Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Creación de tareas cooperativas multiagente en Minecraft

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Luis Armas Suárez

Tutor: José Hernández Orallo

2016-2017

Resumen

Esta es la memoria escrita que presenta el Trabajo de Fin de Grado de la titulación Grado en Ingeniería informática.

Este trabajo realiza un estudio sobre la cooperación y aprendizaje entre agentes inteligentes que utilizan diversas técnicas. En particular, se analiza la utilidad del algoritmo DQN, un algoritmo de *Deep Reinforcement Learning*, para problemas de cooperación entre agentes, así como su coste, utilidad e implementación.

El desarrollo del trabajo consiste en la programación e implementación de un agente inteligente que aprenda a jugar al minijuego colaborativo 'Catch the Pig' del concurso *The Malmo Colaborative Challenge* de Microsoft. El minijuego está implementado en Project Malmo, una plataforma para la investigación y experimentación de inteligencia artificial creado a partir del videojuego Minecraft. Para lograr el aprendizaje del agente se utiliza el algoritmo DQN, el cual será explicado y analizado.

El trabajo también pretende hacer un estudio de la cooperación y aprendizaje entre agentes inteligentes así como del estado y desarrollo de este campo en la actualidad, a demás de hacer un análisis y comparativa de los resultados obtenidos tanto por el agente inteligente desarrollado para este trabajo como por los implementados por los otros participantes del concurso.

La documentación que se presenta en este trabajo es diversa. En la memoria se expone información relevante para la comprensión del trabajo y de sus condiciones, así como de la plataforma utilizada para su desarrollo. También se explica de forma generalizada la implementación del algoritmo DQN adaptada para este juego. A continuación se exponen sus resultados así como los de otros agentes presentados para el concurso y se realiza un análisis de estos para proporcionar un mayor entendimiento de la cooperación entre agentes inteligentes y proponer futuras mejoras para el trabajo.

Palabras clave: Malmo, Deep Reinforcement Learning, agente inteligente, cooperación multiagenete, Minecraft, inteligencia artificial

Tabla de contenidos de la memoria

Tabla de figuras de la memoria

Tabla de contenidos del anexo

Tabla de figuras del anexo

1. Introducción

La inteligencia artificial es hoy en día uno de los campos de la informática y de las ciencias de la computación más estudiados y a la vez más desconocidos dada su amplitud, utilidad y relativa novedad. Se trata de un campo todavía en pleno desarrollo del que aún falta mucho por descubrir y aprender, y dado su amplio rango de utilidad hay tantos enfoques como aplicaciones tiene.

La inteligencia artificial fue introducida a la comunidad científica en 1950 por el matemático Alan Turing [1], mientras que el término como tal fue acuñado en 1956 por John McCarthy definiéndose como "*La ciencia e ingenio de hacer máquinas inteligentes, especialmente programas de cómputo inteligentes*" [2]. En la actualidad, un agente inteligente se considera como "*un agente racional que es capaz de percibir su entorno y, en base a estas percepciones, llevar a cabo acciones que maximicen su éxito en las tareas y objetivos que tenga asignados*" [3]. Ciertas cualidades que hacen que un agente se considere inteligente es su capacidad de reproducir ciertas funciones cognitivas que se asocian a la mente humana o de ciertos animales como el aprendizaje o la resolución de problemas.

Los sistemas inteligentes suelen buscar la solución o el objetivo en el conjunto de estados producido por las acciones posibles, para la cual se vale de diversas técnicas. Algunas de las más importantes son el uso de redes neuronales artificiales, los algoritmos genéticos y de aprendizaje por refuerzo o el uso de la lógica formal para generar razonamientos similares a los producidos por el cerebro humano [2].

Como se ha mencionado antes, dada su versatilidad la inteligencia artificial ha sido aplicada a multitud de campos tan diversos como la medicina, la economía, la industria, la ingeniería o los videojuegos, y según se acelera su avance es utilizada en más áreas.

En este trabajo se explica el desarrollo y evaluación de un agente inteligente implementado en una nueva plataforma de experimentación de IA. Las motivaciones, objetivos y desarrollo del proyecto se expondrán a continuación. Para la terminología y conceptos del proyecto, el lector puede referirse al apéndice *Terminología* que se encuentra al final de este documento.

1.2 Motivaciones

Como se ha comentado anteriormente, la inteligencia artificial es aún un campo en el que queda mucho por estudiar, y ha sido en los últimos años cuando mayor énfasis se la ha dado y mayores avances se han hecho, ya sea debido a la capacidad de la tecnología actual o gracias a el interés que se ha mostrado en los últimos años por este campo.

Las innumerables aplicaciones que puede tener la IA y los resultados obtenidos hasta ahora hacen que se esté considerando como la base de muchas de las tecnologías del futuro, pero para ello es primordial seguir avanzando e investigando.

A pesar de que se han conseguido grandes resultados y avances a partir de la inteligencia artificial en enseñar a agentes a realizar tareas específicas, como por ejemplo que sean capaces de reconocer imágenes o "entender" el lenguaje humano, sigue sin haber mucho éxito y conocimiento en lo que los investigadores llaman inteligencia general. Este tipo de inteligencia es más similar a la forma en la que los humanos aprenden y toman decisiones, es decir, a través de los estímulos externos - la vista, el tacto, el olfato, el sonido o las sensaciones - los humanos aprendemos que ciertas acciones producirán (o al menos, tienen una alta probabilidad de producir) otras reacciones, buscando así las deseadas [4]. Aunque hoy en día haya algoritmos que puedan realizar tareas de forma mejor o más eficiente que los humanos, el avance para lograr esta inteligencia general o cognitiva está aun en sus primeras etapas y de ahí el interés, la motivación y la necesidad de estudiarlas.

Con la plataforma Project Malmo se ha creado la posibilidad de experimentar con sistemas de IA de forma mucho más fácil, eficiente y con muchas menos limitaciones que las que presentaban otras plataformas de experimentación para la inteligencia artificial. Aprovechar esta nueva herramienta para la investigación a la vez que se ayuda al desarrollo de la plataforma es también otra las razones del interés por el trabajo.

El reto propuesto por el concurso *The Malmo Collaborative Challenge* desafía a estudiantes de todo el mundo a buscar diferentes soluciones para un juego multiagente colaborativo con el fin de aportar más información al estudio de la colaboración multiagente y es por tanto otra de las motivaciones para llevar a cabo este proyecto. Este trabajo se inició con la motivación de proponer una solución que se aproxime al modelo de inteligencia “general” que se está buscando con el fin también de aprender de esta y así aportar información a este extenso campo de investigación.

Dada la novedad y la potencia de la plataforma Malmo es necesario explotarla y estudiar cómo puede esta llegar a utilizarse para experimentar con este tipo de tareas colaborativas. Malmo es una plataforma de código abierto, por lo que desde investigadores hasta estudiantes y curiosos pueden utilizarla para realizar sus propios estudios y experimentos. Por lo tanto, junto con el reto propuesto por el concurso, otra motivación para este trabajo ha sido comprobar si es posible, con unos recursos limitados, reproducir algunas de las técnicas más punteras en IA en esta plataforma, así como algunos de los resultados de la competición.

Investigar y trabajar en un campo pionero para aportar en su desarrollo a la vez que se utilizan las más novedosas técnicas supone una gran motivación debido a que permite profundizar y aplicar los conocimientos sobre las ciencias de la computación y la inteligencia artificial obtenidos a lo largo de los estudios de grado, a la vez que brinda la oportunidad de adquirir otros nuevos y más complejos sobre las últimas técnicas, plataformas y algoritmos utilizados en este campo, como es el caso del *Deep Reinforcement Learning* o el uso de la plataforma Malmo.

1.3 Objetivos del trabajo

El objetivo principal de este proyecto es el diseño e implementación en la plataforma Malmo de un agente inteligente que aprenda a jugar al minijuego colaborativo Pig Chase mediante el algoritmo de DQN para a continuación realizar pruebas de su desempeño y, mediante estos resultados y los de los otros participantes del concurso

The Malmo Colaborative Challenge, sacar conclusiones de su utilidad y de la colaboración multiagente entre agentes inteligentes.

Como se ha expuesto en el apartado anterior, las motivaciones que han impulsado este proyecto han sido varias, y de estas se han derivado una serie de objetivos que se han tratado de cumplir con el desarrollo de este proyecto.

El desglose de los objetivos es el siguiente:

- Crear un agente inteligente que aprenda a jugar al mini-juego Pig Chase mediante la adaptación del algoritmo de Deep Reinforcement Learning.
- Realizar una serie de pruebas sobre el agente para medir el aprendizaje, la eficiencia y resultados de este, así como la de los otros agentes creados por los participantes del concurso *The Malmo Colaborative Challenge*.
- Realizar un análisis y estudio sobre la cooperación entre agentes inteligentes en tareas colaborativas.
- Comprobar la utilidad y el desempeño del algoritmo DQN en tareas cooperativas complejas.
- Extraer información para poder proponer futuras mejoras y sugerencias para este trabajo u otros relacionado con la cooperación entre agentes inteligentes
- Poner a prueba la plataforma Malmo tratando de ver cuán reproducibles y accesibles son las creación de tareas y agentes de una competición puntera en el campo de investigación de la inteligencia artificial.

1.4 Estructura de la memoria

En este apartado se resume el contenido expuesto en cada una de las partes de la -memoria:

INTRODUCCIÓN

Da una visión general del proyecto, introduciendo conceptos necesarios para su comprensión y se explica brevemente el software utilizado. Se describen también las motivaciones que han llevado al interés y al desarrollo del proyecto, así como los objetivos de su realización.

EVALUACIÓN EN IA: LA PLATAFORMA MALMO

Se realiza un breve resumen de sobre la historia y estado actual de la investigación en la inteligencia artificial, a la vez que se exponen con mayor detalle la plataforma utilizada para este proyecto así como el contexto de la competición *The Malmo Collaborative AI Challenge*.

DESARROLLO E IMPLEMENTACIÓN DEL AGENTE INTELIGENTE

Se explica paso a paso el desarrollo e implementación del agente inteligente junto con su adaptación para el juego Pig Chase y la plataforma Malmo

PRUEBAS Y RESULTADOS DEL AGENTE

Se exponen los resultados obtenidos tanto por el agente desarrollado para este proyecto como por los de otros participantes del *The Malmo Colaborative Challenge,* seguido de un análisis de estos resultados y de sus posibles aplicaciones

CONCLUSIONES

Se exponen los resultados de la aplicación del proyecto, así como conclusiones sacadas, lecciones aprendidas y posibles mejoras y trabajos futuros para perfeccionarlo.

BIBLIOGRAFÍA Y REFERENCIAS

Bibliografía y referencias utilizadas para la realización del proyecto

2. Evaluación en la IA: La plataforma Malmo

2.1 Historia de la inteligencia artificial

¿Puede una máquina pensar? A pesar de que esta se trate de una pregunta de índole más bien filosófica, fue esta el germen de la idea que llevó al inglés Alan Turing a su investigación y a hacer que la comunidad científica hiciera un mayor enfoque en esta y en sus posibilidades. Esta pregunta inicial también creó los primeros debates entre la ciencia computacional y la psicología, llevando a algunos experimentos y pruebas como la del test de Turing o a la creación de la primera máquina capaz de jugar al ajedrez [1].

Tras la muerte de Turing, su investigación fue continuada por otros que más tarde serían considerados los padres fundadores de la inteligencia artificial, como Von Neumann o Minsky. Es estudio de la IA se ha ido ampliando y afianzando con los años, y a día de hoy se define como *"la inteligencia exhibida por las máquinas"[3].* Se considera que una máquina es inteligente cuando imita funciones cognitivas asociadas a el cerebro humano, como el aprendizaje o solucionar problemas.

La investigación de la inteligencia artificial se divide en diferentes aproximaciones y problemas con el objetivo de obtener diferentes aplicaciones de esta. Los principales objetivos de las investigaciones son generalmente la búsqueda del razonamiento, el aprendizaje, conocimiento, la planificación, el procesamiento del lenguaje natural y la percepción[2]. A través de estos diversos estudios se han creado algoritmos y aproximaciones que hoy en día son la base de la inteligencia artificial, tales como las redes neuronales, redes bayesianas, los modelos de Markov, el aprendizaje por refuerzo, los algoritmos genéticos, la teoría de juego etc.

A través de estos diferentes enfoques se han conseguido implementar varias soluciones para los problemas planteados, dando lugar a múltiples aplicaciones que se utilizan en gran medida tanto en la industria como en la economía o la medicina debido a sus resultados. Entre algunas de estas aplicaciones se encuentran, por ejemplo, el reconocimiento facial, el del lenguaje, la creación de vehículos auto-pilotados, mejora de los algoritmos de búsqueda o los asistentes inteligentes, como Siri o Cortana

2.2 La plataforma *Project Malmo*

Como se comentaba en la introducción, aunque se hayan hecho grandes avances en el estudio de la inteligencia artificial y se hayan obtenido grandes resultados, todas estas soluciones han sido para problemas específicos, siendo difícilmente generalizables. Esto da a entender que la pregunta formulada por Turing está aún muy lejos de ser comprobada y respondida, y que la meta de conseguir una máquina que emule de forma general las capacidades cognitivas del ser humano requiere mucha más investigación, experimentación y desarrollo.

Uno de los principales problemas a los que se enfrenta dicho objetivo es a la manera de poder experimentar de forma fácil, eficiente y sencilla. A pesar de que existen otras plataformas para la experimentación para la IA, muchas de estas presentaban múltiples limitaciones, como por ejemplo, limitar las pruebas que se le pueden hacer a las IAs a ciertos juegos sencillos. Por ello, en el equipo de Cambridge de Microsoft decidieron desarrollar Malmo, una plataforma de experimentación para la inteligencia artificial creada como un *mod* del popular videojuego Minecraft.

Para los que desconozcan Minecraft, se trata de un videojuego de mundo abierto en el que el jugador puede interaccionar prácticamente con todo. Este mundo que se presenta ante el jugador se genera de forma procedimental, es decir, de forma automática, por lo que es infinito y presenta una gran variedad de escenarios diferentes que se podrían encontrar en el mundo real, como zonas de montaña, cuevas, desiertos, ciudades, bosques etc. La razón para elegir este juego como base de la plataforma Malmo es precisamente esta, su capacidad para generar escenarios que son cada vez diferentes, a parte de la posibilidad de interaccionar con prácticamente todos los elementos de este. Para la experimentación en inteligencia artificial, esto supone la posibilidad de experimentar en una "recreación" del mundo real de manera sencilla, fácil y barata, sin necesidad de construir máquinas con aparatos que realmente puedan percibir el mundo real o de verse limitada por las restricciones de las otras plataformas existentes hasta el momento.

Malmo por tanto permite crear todo tipo de misiones con las herramientas que se prestan, a su vez que permiten implementar multitud de diferentes algoritmos y aproximaciones para solucionar estas. Uno de los ejemplos que viene por defecto junto a la plataforma Malmo es una misión en la que un agente debe llegar a una salida utilizando el algoritmo de Q-Learning, como se muestra en la figura 1.

