Aprendizaje por refuerzo en StarCraft II

Reinforcement Learning for StarCraft II

Miriam Leis Baltanás
Pablo Joaquín Rodríguez Hidalgo

Trabajo Fin de Grado en Desarrollo de Videojuegos
Facultad de Informática
Universidad Complutense de Madrid
Curso 2020/2021

Directores:

Pedro Pablo Gómez Martín
Antonio A. Sánchez Ruíz-Granados

Índice general

Índice de figuras

Resumen

En este Trabajo Fin de Grado se estudian distintas técnicas de aprendizaje por refuerzo, una rama del aprendizaje automático que ha demostrado en los últimos años ser una de las opciones más populares dentro de este ámbito. *DeepMind* ha aplicado algoritmos de aprendizaje por refuerzo en distintos videojuegos, poniendo de relieve la utilidad de estas aplicaciones para contribuir al avance de la investigación en el campo del aprendizaje automático. En este marco, la finalidad de este trabajo es la aplicación de técnicas de aprendizaje por refuerzo en distintos entornos del videojuego *StarCraft II*. Las características de este videojuego, en concreto el hecho de que incluye tomas de decisiones a distintos niveles con información parcial del estado del entorno, suponen grandes ventajas a la hora de aplicar técnicas de aprendizaje automático respecto a otros videojuegos.

Tras profundizar en el estudio de los algoritmos de aprendizaje por refuerzo Q-Learning y Deep Q-Learning con objeto de entender su funcionamiento correctamente, ambos algoritmos se han implementado en minijuegos de *StarCraft II*. Esta aplicación ha consistido en el desarrollo de jugadores automáticos que aprenden varios objetivos enfocados a la toma de decisiones a distintos niveles en videojuegos RTS. Para ello, se ha realizado un estudio sobre las estrategias habituales en estos videojuegos y se ha implementado una arquitectura reutilizable que permite intercambiar los distintos agentes y entornos de manera sencilla. Finalmente, se analizan los resultados obtenidos en los diferentes experimentos realizados y se presentan las conclusiones extraídas a partir de dichos resultados.

Palabras clave

Aprendizaje por refuerzo, Deep Q-Learning, Q-Learning, Aprendizaje automático, Jugador automático, PySC2, Starcraft II.

Abstract

In this Bachelor's Degree Final Proyect, different reinforcement learning techniques are studied, a branch of machine learning that has proven in recent years to be one of the most popular options in this field. *DeepMind* has applied reinforcement learning algorithms in different videogames, highlighting the usefulness of these applications to contribute to the advancement of research in the field of machine learning. In this framework, the purpose of this work is the application of reinforcement learning techniques in different environments of the *StarCraft II* videogame. The characteristics of this video game, specifically the fact that it includes decision-making at different levels with partial information about the state of the environment, represent great advantages when applying machine learning techniques compared to other videogames.

After delving into the study of Q-Learning and Deep Q-Learning reinforcement learning algorithms in order to correctly understand how they work, both algorithms have been implemented in *StarCraft II* minigames. This application has consisted of the development of automatic players that learn various objectives focused on decision-making at different levels in RTS video games. To do this, a study has been carried out on the usual strategies in these video games and a reusable architecture has been implemented that allows the different agents and environments to be exchanged easily. Finally, the results obtained in the different experiments carried out are analyzed and the conclusions drawn from these results are presented.

Keywords

Reinforcement learning, Q-Learning, Deep Q-Learning, Machine learning, Automatic player, PySC2, Starcraft II.

Capítulo 1

Introducción

Los videojuegos han experimentado un gran crecimiento en los últimos 50 años, sobre todo, a partir de su consolidación con las máquinas recreativas. En la actualidad, la industria del videojuego se encuentra en pleno auge, obteniendo en el año 2019 unas ganancias de 43.4 billones de dolares [4]. Tradicionalmente, las empresas han centrado sus esfuerzos en los motores gráficos con objeto de mejorar la calidad de elementos como la iluminación, el modelado 3-D y la animación. Sin embargo, pocos avances se han logrado en los elementos relacionados con la inteligencia artificial de los videojuegos. Los avances realizados en la animación y la iluminación aportan al jugador una mayor inmersión y un sentimiento de armonía visual, pero pueden no constituir una experiencia de juego suficiente, ya que el reto a la hora de vencer a los enemigos puede resultar excesivamente fácil para algunos jugadores.

