy en día son muy recurrentes los términos de IA, *Machine Learning (ML)* o *Deep Learning (DL)*; a su vez, también son términos estrechamente relacionados entre sí y algunas veces hay un poco de confusión entre ellos, usándose de forma indistinta. En este capítulo se van a mostrar las diferencias y características que existen entre estos tres conceptos para posteriormente encasillar las dos técnicas que se han desarrollado para el presente trabajo.

3.1 Inteligencia Artificial en general

En la actualidad no hay una definición única y exacta de la IA, pues es un campo de investigación relativamente nuevo, cambiante y experimental, además de que tampoco se sabe cómo definir de forma exacta qué es la inteligencia humana. De una forma generalista se podría describir como *"La capacidad e intento que tiene un software para imitar la inteligencia humana"* [15]. Partiendo de esto último, dentro de la IA existen los siguientes tipos:

- **IA robusta o *Strong AI***. Es una forma de IA teórica donde el software tiene una capacidad cognitiva similar a la que tiene un humano, desarrollando por tanto la habilidad de resolver problemas, aprender y planificar el futuro. Este tipo de inteligencia aún no se ha conseguido pero se aspira a alcanzarla en unos años [16].

- **IA débil o *Weak AI***. Su función es la de desarrollar una tarea específica a través de ciertos algoritmos y un aprendizaje guiado. Este tipo de IA se basa en la inferencia humana para definir los parámetros de sus algoritmos de aprendizaje y proporcionar los datos de entrenamiento relevantes para garantizar la precisión. Es aquí donde entran en juego los conceptos de *Machine Learning* y *Deep Learning*.

3.2 *Machine Learning*

Como se ha expuesto en la Sección 3.1, el ML se encuentra dentro del grupo de la IA débil. Según el informático teórico Arthur Samuel, en 1959 definió este concepto como *"la habilidad que tiene una máquina de aprender sin estar explícitamente programada"* [17]. El ML explora el estudio y la construcción de algoritmos que sean capaces de aprender y hacer predicciones sobre datos. Para ello, dichos algoritmos identifican ciertos patrones para el conjunto de datos dado. Algunos de estos algoritmos más comunes para discriminar los datos son [18]:

- Regresión lineal (*Lineal Regression*).
- Regresión logística (*Logistic Regression*).
- Árbol de decisiones (*Decision Tree*).
- Máquinas de vector de soporte (*Support Vector Machine*).
- Clasificador bayesiano (*Naive Bayes*).
- K vecinos más cercanos (*k-Nearest Neighbors*).
- K medias (*k-Means*).

Dependiendo del algoritmo que sea utilizado, la discriminación de un conjunto de datos se realizará de distinta manera, pero de una forma generalista se podría decir que la misión y el objetivo de estos algoritmos es subdividir estos datos diferenciándolos y buscando un patrón en ellos, para que a la llegada de nuevos datos pueda clasificarlos dentro de estas subdivisiones ya realizadas. Estos algoritmos son utilizados dependiendo del tipo de ML que se esté usando. Los tipos de ML que existen son los siguientes [15]:

- **Aprendizaje supervisado** o *Supervised Learning*. Los algoritmos basados en este tipo son entrenados usando datos etiquetados, es decir, datos de entrada cuya salida de clasificación se conoce, de forma que, una vez entrenado el modelo con estos datos, se les pasan datos sin etiquetar y, mediante el uso de patrones, se predice el valor de etiqueta para cada uno de los datos sin etiquetar. Es comúnmente utilizado en aplicaciones que predicen eventos futuros.
- **Aprendizaje no supervisado** o *Unsupervised Learning*. En este caso, los algoritmos, a diferencia de lo que pasa en el aprendizaje supervisado, reciben como entrada datos sin etiquetas asociadas, y su finalidad es encontrar una estructura para crear de forma automática las etiquetas, de forma que una vez entrenado el modelo, sea capaz de clasificar nuevos datos que vayan a ser validados, asignándoles estas etiquetas creadas automáticamente. Un ejemplo de aplicación de este tipo sería el de la identificación de segmentos de clientes con gustos similares para campañas de *marketing*.
- **Aprendizaje semi-supervisado** o *Semi-supervised Learning*. Este tipo de ML consiste en una combinación de aprendizaje supervisado y no supervisado para el entrenamiento del modelo; normalmente, se parte de una pequeña cantidad de datos etiquetados y el resto de datos no etiquetados. Es interesante su uso cuando los costes asociados al etiquetado de los datos son muy altos. Este tipo de ML se utiliza, entre otras aplicaciones, para el reconocimiento facial en cámaras web [19].
- **Aprendizaje por refuerzo** o *Reinforcement Learning*. La misión del algoritmo en este tipo de ML es que, a través de prueba y error, descubra qué acciones producen una mayor recompensa o *reward*. Tal y como se expondrá en la Sección 6.1, está formado por tres componentes indispensables: el agente, el entorno y las acciones; el objetivo es que el agente maximice la recompensa en una determinada cantidad de tiempo. Este agente llegará antes a la meta propuesta cuanto mejor sea la política que adopta. Este tipo de ML es usado en campos como la robótica, videojuegos o la navegación [20].

Para la realización del segundo algoritmo de este trabajo, Capítulo 6, se tomará como base la teoría del *RL*, puesto que el algoritmo se va a desarrollar en una condición ideal,

donde Mario será nuestro agente, el nivel a superar será el entorno y las acciones serán los movimientos que puede realizar Mario. Además, contará con una *Neural Network* para el procesamiento de los datos, cuya explicación teórica se realizará en la Sección 6.3.

3.3 *Deep Learning*

Esta sección está estrechamente relacionado con la ya presentada de ML, y es que el DL se considera un subconjunto dentro del ML. Se podría decir que el DL consiste en un conjunto de algoritmos dentro del ML que intenta modelar abstracciones de alto nivel en datos mediante el uso de *Neural Networks (NN)* (redes neuronales en castellano) siempre que éstas contengan más de una *hidden layer*. Éstas están basadas en el funcionamiento biológico del cerebro humano, que está compuesto por interconexión entre neuronas.

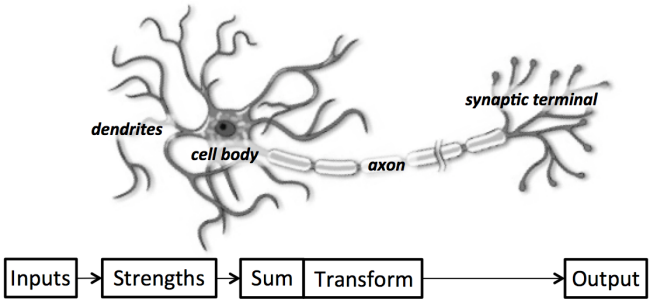

Figura 3.1: Representación de una neurona y sus distintas partes.

En la Figura 3.1 se aprecia una neurona que, como puede verse, está compuesta por varias partes: las dendritas, el cuerpo, el axón y la terminación sináptica. Las dendritas están conectadas con las terminaciones sinápticas de otras neuronas, de forma que se va creando una gran red de conexiones por la que se envían impulsos eléctricos para desarrollar cierta acción. Este concepto fue extrapolado al mundo de la IA dentro de la informática para la creación de NN artificiales, y poder de esta forma procesar datos para realizar ciertas acciones. La estructura que presentan las NN, tal y como se muestra en la Figura 3.2, es la siguiente:

1. ***Input layer*** o capa de entrada, por donde entrarán los datos que hayan sido recopilados. Los nodos de esta capa estarán conectados con la capa oculta mediante conexiones que tendrán cierto peso que posiblemente vaya variando.

2. ***Hidden layer(s)*** o capa(s) oculta(s); aquí los datos pasan a procesarse mediante las distintas conexiones y pesos que haya en las distintas capas ocultas posteriores; la última de estas capas está conectada con la capa de salida.

3. ***Output layer*** o capa de salida, donde, en función del procesamiento de datos que se ha hecho en la capa oculta, se llevará a cabo la activación de cierto número de nodos de la capa de salida, de forma que obtendremos una salida.

Como puede apreciarse, las Figuras 3.1 y 3.2 están estrechamente vinculadas, ya que en el caso de la NN expuesta en la Figura 3.2, las entradas de datos a un nodo consistirían en las distintas dendritas que tiene una neurona, el nodo en sí constituiría el cuerpo y el axón, y finalmente las salidas del nodo vendrían representadas por las terminaciones sinápticas, donde ciertas de éstas se activarían en función de los valores que le lleguen

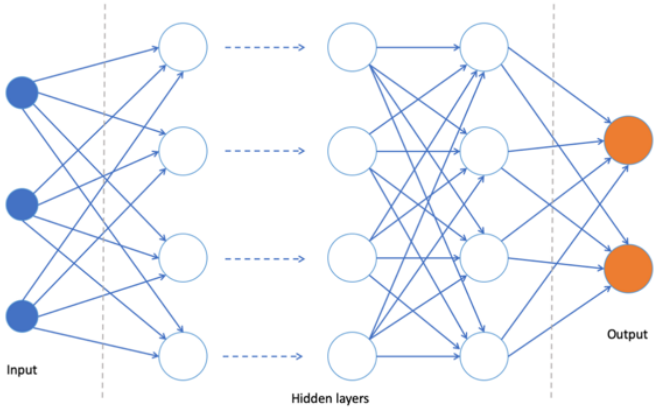

Figura 3.2: Estructura de una *Neural Network*.

de otros nodos. Por tanto, cada uno de los nodos representados sería equiparable a una neurona.

Finalmente se puede ver de una forma gráfica en la Figura 3.3 la estructura de cómo está compuesta la IA, que es de lo que se ha hablado en las Secciones 3.1 a 3.3 de este capítulo.

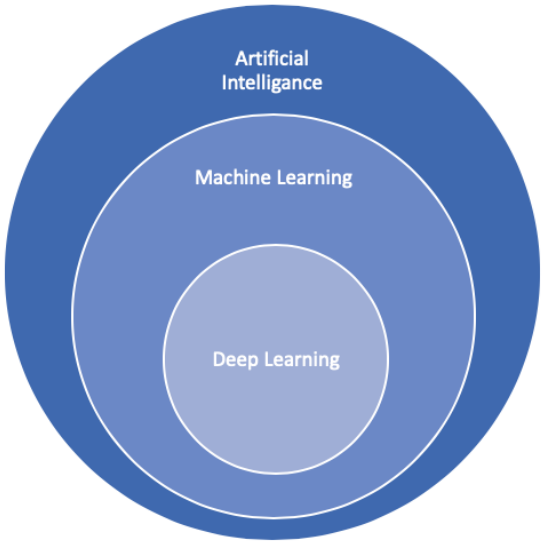

Figura 3.3: Subconjuntos de la IA.

3.4 Algoritmos Evolutivos

Los algoritmos evolutivos o EA pertenecen al campo de la computación evolutiva [21]. Ambos conceptos constituyen una rama de la IA y son utilizados para aquellos problemas en los que los espacios de búsqueda son extensos y no lineales, además de encontrar soluciones en un tiempo razonable. Los EA utilizan mecanismos que se basan en la evolución biológica de las especies, tal y como se muestra en la Figura 3.4. Estos mecanismos son:

1. Reproducción.
2. Mutación.

3. Cruce.

4. Selección.

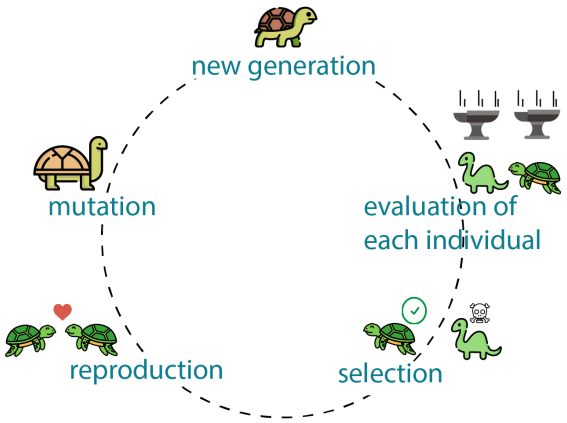

Figura 3.4: Evolución biológica de una especie.

En los EA [22] se mantienen un conjunto de entidades que representan las posibles soluciones; éstas se mezclan entre sí y compiten las unas con las otras de forma que únicamente las más aptas prevalecerán en el tiempo, evolucionando hacia mejores soluciones. Teniendo en cuenta la terminología de la teoría de la evolución, existen los siguientes conceptos que se utilizarán a la hora de programar el algoritmo evolutivo:

- **Individuos** o **cromosomas**: serán las entidades que representan las soluciones al problema.
- **Población**: conjunto de cromosomas.
- **Cruce** de cromosomas: la información asociada a dos o más de éstos será mezclada.
- **Mutación**: un cambio aleatorio en los cromosomas.
- **Selección**: elección de los cromosomas que sobrevivirán para la siguiente **generación**.

Como se puede observar, estos conceptos no son más que un especificación de los mecanismos de la evolución biológica expuestos anteriormente. Dentro de los EA existen tres paradigmas principales que derivan de la idea básica de los EA. Estos paradigmas son:

- **Algoritmos Genéticos (AG)**, donde el conjunto de individuos de una población es codificado como una cadena binaria (cromosoma), que al juntarse con más cromosomas pasa a llamarse genotipo. Los cromosomas irán evolucionando a través de iteraciones (generaciones) donde serán evaluados usando una *fitness function* [23], y aquellos que consigan maximizarla serán los cromosomas que se tendrán en cuenta para cruzarlos y pasar a la siguiente generación.
- **Programación Evolutiva (PE)**, que consiste en una variación de los algoritmos genéticos, donde la representación de cada uno de los cromosomas se hace mediante una terna que representa estados de un autómata finito (estado actual, símbolo, nuevo estado).

- **Estrategias Evolutivas (EE)**, que se caracterizan por el uso de vectores de números reales que codifican las posibles soluciones de problemas numéricos; además, en este paradigma no se genera copia de aquellos individuos con una aptitud por debajo del promedio para las siguientes generaciones.

En el caso del primer algoritmo que se va a realizar en este trabajo, el paradigma que se adoptará es el de Algoritmos Genéticos, el cual, como se verá en la Sección 5.3, también utilizará una NN para el procesamiento de los datos.

CAPÍTULO 4
Biblioteca *gym-retro* de *OpenAI*

OpenAI es una compañía de investigación fundada por el emprendedor Elon Musk en 2015. Se trata de una compañía de IA sin ánimo de lucro, cuyo objetivo es promover y desarrollar IA amigable para que beneficie a la humanidad [24]. Una de las muchas herramientas que ofrece *OpenAI* es *Gym Retro*.

4.1 Qué es *Gym Retro*

Gym Retro consiste en una plataforma [25] para la investigación de RL en videojuegos retro. Actualmente, pone a disposición la biblioteca *gym-retro* para el lenguaje de programación *Python*; en ella se tiene acceso a una gran cantidad de juegos para consolas antiguas (retro) como *Sega Master System, Megadrive, Nintendo Game Boy* o *Super Nintendo Entertainment System*, entre otras. Esta biblioteca permitirá la conexión entre el videojuego y la programación mediante el uso de unas funciones que han sido desarrolladas para tal fin. En el presente trabajo se tendrá como videojuego *Super Mario World*, perteneciente a la consola *Super Nintendo Entertainment System (SNES)*.

4.2 Entorno de desarrollo, *Super Mario World (SNES)*

Super Mario World (SMW) es un juego de plataformas 2D; el objetivo de éste es avanzar por diferentes niveles con el personaje Mario hasta llegar al último para rescatar a la princesa *Peach* venciendo a *Bowser*, que es el enemigo final. Para la realización de este trabajo únicamente se tendrá en cuenta el primero de los niveles, cuyo nombre es *Yoshi's Island 2*. Se ha procedido a dividir este nivel en dos partes:

1. En la primera parte de este nivel, tal y como se muestra en la Figura 4.1, existen una serie de tortugas rojas y verdes como enemigos que se interpondrán en el camino de Mario; éstas pueden ser esquivadas, saltando por encima de ellas, o eliminadas de diversas formas. Al final de esta parte Mario se encontrará con una serie de desniveles que tendrá que ir saltando para progresar en la pantalla; tras ellos le espera un enemigo que consiste en un jugador de rugby que le perseguirá en ambas direcciones.

2. En la segunda parte de este nivel, como puede apreciarse en la Figura 4.2, existe un *checkpoint* donde reaparecerá Mario en caso de que pierda una vida y éstas sean mayores a cero, desde donde podrá continuar con el nivel. Se puede ver, además, que existen una serie de tuberías de color amarillo y violeta que Mario tendrá que sortear para continuar con el nivel; también hay un hueco por el que Mario puede

caerse y perder una vida. En esta parte del nivel como enemigos tenemos a unos topos que saldrán de la tierra y empezarán a seguir a Mario; también habrá una planta piraña dentro de una de las tuberías y, al final del nivel, habrá otro jugador de rugby intentando placar al personaje para que no llegue a la meta.

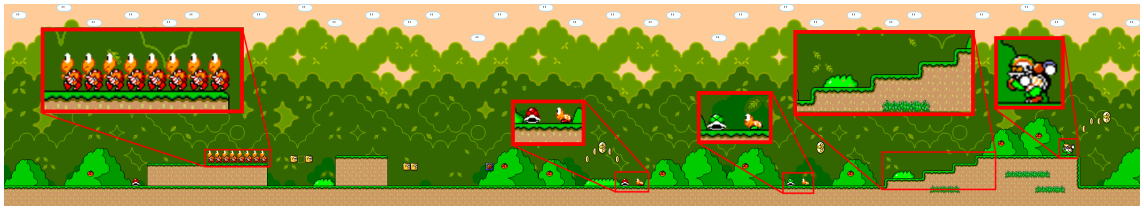

Figura 4.1: Parte 1 del primer nivel de SMW. En los cuadros ampliados pueden verse las tortugas, los desniveles y el jugador de rugby.

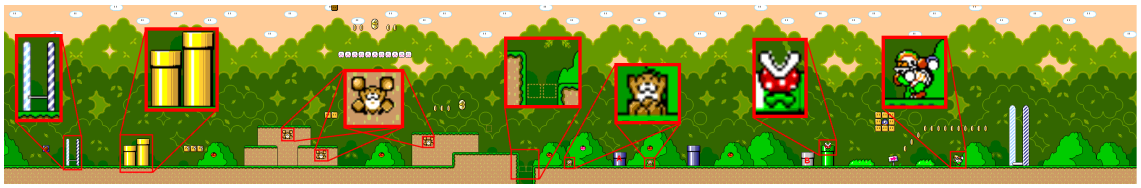

Figura 4.2: Parte 2 del primer nivel de SMW. En los cuadros ampliados pueden verse los topos, las tuberías, el hueco, la planta piraña y el jugador de rugby.

4.3 Cómo funciona *gym-retro*

Como se ha expuesto anteriormente, vamos a utilizar la biblioteca *gym-retro*, que establecerá un flujo de intercambio de información entre las variables del SMW y los algoritmos que van a ser desarrollados en este trabajo, de forma que se va a poder acceder a los datos de las variables del juego para, en función de éstas, poder enviarle a Mario una acción a realizar. Todo ello se realizará gracias a las funciones y variables que nos aporta esta biblioteca, mostradas en el código Listing 4.1.

```
1  import retro
2
3  def main():
4      env = retro.make(game='SuperMarioWorld-Snes',state='YoshiIsland2.state')
5      obs = env.reset()
6
7      print("SNES gamepad buttons -> {}".format(env.buttons))
8      print("Observation space -> {}".format(env.observation_space))
9      print("Action space -> {}".format(env.action_space))
10
11     while True:
12         obs, rew, done, info = env.step(env.action_space.sample())
13
14         if done:
15             obs = env.reset()
16         env.render()
17
18     env.close()
19
20
21 if __name__ == "__main__":
22     main()
```

Listing 4.1: Código básico *gym-retro*.

El código recoge las funciones y variables básicas que se van a utilizar para el desarrollo de los dos algoritmos del trabajo. Como puede observarse en la primera línea de código, se importa la biblioteca *gym-retro*. A continuación van a describirse el funcionamiento de estas funciones y variables:

- ***retro.make(...)***: crea el entorno de trabajo para el juego que se le pasa por el primer parámetro *game*; además, se le pasará también el nivel en el que se desea jugar. En el caso del desarrollo de este trabajo, le pasaremos como primer parámetro el valor *game='SuperMarioWorld-Snes'* y como segundo parámetro el valor *state='YoshiIsland2.state'*. Aparte de estos dos parámetros se le pueden pasar otros como uno para usar un número restringido de acciones, un booleano para iniciar la grabación de la partida, el número de jugadores y el tipo de observación que se desea; estos parámetros tienen un valor por defecto por si no se les pasa como parámetro a la función. Normalmente, el valor de esta función lo dejaremos almacenado en la variable *env*.

- ***env.reset()***: reinicia el entorno para que se asegure de que se empieza la partida justamente desde el principio. Además, devolverá la observación inicial, donde ésta dependerá del tipo de observación cuando se crea el entorno:

  - 0, si se están usando observación mediante imágenes RGB.
  - 1, si se usan observaciones para ver la memoria RAM del juego.

- ***env.step(...)***: a esta función se le pasará como parámetro una acción a realizar; esta acción está compuesta por un vector de doce valores binarios, que representará tanto los botones que están activados en ese momento (con valor a '1') como los que están desactivados (con valor a '0') del mando. Por tanto, el cometido de esta función es ir pasando una acción para cada *frame* de juego. Esta función nos devolverá también los siguientes parámetros:

  - ***obs***, que contendrá la nueva observación tras haber realizado la acción anterior en el entorno.
  - ***rew***, se trata del valor que tendrá asociado Mario como función de recompensa; más adelante en el trabajo se expondrá cómo se utiliza esta variable para el aprendizaje en ambos algoritmos.
  - ***done***, variable booleana que indica si la partida ha finalizado, es decir, cuando Mario pierde una vida o se ha llegado al límite de tiempo en el nivel.
  - ***info***, se trata de un diccionario que devuelve los valores que tiene asociado en un fichero con extensión *.json*, el cual hace referencia a algunas direcciones de RAM del juego; estas direcciones van asociadas a variables (como la cantidad de vidas restantes, la posición de Mario, el número de monedas...) que, como es de esperar, irán cambiando a medida que Mario va jugando. Esta variable es de vital importancia en ambas estrategias de aprendizaje, como se verá cuando se expliquen su desarrollo.

- ***env.render()***: su funcionalidad es la de mostrar en una ventana la imagen que estaría viendo el jugador si estuviera jugando al SMW; esto resultará interesante para ver cómo va aprendiendo Mario a jugar.