Figura 1: Agente usando algoritmo Q-Learning en la plataforma Malmo (Extraída de [5])

La plataforma Malmo es de código abierto, por lo que puede descargarse de manera gratuita y no es necesario comprar el juego para utilizarla. Los agentes creados pueden programarse en el lenguaje que el usuario desee, ya sea por ejemplo Java, C# o Python. Para la realización de este proyecto, todos los agentes han sido programados con este último.

Entra las funcionalidades de Malmo se encuentra la posibilidad de crear entornos de forma fácil y flexible específicos para cada prueba, pudiendo utilizar todos los materiales existentes en Minecraft, así como utilizar los NPCs del juego, enemigos o

cambiar estados del entorno como la hora del día o el clima. Todo el entorno, los NPCs que tendrá, la posición de estos y del jugador/jugadores, las herramientas o armas que tiene cada agente etc. se define en un archivo XML .

En este archivo también se define la misión, es decir, los objetivos que tiene el agente, el tiempo que tiene para cumplirlos y las recompensas/penalizaciones por cada acción realizada. Se pueden asignar penalizaciones/recompensas por tocar diferentes tipos de bloques, por coger X objetos, por llegar a ciertos puntos del mapa o por realizar ciertas acciones. De esta forma, y combinado con el editor de escenarios, se pueden modelar todo tipo de misiones diferentes para realizar pruebas a agentes inteligentes que se adapten a los objetivos de cada investigación.

Para interactuar con el mundo y los agentes, Malmo permite la observación del entorno, pudiendo obtener una "visión" de los bloques y objetos cercanos que rodean al agente, siendo posible ampliar o disminuir el rango de esta observación. Estas son devueltas en forma de archivo JSON.

Una vez el entorno está construido, la misión está definida junto con sus recompensas y penalizaciones y el agente puede percibir su entorno, solo queda definir cómo actuará este en base a esta situación. Como se ha comentado antes, los scripts que definen el comportamiento del agente pueden ser escritos en varios lenguajes de programación, y es aquí donde se pueden enviar comandos al personaje en Minecraft para que interactúe con el mundo. Todas las posibles acciones que puede realizar un jugador normal (como por ejemplo saltar, atacar, cambiar de arma, girarse, andar etc.) puede realizarlas el agente de Malmo a través de los comandos de control, a demás de poder decidir también la cantidad de tiempo que pasan realizando un comando o la velocidad a la que los realizan.

Otra de las ventajas de la plataforma Malmo es que permite realizar misiones con varios agentes, siendo así posible el estudio de su interacción y ampliando la variedad de pruebas y experimentos que pueden realizarse con esta. El hecho de que los agentes puedan implementarse con diferentes lenguajes de programación también permite el uso de diferentes librerías para estos que ayuden a la programación de los diferentes agentes inteligentes.

Con todas estas herramientas y características de la plataforma, el equipo de Microsoft Cambrige ha logrado que la experimentación con agentes inteligentes sea mucho más flexible y barata, y al hacerla gratuita y de código abierto, a día de hoy desde investigadores profesionales hasta estudiantes o autodidactas están contribuyendo a el desarrollo y avance de la inteligencia artificial.

2.3 The Malmo Colaborative AI Challenge

El desarrollo de este proyecto de fin de grado se ha basado principalmente en el concurso *The Malmo Colaborative AI Challenge*, el cual fue propuesto por Microsoft en 2017. Una de las principales metas del desarrollo de la inteligencia artificial es lograr que los agentes inteligentes aprendan a colaborar con humanos y/o con otros agentes para conseguir sus metas. Sin embargo, la propia colaboración entre agentes es un tema sobre el que aún queda mucho que resolver y estudiar, ya que presenta algunos desafíos, como la dificultad de un agente para reconocer las intenciones de otro agente,

la comunicación y coordinación entre agentes para adoptar una estrategia que resuelva el problema o el aprendizaje de qué acciones o comportamientos ayudan a lograr el objetivo común.

Para aportar un poco de luz a estas cuestiones y encontrar soluciones a estos desafíos, o, al menos, acercarnos a ello, se propuso esta competición, con el objetivo de promover la investigación en la colaboración entre agentes inteligentes. El desafío propuesto para esto consiste en crear y entrenar un agente inteligente que consiga obtener las mejores puntuaciones para el mini-juego colaborativo *Pig Chase*.

Pig Chase está basado en la teoría de juego y en el *Stag Hunt*, un juego que describe un dilema entre la cooperación y la seguridad de una recompensa. El *Stag Hunt* define una situación en la que dos individuos van de caza. Cada uno puede elegir entre cazar un ciervo o un conejo, lo cual deciden sin saber la elección del otro agente. Si uno de los individuos decide cazar al ciervo, necesita la ayuda del otro para llevar a cabo la caza con éxito. Si por el contrario se decide cazar un conejo podrá hacerlo de forma individual y siempre tendrá éxito, pero la recompensa que se ofrece por cazarlo es mucho menor que la obtenida por cazar a un ciervo[6].

El dilema que se presenta con esta situación es por tanto el siguiente: ¿Ir a por el ciervo, confiando en que el otro individuo elegirá también la cooperación para obtener una mayor recompensa y arriesgarse a no obtener nada, o ir a por una recompensa segura pero menor? Esta situación presenta dos equilibrios de Nash posibles: Uno en el que ambos deciden ir a por el ciervo y otro en el que ambos deciden ir a por el conejo. Esto quiere decir que, en estos dos equilibrios, ninguno de los dos individuos puede obtener una recompensa mayor que la actual cambiando de estrategia mientras el otro jugador mantenga la suya. En la figura 2 se muestran estos equilibrios junto a el resto de posibles resultados del problema.

		Cazador 1	
		Ciervo	Conejo
Cazador 2	Ciervo	4,4	0,3
	Conejo	3,0	3,3

Figura 2:Tabla de posibles resultados del juego Stag Hunt(Extraída de [6])

Este problema se ha considerado en múltiples ocasiones como una analogía de la cooperación social, y en el caso de *The Malmo Colaborative AI Challenge* nos sirve de ejemplo para estudiar la cooperación entre agentes inteligentes. La caza del ciervo, traducida a la plataforma Malmo en forma de *Pig Chase* tiene las siguientes características:

En este mini-juego, dos agentes se encuentran en un corral encerrados con un cerdo que se mueve libremente por él. Dentro de este corral hay, además, dos plataformas de

color azul, una a cada lado del corral. Para atrapar al cerdo los agentes deben moverse de forma coordinada para acorralar al cerdo en el corral para así poder atraparlo.

Los agentes deben capturar al cerdo en 25 movimientos/acciones o menos, ya que por cada acción que realicen estos reciben una penalización de -1 puntos y la recompensa por atrapar al cerdo es de +25 puntos para cada uno. Los agentes tienen una segunda opción, la cual consiste en no cooperar con el otro agente y dirigirse hacia una de las casillas azules del mapa. El agente que toque primero una de estas casillas recibe +5 puntos, mientras que el otro agente se queda sin recompensa alguna.

Las acciones disponibles para cada agente son girar hacia la derecha, girar hacia la izquierda o avanzar una casilla hacia delante (hacia donde se encuentran mirando en ese momento). Como se ha indicado, por cada una de estas acciones se recibe una penalización de -1 puntos, por lo que cuanto más se tarde en atrapar al cerdo o llegar a una de las casillas de salida, menor será la puntuación total.

Para decidir cómo actuar, el agente es capaz de percibir el tipo de bloques que hay a su alrededor, así como su posición, la del otro agente y la del cerdo.

En este entorno por tanto se plantea un entorno muy similar al propuesto por el *Stag Hunt* que los agentes inteligentes deben solucionar, ya sea mediante la cooperación entre estos o mediante la búsqueda de una recompensa individual pero menor. Tanto el entorno como la misión se encuentran ya definidas para todos los participantes, por lo que este es el punto de partida de la competición.

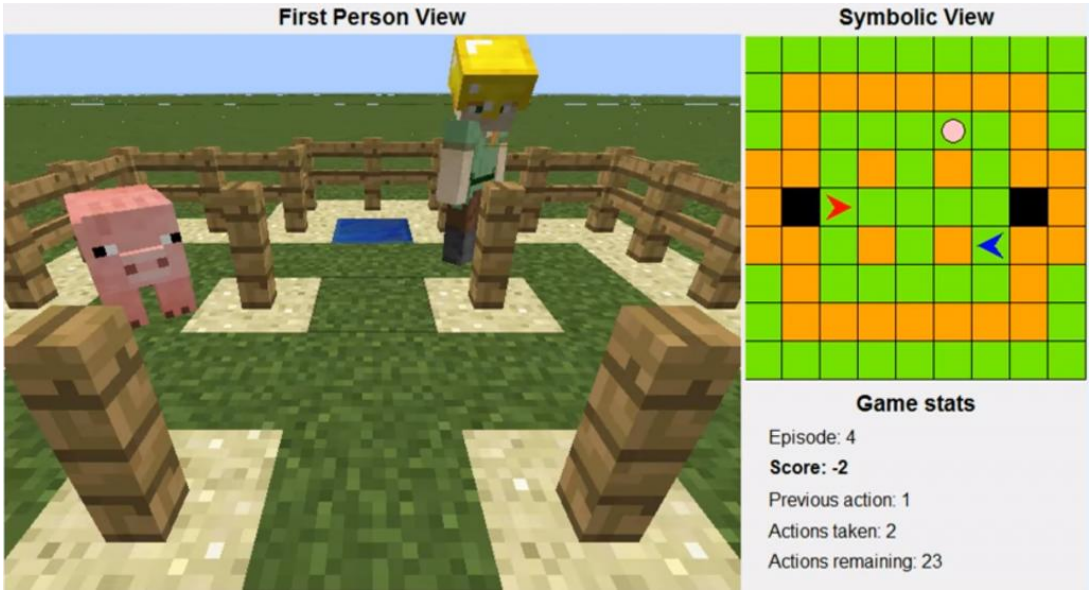

Figura 3: Mini-juego Pig Chase en acción. (Extraída de [7])

3. Desarrollo e implementación del agente inteligente

3.1 Aproximaciones del problema

Con el pretexto de la competición se estudiaron las diferentes aproximaciones que se podrían realizar para que el agente inteligente aprendiera a jugar al mini-juego Pig Chase. Se consideraron algunas posibles soluciones como el uso de heurísticas en base a la proximidad y posición del cerdo y del otro agente o el uso de algoritmos de aprendizaje automático tales como algoritmos genéticos o algoritmos de aprendizaje por refuerzo.

La mayoría de estos enfoques no resultaban demasiado aptos para el caso del mini-juego ya que en cada partida tanto los movimientos del cerdo como el de los dos agentes varían. Por tanto, una aproximación basada en la obtención de una única solución no resulta viable para este caso.

Debe resaltarse que uno de los objetivos con los que se inició este proyecto era crear un agente que aprendiera a jugar al juego con un método que se aproximara lo máximo posible a una inteligencia "genérica". Por tanto, los métodos en planteados para resolver este juego no se contemplaron soluciones hechas a medida para el juego, ya que eso aportaría información sobre la cooperación entre agentes en este juego en concreto y no de una manera general.

Una de las posibles aproximaciones que se plantearon fue el uso de un algoritmo *Q learning* basado en la posición relativa otro agente y del cerdo con respecto al agente observador, es decir, los diferentes estados para el agente se construían en base a si el otro agente y el cerdo están o no al norte, sur, este u oeste relativos del agente. Esto daba lugar a ocho variables con dos estados posibles cada una, dando lugar a 256 estados posibles. Para el algoritmo *Q learning* esto es algo asequible, a pesar de las múltiples repeticiones que tendrían que realizarse para que el agente aprendiera.

Otro de los planteamientos posibles fue el uso del algoritmo *Q learning* en base a la posición real de los dos agentes y del cerdo. Esto, sin embargo, da lugar a una cantidad ingente de posibles estados que el algoritmo *Q learning* no puede abarcar. Partiendo de este planteamiento se llegó a la idea que daría aproximación que se ha tomado finalmente para este trabajo: Deep Reinforcement Learning.

Tras ver los resultados obtenidos por este algoritmo en otros juegos individuales en el estudio *Playing Atari with Deep Reinforcement Learning[8]* en base a la información obtenida por el entorno y a la puntuación obtenida por partida, se consideró el algoritmo DQN como idóneo por su potencial, su similitud a la "inteligencia general" buscada. Por tanto, se decidió poner a prueba la capacidad de aprendizaje de este algoritmo en un juego cooperativo y en un entorno en 3 dimensiones con el objetivo no sólo de que aprendiera satisfactoriamente a jugar sino también de observar su comportamiento en cuanto al aprendizaje de políticas de cooperación multiagente.

3.2 Deep Reinforcement Learning y el algoritmo DQN

A pesar de que se ha conseguido mediante diferentes técnicas que agentes artificiales aprendan a realizar tareas específicas con gran éxito como el reconocimiento de imágenes o del habla, sigue habiendo una gran brecha entre estos y la inteligencia "general" que tiene el ser humano, es decir, que su inteligencia no sea específica para resolver un problema concreto, sino que pueda aprender en base a los estímulos de forma genérica, tal como lo haría un humano a través de los sentidos o las sensaciones como el dolor o el placer.

Con el objetivo de crear una inteligencia artificial que se asemeje lo más posible al nivel de generalidad y desempeño de la inteligencia humana, los investigadores de DeepMind diseñaron un algoritmo que combina dos de los métodos de aprendizaje automático más conocidos: el aprendizaje por refuerzo y el deep learning.

El aprendizaje por refuerzo se basa en que los agentes aprendan a poner en práctica estrategias que llevan a la mayor recompensa a largo plazo mediante un aprendizaje basado en el ensayo y error. De este modo, tras un entrenamiento el agente aprende a realizar tareas de la mejor forma posible en base a una memoria de los errores y éxitos de otros entrenamientos.

Por otro lado, los humanos también aprenden directamente de la información que obtienen, ya sea a través de la vista, el tacto etc. A través de las redes neuronales artificiales y del Deep Learning, los agentes también pueden imitar esta característica del aprendizaje humano, pudiendo aprender directamente de datos sin tratar y sin ayuda de ninguna heurística hecha a mano para cada caso. El Deep Learning trata de aprender de la representación de los datos usando estructuras lógicas similares a las redes neuronales con varias capas ocultas especializadas en detectar las diferentes características de los datos recibidos [9]. La combinación de estas dos aproximaciones, el *Deep Reinforcement Learning*, ha logrado la creación de agentes con un desempeño igual o mejor que el logrado por humanos en diferentes problemas y dominios complejos.

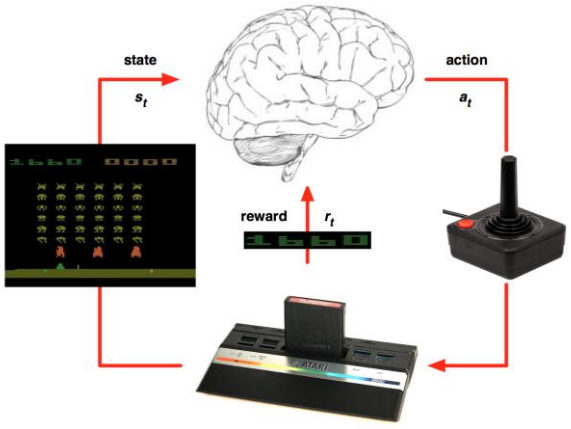

Figura 4: Esquema del modelo de actuación del algoritmo DQN (Extraída de [10])

A pesar de que la combinación de estas dos aproximaciones se había intentado antes no se obtuvieron resultados con éxito hasta 2013 debido a el aprendizaje inestable que

estos producían. Para solucionar este problema, el algoritmo de Deep Reinforcement Learning guarda todas las jugadas y estados previos del agente para después tomar muestras aleatorias de estas y re-entrenar al agente con estas para proporcionar datos de entrenamiento diversos y sin correlación entre ellos [8].