Los jugadores suelen preferir enfrentarse contra otros jugadores humanos, ya que les aporta un mayor grado de dificultad y una oportunidad para aprender nuevas estrategias, haciendo más divertida y satisfactoria la experiencia de juego [22]. Por ello, la atención otorgada a los aspectos relacionados con la inteligencia computacional en los videojuegos ha aumentado en los últimos años. Cada vez son más los jugadores que exigen enemigos que demuestren un comportamiento más inteligente, haciendo que los videojuegos resulten más interesantes y difíciles de superar.

Los videojuegos siempre han sido un campo de pruebas idóneo para las técnicas de inteligencia artificial en boga en cada periodo histórico, y ahora ocurre lo mismo con el aprendizaje automático. Prueba de ello es la publicación de numerosos libros y artículos sobre el tema [7, 25]. Un punto a favor en el uso de videojuegos para la investigación del aprendizaje automático es la retroalimentación visual que se adquiere a la hora de ejecutar una acción, permitiendo analizar fácilmente la actuación de un algoritmo.

Las técnicas de aprendizaje automático permiten que un jugador controlado por ordenador aprenda a actuar dependiendo del entorno en el que se encuentre y de las posibilidades que tiene, desarrollando estrategias que le permitan adaptarse a las progresivas dificultades. Las ventajas de utilizar estas técnicas son evidentes en el mundo de los videojuegos, ya que se pueden emplear tanto en el comportamiento de los personajes no jugadores (NPC) como en la generación de contenido para los videojuegos,

por ejemplo, en los niveles o las texturas [7]. El uso de estas técnicas se extienden a una gran variedad de géneros, habiéndose demostrando su utilidad en videojuegos como *Minecraft* (desarrollado por *Mojang Studios*, en el año 2009).

Entre el amplio rango de géneros de videojuegos que existen, se encuentran los juegos de estrategia en tiempo real (*real-time strategy*) (RTS), con títulos como *Warcraft* (*Blizzard Entertainament*, 1994), *Dune II* (*Westwood Studies*, 1992) o *StarCraft* (*Blizzard Entertainament*, 1998). En estos videojuegos, el jugador debe controlar múltiples unidades o personajes con el objetivo de vencer en una batalla o conflicto a otro jugador, ya sea humano o automático. Los juegos RTS aportan entornos complejos, con una gran variedad de objetos y un entorno dinámico influenciado por la ausencia de turnos [8].

A la hora de diseñar jugadores automáticos en los juegos RTS, el reto reside en el diseño y la ejecución de planes complejos en los cuales interfieren múltiples unidades en distintos periodos de tiempo. Por ello, este reto es significativamente mayor en los videojuegos RTS que en juegos de mesa como el ajedrez, a pesar del componente estratégico que comparten. Las decisiones tomadas en estos juegos son extremadamente importantes, pues las acciones ejecutadas al principio de una partida pueden tener un gran impacto sobre toda la estrategia de juego [24].

Una técnica de aprendizaje automático muy empleada en estos juegos es el aprendizaje por refuerzo (descrito en el capítulo 2), especialmente el aprendizaje por refuerzo profundo. Esto se debe a la capacidad de las redes neuronales para extraer información importante de datos complejos, similares a los que cuentan los juegos RTS. La ventaja de emplear aprendizaje por refuerzo en los juegos RTS reside en que las acciones realizadas pueden tener un gran impacto en la continuidad de la partida. Dado que en estos juegos se necesita tener en cuenta cada acción realizada y su repercusión, la utilización de algoritmos de aprendizaje por refuerzo (como son Q-Learning y Deep Q-Learning) permiten tener en cuenta la evolución que ha seguido el entorno.

La saga de *StarCraft* (*Blizzard Entertainment*) es la más empleada del género RTS en los estudios de aplicación de las técnicas de aprendizaje por refuerzo. Esto se debe en parte a la aparición de la interfaz de programación (API) llamada *The Brood War API* (BWAPI)[1] en 2009. Esta API permite que los investigadores desarrollen jugadores automáticos con acceso al entorno de *StarCraft* para adquirir información, por ejemplo, de los estados de la partida. Además, estos jugadores automáticos pueden comunicarse con el entorno para enviar órdenes a sus unidades, facilitando la tarea de aprendizaje.

Con la aparición de la secuela *StarCraft II* surgió una nueva API más completa y diseñada para probar algoritmos de aprendizaje automático, llamada *PySC2*[2] (detallada en el apartado 3.4). Esta API, desarrollada por *DeepMind* y la propia *Blizzard Entertainment*, facilita estas investigaciones gracias a elementos como una interfaz que permite visualizar de forma simplificada el entorno del juego. También cuenta con un conjunto de minijuegos que permiten explorar mecánicas concretas (como el movimiento y el combate entre unidades) sin tener que lidiar con la complejidad del juego completo.