- ***env.observation_space***: esta variable muestra el espacio de observación en el que trabaja el juego, es decir, la resolución del juego y los canales en los que funciona. En el caso de SMW, al imprimir esta variable por pantalla se imprime:

```
Observation space -> Box(0, 255, (224, 256, 3), uint8)
```

  Como se puede apreciar, devuelve un objeto tipo *Box*, del cual el argumento más importante es el tercero, donde muestra que la resolución del juego es 256x224 píxeles y se representa en los tres canales de color RGB; esto último es bastante importante para cuando se vaya a procesar los datos en la NN de sendos algoritmos.

- ***env.action_space***: esta variable muestra el tipo de variable que tiene que pasarse como variable de acción a la función *step(...)*; cuando se muestra por pantalla el valor de esta variable se obtiene:

```
Action space -> MultiBinary(12)
```

  que, como se había mencionado antes, se trata de un vector binario de 12 elementos, que son los botones de que dispone el mando de la SNES.

- ***env.buttons***: esta variable es importante puesto que muestra cómo están mapeados los botones del mando en un vector de 12 elementos, lo cual ayudará a saber la posición de cada botón en el vector que se le pase a la función *step(...)*. Cuando se muestra por pantalla el valor de esta variable se obtiene:

```
SNES gamepad buttons -> ['B', 'Y', 'SELECT', 'START', 'UP', 'DOWN', 'LEFT
    ', 'RIGHT', 'A', 'X', 'L', 'R']
```

  Ahora, una vez ya se sabe dónde está mapeado cada uno de los botones, tal y como se ve en la Figura 4.3, se pasa a ver la relación que hay entre estos botones y las distintas acciones que puede realizar Mario dentro del juego.

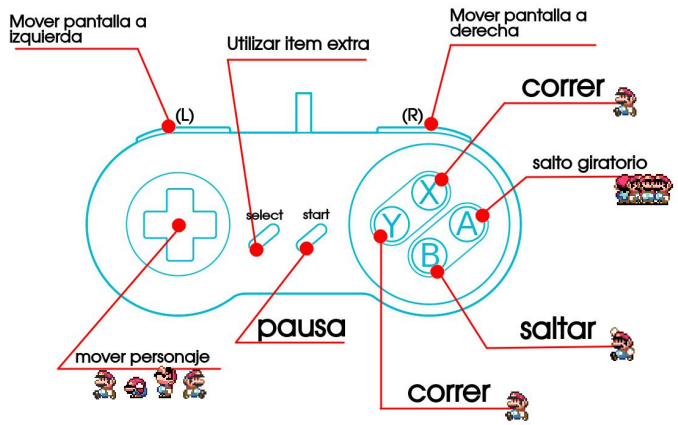

Figura 4.3: Acciones vinculadas con los botones del mando de SNES.

4.4 Herramienta de integración

La biblioteca *gym-retro* trae consigo una herramienta para encontrar de manera sencilla las variables del juego y ver el valor de éstas [26]. Para ello, a través de la herramienta se puede filtrar en función del nombre de la variable, la cual tendrá una dirección de

memoria RAM asociada. El entorno es tal y como se muestra en la Figura 4.4. Un vez se haya obtenido la dirección de memoria RAM de la variable, se procede a representarla en un fichero *JSON* cuyo nombre es *data.json*, el cual se encontrará dentro de la carpeta del videojuego con el que se esté trabajando, que a su vez estará en la carpeta de instalación de la biblioteca. El fichero *data.json* asociado de forma predeterminada al juego SMW es mostrado en el Listing 4.2.

```
{
  "info": {
    "coins": {
      "address": 8261055,
      "type": "|u1"
    },
    "lives": {
      "address": 8261054,
      "type": "|i1"
    },
    "score": {
        "address": 8261428,
        "type": "<u4"
    }
  }
}
```

Listing 4.2: Contenido inicial de *data.json*.

Se puede observar como dentro del fichero *JSON* existe el atributo *info*, el cual está formado a su vez por otros tres atributos, *coins*, *lives* y *score*, los cuales tienen como propiedades la dirección de memoria RAM donde se encuentran y el tipo de variable que son. El punto interesante de este fichero es que en el momento en el que se haga la llamada a la función *step(...)*, uno de los parámetros que nos devuelve es un diccionario que contiene el valor de estas variables; esto resulta de gran interés, puesto que cuando se explique cómo se ha llevado a cabo la programación de los algoritmos en el Capítulo 5, se verá cómo gracias a la herramienta que se expone en este punto y al uso del fichero *JSON* se mejora tanto la *fitness function* en EA como el *reward* en RL.

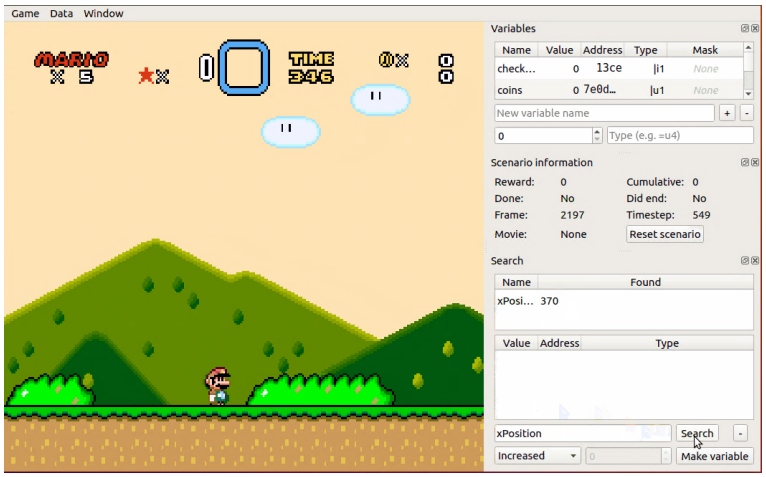

Figura 4.4: Herramienta de integración de *gym-retro*.

CAPÍTULO 5

Desarrollo del algoritmo basado en Algoritmos Genéticos

Tal y como se explicó en la Sección 3.4 de este trabajo, el primer algoritmo a desarrollar está basado en EA, concretamente AG. A lo largo de este capítulo se mostrará cómo funcionan los AG, se explicará la biblioteca *NEAT-Python* utilizada para la implementación de éstos, cómo se ha llevado a cabo el desarrollo y la explicación del código y, finalmente, se mostrarán los resultados obtenidos.

5.1 Algoritmos Genéticos

Los Algortimos Genéticos consisten en una técnica de programación cuyas bases se establecen gracias a la evolución biológica. Esta técnica consiste en una heurística inspirada en la teoría de la selección natural expuesta por Charles Darwin en el año 1859 [27]. El proceso de selección natural empieza con la selección del individuos más aptos de una población; tras ello, éstos producen descendientes que heredan las características de los padres y pasan a formar parte de la siguiente generación. Los hijos serán mejor que los padres y tendrán más probabilidades de sobrevivir; este proceso se realiza iterativamente hasta encontrar una generación que sea lo más apta posible.

Tal y como se explicó en la Sección 3.4, existen una serie de mecanismos en los que se basa un EA; en los AG se vuelve a recurrir a ellos pero esta vez añadiendo una serie de conceptos nuevos como son la población inicial y la *fitness function*; éstos se pasarán a explicar dentro de las cinco etapas que son consideradas dentro de los AG. Estas etapas son las siguientes:

1. **Población inicial**. El proceso comienza con un conjunto de individuos; éstos son la población inicial, donde cada individuo es una solución del problema a resolver. Cada individuo se caracteriza por estar formado por un conjunto de variables booleanas conocidas como **genes**, y al conjunto se le conoce como **cromosoma**, tal y como se ve reflejado en la Figura 5.1.

2. ***Fitness function***. Se trata de un atributo que tiene cada uno de los individuos de la población; éste sirve para cuantificar numéricamente lo apto que es un individuo. La probabilidad de que un individuo sea escogido para la reproducción se basa en este atributo, por lo que esta etapa es de suma importancia.

3. **Selección**. Tras obtener la *fitness function* de cada uno de los individuos, se procede a seleccionar aquellos cuyo valor sea mayor y se deja que sus genes pasen a la

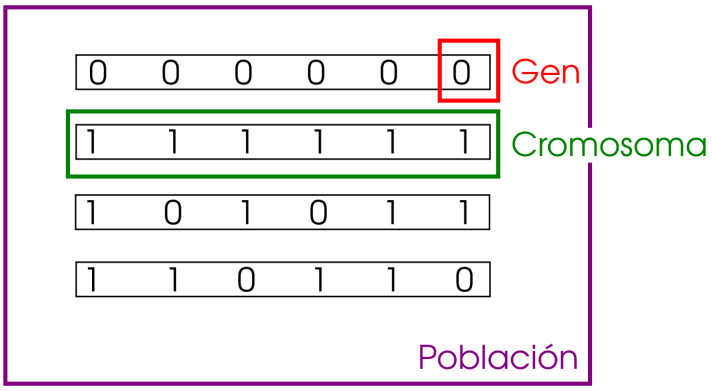

Figura 5.1: Población inicial, genes y cromosomas.

siguiente generación. En este caso se está aplicando una *elitist selection* [28] o selección elitista, pero cabe destacar que existen más criterios a seguir para el proceso de selección, aunque el que se ha expuesto en este punto es el más habitual.

4. **Cruce o *crossover***. En esta etapa, para cada par de padres que han sido seleccionados previamente se elige un **punto de cruce o *crossover point*** de forma aleatoria dentro de los genes; tras esto tiene lugar la descendencia u *offspring*, en la que se intercambian genes de los padres entre ellos hasta llegar al punto de cruce (Figura 5.2).

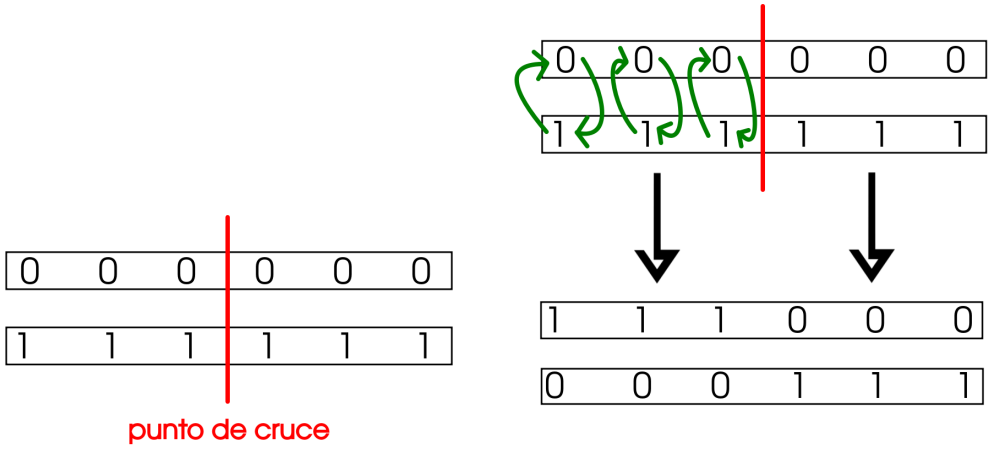

Figura 5.2: Elección del punto de cruce y creación de nueva descendencia.

5. **Mutación**. En ciertas descendencias creadas, alguno de sus genes pueden sufrir una mutación con una probabilidad baja, por lo que algunos de sus valores son invertidos (Figura 5.3); con esta etapa se previene de que se converja de forma prematura.

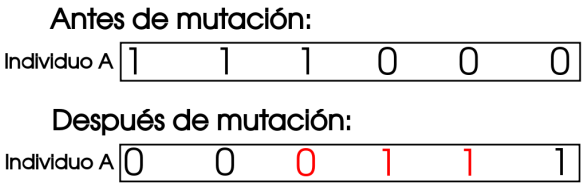

Figura 5.3: Mutación de un individuo.

Las etapas descritas se irán haciendo de forma iterativa hasta que la población converja, es decir, que no se produzca un cambio significativo entre la descendencia y la generación anterior, momento en el que el algoritmo finalizará. Este proceso se ve reflejado en el Algoritmo 5.1.

Algorithm 5.1 Pseudocódigo de los AG

```
start
Generar poblacion inicial
Computar fitness function
while not converge población do
    Selección individuos más aptos
    Cruce individuos seleccionados
    Mutación
    Computar fitness function
end while=0
```

5.2 Biblioteca *NEAT-Python*

NEAT es un acrónimo de *NeuroEvolution of Augmenting Topologies*, y fue desarrollado por Kenneth Standley Owen y Risto Miikkulainen en la Universidad de Texas [29]. Se trata de una codificación basada en nodos para describir la estructura de una red y los pesos asociados a las conexiones entre ellos. Además, tiene una forma verdaderamente ingeniosa para resolver el problema de *Competing Conventions* [30] utilizando el histórico de datos generado cuando se crean nuevos nodos y conexiones. *NEAT* también intenta mantener el tamaño de las redes al mínimo; para ello comienza la evolución utilizando una población de redes de topología mínima, y a partir de ahí va agregando neuronas y conexiones a lo largo de la ejecución (Figura 5.4). La complejidad de todo esto será transparente al desarrollo de este algoritmo para el trabajo, pues la librería hace por sí sola todo lo anteriormente descrito con el uso de las funciones que nos ofrece.

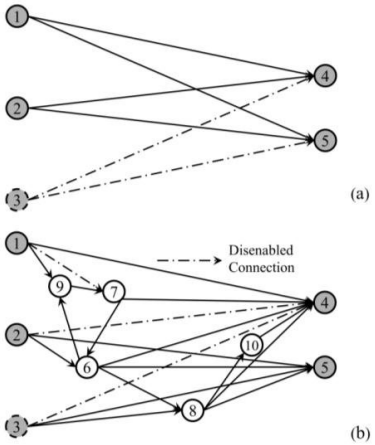

Figura 5.4: Evolución de una red.

Una parte muy importante de esta biblioteca es el archivo de configuración *config-feedforward*; en él se tienen que especificar las condiciones iniciales que serán tenidas en cuenta para la ejecución del algoritmo basado en AG que se desarrolla en el presente trabajo. Se va a proceder a destacar los puntos más importantes de este archivo (Listing 5.1), el cual está al completo en la Sección A.2 del Apéndice A.

```
[NEAT]
fitness_criterion     = max
fitness_threshold     = 100000
pop_size              = 50
reset_on_extinction   = True
```

Listing 5.1: Configuración propiedades generales de *NEAT*.

Como puede observarse se muestran las variables:

- ***fitness_criterion***, la cual indica el criterio que se va a seguir cuando se evalúe la *fitness function*; este criterio puede ser *max* cuando se busque maximizar la *fitness function* o el valor *min* cuando se busque minimizarla. En este caso convendrá maximizarla, pues el personaje Mario irá consiguiendo puntos a medida que vaya tomando ciertas acciones en el juego, los cuales serán parte de la *fitness function*.
- ***fitness_threshold***, que hace referencia al momento en el que se considera que ha convergido el algoritmo; en el caso de este trabajo se ha decido escoger que el valor sea 100000, que se interpretará como infinito puesto que nunca habrá un cromosoma que llegue a tener ese valor en la *fitness function*. Esta decisión ha sido tomada para ver cómo va convergiendo la puntuación de los cromosomas en cada generación y así facilitar el estudio del algoritmo.
- ***pop_size***, que muestra cual es el número de cromosomas de la población en cada generación. En el código se muestra que toma el valor de 50, pero más adelante, en la Sección 5.4, se verá que también tomará los valores 10, 30 y 100.
- ***reset_on_extinction***, que puede tomar el valor de *True*, lo que hará que en el caso de que se extingan todas las especies al mismo tiempo, se cree un nueva población aleatoria. Si toma el valor de *False*, se lanzará una *CompleteExtinctionException*.

```
[DefaultGenome]
# node activation options
activation_default      = sigmoid
activation_mutate_rate  = 0.05
activation_options      = sigmoid gauss
...
# network parameters
num_hidden              = 0
num_inputs              = 896
num_outputs             = 12
...
# node response options
response_mutate_rate    = 0.75
response_replace_rate   = 0.1
...
feed_forward            = False
```

Listing 5.2: Configuración propiedades *DefaultGenome* de *NEAT*.

En el Listing 5.2 se muestran las variables vinculadas al genoma, que consiste en la combinación única de genes que equivale a un cromosoma. Estas variables son:

- ***activation_default***, que indica el tipo de activación por defecto que van a tener los nodos de la NN; en este caso se optado por el uso de la función sigmoide.
- ***activation_mutate_rate***, que indica la probabilidad de que la mutación reemplace la función del nodo con un miembro determinado al azar de las opciones de activación. Se ha decidido que tome el valor de 0.05.

- ***activation_options***, que refleja las funciones de activación que pueden utilizar los nodos; en este caso podrán utilizar la función sigmoide (por defecto) y la función gaussiana.

- ***num_hidden***, que representa el número de nodos en la capa oculta *(hidden layers)* de la NN de la población inicial; se ha decidido que tome el valor 0, por lo que en la población inicial no contaremos con esta capa.

- ***num_inputs***, la cual designa el número de nodos de la capa de entrada *(input layer)* de la NN; donde en el caso del presente trabajo es de 896 nodos, que representa el número de píxeles que muestra el videojuego SMW tras tratar la imagen original. Este tratamiento que se le realiza a la imagen se explica en la Sección 5.3.

- ***num_outputs***, que consiste en el número de nodos en la capa de salida *(output layer)* de la NN. En este caso serán 12 nodos que corresponden con los 12 botones que tiene el mando de la SNES, tal y como se explicó al final de la Sección 4.3.

- ***response_mutate_rate***, que muestra la probabilidad de que la mutación cambie el multiplicador de respuesta de un nodo añadiendo un valor aleatorio; será 0.75 en este caso.

- ***response_replace_rate***, la cual define la probabilidad de que la mutación reemplace al multiplicador de respuesta de un nodo con un valor aleatorio recién elegido como si fuera un nodo nuevo, tomando en este caso el valor de 0.1.

```
1 [DefaultReproduction]
2 elitism            = 1
3 survival_threshold = 0.3
4
5 [DefaultStagnation]
6 species_fitness_func = max
7 max_stagnation       = 20
```

Listing 5.3: Configuración propiedades de *DefaultReproduction* y *DefaultStagnation* de *NEAT*.

Las variables mostradas en el Listing 5.3 hacen referencia a la reproducción y al estancamiento de los cromosomas, respectivamente. Más en detalle:

- ***elitism*** representa el número de individuos (cromosomas) más aptos (que tienen mayor *fitness function*) de cada generación que se conservarán de forma íntegra para la siguiente generación; se ha optado por que tome el valor 1, de forma que en cada generación el mejor cromosoma pasará para estar en la siguiente generación.

- ***survival_threshold_rate***, es el porcentaje que tiene cada especie para permitir reproducirse en cada generación; en este caso tomará el valor 0.3.

- ***species_fitness_func*** es la función utilizada para calcular la aptitud de las especies, es decir, para calcular la *fitness function*, que como ya se había mostrado antes será la función de maximizar.

- ***max_stagnation*** representa el número máximo de generaciones en las que un cromosoma puede permanecer sin mejoras antes de ser eliminado; se ha decidido que sea de 20.

5.3 Implementación del algoritmo basado en Algoritmos Genéticos

En esta sección se va a exponer cómo ha sido desarrollado el algoritmo utilizando la técnica de los AG vista en la Sección 5.1. Puesto que la biblioteca *gym-retro* está disponible únicamente para el lenguaje de programación *Python3*, y además la biblioteca NEAT también está disponible en *Python3*, el algoritmo, por ende, se ha programado también en *Python3*.

Antes de mostrar el código y su implementación es importante resaltar cómo funciona la *fitness function* de Mario, y es que éste consigue que la *fitness function* se vaya incrementando en función de ciertas acciones que tome en el mapa, como conseguir monedas, debilitar a una tortuga (se puede incrementar un multiplicador dependiendo de cómo y cuántas debilites), rompiendo bloques, debilitando a los topos enemigos, cogiendo medallas Yoshi (parecidas a las monedas normales pero con más valor), golpeando y debilitando al jugador de rugby y, finalmente, consiguiendo llegar al final del nivel. Con todas estas acciones sería suficiente para que el algoritmo comenzase a ejecutar y que Mario fuera aprendiendo a jugar, pero se quiso ir un poco más allá. Para optimizar el tiempo de aprendizaje se optó por utilizar la herramienta de integración de *Gym Retro* (explicada en la Sección 4.4) para obtener el valor de la coordenada *x* de Mario y además el evento que se genera cuando se termina el nivel, quedando por lo tanto el archivo *data.json* como se ve en el Listing 5.4.

```
{
  "info": {
    "coins": {
      "address": 8261055,
      "type": "|u1"
    },
    "lives": {
      "address": 8261054,
      "type": "|i1"
    },
    "x": {
      "address": 148,
      "type": "<u2"
    },
    "dead": {
      "address": 8257688,
      "type": "<u4"
    },
    "endOfLevel": {
      "address": 8259846,
      "type": "|i1"
    },
    "score": {
      "address": 8261428,
      "type": "<u4"
    }
  }
}
```

Listing 5.4: Fichero *data.json* actualizado con las nuevas variables.

Se han añadido por lo tanto las variables *x*, *dead* y *endOfLevel* asociadas con su localización en la RAM, por lo que ahora se podrá acceder a ellas en tiempo de ejecución, tal y como se verá a lo largo de esta sección.

Tras haber mostrado lo anterior, el algoritmo comienza con la inicialización del entorno de trabajo, un *array* y la creación de dos ventanas auxiliares tal y como se muestra en el Listing 5.5.

Como puede observarse en el código del Listing 5.5, se hace la llamada a la función *make* que nos ofrece *gym-retro* y le pasamos como argumentos el nombre del juego *(Super Mario World)* y el nombre del nivel *(YoshiIsland2.state)*. Tras ello se crea un *array* donde se almacenará la imagen cuando se le realice el tratamiento, e inmediatamente después se crean dos ventanas auxiliares gracias a la función *namedWindow(...)* que ofrece la biblioteca *OpenCV* [31] (cv2 en Python), que se utiliza para el campo de la visión por computador.

```
env = retro.make('SuperMarioWorld-Snes', 'YoshiIsland2.state')
imageArray = []

cv2.namedWindow('ResizeRGB', cv2.WINDOW_NORMAL)
cv2.namedWindow('ResizeGrayScale', cv2.WINDOW_NORMAL)
```

Listing 5.5: Inicio de la implementación del algoritmo.

Después de haber hecho las inicializaciones anteriores, se procede a la creación e inicialización de las variables que tienen relación con *NEAT-Python*, tal y como se muestra en el Listing 5.6.

```
config = neat.Config(neat.DefaultGenome, neat.DefaultReproduction,
                     neat.DefaultSpeciesSet, neat.DefaultStagnation,
                     'config-feedforward')

p = neat.Population(config)
p.add_reporter(neat.StdOutReporter(True))
stats = neat.StatisticsReporter()
p.add_reporter(stats)

start_time = time.time()
winner = p.run(eval_genomes)
finish_time = time.time()
```

Listing 5.6: Inicialización de variables de *NEAT*.

