eman ta zabal zazu
Universidad del País Vasco
Euskal Herriko Unibertsitatea
INFORMATIKA FAKULTATEA
FACULTAD DE INFORMÁTICA

Grado en Ingeniería Informática

Computación

Trabajo de Fin de Grado

Agentes inteligentes en videojuegos

Autor/a

Tai Nuño Múgica

2019

eman ta zabal zazu
Universidad del País Vasco
Euskal Herriko Unibertsitatea
INFORMATIKA FAKULTATEA
FACULTAD DE INFORMÁTICA

Grado en Ingeniería Informática

Computación

Trabajo de Fin de Grado

Agentes inteligentes en videojuegos

Autor/a

Tai Nuño Múgica

Directore/a(s)

Manuel Graña

Resumen

Este proyecto tiene como objetivo el estudio y la aplicación de modelos de Deep Learning al campo de los videojuegos, en concreto, la interacción con estos en el área de juego.

Se toman los dos acercamientos con mejores resultados que se han observado, Deep Recurrent Q-Networks y agentes de tipo A3C (Asynchronous Advantage Autor-Critic), y se han adaptado sus modelos a Python para luego ser testados en un mismo entorno, rivalizando entre estos.

Índice general

Índice de figuras

Índice de tablas

1. CAPÍTULO

Introducción

Esta sección da el contexto del trabajo y pone en situación al lector. Hablará de hitos que que marcan la evolución de este campo, identificando los grupos de investigación las han llevado adelante, los avances que se han producido y las consecuencias que esto puede conllevar a futuro.

1.1. Inteligencia Artificial

El ser humano siempre ha tenido la obsesión de replicarse a si mismo, un ejemplo evidente son las artes, donde la figura humana es estudiada, replicada y modificada hasta la saciedad. La obsesión del ser humano consigo mismo le ha llevado a tratar de entenderse de todas las formas posibles. La inteligencia, la mente y el pensamiento han sido estudiados por largo tiempo. Cuando el poder computacional de los ordenadores se convirtió en una realidad surgió la idea de replicar la inteligencia humana en base a esta tecnología.

En el campo de la informática, la primera demostración de inteligencia artificial de la que se noticia ocurrió en 1951, en la Universidad de Manchester, donde se escribieron los primeros programas que fueron considerados como tal. Los programas fueron escritos por Dietrich Gunther Prinz y Christopher S. Strachey, las tareas que realizaron eran jugar al ajedrez y a las damas, respectivamente. El ordenador en el que se ejecutaron era un Ferranti Mark 1, el cual puede ser considerado el primer ordenador de propósito general disponible comercialmente.

Hoy en día, las técnicas de *Machine Learning* junto con la abundancia de datos disponibles, el desarrollo de la Inteligencia Artificial (IA) se encuentra en alza, siendo utilizada tanto en academia, investigación, industria, servicios y por particulares. DeepMind, OpenAI y NVIDIA son las compañias más reconocidas hoy en día por sus avances, publicaciones y aplicaciones de IA en diferentes campos, abarcando desde la medicina hasta los videojuegos, donde se sigue haciendo eco a los programas pioneros previamente mencionados.

1.1.1. Deep Leaning

Deep Learning (aprendizaje profundo) es una de las ramas de *Machine Learning*. Esta rama se basa en realizar aprendizaje (supervisado, semisupervisado o no supervisado) basándose en estructuras de redes neuronales. Estas estructuras suelen ser las caraterísticas de las redes neuronales profundas, recurrentes, generativas y convolucionales, lo cual ofrece una amplitud considerable con respecto a sus aplicaciones y con resultados que suelen rivalizar con expertos humanos.

En uno de los tipos de sistemas de *Deep Learning* mas populares, las redes neuronales convolucionales (CNN o ConvNet), la idea principal es la extracción jerarquizada de características de la señal input, generalmente imágenes, desde niveles muy abstractos hasta niveles de bajo nivel de detalle. En base a estas características se realiza la predicción de los valores de una variable output. Dependiendo del tipo de aprendizaje, el resultado de la red se evaluará con una recompensa o no, tratando de enseñar al sistema el comportamiento deseado o esperado.

1.2. Videojuegos

El acto de jugar es es universalmente reconocido y disfrutado, con múltiples formas de presentarse buscando evitar el aburrimiento. Fue precisamente con esa intención que en 1958 el físico William Higginbotham, utilizando un osciloscopio y un ordenador analógico, creó "*Tennis for Two*", simulando un partido de tenis.

Desde este punto la industria de los videojuegos fue evolucionando con una constante mejora tecnológica, pasando por las máquinas recreativas de los arcades, la aparición de las consolas domésticas y portátiles, hasta llegar al punto en el que estamos, donde los videojuegos son uno de los pasatiempos más populares del mundo y una industria

multimillonaria. Sin embargo, los elementos basicos siguen siendo los mismos: un medio de representación/visualización y otro para interactuar con dicha representación. Esto abre el mundo de los videojuegos a más posibilidades que simplemente ocio, permitiendo la expresión artística, narrativa y exploracional.

Es por esto que los videojuegos se pueden considerar como un campo atractivo para tratar de encontrar formas de interactuar previamente no consideradas, probar qué es capaz de realizar una inteligencia artificial y hasta que punto una arquitectura puede ser entrenada para tener buenos resultados cuando sea aplicada a la vida real.

1.3. Hitos recientes y posibles aplicaciones futuras

Nos es conocido el enfrentamiento de Garry Kasparov y Deep Blue en los años 1996 y 1997. Recientemente otro campeón humano fue vencido. En este caso el juego era Go, y la máquina era AlphaGo, desarrollada por Deepmind. AlphaGo se enfrentó en 2016 a uno de los mejores jugadores de Go, Lee Sedol, y obtuvo la victoria. El tablero en cuestión era de dimensión 19 en ambos lados, lo cual permite un total de $1{,}741x10^{172}$ posibles estados de juego. En 2017 Deepmind desveló AlphaZero, que superó con creces a AlphaGo.

A principios de 2019 una nueva frontera fue superada cuando un nuevo enfrentamiento tuvo lugar, esta vez en Starcraft 2[1], un juego en tiempo real de estrategia y gestión de recursos, contra un equipo profesional de jugadores. Debido a las caracteristicas del juego no es posible calcular el total de estados de juego posibles, haciendo que estrategias como el recorrido exhaustivo de estados sean inutilizables. Además el juego exige reaccionar en un tiempo tiempo constante. El resultado final del enfrentamiento fue 10-1 con AlphaStar[2] como ganadora. Cabe destacar que el equipo humano fue sorprendido debido a que AlphaStar utilizó técnicas que son desechadas o simplemente nunca se han concebido por los jugadores profesionales del juego.

Es debido a esto que una de las posibles aplicaciones futuras es el testeo de juegos en desarrollo, donde se utilizaría IA para testar un juego y probar posibilidades que a un *tester* humano posiblemente no se le ocurrirían o que serían el resultado de la casualidad. La IA permitiría obtener una recolección de errores y comportamientos inesperados más completa. Esto sería extrapolable fuera del ambiente de los videojuegos y aplicable al desarrollo general de software con el propósito de hacerlo más robusto.

[1] https://starcraft2.com/en-us/

[2] `https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/`

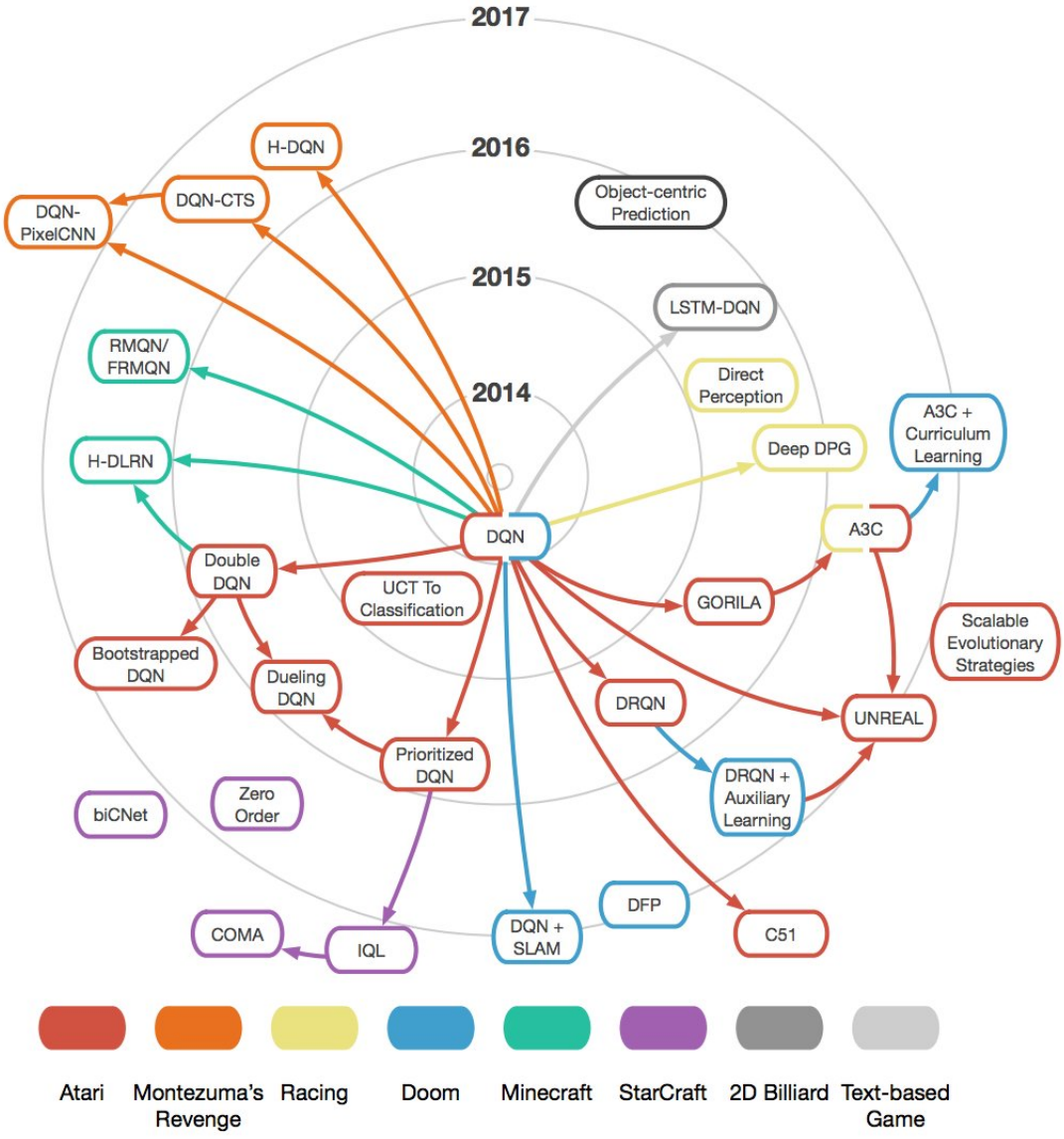


Figura 1.1: Evolución de los algoritmos con los años y en diferentes juegos

2. CAPÍTULO

Desarrollo del proyecto

2.1. Objetivos del proyecto

El objetivo inicial del proyecto es estudio y uso de modelos de *Deep Learning* en el campo de los videojuegos. Esto es, probar diferentes algoritmos actualmente en uso en este campo y tratar de mejorar los resultados obtenidos con ellos.

Para llegar a estos objetivos se va a adaptar un entorno en el que interactúen diferentes agentes, se buscará algoritmos y modelos de arquitecturas de aprendizaje que son utilizados actualmente, se adaptarán sus implementaciones al entorno elegido. Finalmente, se entrenarán para comprobar los resultados de dicho entrenamiento.

Este proyecto conlleva por tanto los siguientes objetivos de aprendizaje:

- Estudiar y comprender los fundamentos de algoritmos de *Deep Learning* junto con sus implementaciones.
- Profundizar el manejo de Tensorflow.
- Investigar el efecto de diferentes recompensas sobre las estrategias y resultados finales.
- Estudiar la puesta en marcha de un servidor central de control y comunicación para los experimentos computacionales con los juegos simulados.

2.2. Fases del proyecto

El desarrollo del proyecto está dividido en diferentes fases, las cuales responden a las diferentes necesidades del proyecto y por tanto conllevan tareas diferentes. Estas fases no necesitan ser completadas en su totalidad antes de pasar a la siguiente. Lo que ofrece la posibilidad de trabajar paralelamente obteniendo una mejor gestión del tiempo dedicado al proyecto. Las fases que permiten con mucha facilidad esto son las de entrenamiento y competición con los modelos de juego simulados. A continuación comento cada una de estas fases en mayor detalle.

2.2.1. Fase Preliminar

- Presentaciones iniciales con propuestas para el proyecto y estudio del estado del arte actual del campo que cubre el proyecto.
- Búsqueda del entorno *open source* de juego a utilizar en el proyecto.

2.2.2. Fase de investigación y diseño

- Búsqueda de proyectos similares recientes y los fundamentos sobre los que se establecen.
- Instalación de los paquetes de software necesarios.
- Realizar el diseño inicial del sistema intermedio que permite la interacción entre el entorno y los agentes.
- Planear las diferentes recompensas para los agentes en el entorno y los resultados que se buscan con estas.

