Facultad
de
Ciencias

Generación de un Jugador Inteligente para el Videojuego Space Invaders

Generation of an Intelligent Player for the Video Game Space Invaders

Trabajo de Fin de Grado
para acceder al

GRADO EN INGENIERÍA INFORMÁTICA

Autor: Jairo González Gómez

Director: Inés González Rodríguez

Junio - 2022

Resumen

La Inteligencia Artificial ha ido tomando más y más importancia durante los últimos años. Desde los negocios hasta las redes sociales y los videojuegos, se ha ido aplicando en la mayoría de sectores. En el campo de los videojuegos, recientemente se han creado jugadores inteligentes capaces de jugar a videojuegos sencillos como el Breakout, Pong o Space Invaders. En el proyecto, se programará un entorno de juego que simule el clásico videojuego Space Invaders y se diseñará un agente inteligente que aprenderá a jugar utilizando aprendizaje profundo por refuerzo, para después comparar los resultados con los de un jugador humano promedio.

Palabras clave: Inteligencia Artificial, Aprendizaje Automático, Aprendizaje Profundo, Aprendizaje por Refuerzo, Red Neuronal Artificial, Space Invaders.

Abstract

Artificial Intelligence has become increasingly relevant in the last years. From business to social media and video games, it has been incorporated in most sectors. In the field of video games, intelligent agents have been recently created, that are capable of playing simple video games like Breakout, Pong or Space Invaders. In this project, a playing environment that emulates the classic video game Space Invaders will be developed, as well as an intelligent agent that will learn to play using Deep Reinforcement Learning. Later, the results achieved by the AI will be compared with the results from an average human.

Key words: Artificial Intelligence, Machine Learning, Deep Learning, Reinforcement Learning, Artificial Neural Network, Space Invaders.

Índice general

Índice de figuras

Agradecimientos

Este proyecto supone para mí la culminación de 4 años de carrera, con sus altos y sus bajos, sus mejores y peores momentos, pero que sin duda dejará un buen recuerdo en mi memoria. He tenido el placer de coincidir con excelentes compañeros y profesores a lo largo de estos años. Además, es un privilegio poder investigar y aplicar conocimientos adquiridos durante la carrera a un tema que me apasiona, como son los videojuegos.

En primer lugar, quiero agradecer a mi directora del proyecto, Inés González, por confiar en mi propuesta de trabajo y aceptar dirigir mi proyecto. Es de agradecer la ayuda que me ha dado durante estos meses de trabajo y la libertad que me ha dejado a la hora de realizar el proyecto a mi gusto.

En segundo lugar, dar las gracias a mi familia, amigos y compañeros por todo el apoyo durante estos años de carrera, necesario para poder seguir en los momentos más difíciles. También a Netkia, la empresa donde hice prácticas y estoy trabajando actualmente, y a los compañeros de trabajo, que me han ayudado a aprender la profesión y gracias a quienes he aprendido muchas cosas que no se enseñan en la carrera.

Por último, parte del desarrollo del proyecto no hubiera sido posible sin la ayuda del Grupo de Ingeniería de Computadores, que me dio acceso al CPD 3Mares, donde pude realizar las pruebas prácticas de mi proyecto. En especial, gracias a Pablo Abad y José Ángel Herrero por la ayuda proporcionada y el trato recibido.

Capítulo 1

Introducción

1.1. Contexto

La Inteligencia Artificial (IA) es una tecnología que, aunque existe desde hace muchos años, está tomando cada vez más importancia debido a varios factores. Uno de ellos es la mejora en el hardware, que hoy en día permite ejecutar algunos algoritmos en tiempos razonables para enormes cantidades de datos, mientras que cuando se inventaron su coste computacional era demasiado alto. Otro factor a tener en cuenta es que muchas empresas están empezando a implantar estas soluciones de IA en su día a día. Desde reconocimiento facial para fichar cuando se entra al trabajo, hasta la predicción de texto y el corrector al escribir un documento, o el detectar cuándo un cliente no va a continuar utilizando un servicio para poder ofrecerle ofertas que le mantengan suscrito.

También nos podría sorprender la masiva cantidad de información, cursos y tutoriales acerca del tema que se pueden encontrar a tan solo unos clicks de distancia, o impartidos en casi cualquier universidad. Esto hace que el acceso a estas tecnologías sea muy sencillo.

Otro campo donde la inteligencia artificial ha destacado es en el de los juegos. Desde juegos de mesa como el ajedrez, donde ya en 1996 el supercomputador Deep Blue consiguió ganar dos partidas contra el campeón del mundo Gary Kaspárov, aunque terminaría perdiendo 4-2. Sin embargo, tan solo un año más tarde, en 1997, una versión mejorada de Deep Blue derrotó a Kaspárov con una puntuación de 3.5-2.5. También, en 2015, el programa AlphaGo se convirtió en el primer programa en ganar a un jugador profesional de Go, concretamente al jugador chino Fan Hui.

En los videojuegos, la IA también está avanzando a un ritmo muy acelerado. En 1972, lo más parecido que había a un jugador inteligente era el script que manejaba el oponente del videojuego Pong. Hoy en día, si se analiza a un personaje secundario de un juego como el Red Dead Redemption 2, lanzado en 2018, se puede observar que tiene una rutina de vida como la de un humano de la época, que además varía según las condiciones climáticas y según el entorno en el que se encuentre.

En general, se ha conseguido dar un aspecto mucho más humano a los personajes de los videojuegos gracias a la IA, pero no es la única aplicación que tiene dentro de este campo. Otro lugar donde se pude ver que se utilizan estos algoritmos es en la eterna lucha entre el desarrollo de trucos o *hacks* y su detección en videojuegos. Se están empezando a desarrollar *hacks* que utilizan algoritmos de IA como el reonocimiento de imagen para tener ventaja con respecto al resto de jugadores. A su vez, también se utilizan algoritmos de inteligencia artificial para detectar comportamientos anómalos dentro de los videojuegos y poder castigar a los jugadores correspondientes.

Figura 1.1: Detección de enemigos utilizando reconocimiento de imagen en el videojuego Call of Duty Cold War [1]

Al igual que en el Ajedrez o en el Go, también se están creando jugadores inteligentes en videojuegos, capaces de competir o incluso superar a los mejores jugadores de un videojuego. Un claro ejemplo de esto es el proyecto OpenAI Five (Berner et al., 2019), que fue capaz de crear un equipo de 5 jugadores inteligentes que ganaron a Team OG, los campeones mundiales del videojuego Dota 2 en abril de 2019.

Dota 2[2] es un juego Multiplayer Online Battle Arena (MOBA) de estrategia en tiempo real, donde se juega 5 contra 5 y el objetivo es destruir la base enemiga. Para ello, cada jugador escoge un héroe con habilidades y roles distintos. En total, en la partida hay 10 héores, varias estructuras que protegen la base, muchas unidades, tanto aliadas como enemigas o neutrales, guardianes que se colocan para obtener visión... Además, cada héroe tiene 5 habilidades y unas runas que otorgan distintas ventajas. Contando con que se juega a 30 fotogramas por segundo y las partidas pueden durar 45 minutos o más, se pueden llegar a tomar más de 20.000 decisiones en una partida, considerando que se procesan 1 de cada 4 fotogramas. Para hacernos una idea, el ajedrez suele durar 80 movimientos y el

[1] https://youtu.be/QSMETfV7pVc
[2] Página oficial del videojuego Dota 2: https://www.dota2.com/home

Figura 1.2: Visión humana de Dota 2 (Berner et al., 2019)

Go 150. En el ajedrez se consideran unos 1.000 movimientos por cada observación y en el Go alrededor de 6.000 (binarios), mientras que en el Dota 2 el agente inteligente escoge de entre 8.000 y 80.000 posibles acciones. Es más, al contrario que en el ajedrez o el Go, donde la información es perfecta, la mecánica de la niebla de guerra que existe en el Dota 2 hace que se tenga que decidir con información imperfecta, ya que un jugador solo puede ver lo que ven los personajes, súbditos y torretas de tu equipo.

Es complicado hacer que una IA alcance el nivel superhumano en escenarios tan complejos, donde las decisiones se toman con información imperfecta, en tiempo real y tienen consecuencias a largo plazo, pero el equipo de OpenAI lo consiguió utilizando 5 redes neuronales entrenadas durante 10 meses mediante técnicas de aprendizaje por refuerzo. Para ello escalaron las técnicas existentes de aprendizaje por refuerzo a niveles sin precedentes, construyendo un sistema de entrenamiento distribuido que contaba con miles de GPUs en las que se entrenaban las masivas redes neuronales mientras jugaban partidas entre sí.

En general, los videojuegos son una muy buena forma de entrenar una IA para comprobar si sería apta o no para el mundo real y comprobar sus posibles aplicaciones (Luzgin, 2019). Por ejemplo, algunos investigadores han estado utilizando el conocido videojuego Grand Theft Auto V para entrenar y probar los algoritmos que se utilizan en los coches autónomos (Martinez et al., 2017), dado que el videojuego está fielmente basado en la ciudad de Los Angeles. Quién sabe si en un futuro podremos ver un equipo de robots jugando al fútbol o al baloncesto en vez de al Dota 2, o si el reconocimiento de enemigos se llega a utilizar en algún momento en la lucha contra el crimen y el terrorismo.

1.2. Space Invaders

Space Invaders es un videojuego de arcade que salió en Junio de 1978 y que contribuyó a la expansión global del entonces incipiente sector de los videojuegos, lo que le convierte en uno de los precursores de los videojuegos tal y como los conocemos hoy en día y en uno de los videojuegos más importantes y conocidos de la historia. Tanto es así que existe una leyenda urbana que dice que fue el causante de la escasez de monedas de 100 yenes.

En 1980, el juego se lanzó para la consola Atari 2600 y fue un rotundo éxito, siendo el videojuego más vendido hasta la fecha y recaudando un total de 450 millones de dólares. También fue el primer videojuego arcade que se lanzaba para una consola.

Figura 1.3: Space Invaders en Atari 2600. El jugador maneja el cañón verde de la parte inferior y las figuras rojas son los escudos destructibles.

Las reglas del juego son bastantes sencillas. El jugador maneja un cañón que se encuentra en la parte inferior de la pantalla y se puede mover de izquierda a derecha y disparar rayos láser. En la parte superior de la pantalla aparecen varias filas de alienígenas que se mueven de extremo a extremo de la pantalla y cada vez que llegan a un borde descienden. De vez en cuando estos alienígenas también disparan láseres que, si impactan al jugador, le quitarán una vida. También hay unos escudos destructibles entre el jugador y los alienígenas. El objetivo del juego es conseguir la máxima puntuación destruyendo el mayor número posible de alienígenas antes de perder todas las vidas o de que éstos lleguen al borde inferior de la pantalla. A medida que se van eliminando alienígenas, el resto se mueven más rápido, por lo que la dificultad aumenta progresivamente[3]. Al finalizar la partida, la puntuación se guarda en un histórico, de forma que el objetivo final es superar la mejor puntuación para lograr ser el número uno del ranking.

[3]En el videojuego original, el aumento de velocidad de los alienígenas se debía a que, cuando el jugador los iba destruyendo, se reducía la carga del microprocesador, permitiendo que el juego se ejecutara más rápido. De hecho, durante el desarrollo, se aumentó la velocidad de la banda sonora para que fuera acorde con la velocidad de los alienígenas.

1.3. Objetivo y Motivación

El objetivo de este trabajo de fin de grado es construir un jugador inteligente para un videojuego basado en el conocido Space Invaders de la consola Atari 2600. Este objetivo general se puede desglosar en los siguientes subobjetivos:

- Desarrollar un entorno de juego que permita jugar tanto al jugador inteligente como a jugadores humanos.
- Programar y entrenar un agente inteligente que sea capaz de jugar a un buen nivel en el entorno desarrollado utilizando Aprendizaje Profundo por Refuerzo.
- Evaluar los resultados obtenidos por el jugador inteligente, tomando como referencia a jugadores humanos.

Se ha escogido el videojuego Space Invaders, en primer lugar, porque es un videojuego de Atari, lo que hace que no suponga un gran coste computacional a la hora de ser jugado por una inteligencia artificial, al ser un juego bastante sencillo en cuanto a gráficos y dinámica, como se puede observar en la figura 1.3. En segundo lugar, he escogido este videojuego frente al Breakout o el Pong porque, personalmente, he llegado a jugar la versión de la consola NES, que tiene bastantes similitudes con el videojuego original de Atari, y me pareció que podría ser interesante como objeto de investigación.

Cabe destacar que OpenAI[4] ya tiene una versión propia del entorno[5] y que se han entrenado modelos para él, pero he preferido programar mi propio entorno, ya que así podría darle un toque personal, además de ajustar parámetros como la velocidad del juego o la resolución que más adelante me pueden ser útiles para entrenar el modelo de inteligencia artificial.

[4]https://openai.com
[5]https://github.com/openai/gym

Capítulo 2

Aprendizaje Profundo por Refuerzo

El aprendizaje por refuerzo es una técnica de aprendizaje en la que el agente inteligente interactúa con el entorno mientras aprende una política óptima para la toma de decisiones mediante prueba y error (Li, 2018).

La integración de aprendizaje por refuerzo y redes neuronales tiene una larga trayectoria, pero no ha sido hasta hace unos años que su popularidad ha despegado, debido a los nuevos avances en aprendizaje profundo, mejoras en el hardware y en los algoritmos.

Las redes neuronales profundas o aprendizaje profundo han estado a la cabeza del aprendizaje por refuerzo desde hace varios años en campos como videojugos, robots, procesado de lenguaje humano, etc. Claro ejemplo de esto puede ser (Mnih et al., 2013), un proyecto de Deepmind en el que consiguen que un agente inteligente aprenda a jugar a varios juegos de Atari utilizando Deep Q-networks.

Lo que hace que el aprendizaje profundo haya ayudado tanto al Aprendizaje por Refuerzo es que no hace falta incorporar conocimiento experto sobre el tema para poder conseguir unos buenos resultados, ya que las redes neuronales se encargan de la ingeniería de características y aprenden desde cero gracias al gradiente descendente. Además, el aprendizaje profundo explota la composición jerárquica de los datos para enfrentarse al reto que supone tener datos de tantas dimensiones.