[1] `https://bwapi.github.io/`

[2] `https://github.com/deepmind/pysc2`

1.1. Objetivos

Como ha quedado de manifiesto en la introducción, la utilización de jugadores automáticos en el desarrollo de videojuegos ha adquirido un creciente auge e importancia en los últimos años. A pesar de ello, no ha sido un objeto de estudio en los estudios de Grado realizados y, por ello, hemos considerado de interés centrar el presente Trabajo Fin de Grado (TFG) en esta temática. En concreto, los objetivos planteados son los siguientes:

1. Profundizar en el estudio del uso de técnicas de aprendizaje por refuerzo en el desarrollo de jugadores automáticos en los videojuegos, especialmente, en los juegos RTS.
2. Investigar los retos que supone la creación de jugadores automáticos en juegos de estrategia en tiempo real.
3. Desarrollar un código reutilizable que permita probar distintos algoritmos de aprendizaje en diferentes entornos de forma sencilla y sin tener que lidiar con los detalles de la plataforma.
4. Estudiar el funcionamiento de los algoritmos Q-Learning y Deep Q-Learning para la resolución de distintos minijuegos de dificultad creciente.

1.2. Herramientas usadas en el desarrollo

El lenguaje de programación empleado para el desarrollo de los jugadores automáticos ha sido Python. Esta elección se ha realizado debido a la flexibilidad, a la estabilidad y a las herramientas que nos aporta este lenguaje de programación. La simplicidad que ofrece Python, gracias a la amplia cantidad de librerías que facilitan el aprendizaje automático, permite a los desarrolladores centrarse en solucionar un problema en particular y dejar de lado otras complicaciones que puedan surgir.

Python cuenta con un gran número de herramientas destinadas al desarrollo de aprendizaje automático. De estas herramientas hemos optado por usar *TensorFlow 2.4.0*[3], la cual incluye *Keras 2.4.3* para el desarrollo de las redes neuronales. Además, hacemos uso en repetidas ocasiones de librerías básicas como *NumPy*[4], para la manipulación de datos, o *absl.flags*[5], para personalizar los parámetros de nuestro programa.

La herramienta más destacable que hemos usado en el desarrollo es *PySC2*, mencionada previamente, que aporta una interfaz especializada para el aprendizaje automático en el juego de *StarCraft II*, aunque también se podrían aplicar otras técnicas de inteligencia artificial. *PySC2* permite acceder a los estados del juego, las recompensas y otros valores necesarios para el desarrollo de un jugador automático de esta índole. El

[3] `https://www.tensorflow.org`
[4] `https://pypi.org/project/numpy/`
[5] `https://pypi.org/project/abs-imports/`

funcionamiento de esta interfaz se describe más adelante con mayor profundidad (sección 3.4). Para emplear este complemento es necesario tener instalado el propio juego de *StarCraft II*, siendo igualmente primordial el uso de este juego para la investigación desarrollada para este TFG.

Finalmente, para implementar los distintos programas y hacer uso de las herramientas mencionadas, se ha optado por emplear *Visual Studio Code*[6] debido a la experiencia que se ha obtenido con esta IDE en el transcurso de los estudios de Grado. Por otra parte, *Visual Studio Code* permite al desarrollador compilar y probar de forma rápida los programas.

1.3. Estructura del documento

El presente TFG está estructurado en 8 capítulos. Tras la introducción, en el capítulo 2 se presentan los fundamentos teóricos de los algoritmos de aprendizaje por refuerzo que se van a emplear, exponiendo la funcionalidad de este área del aprendizaje automático y profundizando en los conceptos de los algoritmos de los que se hace uso en este trabajo: Q-Learning y Deep Q-Learning. Además, debido a la estrecha relación del segundo algoritmo con las redes neuronales, también se incluye una explicación de este modelo computacional.

Seguidamente, en el capítulo 3 se estudia el entorno que se emplea en el desarrollo las pruebas, exponiendo las características comunes de los videojuegos de estrategia en tiempo real (RTS), y qué características hacen que *StarCraft II* destaque no sólo como juego RTS, sino también en el desarrollo de estas pruebas. Por último, se plantea cómo se puede llegar a trabajar con este entorno.

A partir de los fundamentos descritos, en el capítulo 4 se explica la arquitectura que se ha desarrollado, que se caracteriza por ser reutilizable para distintos entornos y algoritmos. En este capítulo se exponen las interfaces y clases básicas que se han llevado a cabo, describiendo sus relaciones y su funcionamiento interno.