2.2.3. Fase de implementación

- Implementación del sistema intermedio y de la generación de recompensas. Conforme avance el proyecto se considerará realizar modificaciones, si fueran necesarias o si aportan mejoras significativas al rendimiento obtenido.

- Adaptación y ajustes de los algoritmos de *Deep Learning* para ser utilizados en el entorno computacional del proyecto.
- Testeo de los algoritmos adaptados para realizar las correcciones necesarias.

2.2.4. Fase de entrenamiento

- Realizar el entrenamiento de las diferentes instancias de agentes y guardar los resultados.
- Recopilar la información obtenida en los entrenamientos.

2.2.5. Fase de competición

- Realizar los enfrentamientos entre las diferentes instancias de agentes entrenados.
- Recopilar la información resultante de los enfrentamientos.

2.2.6. Fase de recopilación

- Generar el documento de memoria del proyecto.
- Sacar conclusiones de los datos recogidos en las fases de entrenamiento y competición.

2.3. Planificación

El proyecto tomará lugar principalmente en la segunda parte del vigente año académico 2018-19, el motivo principal siendo la menor carga de asignaturas. Es solicitado en el 1 de Octubre de 2018, tras lo que se iniciará la actividad principal en Enero de 2019, para finalizar el proyecto y realizar la entrega de la memoria a la plataforma ADDI y publicar lo desarrollado en Zenodo el 23 de Junio de 2019. La defensa del proyecto se prevé para el 2 de Julio de 2019, marcando así la completa finalización del proyecto.

La planificación de las actividades se realiza de forma dinámica, tratando de ajustarse a las necesidades del proyecto pero dando respuesta a posibles desviaciones en la planificación y al resto de obligaciones que corresponde a la segunda parte del curso académico.

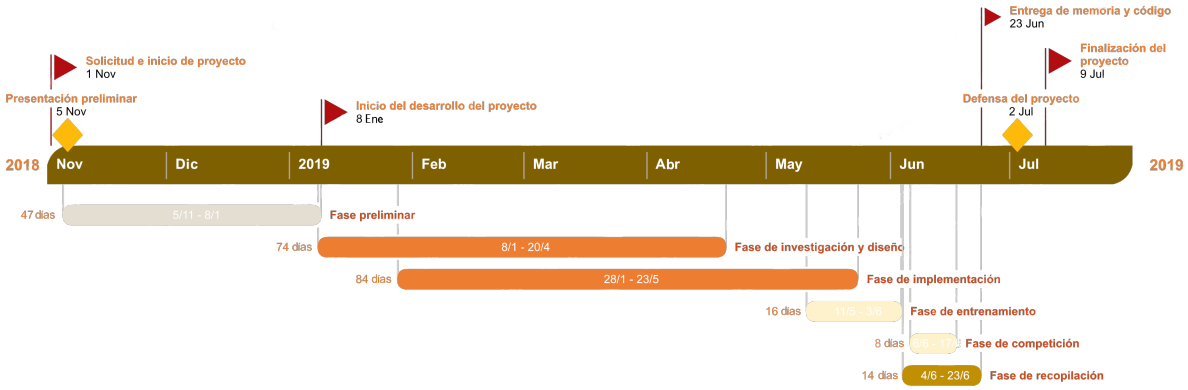


Figura 2.1: Calendario con la planificación de fases

2.4. Herramientas utilizadas

En este apartado se especificarán los diferentes recursos y herramientas utilizadas para la realización del proyecto.

2.4.1. Python3

Python es un lenguaje de programación interpretado de alto nivel de proposito general. Este lenguaje apareció en 1991 y actualmente dispone de una amplia variedad de librerías junto con una comunidad de usuarios y desarrolladores muy activa. Debido al tiempo transcurrido desde su primera aparición, hay actualmente dos versiones principales operativas: Python 3.7.3 y Python 2.7.16. Python2 lleva en funcionamiento desde el año 2001, lo cual le aporta una cantidad de librerías estables mayor que Python3. Sin embargo, en 2020 Python2 será descontinuado, dejando únicamente Python3 de cara a futuro. Es por esto que he elegido esta versión del lenguaje para desarrollar el proyecto, asegurando su continuidad en el futuro. Este futuro aparentemente será brillante debido al éxito que Python tiene en los campos de *Machine Learning* y data science por su legibilidad y facilidad de uso. Una de las razones por las cuales tiene tanto éxito en estos campos es debido a la facilidad de acceso a librerías como Tensorflow, Keras o Pytorch.

2.4.2. Tensorflow

Tensorflow es una plataforma *open source* desarrollada por Google Brain para *Machine Learning* lanzada en 2015, con un ecosistema comprensible y flexible de herramientas, librerías y recursos de la comunidad. Dichas librerías y herramientas están disponibles tanto en Python como en C++.

Tensorflow se basa en la creación de grafos computacionales, que funcionan como los nodos de una red. Cada nodo se interpreta como una operación sobre el flujo de datos que contiene funciones que pueden ir de operaciones simples (como sumas y restas) a operaciones con ecuaciones complejas. Es en base a esta estructura de grafo junto con las diferentes operaciones existentes que se pueden crear redes complejas para la implementación y ejecución de algoritmos y redes de *Machine Learning*.

2.4.3. Jupyter Notebook

Jupyter Notebook es una aplicación web *open source* que permite generar documentos con código activo, ecuaciones, visualizaciones y muchos más elementos. Es una de las herramientas mas populares en las comunidades de *data science* (ciencia de datos) por las posibilidades que ofrece y por la facilidad de compartir los *notebooks* que se construyen con ella. Es una herramienta con una gran flexibilidad debido a la cantidad de *kernels* existentes para diferentes lenguajes, su integración con Anaconda y las facilidades que ofrece para observar código en ejecución e interactuar con él. Por todas estas razones opté por utilizar esta herramienta para la implementación de las diferentes partes del proyecto.

Una vez se hayan obtenido resultados estables se considerará pasar el código a formatos más tradicionales, es decir, módulos en Python3.

2.4.4. Hardware y SO

Para el proyecto se hará uso de un ordenador principal donde se ejecutarán las diferentes partes del proyecto. Las capacidades del dispositivo son de un procesador i7-7700HQ con velocidades en el rango 2,8GHz-3.8GHz y 8 *cores*, 8GB de memoria DDR4 a 2400MHz y una tarjeta gráfica GTX1050 con 4GB de memoria GDDR5 a una frecuencia de 1354-1455 MHz. Con estas características en la actualidad se considera esta como una máquina de gama media-alta, lo cual se estima que dará un rendimiento suficiente para el pro-

yecto pero que puede requerir ajustes en los agentes para que no excedan la capacidad disponible.

El sistema operativo sobre el que se trabajará será Ubuntu 18.04.2, una distribución muy popular y gratuita de Linux basada en la arquitectura de Debian. Esta elección es debida a que el sistema operativo tienen una carga menor sobre el hardware, liberando espacio para su uso en el proyecto, además de ello también por la cantidad de recursos *open source* que están asociados a las distribuciones Linux.

2.5. Cambios de entorno de juego

En la propuesta original se consideraba únicamente como entorno el juego Pac-man, posiblemente en base a el curso de introducción a IA de la Universidad de Califonia, Berkeley[1].

Inicialmente consideré esta linea de trabajo pero llegué a la conclusión que el entorno no aportaba suficiente interés debido a la limitación de opciones de comportamiento de los agentes. Estudié diferentes entornos con diferentes focos de interés en cada uno en proyectos *open source*. Finalmente, terminé optando por un entorno ligero en términos de necesidades de cálculo para su ejecución, buscando no afectar al rendimiento de los agentes a ser entrenados. El entorno tambien debía tener facilidad de modificación para hacer las adaptaciones necesarias tanto para poder reaccionar apropiadamente a corto plazo y como para tener una estrategia de victoria a mas largo plazo.

2.6. Riesgos del Proyecto

Todo proyecto se enfrenta a situaciones de riesgo que amenazan su cumplimiento. En esta sección se elabora una lista de los riesgos identificados y medidas para tratar de limitar sus efectos.

- Estimación de tiempos errónea: Debido a la falta de experiencia en proyectos similares a este no se puede estimar con total fiabilidad el tiempo que se necesitará para las distintas fases de este. Para tratar de prevenir daños por esto se sobrestimará el tiempo necesario para las diferentes fases para dar suficiente tiempo para la realización.

[1] `http://ai.berkeley.edu/project_overview.html`

- Pérdida de información: Como en todo proyecto existe la posibilidad de la pérdida de información. Debido a esto se mantendrán copias actualizadas de los ficheros del proyecto. Además de eso antes de realizar cualquier actividad que requiera de muchos recursos y por tanto no sea repetible con facilidad, se considerará con tiempo cuales serán los datos que se quieran conseguir.

- Recursos limitados: Las tareas de entrenamiento y enfrentamiento de los agentes consumen una cantidad considerable de tiempo y recursos. Para el proyecto tan solo se dispone de un ordenador de gama media, lo cual tan solo permite trabajar con 2 agentes al mismo tiempo. Como medida para este riesgo tan solo se puede tratar de coordinar las diferentes actividades de forma que no se pierda tiempo pasando de una a otra.

3. CAPÍTULO

Entorno de juego y Algoritmos de aprendizaje

3.1. Entorno de juego

Para este proyecto es necesario tanto un entorno de juego sobre el que trabajar como los medios para interactuar con este. Para esto hay dos posible soluciones principales: la unificación de estas necesidades y la separación de estas. En la solución de unificación se le añaden al juego las funciones necesarias para crear outputs e inputs que sean necesarios para el o los agentes que interactúen con este. La unificación requiere de una mayor comprensión de el funcionamiento de el entorno que se está utilizando y, además, tener en cuenta la optimización necesaria para que no haya errores o retrasos que puedan generar un efecto en cadena que termine bloqueando el programa y, por tanto, interrumpiendo las tareas que se estén efectuando. Por contra, cabe destacar que cuando se realiza correctamente el resultado es una mayor centralización y menores pérdidas de tiempos por estar interactuando directamente con el entorno y no a través de un intermediario.

La otra opción mencionada opta por dejar el entorno de juego intacto en la medida de lo posible y crear un servicio intermedio que se encargue de todas las tareas de comunicación necesarias. Las ventajas de esta opción son la compartimentalización de las operaciones y, por tanto, una mayor libertad para realizar modificaciones y ajustes según sea necesario sin comprometer la integridad de el entorno. Además de ello la posibilidad de coordinar diferentes instancias de el entorno o diferentes entornos con la intención de agilizar y mejorar las ejecuciones. Además de esto también aporta la posibilidad de no estar necesariamente en ejecución en la misma máquina sobre la que están el o los agentes. La

desventaja principal de esta solución es el tiempo necesario para realizar las comunicaciones, que puede llegar a bajar la velocidad de los agentes perdiendo eventos que pueden aportar más información al agente, pero esto depende también de el entorno que se vaya a utilizar.

Habiendo considerado ambas opciones, los objetivos de el proyecto y el entorno de juego escogido se optó por la segunda opción. Por tanto el entorno de juego permanece intacto y se trabajará en crear un servidor intermedio que se encargará de la comunicación entre los agentes y el entorno.

Otro de los requisitos de el proyecto era que el entorno de juego fuese un proyecto open source. Esto dejó abierto un amplio rango de posibilidades debido a la accesibilidad actual de recursos para diseñar y programar videojuegos. Inicialmente la búsqueda se redujo a buscar réplicas funcionales de dos tipos de juego: Estrategia y lucha. En el primer tipo, usualmente se le proporciona al jugador una serie de recursos, un repertorio de acciones que puede realizar (que según avanza el juego se ramifican) y un tiempo casi ilimitado para conseguir un cierto objetivo, el cual usualmente suele ser llegar a ciertas condiciones de victoria contra otro jugador o arboles de decisión previamente programados a modo de jugadores artificiales. El segundo tipo de juegos suele presentar un escenario más limitado en el que dos o más jugadores se enfrentan en un periodo de tiempo corto (normalmente no superando los 3 minutos) hasta que solo queda uno de los jugadores sin ser derrotado.

El primer tipo ofrece una mayor complejidad y por tanto un mayor interés sobre los comportamientos que se generarían, pero al ser juegos sin limite de tiempo es difícil realizar los entrenamientos de los agentes en un tiempo razonable con los recursos computacionales disponibles. El segundo tipo tiene una complejidad reducida, pero por otra parte requiere que el agente sea reactivo casi de forma instantánea a los cambios de el entorno, disponiendo de un tiempo limitado para obtener resultados, empujándolo así a acciones más directas para llegar a las condiciones de victoria.

Tras una exploración de las opciones disponibles en proyectos open source y algo más de valoración de los dos tipos de juegos elegidos se optó por el segundo tipo usando una réplica de el popular juego Mortal Kombat adaptado a HTML5.