Ahora, para comprender cómo funciona el aprendizaje profundo por refuerzo, desarrollaremos los conceptos en los que se basa, que son: aprendizaje automático, aprendizaje profundo y aprendizaje por refuerzo, apoyándonos en las ideas de (Li, 2018) y (Goodfellow et al., 2016).

2.1. Aprendizaje Automático

El aprendizaje automático o ML (del inglés *Machine Learning*) es un campo de la Inteligencia Artificial que utiliza distintas técnicas para aprender de los datos y realizar predicciones o decisiones en base a ellos.

Según el tipo de información disponible, se puede distinguir entre aprendizaje supervisado, aprendizaje no supervisado y aprendizaje por refuerzo (Russell and Norvig, 2021). En el aprendizaje supervisado, los datos están etiquetados. Dentro del aprendizaje supervisado tenemos los problemas de regresión y de clasificación, en los que las etiquetas son numéricas o categóricas, respectivamente. A su vez, el aprendizaje no supervisado trata de extraer información de los datos sin etiquetas, por ejemplo para ordenarlos en grupos o para reducir el tamaño de los datos manteniendo tanta información como sea posible. Por último, en el aprendizaje por refuerzo, el agente aprende en base a unas recompensas o penalizaciones que se le otorgan.

Un algoritmo de aprendizaje automático se compone de un conjunto de datos, una función de coste o pérdida, un proceso de optimización y un modelo. El conjunto de datos se divide en subconjuntos de entrenamiento, validación y test. La función de coste o pérdida evalúa el desempeño del modelo teniendo en cuenta medidas como el error cuadrático medio, la precisión, etc. Estas medidas se toman sobre el conjunto de entrenamiento y se minimizan como un problema de optimización. Lo que diferencia la optimización del ML es que también se evalúa sobre el conjunto de test, que contiene datos nuevos con los que no se ha entrenado el modelo. También se intenta que la diferencia entre el error en el conjunto de entrenamiento y el conjunto de test sea mínima. Para ello se modifican los hiperparámetros del modelo con técnicas como la *estimación por máxima verosimilitud (MLE)*. Para la optimización del modelo, se suelen utilizar algoritmos que buscan mínimos globales en funciones derivables, como puede ser el algoritmo del gradiente descendente. Este algoritmo utiliza optimización numérica para estimar los coeficientes que minimizan la función de coste, y se utiliza en modelos como la regresión lineal y polinómica, la regresión logística y modelos de aprendizaje profundo.

2.2. Aprendizaje Profundo

El aprendizaje profundo o *Deep Learning* es un campo del Aprendizaje Automático que se caracteriza por utilizar redes neuronales con una o varias capas ocultas entre la capa de entrada y la de salida, como se puede apreciar en la figura 2.1. Estas capas están compuestas por unidades llamadas neuronas artificiales y entre las unidades de cada capa hay enlaces con pesos que determinan cómo se comporta la red. En cada capa menos la de entrada, se calcula la entrada de cada unidad como la suma ponderada de las unidades de

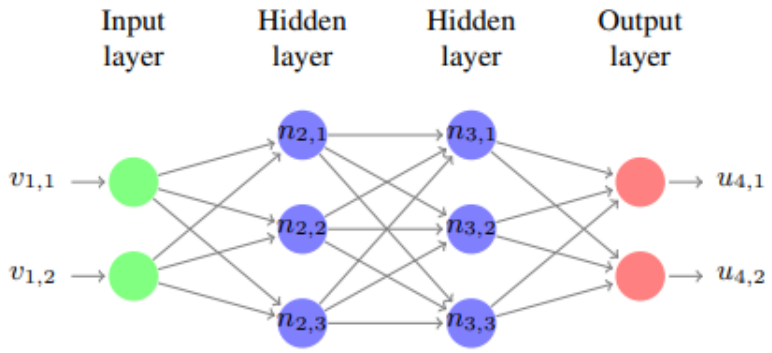

Figura 2.1: Estructura de una red neuronal artificial profunda (Sun et al., 2018)

la capa anterior. Después se utiliza una transformación no lineal o función de activación como la función logística, la tangente hiperbólica (tanh) o la popular función rectificadora (ReLU) que se muestra en la figura 2.2, para obtener una nueva representación de la salida de la capa anterior. Al procesar los datos desde la entrada a la salida, se calcula el error en la capa de salida y en las capas ocultas y los gradientes se propagan hacia atrás hasta la capa de entrada, de forma que se actualizan los pesos para optimizar la función de pérdida. Este algoritmo, conocido como *Backpropagation*, es el que normalmente se utiliza para que las redes neuronales artificiales (ANN) sean capaces de aprender de los datos. El pseudocódigo puede encontrarse en el Algoritmo 1.

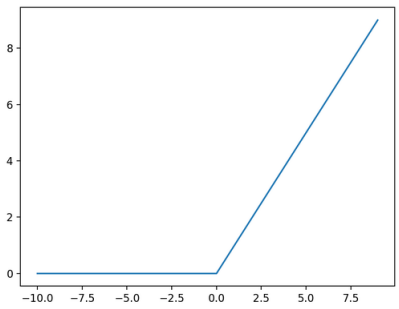

Figura 2.2: Función de activación ReLU ($R(n) = max(0, n)$)

Para entrenar la red, se puede utilizar una estrategia de regularización llamada *Dropout* (Srivastava et al., 2014), que consiste en eliminar de forma aleatoria neuronas y sus respectivos enlaces de la red mientras se entrena (Figura 2.3), para evitar que ésta se sobreajuste a los datos de entrenamiento y rinda mejor con datos nuevos. Al eliminar neuronas de forma aleatoria en la red también se combinan exponencialmente muchas arquitecturas de redes neuronales, concretamente 2^n estructuras distintas, siendo n el número de neuronas de la red original. De esta forma, entrenar una red con Dropout sería equivalente a entrenar 2^n redes más pequeñas que comparten los pesos y donde cada red

Algoritmo 1 Algoritmo de Backpropagation (Rumelhart et al., 1986)

procedure TRAIN
$X \leftarrow$ conjunto de datos de entrenamiento de dimensiones $m \times n$
$y \leftarrow$ etiquetas para los valores de X
$\theta \leftarrow$ los pesos de las respectivas capas
$l \leftarrow$ cada capa de la red neuronal, $1...L$
$D_{ij}^{(l)} \leftarrow$ el error para todo l, i, j
$t_{ij}^{(l)} \leftarrow 0$ para todo l, i, j
for $i = 1$ to m **do**
$a^l \leftarrow feedforward(x^{(i)}, \theta)$
$d^l \leftarrow a(L) - y(i)$
$t_{ij}^{(l)} \leftarrow t_{ij}^{(l)} + a_j^{(l)} \cdot t_i^{l+1}$
end for
if $j \neq 0$ **then**
$D_{ij}^{(l)} \leftarrow \frac{1}{m} t_{ij}^{(l)} + \lambda \theta_{ij}^{(l)}$
else
$D_{ij}^{(l)} \leftarrow \frac{1}{m} t_{ij}^{(l)}$ ▷ donde $\frac{\partial}{\partial \theta_{ij}^{(l)}} J(\theta) = D_{ij}^{(l)}$ y $J(\theta)$ es la función de coste
end if
end procedure

tiene una muy baja probabilidad de entrenarse.

Otra técnica muy utilizada en la actualidad es la normalización por lotes, o *Batch normalization* (Ioffe and Szegedy, 2015). Por defecto, entrenar una red neuronal es una tarea muy complicada, ya que la distribución de cada entrada de las capas cambia a medida que cambian los parámetros de la capa anterior. Esto hace que el entrenamiento sea mucho más lento al requerir bajas tasas de aprendizaje y una inicialización de parámetros concreta. Este fenómeno se conoce como *internal covariate shift* y se puede reducir su impacto normalizando las entradas de las capas. La normalización por lotes se ocupa de esto, normalizando las entradas para una pequeña porción de los datos (mini-batch), lo que permite utilizar tasas de aprendizaje más elevadas, haciendo que el entrenamiento sea mucho más rápido. Además, al entrenar las redes solo para pequeños lotes de datos, se evita el problema del sobreajuste y por lo tanto no es necesario utilizar Dropout.

Por otro lado, junto con la normalización por lotes, es muy común actualmente utilizar una variante del gradiente descendente llamada gradiente descendente estocástico (Bottou et al., 1991). Este algoritmo también optimiza una función de pérdida para llegar a un mínimo, pero en vez de hacerlo para todo el conjunto de datos, lo hace para pequeños lotes, al igual que la normalización por lotes. Se ha demostrado que tiene algunas ventajas sobre el gradiente descendente total, como que converge más rápido cuando los datos son redundantes, que no se queda atascado en mínimos locales llegando a converger en el mínimo global, y que evita el problema del sobreajuste. Es por esto que la mayoría de

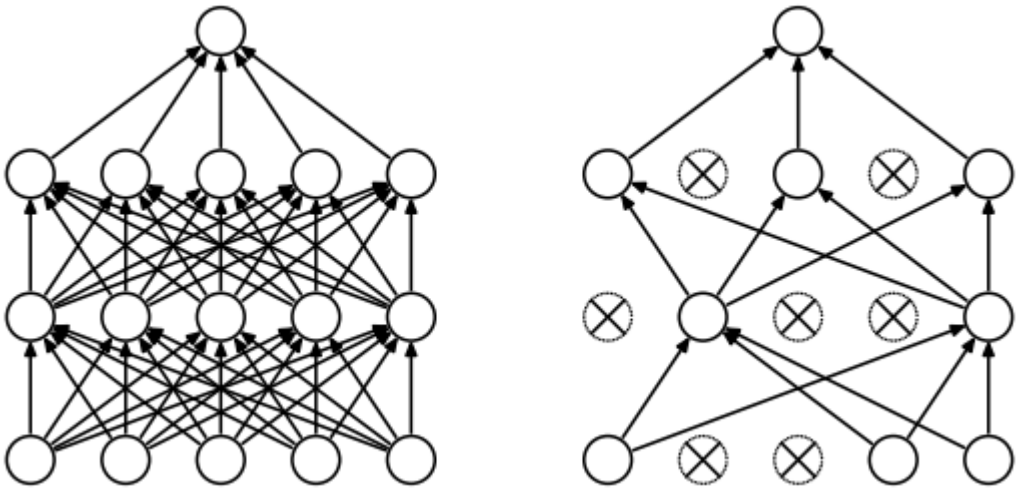

Figura 2.3: **Izquierda:** Red neuronal con 2 capas ocultas. **Derecha:** Ejemplo de una red más pequeña generada al aplicar Dropout a la red de la izquierda

redes neuronales se entrenan utilizando normalización por lotes, gradiente descendente estocástico (SGD) y alguna variante de *Backpropagation*.

Lo que hace que las redes neuronales sean mejores que otros modelos de aprendizaje automático para ciertas tareas es que son capaces de detectar complejas relaciones en los datos sin procesar, lo que las hace muy útiles para datos con una gran dimensionalidad. Por ejemplo, una red neuronal convolucional (CNN) es capaz de detectar relaciones entre píxeles en cualquier lugar de la imagen, puede detectar bordes, objetos, partes, etc. Una red neuronal recurrente (RNN) puede procesar datos secuenciales como el habla o el texto elemento a elemento, con las capas ocultas haciendo de historial de los elementos pasados, aunque es difícil que mantengan la información durante mucho tiempo. Para esa labor existen las redes de gran memoria de corto plazo o *Long-short term memory (LTSM)*, que incorporan una memoria explícita que puede ser actualizada y borrada, lo que les permite aprender dependencias de largo plazo en los datos.

No entraremos más en detalle sobre estos tipos de redes mencionadas, es suficiente con saber que existen muchos tipos de redes neuronales para diversas tareas, e incluso se pueden combinar capas de varios tipos para formar distintas redes neuronales con funciones específicas.

2.3. Aprendizaje por Refuerzo

Siguiendo el análisis de (Li, 2018), expondremos primero qué es el Aprendizaje por Refuerzo o RL (del inglés *Reinforcement Learning*) y después cada una de las partes y

algoritmos que lo componen.

2.3.1. Descripción del Problema

En el RL, un agente inteligente interactúa con el entorno a lo largo del tiempo. En cada instante t, el agente recibe un estado s_t del espacio de estados S y selecciona una acción a_t del espacio de acciones A según una política $\pi(a_t|s_t)$ que marcará el comportamiento del agente. Al encontrarse en el estado s_t y tomar una acción a_t, se recibe una recompensa numérica r_t y se pasa al siguiente estado s_{t+1}, siguiendo las funciones de recompensa $R(s, a)$ y probabilidad de transición de estado $P(s_{t+1}|s_t, a_t)$. Este proceso continúa hasta que el agente llega a un estado final y después se reinicia. La recompensa acumulativa descontada a largo plazo R_t (*infinite-horizon discounted reward* o simplemente *return*), se calcula como la suma de las recompensas aplicando un factor de descuento $\gamma \in (0, 1]$, que determina cuánta importancia tienen las recompensas en un futuro lejano en relación a las de un futuro inmediato.

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

El agente deberá intentar maximizar las recompensas a largo plazo en cada estado.

2.3.2. Función de Valor

Una función de valor es una predicción de la recompensa acumulada futura esperada, que mide cuán buena es una pareja estado-acción. La función de estado, dada por:

$$v_\pi(s) = E[R_t|s_t = s], \text{ donde } R_t = \textstyle\sum_{k=0}^{\infty} \gamma^k r_{t+k},$$

es la recompensa final esperada de seguir la política π a partir del estado s, y se descompone en la ecuación de Bellman:

$$v_\pi = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')].$$

El valor de acción,

$$q_\pi(s, a) = E[R_t|s_t = s, a_t = a],$$

es la recompensa final esperada de realizar la acción a en el estado s y después seguir la política π. Se descompone en la siguiente ecuación de Bellman:

$$q_\pi(s,a) = \sum_{s',r} p(s',r|s,a)[r + \gamma \sum_{a'} \pi(a'|s') q_\pi(s',a')].$$

Una función óptima del valor de una acción,

$$q_*(s,a) = \max_\pi q_\pi(s,a),$$

es el máximo valor para una acción que se puede lograr por cualquier política para un estado s y una acción a. $q_*(s,a)$ se descompone en la siguiente ecuación de Bellman:

$$q_*(s,a) = \sum_{s',r} p(s',r|s,a)[r + \gamma \max_{a'} q_*(s',a')].$$

El valor óptimo de un estado,

$$v_*(s) = \max_\pi v_\pi(s) = \max_a q_{\pi^*}(s,a),$$

es el máximo valor que se puede conseguir por una política para un estado s, y se descompone en la siguiente ecuación de Bellman:

$$v_*(s) = \max_a \sum_{s',r} p(s',r|s,a)[r + \gamma v_*(s')].$$

Denotamos una política óptima por π^*.