Esta variante del algoritmo se presentó en el artículo académico *Playing Atari with Deep Reinforcement Learning* [8]. Las primeras pruebas se realizaron tratando de enseñar a un agente a jugar a juegos de la consola Atari 2600 mediante la observación de los pixeles en pantalla y con una recompensa correspondiente a la puntuación obtenida en el juego. Estos agentes fueron entrenados sin ningún conocimiento previo sobre las reglas del juego. Como se muestra en la figura 5, tras las pruebas los resultados obtenidos mostraron que el algoritmo obtenía unos resultados similares o mejores a los conseguidos por un humano en más de 50 juegos.

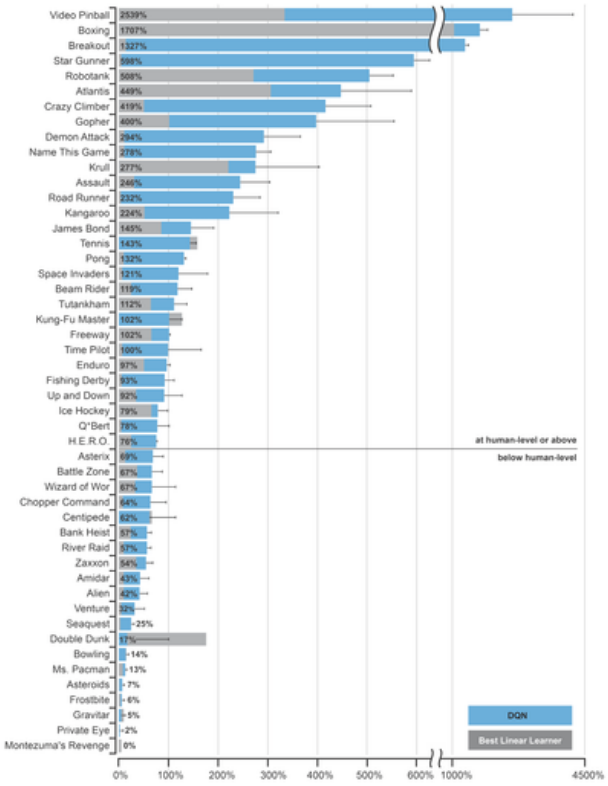

Figura 5: Resultados del algoritmo DQN en diferentes juegos (Extraída de [8])

Desde la publicación del algoritmo en el artículo se han creado diferentes versiones y mejoras del algoritmo, llegando a lograr crear agentes entrenados en juegos que han conseguido vencer a jugadores humanos expertos, demostrando así un considerable progreso en la resolución de muchos problemas y desafíos complejos.

El algoritmo parte de una base en la cual el agente trata de solucionar o cumplir una tarea interaccionando con un entorno mediante una serie de acciones, las cuales toma en base a unas observaciones y recompensas. En cada paso o instante de tiempo, el agente toma una de las acciones posibles, cambiando así el estado del entorno y la puntuación.

Se considera por tanto una serie de secuencia de acciones y observaciones que se asume que terminan en un número finito de pasos, creando así grandes modelos de decisión de Markov a los cuales se pueden aplicar métodos estándar de aprendizaje por refuerzo. El objetivo del agente es interactuar con el entorno de modo que maximice el valor de las recompensas futuras. Para ello, se asume que las recompensas futuras están descontadas por un factor de γ por paso, definiendo la recompensa futura descontada como

$$R_t = \sum_{t'=t}^{T} \gamma^{t'-t} \, r_{t'}$$

Siendo T el momento o paso en el que el juego o la tarea termina [8].

El algoritmo trata de elegir la acción con una mayor recompensa en base al estado actual. En el algoritmo Q Learning tradicional, esto se hace mediante la llamada función Q(s,a), la cual calcula el resultado esperado para un estado s y una acción a. Sin embargo, en el algoritmo DQN, la predicción que aproxima el valor esperado se obtiene mediante el uso de una red neuronal.

Esta red neuronal actúa como una caja negra, es decir, a la red neuronal se le proporcionan unos datos de entrada, los cuales pasan por varias capas ocultas y, mediante la detección de patrones en estos datos predice un resultado o salida.

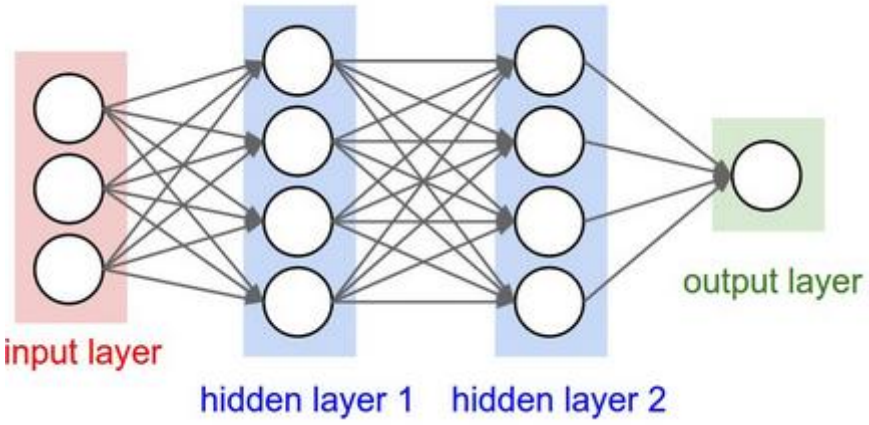

Figura 6: Estructura de una red neuronal artificial (Extraída de [10])

Para entrenar esta red neuronal es necesario darle pares de valores de entrada y de salida, es decir; a través de los datos de entrada sobre el estado del entorno en el que se encuentra el agente y la recompensa obtenida tras la acción la red neuronal se entrena para predecir el resultado en base al estado.

Para tener en cuenta no solo la recompensa directa sino también las futuras recompensas obtenidas por cada acción, es necesario que se optimice una fórmula que calcule el valor referido como *pérdida,* el cual representa la diferencia entre la predicción y el valor objetivo (objetivo = recompensa + factor de descuento * valor máximo obtenido por el estado siguiente) y la predicción obtenida del estado y la acción actual elevado al cuadrado:

$$perdida = (reward + \gamma * \max_{a`} \check{Q}(s, a`) - Q(s, a))^2$$

Al elevar al cuadrado esta función se penalizan más la diferencia entre estos valores y los valores negativos se tratan igual que los positivos. Esta función reduce la diferencia entre la predicción y el valor objetivo en base a la tasa de aprendizaje definida. La aproximación del valor predicho converge con el valor real obtenido mediante un proceso de actualización y aprendizaje (denominado Q-learning update), reduciendo así el valor de pérdida y aumentando los resultados obtenidos [10].

La principal diferencia del algoritmo DQN con los anteriores intentos de mezclar las técnicas de deep learning y de reinforcement learning es la existencia de un re-entrenamiento o *replay* que, como se ha explicado previamente, sirve para proporcionar datos de entrenamiento diversos y sin correlación entre ellos, proporcionando un aprendizaje estable. Con esta técnica las experiencias del agente en cada paso del tipo experiencia = (state, action ,reward, next_state) se guardan en un conjunto de datos o memoria D = e1,...,eN que contiene todas las experiencias del agente obtenidas a través de los diferentes episodios o partidas que va jugando [10].

Durante este proceso de *replay* se aplican las actualizaciones Q-learning usando muestras aleatorias de estas experiencias guardadas en memoria. Una vez realizado este re-entrenamiento, el agente selecciona las acciones a realizar siguiendo una política ε-greedy, es decir, hay una probabilidad de ε de que el agente tome una acción aleatoria y una probabilidad de 1- ε de que tome la acción que l red neuronal prediga que maximizará el valor de la recompensa total. Generalmente, el valor de ε es elevado al comienzo del algoritmo y se va reduciendo según el agente es entrenado, tomándose más acciones aleatorias en un principio y usando mayormente las predicciones de la red neuronal cuando el agente ya se ha entrenado considerablemente.

La base del algoritmo de DQN se presenta a continuación en la figura 7:

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

Figura 7: Algoritmo DQN por pasos (Extraída de [8])

Las ventajas que presenta este algoritmo con respecto al algoritmo estándar de online Q-Learning son varias. En primer lugar, al usar experiencias aleatorias para entrenar al

agente estas tienen probabilidad de usarse más de una vez, por lo que hay una mayor eficiencia en el uso de estos datos. Además, como se ha explicado anteriormente, al utilizar datos obtenidos de manera consecutiva se produce una correlación entre estos, lo cual produce una varianza mayor. Al usar experiencias elegidas de manera aleatoria se rompe esta correlación, solucionando el problema de la varianza [8].

3.3 Implementación del agente

Para implementar el agente usando el algoritmo de DQN se ha hecho uso de la librería para Python Keras, la cual permite la creación de redes neuronales fácilmente. Estas librería posee también funciones para obtener la predicción de las redes neuronales, para su entrenamiento etc. Cabe también recordar que el entorno y la misión del juego Pig Chase están ya creados y proporcionados por los organizadores del reto de Microsoft, por lo que para la implementación del agente solo se han creado una clase nueva que representa el agente y se ha variado el código del bucle del programa que controla el funcionamiento de los agentes y su interacción con el entorno en el programa principal que inicializa y evalúa los agentes.

Una vez aclarado esto, se pasa a explicar la clase agente creada

3.3.1 Inicialización de la red neuronal y del agente

El agente inteligente que se ha creado para este proyecto utilizando el algoritmo de Deep Q-Learning es relativamente sencillo. Con el uso de la librería Keras la implementación de la red neuronal es bastante rápida y los cálculos para actualizar y recalcular los pesos se simplifican. Los métodos principales que tiene el agente aparte del de inicialización son:

- act: Permite al agente decidir cuál será la siguiente acción a tomar, ya sea de forma aleatoria o en base a los resultados obtenidos por la red neuronal.
- _build_model: Crea la red neuronal del agente
- remember: Guarda una experiencia del agente con el estado, acción, recompensa obtenida y estado siguiente para el futuro replay
- replay: Entrena la red neuronal con experiencias previas del agente guardadas en memoria
- adapt_state: Los estados obtenidos directamente de Malmo dan varios arrays con la información concerniente al agente y lo que le rodea. Ya que para entrenar al agente no se necesita toda esta información, este método se encarga de adaptar y obtener la información necesaria de cada estado para entrenar a la red neuronal.

Para comenzar, al inicializar un agente se definen varios parámetros, tales como la tasa de descuento (gamma), de aprendizaje y de exploración (épsilon) junto a su mínimo y la tasa de descenso del épsilon. Se inicia así mismo la memoria del agente que guardará las diferentes experiencias mediante un contenedor de datos de tipo deque de Python, al cual al inicializarlo se define su tamaño. También se define el tamaño de los estados que se proporcionan al agente, ya que este número define el número de entradas que tendrá la red neuronal. Se define del mismo modo el número de acciones que tiene disponible el agente, ya que esto determina el número de salidas que tendrá la red neuronal.

Una vez definida esta información el método init llama al método _build_model para generar la red neuronal, creando una red de tipo secuencial con 6 entradas (x e y del propio agente, x e y del otro agente y finalmente x e y del cerdo), dos capas ocultas de 24 nodos y una capa de salida de tamaño 3 (moverse hacia delante, girar hacia la derecha, girar hacia la izquierda). Una vez hecho esto, el agente está listo para empezar a actuar

3.3.2 Método act y adapt_state

Como se ha explicado antes, el método act recibe el estado actual y devuelve la siguiente acción que ha decidido tomar. Sin embargo, los estados devueltos por Malmo devuelven más información, como puede observarse en la figura 8:

```
(array([
[u'grass', u'grass', u'grass', u'grass', u'grass', u'grass', u'grass', u'grass', u'grass'],
 [u'grass', u'sand', u'sand', u'sand', u'sand', u'sand', u'sand',u'sand', u'grass'],
[u'grass', u'sand', u'grass', u'grass', u'grass', u'grass/Agent_2',u'grass', u'sand', u'grass'],
[u'sand', u'sand', u'grass', u'sand', u'grass', u'sand', u'grass', u'sand', u'sand'],
 [u'sand', u'lapis_block', u'grass', u'grass', u'grass', u'grass', u'grass', u'lapis_block', u'sand'],
[u'sand', u'sand', u'grass/Agent_1', u'sand', u'grass', u'sand', u'grass', u'sand', u'sand'],
[u'grass', u'sand', u'grass', u'grass', u'grass', u'grass/Pig', u'grass', u'sand', u'grass'],
[u'grass', u'sand', u'sand', u'sand', u'sand', u'sand', u'sand', u'sand', u'grass'],
[u'grass', u'grass', u'grass', u'grass', u'grass', u'grass', u'grass', u'grass', u'grass']
], dtype=object),
[{u'name': u'Agent_1', u'yaw': 90.0, u'pitch': 30.0, u'y': 4.0, u 'x': 2.5, u'z': 4.5},
 {u'name': u'Pig', u'yaw': 0.0, u'pitch': 0.0, u'y': 4.0, u'x': 5.5, u'z': 5.5},
{u'name': u'Agent_2', u'yaw': -180.0, u'pitch': 29.53125, u'y': 4.0, u'x': 5.5, u'z': 1.5}])
```

Figura 8: Estados devueltos por el entorno Pig Chase

Como puede observarse, estos estados contienen información sobre el tipo de bloques que rodea al agente así como qué hay encima de cada uno a parte de la posición y ángulo de los dos agentes y del cerdo. La red neuronal creada con la librería Keras solo admite como entrada arrays de datos, y la aproximación elegida para este trabajo solo tiene en cuenta la x e y de los agentes y del cerdo. Sería también posible tener en cuenta más información para cada estado, pero para simplificar el aprendizaje del agente y que este se realice de forma más rápida, se ha elegido tener solo en cuenta estos 6 datos para los estados.

El método act, por tanto, antes de realizar cualquier decisión llama al método adapt_state, el cual devuelve un array de 6 elementos con los datos mencionados. Al igual que en la versión original del algoritmo de Deep Q-Learning explicado en el apartado anterior, el método act sigue una política ε-greedy habiendo una probabilidad de ε de que el agente tome una acción aleatoria y una probabilidad de 1- ε de que tome la acción que la red neuronal prediga que maximizará el valor de la recompensa total. Cuando se da este último caso, mediante el método ya definido para las redes

neuronales de Keras *predict()* se le proporciona el estado a la red neuronal y el método devuelve un array con 3 valores, uno por cada salida de la red neuronal. Finalmente el método devuelve el índice de la acción que ha predicho que proporcionará mejor resultado.