3.1.1. Open source Mortal Kombat

Entre diferentes opciones disponibles he elegido un proyecto compartido en GitHub que imita los primeros juegos de la serie *Mortal Kombat* implementado de HTML5 y JavaS-

Figura 3.1: Capturas de el entorno (izquierda) y de el juego original (derecha)

cript[1]. Esta implementación, aunque carezca de elementos como movimientos especiales (acciones activadas tras una secuencia concreta de inputs) y por tanto disminuye las posibilidades de observar estrategias aprendidas con mayor facilidad, la elegí porque no requiere recursos externos, lo que hace que el proyecto sea autocontenido. Además he verificado que no hay errores que afecten a la jugabilidad, la facilidad para realizar cambios y la poca carga computacional que supone su ejecución sobre el hardware disponible.

La primera instancia de un videojuego con las nociones fundamentales en las que se basa *Mortal Kombat* aparece en el mercado en 1976 en formato de máquina recreativa. Desarrollado por SEGA, *Heavyweight Champ* es el primero de muchos, siendo seguido en 1984 por *Karate Dō*, que popularizó el concepto de las artes marciales y los enfrentamientos 1 contra 1. Poco después, en 1987 surgió uno de los gigantes que aún perviven hoy en día, *Street Fighter*, dejando así a *Mortal Kombat* como una entrada mas tardía en 1992, pero no por ello menos popular, siendo facilmente distinguido de el resto por sus *fatalities* (movimientos finales que muestran una secuencia particularmente violenta con la que se derrota al enemigo).

Con el paso de el tiempo, se realizaron numerosas iteraciones que delinearon las bases fundamentales que definen este tipo de juegos, que son las siguientes:

- Dos o más jugadores comparten un mismo escenario de tamaño limitado, habitualmente bidimensional, pero con la posibilidad de ser tridimensional.
- Los jugadores disponen de una selección de personajes de la que pueden escoger uno. Cada personaje está definido por sus particularidades visuales, movimientos, animaciones, ataques y movimientos especiales.

[1] `https://github.com/mgechev/mk.js`

- Todos los jugadores tienen al menos un recurso principal, su salud, que habitualmente se visualiza en la parte superior de la pantalla.
- Dentro de el escenario, una vez seleccionados los personajes seleccionados, los jugadores utilizarán todo lo que vean necesario para llevar la salud de el resto de jugadores a 0, lo que implica su eliminación de el escenario.
- La condición de victoria consiste en ser el único jugador que queda activo en el escenario.
- La condición de derrota consiste en ser eliminado de el escenario.
- Cuando hay un tiempo limitado para el enfrentamiento, lo cual suele ser lo usual, el caso en que dos o más jugadores sigan activos en el escenario se considera como un resultado de empate. Normalmente, en caso de empate el jugador que tenga más salud al final de el enfrentamiento queda en una posición mas alta.

Aunque estas las bases que definen este tipo de juego también hay otros factores a tomar en cuenta. La relación de espacio y tiempo es uno de estos factores, debido a que si se dispone de más espacio se dispone de más tiempo para ejecutar una serie de acciones de forma exitosa y con mayor precisión. Ser capaz de leer la situación en un juego de lucha y planear como proceder en base a esta es crucial. Hay una gran cantidad de jugadores profesionales de este tipo de juegos que pasan su tiempo explorando todos los escenarios posibles y examinando las diferentes acciones que pueden utilizar para contrarrestar las acciones de el rival de la forma más eficiente.

La implementación escogida incluye todas las bases que se han mencionado. La única excepción es la falta de elección de personajes. Tener la posibilidad de múltiples personajes tan solo aumentaría los tiempos de entrenamiento y la cantidad de agentes a crear, lo cual tomaría demasiado tiempo y no aporta mucho a las conclusiones del proyecto. Las diferencias visuales con respecto al juego original que esta implementación libre replica no son dramáticas. De hecho, la eliminación de ciertos elementos que aparecen originalmente sobre la barra de vida resultarán beneficiosas a la hora de obtener datos de las imágenes.

Los dos jugadores (los cuales se referirán como jugador 1 y 2, siendo 1 representado por el personaje azul, *Sub-Zero*, y 2 representando el personaje rojo, *Kano*, de esta implementación. Habiendo dicho esto, falta mencionar cuales son todas las acciones que puede realizar que están enumeradas en la tabla 3.1. También hay que remarcar que el juego

Desplazamiento	Ataque	Defensa
Derecha	Puñetazo alto	Bloqueo
Izquierda	Puñetazo bajo	
Salto	Patada alta	
Cuclillas	Patada baja	

Tabla 3.1: Tabla de acciones disponibles

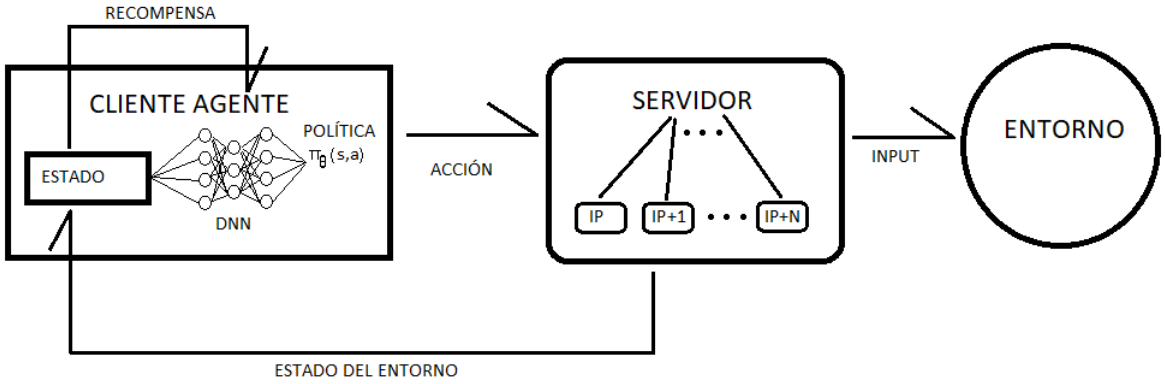


Figura 3.2: Estructura base

admite que se seleccionen dos acciones al mismo tiempo, siempre y cuando no sean la misma acción.

3.1.2. Servidor

Tal como se expone previamente, un sistema intermediario es necesario para realizar la interacción entre los agentes y el entorno. Este intermediario lo implementamos a través de un servidor TCP local ejecutado sobre el mismo hardware que el resto de elementos. La figura 3.7 muestra una simplificación de el esquema de el proyecto en el que está representado también el servidor. En el esquema se aprecia que el servidor se encargará de introducir los inputs que recibe por parte de el cliente al entorno y de proporcionar la información sobre el estado actual del entorno al agente. Las comunicaciones implementadas son limitadas y de corta duración, pero esto es parte de el diseño. Nuestro propósito es tratar de generar el menor tráfico posible y que las solicitudes se respondan lo antes posible. Todo esto es debido a que el servidor también actúa como un cuello de botella que puede llegar a limitar la velocidad de interacción de el agente, lo cual puede resultar nefasto teniendo en cuenta que el entorno solicita acciones y reacciones rápidas. Esto quedó al descubierto en la fase de ejecución, tras implementar cambios en el servidor y probarlos.

El servidor responde a los siguientes comandos:

- ***Set***: Este comando será solicitado por el cliente tan solo al inicio del juego, ya que la información que transmite no es necesario repetirla. La información a transmitir es el *padding* que hay en las posiciones de las barras de salud con respecto al marco de el entorno y la longitud en píxeles de estas.

- ***Start***: Este comando será solicitado por el cliente tan solo cuando se inicie un episodio. Este comando activa diferentes *flags* de el servidor, dependiendo que jugador lo envíe, lo cual se identifica por el número que se encuentra en el texto de el mensaje. Cuando haya recibido esta petición al menos una vez por ambos jugadores y los *flags* hayan sido activados, al recibir esta solicitud una vez mas se concederá permiso para iniciar el enfrentamiento, asegurando así que los jugadores tienen un inicio sincronizado.

- ***Get***: Este comando devuelve una captura de el estado de el juego, compuesta por una imagen RGB.

- **Acción**: Este comando recibe la acción que un jugador ha solicitado, la convierte al input de teclas correspondiente y lo introduce en el juego.

Al inicio del proyecto pensamos que una sola escucha de el servidor bastaría para llevar acabo todas las acciones de forma rápida, pero, como se ha mencionado antes, ese acercamiento no era correcto. Es por esto que se modificó y se crearon escuchas paralelas.

Antes de lanzar las escuchas son necesarias algunas acciones previas, como lanzar el juego junto con su controlador y obtener las medidas de el área de juego. Debido a la implementación de el juego, su ejecución es realizada en un navegador web (en nuestro caso se utiliza Mozilla Firefox) y para ello se utiliza la librería *Selenium*[2]. Esta librería esta pensada para la automatización de las tareas relacionadas con el navegador web y por tanto ofrece las herramientas necesarias para interactuar con él. Al iniciar el servidor lanzamos tantos juegos como se necesiten, se ajusta el tamaño y posición de los navegadores para tener imágenes claras, finalmente guardamos los controladores. Una vez está todo cargado se saca una captura de pantalla de los diferentes navegadores para obtener la medida exacta de donde se encuentra la representación visual de cada juego. Una vez se termina la toma de medidas, las diferentes escuchas son lanzadas de forma paralela, cada

[2] `https://selenium-python.readthedocs.io/`

una con un puerto diferente que identifica la escucha. Las escuchas tan solo son compartidas por los clientes cuando solicitan un inicio de partida, la sincronización se realiza en la escucha que corresponda al jugador 1.

Para facilitar las tareas de localización automática de los jugadores durante el juego se modifica el escenario por un fondo plano con el color RGB (188,120,83), color que no se encuentra presente en ningún otro elemento de el juego. Además de esto para evitar que se activen acciones indeseadas debido a combinaciones de teclas y por tanto pudiendo interrumpir el funcionamiento correcto de el juego, tras varias pruebas se decide un mapeado de teclas para los *inputs* que no causa conflictos.

Para introducir comandos al juego los clientes mandan un código numérico que representa la acción o combinación de acciones elegida, lo cual es luego traducido a las teclas correspondientes dependiendo de el jugador y le son suministradas al controlador cuya escucha está asignada.

La captura de estados del juego es realizada con una captura de pantalla en una zona específica, delimitada por las medidas tomadas al iniciar el servidor. Esto implica que la cantidad de juegos que se pueden lanzar está limitada por el espacio de pantalla de el que se dispone, pero, teniendo en cuenta las necesidades de el proyecto, esto no es un problema.

3.2. Algoritmos

Este proyecto no llegaría a ninguna parte sin la parte de Inteligencia Artificial. Para ello durante la fase de investigación se buscaron noticias, publicaciones, tutoriales e implementaciones aplicables al área de estudio del proyecto, teniendo en cuenta que el proyecto tiene su enfoque hacia *Deep Learning*. Finalmente elegí dos algoritmos que se ajustan a la filosofía de *Deep Learning* y que son muy recientes: *Deep Q-Learning* y *Asynchronous Advantage Actor-Critic*.

Todos los algoritmos serán equipados con un programa cliente ajustado a su posición particular (dependerá del jugador al que representen) con el que se podrá acceder a las escuchas del servidor, pudiendo así solicitar y mandar la información que sea pertinente.

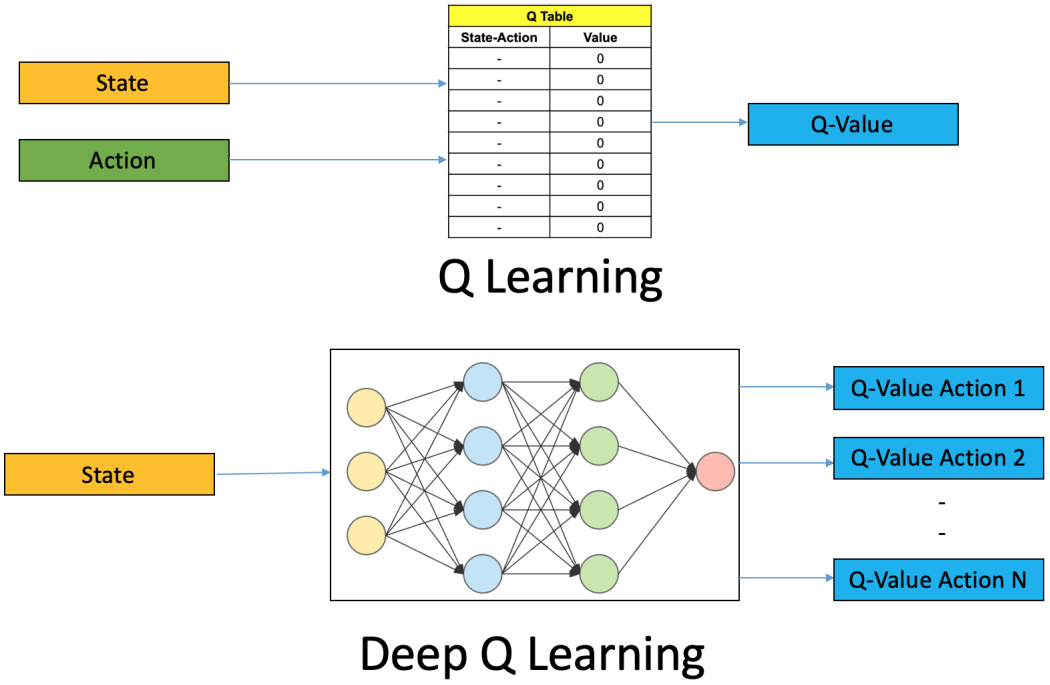


Figura 3.3: Representación del funcionamiento de *Q-Learning* y *Deep Q-Learning*

3.2.1. DQN

Q-Learning es un algoritmo de *Reinforcement Learning* utilizado en *Machine Leaning*. Q-Learning no requiere aprender el modelo del entorno, su objetivo es aprender una política que informe al agente pertinente de que acción debe tomar dependiendo de las circunstancias. Las políticas resultantes para cualquier proceso finito de toma de decisiones de *Markov* reciben una observación del entorno y escogen las acciones que dan una mayor recompensa a largo plazo. Dada una política parcialmente aleatoria y tiempo de exploración ilimitado, el algoritmo *Q-Learning* puede identificar una política óptima de selección de acciones.