2.3.3. Aprendizaje por Diferencia Temporal

El Aprendizaje por Diferencia Temporal o TDL (del inglés *Temporal Difference Learning*) es esencial para el aprendizaje por refuerzo. Este término se refiere a los métodos utilizados para evaluar la función, como *Q-learning* y *SARSA*.

El aprendizaje por diferencia temporal aprende la función de evaluación $V(s)$ directamente de la experiencia, mediante el error de diferencia temporal y estimaciones. TDL es un problema de predicción, que sigue la regla

$$V(s) \leftarrow V(s) + \alpha[r + \gamma V(s') - V(s)],$$

donde α es la tasa de aprendizaje, y $r+\gamma V(s')-V(s)$ es el error de diferencia temporal. En el algoritmo 2 se presenta el código para TDL con programación dinámica, es decir,

Algoritmo 2 TD learning, (Sutton and Barto, 2018)

Input: la política π a ser evaluada
Output: la función de valor V
  inicializar V a 0 para todos los estados
  for cada episodio **do**
    inicializar estado s
    for cada estado s del episodio, si s no es terminal **do**
      $a \leftarrow$ la acción dada por la política π para el estado s
      tomar la acción a y observar r, s'
      $V(s) \leftarrow V(s) + \alpha[r + \gamma V(s') - V(s)]$
      $s \leftarrow s'$
    end for
  end for

guardando los resultados en una tabla. Más concretamente, se trata de *TD(0) learning*, donde el 0 indica que está basado en recompensas de un paso.

En la regla de evaluación de TDL se utiliza *Bootstrapping* que consiste en estimar el valor del estado o la acción en base a estimaciones posteriores, y es común en métodos de RL como TDP, *Q learning*, o *actor-critic*. Estos difieren del gradiente descendente, ya que el objetivo depende de los pesos estimados, por lo que (Sutton and Barto, 2018) introduce el concepto de semi-gradiente descendente.

Q-learning es un método *off-policy* de control, lo que significa que tiene como objetivo encontrar la política óptima sin tener en cuenta las acciones del agente. Este método ajusta la política de forma voraz teniendo en cuenta el máximo de los valores de las acciones para cada estado, donde el valor de la combinación estado-acción $Q(s, a)$ se actualiza según la siguiente fórmula:

$$Q^{new} = Q(s, a) + \alpha \cdot [r_t + \gamma \cdot \max_a Q(s_{t+1}, a)],$$

donde r_t es la recompensa al pasar del estado s_t al estado s_{t+1} y α es la tasa de aprendizaje. El pseudocódigo de *Q(0) learning* se presenta en el Algoritmo 3.

Estos algoritmos utilizan programación dinámica, almacenando los datos en forma de tabla e inicializando los valores a 0.

2.3.4. Aproximación de la Función

Mientras que en los algoritmos anteriores guardábamos los valores de la función o la política en forma de tabla, este método no es eficiente cuando el espacio de acciones o de estados es demasiado grande o no es discreto. Cuando esto ocurre, se utiliza aproximación

Algoritmo 3 Q-learning, (Sutton and Barto, 2018)

Output: la función de valor de acción Q
  inicializar Q a 0 para todos los estados
  inicializar el valor de acción para los estados terminales a 0
  for cada episodio **do**
    inicializar estado s
    for cada estado s del episodio, si s no es terminal **do**
      $a \leftarrow$ la acción para s derivada de Q, e.g. ϵ-greedy
      tomar la acción a y observar r, s'
      $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \ \text{máx}_{a'} Q(s',a') - Q(s,a)]$
      $s \leftarrow s'$
    end for
  end for

de la función, que es un método que generaliza a partir de ejemplos de una función para construir una aproximación de la función completa. Para ello se puede utilizar aproximación lineal o incluso redes neuronales.

En el algoritmo *TD(0) learning* con aproximación de función, $\hat{v}(s, \boldsymbol{w})$ es el valor aproximado de la función, $\boldsymbol{w}$ es el vector de pesos de valores de la función, $\nabla \hat{v}(s, \boldsymbol{w})$ es el gradiente del valor aproximado de la función con respecto al vector de pesos y el vector de pesos se actualiza según la siguiente regla,

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha[r + \gamma \hat{v}(s', \boldsymbol{w}) - \hat{v}(s, \boldsymbol{w})] \nabla \hat{v}(s, \boldsymbol{w}).$$

2.3.5. Aprendizaje Profundo por Refuerzo

Hablamos de aprendizaje profundo por refuerzo cuando se utilizan redes neuronales para aproximar cualquiera de los siguientes componentes del RL: la función de valor $\hat{v}(s;\theta)$ o $\hat{q}(s,a;\theta)$, la política $\pi(a|s;\theta)$, o el modelo(funciones de transición de estado y recompensa). En estas funciones, los parámetros θ son los pesos en las redes neuronales. Normalmente se utiliza gradiente descendente estocástico para actualizar los pesos de las redes, pero en los métodos que no siguen ninguna política pueden ocurrir inestabilidades, haciendo que sea difícil entrenar estas redes.

2.4. Elementos Fundamentales del Aprendizaje Profundo por Refuerzo

Como ya hemos mencionado anteriormente, un agente de RL realiza una serie de acciones y observa los estados y las recompensas. Un problema de RL puede formularse como un problema de control, predicción o planificación, y los métodos para obtener la solución pueden ser basados en el modelo o libres de modelo, y basados en el valor o basados en la política. También se debe tener en cuenta el balance entre exploración y explotación. En la figura 2.4 se muestran los métodos basados en valor y basados en política.

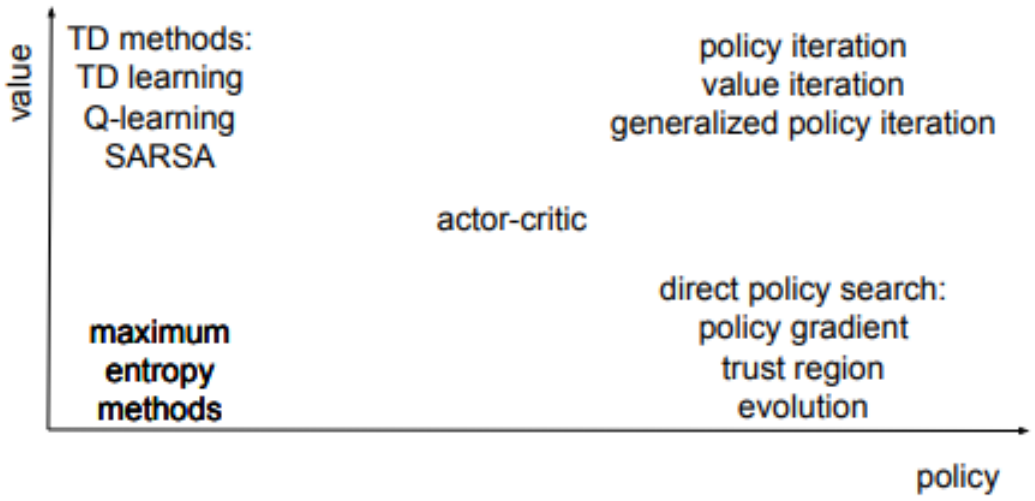

Figura 2.4: Métodos basados en valor y política (Li, 2018)

En esta sección, se explicarán los componentes fundamentales del aprendizaje profundo por refuerzo (Li, 2018): función de valor, política, recompensa, modelo, y el balance entre exploración y explotación.

2.4.1. Función de Valor

La función de valor es un concepto fundamental en el aprendizaje por refuerzo. Es la función que se encarga de predecir la recompensa futura acumulativa descontada, midiendo qué tan bueno es un estado o una pareja estado-acción. En la actualidad, es muy común aproximarla mediante un tipo de red neuronal conocida como *Deep-Q-Network* (DQN) (Mnih et al., 2015) o alguna de sus variantes, como *Double DQN*.

Antes del algoritmo de DQN, el aprendizaje por refuerzo era inestable o incluso divergente cuando la función de valor se aproximaba con una función no lineal como una

red neuronal, por lo que DQN hizo varias contribuciones importantes en el campo del aprendizaje profundo por refuerzo aplicado a los videojuegos:

- Estabilizó el entrenamiento de la aproximación de función de valor con redes neuronales convolucionales utilizando memoria de experiencias y una red distinta para generar los valores de Q y calcular el error, evitando así correlaciones en los datos.
- Diseñó un aprendizaje por refuerzo de principio a fin, con únicamente los píxeles y la puntuación como entrada, lo que indica que se necesita un mínimo conocimiento del juego.
- Se entrenó la red con el mismo algoritmo, arquitectura e hiperparámetros para que obtuviera buenos resultados en distintos juegos, consiguiendo mejores puntuaciones que algoritmos anteriores y obteniendo resultados similares a un humano profesional.

DQN utiliza una red neuronal convolucional (CNN) para aproximar la mejor función de valor de acción,

$$Q^*(s,a) = \max_{\pi} \mathbb{E}[\sum_{0}^{\infty} \gamma^i r_{t+1} | s_t = s, a_t = a, \pi]$$

.

En la actualidad, se han propuesto muchas optimizaciones y variantes, como la doble DQN (D-DQN) (van Hasselt et al., 2016), que arreglaba el problema de sobreestimación del Q-learning. Otra mejora destacable ha sido utilizar una memoria de experiencias con prioridad, obteniendo mejores resultados en juegos de Atari tanto para DQN como para D-DQN (Mnih et al., 2016).

2.4.2. Política

Una política mapea un estado a una acción, y la optimización de una política es encontrar un mapeo óptimo. Dentro del campo del RL existen numerosos tipos de algoritmos para la optimización de políticas, como estrategias evolutivas, episódicas, gradientes, basados en el modelo, búsquedas por optimización de trayectoria, actor-critic, Q-learning, etc.

Actor-critic

Un algoritmo de *actor-critic* aprende una política y una función de valor de estado, y la función de valor se utiliza para actualizar un estado en base a estimaciones, para

Algoritmo 4 Algoritmo de cada actor en A3C, (Mnih et al., 2016)

Vectores compartidos θ, θ_v y específicos a cada actor θ', θ'_v
Contador global compartido $T = 0, T_{max}$
Inicializar el contador de iteraciones $t \leftarrow 1$
for $T \leq T_{max}$ **do**
  Reiniciar gradientes, $d\theta \leftarrow 0$ y $d\theta_v \leftarrow 0$
  Sincronizar parámetros específicos de cada actor $\theta' = \theta$ y $\theta'_v = \theta_v$
  $t_{start} = t$, obtener estado s_t
  for s_t *no terminal y* $t - t_{start} \leq t_{max}$ **do**
    Realizar acción a_t según la política $\pi(a_t|s_t;\theta')$
    Recibir recompensa r_t y nuevo estado s_{t+1}
    $t \leftarrow t+1, T \leftarrow T+1$
  end for
  $R = \begin{cases} 0, & \text{si } s_t \text{ es terminal} \\ V(s_t, \theta'_v), & \text{en otro caso} \end{cases}$
  for $i \in \{t-1, ..., t_{start}\}$ **do**
    $R \leftarrow r_i + \gamma R$
    Acumular gradientes con respecto a $\theta' : d\theta \leftarrow d\theta + \nabla_{\theta'} log\pi(a_i|s_i;\theta')(R - V(s_i;\theta'_v))$
    Acumular gradientes con respecto a $\theta'_v : d\theta_v \leftarrow d\theta_v + \nabla_{\theta'_v}(R - V(s_i;\theta'_v))^2$
  end for
  Actualizar de forma asíncrona θ utilizando $d\theta$ y θ_v con $d\theta_v$
end for

reducir la varianza y acelerar el aprendizaje (Sutton and Barto, 2018). Nos centraremos en *asynchronous advantage actor-critic* (A3C), ya que es el método que mejores puntuaciones obtiene en (Mnih et al., 2016).

En A3C, contamos con distintos actores en paralelo, que emplean distintas políticas de exploración para estabilizar el entrenamiento, por lo que no se utilia la repetición de experiencias. A diferencia de la mayoría de algoritmos de aprendizaje profundo, los algoritmos asíncronos pueden ejecutarse en un único procesador multinúcleo, consiguiendo entrenamientos mucho más rápidos y mejores resultados que DQN, D-DQN y D-DQN con prioridad en juegos de Atari.

A3C mantiene una política $\pi(a_t|s_t;\theta)$ y una estimación de la función de valor $V(s_t;\theta_v)$, que se actualiza con recompensas en n pasos cada t_{max} acciones o cuando se alcanza un estado final, similar a utilizar lotes de datos. En la actualización en n pasos, $V(s)$ se actualiza hacia la enésima recompensa, definida como

$$r_t + \gamma r_{t+1} + \cdots + \gamma^{n-1} r_{t+n-1} + \gamma^n V(s_{t+n}).$$

De esta forma, cada enésima recompensa es una actualización en 1 paso para el último

estado, una actualización en 2 pasos para el penúltimo estado..., hasta llegar a un total de t_{max} actualizaciones, tanto para los parámetros de la función de valor como la de política.

La actualización del gradiente sigue la fórmula

$$\nabla_{\theta'} log\, \pi(a_t|s_t;\theta')A(s_t,a_t;\theta,\theta_v),$$

donde

$$A(s_t,a_t;\theta,\theta_v) = \sum_{i=0}^{k-1} \gamma^i r_{t+1} + \gamma^k V(s_{t+k};\theta_v) - V(s_t;\theta_v)$$

es una estimación de la función de ventaja, con $k \leq t_{max}$.

2.4.3. Recompensa

Las recompensas dan información al agente de RL para que éste pueda tomar decisiones. Estas recompensas pueden estár dispersas a lo largo del tiempo. Por ejemplo, en el Go, la recompensa se otorga al final de la partida, mientras que en el Space Invaders pueden pasar varios fotogramas desde que se dispara un láser hasta que se otorga la recompensa por destruir un alienígena. En algunos casos, la recompensa puede venir dada por una función matemática, pero puede no estar disponible para algunos problemas de RL.