En los siguientes capítulos se describen las pruebas realizadas, abordando en cada uno de estos capítulos los distintos minijuegos que se han empleado. En concreto, en el capítulo 5 se explican las pruebas llevadas a cabo en el minijuego *Moverse a la baliza*, el cual presenta el menor grado de complejidad de los distintos minijuegos utilizados y por ello es el más apropiado para una primera toma de contacto con el aprendizaje por refuerzo. Con este minijuego se han puesto en práctica los dos algoritmos estudiados: Q-Learning y Deep Q-Learning. Para cada uno de estos algoritmos se definen las configuraciones llevadas a cabo, analizándose y comparándose los resultados obtenidos.

En el capítulo 6 se describen las pruebas realizadas con el minijuego *Derrotar zealots con blinks*, que presenta un mayor nivel de complejidad del entorno respecto al anterior minijuego y que nos ha permitido explorar la aplicación del algoritmo Deep Q-Learning sobre acciones a corto plazo. Este minijuego se ha modificado en varias

[6] `https://code.visualstudio.com/`

ocasiones para generar entornos diferentes y permitir el enfrentamiento a distintos retos. Así, en este capítulo se desarrollan las modificaciones y configuraciones que se han realizado en cada una de estas pruebas, analizando los resultados obtenidos.

En el capítulo 7 se describen las pruebas realizadas con el último minijuego, *Construir marines*. Este minijuego aporta un nuevo reto respecto a los anteriores: las acciones durativas, las cuales se caracterizan por influir en el entorno con el paso del tiempo. Al igual que en los capítulos anteriores, se exponen las configuraciones que se han llevado a cabo para la realización de estas pruebas, analizando los resultados obtenidas en las mismas.

Finalmente, en el capítulo 8 se resumen los principales logros del trabajo realizado y se analiza el cumplimiento de los objetivos planteados. Además, se exponen las líneas de trabajo futuras que permitirían dar continuidad a lo realizado en este proyecto.

1.4. Código fuente

La arquitectura desarrollada en este trabajo, junto a todos los jugadores automáticos programados y los distintos entornos empleados, están disponibles en el siguiente repositorio de *GitHub*: `https://github.com/MiriamLeis/TFG-BotSC`

Introduction

Video games have experienced great growth in the last 50 years, especially since their consolidation with arcade machines. Currently, the video game industry is in expansion, obtaining in 2019 a profit of 43.4 billion dollars [4]. Traditionally, companies have focused their efforts on graphics engines in order to improve the quality of elements such as lighting, 3-D modeling and animation. However, little progress has been made in the elements related to artificial intelligence of the game. Advances made in animation and lighting provide the player a greater immersion and a sense of visual harmony, but may not be a sufficient game experience, as the challenge of defeating enemies can be too easy for some players.

Players often prefer to confront other human players, as it gives them a higher degree of difficulty and an opportunity to learn new strategies, making the gaming experience more fun and satisfying [22]. For this reason, the attention given to aspects related to computational intelligence in video games has increased in recent years. More and more players are demanding enemies to demonstrate smarter behaviour, making video games more interesting and difficult to beat.

Video games have always been an ideal testing ground for the most popular artificial intelligence techniques in each historical period, and now the same is happening with machine learning. Proof of this is the publication of numerous books and articles on the subject [7, 25]. A plus in the use of video games for researching machine learning is the visual feedback that is acquired when executing an action, allowing the performance of an algorithm to be easily analyzed.

Machine learning techniques allow a computer-controlled player to learn to act depending on the environment in which he finds himself and the possibilities he has, developing strategies that allow him to adapt to progressive difficulties. The advantages of using these techniques are evident in the world of video games, since they can be used both in the behaviour of non-player characters (NPC) and in the generation of content for video games, for example, in levels or textures [7]. The use of these techniques extends to a wide variety of genres, having proven their usefulness in video games such as *Minecraft* (developed by *Mojang Studios*, in 2009).

Among the wide range of video game genres that exist, *real-time strategy* (RTS) can be found, with titles such as *Warcraft* (*Blizzard Entertainament*, 1994), *Dune II* (*Westwood Studies*, 1992) o *StarCraft* (*Blizzard Entertainament*, 1998). In these video games, the player must control multiple units or characters in order to defeat another player in a battle, either human or automatic. RTS games provide complex environ-

ments, with a great variety of objects and a dynamic environment influenced by the absence of turns [8].