Un proceso de toma de decisiones *Markoviano* es un proceso estocástico de control. Es una herramienta matemática para el modelado de toma de decisiones donde el resultado es parcialmente aleatorio. Estos procesos son útiles para estudiar problemas de optimización mediante programación dinámica y son conocidos desde los años 50.

En su implementación más simple, *Q-Learning* es una tabla de valores por cada estado y acción posible en el entorno. Dentro de cada celda de la tabla se aprende un valor, también conocido como *Q-value*, que representa el beneficio que proporciona ejecutar cierta acción dado un cierto estado. Al explorar este entorno se actualizan las celdas mediante

la ecuación de optimización de Bellman (ecuación 3.1), la cual dice que la recompensa a largo plazo por una cierta acción es igual a la recompensa inmediata por la acción combinado con la recompensa esperada para la mejor acción futura posible dado el siguiente estado. Concretamente, s siendo el estado y a la acción tomada para la recompensa inmediata $r(s,a)$, s' representando el siguiente estado y γ el factor de descuento que controla la contribución de las recompensas futuras. El desarrollo de esta ecuación pasa por 3.2 y finalmente en 3.3 por practicalidad a la hora de implementar.

$$Q(s,a) = r(s,a) + \gamma max_a Q(s',a) \tag{3.1}$$

$$Q(s,a) \rightarrow \gamma Q(s',a) + \gamma^2 Q(s'',a)...\gamma^n Q(s''^{...n},a) \tag{3.2}$$

$$Q(S_t,A_t) \leftarrow Q(S_t,A_t) + \alpha[R_t + 1 + \gamma max_a Q(S_t+1,a) - Q(S_t,A_t)] \tag{3.3}$$

Deep Q-Learning es el siguiente paso, donde se toman las bases de *Q-Learning* y se aplican a redes neuronales profundas, generando *Deep Q-Learning Networks* (DQN). Esta aplicación es particularmente interesante cuando la tabla de estados y acciones empieza a tener grandes dimensiones, que llega a ocupar espacios de memoria descomunales y el tiempo necesario para explorar todo los estados sería irrealizable o simplemente no óptimo.

Con las redes neuronales se aproxima la función *Q-value*, tomando el estado como *input* y devolviendo como el *output* todos los posibles resultados de la función *Q-value*. La diferencia entre *Q-Learning* y *Deep Q-Learning* se representa visualmente en la figura 3.4.

Se ha mostrado ya los buenos resultados que se pueden obtener con estas aplicaciones, uno de los ejemplos más conocidos son las redes DQN aplicadas a juegos de la consola ATARI [NAT, 2015][3], pero también se han cosechado éxitos en otros entornos[Lample and Chaplot, 2016][4]. Existen formas de tratar de mejorar más esta aplicación con variantes como *Double DQN* o *Dueling DQN* y añadidos como la repetición de experiencia en redes neuronales (RNN) o añadir redes neuronales segundarias para generar un *Q-target* con la que se calcula la función de pérdida para cada acción durante el entrenamiento. Todas estas son mejoras

[3] `https://www.nature.com/articles/nature14236`
[4] `https://arxiv.org/abs/1609.05521v2`

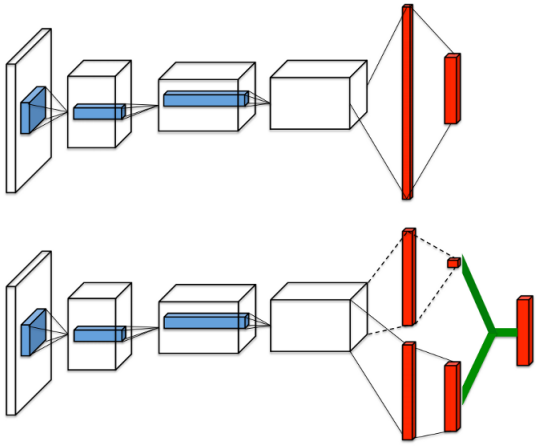


Figura 3.4: Diferencia visual de una DQN (superior) y una *Dueling DQN* (inferior)

que se tendrán en el agente basado en *Q-Learning* de este proyecto.

Double DQN

La intuición principal detrás de el concepto de una *Double DQN* es que una DQN normal tiende a sobrestimar los *Q-values* de las potenciales acciones a tomar dado un cierto estado. Esto no sería de gran importancia si se sobrestimase de forma uniforme, pero este no suele ser el caso. Podemos imaginar fácilmente que si ciertas acciones subóptimas reciben una sobre-estimación considerable (anteponiéndolas a las acciones óptimas), el agente tendría una mayor dificultad a la hora de aprender una política óptima.

Para corregir esto se propuso un simple truco en [van Hasselt et al., 2015]: en vez de obtener el valor máximo de los valores computados, cuando se computa el valor *target-Q* para los pasos de entrenamiento, se utiliza la red primaria para elegir una acción y la red de objetivo genera el valor objetivo del *Q-value* para dicha acción. Separando la elección de acción de la generación del valor objetivo *Q-value* se reduce de forma sustancial la sobrestimación, entrenando de forma mas rápida y segura. La nueva ecuación de

$$Q - Target = r + Q(s', argmax(Q(s', a, \theta), \theta')) \tag{3.4}$$

Dueling DQN

Para poder explicar el razonamiento detrás de los cambios de arquitectura realizados en *Dueling DQN*, es necesario comprender ciertos términos y conceptos basicos de *Reinfor-*

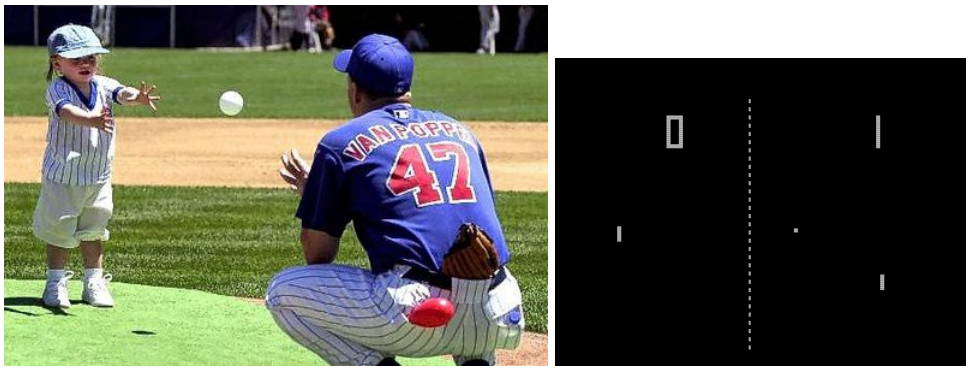


Figura 3.5: Ejemplos de falta de información en imágenes

cement Learning. Los *Q-values* que previamente se han mencionado corresponden a cuan beneficioso es elegir cierta acción cuando el agente está en cierto estado. Esto se denota como $Q(s,a)$ 3.5. Esta acción con ese estado específico puede ser descompuesto en dos funciones más fundamentales de valor. La primera es la función de valor $V(s)$, la cual describe lo beneficioso que es estar en cierto estado. La segunda es la función de ventaja $A(a)$, que describe que acciones obtienen un mejor resultado comparadas con otras.

$$Q(s,a) = V(s) + A(a) \tag{3.5}$$

El objetivo de una *Dueling DQN* es tener una red que de forma separada computa las funciones de valor y ventaja para combinarlas de nuevo en una sola *Q-function* en la última capa de la red neuronal [Ziyu Wang, 2015][5] A primera vista esto puede parecer inútil. La clave de este planteamiento es el beneficio de que la red puede que no necesite de esos valores en todo momento. Se pueden conseguir estimaciones más robustas del valor del estado separándolo de la necesidad de estar unido a acciones específicas.

DRQN

Uno de los mayores problemas a los que se enfrenta una DQN es el conocimiento parcial del entorno[Hausknecht and Stone, 2015][6]. Previamente se ha mencionado que una DQN trabaja con un proceso de decisiones de *Markov* como entorno, lo cual es un buen formalismo, pero en el campo que atañe a este proyecto los desarollos no logran implementar ese estándar. En el entorno hay información que falta para poder tomar decisiones óptimas. Cuando se observa una foto la falta de movimiento nos esconde información.

[5] `https://arxiv.org/abs/1511.06581`.
[6] `https://arxiv.org/abs/1507.06527`

Tomemos como ejemplo la figura 3.5. ¿En que dirección va la bola, a la niña o el jugador? ¿A que velocidad va la bola? Solo con esta imagen no se dispone de suficiente información para hacer una evaluación completa del estado. La imagen está bloqueada temporalmente. Un ejemplo más directamente aplicable al proyecto sería el juego Pong, también representado en la figura 3.5. Tan solo con una imagen no se puede saber la velocidad, dirección ni ángulo en el que está moviéndose la bola, haciendo que sea mucho más difícil predecir la siguiente acción a tomar.

La solución que se propone para este problema es darle al agente la capacidad de hacer integraciones temporales de las observaciones[Hausknecht and Stone, 2015][7]. La intuición detrás de esto es la siguiente: Si la información de un solo momento no es suficiente para tomar una buena decisión, entonces adquirir suficiente información temporal probablemente sea una mejor opción. Revisando los ejemplos de la figura 3.5, en ambos ejemplos las preguntas que se plantean serían mayormente resueltas si se dispone de una secuencia de dos imágenes: en ambos casos la mayoría de las preguntas planteadas se resolverían. Una secuencia más larga puede incluso resolverlas todas.

Dentro del contexto de *Reinforcement Learning* hay diferentes formas de conseguir la integración temporal. Una de las soluciones es propuesta en [NAT, 2015][8], donde en vez de darle a la red neuronal una sola imagen de *input* se crea un *buffer* externo en el que se mantienen las 4 previas imágenes que se han utilizado. Aunque esto sea funcional no es biológicamente correcto (la retina no recibe imágenes de 4 en 4), el *buffer* de experiencia usado consume mucha más memoria por los grupos de 4 imágenes, que ahora se le suministran a la red, y se requieren *buffers* de mayor tamaño para poder recordar información importante que se origina en un pasado más distante. Para lidiar con todo esto de forma más robusta se propone el uso de redes neuronales recurrentes (RNN).

Una RNN tiene la particularidad de que las conexiones entre sus nodos forman un grafo dirigido en base a una secuencia temporal. Esto le permite mostrar un comportamiento dinámico temporal y aprender dependencias temporales. A diferencia de otras redes una RNN puede utilizar su estado interno (memoria) para procesar secuencias de *inputs*. En la arquitectura de la DQN se puede integrar el bloque recurrente característico de una RNN. Haciendo esto el agente resultante puede recibir imágenes únicas del entorno y la red cambiará el *output* dependiendo de el patrón temporal de observaciones que recibe. Esto se consigue manteniendo un estado oculto que computa con cada ciclo. El bloque recurrente puede suministrarse con el estado oculto, actuando como una mejora que le in-

[7] `https://arxiv.org/abs/1507.06527`
[8] `https://www.nature.com/articles/nature14236`

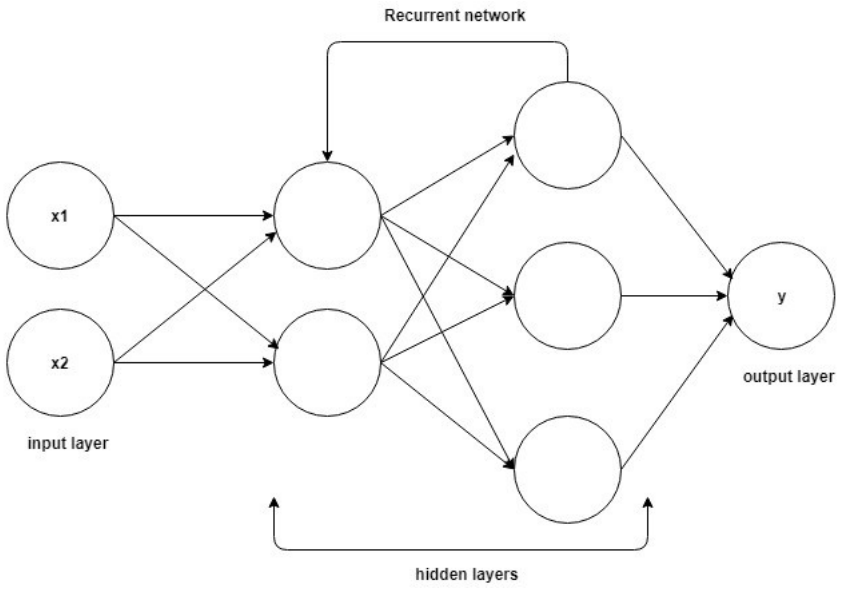


Figura 3.6: Ejemplo de red RNN

dica a la red de lo que ha sucedido con anterioridad. Los agentes de este tipo son referidos como *Deep Recurrent Q-Networks* (DRQN).