También es posible normalizar la función de recompensa para facilitar el entrenamiento mientras que se mantiene una política óptima. Por ejemplo, en (Mnih et al., 2013), todas las recompensas positivas se fijaban a 1 y todas las negativas a -1, sin cambiar las recompensas de valor 0. De esta forma, se limita la escala del error y hace que se pueda utilizar la misma tasa de aprendizaje entre varios juegos, aunque puede afectar al desempeño del agente al no ser capaz de diferenciar entre recompensas de distinta magnitud.

2.4.4. Modelo

Un modelo es la representación que un agente tiene sobre el entorno, incluyendo el modelo de transición de estados y el modelo de recompensas. Normalmente, se asume que el modelo de recompensas es conocido. También distinguimos entre RL libre de modelo y RL basado en el modelo.

En el RL libre de modelo, el agente de RL aprende a lidiar con sistemas dinámicos desconocidos, lo que requiere un gran número de muestras. RL libre de modelo funciona bien para simuladores capaces de proporcionar ese gran número de muestras, como el Go o un videojuego de Atari. Por otro lado, puede ser muy costoso o incluso imposible para sistemas físicos reales.

En el RL basado en el modelo, el agente de RL trata de aprender la función de valor y/o la política siendo eficiente con los datos. Aquí surge el problema de que el agente puede tener problemas para identificar el modelo, por lo que las estimaciones pueden ser imprecisas y el desempeño del agente está limitado por el modelo estimado.

2.4.5. Exploración vs Explotación

Un componente fundamental en el aprendizaje profundo por refuerzo es el balance entre la exploración de nuevas políticas y la explotación de la mejor política actual. Se debe decidir sobre la marcha si es mejor explotar la información que se tiene para tomar la mejor elección o explorar nuevas elecciones para descubrir nueva información. En decisiones secuenciales, esto supone sacrificar pérdidas a corto plazo para aumentar la recompensa a largo plazo.

Existen varios principios de balance entre exploración y explotación, desarrollados para el problema del bandido multibrazo, pero aplicables a problemas de aprendizaje por refuerzo.

El problema del bandido multibrazo (Silver, 2015) es un problema clásico del estudio de la exploración contra la explotación. El problema describe un jugador en una fila de máquinas tragaperras, que debe decidir a qué máquinas jugar, cuántas veces debe jugar a cada máquina, en qué orden y cuándo debe cambiar de máquina. En el problema, cada máquina da una recompensa aleatoria según una distribución específica de cada máquina que no se conoce a-priori. El objetivo del jugador es maximizar la recompensa total mediante una secuencia de tiradas en las máquinas.

El problema del bandido multibrazo se define como una tupla $< A, R >$, donde A es un conjunto de brazos o acciones, y $R(r|a) = \mathbb{P}(r|a)$ es una distribución de probabilidad de las recompensas, desconocida por el agente. En cada paso t, el agente selecciona una acción $a_t \in A$, recibe una recompensa $r_t \sim R(\cdot|a_t)$ del entorno y el objetivo es maximizar la recompensa acumulativa

$$\sum_{r=1}^{t} r_\tau$$

.

La función de acción-valor es la recompensa esperada para la acción $a, Q(a) = \mathbb{E}[r|a]$. El valor óptimo es $V^* = Q(a^*) = \text{máx}_{a \in A}\, Q(a)$. El error es la pérdida en un paso,

$$l_t = \mathbb{E}[V^* - Q(a_t)]$$

.

El error total hasta el paso t es:

$$L_t = \mathbb{E}[\sum_{r=1}^{t} V^* - Q(a_\tau)]$$

.

La máxima recompensa acumulativa es el mínimo error total.

Llamamos $N_t(a)$ al número esperado de veces que se escogerá la acción a hasta el paso t. Un algoritmo voraz selecciona la acción con el máximo valor, $a_t^* = \arg\max_{a \in A} \hat{Q}_t(a)$, donde $\hat{Q}_t(a)$ es una estimación de $Q(a)$ mediante el método Monte-Carlo,

$$\hat{Q}_t(a) = \frac{1}{N_t(a)} \sum_{\tau=1}^{t} r_\tau 1(a_\tau = a),$$

donde

$$\sum_{\tau=1}^{t} r_\tau 1(a_\tau = a)$$

significa que para calcular la recompensa total para la acción a, solo se tendrán en cuenta las recompensas de los instantes t donde la acción escogida es a.

El algoritmo voraz puede tomar una acción subóptima. Sin embargo, la política ϵ-voraz con $\epsilon \in (0, 1)$ puede asegurar un error mínimo con una constante ϵ. En ϵ-voraz, el agente selecciona una acción a según la siguiente fórmula:

$$a_t = \begin{cases} \arg\max_{a \in A} \hat{Q}_t(a), & \text{con probabilidad } 1 - \epsilon \\ \text{acción aleatoria}, & \text{con probabilidad } \epsilon \end{cases}$$

Capítulo 3

Requisitos y Herramientas del Proyecto

En esta sección se desarrollarán los pasos realizados antes de comenzar con el desarrollo del proyecto software, como los requisitos del proyecto o el análisis y selección de herramientas.

3.1. Requisitos

3.1.1. Requisitos Funcionales

RF.1 El jugador humano deberá ser capaz de mover y disparar el cañón utilizando el teclado

RF.2 El juego contará con 48 alienígenas, distribuidos en una matriz 6x8

RF.3 Los alienígenas de las tres primeras filas serán rojos, con un valor de 10 puntos; los de las siguientes filas verdes, con 20 puntos de valor, y los de la última fila serán amarillos y valdrán 30 puntos

RF.4 Un alienígena aletorio disparará un rayo láser hacia el jugador tras un periodo de tiempo, que le quitará una vida en caso de impactarle.

RF.5 Los alienígenas se destruirán tras un impacto de un láser del jugador

RF.6 El juego tendrá varios niveles, accesibles destruyendo todos los alienígenas del nivel anterior

RF.7 El juego finalizará cuando los alienígenas lleguen al planeta de la parte inferior o el jugador se quede sin vidas

RF.8 El juego contará con unos escudos destructibles por los láseres de los alienígenas y del cañón

RF.9 El juego tendrá un menú inicial, donde se podrá escoger un un nombre, así como el tipo de jugador (Humano / IA)

RF.10 Desde el menú se podrá comenzar una partida, visitar las mejores puntuaciones o salir del juego

RF.11 El usuario deberá poder navegar por el menú utilizando el teclado o el ratón

RF.12 Un jugador inteligente deberá ser capaz de jugar al videojuego

RF.13 El juego contará con una tabla de marcadores donde se registrarán las puntuaciones más altas y el nombre del jugador que las ha conseguido

RF.14 El juego contará con una pantalla de Game Over cuando finalice la partida

RF.15 Durante la partida y en la pantalla de Game Over, se mostrará la puntuación obtenida por el jugador

RF.16 El juego deberá aumentar la velocidad a medida que quedan menos alienígenas

RF.17 El juego deberá tener una canción de fondo durante la partida

RF.18 El juego deberá tener sonidos para cuando se disparan lasers o ocurren colisiones

RF.19 El juego deberá tener una pequeña melodía cuando finaliza la partida

RF.20 El juego deberá tener una nave nodriza que aparezca después de un intervalo de tiempo aleatorio

RF.21 El número de vidas restantes se mostrará en la parte superior derecha, en forma de tantas imágenes del cañón como vidas queden.

RF.22 Las mejores puntuaciones se guardarán en un fichero.

3.1.2. Requisitos No Funcionales

RNF.1 El juego correrá a 60 fotogramas por segundo.

RNF.2 El estado del juego se codificará como una matriz de píxeles RGB y un entero que indique la puntuación actual.

RNF.3 La integración entre el juego y el agente inteligente se realizará utilizando matrices de píxeles.

RNF.4 El juego y el menú deberán tener un estilo retro, similar al videojuego de Atari.

RNF.5 El juego deberá tener un pequeño grado de aleatoriedad en el retardo de los disparos de los aliens.

RNF.6 La música y los sonidos del juego deberán tener un toque retro antiguo.

RNF.7 La puntuación que otorga la nave nodriza se calculará contando los disparos que el jugador ha efectuado antes de destruirla.

3.2. Análisis y Selección de Herramientas

En primer lugar, son muchas las herramientas que se podrían utilizar para desarrollar un videojuego; desde motores gráficos como Unity o Unreal Engine hasta programarlo a bajo nivel en lenguajes como C, Java o Python, con ayuda de sus respectivas librerías gráficas. Hay que tener también en cuenta que después del videojuego debemos programar también una Inteligencia Artificial que sea capaz de jugarlo a un buen nivel, por lo que buscaremos que la integración de la IA sea lo más sencilla posible. Por ejemplo, si escogemos un motor gráfico como Unity para el videojuego y Python para la inteligencia artificial, deberíamos pasar datos desde Unity a Python utilizando sockets, para que después Python los procese y envíe el resultado de vuelta a Unity. Esto se podría llegar a implementar, pero, además de que complicaría bastante el proyecto, podría llegar a introducir retardos o fallos que afecten a la IA, que puede llegar a tener que tomar 60 decisiones por segundo.

Para construir el modelo de inteligencia artificial, a priori podríamos utilizar lenguajes como Python, R o Matlab. Aquí nos hemos decantado por Python, ya que es un lenguaje de programación muy polivalente y cuenta con varias librerías que nos pueden ayudar a crear, entrenar y desplegar el modelo(Teoh and Rong, 2022). Algunas de las librerías que cumplen esta función son Keras, Tensorflow, Scikit-learn o PyTorch. Además, Python es un lenguaje de programación que se puede ejecutar en la mayoría de sistemas operativos sin tener que modificar o volver a compilar el código, lo que será de gran ayuda a la hora de ejecutar el entrenamiento en una máquina que a priori no se sabrá si utiliza Windows o Linux.

Tras un análisis más profundo de las librerías, nos hemos decantado por utilizar Keras[1], una API de Deep Learning para Python que funciona sobre TensorFlow[2]. Se ha escogido esta librería porque proporciona una forma sencilla, flexible y potente de crear modelos de Deep Learning y cuenta con una amplia documentación, tanto oficial como de los propios usuarios. Además, los modelos generados con Keras se pueden desplegar en la mayoría de sistemas operativos así como exportarlos a JavaScript para que se ejecuten en buscadores.

Para crear el videojuego, se ha optado por utilizar también Python en vez de un motor gráfico. Se ha escogido esta opción ya que el juego Space Invaders no es un juego

[1] https://keras.io
[2] https://www.tensorflow.org

complejo como los de hoy en día, por lo que debería ser fácil de implementar sin las ayudas que nos proporcionan los motores gráficos. Además, se utilizará la librería Pygame[3], una librería gráfica diseñada para el diseño de videojuegos en Python, que cuenta con las funcionalidades necesarias para ejecutar un juego y obtener la matriz de píxeles de la pantalla y pasársela de entrada a la Inteligencia Artificial. Cabe destacar que, aunque Python es conocido por ser un lenguaje muy poco eficiente[4], en nuestro caso no supone un problema al tratarse de un videojuego sencillo y que no requiere una gran cantidad de recursos, por lo se ha preferido la facilidad de programación y la variedad de librerías antes que la rapidez y la eficiencia.

En resumen, todo el proyecto se programará en Python para favorecer una buena integración entre el videojuego y la IA.

[3]https://www.pygame.org/wiki/about

[4]La mayoría de videojuegos modernos están implementados utilizando motores gráficos y en lenguajes mucho más rápidos y eficientes que Python, como C# o C++

Capítulo 4

Diseño y Desarrollo del Proyecto

En esta sección se abordará la arquitectura del proyecto software, analizando los componentes principales y los patrones software utilizados, aportando los diagramas necesarios para su comprensión. También se analizará todo el proceso de desarrollo, desde la metodología utilizada hasta las pruebas y el despliegue final.

4.1. Metodología Utilizada

Para el desarrollo del proyecto, se ha seguido una metodología iterativa e incremental (Larman and Basili, 2003), un modelo de desarrollo que divide el proyecto en distintos ciclos o iteraciones (iterativa), en los que se trabaja en pequeños bloques, construidos uno sobre el otro (incremental). En las primeras iteraciones se trabaja en implementación simple de una pequeña parte de los requisitos, mientras que en las siguientes versiones se irán añadiendo nuevas modificaciones y funcionalidades, cumpliendo cada vez una mayor parte de los requisitos hasta que se finaliza el proyecto.

Las iteraciones que se han realizado durante el proceso se exponen en la sección 4.4 Cronograma de Iteraciones.

El proceso consiste en tres partes:

- Inicialización: Fase en la que se crea una primera versión funcional del sistema, conteniendo lo más importante, de forma que se tenga una base sobre la que ir añadiendo las funcionalidades.

- Lista de control: Una lista que contiene todas las tareas a realizar. Incluye elementos como las funcionalidades a añadir o las modificaciones que hay que realizar. Esta lista se va actualizando a lo largo de todo el proceso de desarrollo, de forma que se usa de guía para ir realizando las iteraciones.

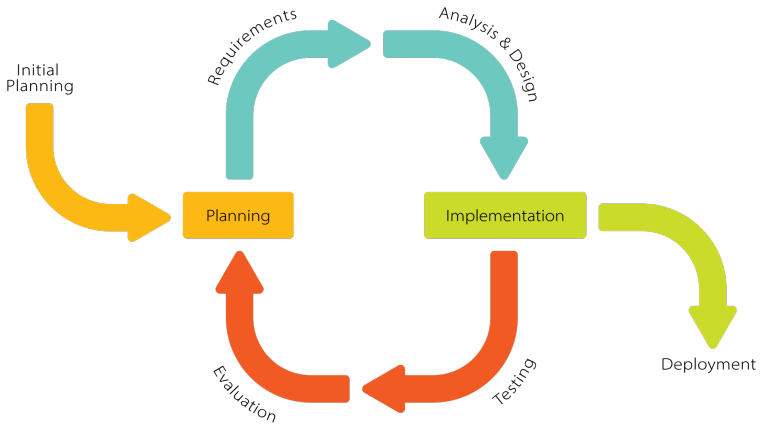

Figura 4.1: Diagrama de la metodología iterativa-incremental
Krupadeluxe - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=104924876

- Iteración: Cada capa o refactorización que se añade sobre la versión anterior del sistema, con el fin de cumplir un mayor número de requisitos y siguiendo la lista de control. No hay especificaciones sobre cómo deben ser las iteraciones, por lo que pueden existir iteraciones más complejas y largas e iteraciones simples de menor duración. También pueden hacerse iteraciones que no cumplan nuevos requisitos, sino que modifiquen el sistema actual por distintos motivos. Por ejemplo, refactorizando el código para seguir un patrón o hacerlo más modular, etc.