Como puede observarse, se hacen una serie de llamadas a los constructores de clase y a sus métodos que nos ofrece *NEAT*. Lo primero que se realiza es la llamada es al constructor de la clase *Config*, donde se le pasa como parámetros *neat.DefaultGenome*, *neat.DefaultReproduction*, *neat.DefaultSpeciesSet* y *neat.DefaultStagnation*; el valor de cada una de estas variables está dentro del fichero de configuración *config-feedforward* (último parámetro) cuya estructura y contenido se explicó en la Sección 5.2. Tras ello se almacena toda la configuración dentro de la variable *config*. Una vez hecho esto, se hace la llamada al constructor de la clase *Population*, la cual es la encargada de evaluar la *fitness function* de los cromosomas, verificar el criterio de finalización, generar las siguientes generaciones y dividir la nueva generación en especies dependiendo de su similitud genética; se le pasa como parámetro la variable *config* que contiene todos los parámetros necesarios; este objeto que se ha creado queda almacenado en la variable *p*. A continuación, a esta última variable se le añaden dos *reporters*, los cuales mostrarán por la salida estándar (el terminal) el progreso de la evolución, todo gracias a la llamada del método *add_reporter()*. Finalmente, existen dos variables que van a manejar el tiempo de ejecución (*start_time*, *finish_time*) y la llamada al método *run(eval_genomes)*, el cual se encargará de poner en marcha la evolución mediante el *callback eval_genomes*. Esta función es la encargada de la evolución de los genomas, que se muestra por partes en el Listing 5.7.

Como puede apreciarse en el Listing 5.7, la función toma como parámetro los genomas y la configuración que había sido almacenada en la variable *p*. Comenzará con un

bucle *for* que irá recorriendo uno a uno los genomas; dentro de él se reseteará el entorno *env*. Tras ello se obtendrá la resolución de pantalla del videojuego, así como el número de canales en el que trabaja gracias a la variable de entorno *env.observation_space.shape*.

```
def eval_genomes(genomes, config):

    for genome_id, genome in genomes:
        obs = env.reset()

        inputY, inputX, numChannels = env.observation_space.shape
        inputX = int(inputX/8)
        inputY = int(inputY/8)

        net = neat.nn.recurrent.RecurrentNetwork.create(genome, config)

        fitnessMax = 0
        fitnessCurrent = 0
        frameCounter = 0
        xPos = 0
        xPosMax = 0

        done = False

        while not done:

            env.render()

            obs = cv2.resize(obs, (inputX, inputY))
            cv2.imshow('ResizeRGB',obs)
            obs = cv2.cvtColor(obs, cv2.COLOR_BGR2GRAY)
            cv2.imshow('ResizeGrayScale',obs)

            imageArray = np.ndarray.flatten(obs)

            nnOutput = net.activate(imageArray)

            obs, rew, done, info = env.step(nnOutput)

            xPos = info['x']

            if xPos > xPosMax:
                xPosMax = xPos
                frameCounter = 0
                fitnessCurrent += 1
            else:
                frameCounter += 1

            if info['endOfLevel']:
                fitnessCurrent += 10000
                done = True

            fitnessCurrent += rew

            if done or frameCounter == 100:
                done = True
                print(genome_id, fitnessCurrent)

            genome.fitness = fitnessCurrent
            cv2.waitKey(1)
```

Listing 5.7: Función *eval_genomes*.

Tal y como se mostró en la Sección 4.3, la resolución a la que funciona el videojuego es de 256x224 píxeles en los tres canales de color RGB, por lo que tendríamos 256x224x3 =

172032 elementos con valores que oscilan en el intervalo [0, 255] . Se procede a normalizar la resolución dividiendo tanto la anchura como la altura entre 8, obteniendo una nueva resolución de 32x28 píxeles, que utilizando los 3 canales de color RGB serían 2688 elementos; estos dos nuevos valores de resolución son introducidos en las variables *inputX* e *inputY*, respectivamente. Tras ello se procede a crear una NN recurrente para ese genoma, que, tal y como se explicó en la Sección 5.2, tendrá 12 nodos en la capa de salida y 896 nodos en la capa de entrada; este número proviene del resultado de la resolución normalizada 32x28=896, trabajando únicamente en un canal de color. En lo que a capas ocultas y sus nodos respecta, éstos se irán creando en función de la evolución que vaya sufriendo la red a lo largo de los distintos cromosomas en las distintas generaciones.

Continuando con la función *eval_genomes*, entra dentro de un bucle *while* en el que se estará mientras la variable *done* tome el valor *False*. En este bucle, lo primero que se ejecuta es la función *render()*, que, tal y como se explica en la Sección 4.3, hará que se renderice el juego y se muestren por pantalla las acciones que realiza Mario. Tras ello, se pasa al tratamiento de la imagen (Figura 5.5), donde con la ayuda de la biblioteca *OpenCV* se hace un reescalado de la misma para pasar de la resolución original a la normalizada mediante la función *resize*, pasándole como primer parámetro el *frame* actual que se está renderizando (variable *obs*) y como segundo parámetro una tupla con los valores de la nueva resolución (*inputX, inputY*). Inmediatamente después se muestra esta transformación en una de las dos ventanas auxiliares que se crearon en el Listing 5.5. Tras esta primera transformación de la imagen se va a realizar otra más sobre esta última, en la que se va a pasar de los 3 canales de colores que se tenían a un único canal de color gris, gracias a la función *cvtColor*, que toma como primer parámetro la imagen con la primera transformación y como segundo parámetro una variable global de entorno de *OpenCV*, la cual hace referencia al cambio de color RGB a gris (*COLOR_BGR2GRAY*). Finalmente, esta segunda transformación se almacena en la variable *obs* y se muestra en otra de las ventanas auxiliares. Tras el tratamiento de imagen se procede a convertir la matriz de

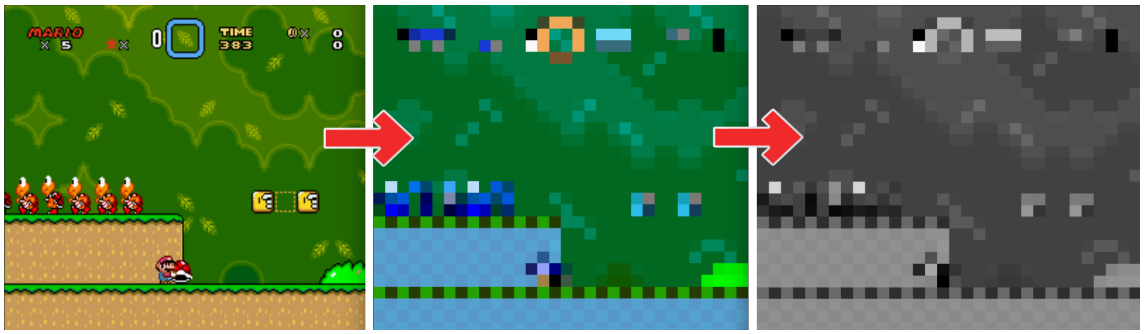

Figura 5.5: Imagen original y sus tranformaciones.

la imagen en un *array* de una única dimensióm mediante la función *flatten* que ofrece la biblioteca *numpy*. A pesar de no haber sido comentado anteriormente, se ha optado por tranformar la imagen para que a la hora de introducirla a la NN minimicemos los nodos de la capa de entrada para no realizar un gran cantidad de cálculos de pesos entre los nodos de otras capas y así mejorar la eficiencia y el tiempo de entrenamiento. Con el proceso descrito pasamos de unos supuestos 172032 nodos de entrada a 896, siendo 192 veces menor. Se tendrá almacenada pues, en la variable *imageArray* un *array* con la información de cada uno de los píxeles de la imagen tratada, por lo que tendrá una longitud de 896 elementos. Un vez ya se tiene la información anterior almacenada, se procede a introducirla dentro de la NN mediante el uso de la función *activate* de la red. Esto nos devolverá como información de la capa de salida un *array* de 12 elementos correspondientes a los botones del mando de la SNES, que en función del valor de los pesos de los nodos y de la función de activación de éstos, habrán activado o no ciertos botones. Este

array de salida de la NN se almacena en la variable *nnOutput*, que contiene la acción que va a realizar Mario. Finalmente, se hace una llamada a la función *step* de *gym-retro*, a la cual, como se explicó en la Sección 4.3, se le pasará la acción que va a realizar Mario para que la ejecute, por lo que se le pasa la variable *nnOutput* como parámetro. Esta última función devuelve además las variables *obs*, *rew*, *done* e *info*, que son también explicadas en la Sección 4.3. Continuando con el Listing 5.7, se muestra la parte final de la función. Se procede a obtener la posición del eje "x"de coordenadas gracias al diccionario *info*, que como se explica en la Sección 4.3, contiene los valores de las variables que están reflejadas en el fichero *data.json*. Tras ello se procede a comprobar, mediante una sentencia *if/else*, si la posición actual en el eje *x* (almacenada en la variable *xPos*) es mayor que la máxima que se haya tomado hasta el momento (almacenada en *xPosMax*) donde en caso de ser:

- **Cierto**, se procede a sobreescribir el valor de la variable *xPosMax* con el nuevo valor de la coordenada "x", se resetea la variable *frameCounter* a 0 y se aumenta la *fitness function* en uno.
- **Falso**, se aumenta la variable *frameCounter* en uno.

Es aquí donde toma el protagonismo el uso del valor del eje "x"de coordenadas, ya que en el caso de que Mario vaya aumentando su posición en este eje (esto designa moverse hacia la derecha en el mapa) se va aumentando el valor de la *fitness function*. Además, en caso de que Mario no aumente su posición en este eje, empezará a incrementarse la variable *frameCounter*, cuya finalidad se mostrará en unos instantes.

Tras lo anteriormente expuesto, lo siguiente que se pasa a comprobar es si Mario ha llegado al final del nivel; esto se comprueba, al igual que el valor de la posición "x", en el diccionario *info*, que devolverá un valor booleano. En caso de que sea cierto, se incrementa considerablemente la *fitness function* y la variable *done* tomará el valor *True*. Gracias a este incremento tan grande en la variable *fitnessCurrent*, se potenciará la reproducción de estos cromosomas, puesto que el criterio que se toma para la selección de éstos es el de maximizar el valor de la *fitness function* (*fitness_criterion* en el Listing 5.1). Inmediatamente después de lo anteriormente expuesto, se incrementa a la variable *fitnessCurrent* con el valor de *rew*, que, como se explica en la Sección 4.3, devuelve la puntuación que ha conseguido Mario en esa iteración del bucle *while* (consiguiendo monedas, rompiendo algún bloque, debilitando a algun enemigo...).

A continuación está la última sentencia *if*, que comprueba si Mario ha terminado (ya sea pasándose el nivel o siendo debilitado por un enemigo o por haberse pasado el nivel) consultando el valor de la variable *done*, o de si la variable *frameCounter* ha tomado el valor de 100. En caso de que se cumpla una condición u otra se sobreescribirá el valor de *done* (por si se entra mediante las segunda condición) e imprimirá por pantalla el identificador del cromosoma y el valor de la *fitness function* que ha conseguido. El objetivo que tiene la variable *frameCounter* es que en el momento en el que Mario no avance hacia la derecha se irá incrementando este contador, donde en caso de que llegue a 100 se dará por terminada la vida de ese genoma y se dará paso al siguiente. De esta forma se evitará que Mario se quede estancado sin avanzar en el nivel perdiendo así tiempo en el entrenamiento. Finalmente se guardará el valor de la *fitness function* para ese genoma y se volverá a comprobar la condición del *while*.

5.4 Análisis de los resultados obtenidos

Antes de poner en marcha la ejecución del código visto en la sección anterior, se pensó en el tipo de estadísticas que debían obtenerse para realizar el estudio de este algoritmo.

Tal y como se expuso en la Sección 5.2, se va a realizar la ejecución con diferentes tamaños de población; éstos van a ser de 10, 30, 50 y 100 cromosomas. En los diferentes tamaños de poblaciones siempre se tendrá el mismo número de generaciones, en este caso se ha optado por realizar 70 generaciones, donde para cada una de ellas se obtendrá el valor de las siguientes variables:

- **Valor medio de la *fitness function*.**
- **Valor máximo de la *fitness function*.**
- **Tiempo de ejecución.**
- **Número de veces que se llega a la meta.**

Tras poner en ejecución el algoritmo con las distintas poblaciones, se obtuvieron las estadísticas que son reflejadas en la Tabla 5.1. Puesto que la cantidad de datos a mostrar es considerablemente grande, se ha decidido que en la Tabla 5.1 se muestre las estadísticas de las las generaciones de 10 en 10; al final del presente documento, en el Apéndice B, se adjunta las tablas con la totalidad de los datos para cada una de las distintas poblaciones. Para el análisis se va a tomar como referencia la información de las Tablas B.1, B.2, B.3 y B.4 que tienen todos los datos, por lo que se recomienda que se recurra a ellas.

	10 genomas				30 genomas			
	Media fit	Max fit	Tiempo	Veces llegadas a la meta	Media fit	Max fit	Tiempo	Veces llegadas a la meta
0	17,9	170	33,323	0	208,36667	1824	115,172	0
1	53,6	182	42,702	0	935,56667	4699	286,368	0
10	5407,7	21136	72,322	0	3172,1	19177	331,704	0
20	6360,4	31186	126,982	1	3173,8	19589	257,722	1
30	13530,8	31186	138,785	1	3807,2	20105	270,325	0
40	8508,6	31549	94,799	1	5991,1	23660	284,076	2
50	8187,8	31549	98,588	1	4166,86207	31586	360,918	1
60	7712,2	31549	123,576	2	6278,06667	31586	505,953	1
70	5201,1	31549	91,888	1	8349,79914	31586	333,611	1

	50 genomas				100 genomas			
	Media fit	Max fit	Tiempo	Veces llegadas a la meta	Media fit	Max fit	Tiempo	Veces llegadas a la meta
0	351,18	8537	221,79	0	575,32	18627	505,456	0
1	1020,7	8537	461,34	0	1323,12	20469	729,07	0
10	5507,64706	24810	613,121	0	4688,14	29198	1174,05	1
20	4904,04	31595	516,098	4	5275,24242	31352	1085,521	4
30	7926,48	35432	566,224	5	3837,2	31352	1085,378	5
40	7690,48	35445	547,237	4	5723,48485	31616	941,572	5
50	9369,43137	35445	642,535	7	5061,41837	31763	1071,84	6
60	8381,16327	35445	615,661	5	5008,24242	31763	1096,718	6
70	10637,25	35445	652,85	6	5869,478	44428	1362,652	5

Tabla 5.1: Estadísticas cada 10 generaciones de los distintos tamaños de población.

Tal y como se puede observar en las tablas adjuntas en el Apéndice B, al igual que en la Tabla 5.1, existe una columna para cada tamaño de población; dentro de ella se encuentran otras columnas referentes a las variables de las cuales se obtiene valor, y todo esto a lo largo de las 70 generaciones. Para facilitar la observación de cada una de estas

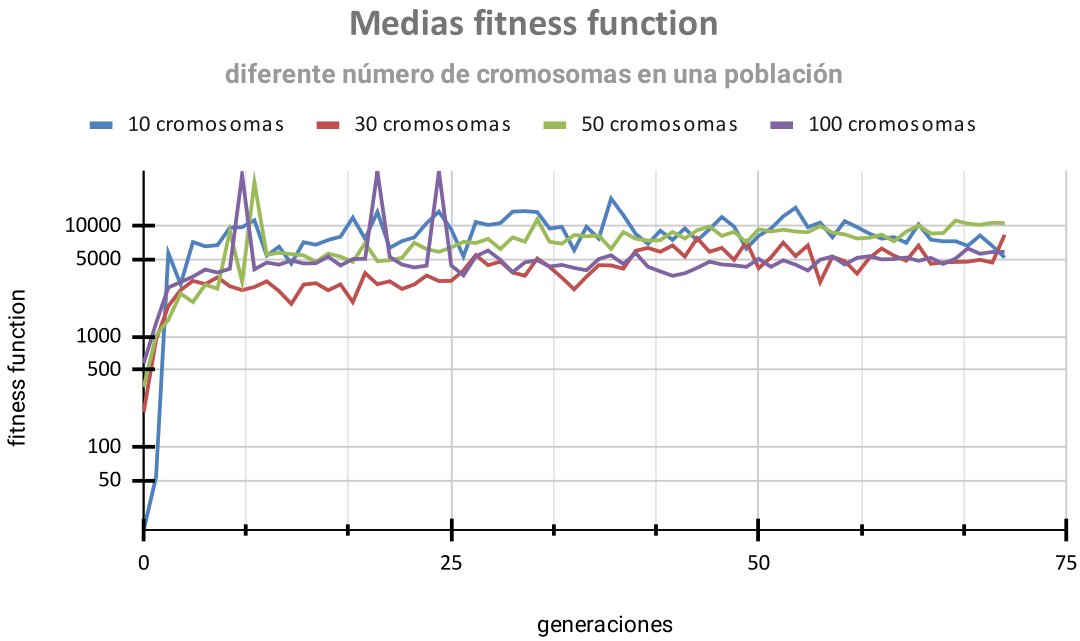

Figura 5.6: Valor medio de la *fitness function* para cada una de las poblaciones en las distintas generaciones.

variables en las distintas poblaciones se van a representar los datos mediante el uso de gráficas.

Como se puede observar en la gráfica de la Figura 5.6, cada población está representada con un color diferente. Es importante tener en cuenta que la escala del eje 'y' está en escala logarítmica. Puede apreciarse cómo en las primeras 6 generaciones el crecimiento del valor de la *fitness function* aumenta radicalmente; esto es debido a que en esas generaciones se van creando nuevos nodos y conexiones en la NN y ésta va modificándose para maximizar la puntuación. En este caso, el tamaño de población que consigue maximizar más puntuación, es la población de 10 cromosomas. Eso se debe a que al ser únicamente 10 individuos en cada generación, en caso de que haya uno cuya *fitness function* sea relativamente buena, se reproducirá con el siguiente que tenga mejor puntuación, y al ser una población tan pequeña, se extenderá la genética de forma más rápida que en las otras poblaciones. Las otras poblaciones, como puede observarse en la primera generación, consiguen mejores puntuaciones a las de la población de 10 cromosomas. Esto es debido a que al existir más número de individuos, también existe mayor posibilidad de que haya más individuos con una puntuación media más alta. A medida que se incrementa el número de generaciones el valor de la media deja de fluctuar y se estabiliza. Como puede observarse, la población de 50 cromosomas es la que, además de tener un crecimiento más uniforme, obtiene la mejor media de puntuación de la *fitness function*, obteniendo al final de las generaciones una media de 10000. Por lo tanto, de momento es la población más interesante.

En la Figura 5.7 se muestra cuál es la máxima puntuación que ha conseguido alguno de los individuos de esa generación concreta. Cabe destacar aquí que la población de 100 cromosomas es la que predomina al final de la ejecución, ya que es la que consigue una mayor puntuación en la *fitness function*.

En lo que a la Figura 5.8 respecta, se puede observar el número de veces que se llega a la meta por cada generación. La primera población que consigue llegar al final del nivel es la de 100 cromosomas, que lo hace en la sexta generación. Le siguen la población de 50

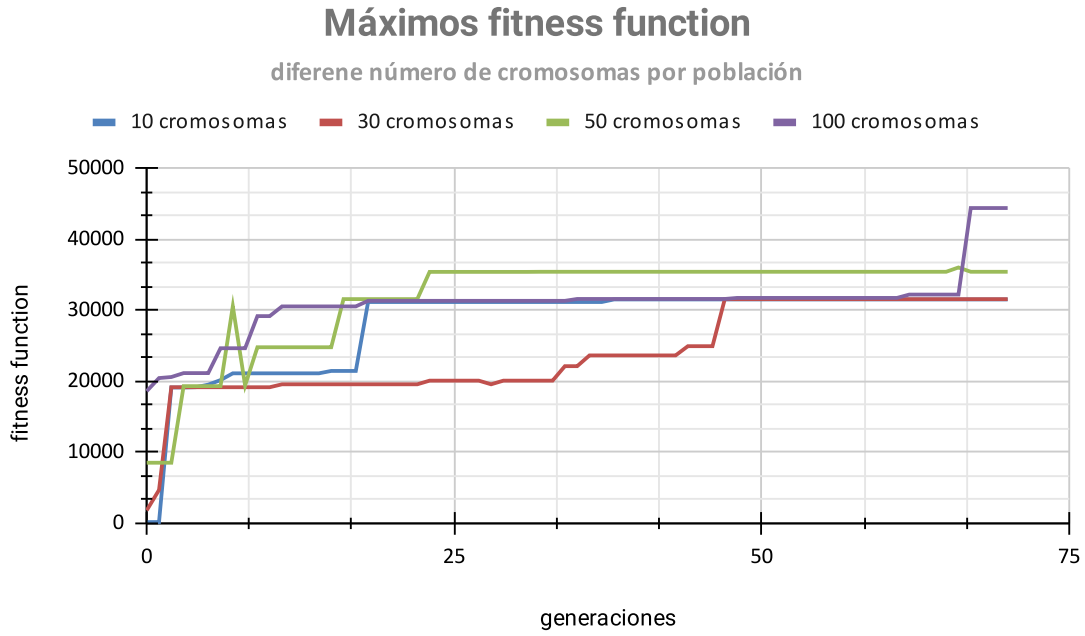

Figura 5.7: Máximo valor de la *fitness function* para cada una de las poblaciones en las distintas generaciones.