3.2.2. A3C

El algoritmo *Asynchronous Advantage Actor-Critic* (A3C) fue ideado y publicado por DeepMind. Según algunas opiniones A3C hace obsoletas las redes DQN. Esto es debido a que es considerado como más veloz, simple, robusto y capaz de conseguir mejores resultados en una colección de tareas estándar correspondientes a *Deep Reinforcement Learning* [Volodymyr Mnih and Kavukcuoglu, 2016][9]. OpenAI, otra iniciativa en una dirección diferente, se basó en este mismo algoritmo para crear su agente inicial universal[10] para trabajar con su universo de entornos.

El algoritmo se basa en los diferentes bloques que conforman su nombre.

- *Asynchronous*: A diferencia del algoritmo DQN previamente presentado y otros modelos donde un solo agente representa la totalidad de la red neuronal que interactua con un solo entorno, el algoritmo A3C utiliza múltiples encarnaciones de agentes y entornos para aumentar la eficiencia de el aprendizaje. El algoritmo dispone de una red global y múltiples agentes trabajadores, cada uno con sus propio

[9] `https://arxiv.org/abs/1602.01783v2`
[10] `https://github.com/openai/universe-starter-agent`

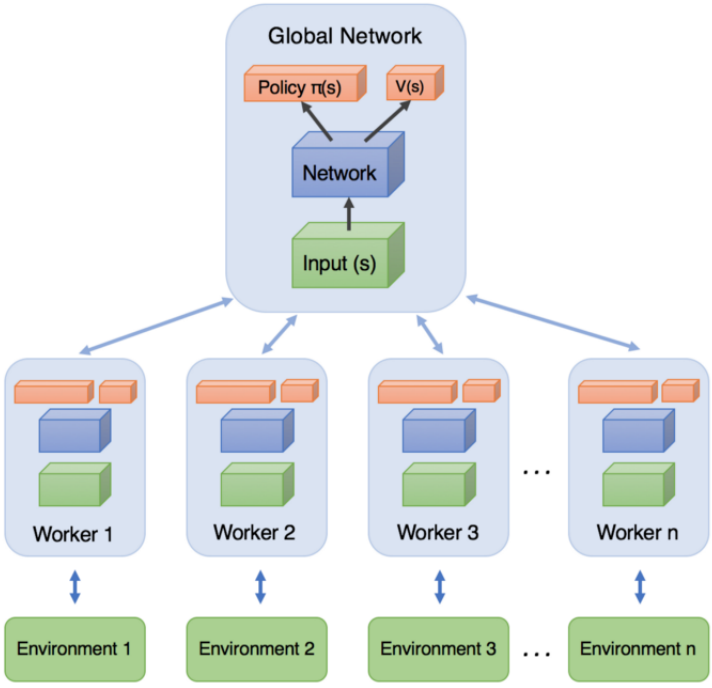


Figura 3.7: Representación del funcionamiento de *Asynchronous Advantage Actor-Critic*

conjunto de parámetros de la red. Cada uno de los trabajadores interactua con su propia copia de el entorno al mismo tiempo que el resto. El motivo por el que esto funciona mejor que un solo agente, aparte de acelerar el proceso, es debido a que la experiencia de cada agente es independiente a la de el resto. De esta forma el total de experiencia disponible para entrenar se vuelve más diversa.

- *Actor-Critic*: El acercamiento *Actor-Critic* combina los beneficios de los acercamientos de *Q-learning* y gradiantes de políticas con respecto al valor de las iteraciones. La red estima tanto la función de valor $V(s)$ y la política $\pi(s)$, la cual representa un grupo de *outputs* de probabilidad de acción a ser tomada. Estas funciones se separan en diferentes capas completamente conectadas en la cima de la red. De forma crítica el agente utilizará la estimación de valor (la parte crítica) para actualizar la política (la parte de actor) de forma mas inteligente que los tradicionales métodos de gradientes de políticas.

- *Advantage*: Esto marca el uso de estimaciones de ventaja en vez de recompensas con cierto descuento, como se aplicaba en el modelo DQN. Esto permite al agente determina no solo cuan beneficiosas son sus acciones, sino que también ver cuan

mejor paradas han terminado comparado con lo que se esperaba inicialmente. De esta forma el algoritmo se puede enfocar en buscar donde las predicciones de la red fallan. Recordemos que la función de ventaja se encuentra en la ecuación 3.5.

La parte de asincronicidad es a veces considerada una ventaja y en otras un estorbo, lo cual depende de el entorno. Existe una variante de el algoritmo (A2C) que sincroniza a todos los trabajadores, de forma que hasta que todos no finalizan sus respectivos episodios no se pasa al siguiente.

Este algoritmo además también es caracterizado por su estructura de *workflow* en sus trabajadores, la cual toma este orden y es repitida de forma cíclica (su pseudocódigo está representado en la figura 3.8):

1. Resetear el trabajador en base a la red global.
2. Interacción del trabajador con el entorno hasta llegar a un estado terminal.
3. El trabajador calcula las funciones de valor y de pérdida de política.
4. El trabajador obtiene nuevas gradientes en base a las funciones de pérdida.
5. El trabajador actualiza la red global con los gradientes obtenidos.

3.2.3. Recompensas

Por último, las recompensas que recibirán los diferentes agentes y que por tanto los caracterizará. Tomando inspiración de diferentes patrones comúnmente encontrados en jugadores hemos intentado reflejarlos en las diferentes recompensas:

- *Vanilla*: Inspirado en un comportamiento balanceado con únicamente la intención de ganar, sin estrategia premeditada, pero por ello con mayor libertad de exploración de estrategias.
- *Aggro*: Inspirado en comportamientos agresivos que buscan la ofensiva y ganar con rapidez. Para tratar de reforzar este comportamiento a las operaciones de la recompensa textitVanilla (salvo la derrota) se las multiplicará con la la división de el cardinal de acciones totales realizables en un enfrentamiento entre el cardinal de acciones realizadas.

Algorithm S3 Asynchronous advantage actor-critic - pseudocode for each actor-learner thread.

// Assume global shared parameter vectors θ *and* θ_v *and global shared counter* $T = 0$
// Assume thread-specific parameter vectors θ' *and* θ'_v
Initialize thread step counter $t \leftarrow 1$
repeat
  Reset gradients: $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$.
  Synchronize thread-specific parameters $\theta' = \theta$ and $\theta'_v = \theta_v$
  $t_{start} = t$
  Get state s_t
  repeat
    Perform a_t according to policy $\pi(a_t|s_t;\theta')$
    Receive reward r_t and new state s_{t+1}
    $t \leftarrow t + 1$
    $T \leftarrow T + 1$
  until terminal s_t **or** $t - t_{start} == t_{max}$
  $R = \begin{cases} 0 & \text{for terminal } s_t \\ V(s_t, \theta'_v) & \text{for non-terminal } s_t \text{// Bootstrap from last state} \end{cases}$
  for $i \in \{t-1, \ldots, t_{start}\}$ **do**
    $R \leftarrow r_i + \gamma R$
    Accumulate gradients wrt θ': $d\theta \leftarrow d\theta + \nabla_{\theta'} \log \pi(a_i|s_i;\theta')(R - V(s_i;\theta'_v))$
    Accumulate gradients wrt θ'_v: $d\theta_v \leftarrow d\theta_v + \partial \left(R - V(s_i;\theta'_v)\right)^2 / \partial \theta'_v$
  end for
  Perform asynchronous update of θ using $d\theta$ and of θ_v using $d\theta_v$.
until $T > T_{max}$

Figura 3.8: Pseudoalgorítmo de los trabajadores en *Asynchronous Advantage Actor-Critic*

- *Kite*: Inspirado en comportamientos que maximizan el movimiento. El nombre hace referencia a una estrategia común en jugadores que mantienen la distancia durante mucho tiempo para acercarse un momento y realizar daño antes de volver a la seguridad de la distancia. Para tratar de reforzar este comportamiento a las operaciones de la recompensa textitVanilla (salvo la derrota) se las multiplicará con la función $M(A)$. Esta función toma el *buffer* de acciones realizadas A y contabiliza que porcentaje de estas tienen como acción principal una que conlleva movimiento.

- *Tank*: Inspirado en comportamientos defensivos que buscan bloquear todo el daño que se le brinde al jugador y romper ataques encadenados uno tras otro por parte del jugador enemigo. Para tratar de reforzar este comportamiento a las operaciones de la recompensa textitVanilla (salvo la derrota) se las multiplicará con la función $B(A)$. Esta función toma el *buffer* de acciones realizadas A y contabiliza que porcentaje de estas tienen como acción principal una que conlleva un bloqueo.

	Victoria	*Derrota*	*Empate*
Vanilla	1	-1	$\frac{HP_{jugador} - HP_{enemigo}}{200}$
Aggro	$\frac{T_{max}}{T_{enfrentamiento}}$	-1	$\frac{HP_{jugador} - HP_{enemigo}}{200}$
Kite	$M(A)$	-1	$\frac{HP_{jugador} - HP_{enemigo}}{200} * M(A)$
Tank	$B(A)$	-1	$\frac{HP_{jugador} - HP_{enemigo}}{200} * B(A)$

Además de esto para todos los agentes como una recompensa intermediaria recibirán $\frac{HP_{jugador} - HP_{enemigo}}{200}/100$ con cada acción que tomen. El sumatorio de estas recompensas intermediarias no rivalizan con las recompensas finales, pero se consideran un incentivo necesario para que los agentes mejoren con mayor facilidad.

3.3. Resumen de herramientas y algoritmos utilizados

Las herramientas a utilizar serán una réplica *open source* del juego Mortal Kombat con un servidor intermedio que se encarga de las tareas de interacción entre el juego y los agentes. Por otra parte, los agentes que se han escogido para el proyecto son adaptaciones de los algoritmos *Deep Q-Learning* y *Asynchronous Advantage Actor-Critic*. Los agentes de estos pasarán a ser llamados DRQN (este por los añadidos al algoritmo DQN) Y A3C. Las recompensas se han preparado de forma manual para tratar de influir a los diferentes agentes a tomar comportamientos diferentes.

4. CAPÍTULO

Aplicación y resultados

4.1. Entrenamiento de los agentes

Una vez implementados y testeados para corregir potenciales errores, los agentes proceden a la fase de entrenamiento, durate la cual se realizará el aprendizaje sobre el entorno de juego en el que se encuentran. Para ello se definen los parámetros de los episodios de entrenamiento y de los límites y exigencias que se le aplicarán a los agentes que estén tomando parte:

- Los entrenamientos se realizarán con una pareja de agentes (salvo en el caso de los agentes A3C de propósito global) que serán emparejados para tratar de tomar ventaja de sus recompensas, cada uno tomando control de uno de los dos jugadores y siendo caracterizados por estos.
- Una sesión de entrenamiento se compondrá de 700 episodios.
- Cada episodio tendrá una duración máxima de 1 minuto, siendo posible finalizar antes si se llega a una situación de victoria.
- Se establece un máximo de 2.4 acciones por segundo, lo cual nos da 144 posibles acciones por episodio como máximo. He decidido esta cifra en base a la capacidad de cálculo del servidor para evitar bloqueos de ejecución.
- Para los agentes DRQN los primeros pasos de 100 episodios de duración completa serán compuestos exclusivamente por acciones aleatorias, los siguientes pasos de

Jugador 1	Jugador 2
DRQN Vanilla	DRQN Vanilla
DRQN Aggro	DRQN Kite
DRQN Kite	DRQN Aggro
A3C Vanilla	A3C Vanilla
A3C Aggro	A3C Tank
A3C Tank	A3C Kite
A3C Kite	A3C Kite

Tabla 4.1: Tabla de parejas de entrenamientos

200 episodios de duración completa reducirán la aleatoriedad hasta que se llegar a un 10 % de aleatoriedad. En caso de llegar a múltiples episodios con esa cifra en la que se llega a una situación de empate se aumentará la aleatoriedad a un 30 % con tal de tratar de añadir nuevas acciones a la memoria y así tratar de encontrar comportamientos con mejores resultados.

- Debido a las características de el algoritmo A3C también entrenaremos 4 agentes (uno por tipo de recompensa) de forma que entre todos los trabajadores tomen el control de todos los jugadores en varios entornos, con la intención de que traten de aprender una perspectiva más global del entorno y así poder encarnar cualquiera de los jugadores en enfrentamientos futuros o tener una mejor comprensión de el propio entorno.

Debido a problemas que se encontraron con el servidor durante los entrenamientos de las redes DRQN, no se ha entrenado uno de los pares de agentes previstos con todas las diferentes recompensas, dejando un total de 3 tipos de agentes, cada uno con una versión por cada jugador.