4.2. Arquitectura Software

El diagrama de clases de la figura 4.2 permite visualizar la estructura del proyecto. No se muestran en el diagrama los atributos ni los métodos de las clases por simplicidad, al haber demasiados como para mostrarlos en el diagrama. Se ha seguido la convención de utilizar atributos privados en las clases, así como declarar públicos los métodos que se utilizan en otros lugares del programa y privados los que solo utiliza la propia clase.

Para estructurar el proyecto, se ha seguido una arquitectura en 3 capas, que son las siguientes:

- **Capa de presentación**: Es la interfaz gráfica que ve el usuario, que en el caso de este proyecto, se encarga del menú y la pantalla del juego, por lo que es la encargada de mostrar todo por pantalla, preguntando lo necesario a la capa de negocio. En esta capa únicamente está la clase GUI.
- **Capa de negocio**: Esta capa es la que contiene toda la lógica de la aplicación. Se comunica con la capa de presentación para recibir los datos necesarios de la

interfaz gráfica y también con la capa de persistencia para acceder a los datos almacenados. Se encuentran en esta capa las clases **Game Manager**, **Controller**, **Player**, **Laser**, **Alien**, **Mothership** y **Deep Q Agent**, así como todas las que heredan de las mismas.

- **Capa de acceso a datos**: Es donde se almacenan los datos y la lógica para acceder a los mismos. En este proyecto, se encargará únicamente de controlar el acceso a las puntuaciones que se guardan en un fichero. En esta capa solo está la clase **Leaderboard Manager**.

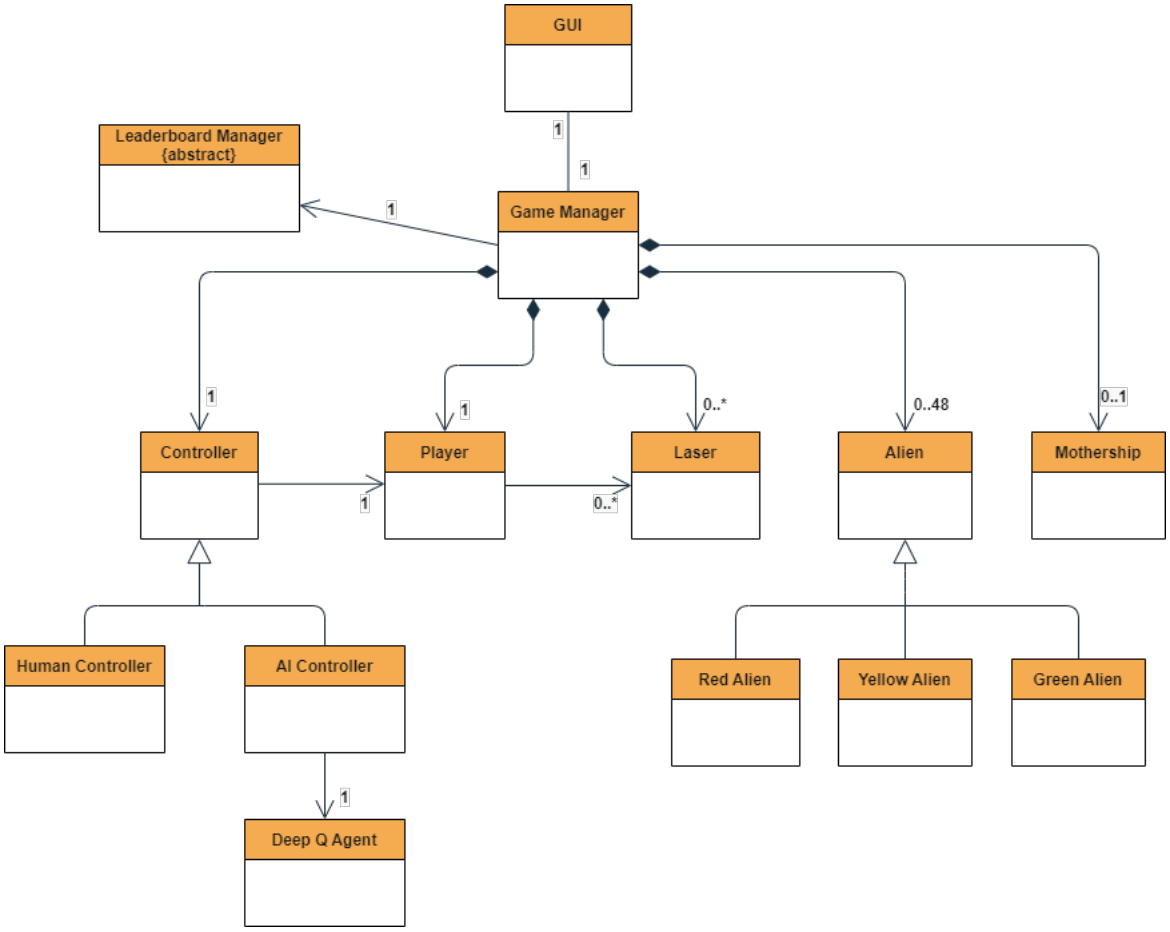

Figura 4.2: Diagrama de Clases Simplificado

4.3. Diseño

En esta sección analizaremos en profundidad las funcionalidades y peculiaridades de cada clase, así como las interacciones entre ellas.

En primer lugar, **Game Manager** es la clase que orquesta la mayor parte de la lógica del videojuego. Se encarga de inicializar los alienígenas, el controlador y el jugador al empezar una partida. En cada iteración del bucle principal, comprueba las colisiones entre objetos y con los bordes, actualiza los estados, posiciones, vidas, etc. También lleva la cuenta de la puntuación, las vidas y los niveles superados además de ocuparse de los temporizadores necesarios, haciendo que los aliens disparen cada cierto tiempo o que la nave nodriza aparezca de vez en cuando. Cuando la partida finaliza, se cambia un parámetro para que la interfaz se dé cuenta la próxima vez que pregunte a esta clase y pueda cambiar a la pantalla correspondiente. En la figura 4.3 se muestra un diagrama de flujo del bucle principal de la clase **Game Manager**. Esta clase está implementada con un patrón Singleton (Gamma et al., 1994), lo que hace que tenga una única instancia y un punto de acceso global a la misma, puesto que no tiene sentido que exista más de un Game Manager en el programa la mismo tiempo.

La clase **Player** representa toda la lógica del jugador, con métodos para moverse y disparar, además de gestionar el temporizador de disparo, para que el jugador tenga que esperar un cierto tiempo entre disparos consecutivos. La clase también contiene el *sprite*[1] del cañón y las coordenadas del jugador en cada momento.

La clase **Controller** es la intermediaria entre el usuario humano o el agente inteligente y el jugador, de forma que es la que decide las acciones del jugador. De esta clase heredan **Human Controller** y **AI Controller**. La primera registra las teclas pulsadas para pasarle la acción correspondiente al **Game Manager**, y la segunda cumple varias funciones, como preprocesar los datos necesarios para pasárselos al agente inteligente, entrenar al agente o recoger su acción y pasársela al **Game Manager**. El preprocesado y entrenamiento del agente inteligente se exponen en las secciones 5.2 Preprocesado y 5.3 Entrenamiento.

Deep Q Agent es la clase que contiene la *Deep-Q-Network*. Tiene los métodos necesarios para entrenar la red y predecir la mejor acción en cada instante cuando la red ya esté entrenada. También abordaremos todo lo relacionado con la red neuronal en el capítulo 5 Agente Inteligente.

La clase **Laser** almacena las coordenadas y velocidades de los rayos láser y contiene el método para actualizar su posición. No distingue si los láseres son del jugador o de los alienígenas; para los láseres del jugador la velocidad es negativa y para los de los alienígenas es positiva, tomando como origen la parte superior de la pantalla. De esta forma, al actualizar la posición solo se deberá sumar la velocidad al la coordenada y, independientemente de que sea del jugador o del alienígena, ya que de las colisiones se encarga la clase **Game Manager**.

[1]Los *sprites* (del inglés "duendecillos") son mapas de bits o imágenes de píxeles en 2 dimensiones que se pueden dibujar en una pantalla de ordenador y que no están obligados a tener una dimensión rectangular.

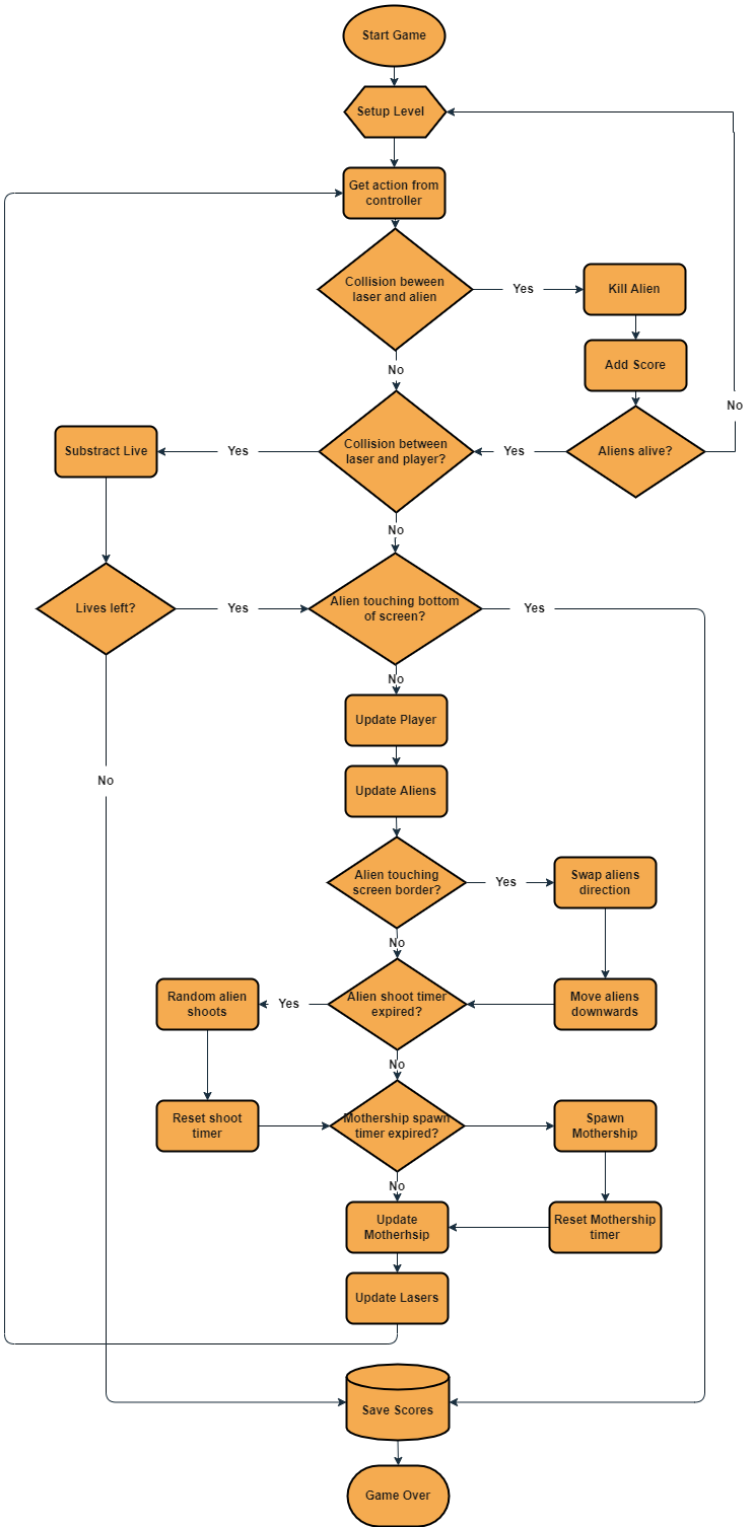

Figura 4.3: Diagrama de Flujo del bucle principal de Game Manager

La clase **Alien**, de la que heredan **Red Alien**, **Yellow Alien** y **Green Alien**, contiene la funcionalidad y atributos comunes a todos los alienígenas, como la posición, la dirección o el valor que da al morir, así como el método de actualizar la posición. Los atributos son sobreescritos por las clases hijas, de forma que al final cada alienígena tiene una imagen, posición y valor propios.

Mothership es la clase que se encarga de la nave nodriza. No hereda de la clase Alien porque tiene un comportamiento diferente, al aparecer por un lado aleatorio de la pantalla e irse por el otro. En la clase se encuentran los atributos necesarios como la velocidad, imagen y dimensiones, y sólo contiene el método para actualizar la posición de la nave.

La clase **Leaderboard Manager** es la que hace de intermediario entre las clases **GUI** y **Game Manager** y el archivo donde se guardan los datos de las mejores puntuaciones. Por ello, esta clase contiene la funcionalidad de guardar y cargar las mejores puntuaciones para que las demás clases no tengan que hacerlo. Para ello se utiliza un archivo CSV, ya que no es necesario programar toda la lógica de acceso a una base de datos para guardar unos nombres y unas puntuaciones. Es una clase abstracta puesto que solo contiene lógica, por lo que no es necesario instanciar un objeto de la clase para realizar operaciones sobre el archivo que contiene los datos.

La clase **GUI** es la encargada de la interfaz gráfica del juego. En primer lugar, muestra el menú principal (Figura 4.4), desde el que se pueden modificar los datos del jugador, mostrar las mejores puntuaciones, comenzar una nueva partida o salir del programa. Al seleccionar la opción "JUGAR", se encarga de pasar los parámetros necesarios a la clase **Game Manager** para que pueda crear una nueva partida. Durante la partida, se encarga de mostrar los *sprites* y la puntuación por pantalla. Para obtener estos datos, la clase **GUI** pregunta a **Game Manager** la información necesaria. También es labor de la clase **GUI** el cambiar entre distintas pantallas, como la del Menú, la de puntuaciones, la pantalla del juego o la pantalla de *Game Over*.Además, contiene el bucle principal de la aplicación, ya que es la primera que se ejecuta al lanzar el programa.