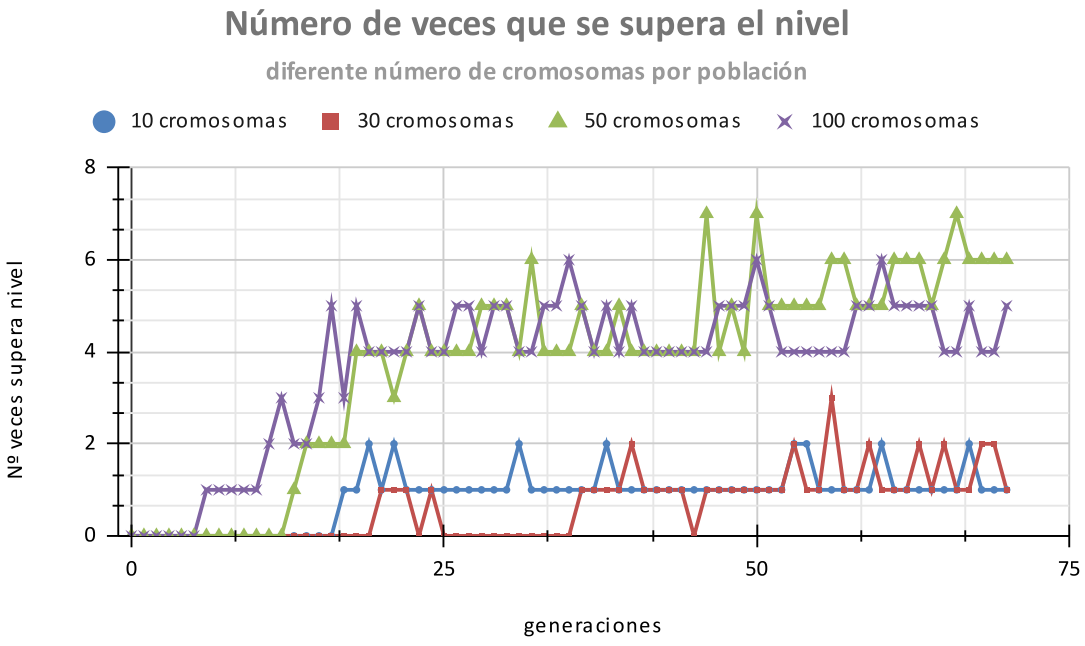

Figura 5.8: Número de veces que se llega a la meta por cada generación.

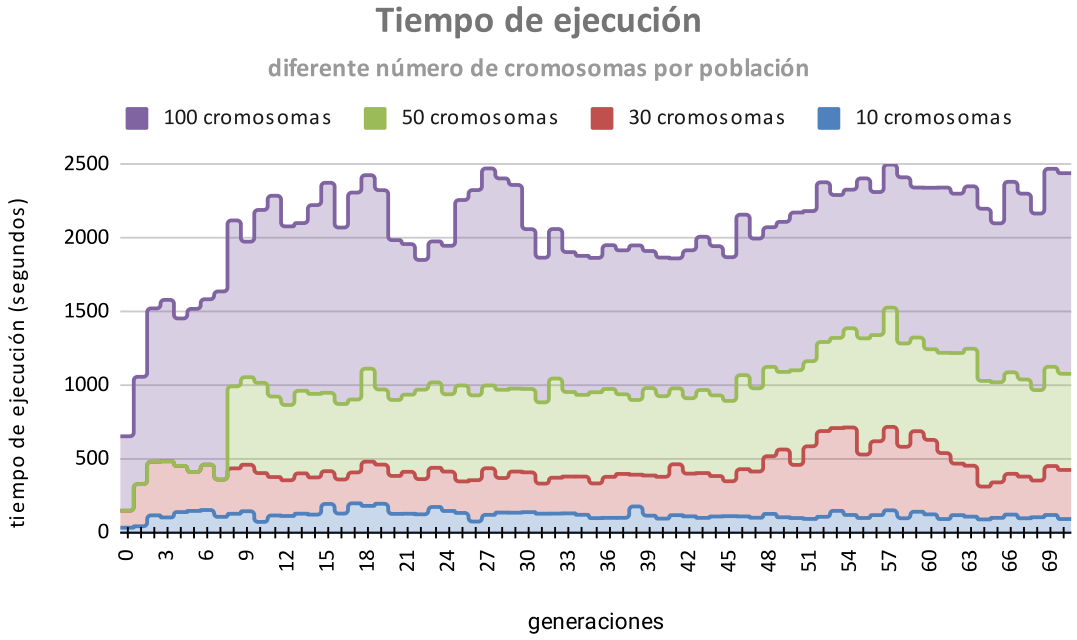

Figura 5.9: Tiempo empleado en cada generación.

cromosomas y tras ella las de 10 y 30 cromosomas. Esto se debe a lo que se comenta en la primera gráfica, y es que al tener mayor número de individuos, existe mayor posibilidad de tener uno muy apto; eso es lo que ocurre en la población de 100 cromosomas, llegando así antes que ninguna otra población a la meta. Pero esto es un arma de doble filo, pues a mayor número de individuos, también es más difícil que se obtengan más cromosomas muy aptos, pues la reproducción de estos individuos está limitada en cada generación. La población de 50 cromosomas empieza a llegar a la meta en la decimotercera generación, puede observarse cómo gracias a la reproducción de cromosomas aptos en un menor número de individuos respecto a la población de 100 cromosomas. Esto hace que en cuestión de diez generaciones iguales a la población de 100, y que además, a partir de la quincuagésima generación, la supere, teniendo más número de individuos que llegan a la meta.

Como se puede apreciar en la Figura 5.9, viene representado la cantidad de tiempo, en segundos, que te tarda en realizar el aprendizaje de toda la población en cada generación. A simple vista, la población de 100 cromosomas es la que más tiempo emplea en cada generación, superando con creces incluso a la población de 50. Las que menos tiempo invierten en el aprendizaje son la población de 10 y la de 30 cromosomas, mientras que la de 50 se sitúa entre la población de 100 cromosomas y estas últimas. Esto se debe, como es de esperar, al número de individuos de cada población, ya que a mayor número de individuos, mayor número de ejecuciones de Mario y mayor tiempo invertido jugando al videojuego para aprender.

Es en este punto donde uno se cuestiona cuál de estas poblaciones es mejor para el algoritmo que ha sido implementado. En relación al tiempo de aprendizaje, número de veces en pasarse el nivel (por generación) y puntuación media obtenida (valor de la *fitness function*), se concluye con que la población de 50 cromosomas es la más interesante, y es por tanto la que será utilizada para la comparación, en el Capítulo 7, con el algoritmo basado en RL que se verá en el Capítulo 6.

CAPÍTULO 6

Desarrollo del algoritmo basado en *Reinforcement Learning*

Tal y como se muestra en la Sección 3.4, el segundo algoritmo que se ha optado a desarrollar está basado en RL. A lo largo del capítulo se mostrarán las bases del RL, así como la técnica *Q-Learning* y su evolución, el *Deep Q-Learning*. Tras ello se explicará cómo se ha llevado a cabo la implementación del código, y, finalmente, se comentarán los resultados obtenidos de la ejecución.

6.1 Bases del *Reinforcement Learning*

Como se expone en la Sección 3.2, el RL consiste en un tipo de ML que tomará ciertas decisiones en función de la que le reporte mayor recompensa o *reward*. Este tipo de ML se inspira en la psicología del comportamiento, pues su aprendizaje podría ser equiparable al de un niño que aprende a realizar una nueva tarea, ya que si realiza las tareas de forma adecuada se le recompensa, pero en el caso contrario se le castiga.

Es importante destacar ciertos conceptos que son de vital importancia dentro del RL (además, se va a establecer una relación de ellos con el videojuego). Estos conceptos son [32]:

- **Agente**, se trata de de un componente software que toma decisiones inteligentes. Son los aprendices en el RL; éstos interactúan con el entorno mediante la realización de acciones, donde a cambio reciben un tipo u otro de recompensa en función de la acción. En el caso del videojuego que se trata en el presente trabajo, el agente sería el personaje Mario.
- **Entorno**, consiste en la demostración del problema a ser resuelto; existen dos tipos de entornos, los reales y los simulados. En este caso, Mario se encuentra en uno simulado, siendo el entorno el propio nivel.
- **Acciones**, consiste en el repertorio de movimientos que puede hacer un agente en un entorno. En el caso de Mario, las acciones que puede tomar son aquellas que están vinculadas con los botones del mando de la SNES.
- **Estado**, representa la posición del agente en un instante específico de tiempo, de modo que cuando el agente realice una acción el entorno le adjudica al agente (a Mario) una recompensa y un nuevo estado.
- **Transición**, consisten en el paso de un estado a otro.

- **Probabilidad de transición**, es la probabilidad de que un agente escoja el próximo estado en función del estado en el que esté.

En el caso frente al que nos encontramos, el agente, para tomar una decisión de qué acción va a realizar para llegar al siguiente estado, no va a emplear el histórico de estados anteriores, sino que tan sólo es necesario el estado actual. Por este motivo cumple con la propiedad de Markov [33], la cual cita que *'Los estados futuros son dependientes del presente pero independientes de los estados pasados'*, y que matemáticamente se expresa de la siguiente forma:

$$P[S_{t+1}|S_t] \leftarrow P[S_{t+1}|S_1, ..., S_t]$$

Donde:

- S_{t+1} designa el siguiente estado.
- S_t es el estado actual.
- S_1, ..., S_{t-1} hace referencia a los estados tomados hasta llegar al estado actual.

Esta propiedad es ideal para la ejecución del algoritmo que se va a implementar, puesto que a la hora de que Mario tenga que decidir qué acción llevar a cabo para llegar al siguiente estado únicamente necesitará el estado actual para la decisión, y no todos los anteriores, de forma que se reduce el coste computacional drásticamente.

6.2 *Q-Learning*

El *Q-Learning* es un algoritmo muy conocido dentro del RL que puede ser usado para resolver de forma óptima los procesos de decisión de Markov, más conocidos como *Markov Decision Processes (MDP)* **[33]**, con información incompleta.

Este tipo de RL se basa en la existencia de una tabla, llamada Q-tabla [34] [35], en la que se almacena la recompensa esperada si se realiza determinada acción en un estado concreto. Como se puede apreciar en el ejemplo que ilustra la Figura 6.1, el entorno está formado por una serie de casillas, donde el agente es el coche rojo y su objetivo es maximizar el *reward*; éste se obtiene en función del siguiente estado que se alcance. Las acciones que puede realizar el coche son moverse hacia arriba, abajo, izquierda y derecha; tal y como se muestra en su Q-tabla, dependiendo del estado al que se vaya y con la acción que se haga, se obtiene un *reward* diferente.

Q-tabla:

	000 100	000 010	000 001	100 000	010 000	001 000
⇧	0.2	0.3	1.0	-0.22	-0.3	0.0
⇩	-0.5	-0.4	-0.2	-0.04	-0.02	0.0
⇨	0.21	0.4	-0.3	0.5	1.0	0.0
⇦	-0.6	-0.1	-0.1	-0.31	-0.01	0.0

Figura 6.1: Ejemplo visual de la Q-tabla.

El *Q-Learning* pues se trata de un método adaptativo cuyo algoritmo se muestra En el Algorithm 6.1.

Algorithm 6.1 Algoritmo *Q-Learning*

1: **start**
2: Inicializar $Q(s, a)$ para todo $s \in S$ y $a \in A(s)$ (con 0's o valores aleatorios)
3: **repeat**
4: Inicializar s
5: **repeat**
6: Elegir a desde s usando una política
7: Ejecutar a, observando el estado resultante s' y *reward* r
8: $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$
9: $s \leftarrow s'$
10: **until** s sea terminal
11: **until** se recorra todos los episodios =0

En este algoritmo:

- ***s*** designa el estado en el momento de realizar la acción.
- ***a*** representa la acción realizada.
- ***s'*** es el estado tras realizar la acción.
- ***a'*** es la acción a realizar en el futuro
- ***Q(s',a')*** designa el nuevo valor que tendrá la Q-tabla para la fila s y la columna a, en otras palabras, la recompensa esperada dada al agente si toma esa acción en ese estado.
- ***Q(s,a)*** representa el valor actual de la Q-tabla para el estado s y la acción a.
- ***r*** es la recompensa que se obtiene al realizar la acción a en el estado s.
- α es el ratio de aprendizaje; tal y como se aprecia en la línea 8 del Algorithm 6.1, multiplica tanto a la recompensa obtenida como a la recompensa esperada. El valor de dicha multiplicación se suma al valor actual de la Q-tabla; dicho valor suele comprenderse en el intervalo [0.1, 0.3], donde a menor valor más lenta será la actualización de la tabla.
- ***$max_{a'}$ Q(s',a')*** representa el máximo valor que se espera para el estado s' de todas las posibles acciones posibles para ese estado.
- γ es el factor de descuento; cuanto mayor sea su valor, mayor es la importancia de las acciones futuras, mientras que cuanto menor sea su valor, mayor será la importancia de las acciones inmediatas.

El problema que presenta el *Q-Learning* es el uso de la Q-tabla. Ésta crece rápidamente con el número de acciones y de pasos posibles, donde el tamaño de la tabla que contiene todos los estados posibles será:

$$sizeQtabla = S \cdot a$$

Donde S representa el número de estados posibles y a el número de posibles acciones que se pueden tomar en cada estado. A simple vista, este valor no parece que vaya a ser demasiado grande, pero veamos qué valores toman estas variables en el presente estudio:

- Las posibles acciones que se podrían realizar vienen definidas por el factorial del número de botones (todas las posibles combinaciones de pulsados o no a la vez) que tiene el mando de la SNES, que, como se ha mostrado a lo largo del trabajo, son 12; por lo tanto, las posibles acciones que tendríamos serían:

$$12! = 479001600$$

- Pasamos a definir S como el conjunto de todos los posibles estados (s). En el caso de SMW, un estado s viene definido por la información que se recibe por pantalla, es decir, la resolución del juego y los tres canales de color RGB, que como ya se ha visto a lo largo de trabajo es de:

$$256 \cdot 224 \cdot 3 = 172032$$

  Donde además, para para poder apreciar dirección y velocidad de los objetos en movimientos, se almacenarían mínimo 4 *frames* consecutivos, lo cual cuadriplica el valor anterior, obteniendo el valor de:

$$256 \cdot 224 \cdot 3 \cdot 4 = 688128$$

  Aunque si se convirtiera la imagen a escala de grises el valor se reduciría a:

$$256 \cdot 224 \cdot 1 \cdot 4 = 227584$$

  Este último valor representaría la dimensión en una variable de tipo *array* para únicamente un estado. Tal y como se ha comentado anteriormente, la Q-tabla no almacena únicamente un estado, sino que almacena todos los posibles estados del juego, lo que implica 255 valores en la escala de grises; así pues, el número de posibles estados sería:

$$255^{227584}$$

Tras ser expuesto cómo serían las acciones y los estados en correspondencia a la Q-tabla de SMW, se puede deducir que sería una imprudencia implementar esta estrategia, pues el número resultante sería absurdamente grande, ocupando así un tamaño gigantesco de memoria que sería imposible de almacenar:

$$sizeQtabla = 255^{227584} \cdot 479001600$$

Incluso aunque la imagen se tratase más para reducir su tamaño, no sería viable, por lo que se ha optado por descartar el uso de una Q-tabla y buscar otra estrategia, que es mostrada en la Sección 6.3.

6.3 *Deep Q-Learning*

El *Deep Q-Learning* se basa en los mismos principios y conceptos que el *Q-Learning*, a excepción de que en lugar de utilizar una Q-tabla se usa una *Deep Neural Network (DNN)*. De esta forma se solucionan los problemas de memoria que se presentaban con el uso de las Q-tablas, como se puede ver en la comparativa en la Figura 6.2. Para el presente trabajo, la topología de la DNN - a partir de ahora *Deep Q Network (DQN)* - que se ha decidido implementar es la que se muestra en la Figura 6.3. Para ello nos hemos basado en una DQN que ha sido previamente implementada, obteniendo buenos resultados, para el videojuego *Sonic the hedgehog* para la videoconsola Megadrive [36].

Tal y como se puede apreciar en la Figura 6.3, la imagen (después de una serie de transformaciones que se explicarán en la Sección 6.5) se introduce dentro de la NN; ésta

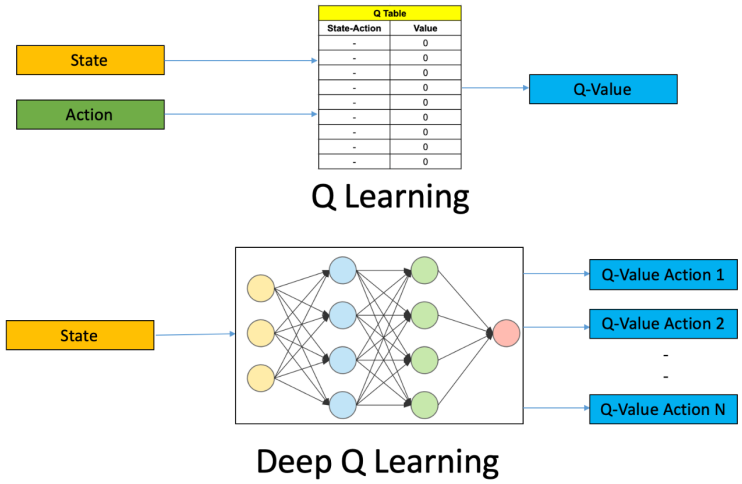

Figura 6.2: Comparativa *Q-Learning* y *Deep Q-Learning*.

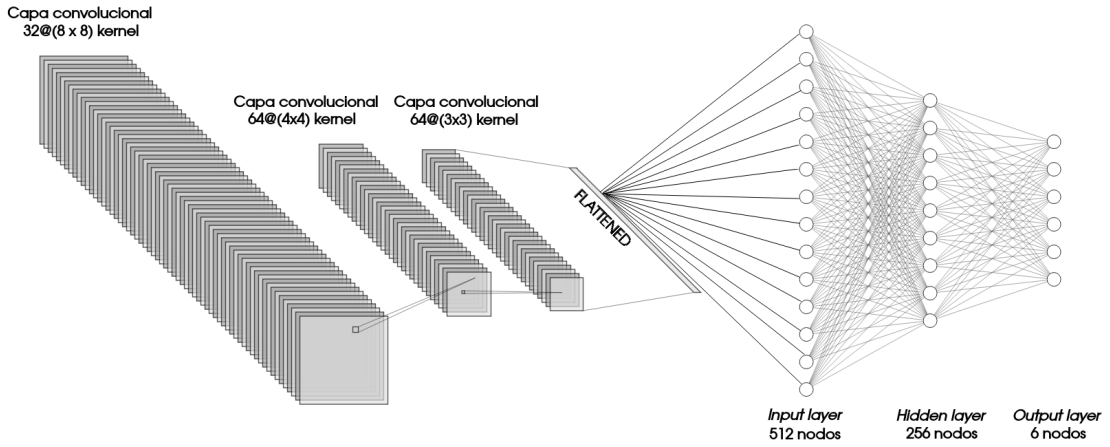

Figura 6.3: Topología de la *DQN*.

pasará a través de 3 capas convolucionales con diferentes *kernel sizes*, de 8x8 la primera, 4x4 la segunda y 3x3 la tercera. La función de las capas convolucionales es que mediante la información que le llega a cada una de las neuronas de la capa, se le realiza una operación algebraica y, tras ello, una función de activación no lineal (función *relu* en el presente caso); de esta forma se irá reduciendo la dimensión de la imagen de entrada. Al final de la tercera capa convolucional se realiza una función *flatten*, de forma que reduce a una sola dimensión los datos obtenidos hasta el momento. Tras esta reducción, la información pasará a la segunda parte de la DQN; ésta está compuesta por 3 capas totalmente conectadas (también conocidas como *Dense* o *Fully connected*); estas capas son:

- Capa de entrada o *input layer*, formada por 512 nodos que reciben la información de la última capa convolucional a la que le se le ha aplicado la reducción de dimensionalidad; usa una función de activación *relu*.
- Capa oculta o *hidden layer*, compuesta por 256 nodos y *relu* como función de activación.
- Capa de salida u *output layer*, formada por 6 nodos, que son los correspondientes al número de acciones que Mario podrá realizar; éstas se expondrán posteriormente en la Sección 6.5. La activación de estos nodos se realizará con una función lineal.

Es importante recordar que en las capas totalmente conectadas, como su nombre indica, cada uno de los nodos de una capa está conectado con todos y cada uno de los nodos de la capa siguiente.

6.4 Extensión del entorno de *retro-gym* mediante *Wrappers*

Antes de pasar a la explicación de la implementación del algoritmo, se va a exponer una extensión que nos ofrece la biblioteca *gym-retro* y que ha sido usada para mejorar la implementación del algoritmo basado en RL. Esta extensión consiste en una serie de clases envoltorio, más conocidas como *Wrappers*, y cuya jerarquía se muestra en la Figura 6.4. Como se aprecia en la Figura 6.4, la clase *Wrapper* hereda de la clase *Env*, y además es

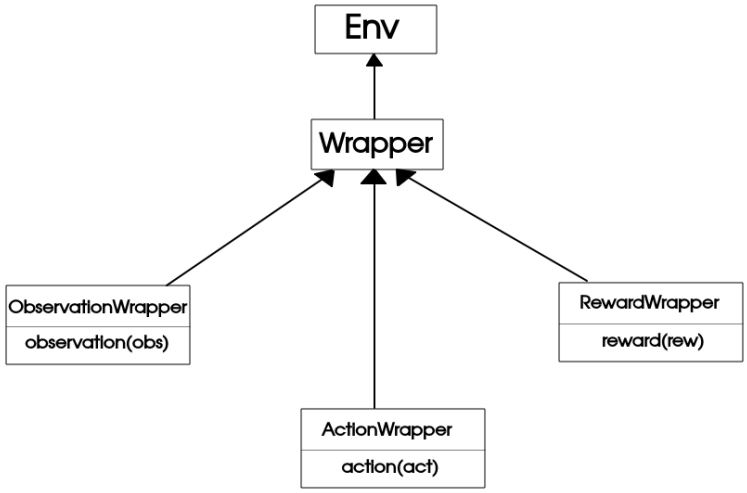

Figura 6.4: Jerarquía de *Wrappers gym-retro*.

padre de las siguientes clases:

- *ObservationWrapper*. Este *Wrapper* sirve para redefinir el método *observation(obs)*, donde el argumento *obs* es la observación que se obtiene desde el entorno; una vez ya se hayan realizado las operaciones oportunas, debe devolverse al agente en la sentencia *return*.
- *RewardWrapper*. Con él se redefine el método *reward(rew)*, que puede modificar el *reward* que se le asigna al agente.
- *ActionWrapper*. Con este *Wrapper* se redefine el método *action(act)*, que permite modificar la acción del entorno que se le pasa al agente.

En el caso del presente trabajo se han utilizado el *ObservationWrapper* y el *ActionWrapper*, a partir de las que se han creado clases que heredan de éstas. La jerarquía que se ha desarrollado es la que se muestra en la Figura 6.5. Como se puede apreciar, se ha creado una clase *MarioObservationDiscretizer* que hereda de *ObservationDiscretizer*, la cual a su vez hereda de *ObservationWrapper*; en la clase *ObservationDiscretizer* se han sobrecargado los métodos *step()* y *observation()*. Por otro lado, se puede visualizar que, al igual que se ha realizado anteriormente, se ha creado una clase *MarioActionDiscretizer* que hereda de *ActionDiscretizer*, la cual a su vez hereda de *ActionWrapper*. Esta vez, en la clase *MarioActionDiscretizer*, a parte de sobrecargar el método *action(act)*, también se ha sobrecargado el método *reset()*. Finalmente, es conveniente resaltar la importancia de las clases *MarioObservationDiscretizer* y *MarioActionDiscretizer*, que, aunque no sobrecarguen ningún

método, hacen de clase envoltorio heredando las variables y los métodos de sus padres, de forma que al crear una instancia de ellos podremos acceder a todas la funcionalidades de estas clases. La implementación de estas clases se expondrá en la Sección 6.5.