3.3.2 Método replay y remember

Como se ha expuesto en la explicación del algoritmo DQN, la principal diferencia de las aproximaciones anteriores que se habían hecho antes de este algoritmo es el método replay, mediante el cual entrena al agente con experiencias previas elegidas de forma aleatoria para evitar correlación entre estas. Para realizar esto con nuestro agente utilizamos el primer lugar el método remember, el cual es llamado después de ejecutar una acción. Este simplemente realiza un append de un dato tipo tuple a la memoria que se ha definido al iniciar el agente con los datos del estado anterior , la acción tomada, la recompensa, un booleano que indica si el juego ha acabado con esa última acción y finalmente, el nuevo estado.

El método replay es el más complejo del agente, ya que es este el que aplica el entrenamiento a la red neuronal tal y como se ha explicado en el algoritmo de DQN. Para tener en cuenta no solo las recompensas inmediatas de las acciones sino también las recompensas a largo plazo, se aplica la fórmula mostrada previamente:

$$loss = (reward + \gamma * \max_{a`} \check{Q}(next\_state, a`) - Q(state, a))^2$$

Mediante los siguientes pasos, el agente aprende a maximizar la recompensa futura basada en el estado dado.

En primer lugar, se eligen aleatoriamente X muestras (en este caso, 32) de las guardadas en memoria para entrenar al agente. Se comienza entonces un bucle que en cada iteración utiliza una de esas experiencias aleatorias elegidas de la siguiente forma:

1.- Si la experiencia con la que estamos entrenando la red neuronal es la última que se tomó en esa partida (done= True) se iguala el valor objetivo (Target) a la recompensa obtenida en esa experiencia:

$$Target = reward$$

En caso de que no sea la última acción de la partida, se iguala el valor objetivo al definido en la fórmula:

$$Target = (reward + \gamma * \max_{a`} \check{Q}(s, a`)))$$

2.- El valor de la recompensa de la acción tomada en esa experiencia se iguala a el valor objetivo o target:

$$\max_{a} Q(state, a)) = Target$$

3.- La librería Keras tiene de antemano la función fit(state,labels) que se encarga de realizar el reajuste de pesos de la red neuronal y de entrenar el modelo. Esta función recibe como datos más importantes los datos de entrada de la red, en este caso, el estado con el que se realiza la predicción, y las etiquetas de los valores esperados. En

este caso, esos valores son los obtenidos de la predicción *Q(state,a) junto* con el valor ajustado de la acción que se estima que maximiza la recompensa total ($\max_a Q(state, a) = Target$). Esta función se encarga de aplicar la fórmula completa con estos datos

$$loss = (reward + \gamma * \max_{a`} \check{Q}(next\_state, a`) - Q(state, a))^2$$

encargándose por tanto de sustraer el valor objetivo del valor de salida de la red neuronal y elevarlo al cuadrado, reduciendo así la diferencia entre la predicción y el valor objetivo. La función fit(), por tanto, actualiza los pesos de la red neuronal consiguiendo que esta aprenda.

4.- Por último, si el valor épsilon (el de la tasa de exploración) no está en su mínimo, se reduce multiplicándose por su tasa de descenso. Esto hace que cuanto más entrena el agente más tienda a predecir él mismo la acción a tomar en lugar de tomar una acción aleatoria.

Estas acciones se repiten con cada experiencia de las obtenidas para el entrenamiento, realizando el replay y entrenando la red neuronal. Todos los métodos explicados del agente se llaman en el bucle principal de actuación del agente que se explicará a continuación.

3.4 Implementación del algoritmo DQN

Una vez el agente está implementado y con todos los métodos necesarios definidos puede pasarse a su uso en el programa principal que proporciona el concurso para ejecutar el juego y los agentes. Como se ha explicado previamente, la misión, el escenario y el programa que pone en marcha el mini-juego y los agentes ya viene proporcionado por Microsoft para el concurso. Este código está escrito en Python y puede modificarse libremente para utilizar otros agentes a los definidos al ejemplo, modificar la misión o modificar el bucle que ejecuta cada agente. No se va a entrar en detalles de cómo interactúa el código de Python con el mod de Malmo para iniciar todo lo necesario y poner en marcha el juego, pero si es conveniente destacar que para las pruebas se utilizan dos agentes del mismo tipo (en este caso, del tipo creado específicamente para este proyecto con Deep Q-Learning), para lo cual lanza un hilo o proceso para cada agente.

Una vez los agentes que van a usarse para el juego han sido puestos en marcha, cada hilo ejecuta el código de un método definido llamado agent_loop(agent, environment) en el cual el agente obtiene información del entorno y decide cómo actuar. Este código ha sido modificado para que se ajuste al algoritmo de Deep Reinforcement Learning de la siguiente manera:

1.- Nada más comenzar la partida, el entorno se resetea (se inicia una partida de cero) y tanto el cerdo como os agentes son colocados en una posición aleatoria. A continuación se entra en un bucle que se ejecuta tantas veces como partidas se quiera que juegue el agente.

2.- Dentro del bucle, en cada iteración se comprueba si con la última acción tomada se ha terminado la partida. Si esta no ha acabado, se procede de la siguiente forma

- Pasando el estado actual al agente, se llama al método act que devuelve la acción a utilizar en este turno.
- Se ejecuta la acción elegida en el juego mediante el método ya definido de Malmo environment.do(action), obteniendo la información de cuál ha sido la recompensa, la información del nuevo estado y si la partida ha acabado o no
- Se llama al método del agnte remember, que guarda la experiencia con los datos del estado anterior, la acción tomada, la recompensa recibida, el siguiente estado y si ha sido la acción final de la partida
- Finalmente, se iguala el estado actual como el estado nuevo obtenido por la acción

3.- En caso de que la partida haya finalizado con la última acción tomada, se procede de la siguiente forma:

- Se comprueba si hay suficientes experiencias guardadas en memoria como para realizar el entrenamiento por replay (en nuestro caso, este número se ha definido como 32). En caso de que sea posible, se llama al método replay para entrenar la red neuronal.
- El entorno de juego se resetea (se comienza una nueva partida) y se suma +1 al contador de partidas jugadas.

Como se ha dicho, el agente juega tantas partidas como se hayan definido, reiniciando el ciclo y volviendo a tomar acciones cada unidad de tiempo hasta que se termina la partida y se ejecuta el replay, volviendo después a comenzar. En la figura 9 se muestra paso a paso las acciones que va tomando el agente según las condiciones:

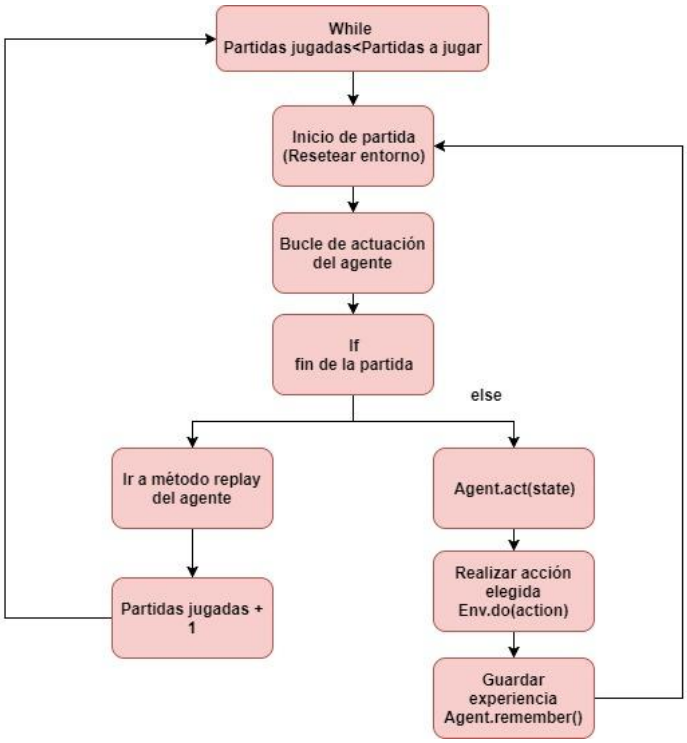

Figura 9: Diagrama de actuación del agente

Con el agente ya definido y el programa principal adaptado al algoritmo Deep Q-Learning, es posible pasar a realizar los experimentos y análisis que se exponen a continuación.

4. Entrenamiento, resultados y análisis del agente

4.1 Entrenamiento y resultados del agente

Para evaluar a los agentes en el concurso *The Malmo Collaborative Challenge* los agentes fueron evaluados obteniendo la media y el error estándar de la media en 100 partidas de los agentes presentados al concurso, estando estos ya entrenados con 100.000 y 500.000 partidas.

La idea inicial para el entrenamiento del agente creado para este proyecto era realizarlo con ese mismo número de repeticiones en la nube mediante el uso de una cuenta de Windows Azure. Sin embargo, por motivos ajenos al proyecto, el uso de esa cuenta no ha sido posible y por tanto el entrenamiento ha tenido que realizarse en un ordenador portátil de uso personal, limitando la posibilidad de realizar tantas partidas como se tenía previsto. El agente ha sido entrenado a su vez jugando junto con un agente del mismo tipo (el tipo de agente DQN creado para este trabajo) entrenado con el mismo número de iteraciones que su compañero en todas las pruebas.

Para realizar las pruebas y estudiar la progresión de los resultados conseguidos por el agente se han sacado datos del este entrenado con diferente número de iteraciones para ver su progresión. El máximo de partidas con las que se ha podido entrenar al agente ha sido con 50.000, por lo que los datos se han sacado cuando el agente estaba entrenado con 1.000 iteraciones, luego con 10.000 y luego de 10.000 en 10.000 hasta llegar al agente entrenado con 50.000 partidas.

El algoritmo DQN requiere de muchísimas repeticiones para aprender y dar buenos resultados, especialmente cuanto más complejo sea el problema al que se enfrenta. El hecho de que no hayan podido realizarse tantas repeticiones de entrenamiento como en un inicio se esperaba ha afectado a que los resultados no fueran tan buenos como cabría esperar, ya que el agente necesita de más repeticiones para entrenarse y obtener resultados más visibles. Sin embargo, con los resultados obtenidos si es posible obtener una muestra de que el algoritmo efectivamente aprende y de que va mejorando sus resultados con el tiempo, por lo que con un mayor número de repeticiones de entrenamiento podría llegar a obtener mejores resultados.

Como se sugiere en *Playing Atari with Deep Reinforcement Learning [8]* la evaluación de si el agente está aprendiendo o no es difícil, puesto que los resultados basados en la puntuación final media de cada juego pueden variar considerablemente de un agente entrenado con un número de iteraciones a otro debido a que pequeños cambios en los pesos de la red neuronal pueden llevar a grandes cambios en las políticas de elección del agente. Además, como se ha dicho previamente el algoritmo de DQN requiere de miles de iteraciones para comenzar a tener resultados visibles, por lo cual, con las limitaciones que se han tenido para realizar el entrenamiento ha sido difícil evaluar.

Para comprobar si el agente efectivamente está aprendiendo independientemente de la media de puntuación obtenida, se ha obtenido también la media de el valor máximo de la función Q del algoritmo, la cual "proporciona una estimación de cuál es la recompensa esperada que un agente puede obtener siguiendo su estrategia o política desde cualquier estado dado" [8].

A su vez, se han obtenido cuántas veces el agente decide no cooperar (el agente va a la casilla azul para una recompensa individual) y cuántas veces los agentes cooperan para obtener una recompensa conjunta. Estas pruebas se han realizado con el agente entrenado con diferentes números de partidas jugadas previamente, jugando junto a otro agente entrenado con el mismo número de iteraciones a lo largo de 100 partidas

A continuación, se va a pasar a mostrar y analizar los resultados obtenidos.

4.1.1 Resultados obtenidos

Como se ha dicho previamente, las se han obtenido poniendo a jugar al agente entrenado con diferente número de iteraciones durante 100 partidas. De estas 100 partidas, se ha obtenido la puntuación media por paso/acción que toma el agente, el error estándar de esta media, la media del valor máximo de la función Q o valor estimado de la recompensa y finalmente el número de veces que a lo largo de esas 100 partidas el agente ha optado por cooperar o por una recompensa individual, si es que ha conseguido llegar a una de esas dos soluciones. Los datos obtenidos han sido los siguientes:

Nº repeticiones entrenamiento	**Recompensa media por acción**	**Error estándar de la recompensa media por acción**	**Valor medio función Q**	**Recompensas por cooperación**	**Recompensas individuales**
1.000	-0.946	0.925	-12.93	3	7
10.000	-0.951	0.855	-8.86	4	3
20.000	-0.947	1.072	-8.08	8	0
30.000	-0.874	2.067	-7.68	4	13
40.000	-0.857	1.921	-6.89	6	16
50.000	-0.797	1.672	-5.46	9	26

Figura 10: Tabla de resultados del agente

Estos resultados obtenidos de las pruebas serán analizados a continuación para ofrecer una mejor perspectiva del desempeño y aprendizaje del agente

4.1.2 Análisis de los resultados

Dado que las pruebas ofrecen distintos resultados y que estos ofrecen información sobre diferentes factores del aprendizaje del agente, se va a proceder a analizar los datos por separado para ofrecer una mejor comprensión de estos:

En primer lugar, se van a analizar los datos de la recompensa media por acción. Estos ofrecen información sobre los resultados que obtiene el agente en el juego. Hace falta aclarar que esta no es la puntuación media por partida, sino la puntuación media por

acción que toma el agente. Teniendo en cuenta que en el juego Pig Chase el agente recibe una recompensa de -1 puntos por realizar cualquier acción y solo recibe una recompensa positiva si llega a la casilla de salida (recompensa individual) o si captura al cerdo (recompensa cooperativa) esta puntuación, por defecto, tiende a ser baja y a acercarse al valor -1.

En cuanto a los resultados obtenidos sobre esta puntuación, se puede observar que el agente tiene una puntuación media similar cuando está entrenado con 1.000, 10.000 y 20.000 partidas. Las puntuaciones obtenidas para estas tres pruebas están entre -0.94 y -0.95, mostrando que en las primeras 20.000 partidas de entrenamiento no se consigue una mejoría visible. Sin embargo, en las pruebas con 30.000, 40.000 y 50.000 partidas de entrenamiento si se percibe una mejoría en la puntuación media pasando de -0.94 en las pruebas con 20.000 iteraciones a -0.78 en las pruebas con 50.000 iteraciones.