Por último, cabe destacar que además de las clases mencionadas, existe un fichero de configuración que contiene todos los parámetros necesarios para la ejecución, de forma que si se quiere modificar alguno no haya que buscar dónde se encuentra en el código. En este fichero se encuentran parámetros como las dimensiones y posiciones iniciales de los *sprites*, los fotogramas por segundo a los que correrá el juego, las velocidades del jugador, láser, alienígenas, etc. También hay parámetros para mostrar mensajes de *debug* o un parámetro que decide si la IA está en modo entrenamiento o no.

Figura 4.4: Menú principal del juego

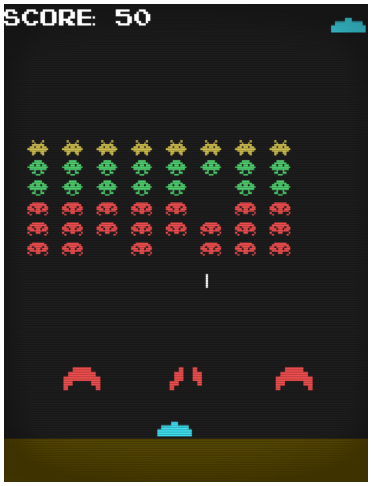

Figura 4.5: Pantalla del Juego

4.4. Cronograma de Iteraciones

En el paso de inicialización, se ha creado una primera versión del entorno de juego, cumpliendo los requisitos **RF.1** y **R.N.F.1**. En esta versión del entorno se contaba con una pestaña en la que se podía mover un jugador de lado a lado y disparar un rayo láser. Puede parecer poco, pero el hecho de crear una pestaña y el bucle principal de la aplicación no fue trabajo sencillo. También se generó una primera versión del diagrama de clases, que fue imprescindible para ir generando el código de una manera limpia y ordenada. Se crearon los distintos módulos de Python para organizar las clases y se creó la estructura del proyecto, con directorios para los datos, los recursos necesarios, como imágenes o fuentes, y otra carpeta para el código. También en este paso se creó un repositorio de Git donde se iría actualizando el código y todo lo necesario para trabajar en el proyecto desde cualquier lugar, disponible en https://github.com/jairusgz/TFG/. La figura 4.6 muestra el número de requisitos (funcionales, no funcionales y totales cumplidos al final de cada iteración).

En la primera iteración, se cumplieron los requisitos **R.F.2**, **R.F.3**. Se generaron los alienígenas, con un movimiento de extremo a extremo de la pantalla y hacia abajo cuando tocan los extremos.

En la segunda iteración se programó que los alienígenas dispararan tras un tiempo aleatorio, y se creó toda la lógica de colisiones. Los láseres de los alienígenas colisionarían con el jugador, los del jugador con los alienígenas y los láseres colisionarían entre sí. De esta forma, si un láser del jugador colisionaba contra un láser de un alienígena, el láser del jugador se destruiría mientras que el del alienígena tendría una posibilidad de seguir su trayectoria. Esto cumplía el requisito **R.F.5**.

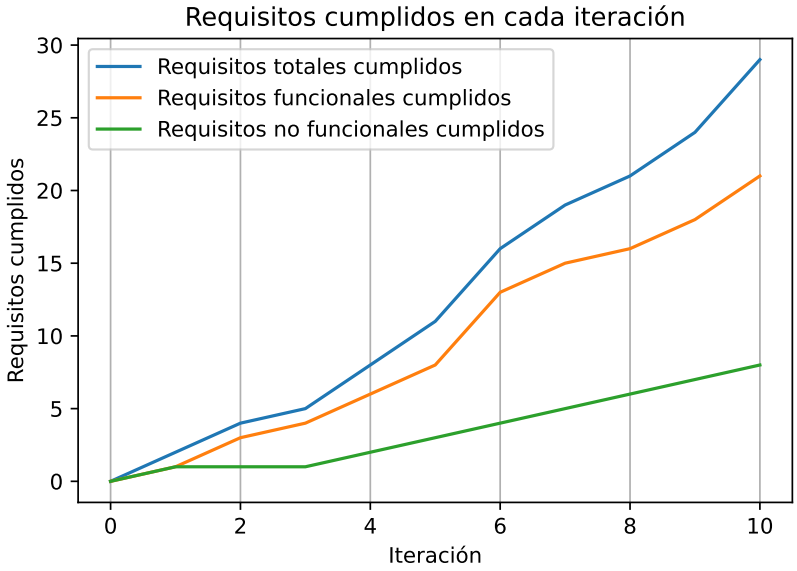

Figura 4.6: Gráfico de los requisitos totales, funcionales y no funcionales cumplidos en cada iteración, siendo la iteración 0 el paso de inicialización

Con la tercera iteración se cumplieron los requisitos **R.F.8**, **R.F.20** y **R.N.F.5**. Se generó el código encargado de la nave nodriza, que aparecería de un extremo aleatorio de la pantalla y se iría por el otro extremo. También se crearon los obstáculos y se añadieron, junto a la nave nodriza, a la lógica de colisiones.

En la cuarta iteración se programó el sistema de puntuación y el sistema de vidas, de forma que ahora las colisiones de los láseres con los alienígenas otorgaban puntos al jugador, y las colisiones con el jugador restaban vidas. Cuando el número de vidas llegaba a 0 o los alienígenas tocaban el extremo inferior de la pantalla, se finalizaba el programa. Esto cumplía los requisitos **R.F.7**, **R.F.21** y **R.N.F.7**

En la quinta iteración se crearon las pantallas de menú y Game Over, acorde a los requisitos **R.F.9**, **R.F.10**, **R.F.11** y **R.F.14**, **R.F.15**.

En la sexta iteración se realizaron diversas mejoras en la jugabilidad y en el código. En primer lugar, se refactorizó el código para reorganizar todo lo relacionado con la interfaz de la lógica del juego en dos clases y se pasaron todas las constantes de las clases a un mismo fichero de forma que el código fuera mucho más limpio y manejable. En cuanto a las mejoras de jugabilidad, se cambiaron algunos temporizadores fijos por temporizadores aleatorios, se hizo que el juego fuera aumentando de velocidad a medida que quedaran menos alienígenas y se crearon múltiples niveles con mayor velocidad cada uno. También se ajustaron las velocidades, retardos, etc, para ajustar la dificultad general del juego. Esto cumplía los requisitos **R.F.6**, **R.F.16** y **R.N.F.5**

En la séptima iteración se codificó la lógica del sistema de mejores puntuaciones, según los requisitos **R.F.13** y **R.F.22**.

En la octava iteración se programó, entrenó y probó el agente inteligente que jugara al videojuego, cumpliendo los requisitos **R.F.12**, **R.N.F.2** y **R.N.F.3**.

Por último, en la novena iteración se realizaron muchas mejoras del aspecto del juego, como el hacer que parezca más antiguo o el añadir la música y sonidos necesarios. Con esto se cumplen los requisitos **R.F.17**, **R.F.18**, **R.F.19**, **R.N.F.4** y **R.N.F.6**, lo que completa el ciclo de diseño del proyecto, cumpliendo la totalidad de los requisitos preestablecidos.

4.5. Pruebas Realizadas

A medida que se iban incorporando funcionalidades al entorno del juego se han ido probando para poder seguir la metodología iterativa incremental de forma correcta. Es decir, antes de incorporar una nueva funcionalidad, se ha probado que todo lo anterior funciona correctamente.

Como se trata de un videojuego en el que las posibilidades son infinitas, no se han diseñado pruebas unitarias, sino que las pruebas han sido realizadas manualmente interactuando con el entorno como lo haría cualquier otro jugador, modificando el código cuando ha sido necesario; por ejemplo, para probar el paso de un nivel a otro en la sexta iteración, se eliminaron las colisiones de los láseres con el jugador de forma que era imposible perder vidas.

Por otro lado, para realizar las pruebas con el agente inteligente, se entrenó durante un número limitado de fotogramas en un ordenador de sobremesa para comprobar que se le estaban pasando los datos correctos o que estaba produciendo la salida esperada. Por ejemplo, al probar el agente inteligente durante un pequeño número de fotogramas, se pudo comprobar que el agente no estaba mejorando su puntuación. Tras una búsqueda exhaustiva en el código se vio que al agente se le estaba pasando la matriz de píxeles de la pantalla del juego antes de dibujar todos los *sprites*, por lo que recibía una imagen completamente vacía.

Por último, también se han probado casos extremos, como ejecutar el juego a una alta tasa de fotogramas por segundo o a comenzar y finalizar un gran número de partidas para comprobar que no se dejaban procesos en segundo plano. Ejecutando el juego a altas tasas de fotogramas por segundo se podía observar que el jugador disparaba demasiado lento en comparación con la velocidad del juego. Esto se producía porque en ocasiones el programa se ralentizaba, sobre todo cuando se estaba entrenando la red. Este fallo se solucionó haciendo que los temporizadores llevaran la cuenta de los fotogramas en vez de los segundos, siguiendo así la velocidad real a la que se ejecutaba el juego.

Capítulo 5

Agente Inteligente

En este capítulo se expondrá todo lo relacionado con el agente inteligente entrenado para jugar al videojuego; desde el modelo utilizado, el preprocesado de los datos y los algoritmos de entrenamiento utilizados para crear el modelo final.

5.1. Modelo: Deep-Q Network

Para el modelo del agente inteligente se ha utilizado, siguiendo los pasos de (Mnih et al., 2013) y (Mnih et al., 2015), un tipo especial de red neuronal conocida como Deep-Q Network (DQN), que combina técnicas de aprendizaje por refuerzo con redes neuronales profundas. Esta red neuronal profunda utiliza una estructura conocida como red neuronal convolucional, capaz de explotar las correlaciones espaciales locales presentes en las imágenes y adaptarse para detectar estas correlaciones incluso en otra escala, posición o punto de vista. Analizaremos esta estructura en profundidad, ya que es clave entender cómo funcionan este tipo de redes para poder entender la totalidad de nuestro modelo.

5.1.1. Red Neuronal Convolucional

Las redes neuronales convolucionales (CNN) (Albawi et al., 2017) son un tipo de red neuronal profunda que se utiliza en el campo de reconocimiento de patrones en imágenes, permitiendo codificar ciertos patrones específicos de las imágenes en la arquitectura de la red. Esto se debe a que las redes neuronalestradicionales requerirían de un esfuerzo computacional excesivo, dada la alta dimensionalidad de los datos de entrada. Por ejemplo, una imagen en HD de hoy en día tiene una dimensión de 1920x1080 píxeles, cada uno con 3 canales RGB, sumando 6.220.800 parámetros para una única neurona. Si la primera capa oculta tuviera 2 neuronas, este valor se multiplicaría por 2, y así sucesivamente. Para entrenar una red neuronal tradicional con imágenes como esta se necesitarían muchas más capas ocultas en la estructura, lo que supone una gran carga computacional y un

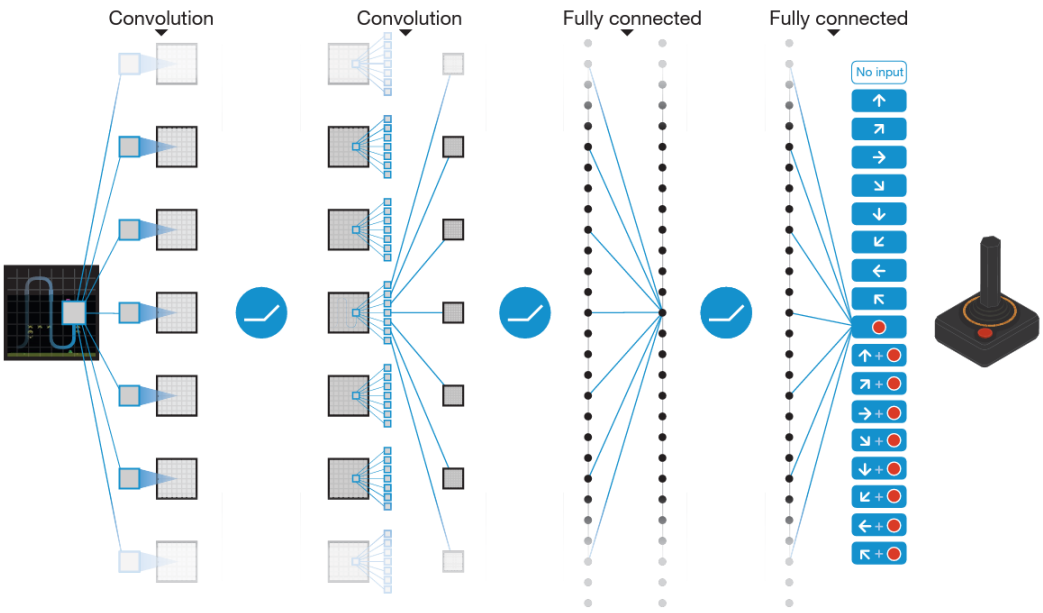

Figura 5.1: Arquitectura de CNN utilizada (Mnih et al., 2015)

tiempo de entrenamiento que no es viable en la mayoría de casos. Además, esto haría que el modelo se sobreajustase a los datos, obteniendo peores resultados para datos nuevos. En cambio, las redes neuronales convolucionales aprovechan la correlación espacial de los píxeles que se encuentran próximos en la imagen para reducir la carga computacional.

Las CNN son todas aquellas redes que tienen al menos una capa convolucional, como la red de la figura 5.1, así que para entenderlas introduciremos el concepto de convolución (Goodfellow et al., 2016). Comenzamos suponiendo que, como la entrada de la red es una imagen, esta tendrá una anchura, altura y profundidad, como se puede apreciar en la figura 5.2. En este caso, la anchura y la altura son las dimensiones de la imagen y la profundidad se puede corresponder con varios parámetros, como el número de canales de color (3 si es RGB o 1 si es escala de grises) o el número de imágenes, si nuestra entrada es, por ejemplo, un vídeo.