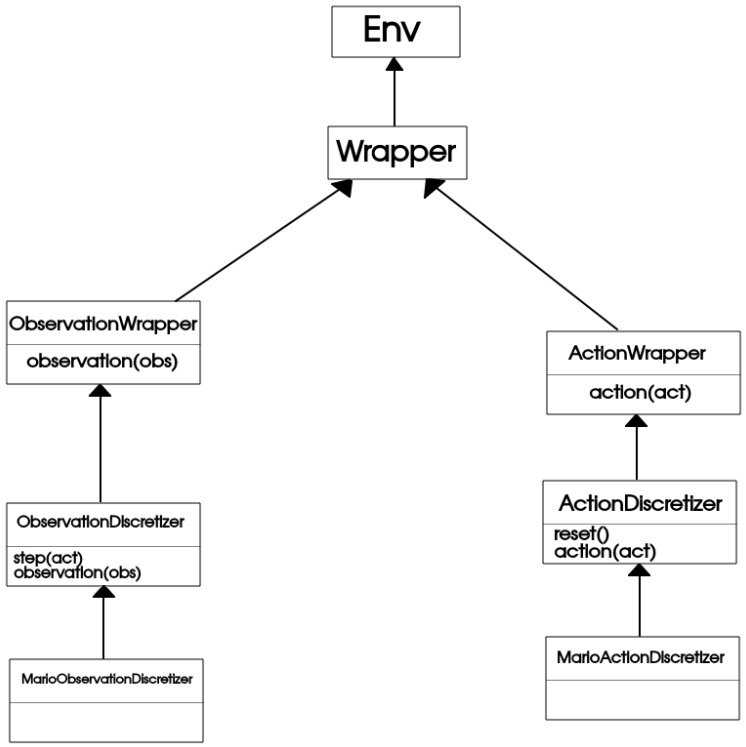

Figura 6.5: Jerarquía de *Wrappers* implementada para el desarrollo del algoritmo basado en RL.

6.5 Implementación del algoritmo basado en *Reinforcement Learning*

Antes de explicar cómo se ha implementado el algoritmo, es importante matizar ciertos puntos que han mejorado y facilitado la realización del mismo:

- Se ha utilizado la biblioteca *TensorFlow 2.3.1*, la cual internamente integra la biblioteca *Keras*; ambas van a proporcionar la facilidad de crear la topología de la DQN expuesta en la Sección 6.3.
- Uso de la biblioteca *keras-rl*, que consiste en una extensión de *Keras* que no viene incluida en *TensorFlow*; ésta ofrece funciones relacionadas con el RL para implementar el algoritmo visto en en la Sección 6.2.
- Tal y como se muestra en la Sección A.1 del Apéndice A, gracias a que en el hardware donde se ha entrenado la DQN alberga una tarjeta gráfica de *Nvidia*, se ha usado la tecnología *CUDA* e instalado la herramienta *Nvidia cuDNN*, mediante las cuales se ha acelerado el entrenamiento de una *NN*.

Una vez expuesto lo anterior, se va a proceder a explicar el código implementado para la realización de este algoritmo basado en RL. Es importante recordar que los códigos se encuentran completos en la Sección A.3 del Apéndice A, ya que el código que se va a mostrar a lo largo de esta sección no es completo; además, se han omitido funciones externas que no están relacionadas con la creación del algoritmo.

```
def main():

    env = MarioActionDiscretizer(retro.make(game='SuperMarioWorld-Snes', state=
        'YoshiIsland2.state'))
    env = MarioObservationDiscretizer(env)

    height, width, channels = 100, 100, 1
    actions = env.action_space.n

    model = build_model(height, width, channels, actions)

    agent = build_agent(model, actions)
    agent.compile(Adam(lr=1e-4))

    training_stats = agent.fit(env, nb_steps=1000000, visualize=True, verbose
        =2)

    scores = agent.test(env, nb_episodes=10, visualize=True)

    env.close()
```

Listing 6.1: Función *main*.

Como se puede apreciar en el Listing 6.1, se utilizan las clases que han sido creadas, tal y como se muestra en la Figura 6.5 de la Sección 6.4, a partir de las *Wrappers*, heredando y sobrecargando algunos de sus métodos. Es por ello por lo que creamos el entorno mediante la instanciación del objeto tipo *MarioActionDiscretizer*, al que se le pasa como parámetro el entorno (creado con la función *make(...)*). Internamente esta clase sobrecarga los métodos *reset* y *action*. Tras ello se le pasa el objeto anterior como parámetro al nuevo objeto de tipo *MarioObservationDiscretizer*, para sobrecargar también los métodos *observation* y *step*; de esta forma se tendrá en la variable *env* almacenado un entorno totalmente personalizado.

Tras ello se almacena en las variables *height*, *width* y *channels* la resolución y el número de canales en los que se va a trabajar. En este caso, la transformación que se ha decidido tomar para la imagen es ligeramente diferente a como se expuso en la implementación del algoritmo basado en AG.

```
def processFrame(frame, shape=(100,100)):

    frame = frame.astype(np.uint8)
    frame = cv2.cvtColor(frame, cv2.COLOR_RGB2GRAY)
    frame = frame[40:40+224, :224]
    frame = cv2.resize(frame, shape, interpolation=cv2.INTER_NEAREST)
    frame = frame.reshape((*shape, 1))

    return frame
```

Listing 6.2: Función *processFrame* para la transformación de la imagen.

Esta vez, mediante la llamada a la función *processFrame*, realizará un cambio de los 3 canales de color RGB a escala de grises en la imagen, un recortado, un reajuste en la dimensionalidad, y, finalmente, una interpolación, tal y como se muestra en la Figura 6.6. Todo esto, tal y como se muestra en el Listing 6.2, es posible gracias al uso de la biblioteca *OpenCV*. La transformación pues, será desde la resolución de 256x224x3 a la resolución de 100x100x1; de esta forma se facilitará y agilizará la ejecución y el entrenamiento de la DQN.

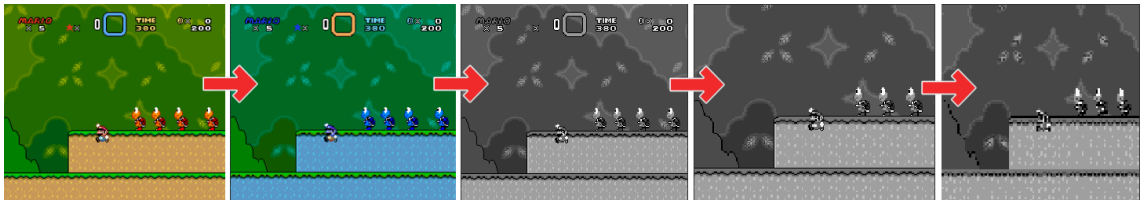

Figura 6.6: Transformaciones realizadas a la imagen.

Continuando con la explicación de la función *main*, tras realizar la transformación de la imagen, almacenaremos las acciones posibles que podrá realizar Mario. Esta vez, a diferencia de como se hacía en AG, las acciones estarán acotadas a las siguientes:

```
combos=[['LEFT'], ['RIGHT'], ['RIGHT', 'B'], ['RIGHT', 'Y'], ['B'], ['A']]
```

Donde, gracias a la sobrecarga del método *action* en la clase *MarioActionDiscretizer*, transformará el *array combo* con las acciones posibles al *array* de los 12 booleanos vinculados con los botones del mando. Esta reducción en el número de acciones resultará en que Mario pueda aprender de forma más rápida, pues son las combinaciones de botones que más se usan a la hora de jugar al SMW, como se mostró en la Sección 6.2; ahora, en lugar de tener 12! = 479001600 acciones se tendrán únicamente 6 (almacenadas en la variable *actions*).

Prosiguiendo con la función *main*, lo siguiente con lo que nos encontramos es con la llamada a la función *build_model*, la cual va a ser la encargada de construir la estructura de la DQN que ha sido expuesta en la Sección 6.3.

```
def build_model(height, width, channels, actions):
    model = Sequential()

    model.add(Convolution2D(32, (8,8), strides=(4,4), activation='relu',
        input_shape=(3, height, width, 1)))
    model.add(Convolution2D(64, (4,4), strides=(2,2), activation='relu'))
    model.add(Convolution2D(64, (3,3), activation='relu'))
    model.add(Flatten())

    model.add(Dense(512, activation='relu'))
    model.add(Dense(256, activation='relu'))
    model.add(Dense(actions, activation='linear'))

    return model
```

Listing 6.3: Función *build_model* para la construcción de la DQN.

Como se aprecia en el Listing 6.3, se le pasará a la función las variables *height*, *width*, *channels* y *actions*, y mediante el uso de los métodos que nos ofrece la biblioteca *TensorFlow 2*, concretamente *Keras*, como son *Convolution2D* (para crear las capas convolucionales) y *Dense* (para crear las capas totalmente conectadas) se va creando la estructura de la DQN y éstas se van añadiendo en la variable *model*.

Tras haber creado la estructura de la DQN se procede a crear el agente mediante la función *build_agent*, a la cual se le pasará la NN creada anteriormente y las posibles acciones que puede realizar Mario. Antes de explicar la implementación de dicha función, es importante tener en cuenta que se va a utilizar la estrategia ***experience replay***; ésta consiste en que, en lugar de que se actualice la DQN a cada paso, se almacenan los pares estado-acción en un *buffer*, llamado *replay buffer*, y cuando éste ya esté completo, se seleccionará uno de esos pares de forma aleatoria; así se evitará el sobreentrenamiento de la NN. Esto último puede verse reflejado en la segunda línea del Listing 6.4, donde, mediante el uso de las funciones ofrecidas por la biblioteca *keras-rl*, se crea una política,

a la cual se le pasa como parámetro la función *EpsGreedyQPolicy*, que será el que cogerá estos pares del *replay buffer* de forma aleatoria. Tras ello, en la línea 6 se le asignará al agente la NN, la política, las acciones posibles y se activará el *Dueling Network*.

```
1  def build_agent(model, actions):
2      policy = LinearAnnealedPolicy(EpsGreedyQPolicy(), attr='eps', value_max=1.,
           value_min=.1, value_test=.2, nb_steps=10000)
3
4      memory = SequentialMemory(limit=1000, window_length=3)
5
6      dqnAgent = DQNAgent(model=model, memory=memory, policy=policy,
7                          enable_dueling_network=True, dueling_type='avg',
8                          nb_actions=actions, nb_steps_warmup=1000
9                         )
10
11     return dqnAgent
```

Listing 6.4: Función *build_model* para la construcción de la DQN.

El *Dueling Network* [37] hace que la parte totalmente conectada de la DQN se altere, creando así dos flujos de capas totalmente conectadas (Figura 6.7), de forma que uno de los flujos tiene el valor que se asocia a cada estado *(v)* y el otro flujo tiene la ventaja o desventaja asociada a cada uno de los estados *(adv)*. Finalmente, estos dos flujos se conectan mediante la operación:

$$q = v + adv - mean(adv)$$

y se obtiene un *q-value (q)* para cada uno de los estados. Por último, se realizará aquella acción cuyo estado tenga el *q-value* máximo. La ventaja que aporta esta estrategia es que la DQN puede aprender qué estados son (o no son) valiosos sin tener que aprender el efecto de cada acción para cada estado. Tras todo lo anterior descrito, la función *build_agent* devolverá el agente a la función *main*, almacenándola en la variable *agent*.

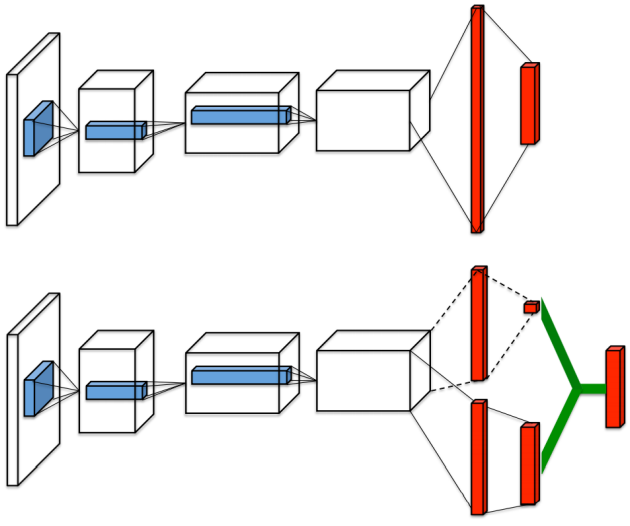

Figura 6.7: Comparativa gráfica *Single Network* y *Dueling Network*.

Prosiguiendo por la línea 14 del Listing 6.1, mediante el método *fit* de la biblioteca *keras-rl* comenzará el entrenamiento, donde la variable *nb_steps* tomará el valor de 1000000; ésta representa el número de iteraciones que se van a realizar para el entrenamiento. En cada iteración se procesará el *frame* con la función *proccessFrame* y se utilizarán los métodos que se han sobreescrito para las clases *MarioObservationDiscretizer* y *MarioActionDiscretizer*, donde concretamente cabe destacar el método *step* de la clase

MarioObservationDiscretizer. Se han seguido dos estrategias diferentes para la implementación de esta función. En la primera implementación, representada en el Listing 6.5, el método que se ha elegido para la penalización de las acciones que no son prósperas es el de restar cierta puntuación al *reward*. En la segunda, el medio por el que se penaliza es mediante la resta de un cierto porcentaje al *reward*, tal y como se muestra en el Listing 6.6. En lo que a la recompensa respecta, en ambos métodos se asignará de la misma manera; aparte de la puntuación que Mario obtenga mediante la recolección de monedas, la destrucción/activación de bloques y la exterminación de los enemigos, también se incrementará en 5 el *reward* cuando Mario incrementa su desplazamiento hacia la derecha en el eje x. Esta misma estrategia es la que se aplicó en la implementación del algoritmo genético expuesto en el Capítulo 5.

```
def step(self, act):
        global episodeReward
        global xPosMax
        global xPos
        global frameCounter
        global numIteracion
        global f_real

        obs, rew, done, info = super().step(act)
        episodeReward += rew

        xPos = info['x']

        if (xPosMax < xPos):
            frameCounter = 1
            xPosMax = xPos
            episodeReward += 5
        else:
            frameCounter += 1
            episodeReward -= 1

        if frameCounter %300 == 0:
            episodeReward -= 10

        if frameCounter == 1200:
            episodeReward -= 100
            done = True

        if info['endOfLevel']:
            episodeReward += 30000
            done = True

        if info['dead'] == 0:
            episodeReward -= 1000
            done = True

        rew = episodeReward

        if done:
            numIteracion += 1
            f_real.write(str(numIteracion) + ";" + str(rew) + "\n")

        return obs, rew, done, info
```

Listing 6.5: Función *step* versión resta de valores concretos.

Como puede apreciarse en el Listing 6.5, la penalización de las acciones que tome Mario será cuando:

- Se desplace a la izquierda o no se desplace: se restará en una unidad el *reward* por cada *frame* en el que no avance hacia la derecha.
- Pasen 300 *frames* en los que Mario no haya avanzado más allá de la posición máxima en el eje x en esa partida: se le restará en 10 unidades el valor del *reward*.
- Pasen 1200 *frames* sin avanzar mas allá de la posición máxima alcanzada en esa partida: en tal caso se decrementará en 100 el valor del *reward*.
- Mario muere: se le restarán 1000 unidades al *reward*.

```
def step(self, act):
        global episodeReward
        global xPosMax
        global xPos
        global frameCounter
        global numIteracion
        global f_real

        obs, rew, done, info = super().step(act)
        episodeReward += rew

        xPos = info['x']

        if (xPosMax < xPos):
            frameCounter = 1
            xPosMax = xPos
            episodeReward += 5
        else:
            frameCounter += 1
            episodeReward = float(int(episodeReward * 0.999)) #Restamos el 0.1%

        if frameCounter %300 == 0:
            episodeReward = float(int(episodeReward * 0.95)) #Restamos el 5%

        if frameCounter == 1200:
            done = True

        if info['endOfLevel']:
            episodeReward += 30000
            done = True

        if info['dead'] == 0:
            episodeReward = int(episodeReward * 0.70) #Restamos el 30%
            done = True

        rew = episodeReward

        if done:
            numIteracion += 1
            f_real.write(str(numIteracion) + ";" + str(rew) + "\n")

        return obs, rew, done, info
```

Listing 6.6: Función *step*

Tal y como se muestra en el Listing 6.6, en la versión de la resta de porcentajes la forma de penalizar a Mario será cuando:

- Se desplace a la izquierda o no se desplace: se restará en un 0.1 % el valor del *reward* por cada *frame* en el que Mario no se desplace hacia la derecha.

- Pasen 300 *frames* en los que Mario no haya avanzado más allá de la posición máxima en el eje *x* en esa partida: se le restará un 5 % el valor del *reward*.
- Mario muere: se decrementará en un 30 % el valor del *reward*.

En la Sección 6.6 se expondrán los resultados obtenidos realizando el aprendizaje de Mario mediante los dos métodos expuesto, eligiendo finalmente aquel con el que se hayan obtenido mejores resultados.

6.6 Análisis de los resultados obtenidos

En esta sección se exponen los resultados obtenidos del entrenamiento de la DQN. Como se ha mostrado al final de la Sección 6.5, se han utilizado dos métodos para el aprendizaje de Mario. Estos métodos son:

- El método de la resta de valores concretos, que a partir de este momento se referirá a él como el *Método Resta (MR)*.
- El método de la resta de porcentajes, que pasará a llamarse el *Método Porcentajes (MP)*.

Antes de pasar a interpretar los datos, es importante recordar que para el entrenamiento de la DQN se ha decidido que la variable *nb_steps* tome el valor de 1000000 (Listing 6.1), que, como se expuso en la Sección 6.5, representará el número de *frames* que se procesarán para el entrenamiento. En cada procesamiento del *frame* entrará la imagen procesada en la primera parte de la DQN y, tras una serie de reducciones de dimensionalidad de la misma, modificará los pesos de los diferentes nodos de las diferentes capas de la segunda parte de la red neuronal, para finalmente activar los nodos de la *output layer* y asignar una acción a realizar a Mario. Con 1000000 *frames* se ha obtenido un total de 591 iteraciones; en cada una de ellas cabe destacar las variables:

- ***mean_reward***, valor medio del *reward*.
- ***mean_max_action***, valor máximo del *reward* alcanzado en cierto *frame*.
- ***episode_reward***, valor total del *reward* obtenido en esa iteración.
- ***duration***, cantidad de tiempo (en segundos) que ha tardado en ejecutar esa iteración.

Tal y como se expone en la Sección A.1, la DQN ha sido entrenada en una gráfica *Nvidia* compatible con la tecnología *CUDA*, obteniendo los siguientes tiempos de entrenamiento:

- Tiempo en MR: 158731 segundos, aproximadamente 44 horas.
- Tiempo en MP: 197706 segundos, aproximadamente 55 horas.

Como se puede observar, el tiempo de entrenamiento en el MR es mucho menor que en en el MP, habiendo una diferencia de aproximadamente 11 horas (un 20 % aproximadamente menos de tiempo). Existe la posibilidad de que gran parte de la diferencia de tiempo entre los dos métodos sea debido a que, en el MP se hace operaciones de multiplicación con *floats* y además se hace un *casting* a tipo *int* para redondear valores para posteriormente retornar al tipo *float* (líneas 20 y 23 en el Listing 6.6); resulta que este tipo

de operaciones tienen un coste computacional adicional. Por lo tanto, en lo que al tiempo de entrenamiento respecta, el MR sería mejor opción.

Todos los datos de las variables anteriormente mencionadas, además de otras, han sido recogidas en una tabla conjunta. Al tener una dimensión demasiado grande, en lugar de adjuntarla en el presente trabajo, se ha decidido que su consulta se realice por medio de la URL a pie de página [1].