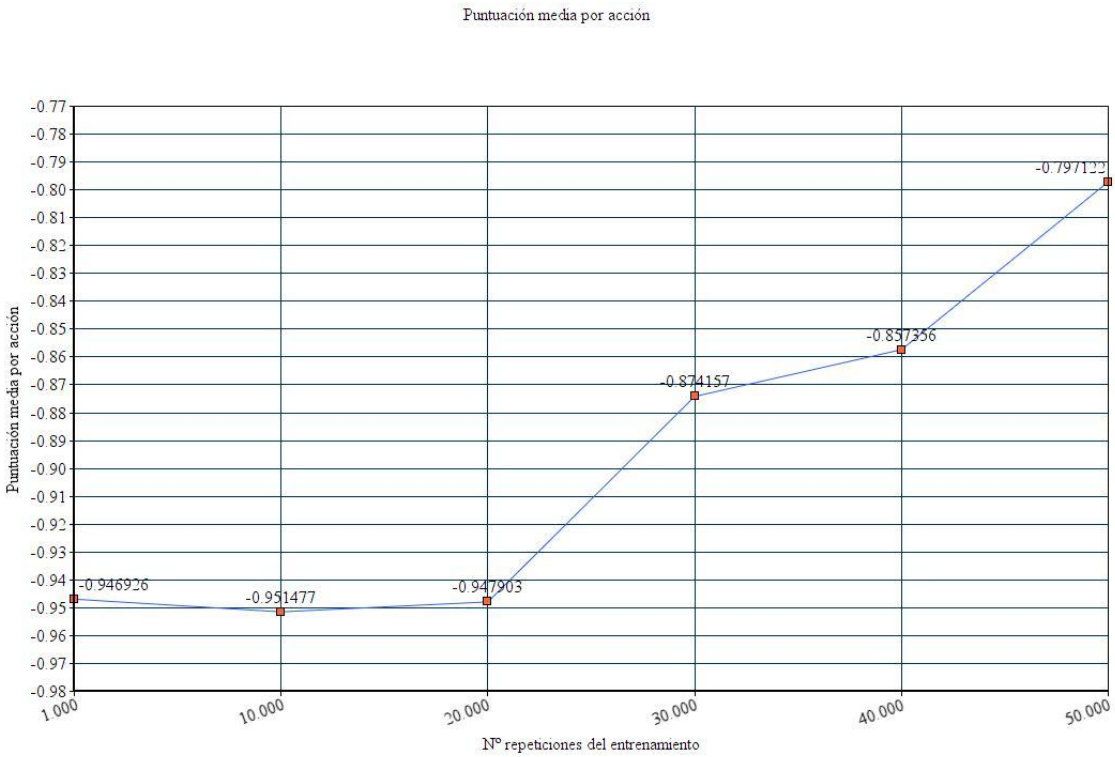

Figura 11: Media de puntuación por paso del agente

A pesar de que estas no sean puntuaciones especialmente buenas, si puede apreciarse que con el entrenamiento de la red neuronal los resultados van mejorando según el agente toma acciones que le llevan a obtener recompensas más a menudo. Cabe también destacar que es posible que a lo largo del entrenamiento estas puntuaciones empeoren y mejoren, ya que como se ha dicho previamente los cambios en los pesos de la red neuronal pueden llevar a grandes cambios en las políticas de actuación del agente.

El valor obtenido por la función Q nos indica la recompensa que espera el agente espera obtener siguiendo su estrategia o política desde cualquier estado dado. A pesar de que los valores obtenidos de la media de la puntuación por paso no han ido mejorando de forma uniforme, el valor máximo de la función Q del algoritmo si mejora de forma uniforme conforme se va incrementando el número de partidas que ha jugado. Esto significa que el algoritmo según aprende sí cree mejorar los resultados o, al menos, espera obtener una recompensa mayor siguiendo esa política. Esto da una muestra de que el agente efectivamente está aprendiendo y mejorando con el entrenamiento puesto que el resultado esperado es mejor y aumenta de forma más regular que la puntuación media por acción.

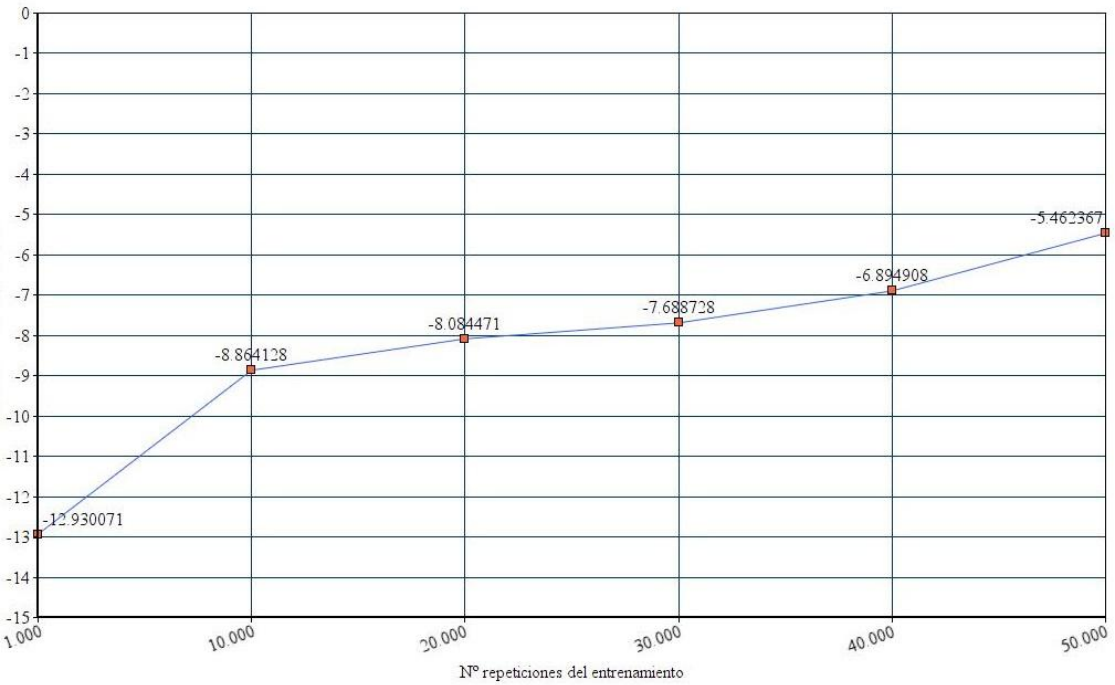

Figura 12: Valor Q máximo medio del agente

Finalmente, si se observan los datos con respecto al número de veces que, en esas 100 partidas de prueba el agente ha llegado a una solución individual o cooperativa se pueden sacar diversas conclusiones.

Cuando el agente está entrenado con 1.000 partidas este tiene unos resultados similares a los de un agente de tipo aleatorio puesto que la mayoría de las veces este no llega a ninguna solución, ni cooperativa ni individual, por lo que las puntuaciones que obtiene no son demasiado buenas. Con 10.000 y 20.000 partidas de entrenamiento puede observarse que el número de recompensas individuales desciende mientras que el número de recompensas cooperativas aumenta. La puntuación media por paso sigue siendo similar en estos primeros tres casos, pero cabe destacar que, al realizar las pruebas con el agente entrenado con 20.000 iteraciones se observó que las recompensas individuales habían descendido a cero. Al ser esto un tanto extraño, ya que es mucho más fácil que incluso de forma aleatoria se obtengan más recompensas individuales que cooperativas, se realizaron varias veces las pruebas con el agente entrenado con 20.000 repeticiones y en todos los casos el número de recompensas

individuales fueron muy bajas o incluso nulas. Esto puede dar a entender que durante su entrenamiento, la tendencia aprendida en torno a las 20.000 repeticiones de entrenamiento llevaba al agente a buscar recompensas cooperativas por encima de buscar una recompensa individual. Sin embargo, como puede observarse, esta política de actuación no daba buenos resultados.

A partir de las 30.000 repeticiones comienza a observarse una mejoría en la puntuación obtenida ya que el agente, aunque reduce el número de recompensas cooperativas con respecto a las 20.000 repeticiones, si aumenta en gran medida el número de recompensas individuales, lo cual indica que el agente comienza a encontrar una solución para el juego, ya sea individual o cooperativa, con mayor frecuencia.

Con respecto a el agente 40.000 y 50.000 repeticiones, puede observarse que el número de recompensas obtenidas de forma cooperativa aumenta levemente, mientras que el número de recompensas individuales aumenta de forma mucho más visible. Estos datos indican que según el agente aprende logra llegar a una solución un mayor número de veces, llegando a encontrar con 50.000 repeticiones una solución un 35% de las veces.

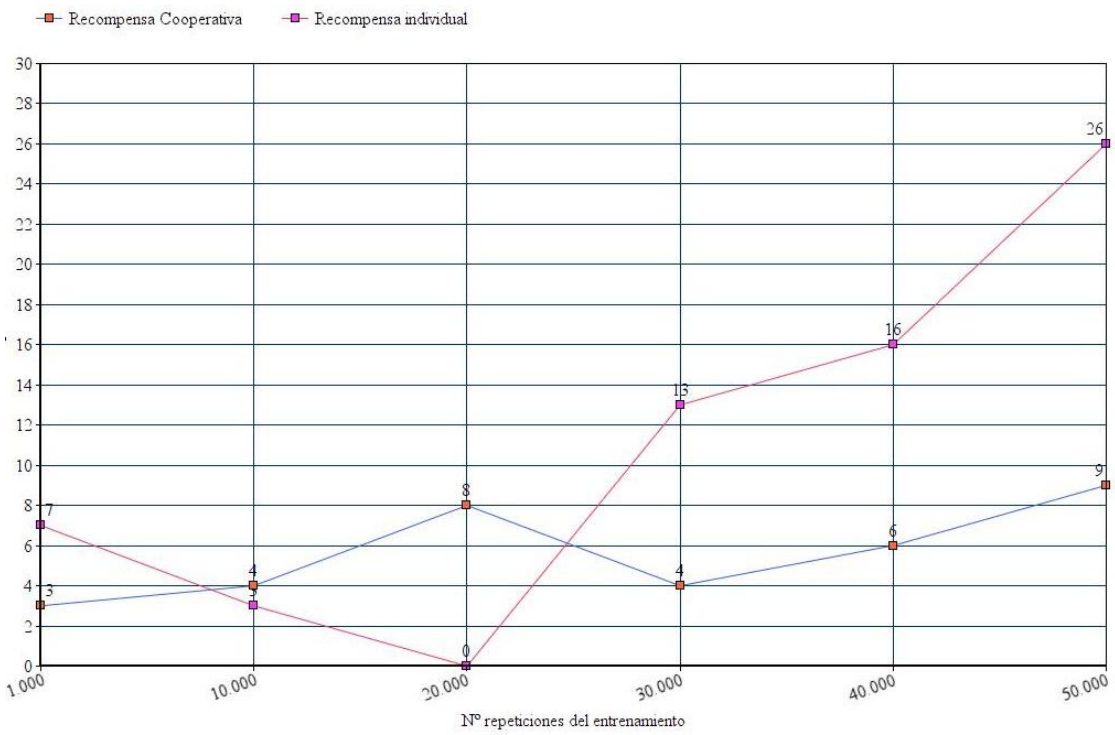

Figura 13: Soluciones individuales y cooperativas obtenidas por el agente

De estos datos también se puede concluir que el agente tiende a hallar una solución individual mucho más que una solución cooperativa. Esto podría deberse a que las políticas de actuación cooperativas son mucho más complejas y además no depende solo del agente sino también de su compañero, por lo que aunque se intentaran más veces llegar a una solución cooperativa esta no tendría éxito muchas de las veces. Esto puede hacer que el agente, por tanto, tienda a buscar la recompensa segura en lugar de optar por cooperar. Sin embargo, a pesar de esto el número de recompensas cooperativas obtenidas sí que ha mejorado según ha avanzado el entrenamiento del

agente, por lo que puede deducirse que el agente sí está aprendiendo a colaborar en cierto modo.

Aprender a realizar mejores políticas de actuación cooperativas podría requerir de un mayor número de iteraciones de entrenamiento o incluso de una red neuronal con más nodos ocultos. Como se sugiere en *Multi-agent Reinforcement Learning in Sequential Social Dilemas [11]* un mayor número de nodos en una red neuronal lleva a una mejor aprendizaje de comportamientos complejos, como el cooperativo en este caso. Un mayor número de nodos en la red neuronal podría traducirse como una mayor "capacidad cognitiva" de la red.

Un elemento a tener en cuenta de los resultados es que el agente ha sido entrenado mientras jugaba contra otro agente también en entrenamiento y que cambiaba sus políticas de actuación con el tiempo. Si este hubiera sido entrenado con un agente estable que tiene u tipo de comportamiento definido y no variable, es posible que los resultados hubieran sido distintos.

4.2 Resultados y análisis del Malmo Collaborative AI Challenge

El concurso colaborativo de Microsoft y la plataforma Malmo finalizó el 5 de junio de 2017 y en él participaron más de 80 equipos formados por alumnos de posgrado de diferentes países, presentando soluciones para el problema muy diversas, tales como métodos de planificación, métodos evolutivos, modelos de decisión de Markov, heurísticas o Deep Reinforcement Learning, por ejemplo.

Los resultados obtenidos por las soluciones propuestas han sido también muy variados, pero a pesar de que algunas propuestas han tenido visiblemente mejores puntuaciones que otras, entre las mejores no ha habido una aproximación que destaque por encima de las demás a pesar de ser soluciones muy diferentes entre ellas.

Con el fin de poder hacer un mejor análisis de los resultados y poder obtener conclusiones y comparativas con respecto a la solución propuesta para este trabajo, se va a proceder a exponer brevemente las mejores soluciones de los participantes del concurso así como sus resultados

4.2.1 Equipo HogRider

Este equipo ha creado una solución a medida para el problema centrándose en el mejor desempeño del agente mediante un sistema de decisión específico para el juego en el que el agente decide en base a diversos factores para saber cómo actuar. Estos factores son:

- Tipo de agente contra el que está jugando: Con el objetivo de hacer que el agente funcionara bien con los dos tipos de agente simple proporcionados para el concurso por Malmo. Es posible por tanto que el agente con el que esté jugando sea uno aleatorio (toma decisiones de forma aleatoria) o de que sea un agente centrado (trata de ir mediante el camino más rápido a la posición del cerdo). Ya que esta información no puede obtenerla del entorno, el agente comienza con una probabilidad del 75%-25% de determinar si es un agente u otro y va adaptando estas probabilidades mediante una red bayesiana de probabilidad, aprendiendo así a detectar el tipo de agente de forma más exacta.

- La posición del cerdo en el juego: El agente, según la posición del cerdo en el mapa, determina si es posible atraparlo (Posición A), si es posible que lo atrape un agente en solitario (Posición B) o si es posible atraparlo entre dos agentes (Resto de posiciones). Dependiendo de esta posición, el número de pasos restantes que es posible dar y del tipo de agente con el que está jugando, el agente decidirá si ir hacia la salida para obtener una recompensa individual, ir a por el cerdo o esperar a que el cerdo vaya a otra posición.

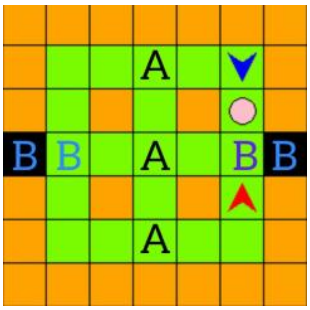

Figura 14: Posiciones según la posibilidad de atrapar al cerdo (Extraída de [7])

- El numero de pasos que falta para terminar el juego.