Las capas convolucionales son las que utilizan una operación matemática llamada convolución en lugar de la multiplicación de matrices. La convolución es una operación matemática entre dos funciones que produce como resultado una nueva función, según la siguiente fórmula:

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau,$$

donde el operador $*$ indica la operación de convolución:

$$s(t) = (f * g)(t).$$

En las CNN, el primer argumento (f) es la entrada de la capa convolucional y el se-

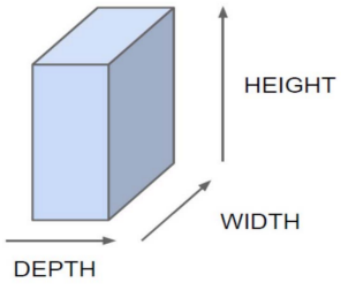

Figura 5.2: Entrada de una CNN

gundo (g) es el denominado *kernel*, que no es más que otra función matemática. τ es lo que Godfellow denomina edad de la medida, por lo que $t - \tau$ significa que se le da mayor importancia a las medidas más recientes. La salida se suele conocer como mapa de características o *feature map*.

Como trabajar con datos continuos en computación no es posible, normalmente discretizamos los valores, por lo que, si τ solo puede ser un valor entero, por ejemplo, supondremos que las funciones f y g solo están definidas para valores enteros. Definimos así la convolución discreta como:

$$(f * g)(t) = \sum_{\tau=-\infty}^{\infty} f(\tau)g(t-\tau).$$

En las CNN, la entrada suele ser un array multidimensional de datos y el *kernel* es un array multidimensional de parámetros que se va modificando por el algoritmo de entrenamiento. Como es necesario almacenar cada elemento de la entrada y del *kernel* por separado, se asume que estas funciones tienen siempre valor 0 excepto en el conjunto finito en el que guardamos los valores. De esta forma, podemos implementar el sumatorio infinito como el sumatorio de un número finito de elementos de un array.

También se puede utilizar la convolución en más de un eje al mismo tiempo. Por ejemplo, si utilizamos una imagen bidimensional como entrada, normalmente utilizaremos un *kernel* bidimensional:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n).$$

A la hora de realizar la convolución sobre una imagen, existe un parámtro conocido como paso o *stride*. Este define el número de posiciones que se moverá el kernel tras cada operación, como en la imagen 5.3.

Por último, destacar que las capas convolucionales de las CNN utilizan múltiples filtros, que no son más que agrupaciones de *kernels*, donde cada *kernel* está asignado a un canal

Figura 5.3: Resultado de aplicar un kernel con paso=2

de la imagen de entrada. Los filtros siempre tienen una dimensión más que los *kernels*. Por ejemplo, para realizar una convolución sobre una imagen en 2 dimensiones, se utilizan filtros de 3 dimensiones, que agrupan *kernels* de 2 dimensiones.

5.1.2. Arquitectura de la Red Neuronal Utilizada

La arquitectura de la red que hemos utilizado, que se puede ver en las figuras 5.1 y 5.4, es la de una red neuronal convolucional, con una entrada de 84x84x4, es decir, 4 imágenes de dimensión 84x84. La primera capa se compone de 32 filtros de 8x8 con paso 4 y aplica la función de activación ReLU (figura 2.2). La segunda capa utiliza 64 filtros de 4x4 con paso 2, también aplicando la función ReLU. La tercera capa utiliza 64 filtros de 3x3 con paso 1 y activación ReLU. La salida de esta tercera capa convolucional se pasa a una cuarta capa *Flatten*, que aplana la entrada. Es decir, la transforma para que tenga una única dimensión. La última capa oculta es una capa *Fully Connected* con 512 neuronas que, de nuevo, aplican la función ReLU. Por último, la capa de salida tiene 6 neuronas, una por cada posible acción. De esta forma, las capas convolucionales se encargan de reducir la dimensión de la imagen y detectar las relaciones entre los píxeles próximos, mientras que las capas *Fully Connected* son las que explotarán esas relaciones para aprender a predecir los valores de Q.

Otra forma de implementar la red sería tener el estado y la acción como entrada de la red, y una única neurona que devolviera el valor de Q para ese estado-acción, pero esto significaría que se necesite computar la red una vez por cada posible acción, lo que haría que el coste computacional escalara con el número de acciones.

En conclusión, esta red neuronal será la utilizada para aproximar la función Q, recibiendo un estado que se le pasa como entrada y devolviendo los valores de Q para cada posible acción. Después utilizaremos estos valores para escoger la mejor acción en cada estado, es decir, la que tenga un mayor valor de Q o, lo que es lo mismo, una mayor recompensa futura.

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	[(None, 84, 84, 4)]	0
conv2d (Conv2D)	(None, 20, 20, 32)	8224
conv2d_1 (Conv2D)	(None, 9, 9, 64)	32832
conv2d_2 (Conv2D)	(None, 7, 7, 64)	36928
flatten (Flatten)	(None, 3136)	0
dense (Dense)	(None, 512)	1606144
dense_1 (Dense)	(None, 6)	3078

Total params: 1,687,206
Trainable params: 1,687,206
Non-trainable params: 0

Figura 5.4: Información de la estructura de la red generada por Keras

5.2. Preprocesado

Como hemos mencionado anteriormente, una imagen tiene una enorme cantidad de parámetros, por lo que el preprocesado tiene como objetivo reducir este número para que la DQN pueda entrenarse de manera eficiente. También analizaremos el salto de fotogramas o *Frame Skipping* y el agrupamiento de forogramas o *Frame stacking*, dos técnicas que hemos usado para preprocesar la entrada de la red.

5.2.1. Preprocesado de las Recompensas

En primer lugar, se ha realizado un procesado de las recompensas antes de pasárselas a la red neuronal. Como pasarle la puntuación sin procesar puede hacer que la red acabe prediciendo valores exageradamente altos de Q, se han normalizado las recompensas para que estén en el intervalo $[-1, 1]$. Como el juego no tiene puntuaciones negativas, se le ha asignado una recompensa de $-0{,}3$ a cada vida perdida y una recompensa de $-0{,}001$ a cada fotograma que pasa, de forma que el agente trate de eliminar los alienígenas antes de que lleguen al borde inferior de la pantalla. Las recompensas positivas se otorgan al destruir alienígenas o la nave nodriza, según la tabla de la tabla 5.1

Recompensa	Acción
-0.3	Vida perdida
-0.001	Siempre
0.1	Alienígena rojo
0.2	Alienígena verde
0.3	Alienígena amarillo
1	Nave nodriza

Tabla 5.1: Recompensas procesadas

5.2.2. Preprocesado de la Imagen

Se ha recortado la imagen para que sea cuadrada, ignorando los márgenes superior e inferior de la pantalla, ya que no contienen información relevante para la red. La parte superior muestra el número de vidas y la puntuación y el borde inferior no es más que una superficie marrón que representa el planeta. La imagen original es de dimensión 480x630, y el recorte nos deja una imagen de 480x480. Después, se ha reescalado a 84x84 utilizando las propias funcionalidades que ofrece la librería Pygame.

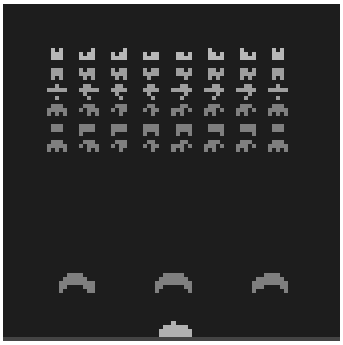

Figura 5.5: Fotograma del juego después de aplicar el preprocesado

Después, se ha pasado la imagen a escala de grises, para reducir el número de canales de color de 3 canales RGB a un único canal con valores de 0 a 255. Este preprocesado ha logrado reducir el número de parámetros de 480x630x3x4 (recordamos que a la red se le pasan 4 imágenes como entrada) a 84x84x4, dando como resultado imágenes como las de la figura 5.5.

5.2.3. Salto de Fotogramas

Para reducir la cantidad de cómputo necesaria para entrenar la red, se ha implementado una técnica de salto de fotogramas, que consiste en que la red escoge una acción cada 4 fotogramas y la acción se repite durante los próximos 4 fotogramas. De esta forma, solo

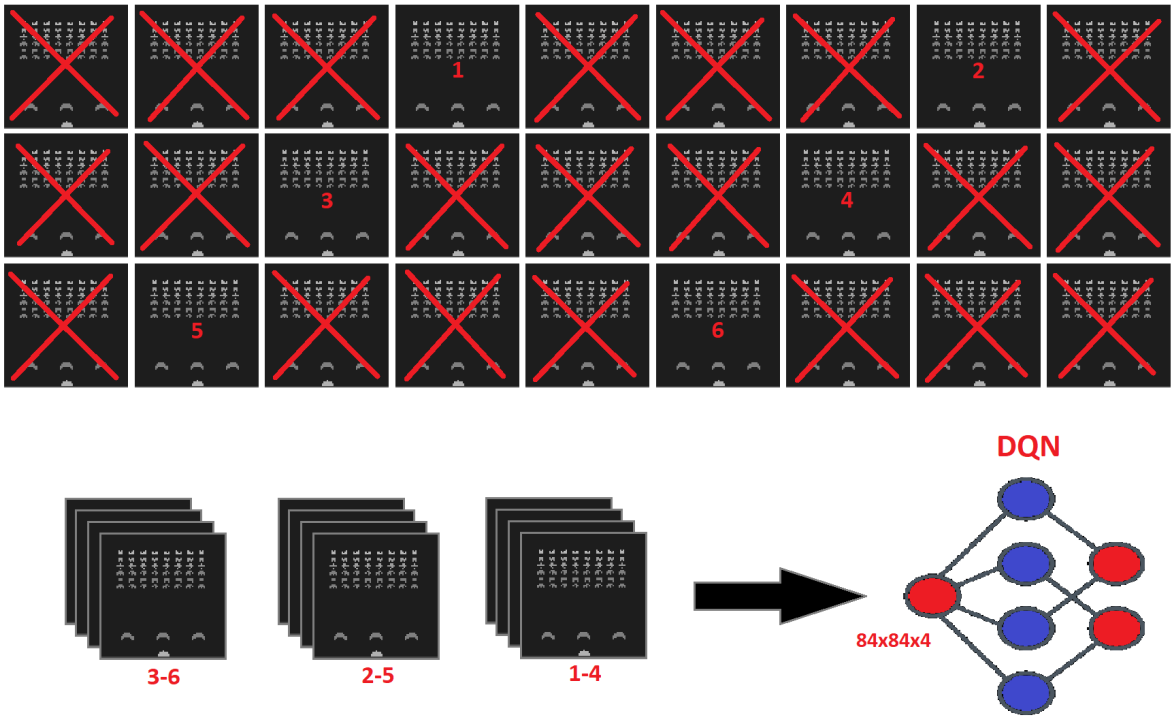

Figura 5.6: Salto y agrupación de fotogramas

se entrena la red cada 4 fotogramas y apenas se pierde información, ya que la mayoría de la información se mantiene de un fotograma al siguiente. La recompensa que se le pasa a la red es la suma de la recompensa obtenida en los 4 fotogramas. La figura 5.6 ilustra este salto de fotogramas.

5.2.4. Agrupación de Fotogramas

También utilizamos una técnica de agrupar fotogramas, agrupando los últimos 4 fotogramas después de aplicar el salto de fotogramas. Esto se hace para que la red pueda aprender del movimiento de los objetos de la imagen. Con un único fotograma, sería imposible saber si un alienígena se está moviendo de izquierda a derecha o al revés, o si un láser va hacia el jugador o hacia los alienígenas. Sin embargo, al aplicar esta técnica y agrupar los fotogramas de 4 en 4, la red tiene en cuenta este movimiento para el entrenamiento y por tanto es capaz de detectarlo. Como se puede ver en parte inferior de la figura 5.6, al agrupar las imágenes algunas se solapan.

5.3. Entrenamiento

5.3.1. Algoritmo de Entrenamiento

El algoritmo de entrenamiento de la red DQN se presenta en el Algoritmo 5, donde el preprocesado descrito en la sección anterior se denota como ϕ. El agente selecciona y ejecuta acciones según una política ϵ-voraz basada en Q. El algoritmo modifica el *Q-Learning* del Algoritmo 3 con aproximación de función utilizando dos técnicas distintas, para adaptarlo al entrenamiento de redes neuronales de gran tamaño sin que llegue a diverger.

Algoritmo 5 Deep Q-learning con memoria de experiencias, (Mnih et al., 2015)

Input: Los píxeles de la pantalla y la puntuación del juego
Output: Q, la función de valor(de donde se obtiene la política y se selecciona una acción)
  Inicializar la memoria de experiencias D
  Inicializar la función de valor Q con pesos aleatorios θ
  Inicializar la función de valor objetivo $\hat{Q}$ con pesos $\theta^- = \theta$
  for *episodio = 1 hasta M* **do**
    Inicializar la secuencia $s_1 = \{x_1\}$ y la secuencia preprocesada $\phi_1 = \phi(s_1)$
    for $t = 1$ hasta T **do**
      Según política ϵ-voraz, escoger $a_t = \begin{cases} \text{acción aleatoria}, & \text{probabilidad } \epsilon \\ \arg\max_a Q(\phi(s_t), a; \theta), & \text{en otro caso} \end{cases}$
      Ejecutar la acción a_t y observar la recompensa r_t y la imagen x_{t+1}
      Fijar $s_{t+1} = s_t, a_t, x_{t+1}$ y preprocesar $\phi_{t+1} = \phi(s_{t+1})$
      Guardar la transición $(\phi_t, a_t, r_t, \phi_{t+1})$ en D
      Extraer un lote de transiciones aleatorio $(\phi_j, a_j, r_j, \phi_{j+1})$ de D
      $y_j = \begin{cases} r_j, & \text{si el episodio acaba en } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-), & \text{en otro caso} \end{cases}$
      Realizar un paso de gradiente descendente en $(y_j - Q(\phi_j, a_j; \theta))^2$ en relación a θ
      Cada C iteraciones resetear $\hat{Q} = Q$ y fijar $\theta^- = \theta$
    end for
  end for

La primera técnica es la repetición de experiencias. En esta técnica, guardamos las experiencias del agente en cada paso $e_t = (s_t, a_t, r_t, s_{t+1})$ en una memoria de experiencias $M_t = \{e_1, ..., e_t\}$. Durante el bucle interno del algoritmo, aplicamos actualizaciones de *Q-learning* a lotes de experiencias (s, a, r, s') que recogemos aleatoriamente de la memoria de experiencias. Así, al no trabajar con muestras consecutivas, se rompe la correlación y se reduce la varianza de las actualizaciones. También esto hace que cada experiencia se pueda utilizar más de una vez, por lo que se aumenta la eficiencia de los datos.