Comenzando por la variable *mean_reward*, el valor de ésta se calcula mediante la división del *episode_reward* entre el número de *frames* que se ha procesado en esa iteración. Como se puede apreciar en la Figura 6.8, el valor del *mean_reward* va aumentando rápidamente en ambos métodos a lo largo de las primeras iteraciones hasta llegar a la iteración 37 en el caso del MR y la iteración 21 en el caso del MP. A partir de ahí el crecimiento empieza a ser más lento, habiendo picos de subida y de bajada. El motivo por el que ocu-

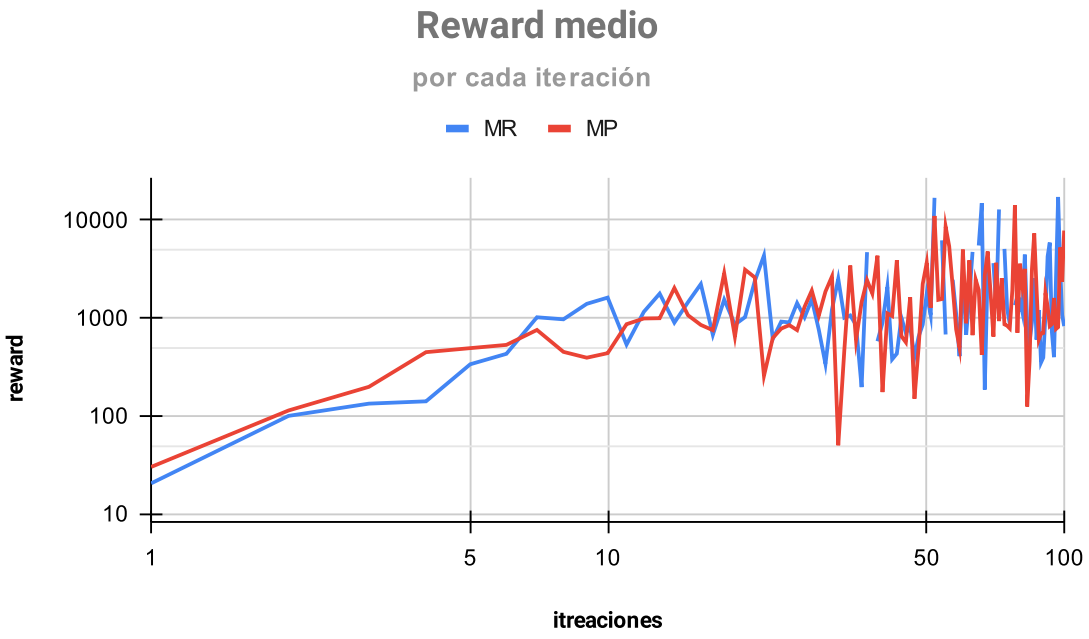

Figura 6.8: Valor del *reward* medio en las primeras 100 iteraciones.

rre este repentino aumento del *mean_reward* en primeras iteraciones es debido a que, al principio del nivel, Mario se encuentra en un entorno determinista, en el que los enemigos (como son las tortugas de color verde y rojo) se mueven siempre de la misma forma, por lo que eso hace que el entrenamiento de Mario sea más fácil. Cuando se empieza a estabilizar el valor medio del *reward* se debe a que Mario pasa a la parte del nivel donde se encuentra con otro tipo de enemigos (jugador de rugby y los topos), que, a diferencia de los anteriores, comienzan a perseguirle, haciendo de esta forma que el entorno pase a ser más aleatorio. Esto puede verse reflejado en las siguientes iteraciones, pues el valor medio del *reward* empieza a crecer poco a poco y, además, se pueden apreciar picos de subida y de bajada. En picos de subida Mario consigue gran cantidad de puntuación debido a que cuando llega a la meta se le otorga una gran cantidad de puntos (línea 26 del Listing 6.6), por lo que eso hace que se dispare el valor del *mean_reward*; por tanto son esos picos los que muestran que Mario ha concluido con el nivel. Por el contrario, en los picos de bajada se debe a que Mario comienza a ir hacia la izquierda en el nivel o a quedarse parado, esto ocurre únicamente en la parte aleatoria del nivel. Aunque es casi

[1] *URL* de la tabla conjunta para MR y MP: https://docs.google.com/spreadsheets/d/113whPaDOUSHQbOQWE5kAaZq2wOMq7w_Ji7spzXk1GFI/edit?usp=sharing

imperceptible, en la Figura 6.8, en la representación de los datos del MR, hay intervalos en los que parece no haber un valor de *mean_reward*; esto se debe a que el eje *y* de la gráfica está en escala logarítmica y el valor del *mean_reward*, al ser negativo, ni siquiera es representado en la gráfica. Extrapolando estos valores negativos al SMW, éstos se originan porque, cuando Mario se desplaza hacia la izquierda, se le va restando un valor constante al *reward* total, sin importar si el el valor de éste es menor a 0. Este hecho puede verse de forma más clara en la Figura 6.9, donde se muestran únicamente las 100 primeras iteraciones del entrenamiento y sin representar el eje *y* en escala logarítmica. Como se

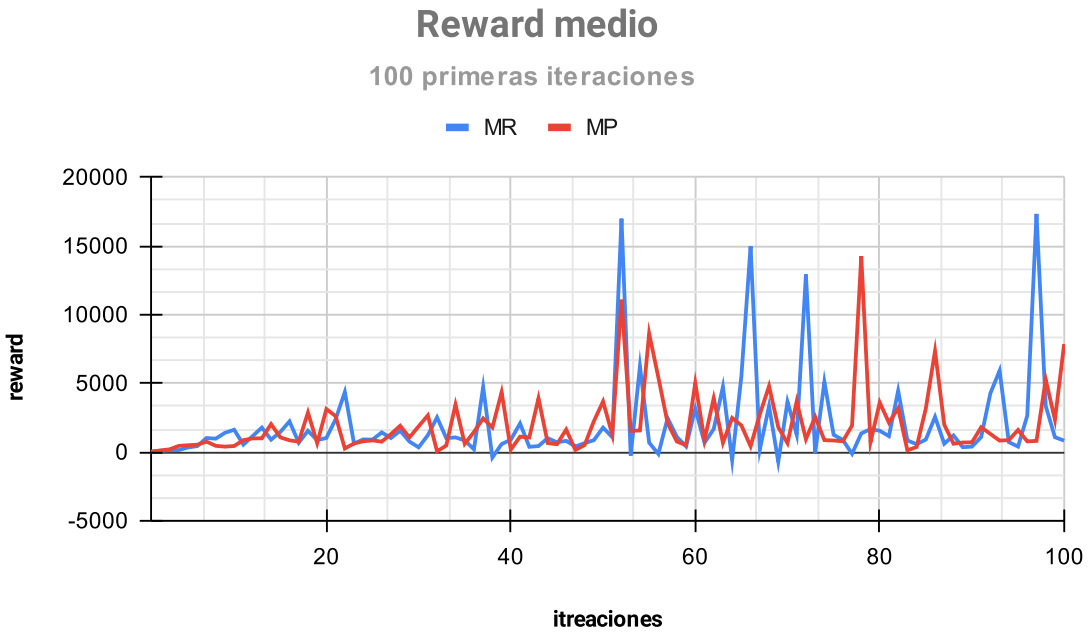

Figura 6.9: Valor del *reward* medio en las 100 primeras iteraciones (sin escala logarítmica).

puede apreciar en la Figura 6.9, ahora, en iteraciones como la 38, la 64 o la 69 se muestran estos valores negativos.

Prosiguiendo con el análisis de los resultados, la variable *episode_reward*, tal y como se había explicado anteriormente, está ligada a la variable *mean_reward*. Como se muestra en la la Figura 6.10, su interpretación es equivalente a la que se había hecho para la anterior variable, a diferencia de que, tanto para el MR como para el MP, los valores mínimos, proporcionalmente, no son tan bajos como lo eran en el *mean_reward*. Esto se debe a que, como es una variable acumulativa, se acumula el *reward* de cada *frame*; entonces, a poca puntuación que consiga Mario, ésta va escalando exponencialmente, de ahí que salgan valores tan grandes. Esto lo realiza automáticamente la librería *keras-rl*, pero no supone ninguna desventaja a la hora de interpretar. Finalmente, al igual que ocurría en el *mean_reward*, y de forma excepcional, también existen valores negativos que no son mostrados, como ocurre en la iteración 15; en estos casos Mario ni siquiera se desplazaría a la derecha en ningún momento, sino que se quedaría quieto o se desplazaría continuadamente hacia la izquierda.

En la Figura 6.11, se representa la variable *mean_max_action*, que consiste en el valor máximo de *reward* que ha alcanzado Mario en un cierto *frame* a lo largo de la iteración. En ella puede verse que, en ambos métodos, a medida que se va avanzando en el entrenamiento, esta variable va aumentando ligeramente; esto se debe a que Mario va aprendiendo qué acciones le van a reportar un mayor *reward* en función del estado en el se encuentre. A partir de la iteración 100, comienza a haber picos de subida considerables;

Figura 6.10: Valor del *reward* total en cada episodio (100 primeras iteraciones).

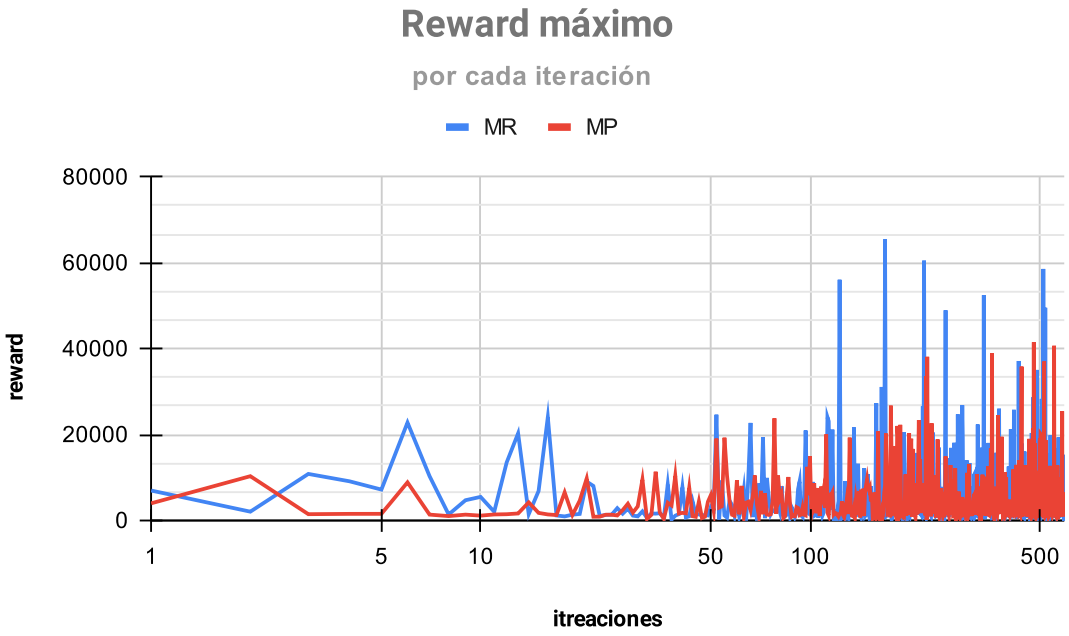

Figura 6.11: Valor máximo de *reward* obtenido en un *frame* para cada iteración.

eso lo que quiere decir es que Mario ha alcanzado la meta en esa iteración, puesto que, como se comenta a lo largo de esta sección, cuando Mario consigue llegar a la meta, se le otorga una gran cantidad de puntuación a su *reward*, que sumados al *reward* del *frame* anterior dan lugar a un máximo de *reward*. En este caso, tal y como se muestra en la Figura 6.11, el MR obtiene mayor puntuación máxima que el MP; además, lo hace mucho antes, lo cual quiere decir que la opción de resta de valores concretos resulta de mayor interés.

La última variable a estudiar es la *duration*, que representa la cantidad de tiempo (en segundos) que se ha tardado en realizar cada iteración. Como muestra la Figura 6.12, a simple vista se puede apreciar cómo el valor de *duration* en el MP es mayor al del MR en la mayoría de iteraciones, y esto se ve reflejado en la diferencia de tiempo de entrenamiento de los dos modelos que se comenta al principio de esta sección. Por tanto, en este ámbito el MR vuelve a resultar más interesante que el MP.

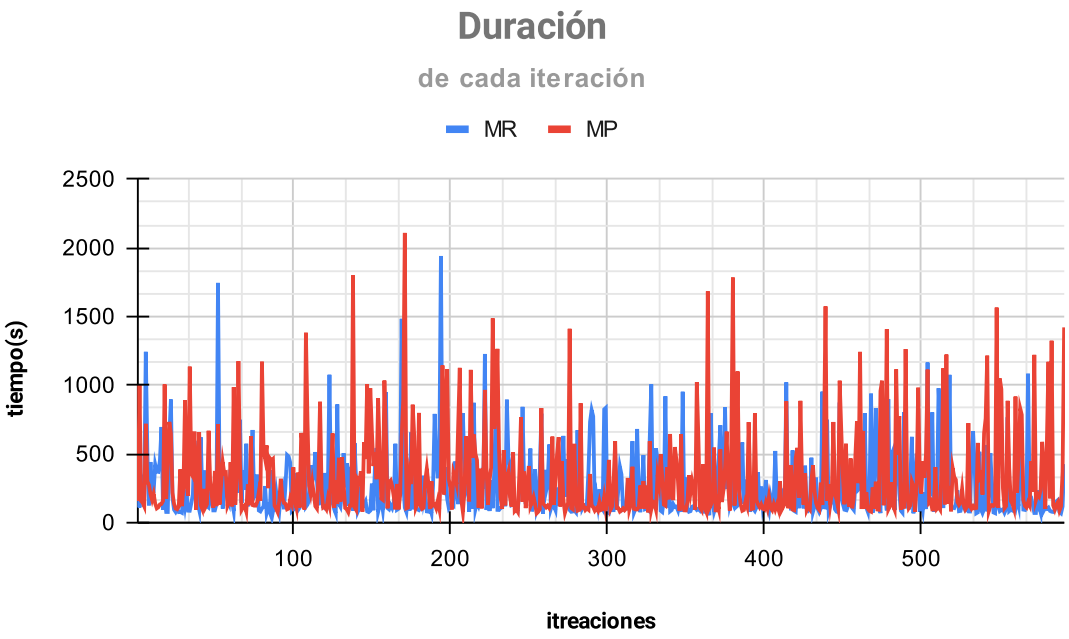

Figura 6.12: Duración de cada iteración.

En conclusión, tal y como se ha mostrado a lo largo de la presente sección, el método de la resta de valores concretos (MR) resulta más interesante que el método de la resta de porcentajes, tanto a nivel de eficiencia como a nivel de eficacia, pues además de tardar menos tiempo en entrenar la DQN, obtiene mejores valores en el *reward*. Un aspecto importante a tener en cuenta para una futura mejora de este método es el de hacer que cuando el valor del *reward* sea 0, no sea posible seguir aplicándole resta de valores concretos, para de esta forma no obtener resultados negativos.

CAPÍTULO 7
Comparativa de los algoritmos implementados

A lo largo de este capítulo se va a mostrar una comparativa de los dos algoritmos que se han implementado en el presente trabajo; esta comparativa se hace en diferentes ámbitos, como son la efectividad y eficiencia del entrenamiento, y la carga computacional. Es importante matizar que las versiones que van a compararse son aquellas cuyos resultados resultaron más interesantes; éstas son la ejecución con 50 cromosomas en AG y el entrenamiento de la DQN mediante el MR. Finalmente se obtendrá una conclusión de qué algoritmo resulta de mayor interés en función del objetivo final que se busque.

7.1 Eficacia y eficiencia del entrenamiento

Como se ha podido observar en el análisis de sendos algoritmos (Secciones 5.4 y 6.6 respectivamente), en lo que a eficacia respecta han cumplido con creces, pues han conseguido hacer que Mario llegue al final del nivel, que era el objetivo principal de este trabajo. Ahora bien, la forma en la que se ha conseguido el objetivo dista una de la otra. En el primer caso, nos encontramos con un AG que va modificando la topología de una NN para obtener la mejor versión de ésta y hacer que Mario consiga la máxima puntuación posible; en el segundo caso se ha construido una NN a la que se le ha bautizado con el nombre de DQN; en ella la topología de la NN es estática y se basa en los principios del *Q-Learning*.

Centrándose en la eficiencia hay una clara diferencia en ambas ejecuciones; ésta se ve reflejada en el tiempo de entrenamiento. Para la comparativa, lo más importante es centrarse en la primera vez que Mario alcanzó la meta:

- En el algoritmo basado en AG, basándose en los datos que se muestran en el Apéndice B, puede apreciarse como la primera vez que se llega a la meta es en la decimotercera generación, consumiendo un tiempo de 7.348 segundos, aproximadamente 2.1 horas.

- En algoritmo basado en RL, como se muestra en los datos de la tabla que se encuentra en la URL adjunta al comienzo de la Sección 6.6, la primera vez que Mario consigue llegar a la meta es en la iteración número 123, tomándole un tiempo de 33977 segundos, aproximadamente 9.4 horas.

Se puede apreciar que el algoritmo basado en AG consigue que Mario llegue a la meta mucho antes, tardando una quinta parte de lo que le toma al algoritmo basado en *RL*,

por lo que se podría concluir que en términos de eficiencia, para conseguir el objetivo propuesto, el AG sale victorioso frente al RL.

7.2 Carga computacional

En esta sección se va a tener el cuenta la cantidad de CPU y memoria RAM que abarca cada uno de ambos algoritmos. En la Figura 7.1 se puede apreciar la carga que está teniendo la ejecución del primer algoritmo; éste está usando uno de los 8 *cores* de los que dispone la CPU, y lo está usando al 87.3 %. Por otro lado, está utilizando un 2.5 % de los 8 GB de memoria RAM disponibles, lo que equivaldría a aproximadamente 205 MB.

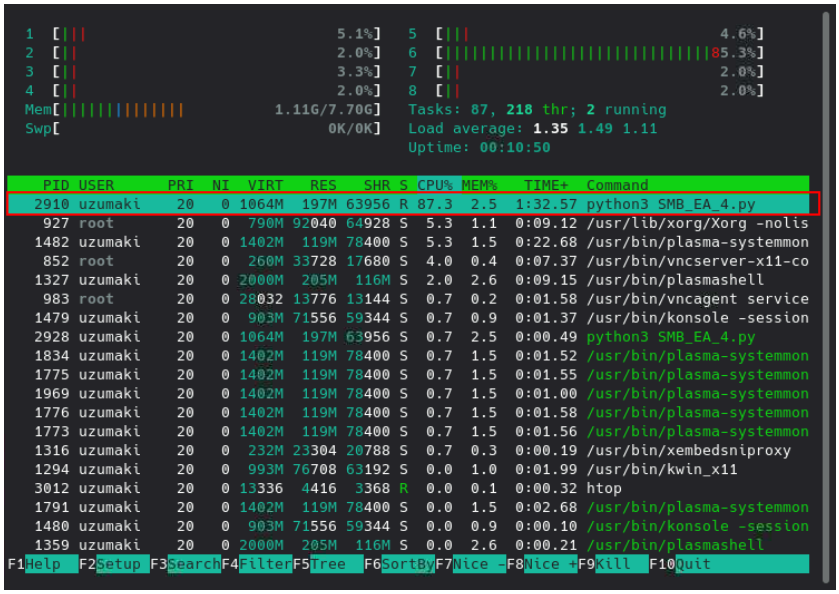

Figura 7.1: Carga computacional algoritmo basado en AG.

Por otro lado, en la Figura 7.2 se representa la carga computacional del algoritmo basado en RL; como puede apreciarse, éste utiliza la totalidad de los 8 *cores*, y haciéndolo a una media del 84 % en cada uno de ellos. En cuanto a la cantidad de RAM que utiliza ronda el 15 %, aproximadamente 1.23 GB.

Como se puede deducir, el algoritmo basado en AG computacionalmente es menos pesado que el algoritmo RL, por lo que en una primera instancia, se tendería a pensar que el AG es mejor computacionalmente que el RL. Así es en lo que a costes computacionales se refiere, el AG es mucho más liviano. Uno de los principales motivos por el que el RL tiene esta gran carga computacional es que los datos que entran en la NN pasan por más número de capas que en el caso del AG; en ellas (la primera parte de la DQN, la parte convolucional) se hace una serie de operaciones matriciales que conllevan gran coste computacional por la gran cantidad de operaciones que realiza.

7.3 ¿Qué algoritmo es mejor?

Frente a la comparativa que acaba de ser realizada, uno puede cuestionarse la utilidad del RL, pues tanto en términos de eficiencia y carga computacional el AG ha resultado superior. En primer lugar, es importante comentar el hecho de que existirían formas más óptimas de implementar el algoritmo basado en RL, pero, independientemente de eso, a pesar de que el RL presente un mayor coste computacional, tiene la virtud de que con ese

Figura 7.2: Carga computacional algoritmo basado en *Reinforcement Learning*

mismo entrenamiento realizado puede trasladarse a otro entorno parecido y seguir obteniendo buenos resultados [38]. Por el contrario el algoritmo basado en AG ha aprendido únicamente a jugar en el nivel mostrado en este trabajo.

Por tanto, podría concluirse que en función del objetivo final que se tenga, un algoritmo va a resultar más interesante que el otro. Si se quiere una solución lo más rápida posible y consumiendo pocos recursos computacionales, la mejor opción será el algoritmo basado en AG; en cambio, si lo que se busca es un entrenamiento que después pueda extrapolarse a otros entornos similares y siga obteniendo buenos resultados, entonces será más interesante el algoritmo basado en RL.

CAPÍTULO 8
Conclusión

Llegamos al último capítulo del trabajo; en él van a mostrarse las diferentes asignaturas que han resultado de indispensable utilidad para la planificación, abstracción, realización y ejecución de ambos algoritmos, y además se cerrará el presente trabajo mediante una conclusión final sobre dichos algoritmos y su extrapolación a otros entornos.

8.1 Relación con el grado

El presente trabajo ha utilizado conceptos y habilidades adquiridas en las distintas asignaturas a través de los distintos cursos del grado. Éstas son:

- **Fundamentos de Sistemas Operativos** (FSO), ha aportado soltura y conocimientos sobre el sistema operativo GNU/Linux, que es el sistema operativo que ha sido usado para el desarrollo y ejecución de los algoritmos.
- **Agentes Inteligentes** (AIN), aportando los conceptos de agente, entorno, juego, etc, que son fundamentales para el entendimiento de las bases del trabajo.
- **Sistemas de Almacenamiento y recuperación de información** (SAR), ha aportado la soltura y el uso del lenguaje de programación *Python*, además de las utilidades fundamentales como son: el acceso a ficheros para leer y escribir, almacenamiento de la información en listas y diccionarios, y ciertas funcionalidades características de dicho lenguaje.
- **Percepción** (PER) y **Aprendizaje automático** (APR) han aportado conceptos imprescindibles como el uso de las NN y distintas técnicas de ML.
- **Técnicas, entornos y aplicaciones de Inteligencia Artificial** (TIA), que nos ha aportado la base para entender los EA.
- **Sistemas Multimedia Interactivos e Inmersivos** (SMII) ha aportado soltura y conocimiento sobre el uso de la biblioteca *OpenCV* para el tratado de la imagen de entrada a la NN.

8.2 Cierre del trabajo

Tal y como se ha mostrado a lo largo del trabajo, los algoritmos evolutivos y el *Machine Learning* han sido los encargados de conseguir la meta que se propuso como objetivo en el trabajo, la de que el agente Mario completase el primer nivel. A pesar de que hay

una variación evidente de coste temporal y computacional entre ambos algoritmos, ésta ha sido justificada mediante la capacidad que tienen cada uno de estos algoritmos de extrapolación a otro entorno similar.

En el caso del presente trabajo, tal y como se muestra a lo largo del Capítulo 7, el algoritmo basado en AG es el que salió victorioso en lo que a eficiencia y coste computacional respecta, pero, el algoritmo basado en RL, a pesar de ser mucho más costoso que el anterior, tiene la posibilidad de, una vez entrenado, extrapolarse a otro entorno similar, como por ejemplo sería que Mario jugase en otro nivel cuyo entorno sea semejante al primero, o como por ejemplo a otro juego de características similares, como podría ser otro juego de la saga *Super Mario Bros*.

A pesar de que este trabajo ha ido enfocado al mundo de los videojuegos, los principios en los que están basados los dos algoritmos implementados podrían ser de especial interés en otros campos del conocimiento. En el caso de los AG, con la situación actual que ha generado la pandemia del *COVID-19*, podría implementarse un sistema que, en función de distintas variables como serían los distintos rangos de edad de la población, las distintas vacunas disponibles en el país y del presupuesto del que se disponga, se buscase una combinación óptima para minimizar tanto el coste como el tiempo de vacunación. Por otro lado, en el caso del RL, sería de especial interés si, por ejemplo, se implementase un simulador de conducción y circulación con físicas muy cercas a la realidad, para que en función de una DQN se entrenase al coche a realizar todas las acciones posibles en función del lo que recoja del entorno; posteriormente, tras un gran entrenamiento en el simulador, se podría implementar dentro de un automóvil real adaptado, para que fuese totalmente autónomo.