Considerando todo esto, el funcionamiento del servidor y asumiendo eficiencia perfecta entre los sistemas se estima que necesitaremos en el caso peor un tiempo total de 12 horas 40 minutos por cada sesión de entrenamiento. La totalidad de agentes a ser entrenados es de 18 (de los cuales 4 son agentes A3C globales), lo cual constituye un total de 11 sesiones de entrenamiento. Esto implica que necesitamos un total de 5 días 19 horas y 40 minutos para realizar todos los entrenamientos.

Para tratar de observar diferencias entre los algoritmos y las recompensas, diferentes parejas de agentes son utilizadas en los entrenamientos de los agentes no globales, listadas en la tabla 4.1.

4.1.1. Observaciones

Durante el entrenamiento original de los agentes DRQN las recompensas otorgadas fueron alteradas con el propósito de mejorar los comportamientos obtenidos. Por eso realizamos una segunda instancia de entrenamiento de los agentes DRQN, cuyos resultados son la versión final que se llevan a la fase de competición.

Las figuras 4.1, 4.2 y 4.3 muestran de forma gráfica todas las recompensas obtenidas a lo largo de las sesiones de entrenamiento. Aquí se pone en evidencia que los agentes A3C reciben una mayor cantidad de recompensas debido a sus ejecuciones paralelas, lo que les da más información y ,si los entrenamientos son fructíferos, resultados más interesantes. Sin embargo, en los agentes DRQN es más fácil ver patrones, los picos de valores que se obtienen y las correlaciones de las victorias y derrotas entre los agentes, lo cual resulta mas indicativo a simple vista. Por otra parte, en las recompensas de los agentes no aparecen patrones fácilmente identificables, lo cual dificulta la tarea de juzgar a simple vista si el entrenamiento ha sido óptimo o si podría haber sido mejor.

En la figura 4.4 se muestran los tiempos de los entrenamientos de los agentes A3C, que superan el tiempo máximo calculado por margenes que van desde los 20 minutos a los 40 minutos. Las secciones de estos entrenamientos que fueron examinadas en detalle mostraron que una buena porción de ellas finalizaba antes de llegar al límite de tiempo, esto sugiere que los tiempos de almacenamiento de datos, operaciones exteriores a los agentes y del propio servidor tienen que ser mejor optimizadas para no tener estas pérdidas de tiempo en el futuro, aunque tan solo sea un añadido de un 5 % al tiempo máximo calculado. Aparte de eso por el momento las diferencias no son suficientemente significativas como para extraer conclusiones de los entrenamientos. Es necesario realizar más entrenamientos y llevar un mejor control de la información que se puede obtener de estos para inferir alguna conclusión interesante.

4.2. Competición

Tomando inspiración en la estrategia de desarrollo de los agentes que se siguió en AlphaStar, tras finalizar el entrenamiento de todos los agentes se procederá a realizar una competición entre ellos para observar cuales son los más efectivos, no solo en términos de victorias, pero también en margenes de salud entre los jugadores y el tiempo que toman los enfrentamientos.

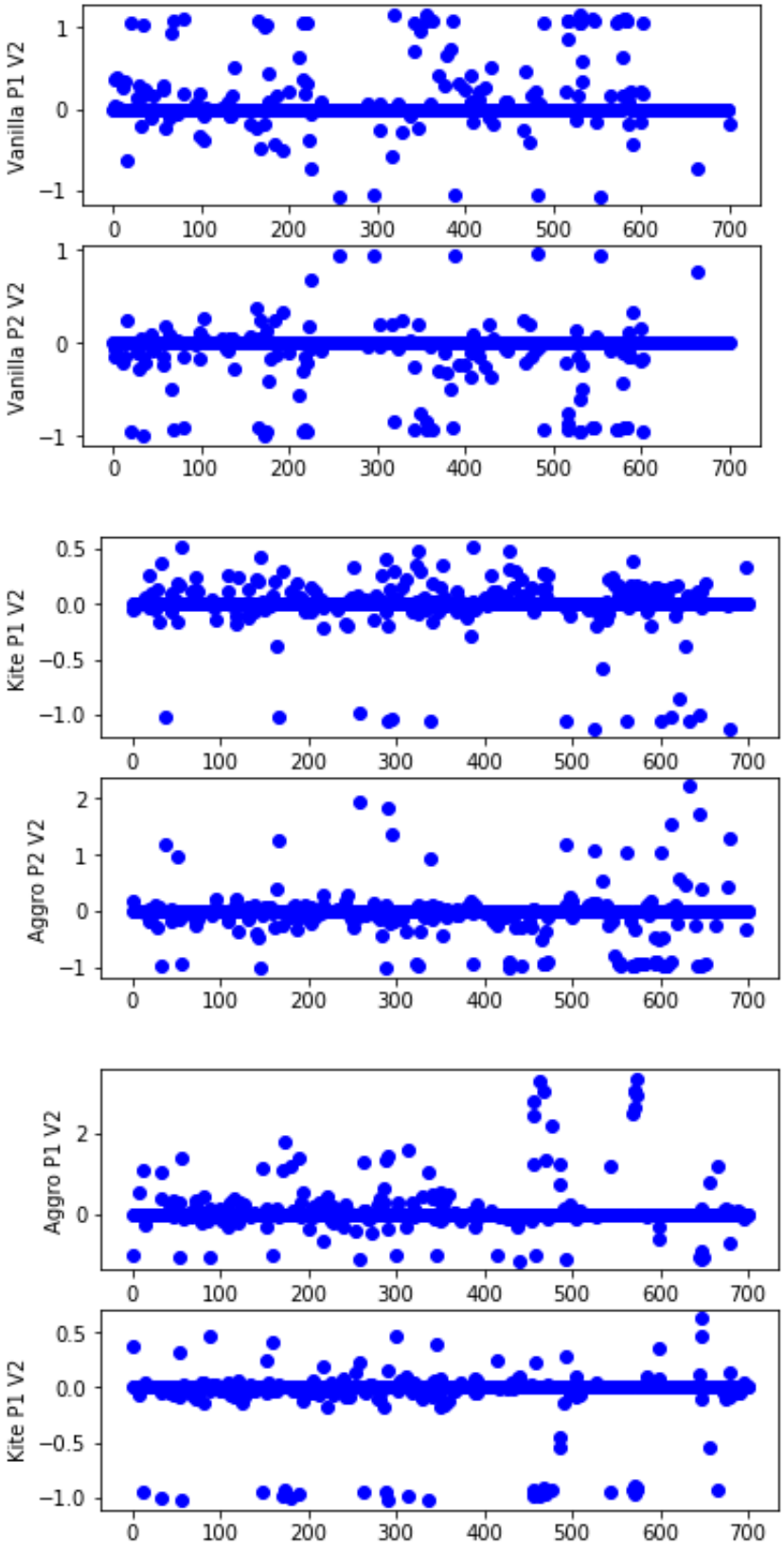


Figura 4.1: Recompensas de los agentes DRQN

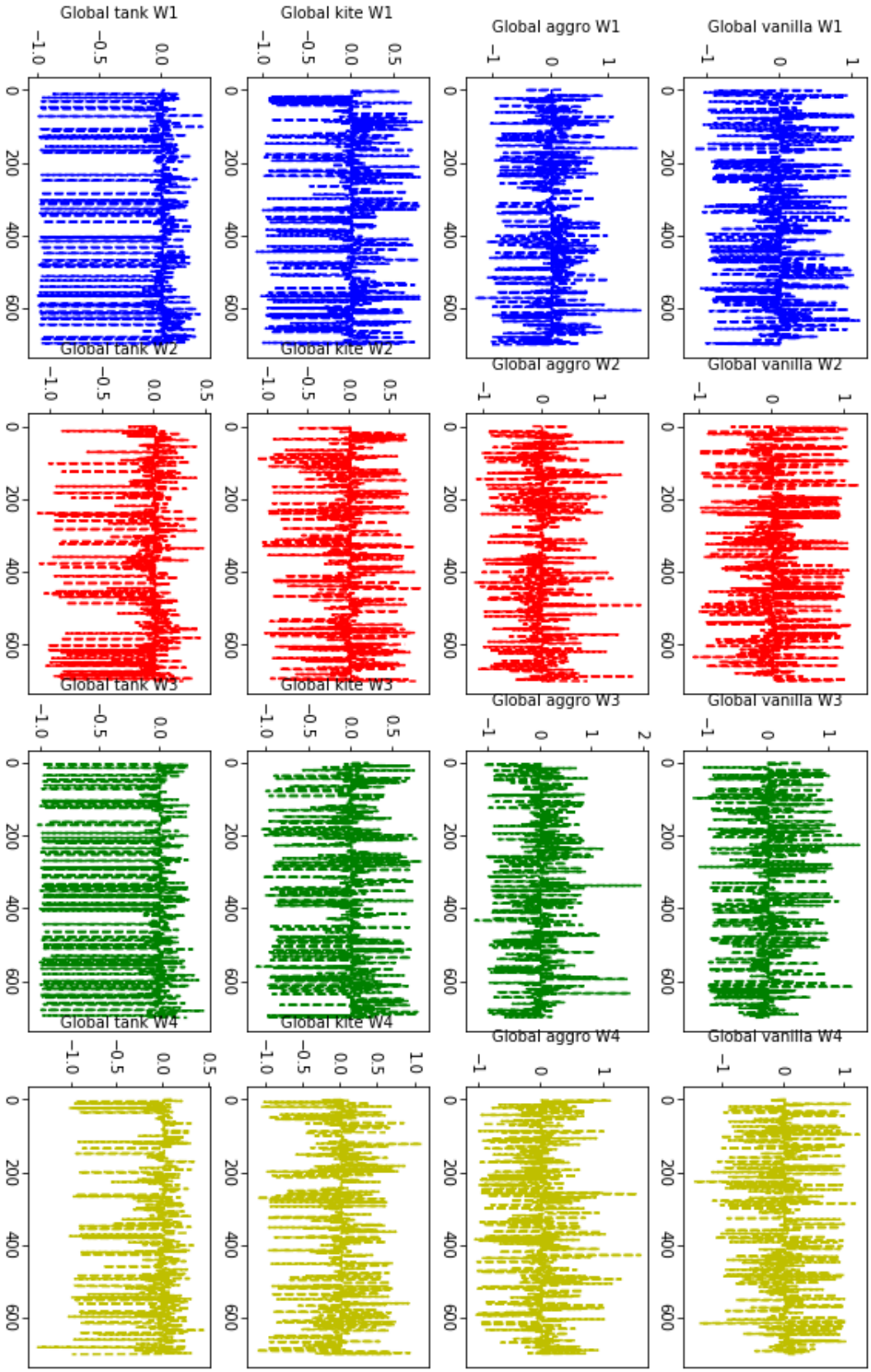


Figura 4.2: Recompensas de los agentes A3C globales en los diferentes trabajadores

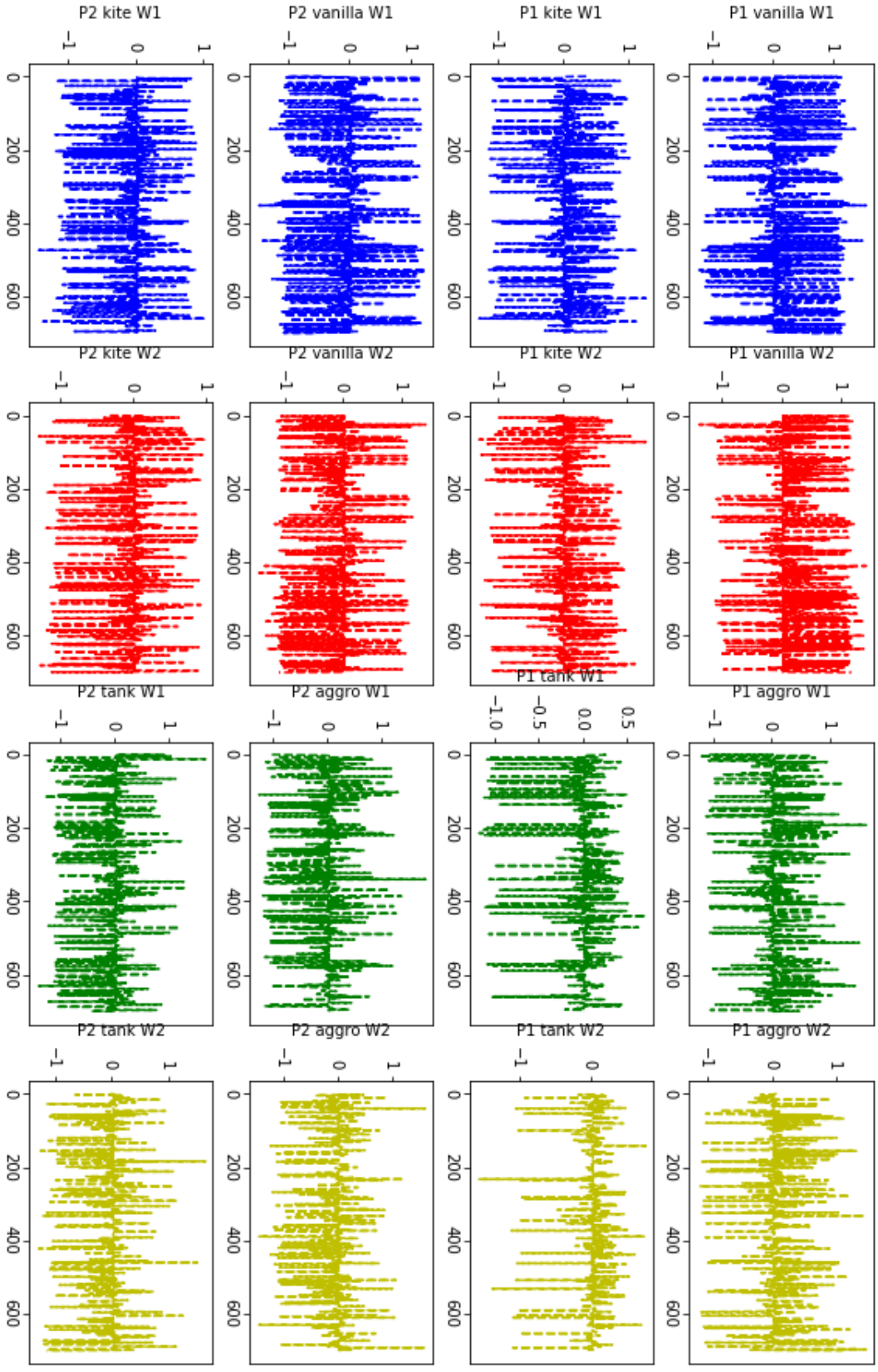


Figura 4.3: Recompensas de los agentes A3C individuales en los diferentes trabajadores