When it comes to designing automatic players in RTS games, the challenge lies in the design and execution of complex plans in which multiple units interfere in different periods of time. For this reason, this challenge is significantly greater in RTS video games than in board games such as chess, despite the strategic component they share. The decisions made in these games are extremely important, as the actions taken at the beginning of a game can have a great impact on the entire game strategy [24].

A widely used machine learning technique in these games is reinforcement learning (described in chapter 2), especially deep reinforcement learning. This is due to the ability of neural networks to extract important information from complex data, similar to those found in RTS games. The advantage of using reinforcement learning in RTS games is that the actions taken can have a great impact on the continuity of the game. Given that in these games it is necessary to take into account each action carried out and its repercussion, the use of reinforcement learning algorithms (such as Q-Learning and Deep Q-Learning) allow us to take into account the evolution that the environment has followed.

The *StarCraft* saga (*Blizzard Entertainment*) is the most widely used of the RTS genre in studies of the application of reinforcement learning techniques. This is partly due to the appearance of the programming interface(API) called *The Brood War API* (BWAPI) [7] in 2009. This API allows researchers to develop automated players with access to the *StarCraft* environment to acquire information, for example, on the states of the games. In addition, these automatic players can communicate with the environment to send orders to their units, facilitating the learning task.

With the appearance of the sequel *StarCraft II* came a new, more complete API designed to test machine learning algorithms, called *PySC2* [8] (detailed in section 3.4). This API, developed by *DeepMind* and *Blizzard Entertainment* itself, facilitates these investigations thanks to elements such as an interface that allows the game environment to be seen in a simplified way. It also has a set of minigames that allow you to explore specific mechanics (such as movement and combat between units) without having to deal with the complexity of the full game.

Objectives

As the introduction has shown, the use of automatic players in video game development has become increasingly popular and important in recent years. Despite this, it has not been an object of study in the degree, therefore, we have considered it of interest to focus this Bachelor's Degree Final Proyect (TFG) on this subject. Specifically, the objectives set are the following:

[7] `https://bwapi.github.io/`

[8] `https://github.com/deepmind/pysc2`

1. Deepen the study of the use of reinforcement learning techniques in the development of automatic players in video games, especially in RTS games.

2. Investigate the challenges of creating automated players in real-time strategy games.

3. Develop a reusable code that allows you to test different learning algorithms in different environments in a simple way and without having to deal with the details of the platform.

4. Study how Q-Learning and Deep Q-Learning algorithms work, and solving different minigames of increasing difficulty.

Tools used in development

The programming language used for the development of the bots has been Python. This choice has been made due to the flexibility, stability and tools that this programming language gives us. The simplicity that Python offers, thanks to the vast number of libraries that facilitate machine learning, allows developers to focus on solving a particular problem and ignore other complications that may arise.

Python has a large number of tools for machine learning development. Of these tools we have chosen to use *TensorFlow 2.4.0*[9], which includes *Keras 2.4.3* for the development of neural networks. In addition, we repeatedly make use of basic libraries such as *NumPy*[10], for data manipulation, or *absl.flags*[11], to customize the parameters of our program.

The most notable tool we have used in development is *PySC2*, mentioned previously, which provides a specialized interface for machine learning in the *StarCraft II* game, although other artificial intelligence techniques could also be applied. *PySC2* allows access to the game statuses, rewards and other values necessary for the development of an automatic player of this nature. The operation of this interface is described in more detail later (section 3.4). To use this library, it is necessary to have *StarCraft II* installed, being equally primary the use of this game for the the research developed for this TFG.

Finally, to implement the different programs and make use of the aforementioned tools, it has been chosen to use *Visual Studio Code* [12] due to the experience that it has been obtained with this IDE in the course of degree studied. On the other hand, *Visual Studio Code* allows the developer to quickly compile and test programs.

[9] `https://www.tensorflow.org`
[10] `https://pypi.org/project/numpy/`
[11] `https://pypi.org/project/abs-imports/`
[12] `https://code.visualstudio.com/`

Document structure

This TFG is structured in 8 chapters. After the introduction, the chapter 2 presents the theoretical foundations of the reinforcement learning algorithms to be used, exposing the functionality of this area of machine learning and delving into the concepts of algorithms which are used in this work: Q-Learning and Deep Q-Learning. Furthermore, due to the close relationship of the second algorithm with neural networks, an explanation of this computational model is also included.

Next, in the chapter 3, the environment used in the development of the tests is studied, exposing, on the one hand, the common characteristics of real-time strategy (RTS) video games, and on the other, what features make *StarCraft II* stand out not only as an RTS game, but also for the development of these tests. Finally, it is considered how to get to work with this environment.