Puntuaciones obtenidas:

Media por paso con 100k partidas: 2.411 puntos

Media por paso con 500k partidas: 1.894 puntos

4.2.2 Equipo Bacon Gulch

La aproximación tomada por este equipo se basa en el tipo de agente contra el que está jugando (aleatorio o centrado). En base a esta creencia, el agente decide si ir hacia el cerdo y cooperar en caso de que crea que el otro agente es de tipo centrado, o si cree que el agente es aleatorio decidirá entonces ir a la salida más cercana. Para determinar el tipo de agente contra el que se está jugando utiliza una red de inferencia bayesiana, y usa el algoritmo Monte Carlo Tree Search (un algoritmo heurístico de búsqueda) usado como algoritmo de planificación para elegir la mejor acción posible en base a la creencia sobre el tipo de agente contra el que se está jugando. Este equipo se ha centrado principalmente en la eficiencia del código para tomar las decisiones lo más rápido posible y así evitar que los datos del entorno varíen mientras se toman las acciones decididas (ya que el cerdo no se mueve a la misma velocidad de los agentes, por lo que cuanto más rápido tomen estos las decisiones y las acciones, menos cambiará la posición del cerdo)

Puntuaciones obtenidas:

Media por paso con 100k partidas: 1.894 puntos

Media por paso con 500k partidas: 1.681 puntos

4.2.3 Equipo Village People

La aproximación tomada por este equipo es la más similar a la propuesta por este trabajo ya que su objetivo principal ha sido construir un modelo capaz de aprender políticas de actuación complejas y de determinar cómo se comportan otros agentes en lugar de un modelos con heurísticas hechas a medida. Para ello este equipo se ha valido de métodos de Deep Reinforcement Learning, usando finalmente una red neuronal convolucional de cuatro capas, la cual es un tipo de red utilizada generalmente para tareas de visión artificial. El algoritmo finalmente usado de Deep Reinforcement Learning es el conocido como Asynchronous Advantage Actor Critic (A3C), otro algoritmo creado por DeepMind más complejo que DQN pero con mejores resultados por lo general.

"Este algoritmo crea agente asíncronos similares a los implementados en el algoritmo DQN que ejecutan acciones en entornos diferentes para después actualizar la red neuronal principal cada cierto número de pasos. Estos agentes también sincronizan sus pesos con el del agente maestro después de cada actualización de pesos de la red neuronal principal." [12]

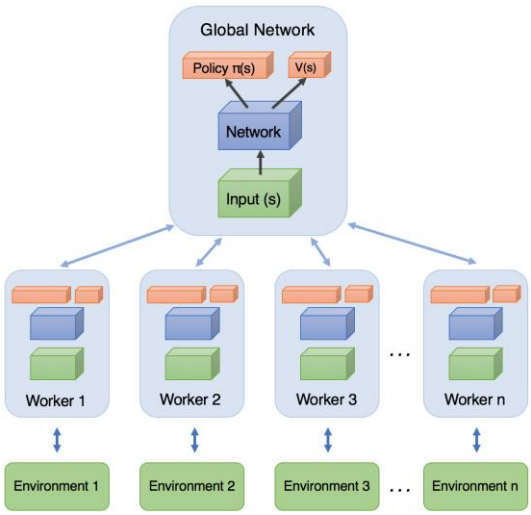

Figura 15: Diagrama de actuación del algoritmo A3C. (Extraída de [12])

Aplicando este algoritmo, el equipo de Village People ha entrenado al agente mediante otros sub-agentes que jugaban en entornos diferentes de forma independiente, proporcionando un aprendizaje más rápido y datos con menor correlación. También, para ayudar al aprendizaje de la tarea principal, se entrenó también al agente para predecir la recompensa inmediata del estado siguiente al actual y a predecir cómo será el estado del entorno en los pasos siguientes (es decir, las posiciones siguientes del cerdo y del otro agente).

Puntuaciones obtenidas:

Media por paso con 100k partidas: 1.618 puntos

Media por paso con 500k partidas: 1.318 puntos

4.2.4 Equipo The Danish Puppeteers

Al igual que otros equipos anteriores, The Danish Puppeteers han basado su solución en un agente que decide sus acciones en función de si cree que el agente con el que está jugando va a cooperar o no. El agente tiene dos modos de decidir si el otro agente va a cooperar o no:

- Si el otro agente se mueve hacia el cerdo un 60% de las veces, decide entonces que va a cooperar. Si se da el caso contrario, el agente simplemente decide ir hacia la salida más cercana
- El agente decide si el otro jugador va a cooperar mediante un modelo de Markov oculto, cuyos datos de observación son el color del casco del otro jugador, el signo de la distancia hacia el cerdo (si es negativa, positiva o cero) y el signo de la distancia a la salida más cercana. Si cree que el otro agente no va a cooperar, se dirige hacia la salida.

Ambas versiones del agente, una vez han decidido si cooperar o no, utiliza una variación del algoritmo de búsqueda A-Star hecha a medida para el juego Pig-Chase para decidir qué acciones debe tomar ya sea para ir a la salida más cercana o para cooperar y capturar al cerdo.

Media por paso con 100k partidas: 2.189 puntos

Media por paso con 500k partidas: 2.435 puntos

Finalmente, en la figura 16 se pueden observar los resultados de los diferentes agentes del concurso comparados. Ya que el agente creado para este proyecto no ha sido entrenado con las mismas repeticiones, no se ha considerado para esta comparativa. Como puede observarse, todos los resultados están bastante igualados.

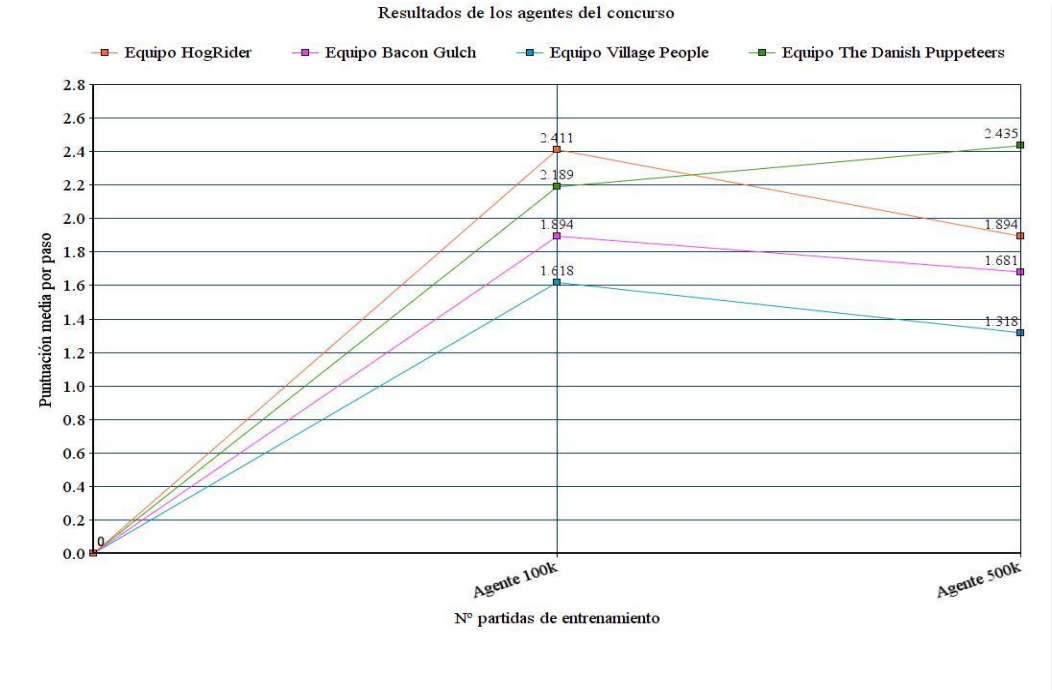

Figura 16: Resultados de los agentes concursantes en la competición

4.2.5 Conclusiones y evaluación de los agentes del concurso.

Tras observar las mejores propuestas para el concurso se han podido obtener diversas conclusiones que aportan un mayor entendimiento de la cooperación entre agentes inteligentes y que pueden ayudar a mejorar el presente trabajo en un futuro.

En primer lugar, puede observarse que las mejores soluciones o, las que han obtenido mayores puntuaciones han sido las que han creado un sistema de decisión específico para el juego. Si bien es cierto que estos agentes pueden determinar si el agente con el que están jugando va a cooperar o no con bastante fiabilidad, estos sólo están entrenados para jugar contra los dos tipos de agentes proporcionados por defecto para el concurso (el agente aleatorio y el agente centrado). En caso de que estos jugaran con otro tipo de agente que actuara o decidiera de una manera distinta, tendría que estudiarse si estos agentes siguen obteniendo buenos resultados o si tendrían que modificarse para adaptarse a los nuevos tipos de agente.

En segundo lugar, como se ha comentado previamente los agentes ganadores por lo general aprenden a identificar si el agente contra el que juegan va a cooperar o no. Sin embargo, estas decisiones se toman en base a información que el algoritmo de decisión "sabe" que es importante por parte de la experiencia humana introducida en el algoritmo. Esto significa que la información utilizada para decidir si cooperar es información que de antemano se sabe que es relevante, como la posición del cerdo o el color del casco del otro agente. Esta división entre la información relevante y la irrelevante ha sido hecha por humanos, por lo que aunque los agentes aprendan a diferenciar la cooperación en el otro agente estas soluciones están en gran parte informadas por conocimiento humano externo, haciendo que sean soluciones a medida y no genéricas.

Otro punto a tener en cuenta es que una vez han decidido si van a tratar de lograr una solución cooperativa o individual para el problema, las acciones que toman para ir a la salida o capturar al cerdo se basan en heurísticas o variaciones de algoritmos hechos a medida para el problema el concreto, por lo que aunque las acciones tomadas de una partida a otra varíen y sepan desenvolverse correctamente en el entorno de juego, estos algoritmos y heurísticas están informados en gran medida por la experiencia humana. Esto hace que los agentes "sepan" de antemano que las dos posibles soluciones para el problema son atrapar al cerdo o ir a las casillas de salida, por lo que lo único que tienen que hacer una vez hayan decidido por qué solución van a optar es usar un algoritmo que les permita llegar a su objetivo de una forma eficiente. Los algoritmos de *Deep Reinforcement Learning*, por su contrario, pretenden que el agente aprenda a resolver el problema con tan sólo la información que recibe del entorno y las recompensas o penalizaciones obtenidas, por lo que en este caso tendría que aprender no solo sobre si el agente contra el que juega va a colaborar o no sino también a llegar a las casillas de salida o a colaborar para atrapar al cerdo por su cuenta de una forma eficiente.

El único representante de los algoritmos de *Deep Reinforcement Learning* entre las mejores soluciones de los concursantes es el agente del equipo Village People. Esta es la solución más similar o, al menos, de la que más se podría aprender para mejorar el agente realizado para este trabajo. Como se ha explicado en el apartado dedicado a este

equipo, el agente utiliza el algoritmo *Asynchronous Advantage Actor Critic*, un algoritmo similar a DQN pero más complejo pero con mejor desempeño. Esta solución es la más genérica de las obtenidas en el concurso puesto que al agente aprende por su cuenta políticas complejas de actuación y decisión para el problema sin información previa proveniente de la experiencia humana, por lo que resulta la más interesante en cuanto a la investigación del aprendizaje y la cooperación multiagente ya que es la que más se acerca a esa inteligencia "general" que la investigación en la inteligencia artificial busca. Esta solución podría adaptarse fácilmente a otros problemas y demuestra que usando algoritmos de *Deep Reinforcement Learning* los agentes pueden no sólo aprender a solucionar un problema sino que también pueden detectar la cooperación de otros agentes y aprender a trabajar en conjunto para obtener mejores soluciones.

Por último, cabe destacar que las diferentes soluciones propuestas para el concurso combinan por lo general diversos algoritmos y heurísticas diferentes para obtener mejores resultados, y que además para obtener buenos resultados no sólo importa la técnica utilizada sino también otros factores como la eficiencia.

5. Conclusiones

Este trabajo no ha tenido solo en cuenta la creación de un agente inteligente que aprendiera a jugar al juego propuesto para el concurso The Malmo Collaborative AI Challenge, sino que también ha tenido en cuenta el desarrollo de otros agentes para el concurso y otros estudios hechos con anterioridad con el fin de aportar un poco más de luz al estudio de la cooperación entre agentes inteligentes, por lo que de este pueden sacarse diversas conclusiones.

En primer lugar y con respecto al agente creado, la motivación e idea principal para su creación era programar un agente que aprendiera a colaborar con otro agente con el cual no tiene comunicación directa solo mediante la información y recompensas que obtiene del entorno. Este enfoque se decidió tras observar su carácter genérico y los buenos resultados de otros experimentos en los que un agente DQN aprendía a jugar a juegos solo en base a la información que podía obtener del entorno. Con la intención de comprobar los resultados que podría obtener este algoritmo en un entorno en el que la cooperación es necesaria para obtener mejores puntuaciones, se pretendía también observar el tipo del comportamiento al que el agente tiende tras muchas partidas de entrenamiento y si, efectivamente aprende a cooperar.

Tomando también como referencia otros estudios en los que se han usado agentes DQN para tareas competitivas, el desempeño del algoritmo DQN en políticas de actuación más complejas como las que requieren la detección y ejecución de la cooperación se consideraron interesantes como caso de estudio.

En cuanto a los resultados obtenidos, si bien es cierto que estos no han sido tan buenos como se esperaban y especialmente en comparación con los de los otros agentes del concurso, si es posible concluir que el agente creado efectivamente aprende y mejora sus resultados con el entrenamiento. También cabe decir que, en cuanto a su comportamiento con el nivel de repeticiones con los que se han realizado las pruebas, tiende más a recurrir a una recompensa individual que a una cooperativa. Sin embargo debe resaltarse que a pesar de esto, si aumenta el número de veces que consigue colaborar para obtener una recompensa.

Un factor a tener en cuenta es que estas las comparaciones con otros agentes son en cierto modo relativas, puesto que como se ha explicado previamente, por problemas ajenos al trabajo no ha sido posible entrenar al agente con tantas repeticiones como se entrenó a los otros agentes del concurso. Sin embargo, con la información y conocimientos obtenidos sobre la cooperación multiagente y la IA durante el desarrollo de este proyecto podrían proponerse futuras mejoras para este mismo.