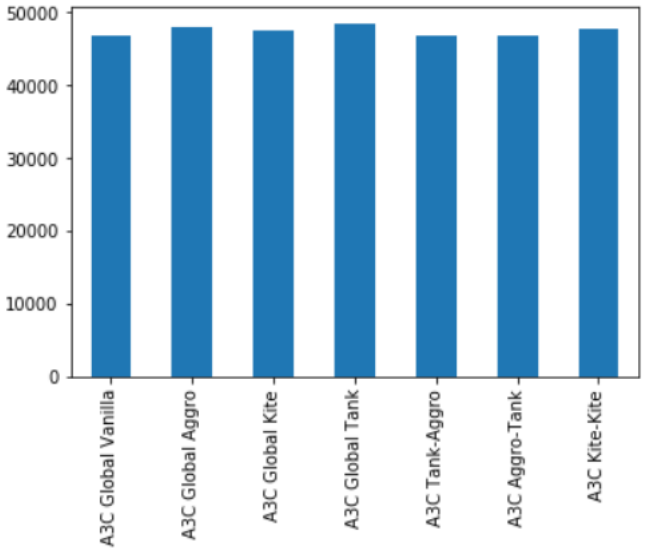


Figura 4.4: Tiempos de entrenamiento de los agentes A3C

Los enfrentamientos se atendrán a las siguientes normas:

- Por cada par de agentes se jugarán un total de 25 rondas. Estas rondas serán exclusivas, es decir, no habrán otros enfrentamientos paralelos en acción, esto es para tratar de no privar de recursos de cálculo a ninguno de los agentes.

- Se dará tiempo ilimitado a las rondas, de forma que la única forma de terminarlas será obtener una victoria definitiva, establecida por el hecho de que uno de los dos jugadores pierde toda su salud.

- En caso de que un par de agentes en una misma ronda sobrepasen el tiempo de 30 minutos más de tres veces, se cancelarán los enfrentamientos, considerando que el entrenamiento no ha sido suficiente para obtener resultados deseados y para evitar pérdidas adicionales de tiempo.

- No se realizan enfrentamientos de los agentes A3C globales entre sí, debido a que sus entrenamientos han consistido precisamente en ello y no se obtendrá información nueva al respecto.

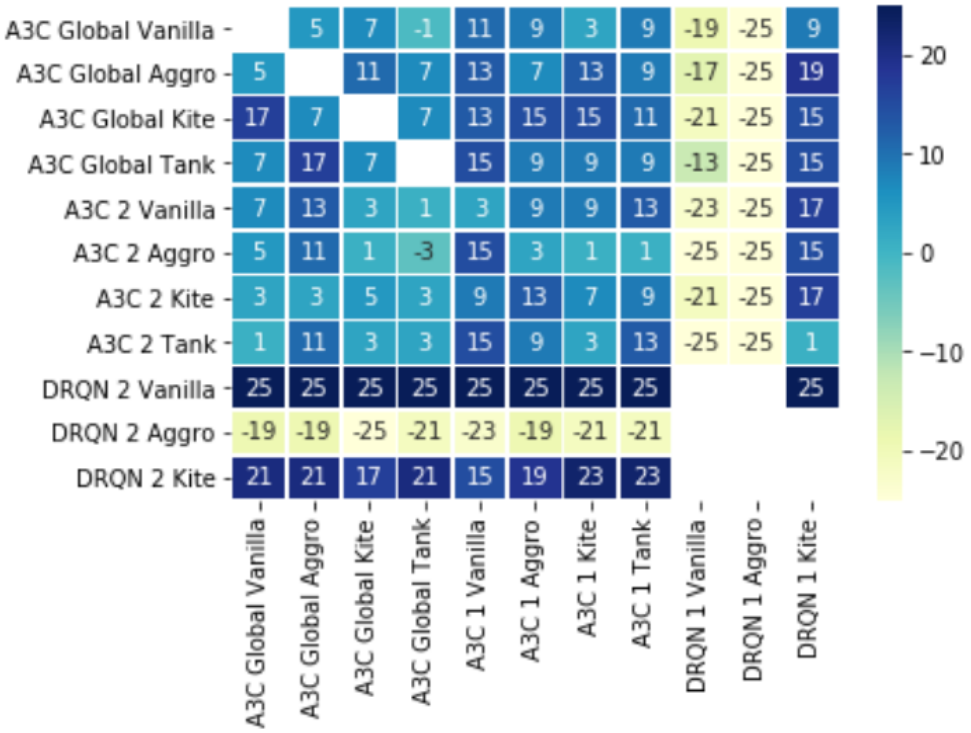


Figura 4.5: Mapa de victorias

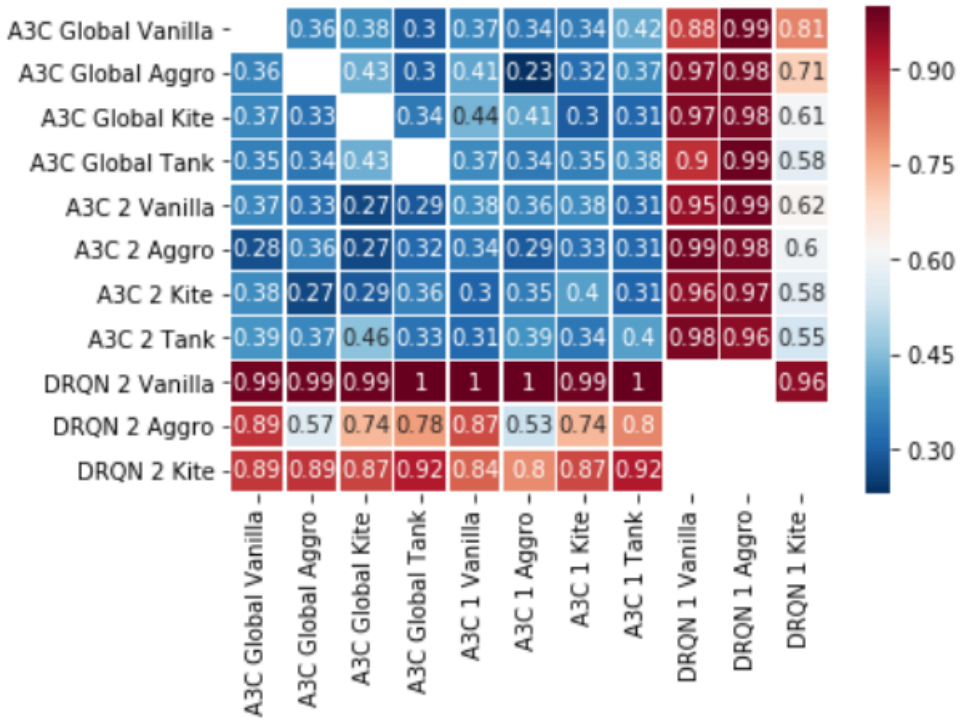


Figura 4.6: Mapa de diferencia de salud

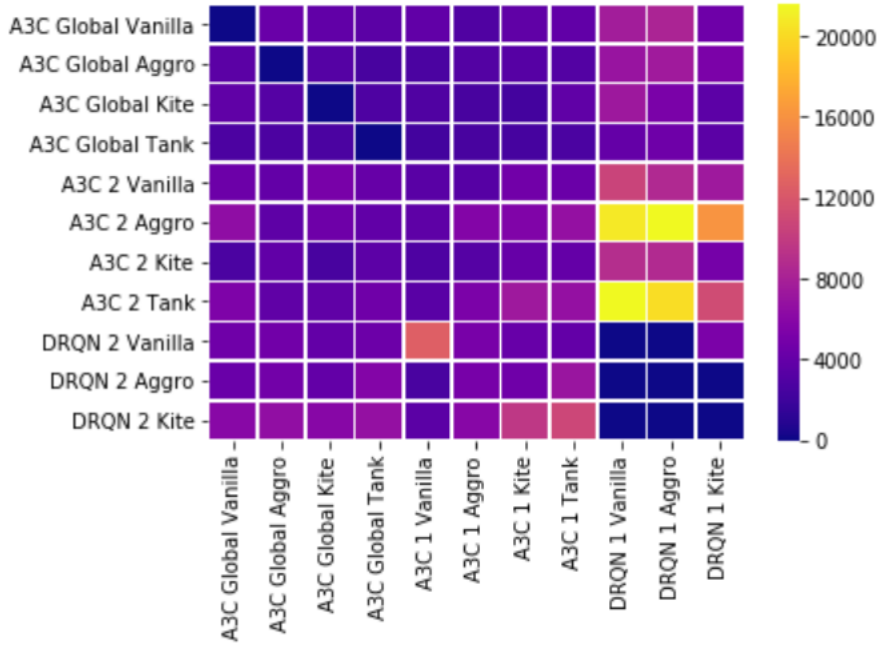


Figura 4.7: Mapa de tiempos

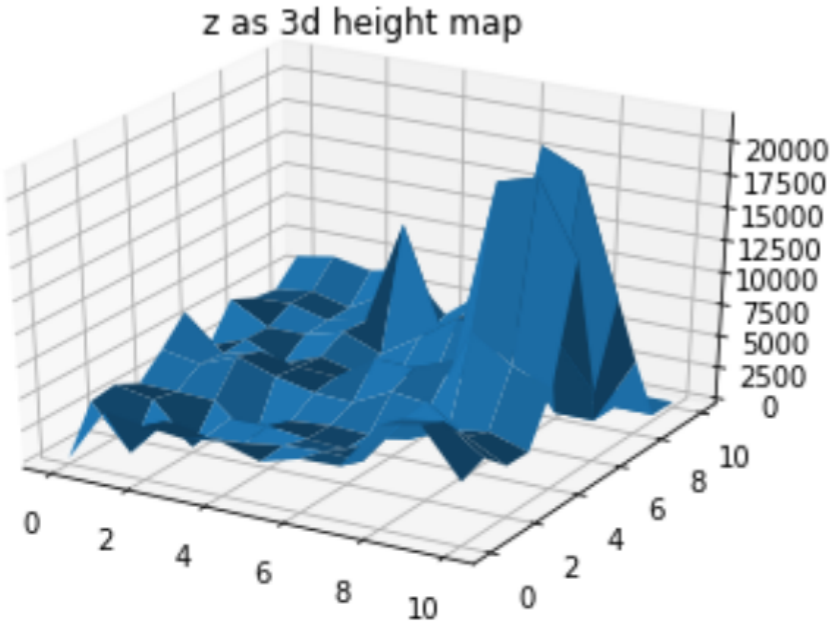


Figura 4.8: Representación 3D de los tiempos

4.2.1. Resultados de la competición

Las figuras 4.5 , 4.6 , 4.7 y 4.8 muestran visualmente diferentes parámetros que recogí en los enfrentamientos: Relación de victorias y derrotas entre los agentes, la diferencia de salud al final del enfrentamiento y los tiempos de ejecución de los enfrentamientos.

La relación de victorias es calculada con la operación 4.1, siendo V1 las victorias del jugador 1, V2 las victorias del jugador 2 y X el resultante representado en la figura 4.5.

La diferencia de salud final se calcula con la ecuación 4.2, los resultados finales siendo representados por la figura 4.6.

Por último, los tiempos de ejecución, estos simplemente el total de segundos que han tomado en ejecutarse.

$$X = V1 - V2 \tag{4.1}$$

$$Y = \frac{\sum_{n=1}^{25} |H1_n - H2_n|}{25} \tag{4.2}$$

Las mejores marcas por agente en los diferentes datos son las siguientes:

- Relación de victorias y derrotas: Agente DRQN como jugador 2 con recompensa *aggro*, consiguiendo un promedio de 23 victorias en sus enfrenamientos.
- Diferencia de salud al final del enfrentamiento: Agente DRQN como jugador 2 con recompensa *vanilla*, consiguiendo una media del 99 %.
- Tiempo de ejecución media por enfrentamiento: Agente A3C como jugador 2 con recompensa *tank*, con una media de 46 minutos y 19 segundos para la totalidad de los enfrentamientos.