Based on the fundamentals described, the chapter 4 explains the architecture that has been developed, which is characteristic by being reusable for different environments and algorithms. In this chapter, the basic interfaces and classes that have been implemented are exposed, describing their relationships and their internal workings.

The following chapters describe the tests carried out, addressing in each of these chapters the different minigames that have been used. Specifically, chapter 5 explains the tests carried out in the minigame *Move to the beacon*, which presents the least degree of complexity of the different minigames used and, therefore, it was the most appropriate for a first contact with reinforcement learning. With this minigame, the two algorithms studied have been put into practice: Q-Learning and Deep Q-Learning. For each of these algorithms, the configurations carried out are defined, analyzing and comparing the results obtained.

In the chapter 6 the tests carried out with the minigame *Defeat zealots with blinks* are described, which presents a higher level of complexity of the environment compared to the previous minigame and which has allowed us to explore the application of the algorithm Deep Q-Learning on short-term decisions. This minigame has been modified several times to generate different environments and allow facing different challenges. Thus, in this chapter the modifications and configurations that have been made in each of these tests are developed, analyzing the results obtained.

The chapter 7 describes the tests performed with the latest minigame, *Build Marines*. This minigame provides a new challenge compared to the previous ones: latent actions, which are characteristic by influencing the environment over time. As in the previous chapters, the configurations that have been carried out to carry out these tests are exposed, analyzing the results obtained in them.

Finally, in the chapter 8 the main achievements of the work carried out are summarized and the fulfillment of the proposed objectives are analyzed. In addition, future lines of work are exposed that would allow continuity to what has been done in this project.

Source code

The architecture developed in this work, together with all the programmed automatic players and the different environments used, are available in the following *GitHub* repository: `https://github.com/MiriamLeis/TFG-BotSC`

Capítulo 2

Aprendizaje por refuerzo

2.1. Aprendizaje automático

Desde finales del siglo XX, más concretamente desde que en el año 1997 la supercomputadora *Deep Blue* consiguiera vencer al campeón del mundo de ajedrez Garry Kasparov [12], la inteligencia artificial entró en auge, fomentando el desarrollo del aprendizaje automático. Las cantidades masivas de información disponibles hoy en día para entrenar modelos, unidos al aumento de la potencia de cálculo de los ordenadores, han sido también cruciales para este campo de investigación.

El aprendizaje automático se conforma como un subcampo de las ciencias de la computación y una rama de la inteligencia artificial [16]. Su objetivo principal es el desarrollo de técnicas de aprendizaje que permitan a las computadoras actuar en un entorno, utilizando como base de mejora la experiencia a través de la identificación de patrones, la revisión de datos y la comparación de resultados. El conjunto de técnicas que conforman el aprendizaje automático, pueden ser considerados como un intento de automatizar partes del método científico a través de algoritmos matemáticos. Estas técnicas son entendidas como un proceso de inducción del conocimiento, aportando una amplia gama de aplicaciones, como son los motores de búsqueda, los diagnósticos médicos y el análisis del mercado de valores, entre otras.

En los últimos años, la evolución exponencial del aprendizaje automático ha supuesto que este esté presente en algunos de los programas que utilizamos cada día. Un ejemplo es *Google*, que emplea estas técnicas de aprendizaje para la búsqueda de imágenes, analizando patrones de píxeles y colores. De esta forma, *Google* logra identificar las diferencias y las comparara con su base de datos, ofreciéndonos imágenes que se encuentren relacionadas con nuestra búsqueda inicial. En el mundo de los videojuegos, el aprendizaje automático se utiliza en algunas tareas como pueden ser el control de los personajes no jugables (NPC), la generación de contenido procedural (PCG) [37] para la creación de texturas y misiones o, como en el caso de *Minecraft*, para la propia generación del mundo.

Los algoritmos de aprendizaje automático se clasifican habitualmente en las si-

guientes categorías:

- Aprendizaje supervisado. Clasifica los datos por etiquetas y construye un modelo que predice la etiqueta de un nuevo dato a partir de los ya almacenados

- Aprendizaje no supervisado. Busca estructuras o patrones entre los datos, como por ejemplo la agrupación o *clustering*, el cual agrupa los datos según sus similitudes.

- Aprendizaje semi-supervisado. Realiza una combinación de los dos anteriores, diferenciando los datos etiquetados de los no etiquetados.

- Aprendizaje por refuerzo. Almacena los resultados de un agente que actúa en un entorno, aprendiendo qué acción es más favorable en cada estado del mismo. El presente trabajo se centra principalmente en este último tipo.