Algunas de las modificaciones que podrían realizarse para mejorar los resultados del agente son las siguientes:

- **Modificación de la red neuronal utilizada en el algoritmo:** Como se da a entender en el estudio de *Multi-agent Reinforcement Learning in Sequential Social Dilemmas*, el número de nodos y de capas ocultas de una red neuronal afecta al comportamiento del agente y a sus predicciones. Un mayor número de

nodos en una red neuronal ayuda al mejor aprendizaje por parte del agente de tareas o políticas complejas [11]. En el caso de Pig Chase, la cooperación es de por sí mucho más compleja que la solución individual, por lo que es más fácil que el agente tienda a esta última. Como se muestra en los experimentos hechos por DeepMind con el juego Wolfpack [11], una red neuronal mayor hace que el agente aprenda mejor las políticas complejas, en este caso, la cooperativa. Se puede observar por tanto cómo desciende la búsqueda de la solución individual cuando la red neuronal usada tiene más nodos

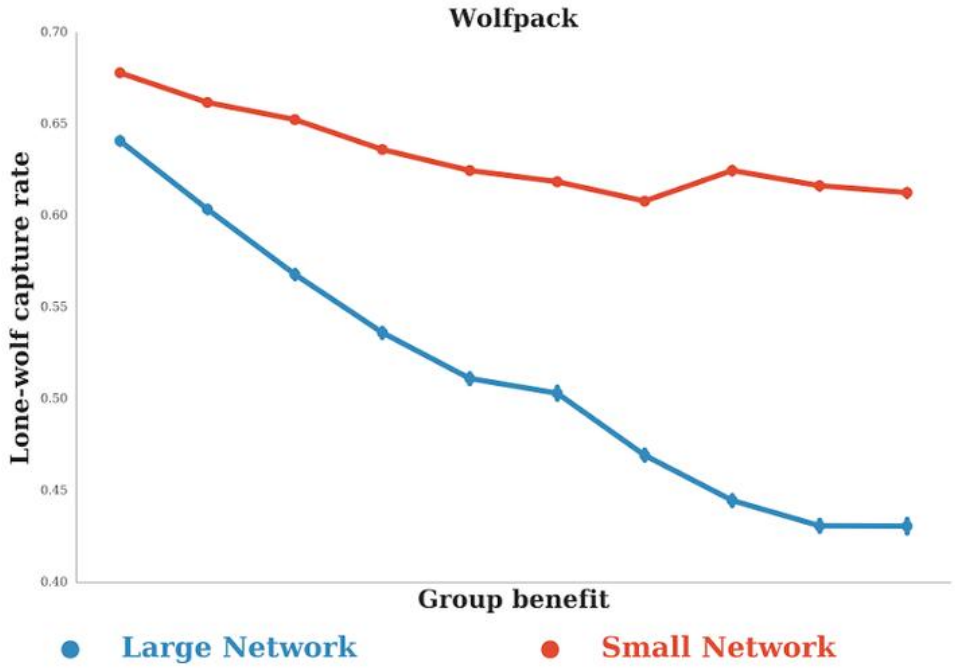

Figura 17: Resultados individuales vs. cooperativos de un agente DQN según el tamaño de su red neuronal. (Extraída de [11])

También podría ser beneficioso para el aprendizaje del agente utilizar otro tipo de red neuronal. Para este proyecto se utilizó una red neuronal secuencial. Sin embargo, el equipo Village People (equipo que utilizó el algoritmo A3C) utilizó una red neuronal convolucional, un tipo de red usada generalmente para tareas de visión artificial. Este tipo de red tiene un comportamiento que se asemeja a las neuronas de la corteza visual de un cerebro biológico. Como puede observarse por los resultados obtenidos por el equipo, su aproximación ha resultado más exitosa que la propuesta ya que, entre otras cosas, la red neuronal utilizada aprendía mejor políticas complejas. A pesar de que estos resultados hayan sido obtenidos de otro algoritmo, cambiar el tipo de red neuronal de una secuencial a una convolucional sin cambiar el tipo de algoritmo utilizado podría llevar al algoritmo DQN desarrollado para este proyecto a un mejor aprendizaje y , por tanto, a mejores resultados.

- **Utilización del algoritmo de Deep Reinforcement Learning A3C:** Como se ha dicho previamente, el equipo Village People es el que propuso la aproximación más similar y por lo tanto más interesante para este trabajo. El algoritmo A3C es un algoritmo de la familia DQN más complejo, pero que en este caso ha dado uno de los mejores resultados. Este requeriría para el caso de Pig Chase y Malmo, sin embargo, poner en marcha múltiples sesiones de

Minecraft a la vez para realizar los entrenamientos, lo cual, para un ordenador personal es costoso. A pesar de esto, este ha demostrado su buen funcionamiento, y para futuras mejoras de este trabajo o incluso para otros problemas de IA este algoritmo es una aproximación muy a tener en cuenta debido a su potencia y mejores resultados que DQN

- **Aportar información simplificada al agente para su entrenamiento:** La información con la que se entrenaba el agente ha sido, como se ha explicado previamente, las posiciones del propio agente, del cerdo y del otro agente. Sin embargo, estas posiciones se entregaban sin tratar, y es posible que redondear estas a números enteros en lugar a la posición exacta con decimales pudiera ayudar al agente a aprender más rápidamente, puesto que estas variaciones en el juego real no tienen importancia y complican más la información dada al agente para su aprendizaje. Otra de las propuestas en las que se pensó en un inicio fue dar una posición relativa del cerdo y del otro agente en lugar de dar la información exacta, indicando si estos están o no al norte, sur, este u oeste. Se trata de mera especulación, pero es posible que la simplificación de los datos de entrada a la red neuronal pudiera agilizar el proceso de aprendizaje.
- **Combinar el algoritmo DQN con heurísticas y algoritmos adaptados para el problema específico:** Como se ha visto en los otros participantes del concurso, la mayoría de las soluciones propuestas se han basado principalmente en la decisión de si el agente creía mejor intentar cooperar o no. Una vez tomada esta decisión, la solución para encontrar el camino a la salida o atrapar al cerdo se hacía mediante algoritmos de búsqueda o heurísticas. Si se quisiera mejorar la puntuación de nuestro agente, una propuesta que combinara DQN con estos algoritmos para tomar una decisión y ejecutarla podría mejorar los resultados de la puntuación obtenidos. Sin embargo, dado que uno de los objetivos principales del trabajo era crear una agente que aprendiera a jugar con la menor información específica sobre el juego posible, se decidió por la aproximación del algoritmo DQN.

A pesar de que estas propuestas de modificaciones están enfocadas para el trabajo propuesto, también pueden servir de referencia para otras aplicaciones del algoritmo DQN u otros algoritmos de Deep Reinforcement Learning para otros problemas o juegos distintos, puesto que el carácter más genérico para solucionar problemas de esta familia de algoritmos lo permite.

Es también posible decir tras observar los resultados del trabajo y de otros agentes que el algoritmo DQN, aunque sí que proporciona mejoras y con entrenamiento aprende a solucionar el problema, requiere un mayor refinamiento para las tareas más complejas como son las tareas cooperativas para dar mejores resultados. Sin embargo, tras este trabajo también es posible concluir que los algoritmos de *Deep Reinforcement Learning* no están limitados solamente a tareas u objetivos individuales, puesto que con diferentes enfoques y entrenamiento es capaz de aprender a realizar tareas cooperativas sin una información en su algoritmo proveniente de la experiencia humana que las impulse a ello. Esto motiva a continuar con la investigación y desarrollo de este tipo de algoritmos puesto que se acercan a ese tipo de inteligencia "genérica" que busca la investigación de la IA, capaz de aprender en base a la información que recibe del exterior y de las recompensas o penalizaciones que recibe

por interactuar con el entorno que le rodea en lugar de necesitar una aproximación diferente y específica para solucionar cada nuevo problema que se plantea.

En cuanto a la cooperación entre agentes, el principal objetivo del concurso y en parte del trabajo era aportar un poco más de información en este tema. Como se ha observado, hay muchas aproximaciones diferentes para crear agentes que puedan elegir o no cooperar con otros en base a la información del entorno y al entrenamiento. Sin embargo, a pesar de todas las soluciones diferentes propuestas aún es difícil responder a preguntas sobre la cooperación entre agentes, como cuál es el modo de hacer que dos agentes se coordinen para cooperar o como pueden estos discernir en cuándo un comportamiento o acción tomada es beneficioso cuando se está trabajando para conseguir una meta común.

La colaboración entre agentes inteligentes y la inteligencia artificial son campos de estudio muy amplios y que aún tienen mucho que desarrollar. Por ello, plataformas como la de Malmo que ayudan a su estudio y experimentación son de gran valor, y mediante el desarrollo de diferentes propuestas y enfoques que aporten más información sobre el tema será posible asentar una base sobre la que seguir mejorando y avanzando.

Anexos

Anexo 1: Bibliografía y referencias

Bibliografía

Johnson M., Hofmann K., Hutton T., Bignell D. (2016) *The Malmo Platform for Artificial Intelligence Experimentation*. Proc. 25th International Joint Conference on Artificial Intelligence, Ed. Kambhampati S., p. 4246. AAAI Press, Palo Alto, California USA.

Itam (1987) Breve historia de la inteligencia artificial. Obtenido de http://biblioteca.itam.mx/estudios/estudio/estudio10/sec_16.html

En Wikipedia. (Sin fecha) History of Artificial Intelligence. Obtenido de https://en.wikipedia.org/wiki/History_of_artificial_intelligence

Yoshida, W., Dolan, R. J., & Friston, K. J. (2008). Game theory of mind. *PLoS computational biology*, *4*(12), e1000254.
En Wikipedia (Extraído el 22-7-2017) Stag Hunt. Obtenido de https://en.wikipedia.org/wiki/Stag_hunt

Microsoft Research. (2017) *The Malmo Collabortive AI Challenge*. Obtenido de https://www.microsoft.com/en-us/research/academic-program/collaborative-ai-challenge/

Keon (2017) Deep Q Learning with Keras and Gym. Obtenido de https://keon.io/deep-q-learning/

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

Leibo, J. Z., Zambaldi, V., Lanctot, M., Marecki, J., & Graepel, T. (2017, May). Multi-agent Reinforcement Learning in Sequential Social Dilemmas. In *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems* (pp. 464-473). International Foundation for Autonomous Agents and Multiagent Systems.

Henry Mao (2017) Reinforcement Learning using Asynchronous Advantage Actor Critic. Obtenido de https://medium.com/@henrymao/reinforcement-learning-using-asynchronous-advantage-actor-critic-704147f91686

François Chollet (2015) Keras, obtenido de https://keras.io/

Referencias

[1] Itam (1987) Breve historia de la inteligencia artificial. Obtenido de http://biblioteca.itam.mx/estudios/estudio/estudio10/sec_16.html

[2]Wikipedia (Extraido el 22-7-2017) Inteligencia artificial. Obtenido de https://es.wikipedia.org/wiki/Inteligencia_artificial

[3]Wikipedia (Extraido el 22-7-2017) Agente inteligente. Obtenido de https://es.wikipedia.org/wiki/Agente_inteligente_(inteligencia_artificial)

[4]Allison Linn (2016) Project Malmo: Using Minecraft to build more intelligent technology. Obtenido de https://blogs.microsoft.com/ai/2016/03/13/project-malmo-using-minecraft-build-intelligent-technology/

[5] Johnson M., Hofmann K., Hutton T., Bignell D. (2016) *The Malmo Platform for Artificial Intelligence Experimentation*. Proc. 25th International Joint Conference on Artificial Intelligence, Ed. Kambhampati S., p. 4246. AAAI Press, Palo Alto, California USA.

[6] En Wikipedia (Extraído el 22-7-2017) Stag Hunt. Obtenido de https://en.wikipedia.org/wiki/Stag_hunt

[7] Microsoft Research. (2017) *The Malmo Collabortive AI Challenge*. Obtenido de https://www.microsoft.com/en-us/research/academic-program/collaborative-ai-challenge/

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

[9]Raúl Arrabales (2016) Deep Learning: Qué es y por qué va a ser una tecnología clave en el futuro de la inteligencia artificial. Obtenido de https://www.xataka.com/robotica-e-ia/deep-learning-que-es-y-por-que-va-a-ser-una-tecnologia-clave-en-el-futuro-de-la-inteligencia-artificial

[10] Keon (2017) Deep Q Learning with Keras and Gym. Obtenido de https://keon.io/deep-q-learning/

[11] Leibo, J. Z., Zambaldi, V., Lanctot, M., Marecki, J., & Graepel, T. (2017, May). Multi-agent Reinforcement Learning in Sequential Social Dilemmas. In *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems* (pp. 464-473). International Foundation for Autonomous Agents and Multiagent Systems.

[12] Henry Mao (2017) Reinforcement Learning using Asynchronous Advantage Actor Critic. Obtenido de https://medium.com/@henrymao/reinforcement-learning-using-asynchronous-advantage-actor-critic-704147f91686

Anexo 2: Terminología y software empleado

1. Conceptos teóricos

Se van a definir a continuación una serie de conceptos que serán utilizados a lo largo de este documento con la intención de facilitar la comprensión del mismo:

Deep Reinforcement Learning: Familia de algoritmos que combina las técnicas con de aprendizaje automático con redes neuronales (Deep Learning) con las técnicas de aprendizaje por refuerzo (Reinforcement Learning)

Deep Learning: Métodos de aprendizaje automático que se basan en que los agentes aprendan a poner en práctica estrategias que llevan a la mayor recompensa a largo plazo mediante un aprendizaje basado en el ensayo y error.

Reinforcement Learning: Familia de métodos de aprendizaje automático que se basa en aprender de la representación de los datos usando estructuras lógicas similares a las redes neuronales con varias capas ocultas especializadas en detectar las diferentes características de los datos recibidos.

Redes Neuronales Artificiales: Modelo de computación basado en el comportamiento de las neuronas biológicas formado por un conjunto de nodo interconectado.

Equilibrio de Nash: Concepto de solución proveniente de la Teoría de Juegos en la que se supone que cada jugador ha optado por su mejor estrategia y en la que el resto de jugadores conocen la estrategia de los demás, por lo que ningún jugador puede ganar una mayor recompensa cambiando de estrategia a no ser que otro jugador cambie antes.

2. Introducción al software empleado

Se va a presentar a continuación las características y funciones que presenta la plataforma *Project Malmo*, así como la de los lenguajes de programación y librerías específicas utilizadas para el desarrollo de este proyecto, con el fin de darle al lector una serie de conocimientos básicos que faciliten la comprensión de esta memoria:

- Project Malmo: Project Malmo es una sofisticada plataforma de experimentación para la inteligencia artificial diseñada para ayudar a la investigación de esta e implementada en el popular videojuego Minecraft. Se trata de un mod para el juego para su versión en Java que permite a agentes inteligentes percibir e interactuar con el entorno de Minecraft.
- Microsoft Azure: Microsoft Azure es un servicio en la nube que ofrece diversas funcionalidades como servicios de cloud computing a servicios de seguridad y comunicación segura entre aplicaciones.

- Python: Python es un lenguaje de programación simple y potente de fácil comprensión con una sintaxis que hace que su código sea más legible. Se trata de un lenguaje multiparadigma que soporta orientación objetos, programación operativa y funcional, además de contar con una amplia gama de librerías disponibles.
- Keras: Keras es una API de redes neuronales de alto nivel escrita en Python y capaz de trabajar con la base de otras librerías matemáticas como Theano, TensorFlow o CNTK. Fue desarrollada con el objetivo de proporcionar una rápida experimentación. Es mayormente utilizada para utilizar técnicas de Deep Learning
- Theano: Theano es una librería de Python que te permite definir, optimizar y evaluar expresiones matemáticas que implican arrays multidimensionales de forma eficiente.

Anexo 3: Programación del agente

1. Introducción

Este anexo contiene una explicación más detallada de la implementación, así como fragmentos de código relevantes para su exposición que pretende completar lo explicado en el apartado 3 de la memoria.

En primer lugar para utilizar la plataforma Malmo esta debe estar instalada en el equipo. Se trata de una instalación relativamente rápida explicada en la propia página de la plataforma. Como se ha explicado en la memoria, la misión, el entorno y otros elementos para hacer funcionar el juego también vienen dados ya preparados por Microsoft, y pueden descargarse de la página oficial del concurso o de su github. Puesto que ejemplos de cómo poner el juego en marcha y utilizar diferentes tipos de agente para este está ya explicado en la respectiva página, no se entrará en detalle de estos puntos y se pasará a explicar el código de la parte de la implementación del agente.