Finalmente, me gustaría concluir el presente trabajo expresando mi máxima satisfacción personal por el procedimiento seguido para la implementación de los algoritmos. A medida que iba leyendo información sobre las bases de éstos, me he ido dado cuenta de que actualmente hay una gran ramificación de conocimientos y de conceptos dentro de cada uno de ellos, por lo que hace que te envuelvas dentro de una atmósfera de conocimiento en el que, gracias a los conocimientos y aptitudes adquiridos a lo largo del grado, pueden entenderse y aplicarse de forma satisfactoria.

Bibliografía

[1] National Geographic. Breve historia visual de la inteligencia artificial, 05 de agosto de 2019. Consultado en https://www.nationalgeographic.com.es/ciencia/breve-historia-visual-inteligencia-artificial_14419.

[2] Marco Wiering y Martijn van Otterlo. *Least-Squares Methods for Policy Iteration*. Reinforcement Learning State-of-the-Art. pp 75-104. DOI 10.1007/978-3-642-27645-3. ISBN 978-3-642-27644-6

[3] Bingdong Li, Jinlong Li, Ke Tang y Xin Yao. *Many-Objective Evolutionary Algorithms: A Survey*. ACM Comput. Surv. 48, 1, Article 13 (September 2015), pp 1-5. DOI: http://dx.doi.org/10.1145/2792984

[4] TechCrunch. Chris Nicholson, 17 junio 2021. *Deep reinforcement learning will transform manufacturing as we know it*. Consultado en https://techcrunch.com/2021/06/17/deep-reinforcement-learning-will-transform-manufacturing-as-we-know-it/.

[5] Gregory S. Hornby, Al Globus, Derek S. Linden y Jason D. Lohn. *Automated Antenna Design with Evolutionary Algorithms*. NASA Ames Research Center (2006). pp 1-5.

[6] Facultat d'Informàtica de Barcelona. Historia de los videojuegos Consultado en https://www.fib.upc.edu/retro-informatica/historia/videojocs.html.

[7] Julian Togelius. *AI Researchers, Video Games Are Your Friends!* Computational Intelligence, Springer International Publishing 2017. pp 3-18. DOI: 10.1007/978-3-319-48506-5_1. ISBN: 978-3-319-48506-5.

[8] Britannica. Go. Consultado en https://www.britannica.com/topic/go-game.

[9] Scott R. Granter, Andrew H. Beck y David J. Papke. *AlphaGo, Deep Learning, and the Future of the Human Microscopist*. Arch Pathol Lab Med (2017) 141 (5): 619–621. DOI: 10.5858/arpa.2016-0471-ED

[10] M. Fu. *AlphaGo and Monte Carlo tree search: The simulation optimization perspective.* 2016 Winter Simulation Conference (WSC) (2016). pp 659-670.

[11] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran,Thore Graepel, Timothy Lillicrap, Karen Simonyan y Demis Hassabis, *A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play.* Science 07 Dec 2018. Vol. 362, Issue 6419, pp. 1140-1144. DOI: 10.1126/science.aar6404

[12] HTML5 Genetic Algorithm 2D car thingy. Consultado en https://rednuht.org/genetic_cars_2/

[13] Ali Ghaheri, Saeed Shoar, Mohammad Naderan y Sayed Shahabuddin Hoseini. *The Applications of Genetic Algorithms in Medicine.* Oman Medical Journal (2015), Vol. 30. pp 406–416. DOI: 10.5001/omj.2015.82.

[14] Anthony Ede John Oluwatobi Oluwafemi y Solomon Olakunle Oyebisi. *Structural analysis of a genetic algorithm optimized steel truss structure according to BS 5950.* International Journal of Civil Engineering and Technology, August 2018. pp 358-364. ISSN Online: 0976-6316.

[15] Pariwat Ongsulee. *Artificial Intelligence, Machine Learning and Deep Learning*. 2017 Fifteenth International Conference on ICT and Knowledge Engineering. pp 1-6. DOI: 10.1109/ICTKE.2017.8259629.

[16] Gee-Wah Ng y Wang Chi Leung. *Strong Artificial Intelligence and Consciousness*. Journal of Artificial Intelligence and Consciousness. Marzo 2020. pp 63-72. DOI: 10.1142/S2705078520300042

[17] Txema Rodríguez. Machine Learning y Deep Learning: cómo entender las claves del presente y futuro de la inteligencia artificial. 16 Octubre 2020. Consultado en https://www.xataka.com/robotica-e-ia/machine-learning-y-deep-learning-como-entender-las-claves-del-presente-y-futuro-de-la

[18] Sunil Ray. Commonly used Machine Learning Algorithms (with Python and R Codes) 9 Septiembre 2017. Consultado en https://www.analyticsvidhya.com/blog/2017/09/common-machine-learning-algorithms/.

[19] Quanxue Gaoa, Yunfang Huanga, Xinbo Gaoab, Weiguo Shena y Hailin Zhanga. *A novel semi-supervised learning for face recognition*. Neurocomputing Volume 152, 25 Marzo 2015. pp 69-76. DOI: 10.1016/j.neucom.2014.11.018

[20] Yuke Zhu; Roozbeh Mottaghi; Eric Kolve; Joseph J. Lim; Abhinav Gupta; Li Fei-Fei; Ali Farhad. *Target-driven visual navigation in indoor scenes using deep reinforcement learning*. IEEE International Conference on Robotics and Automation (ICRA), 2017. pp. 3357-3364. DOI: 10.1109/ICRA.2017.7989381.

[21] Bing Xue, Mengjie Zhang, Will N. Browne y Xin Yao. *A Survey on Evolutionary Computation Approaches to Feature Selection.* IEEE Transactions on Evolutionary Computation (Volume: 20, Issue: 4, Aug. 2016). pp 606-626. DOI: 10.1109/TEVC.2015.2504420

[22] Algoritmo evolutivo. Wikipedia. Consultado en https://es.wikipedia.org/wiki/Algoritmo_evolutivo

[23] Sourabh Katoch, Sumit Singh Chauhan y Vijay Kumar. *A review on genetic algorithm: past, present, and future.* Multimed Tools Appl 80 (2021). pp 8091–8126. DOI: 10.1007/s11042-020-10139-6

[24] *OpenIA*. Wikipedia. Consultado en https://es.wikipedia.org/wiki/OpenAI

[25] *Gym Retro*. 25 Mayo 2018. Consultado en https://openai.com/blog/gym-retro/

[26] *Documentación gym-retro*. Consultado en https://retro.readthedocs.io/en/latest/getting_started.html

[27] Vijini Mallawaarachchi *Introduction to Genetic Algorithms — Including Example Code*. 8 Julio 2017. Consultado en https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3.

[28] Anita Thengade y Rucha Dondal. *Genetic Algorithm - Survey Paper*. 7 Abril 2012. MPGI National Multi Conference 2012 (MPGINMC-2012). pp 1-5. (IJCA)ISSN: 0975 - 8887.

[29] Mat Buckland y André LaMothe. *AI Techniques for game programming*. En Evolving Neural Network Topology pp. 358-368. ISBN-10: 193184108X.

[30] Kenneth O. Stanley and Risto Miikkulainen. *Evolving Neural Networks through Augmenting Topologies*. The MIT Press Journals. 1 Junio 2002. pág 11.

[31] Documentación de OpenCV para *Python*. *Open Source Computer Vision*. Consultado en: https://docs.opencv.org/4.5.2/d6/d00/tutorial_py_root.html.

[32] Leslie Pack Kaelbling, Michael L. Littman y Andrew W. Moore. *Reinforcement Learning: A Survey*. Journal of Artificial Intelligence Research 4 (1996). pp. 238-240.

[33] Zhang, Yu-Fen and Zhang, Qun-Feng and Yu, Rui-Hua. *Markov property of Markov chains and its test*. 2010 International Conference on Machine Learning and Cybernetics (2010). pp. 1864-1867. 10.1109/ICMLC.2010.5580952

[34] Christopher J.C.H. Watkins y Peter Dayan. *Technical note Q-Learning*. Machine Learning, 8, 279-292 (1992). pp. 1-4.

[35] Nitchakun Kantasewi, Somying Thainimit, Sanparith Marukatat y Okumura Manabu. *Multi Q-Table Q-Learning*. 2019 10th International Conference of Information and Communication Technology for Embedded Systems (IC-ICTES). pp. 2-3. DOI: 10.1109/ICTEmSys.2019.8695963.

[36] Belfor. *DQN creada para el videojuego Sonic the hedgehog*. Sonic-Deep-Q-Learning/SonicAgent.py. https://github.com/Belfor/Sonic-Deep-Q-Learning/blob/master/SonicAgent.py

[37] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot y Nando de Freitas. *Dueling Network Architectures for Deep Reinforcement Learning*. Google Deep Mind (2015). pp. 4-6. DOI: 1511.06581.

[38] Charles Packer, Katelyn Gao, Jernej Kos, Philipp K., Vladlen Koltun y Dawn Song *Assessing Generalization in Deep Reinforcement Learning*. Technical Report, 15 Marzo 2019. pp 1-10. DOI: 1810.12282

APÉNDICE A
Información complementaria

A.1 Hardware utilizado para el entrenamiento

Para el entrenamiento de ambos algoritmos se ha utilizado un ordenador MSI CX61 2QC cuyo hardware a destacar es:

- Intel Core i7-4712MQ 2.3GHz, 4 núcleos físicos (8 núcleos lógicos).
- 8GB RAM 1600MHz.
- NVIDIA GeForce® 920M.

La ventaja que ofrece este dispositivo es que, aparte de que se dispone de 8 núcleos lógicos, su tarjeta gráfica de *Nvidia* es compatible con la tecnología *CUDA*, que consiste es una plataforma de computación paralela y un modelo de programación que permite incrementos drásticos en el rendimiento de computación al aprovechar la potencia de la unidad de procesamiento de gráficos. Además de esto, *Nvidia* ofrece *Nvidia cuDNN*, una biblioteca de primitivas acelerada por *GPU* para NN profundas; además, proporciona implementaciones altamente ajustadas para rutinas estándar como convolución hacia adelante y hacia atrás, agrupación, normalización y capas de activación. Por tanto, se optó por la instalación de estas dos ayudas para optimizar el proceso de aprendizaje de Mario.

A.2 Herramientas software utilizadas

Tanto para el desarrollo como para la ejecución de los algoritmos se han utilizado las siguientes herramientas software:

- *KDE Neon* basado en *Ubuntu 20.04*, como sistema operativo.
- *Python* en su versión 3.8.
- Bibliotecas:
  - *TensorFlow* versión 2.3.1.
  - *NEAT* versión 0.4.1.
  - *NEAT-Python* versión 0.92.
  - *OpenCV-Python* versión 4.5.1.48.
  - *gym-retro* versión 0.8.0.

A.3 Códigos completos

```python
import retro

def main():
    env = retro.make(game='SuperMarioWorld-Snes',state='YoshiIsland2.state')
    obs = env.reset()

    print("SNES gamepad buttons -> {}".format(env.buttons))
    print("Observation space -> {}".format(env.observation_space))
    print("Action space -> {}".format(env.action_space))

    while True:
        obs, rew, done, info = env.step(env.action_space.sample())

        if done:
            obs = env.reset()
        env.render()

    env.close()


if __name__ == "__main__":
    main()
```

Listing A.1: Código básico inicial *gym-retro*.

```
[NEAT]
fitness_criterion     = max
fitness_threshold     = 100000
pop_size              = 100
reset_on_extinction   = True

[DefaultGenome]
# node activation options
activation_default      = sigmoid
activation_mutate_rate  = 0.05
activation_options      = sigmoid gauss
#abs clamped cube exp gauss hat identity inv log relu sigmoid sin softplus
    square tanh

# node aggregation options
aggregation_default     = random
aggregation_mutate_rate = 0.05
aggregation_options     = sum product min max mean median maxabs

# node bias options
bias_init_mean          = 0.05
bias_init_stdev         = 1.0
bias_max_value          = 30.0
bias_min_value          = -30.0
bias_mutate_power       = 0.5
bias_mutate_rate        = 0.7
bias_replace_rate       = 0.1

# genome compatibility options
compatibility_disjoint_coefficient = 1.0
compatibility_weight_coefficient   = 0.5

# connection add/remove rates
conn_add_prob           = 0.5
conn_delete_prob        = 0.1

# connection enable options
```

```
enabled_default         = True
enabled_mutate_rate     = 0.2

feed_forward            = False
#initial_connection      = unconnected
initial_connection      = partial_nodirect 0.5

# node add/remove rates
node_add_prob           = 0.5
node_delete_prob        = 0.5

# network parameters
num_hidden              = 0
num_inputs              = 896
num_outputs             = 12

# node response options
response_init_mean      = 1.0
response_init_stdev     = 0.05
response_max_value      = 30.0
response_min_value      = -30.0
response_mutate_power   = 0.1
response_mutate_rate    = 0.75
response_replace_rate   = 0.1

# connection weight options
weight_init_mean        = 0.1
weight_init_stdev       = 1.0
weight_max_value        = 30
weight_min_value        = -30
weight_mutate_power     = 0.5
weight_mutate_rate      = 0.8
weight_replace_rate     = 0.1

[DefaultSpeciesSet]
compatibility_threshold = 2.5

[DefaultStagnation]
species_fitness_func = max
max_stagnation       = 50
species_elitism      = 0

[DefaultReproduction]
elitism            = 1
survival_threshold = 0.3
```

Listing A.2: Archivo de configuración *config-feedforward*.

```
import retro
import numpy as np
import cv2
import neat
import pickle
from datetime import datetime
import time

env = retro.make('SuperMarioWorld-Snes', 'YoshiIsland2.state')
imageArray = []

cv2.namedWindow('ResizeRGB', cv2.WINDOW_NORMAL)
cv2.namedWindow('ResizeGrayScale', cv2.WINDOW_NORMAL)


def eval_genomes(genomes, config):
```

```
17
18      for genome_id, genome in genomes:
19          obs = env.reset()
20
21          inputY, inputX, numChannels = env.observation_space.shape
22          inputX = int(inputX/8)
23          inputY = int(inputY/8)
24
25          net = neat.nn.recurrent.RecurrentNetwork.create(genome, config)
26
27          fitnessMax = 0
28          fitnessCurrent = 0
29          frameCounter = 0
30          xPos = 0
31          xPosMax = 0
32
33          done = False
34
35          while not done:
36
37              env.render()
38
39              obs = cv2.resize(obs, (inputX, inputY))
40              cv2.imshow('ResizeRGB',obs)
41              obs = cv2.cvtColor(obs, cv2.COLOR_BGR2GRAY)
42              cv2.imshow('ResizeGrayScale',obs)
43
44              imageArray = np.ndarray.flatten(obs)
45
46              nnOutput = net.activate(imageArray)
47
48              obs, rew, done, info = env.step(nnOutput)
49
50              xPos = info['x']
51
52              if xPos > xPosMax:
53                  xPosMax = xPos
54                  frameCounter = 0
55                  fitnessCurrent += 1
56              else:
57                  frameCounter += 1
58
59              if info['endOfLevel']:
60                  fitnessCurrent += 10000
61                  done = True
62
63              fitnessCurrent += rew
64
65              if done or frameCounter == 100:
66                  done = True
67                  print(genome_id, fitnessCurrent)
68
69              genome.fitness = fitnessCurrent
70              cv2.waitKey(1)
71
72  def main():
73      config = neat.Config(neat.DefaultGenome, neat.DefaultReproduction,
74                           neat.DefaultSpeciesSet, neat.DefaultStagnation,
75                           'config-feedforward')
76
77      p = neat.Population(config)
78      p.add_reporter(neat.StdOutReporter(True))
79      stats = neat.StatisticsReporter()
80      p.add_reporter(stats)
```

```
81
82      start_time = time.time()
83      winner = p.run(eval_genomes)
84      finish_time = time.time()
85
86      with open('winner.pkl', 'wb') as output:
87          pickle.dump(winner, output, 1)
88
89      cv2.destroyAllWindows()
90
91
92  if __name__ == '__main__':
93      main()
```

Listing A.3: Código completo algoritmo basado en AG.

```
1  import gym
2  import retro
3  import numpy as np
4  import random
5  import copy
6  import tensorflow as tf
7  import cv2
8  import time
9  import sys
10
11 from datetime import datetime
12 from tensorflow.keras.models import Sequential
13 from tensorflow.keras.layers import Dense, Flatten, Convolution2D
14 from tensorflow.keras.optimizers import Adam
15 from rl.agents import DQNAgent
16 from rl.memory import SequentialMemory
17 from rl.policy import LinearAnnealedPolicy, EpsGreedyQPolicy
18 from rl.callbacks import ModelIntervalCheckpoint
19 from gym.spaces import Box
20
21 dateTimeNow = datetime.now()
22 episodeReward = 0
23 xPos = 0
24 xPosMax = 0
25 frameCounter = 1
26 numIteracion = 0
27 showImageTransformation = False
28
29 if len(sys.argv) == 2 and (sys.argv[1] == '-v' or sys.argv[1] == '-V'):
30     showImageTransformation = True
31     cv2.namedWindow('image', cv2.WINDOW_NORMAL)
32     cv2.namedWindow('image2', cv2.WINDOW_NORMAL)
33     cv2.namedWindow('image3', cv2.WINDOW_NORMAL)
34     cv2.namedWindow('image4', cv2.WINDOW_NORMAL)
35
36 f_real = open("training_real_stats_"+str(dateTimeNow)+".csv","w+")
37 f_real.write("Iteracion" + ";" + "Reward" + "\n")
38
39 def processFrame(frame, shape=(100,100)):
40     frame = frame.astype(np.uint8)
41     if showImageTransformation: cv2.imshow('image',frame)
42
43     frame = cv2.cvtColor(frame, cv2.COLOR_RGB2GRAY)
44     if showImageTransformation: cv2.imshow('image2',frame)
45
46     frame = frame[40:40+224, :224]
47     if showImageTransformation: cv2.imshow('image3',frame)
48
```

```
    frame = cv2.resize(frame, shape, interpolation=cv2.INTER_NEAREST)
    if showImageTransformation: cv2.imshow('image4',frame)

    frame = frame.reshape((*shape, 1))

    if showImageTransformation: cv2.waitKey(1)

    return frame

class ObservationDiscretizer(gym.ObservationWrapper):
    def __init__(self, env):
        super().__init__(env)

    def step(self, act):
        global episodeReward
        global xPosMax
        global xPos
        global frameCounter
        global numIteracion
        global f_real

        obs, rew, done, info = super().step(act)
        episodeReward += rew

        xPos = info['x']

        if (xPosMax < xPos):
            frameCounter = 1
            xPosMax = xPos
            episodeReward += 5
        else:
            frameCounter += 1
            episodeReward -= 1

        if frameCounter %300 == 0:
            episodeReward -= 10

        if frameCounter == 1200:
            episodeReward -= 100
            done = True

        if info['endOfLevel']:
            #print("FIN NIVEL")
            episodeReward += 30000
            done = True

        if info['dead'] == 0:
            #print("MUERTO")
            episodeReward -= 1000
            done = True

        rew = episodeReward

        if done:
            numIteracion += 1
            f_real.write(str(numIteracion) + ";" + str(rew) + "\n")

        #print("Total acumulado: " + str(rew) + " Valor frame counter: " + str(
            frameCounter))

        return obs, rew, done, info

    def observation(self, obs):
        obs = processFrame(obs)
```

```
112         return obs
113
114 class MarioObservationDiscretizer(ObservationDiscretizer):
115     def __init__(self, env):
116         super().__init__(env)
117
118 class ActionDiscretizer(gym.ActionWrapper):
119     def __init__(self, env, combos):
120         super(ActionDiscretizer,self).__init__(env)
121         self.env = env
122         assert isinstance(env.action_space, gym.spaces.MultiBinary)
123         buttons = env.unwrapped.buttons
124         self._decode_discrete_action = []
125         for combo in combos:
126             arr = np.array([False] * env.action_space.n)
127             for button in combo:
128                 arr[buttons.index(button)] = True
129             self._decode_discrete_action.append(arr)
130         self.action_space = gym.spaces.Discrete(len(self.
            _decode_discrete_action))
131
132     def reset(self, **kwargs):
133         global episodeReward
134         global xPosMax
135         global xPos
136         global frameCounter
137
138         xPos = 0
139         xPosMax = 0
140         episodeReward = 0
141         frameCounter = 1
142
143         return self.env.reset(**kwargs)
144
145     def action(self, act):
146         return self._decode_discrete_action[act].copy()
147
148 class MarioActionDiscretizer(ActionDiscretizer):
149     def __init__(self, env):
150         obs= env.reset()
151         super().__init__(env, combos=[['LEFT'], ['RIGHT'], ['RIGHT', 'B'], ['
            RIGHT', 'Y'], ['B'], ['A']])
152
153
154 def build_model(height, width, channels, actions):
155     model = Sequential()
156
157     model.add(Convolution2D(32, (8,8), strides=(4,4), activation='relu',
        input_shape=(3, height, width, 1)))
158     model.add(Convolution2D(64, (4,4), strides=(2,2), activation='relu'))
159     model.add(Convolution2D(64, (3,3), activation='relu'))
160     model.add(Flatten())
161
162     model.add(Dense(512, activation='relu'))
163     model.add(Dense(256, activation='relu'))
164     model.add(Dense(actions, activation='linear'))
165
166     return model
167
168 def build_agent(model, actions):
169     policy = LinearAnnealedPolicy(EpsGreedyQPolicy(), attr='eps', value_max=1.,
          value_min=.1, value_test=.2, nb_steps=10000)
170
171     memory = SequentialMemory(limit=1000, window_length=3)
```

```
172
173     dqnAgent = DQNAgent(model=model, memory=memory, policy=policy,
174                         enable_dueling_network=True, dueling_type='avg',
175                         nb_actions=actions, nb_steps_warmup=100
176                        )
177
178     return dqnAgent
179
180 def main():
181     global f_real
182     global numIteracion
183
184     env = MarioActionDiscretizer(retro.make(game='SuperMarioWorld-Snes', state=
            'YoshiIsland2.state', record='./records/'))
185     env = MarioObservationDiscretizer(env)
186
187     height, width, channels = 100, 100, 1
188     actions = env.action_space.n
189
190     model = build_model(height, width, channels, actions)
191
192     agent = build_agent(model, actions)
193     agent.compile(Adam(lr=1e-4))
194
195     '''
196         Creamos un callback para ir guardando checkpoints con los pesos
197     '''
198     filepath='./checkpoints/'
199     checkpoint = ModelIntervalCheckpoint(filepath,interval=10)
200
201     time.sleep(5)
202
203     print("\n\n-> ENTRENAMIENTO DEL AGENTE")
204     start_time = time.time()
205     training_stats = agent.fit(env, nb_steps=1000000, visualize=False, verbose
            =2)
206     finish_time = time.time()
207     f = open("training_stats_"+str(dateTimeNow)+".txt","w+")
208     f.write(str(training_stats.params))
209     f.write("\n##########################################\n")
210     f.write(str(training_stats.history.keys()))
211     f.write("\n##########################################\n")
212     f.write(str(training_stats.history))
213     f.write("\n##########################################\n")
214     f.write("Tiempo entrenando: " + str(finish_time - start_time))
215     f.close()
216     f_real.close()
217
218     print("\n\n-> VALIDACION DEL AGENTE")
219     numIteracion = 0
220     f_real = open("validating_real_stats_"+str(dateTimeNow)+".csv","w+")
221     f_real.write("Iteracion" + ";" + "Reward" + "\n")
222     start_time = time.time()
223     scores = agent.test(env, nb_episodes=10, visualize=False)
224     finish_time = time.time()
225     print("\n\n-> GUARDANDO LOS SCORES EN FICHERO")
226     f = open("validating_stats_"+str(dateTimeNow)+".txt","w+")
227     f.write(str(scores.params))
228     f.write("\n##########################################\n")
229     f.write(str(scores.history.keys()))
230     f.write("\n##########################################\n")
231     f.write(str(scores.history))
232     f.write("\n##########################################\n")
233     f.write("Tiempo validando: " + str(finish_time - start_time))
```

```
234     f.close()
235     f_real.close()
236 
237     env.close()
238 
239 if __name__ == '__main__':
240     main()
```