2.2. Aprendizaje por refuerzo

El aprendizaje por refuerzo (*Reinforcement Learning*) (RL), permite al agente aprender a resolver problemas que suceden dentro de un entorno con la finalidad de conseguir un objetivo [38]. El agente aprende a priorizar el conseguir la mayor recompensa posible en cada entorno para lograr sus objetivos.

En el aprendizaje por refuerzo el agente interactúa con un entorno a través de tres variables: el estado del mismo, la acción con la cual el agente influye en él y la recompensa que recibe al realizar esa acción [36, 6]. Así pues, cada cierto periodo de tiempo el agente recibe un estado del sistema, lo que le lleva a realizar una acción que transforma el entorno y produce un nuevo estado. Dependiendo de las características del estado, se calcula la recompensa que va a recibir y se le proporciona la misma, repitiendo estos pasos de forma cíclica (figura 2.1).

El agente elige las acciones que va a realizar en cada estado en función de la acción con la que haya obtenido una mayor recompensa en el pasado. Para poder tomar esta decisión, el agente ha debido realizar las acciones de antemano para así elegir la más favorable y, a su vez, explorar otras acciones que le permitan lograr una mejor elección en el futuro.

Todo algoritmo de aprendizaje por refuerzo debe seguir alguna política que permita decidir qué acción realizar en cada uno de los estados del entorno [9]. Los algoritmos que utilizan una sola política para explorar el espacio de posibilidades son conocidos como *algoritmos on-policy*. En cambio, los que emplean más de una se denominan *algoritmos off-policy*.

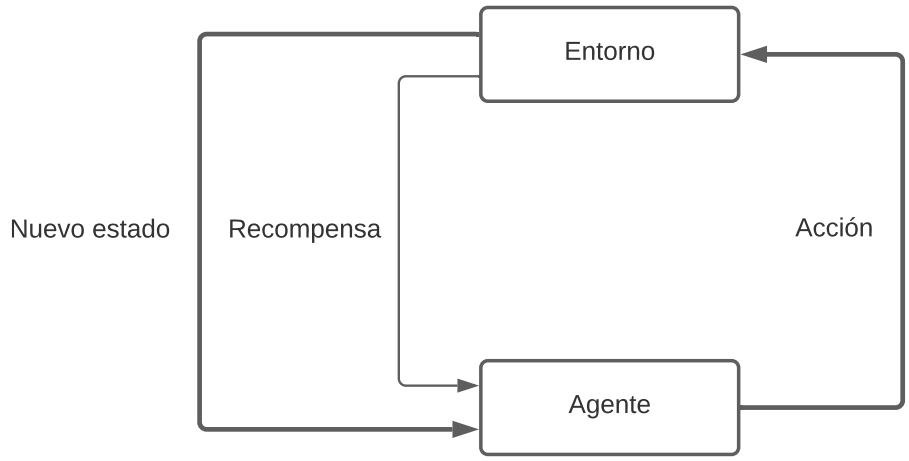


Figura 2.1: Interacción del agente con el entorno en el aprendizaje por refuerzo [38]

2.3. Procesos de decisión de Markov

Los problemas de aprendizaje por refuerzo se basan en tomar decisiones fundamentado en el estado del entorno en un momento específico. Si este estado contiene toda la información, tanto actual como anterior, el problema puede ser formulado con la propiedad de Markov, nombrada así por el matemático ruso Andrei A. Markov (1856-1922).

Los problemas de aprendizaje por refuerzo que presentan la propiedad de Markov se denominan como *proceso de decisiones de Markov*, conocido en inglés como *Markov decision process* (MDP). Los MDP están definidos por la siguiente tupla:

$$M = \{S, A, R, P\} \tag{2.1}$$

S representa el conjunto de estados del entorno, siendo *s* un estado concreto del mismo. De la misma forma, *A* es el conjunto de las posibles acciones que puede ejecutar el agente en ese entorno, siendo *a* una de las acciones que puede realizar. *R* representa la recompensa que gana el agente al realizar la acción *a* en el estado *s*. Y finalmente, *P* representa la probabilidad de transición a un estado nuevo *s'* cuando se realiza la acción *a* en el estado *s*. Esta probabilidad se representa como P(s,a,s') [34].

Por otro lado, los procesos de decisión de Markov cuentan con un estado *s* que detiene al agente una vez que este consigue llegar a él. Los procesos que tienen lugar desde el primer estado al último son denominados episodios. Para aprender apropiadamente la política que mejor maximice la recompensa final, los algoritmos de aprendizaje por refuerzo requieren miles de episodios [39].