Como dato adicional, recalcar que de todos las rondas en las que ambas partes son agentes DRQN tan solo aquella en la que el jugador 1 con recompensa *kite* y el jugador 2 con recompensa *vanilla* se enfrentaron llegó a cumplir la normativa de tiempo establecida previamente.

5. CAPÍTULO

Conclusiones

5.1. Conclusiones del proyecto

Aun con los resultados individuales observados en el capitulo previo, al correlacionarlos vemos que las mejores marcas no necesariamente implican los mejores resultados. Y más aún si se ven los resultados en directo. Al realizar los enfrentamientos los agentes DRQN adoptaron un comportamiento con el que realizaban una acción principal y una secundaria con menor frecuencia, el 1 % de probabilidad de producir un movimiento aleatorio generando ínfimas diferencias. Si bien adoptaron diferentes acciones en base a qué jugador representaban y que recompensa recibían, esa clase de comportamiento no permite obtener resultados óptimos.

Al observar al agente que ha obtenido la mayor diferencia de salud en el resto de figuras vemos que ha perdido en todos sus enfrentamientos y que además 2 de sus 3 rondas con el resto de agentes DRQN fueron canceladas por superar los límites de tiempo establecidos, siendo esto una indicación de que no se han obtenido buenos resultados con este agente.

Nuevamente miramos a otro agente con las mejores marcas, en este caso el que ha tenido la mejor relación de victorias y derrotas. Aunque también tenga buenos resultados en los márgenes de salud los tiempos tienden a ser más elevados y todas sus rondas con otros agentes DRQN terminaron sobrepasando los límites de tiempo establecidos. Esto es debido a que su comportamiento consistía en estar atacando constantemente, de vez en cuando se desplaza al lado contrario del escenario, pero una vez llega al final del escenario se quedaba quieto. Y aunque este comportamiento muestra que obtiene muchas victorias

contra los otros agentes A3C, estos encontraron formas de flanquearlo, con ataques de mayor rango o bloqueando para romper la cadena de ataques y entonces iniciar estos su propia cadena de ataques. Esto implica que un agente más desarrollado podría identificar este comportamiento y posiblemente lidiar con mayor facilidad con este, eventualmente convirtiendo esta estrategia de ataque constante en algo trivial con lo que lidiar.

La figura 4.7 ayuda a corroborar el hecho de que los agentes DRQN no tienen unos resultados mejores, debido a que en aquellas rondas en las que están presentes tienden a extender el tiempo de forma considerable. La figura 4.5 también indica que no obtienen resultados favorables.

Cuando miramos a los resultados de los agentes A3C vemos que hay un mayor balance en los resultados, esto siendo debido a que probablemente han obtenido una complejidad mayor, aunque no sea siempre evidente las estrategias que utilizan y que algunos movimientos parecen errores son mucho mas efectivos en términos de tiempo, variabilidad y mantenerse competitivos entre ellos. En resumen, los agentes DRQN no han obtenido buenos resultados en esta configuración, pero se tiene que tratar de probar variabilidad antes de descartarlos completamente y enfocar futuros entrenamientos exclusivamente hacia los agentes A3C.

Sin embargo, hay un fenómeno que se repite en todas las rondas: El agente que tome control del jugador 1 tiende a obtener mejores resultados. La razón por la que esto sucede no es clara. Esto es interesante en parte debido a que estudios realizados hace algunos años encontraron cierta correlación entre los colores de los jugadores y los resultados atribuidos a factores psicológicos [Ilie, 2008][1]. En el estudio se analizaron los datos de 1347 partidas de jugadores humanos con niveles de habilidad similar en un entorno pensado para no dar ventaja a ningún grupo y se observó que el 54 % de las partidas eran ganadas por el equipo rojo. Volviendo a nuestro proyecto, pasa lo contrario, siendo el jugador caracterizado por el color azul el que obtiene una mayor cantidad de victorias. Pero aún es temprano para poder atribuir esto a un fenómeno psicológico, lo más probable es que sea un efecto colateral del propio entorno, el servidor, las recompensas o estos tres factores combinados.

La sensación general que los resultados han producido es que los agentes DRQN han terminado dando unos resultados poco óptimos aun con todas las mejoras que tienen implementadas. Para corroborar que este tipo de agentes no son los apropiados para este juego en particular se debería realizar una batería de entrenamientos mas amplia, cambiando la arquitectura de la red neuronal y ajustando los diferentes gradientes de valores

[1] `https://www.liebertpub.com/doi/10.1089/cpb.2007.0122`

de actualización, para poder inferir si existe una forma de obtener mejores resultados.

Por otra parte los agentes A3C, aunque aparentemente con comportamientos más aleatorios dan una sensación de tener mejor futuro a corto plazo si se les da más tiempo para entrenar y se les da una mayor amplitud de oponentes, lo cual es ahora realizable con todos los agentes actualmente entrenados.

Como resultado adicional de este proyecto, el servidor producido en el proyecto puede ser utilizado (tras leves modificaciones) en otros juegos y entornos que sean representables en un navegador web. Por tanto el proyecto es ampliable no solo a videojuegos, también a interacción en la web. Servicios de *streaming* permitirían la ejecución remota de entornos más complejos y restar la carga de la máquina en la que se realice el entrenamiento o uso de los agentes.

5.2. Conclusiones personales

Las conclusiones personales derivadas de este proyecto han sido el resultado de todo el tiempo invertido en este y todo el aprendizaje que ha conllevado. No solo he obtenido una mayor comprensión sobre los algoritmos que actualmente están en uso para generar IA en videojuegos, sino que también lo importante que es la correcta realización de las planificaciones y cumplimiento de plazos de tiempo cuando esto se aplica a tareas que consumen muchos recursos. Esto lo observé mejor en las fases de entrenamiento y de competición, las cuales al extenderse más de lo previsto dificultaron ahorrar todo el tiempo posible, en muchos casos debido a que los entrenamientos finalizaban en horas nocturnas en las que no estaba despierto para activar el siguiente, perdiendo así un cúmulo de horas pasivas importante.

También ha quedado en evidencia que para este tipo de proyectos tener acceso a más hardware con el que realizar las tareas de forma paralela es de vital importancia, en especial cuando un proyecto lo he llevado en su mayor parte por mi cuenta. De haber tenido en disposición de ese hardware añadido el proyecto podría haber pasado por una segunda fase de entrenamiento y más competiciones, todo esto para obtener más datos finales y conclusiones más concretas sobre los resultados obtenidos.

Aun con todas las dificultades pasadas (y las ganas recurrentes de lanzar el ordenador por la ventana cuando no localizaba el origen de errores), el proyecto en sí ha sido un trabajo gratificante y que me ha dejado con ganas de poder continuarlo para ver hasta donde se puede llevar el aprendizaje profundo y que aplicaciones del mundo real podría tener. Sería

mentir si negase que quisiera poder verlo en acción contra jugadores humanos y que estos fueran pulverizados por alguno de los agentes.

5.3. Líneas futuras

- Ampliar y escalar el servidor para poder interactuar con más entornos. Tratar los diferentes entornos como paquetes individuales que se le suministran al servidor.
- Optimización y compartimentalización del código para mayor legibilidad de este.
- Modificar las redes neuronales, gradientes y otros factores para observar si se obtienen mejores resultados.
- Continuar los entrenamientos para aumentar el refinamiento de los agentes.
- Prestar atención a la posible aparición de nuevos algoritmos o mejoras sobre los que ya existen para poder probarlos.
- Cuando se obtengan resultados satisfactorios en los agentes, enfrentarlos a jugadores humanos.

Anexos

A. ANEXO

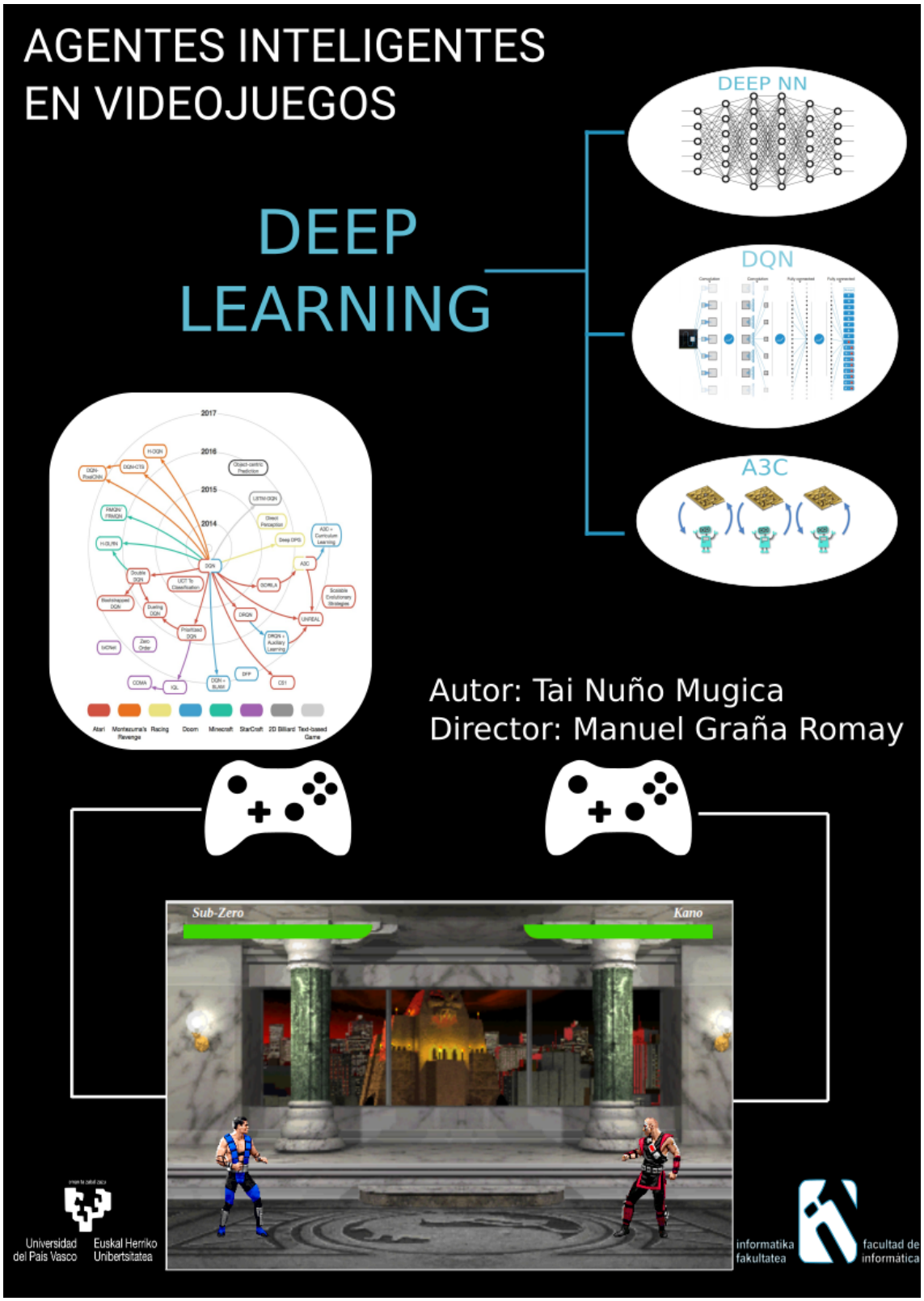
Anexos

A.1. Código del proyecto

El código resultante del proyecto está disponible en las plataformas GitHub[1] y Zenodo[2].

[1] https://github.com/TaiPrev/TFG_2018-2019
[2] https://zenodo.org/record/3250976#.XQy8EyZS-xt

A.2. Póster del proyecto



Bibliografía

[NAT, 2015] (2015). Human-level control through deep reinforcement learning. *Nature volumen 518, paginas 529–533*.

[Hausknecht and Stone, 2015] Hausknecht, M. and Stone, P. (2015). Deep Recurrent Q-Learning for Partially Observable MDPs.

[Ilie, 2008] Ilie, A., I. S. Z. L. . M. M. (2008). Better to be red than blue in virtual competition. *Cyberpsychology, Behavior, and Social Networking volumen 11, paginas 375–377*.

[Lample and Chaplot, 2016] Lample, G. and Chaplot, D. S. (2016). Playing FPS Games with Deep Reinforcement Learning.

[van Hasselt et al., 2015] van Hasselt, H., Guez, A., and Silver, D. (2015). Deep Reinforcement Learning with Double Q-learning.

[Volodymyr Mnih and Kavukcuoglu, 2016] Volodymyr Mnih, Adrià Puigdomènech Badia, M. M. A. G. T. P. L. T. H. D. S. and Kavukcuoglu, K. (2016). Asynchronous Methods for Deep Reinforcement Learning.

[Ziyu Wang, 2015] Ziyu Wang, Tom Schaul, M. H. H. v. H. M. L. N. d. F. (2015). Dueling Network Architectures for Deep Reinforcement Learning.