Listing A.4: Código completo algoritmo basado en RL versión MR.

```
1  import gym
2  import retro
3  import numpy as np
4  import random
5  import copy
6  import tensorflow as tf
7  import cv2
8  import time
9  import sys
10 
11 from datetime import datetime
12 from tensorflow.keras.models import Sequential
13 from tensorflow.keras.layers import Dense, Flatten, Convolution2D
14 from tensorflow.keras.optimizers import Adam
15 from rl.agents import DQNAgent
16 from rl.memory import SequentialMemory
17 from rl.policy import LinearAnnealedPolicy, EpsGreedyQPolicy
18 from rl.callbacks import ModelIntervalCheckpoint
19 from gym.spaces import Box
20 
21 dateTimeNow = datetime.now()
22 episodeReward = 0
23 xPos = 0
24 xPosMax = 0
25 frameCounter = 1
26 numIteracion = 0
27 showImageTransformation = False
28 
29 if len(sys.argv) == 2 and (sys.argv[1] == '-v' or sys.argv[1] == '-V'):
30     showImageTransformation = True
31     cv2.namedWindow('image', cv2.WINDOW_NORMAL)
32     cv2.namedWindow('image2', cv2.WINDOW_NORMAL)
33     cv2.namedWindow('image3', cv2.WINDOW_NORMAL)
34     cv2.namedWindow('image4', cv2.WINDOW_NORMAL)
35 
36 f_real = open("training_real_stats_"+str(dateTimeNow)+".csv","w+")
37 f_real.write("Iteracion" + ";" + "Reward" + "\n")
38 
39 def processFrame(frame, shape=(100,100)):
40     frame = frame.astype(np.uint8)
41     if showImageTransformation: cv2.imshow('image',frame)
42 
43     frame = cv2.cvtColor(frame, cv2.COLOR_RGB2GRAY)
44     if showImageTransformation: cv2.imshow('image2',frame)
45 
46     frame = frame[40:40+224, :224]
47     if showImageTransformation: cv2.imshow('image3',frame)
48 
49     frame = cv2.resize(frame, shape, interpolation=cv2.INTER_NEAREST)
50     if showImageTransformation: cv2.imshow('image4',frame)
51 
52     frame = frame.reshape((*shape, 1))
53 
54     if showImageTransformation: cv2.waitKey(1)
```

```
55
56      return frame
57
58  class ObservationDiscretizer(gym.ObservationWrapper):
59      def __init__(self, env):
60          super().__init__(env)
61
62      def step(self, act):
63          global episodeReward
64          global xPosMax
65          global xPos
66          global frameCounter
67          global numIteracion
68          global f_real
69
70          obs, rew, done, info = super().step(act)
71          episodeReward += rew
72
73          xPos = info['x']
74
75          if (xPosMax < xPos):
76              frameCounter = 1
77              xPosMax = xPos
78              episodeReward += 5
79          else:
80              frameCounter += 1
81              episodeReward = float(int(episodeReward * 0.999)) #Restamos el 0.1%
82
83          if frameCounter %300 == 0:
84              episodeReward = float(int(episodeReward * 0.95)) #Restamos el 5%
85
86          if frameCounter == 1200:
87              #print("RESETEAR")
88              done = True
89
90          if info['endOfLevel']:
91              #print("FIN NIVEL")
92              episodeReward += 30000
93              done = True
94
95          if info['dead'] == 0:
96              #print("MUERTO")
97              episodeReward = int(episodeReward * 0.70) #Restamos el 30%
98              done = True
99
100         rew = episodeReward
101
102         if done:
103             numIteracion += 1
104             f_real.write(str(numIteracion) + ";" + str(rew) + "\n")
105
106         #print("Total acumulado: " + str(rew) + " Valor frame counter: " + str(
                frameCounter))
107
108         return obs, rew, done, info
109
110     def observation(self, obs):
111         obs = processFrame(obs)
112         return obs
113
114 class MarioObservationDiscretizer(ObservationDiscretizer):
115     def __init__(self, env):
116         super().__init__(env)
117
```

```
class ActionDiscretizer(gym.ActionWrapper):
    def __init__(self, env, combos):
        super(ActionDiscretizer,self).__init__(env)
        self.env = env
        assert isinstance(env.action_space, gym.spaces.MultiBinary)
        buttons = env.unwrapped.buttons
        self._decode_discrete_action = []
        for combo in combos:
            arr = np.array([False] * env.action_space.n)
            for button in combo:
                arr[buttons.index(button)] = True
            self._decode_discrete_action.append(arr)
        self.action_space = gym.spaces.Discrete(len(self.
            _decode_discrete_action))

    def reset(self, **kwargs):
        global episodeReward
        global xPosMax
        global xPos
        global frameCounter

        xPos = 0
        xPosMax = 0
        episodeReward = 0
        frameCounter = 1

        return self.env.reset(**kwargs)

    def action(self, act):
        return self._decode_discrete_action[act].copy()

class MarioActionDiscretizer(ActionDiscretizer):
    def __init__(self, env):
        obs= env.reset()
        super().__init__(env, combos=[['LEFT'], ['RIGHT'], ['RIGHT', 'B'], ['
            RIGHT', 'Y'], ['B'], ['A']])


def build_model(height, width, channels, actions):
    model = Sequential()

    model.add(Convolution2D(32, (8,8), strides=(4,4), activation='relu',
        input_shape=(3, height, width, 1)))
    model.add(Convolution2D(64, (4,4), strides=(2,2), activation='relu'))
    model.add(Convolution2D(64, (3,3), activation='relu'))
    model.add(Flatten())

    model.add(Dense(512, activation='relu'))
    model.add(Dense(256, activation='relu'))
    model.add(Dense(actions, activation='linear'))

    return model

def build_agent(model, actions):
    policy = LinearAnnealedPolicy(EpsGreedyQPolicy(), attr='eps', value_max=1.,
         value_min=.1, value_test=.2, nb_steps=10000)

    memory = SequentialMemory(limit=1000, window_length=3)

    dqnAgent = DQNAgent(model=model, memory=memory, policy=policy,
                        enable_dueling_network=True, dueling_type='avg',
                        nb_actions=actions, nb_steps_warmup=100
                       )

```

```
178     return dqnAgent
179
180 def main():
181     global f_real
182     global numIteracion
183
184     env = MarioActionDiscretizer(retro.make(game='SuperMarioWorld-Snes', state=
            'YoshiIsland2.state', record='./records/'))
185     env = MarioObservationDiscretizer(env)
186
187     height, width, channels = 100, 100, 1
188     actions = env.action_space.n
189
190     model = build_model(height, width, channels, actions)
191
192     agent = build_agent(model, actions)
193     agent.compile(Adam(lr=1e-4))
194
195     '''
196         Creamos un callback para ir guardando checkpoints con los pesos
197     '''
198     filepath='./checkpoints/'
199     checkpoint = ModelIntervalCheckpoint(filepath,interval=10)
200
201     time.sleep(5)
202
203     print("\n\n-> ENTRENAMIENTO DEL AGENTE")
204     start_time = time.time()
205     training_stats = agent.fit(env, nb_steps=1000000, visualize=False, verbose
            =2)
206     finish_time = time.time()
207     f = open("training_stats_"+str(dateTimeNow)+".txt","w+")
208     f.write(str(training_stats.params))
209     f.write("\n##############################################\n")
210     f.write(str(training_stats.history.keys()))
211     f.write("\n##############################################\n")
212     f.write(str(training_stats.history))
213     f.write("\n##############################################\n")
214     f.write("Tiempo entrenando: " + str(finish_time - start_time))
215     f.close()
216     f_real.close()
217
218     print("\n\n-> VALIDACION DEL AGENTE")
219     numIteracion = 0
220     f_real = open("validating_real_stats_"+str(dateTimeNow)+".csv","w+")
221     f_real.write("Iteracion" + ";" + "Reward" + "\n")
222     start_time = time.time()
223     scores = agent.test(env, nb_episodes=10, visualize=False)
224     finish_time = time.time()
225     print("\n\n-> GUARDANDO LOS SCORES EN FICHERO")
226     f = open("validating_stats_"+str(dateTimeNow)+".txt","w+")
227     f.write(str(scores.params))
228     f.write("\n##############################################\n")
229     f.write(str(scores.history.keys()))
230     f.write("\n##############################################\n")
231     f.write(str(scores.history))
232     f.write("\n##############################################\n")
233     f.write("Tiempo validando: " + str(finish_time - start_time))
234     f.close()
235     f_real.close()
236
237     env.close()
238
239 if __name__ == '__main__':
```

```
240 main()
```

Listing A.5: Código completo algoritmo basado en RL versión MP.

APÉNDICE B

Tablas de las estadísticas obtenidas para el Algoritmo Genético

En este apéndice se procede a adjuntar una serie de tablas con los datos obtenidos a partir de la ejecución del algoritmo basado en AG. Cada tabla representa los datos de la ejecución de cada una de las distintas poblaciones. Cada tabla está subdividida en columnas con las etiquetas de:

- *Media fit*, una media del valor de la *fitness function* para esa generación.
- *Max fit*, el valor máximo que ha tomado la *fitness function* para esa generación.
- *Tiempo*, en segundos que ha tomado esa generación.
- *Veces meta*, que indica las veces que se ha superado el nivel para esa generación.

RESTO DE PAGÍNA EN BLANCO INTENCIONALMENTE
TABLAS A PARTIR DE LA SIGUIENTE PÁGINA

* * *

	10 genomas								
	Media fit	Max fit	Tiempo	Veces meta		Media fit	Max fit	Tiempo	Veces meta
0	17,9	170	33,323	0	36	9880,6	31186	99,888	1
1	53,6	182	42,702	0	37	7719,7	31186	100,158	1
2	5708,9	19139	116,92	0	38	17760,8	31549	177,446	2
3	2977,1	19139	102,837	0	39	12583,6	31549	114,762	1
4	7188,6	19182	139,232	0	40	8508,6	31549	94,799	1
5	6568,7	19539	147,038	0	41	6921	31549	118,235	1
6	6725,6	20182	152,983	0	42	9093,1	31549	110,248	1
7	9683,6	21136	106,825	0	43	7510,3	31549	99,819	1
8	9786,8	21136	127,713	0	44	9551,2	31549	109,239	1
9	11315,6	21136	145,01	0	45	7410,9	31549	111,457	1
10	5407,7	21136	72,322	0	46	9369,3	31549	109,763	1
11	6522,5	21136	116,118	0	47	12101,2	31549	100,833	1
12	4582,3	21136	112,707	0	48	9914,8	31549	126,773	1
13	7160,7	21136	128,338	0	49	6260,6	31549	103,84	1
14	6789,1	21136	121,882	0	50	8187,8	31549	98,588	1
15	7503,4	21473	194,613	0	51	9508,8	31549	92,654	1
16	8030,2	21473	129,057	0	52	12270,6	31549	106,98	1
17	11953,5	21473	198,617	1	53	14679,2	31549	147,547	2
18	7678	31186	181,298	1	54	9784,3	31549	119,592	2
19	13454,3	31186	195,721	2	55	10759,4	31549	99,175	1
20	6360,4	31186	126,982	1	56	7936,1	31549	118,506	1
21	7333,3	31186	127,85	2	57	11109,8	31549	151,444	1
22	7907,7	31186	123,977	1	58	9803,8	31549	97,125	1
23	10639,4	31186	173,998	1	59	8569	31549	141,443	1
24	13527,7	31186	146,652	1	60	7712,2	31549	123,576	2
25	9372,1	31186	133,405	1	61	7961,9	31549	91,869	1
26	5317,8	31186	74,889	1	62	7062	31549	118,323	1
27	10909,4	31186	120,394	1	63	10375,3	31549	108,258	1
28	10259	31186	135,932	1	64	7541,9	31549	89,505	1
29	10651,9	31186	135,033	1	65	7293,3	31549	100,029	1
30	13530,8	31186	138,785	1	66	7310,6	31549	122,079	1
31	13681	31186	128,31	2	67	6639,8	31549	97,056	2
32	13397,6	31186	128,867	1	68	8239,9	31549	103,956	1
33	9517,4	31186	130,416	1	69	6623,9	31549	119,45	1
34	9853,8	31186	120,91	1	70	5201,1	31549	91,888	1
35	6055,6	31186	98,102	1					

Tabla B.1: Tabla de estadísticas de la población de 10 genomas.

	30 genomas								
	Media fit	Max fit	Tiempo	Veces meta		Media fit	Max fit	Tiempo	Veces meta
0	208,36667	1824	115,172	0	36	3489,25806	23660	278,821	1
1	935,56667	4699	286,368	0	37	4436,53333	23660	298,159	1
2	1893,19355	19177	364,517	0	38	4417,76667	23660	215,712	1
3	2641,9	19177	381,446	0	39	4124,62069	23660	272,822	1
4	3200,76667	19177	313,759	0	40	5991,1	23660	284,076	2
5	2995,74194	19177	263,994	0	41	6359,26667	23660	345,778	1
6	3443,8	19177	308,807	0	42	5862,03333	23660	290,139	1
7	2861,68966	19177	252,884	0	43	6703,2	23660	304,34	1
8	2640,8	19177	308,259	0	44	5308,29032	24960	274,69	1
9	2816,74194	19177	315,638	0	45	7796,08527	24960	236,993	0
10	3172,1	19177	331,704	0	46	5865,24138	24960	320,766	1
11	2589,5	19589	261,063	0	47	6360,33333	31586	313,932	1
12	1981,93333	19589	242,539	0	48	4961,32258	31586	391,906	1
13	2965,87097	19589	273,761	0	49	7218,86667	31586	460,322	1
14	3049,96774	19589	252,592	0	50	4166,86207	31586	360,918	1
15	2629,13333	19589	222,366	0	51	5217,51724	31586	493,341	1
16	2975,7	19589	231,484	0	52	7106,03448	31586	582,54	1
17	2055,23333	19589	209,747	0	53	5356,41379	31586	562,109	2
18	3765,9	19589	300,386	0	54	6704,33333	31586	594,388	1
19	2980,8	19589	266,003	0	55	3116,67742	31586	430,642	1
20	3173,8	19589	257,722	1	56	5293,09677	31586	501,747	3
21	2703,65517	19589	284,879	1	57	4821,46667	31586	566,487	1
22	2971,83333	19589	239,009	1	58	3724,63333	31586	485,537	1
23	3579,83333	20105	265,976	0	59	5155,16129	31586	547,2	2
24	3189,23333	20105	267,674	1	60	6278,06667	31586	505,953	1
25	3200,06452	20105	213,991	0	61	5413,13333	31586	448,055	1
26	3975,3	20105	280,736	0	62	4873,1	31586	349,987	1
27	5478,43333	20105	316,569	0	63	6693,93333	31586	346,067	2
28	4435,63	19589	234,512	0	64	4565,5	31586	222,974	1
29	4778,93333	20105	279,393	0	65	4699,68966	31586	240,794	2
30	3807,2	20105	270,325	0	66	4745,32258	31586	277,089	1
31	3563,56667	20105	204,309	0	67	4768,76667	31586	283,228	1
32	5100,35484	20105	243,675	0	68	4963,44828	31586	249,448	2
33	4278,06667	20105	250,09	0	69	4682,2	31586	332,437	2
34	3423,58065	22145	259,462	0	70	8349,79914	31586	333,611	1
35	2683,22581	22145	235,578	0					

Tabla B.2: Tabla de estadísticas de la población de 30 genomas.

	50 genomas								
	Media fit	Max fit	Tiempo	Veces meta		Media fit	Max fit	Tiempo	Veces meta
0	351,18	8537	221,79	0	36	8103,23529	35445	596,489	5
1	1020,7	8537	461,34	0	37	8223,14	35445	540,92	4
2	1421,44	8537	533,65	0	38	6224,5098	35445	507,464	4
3	2458,1	19330	579,1	0	39	8855,5	35445	593,884	5
4	2057,04	19330	510,74	0	40	7690,48	35445	547,237	4
5	2951,88235	19330	475,25	0	41	7400,47059	35445	515,76	4
6	2726,2	19330	573,65	0	42	7465,56863	35445	511,939	4
7	8988,4898	30590	608,65	0	43	8861,88	35445	564,544	4
8	3150,94	19423	557,382	0	44	7730,18	35445	547,765	4
9	24810	24810	594,696	0	45	9183,84314	35445	545,667	4
10	5507,64706	24810	613,121	0	46	9927,11538	35445	639,08	7
11	5754,85714	24810	546,844	0	47	8158,31373	35445	565,09	4
12	5591,78	24810	511,137	0	48	8888,74	35445	606,729	5
13	5445,66	24810	560,843	1	49	7144,45098	35445	526,93	4
14	4765,80392	24810	567,543	2	50	9369,43137	35445	642,535	7
15	5648,15686	24810	532,543	2	51	8947,58	35445	578,242	5
16	5293,15686	31595	512,888	2	52	9292,42857	35445	604,931	5
17	4752,74	31595	495,258	2	53	8931,39583	35445	612,24	5
18	7020,68627	31595	631,641	4	54	8820,36735	35445	674,038	5
19	4826,62	31595	510,329	4	55	10010,6458	35445	789,329	5
20	4904,04	31595	516,098	4	56	8736,375	35445	720,991	6
21	5172,55102	31595	524,082	3	57	8433,08	35445	809,878	6
22	7078,44	31595	607,588	4	58	7732,59184	35445	701,579	5
23	6195,74	35432	580,144	5	59	7847,26	35445	636,12	5
24	5861,96	35432	524,98	4	60	8381,16327	35445	615,661	5
25	6405,5098	35432	651,868	4	61	7287,46939	35445	681,226	6
26	7216,38	35432	576,233	4	62	8922,12	35445	751,161	6
27	7035,82353	35432	562,349	4	63	10120,0980	35445	794,316	6
28	7671,7451	35432	599,386	5	64	8592,71429	35445	717,564	5
29	6209,5	35432	563,44	5	65	8655,91667	35445	679,859	6
30	7926,48	35432	566,224	5	66	11237,25	36053	689,35	7
31	7226,85714	35432	551,659	4	67	10537,25	35445	659,48	6
32	11513,58	35445	673,816	6	68	10277,75	35445	614,15	6
33	7225,46	35445	574,203	4	69	10734,25	35445	674,35	6
34	6945,3125	35445	555,065	4	70	10637,25	35445	652,85	6
35	8314,47059	35445	619,597	4					

Tabla B.3: Tabla de estadísticas de la población de 50 genomas.

	100 genomas								
	Media fit	Max fit	Tiempo	Veces meta		Media fit	Max fit	Tiempo	Veces meta
0	575,32	18627	505,456	0	36	3991,21	31616	976,813	5
1	1323,12	20469	729,07	0	37	5042,10891	31616	977,83	4
2	2787,79	20616	1041,105	0	38	5453,20588	31616	1049,739	5
3	3119,87129	21168	1096,196	0	39	4541,46	31616	930,826	4
4	3492,72	21168	1001,27	0	40	5723,48485	31616	941,572	5
5	4053,22	21168	1108,262	0	41	4285,23	31616	882,446	4
6	3805,5	24672	1122,828	1	42	3900,91089	31616	1004,924	4
7	4116,90909	24672	1278,271	1	43	3554,08911	31616	1040,766	4
8	29198	24672	1126,71	1	44	3748,62	31616	1013,567	4
9	4062,92857	29198	920,597	1	45	4194,19192	31616	976,059	4
10	4688,14	29198	1174,05	1	46	4769,79592	31616	1089,73	4
11	4483,37255	30567	1363,583	2	47	4506,49	31616	1017,063	5
12	4899,81	30567	1213,331	3	48	4419,85859	31763	948,579	5
13	4598,13725	30567	1138,957	2	49	4284,77551	31763	1019,256	5
14	4626,07	30567	1282,021	2	50	5061,41837	31763	1071,84	6
15	5300,0198	30567	1426,297	3	51	4285,82	31763	1019,316	5
16	4406,91919	30567	1198,324	5	52	4912,81188	31763	1084,778	4
17	5040,93939	30567	1405,687	3	53	4479,68627	31763	970,833	4
18	5070,73	31352	1314,193	5	54	3969,88889	31763	940,956	4
19	31352	31352	1353,625	4	55	4976,21	31763	1086,537	4
20	5275,24242	31352	1085,521	4	56	5311,54	31763	971,603	4
21	4494,59596	31352	1022,745	4	57	4495,27551	31763	971,513	4
22	4233,48	31352	880,978	4	58	5171,54082	31763	1128,835	5
23	4393,45545	31352	957,554	5	59	5351,83838	31763	1018,67	5
24	31352	31352	1008,422	4	60	5008,24242	31763	1096,718	6
25	4421,91	31352	1259,535	4	61	5039,3	31763	1122,076	5
26	3583,94059	31352	1394,201	5	62	5200,44554	32234	1082,22	5
27	5337,26263	31352	1473,986	5	63	4856,78431	32234	1103,221	5
28	6023,35	31352	1435,539	4	64	5166,46465	32234	1169,478	5
29	4961,16832	31352	1383,075	5	65	4554,59	32234	1079,598	4
30	3837,2	31352	1085,378	5	66	5085,10891	32234	1293,965	4
31	4710,21	31352	982,743	4	67	6288,26471	44428	1262,181	5
32	4915,4902	31352	1015,141	4	68	5626,22772	44428	1199,411	4
33	4313,83168	31352	950,194	5	69	5842,87	44428	1344,643	4
34	4459,5	31352	943,97	5	70	5869,478	44428	1362,652	5
35	4191,43878	31616	912,356	6					

Tabla B.4: Tabla de estadísticas de la población de 100 genomas.