Como también se ha explicado previamente en la memoria, para implementar la red neuronal y actualizar los pesos de esta se ha utilizado la API Keras. Para utilizar esta API y las funciones y métodos que proporciona es necesario tenerla instalada. Todo el código está escrito en el lenguaje Python. Una vez aclarados estos puntos iniciales, se va a proceder a explicar la implementación del agente junto con su código.

2. Implementación de la clase agente DQN

En este apartado se explicará la programación del agente que utiliza el algoritmo DQN para aprender a jugar al minijuego Pig Chase. Si bien se proporcionan clases de diferentes tipos de agente en el código proporcionado por el concurso, la clase agente DQN se ha creado de cero a partir de los conocimientos obtenidos para este trabajo. Todo el código que se muestra está comentado para su mayor claridad. Esta clase tiene como métodos principales, a parte del método de inicio, los métodos act, replay, remember y buil_model. Estos métodos se explicarán con parte de su código más detalladamente a continuación.

2.1 Inicialización del agente

En el método init se definen principalmente las diferentes variables del algoritmo, tales como la tasa de exploración, la de aprendizaje, el número de nodos de entra y de salida de la red etc. Además, se crea la memoria para guardar las experiencias de re-entrenamiento y finalmente se llama al método build_model para general la red neuronal base. Cada uno de los valores que se inicializan tiene su función comentada en el código.

```
def __init__(self, name, target, visualizer = None):
    ##Inicializacion del agente
    super(FocusedAgent, self).__init__(name, len(FocusedAgent.ACTIONS),
                                       visualizer = visualizer)
    self._target = str(target)
    ## Se definen las variables para crear la red neuronal, asi como otras variables necesarias para el algoritmo
    self.state_size = 6 ##Posicion x e y de los agentes y el cerdo
    self.action_size = 3 ##Tres posibles acciones (girar derecha, izquierda o ir hacia delante)
    self.memory = deque(maxlen=5000) ##Inicializacion de la memoria
    self.gamma = 0.95    # tasa de descuento por entrenamiento
    self.epsilon = 1.0  # tasa de exploracion
    self.epsilon_min = 0.01 # tasa minima de exploracion
    self.epsilon_decay = 0.995
    self.learning_rate = 0.001 # tasa de aprendizaje
    self.qvalue= {'MaxQprediction': []} # Estructura para guardar los valores maximos de la funcion Q obtenidos
    self.model = self._build_model()
```

Figura 1: Método init del agente DQN

2.2 Build_model

```
def _build_model(self):
    # Generacion de la red neuronal para el algoritmo DQN
    # Sequential() creates the foundation of the layers. El metodo Sequential() de Keras crea la base de las capas de la red
    model = Sequential()
    # Se creean las capas de la red neuronal
    # Capa de entrada con 6 nodos y una capa oculta con 24 nodos
    model.add(Dense(24, input_dim=self.state_size, activation='relu'))
    # Otra capa oculta con 24 nodos
    model.add(Dense(24, activation='relu'))
    # Capa de salida con 3 nodos, uno por accion posible (move 1, turn 1, turn -1)
    model.add(Dense(self.action_size, activation='linear'))
    model.compile(loss='mse', optimizer=Adam(lr=self.learning_rate))
    return model
```

Figura 2: Método build_model del agente DQN

La clase build_model es la encargada de crear la red neuronal que se utilizará en el algoritmo DQN. Para ello, se utilizan métodos proporcionados por la API Keras que, con los parámetros necesarios crean la red neuronal con el número de nodos, capas ocultas y tasa de aprendizaje que se le ha indicado. En este caso, se crea una red neuronal con una capa de entrada de 6 nodos (uno por cada dato de entrada del estado que le vamos a proporcionar), 3 nodos de salida (uno por cada posible acción del agente) y dos capas ocultas con 24 nodos. La función de cada método está comentada en el código.

2.3 Act

```
def act(self, state, reward, done, is_training=False):
    ##Se adapta el estado para que solo la informacion que nos interesa
    state=self.adapt_state(state)
    ##Elige una accion aleatoria si el valor es menor que el epsilon definido
    if np.random.rand() <= self.epsilon:
        print('Accion aleatoria del agente'+ self.name)
        return random.randrange(self.action_size)
    ##Si no, predice cual de las dos acciones va a teer una mayor recompensa y elige esa
    act_values = self.model.predict(state) #Prediccion por parte de la red neuronal de la mejor accion a tomar
    print('accion PREDECIDA del agente '+ self.name)
    ##Se guarda el valor maximo obtenido por la red neuronal para obtener datos
    self.qvalue['MaxQprediction'].append(np.amax(act_values[0]))
    return np.argmax(act_values[0])  # Devuelve la accion que se predice que proporcionara mejor resultado
```

Figura 3: Método act del agente DQN

Como se explica en la memoria, este método se encarga de devolver el índice de la acción que se va a tomar en cada paso de la partida. Para ello, tiene una probabilidad de épsilon (variable definida en la inicialización del agente y que va decreciendo según el agente se va entrenando) de elegir la acción de forma aleatoria, y una probabilidad de 1-epsilon de tomar una acción basada en la predicción de la red neuronal. Esta predicción se obtiene proporcionando los datos del estado actual del entorno de juego a la red neuronal, que son las posiciones del propio agente, del cerdo y del otro agente, a la red neuronal y se toma la acción cuyo valor espere la red neuronal que sea el mejor de los obtenidos para el resto de accione posibles que puede tomar el agente. Como se explicará en el siguiente apartado, la información que devuelve el juego Pig Chase del entorno es más de la que se necesita, por lo cual el estado debe de ser adaptado antes de ser proporcionado a la red neuronal para la predicción. El resultado de la predicción se obtiene mediante la función predict() de Keras. Aquí también se guarda el valor de la predicción máxima para el posterior estudio de los datos proporcionados.

2.4 Replay

```
def replay(self, batch_size):
    ##Se sacan 32 muestras aleatorias de las experiencias guardadas
    minibatch = random.sample(self.memory, batch_size)
    for state, action, reward, next_state, done in minibatch:
        target = reward
        ##Si no ha sido el ultimo episodio de la partida
        if not done:
          target = reward + self.gamma * \
                   np.amax(self.model.predict(next_state)[0])
        target_f = self.model.predict(state)
        target_f[0][action] = target
        ##Se ajustan los pesos de la red neuronal para que la red aprenda de los resultados obtenidos
        self.model.fit(state, target_f, epochs=1, verbose=0)
    ##Descender el valor de la tasa de exploracion
    if self.epsilon > self.epsilon_min:
        self.epsilon *= self.epsilon_decay
```

Figura 4: Método replay del agente DQN

El método replay es el método clave del algoritmo DQN, ya que es en el que se entrena el agente con experiencias aleatorias para su aprendizaje. Es aquí por tanto donde se reajustan los pesos de la red neuronal. Puesto que las fórmulas que utiliza el algoritmo DQN para hacer esto se han explicado en el desarrollo de la memoria, solo se comentará que mediante el estado, la acción tomada, la puntuación obtenida y el estado siguiente que se ha conseguido de las experiencias previas guardadas se consigue ajustar los pesos para que la red neuronal tienda a maximizar la mayor recompensa a largo plazo. En el código proporcionado se pueden observar parte de los cálculos necesarios para ello.

Esta función extrae tantas experiencias como se haya indicado en la variable batch_size para el entrenamiento del agente. Con cada una de esas experiencias realiza una serie de cálculos necesarios para el ajuste de los pesos. Finalmente, se utiliza el método fit() de Keras para reajustar los pesos con los valores dados. Una vez se ha terminado el entrenamiento, se reduce si es posible la tasa de exploración para que el agente use cada vez más sus propias predicciones.

2.4 Remember

```
def remember(self, state, action, reward, next_state, done):
    state=self.adapt_state(state)
    ##Se guarda en la memoria definida la experiencia con los siguientes datos
    self.memory.append((state, action, reward, next_state, done))
```

Figura 5: Método remember del agente DQN

Para proporcionar un entrenamiento mejor, este necesita entrenarse utilizando experiencias que no tengan correlación. Para ello, es necesario guardarlas en una memoria para su posterior uso y selección de forma aleatoria para el entrenamiento. Para ello, los datos más importantes de cada experiencia se guardan en la memoria definida en la inicialización del agente de una forma sencilla.

Es necesario comentar que el agente tiene otros métodos y funcionalidades pero que, al no ser necesarias para el algoritmo puesto que principalmente se han usado para obtener datos del agente, no se va a exponer su código. Una vez expuesto el código principal de la clase agente, se va a pasar a exponer el código del bucle principal que realiza el agente.

3. Implementación del algoritmo DQN en Malmo

Como se ha explicado anteriormente, la misión, el entorno y ejemplos de código para poner en marcha el juego son proporcionados por los patrocinadores del concurso. Para este trabajo el código que pone en marcha el juego, selecciona el tipo de agentes que van a jugar y controla el bucle que ejecuta cada agente en cada paso de la partida ha sido cambiado en múltiples puntos, adaptándolo para el algoritmo DQN, para que el agente funcione correctamente y para obtener toda la información necesaria para el análisis del proyecto y los resultados. Estos cambios se han hecho en base al código proporcionado llamado eval_sample.py y evaluation.py. Esos dos archivos ponen en marcha el juego usando dos agentes del tipo que se haya especificado en el código y mediante dos threads o hilos ejecutan el bucle de actuación por cada paso de la partida de los agentes durante el número de partidas que se haya especificado. Dado que estas variaciones en el código han sido para poner el agente en marcha y obtener información del agente, solo se va a proceder a exponer el código del bucle del agente, el cual es la parte concerniente al algoritmo DQN.

```
def agent_loop(agent, env, metrics_acc):
    EVAL_EPISODES = 100
    ##Aqui se guardan las recompensas que se van obteniendo para finalmente obtener la media
    accumulators = {'100k': []}
    agent_done = False
    reward = 0
    ##Numero de recompensas individuales obtenidas
    count5 = 0
    ##Numero de recomepensas cooperativas obtenidas
    count25 = 0
    ##Numero de experiencias que se escogen aleatoriamente en cada replay
    batch_size = 32
    episode = 0
    num_memory=0
    state = env.reset()
```

Figura 6: Iniciación de variables al inicio del bucle del agente

En primer lugar, antes de entrar en el bucle se definen el número de partidas a jugar, el tamaño del batch, es decir, el número de experiencias que se utilizarán para el replay y se reinicia el entorno de juego. El resto de variables definidas son utilizadas para la recolección de datos del agente.

```
## Se selecciona una accion a tomar
action = agent.act(state, reward, agent_done, is_training=True)
## Se toma la accion elegida, obteniendose los datos del nuevo estado
next_state, reward, agent_done = env.do(action)
next_state2=adapt_state(next_state)
##Se guarda la experiencia en memoria para el posterior reentrenamiento
agent.remember(state, action, reward, next_state2, agent_done)
##Se actualiza el estado actual al nuevo obtenido
state = next_state
num_memory=num_memory+1
accumulators['100k'].append(reward)
if metrics_acc is not None:
    metrics_acc.append(reward)
```

Figura 7: Interior del bucle de actuación del agente

Una vez empieza el bucle, en cada paso de la partida el agente debe elegir una acción a tomar y realizarla. Una vez esta acción se ha realizado el agente obtiene la información nueva del entorno de juego, la cual junto al estado anterior y la acción tomada se guarda para su posterior uso en el replay. Se guarda así mismo información para obtener datos de la actuación del agente. Este bucle de acciones se repite en cada partida un máximo de 25 veces hasta que se agota el número de acciones tomadas o se llega a una solución cooperativa o individual del juego. Una vez la partida acaba, se pasa a la siguiente parte del bucle, la cual se ejecuta solamente una vez al final de cada partida:

```
if env.done:
    ##Comprobar si se ha obtenido una recomepnsa individual o alreatoria
    if(reward == 24):
        count25= count25+1
    if(reward == 4):
        count5= count5+1
    print('Numero recompensa cooperacion: ')
    print(count25)
    print('numero recomepnsa individual: ')
    print(count5)
    print('Metrics:')
    ##Con los datos guardados se calcula la media de la puntuacion por paso
    metrics = {key: {'mean': mean(buffer),
                     'var': var(buffer),
                     'count': len(buffer)}
               for key, buffer in accumulators.items()}
    print(metrics)

    print('Episode %d (%.2f)%%' % (episode, (episode / EVAL_EPISODES) * 100.))
    ##Se comprueba que hay suficientes experiencias guardadas en memoria oara realizar un replay y si es asi se entra en el
    if num_memory>batch_size:
        print('Entrando a replay del agent :'+ agent.name)
        agent.replay(batch_size)
    ##Se resetea el entorno para comenzar una nueva partida
    state = env.reset()
    while state is None:
        # this can happen if the episode ended with the first
        # action of the other agent
        print('Warning: received state == None.')
        state = env.reset()

    episode += 1
```

Figura 8: Código del agente cuando termina una partida

Una vez el agente ha terminado la partida, imprime por pantalla la información recabada del agente y si tiene suficientes experiencias guardadas en memoria para realizar el replay, pasa a realizar el entrenamiento. Una vez el agente termina el replay, se reinicia el entorno de juego y se comienza una partida nueva, volviendo a ejecutar el bucle de actuación del agente hasta que acaba otra partida. Se juegan tantas partidas como se haya definido previamente en el código.

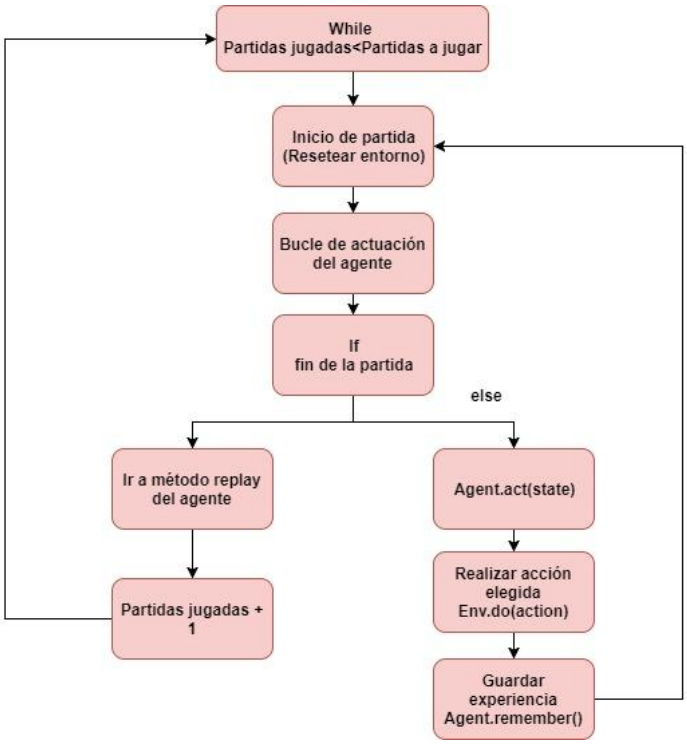

Figura 9: Diagrama de actuación del agente

En la figura 9 se muestra el diagrama de actuación del agente completo con todos los posibles casos de actuación según el estado de juego.