La segunda técnica consiste en utilizar una red neuronal distinta para generar los objetivos y_j en cada actualización del *Q-learning*. Cada C actualizaciones, clonamos la red actual Q para obtener una nueva red separada $\hat{Q}$ y utilizamos esta nueva red para generar y_j en las próximas C actualizaciones. Esta modificación hace que el entrenamiento sea más estable, al añadir un retardo entre el momento en el que actualizamos la función de valor Q y el momento en el que actualizamos los parámetros y_t. Así, se reduce la correlación entre los valores de acción Q y el objetivo $r + \gamma \max_{a'} Q(s', a')$. La función de pérdida que se utiliza para actualizar los parámetros de la red en la iteración t viene dada por siguiente fórmula:

$$L_t(\theta_t) = (r + \gamma \max_{a'} Q(s', a'; \theta_t^-) - Q(s, a, \theta_t))^2,$$

donde θ_t son los parámetros de la DQN en la iteración t y θ_t^- son los parámetros de la red objetivo en la iteración t.

En la tabla 5.2 se encuentran los hiperparámetros utilizados para el entrenemiento de la red.

5.3.2. Proceso de Entrenamiento e Infraestructura

Para entrenar una red neuronal tan grande y con tantos datos, resulta inviable ejecutar el código en un portátil o en un PC de sobremesa corriente. Por ello, para ejecutar el proceso de entrenamiento se contactó tanto con el Instituto de Física de Cantabria (IFCA) como con el Grupo de Ingeniería de Computadores de la UC, con el objetivo de poder utilizar sus infraestructuras.

Por un lado, el IFCA proporcionó acceso a un nodo de cómputo del supercomputador Altamira[1], pero que resultó insuficiente, ya que sólo contaba con varios procesadores y el entrenamiento requiere utilizar tarjetas gráficas (GPUs). Esto se debe a que las GPUs tienen una mayor eficiencia en operaciones de coma flotante y una mayor capacidad de paralelización, lo que las hacen idóneas para tareas que requieren un gran número de estas operaciones, como realizar operaciones con matrices, que es lo que hace el entrenamiento de una red neuronal.

El Grupo de Ingeniería de Computadores, por otro lado, proporcionó acceso a un nodo del clúster HPC Calderon, en el CPD 3Mares, permitiendo utilizar una GPU Nvidia Quadro RTX 4000[2], que cuenta con 8GB de memoria VRAM y una capacidad de cómputo de 7.46 TFLOPS, suficiente para las necesidades de cómputo de este proyecto.

El nodo utiliza un sistema operativo Debian 4.9.210-1 y se ha accedido a través de

[1] https://www.res.es/es/nodos-de-la-res/altamira
[2] https://www.geektopia.es/es/product/nvidia/quadro-rtx-4000/

Hiperparámetros	**Valor**	**Descripción**
Tamaño de lote	32	Nº de muestras sobre las que se se calcula cada paso de gradiente descendente estocástico
Tamaño de la memoria de experiencias	12000	Tamaño de la memoria de experiencias de la que se obtienen las muestras para entrenar
Frecuencia de actualización de red objetivo	10000	La frecuencia con la que se actualiza la red objetivo (C, en el algoritmo 5)
Factor de descuento γ	0.99	Factor de descuento utilizado en cada actualización de *Q-learning*
Repetición de acciones	4	Nº de fotogramas en los que el agente repite la última acción escogida
Tasa de aprendizaje	0.00025	Tasa de aprendizaje utilizada por el optimizador en el entrenamiento de la red
Valor inicial de exploración	1	Valor inicial de ϵ en la política de exploración ϵ-voraz
Valor final de exploración	0.1	Valor final de ϵ en la política de exploración ϵ-voraz
Decrecimiento de ϵ	0.995	Valor de decrecimiento de ϵ en la política ϵ-voraz
Ultimo fotograma de exploración	1000000	Ultimo fotograma en el que decrece el valor de ϵ
Optimizador utilizado	Adam	Optimizador utilizado para entrenar la red

Tabla 5.2: Hiperparámetros utilizados para el entrenamiento de la red

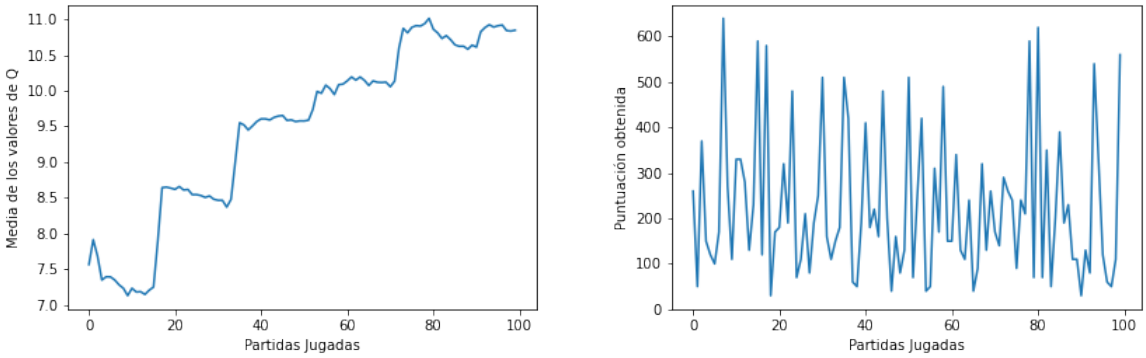

Figura 5.7: Puntuaciones y valores de Q en las primeras partidas del entrenamiento

SSH, utilizando una VPN para poder acceder a la red de la universidad y así al nodo asignado.

5.4. Resultados Obtenidos

Los resultados que se describirán a continuación han sido obtenidos tras un entrenamiento de unos 2M de fotogramas, ejecutado durante unos 3 días en el nodo proporcionado del clúster Calderón.

En primer lugar, la primera imagen de la figura 5.7, muestra una línea que sigue la forma de unos escalones, debido a que éstos se corresponden con la actualización de la red separada $\hat{Q}$, que comienza a predecir unos valores más altos de Q cuando se actualiza con los pesos de Q. Por otro lado, estos valores aún no se reflejan en las puntuaciones obtenidas, que forman una gráfica bastante ruidosa al comienzo del entrenamiento, cuando la política ϵ voraz aún escoge un gran número de acciones aleatorias en lugar de las que otorgan un valor de Q máximo.

Es complicado obtener conclusiones a partir de los datos del entrenamiento, debido a la técnica de repetición de experiencias ya mencionada. Esto se debe a la red se está entrenando con muestras de la memoria de experiencias, que pueden ser de partidas anteriores. Por eso, nos centraremos en los resultados obtenidos tras completar el entrenamiento. Compararemos las puntuaciones obtenidas por el agente inteligente con las de 3 jugadores humanos distintos. Dos de ellos están acostumbrados a jugar a videojuegos y el restante no.

Como se muestra en la figura 5.8, los jugadores humanos H1 y H3 han obtenido una media de entre 600 y 700 puntos, ya que son los que están familiarizados con el mundo

de los videojuegos. El jugador H2 ha obtenido una puntuación media de 406 puntos. Por otro lado, el agente inteligente ha superado a los jugadores humanos, obteniendo una puntuación media de 778 puntos. Teniendo en cuenta las puntuaciones de los 3 jugadores humanos, el agente inteligente ha obtenido una puntuación media un 37 % mejor que la de un humano promedio.

Sin embargo, como mencionamos en la sección 1.2 Space Invaders, en el juego existe un ranking de puntuaciones, por lo que no sería justo comparar las puntuaciones medias cuando en el juego original lo que importa es quién ha obtenido la mejor puntuación para estar en las mejores posiciones de la tabla. Por ello, también se muestra en la figura 5.8 un gráfico con las mejores puntuaciones obtenidas por cada jugador, donde en este caso el primer jugador humano H1 ha obtenido la mejor puntuación de 1300 puntos, seguido del agente inteligente con una mejor puntuación de 1030 puntos.

La estrategia seguida por los jugadores a la hora de jugar ha sido bastante parecida, tratando de eliminar todos los alienígenas posibles de forma más o menos aleatoria, moviéndose de un lado a otro junto a los alienígenas sin parar de disparar, y disparando a la nave nodriza siempre que sea posible. Por otro lado, el agente inteligente ha desarrollado una estrategia distinta; al comenzar la partida, destruye el escudo central hasta que crea un agujero a través del que luego dispara para eliminar a los alienígenas del centro de la pantalla. Posteriormente, se moverá hacia los laterales para seguir destruyendo enemigos, siempre sin dejar de disparar y centrándose en destruir la nave nodriza cuando aparece.

Recordamos que al agente no se le ha otorgado la información de que disparar siempre es la mejor acción o que destruir la nave nodriza es lo que más puntos da, sino que lo ha aprendido mediante prueba y error, por lo que se podría decir que el agente ha aprendido a jugar por sí mismo.

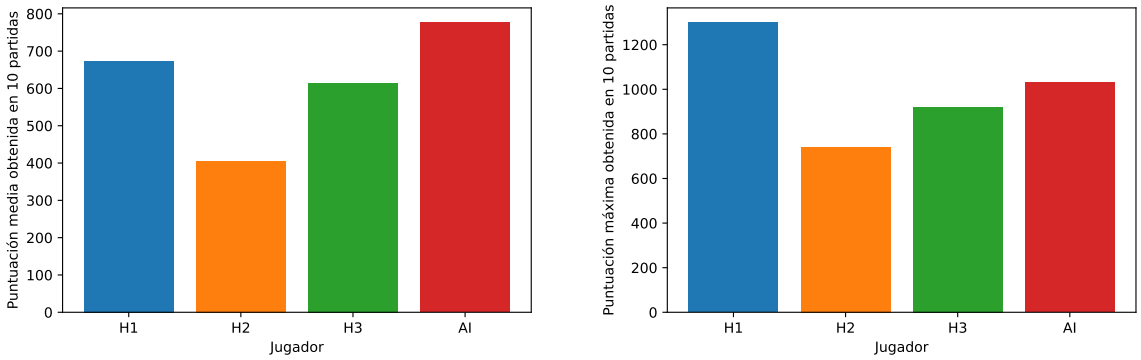

Figura 5.8: Puntuaciones medias y máximas obtenidas por los jugadores humanos y el agente inteligente en 10 partidas

Capítulo 6

Conclusiones

En primer lugar, destacar que se ha adquirido un conocimiento general en temas como aprendizaje profundo y aprendizaje por refuerzo que no se cursan en el grado, por lo que se ha necesitado de un proceso de investigación bastante amplio. Se han aprendido conceptos como aprendizaje por refuerzo, *Q-learning* o red neuronal convolucional y se han llevado a la práctica junto a conocimientos que sí se adquieren en la carrera, como redes neuronales artificiales, aprendizaje automático o programación en Python.

Se han cumplido los objetivos propuestos en la sección 1.3 Objetivo y Motivación, programando un entorno de juego en el que han podido jugar tanto jugadores humanos como el agente inteligente. Se ha programado una *Deep Q-Network* y se ha analizado su desempeño en el entorno programado, comparando los resultados con los obtenidos por jugadores humanos en el mismo entorno, llegando a la conclusión de que el agente inteligente ha alcanzado un nivel humano en el videojuego Space Invaders.

Todo el código fuente del proyecto está disponible en https://github.com/jairusgz/TFG junto a un vídeo de demostración del agente inteligente en el entorno y una guía de instalación y ejecución para todo aquel que desee probar el entorno de juego o ver cómo juega el agente inteligente. También se encuentra en la raíz del proyecto un cuaderno de Jupyter donde se han analizado los resultados y se han generado algunas de las figuras utilizadas en la memoria.

Bibliografía

Albawi, S., Mohammed, T. A., and Al-Zawi, S. (2017). Understanding of a convolutional neural network. In *2017 International Conference on Engineering and Technology (ICET)*, pages 1–6.

Berner, C., Brockman, G., Chan, B., Cheung, V., Debiak, P., Dennison, C., Farhi, D., Fischer, Q., Hashme, S., Hesse, C., et al. (2019). Dota 2 with large scale deep reinforcement learning. *arXiv preprint arXiv:1912.06680*.

Bottou, L. et al. (1991). Stochastic gradient learning in neural networks. *Proceedings of Neuro-Nımes*, 91(8):12.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Pearson Education.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press. `http://www.deeplearningbook.org`.

Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. In Bach, F. and Blei, D., editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 448–456, Lille, France. PMLR.

Larman, C. and Basili, V. (2003). Iterative and incremental developments. a brief history. *Computer*, 36(6):47–56.

Li, Y. (2018). Deep reinforcement learning. *CoRR*, abs/1810.06339.

Luzgin, R. (2019). Video games as a perfect playground for artificial intelligence. *Towards Data Science*. `https://towardsdatascience.com/video-games-as-a-perfect-playground-for-artificial-intelligence-3b4ebeea36ce`.

Martinez, M., Sitawarin, C., Finch, K., Meincke, L., Yablonski, A., and Kornhauser, A. (2017). Beyond Grand Theft Auto V for training, testing and enhancing deep learning in self driving cars. *arXiv preprint arXiv:1712.01397*.

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., and Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937. PMLR.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. (2013). Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A., Veness, J., Bellemare, M., Graves, A., Riedmiller, M., Fidjeland, A., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, 518:529–33.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, 323(6088):533–536.

Russell, S. and Norvig, P. (2021). *Artificial Intelligence. A Modern Approach.* Pearson, 4th edition.

Silver, D. (2015). Lectures on reinforcement learning. *URL: https://www. davidsilver. uk/teaching.* visitado en junio de 2022.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research.*, 15(1):1929–1958.

Sun, Y., Huang, X., and Kroening, D. (2018). Testing deep neural networks. *CoRR*, abs/1803.04792.

Sutton, R. S. and Barto, A. G. (2018). *Reinforcement Learning: An Introduction.* The MIT Press, second edition.

Teoh, T. T. and Rong, Z. (2022). *Python for Artificial Intelligence.* Springer Singapore, Singapore.

van Hasselt, H., Guez, A., and Silver, D. (2016). Deep reinforcement learning with double Q-learning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 30(1).