El objetivo del aprendizaje de un MDP es lograr recompensas. Si el agente se preocupase únicamente por la recompensa inmediata, sería suficiente un criterio de optimi-

zación simple. No obstante, hay varias maneras de tener en cuenta los comportamientos futuros a la hora de recibir recompensas. Aunque existen tres modelos de optimización en un MDP, las aplicaciones de aprendizaje por refuerzo usan principalmente el criterio de recompensa total descontada.

Este modelo está definido por la ecuación 2.2 y en él se tienen en cuenta las recompensas recibidas hasta el momento, aplicándole un descuento en función de lo alejada en el tiempo se encuentra su obtención. Este valor de descuento se representa con un valor γ entre cero y uno.

$$E\big[\sum_{t=1}^{\infty}\gamma^t \cdot r_t\big] \tag{2.2}$$

El objetivo de todo MDP es encontrar la política que le permita conseguir la mayor recompensa posible. Una política es una función que ofrece para cada estado *s* perteneciente a *S* una acción *a* perteneciente al conjunto *A* [5]. Para analizar si en MDP una política π es óptima, se emplea el criterio de recompensa total descontada. Si, respecto a las políticas posibles, π maximiza la recompensa total en todos los estados, esta se considera óptima. No obstante, resulta muy complejo maximizar todas las posibles políticas que se pueden usar para conseguir la recompensa total maximizada debido al extenso número de políticas a elegir.

Una política π es estacionaria y determinista si mapea el estado actual directamente hacia una acción, ignorando el resto del proceso de decisión. Esta propiedad evita maximizar sobre el extenso espacio de políticas posibles. En los entornos MDP, siempre hay una política de esta índole que es óptima.

2.3.1. Funciones de valor

En un MDP el correcto comportamiento de un agente depende del criterio de optimización. Una forma de unir el criterio de optimización a las políticas es definiendo funciones de valor [34]. Una función de valor representa cómo de bueno es estar en cierto estado para un agente o, dicho de otra manera, cómo de beneficioso es realizar una acción en un estado. Las funciones de valor están definidas para unas políticas determinadas.

Una propiedad fundamental de una función de valor es que soporta propiedades recursivas. El valor de un estado *s* bajo una política π, denominada $V^{\pi}(s)$, es el resultado obtenido cuando se está en el estado *s* y se sigue dicha política. Para cualquier estado *s* y política π, definidos recursivamente, se puede emplear la *Ecuación de Bellman*:

$$V^{\pi}(s) = \sum_{s'} P(s, \pi(s), s') \cdot \big(R(s, a, s') + \gamma \cdot V^{\pi}(s')\big) \tag{2.3}$$

En un MDP con un criterio descontado finito, el comportamiento óptimo puede

conseguirse identificando el valor óptimo definido por la siguiente ecuación:

$$V^{\pi}(s) = R(s,a) + \gamma \cdot \sum_{s'} P(s,a,s') \cdot V^{\pi}(s') \tag{2.4}$$

También es posible condicionar la recompensa en el estado *s'*. El valor óptimo de la ecuación 2.4 es el valor de la función de una política óptima y es único para el valor γ. La política que maximiza la recompensa media inmediata (*myopic policy*) respecto al valor $V^{\pi}(s)$ es la política π_V tal que:

$$\pi_V(s) = argmax_a(R(s,a) + \gamma \cdot \sum_{s'} P(s,a,s') \cdot V(s')) \tag{2.5}$$

2.4. Q-Learning

Q-Learning [43] es una técnica básica del aprendizaje por refuerzo creada por Chris Watkins en 1989 que amplió e integró otros trabajos existentes de investigación sobre aprendizaje por refuerzo. Esta técnica aporta a los agentes la capacidad de aprender a actuar de forma óptima en los problemas de tipo MDP descritos en el apartado anterior.

El proceso de aprendizaje es el siguiente: un agente realiza una acción *a* en un estado concreto *s*, se evalúa el efecto de esa acción en el entorno, se observa el nuevo estado *s'* y se le asocia una recompensa *r*. A la recompensa recibida, se le suma la máxima recompensa asociada al estado *s'* multiplicada por un factor de descuento, lo que permite que esa recompensa se propague a las acciones que han llevado a ese estado *s'*.

Para que el aprendizaje del agente se realice de forma correcta y precisa, este debe probar repetidamente todas las acciones posibles en cada uno de los estados, aprendiendo así qué acción es la idónea.

En esta metodología se emplea una tabla-Q que asocia pares (acción-estado) a valores numéricos, llamados valores-Q, para mejorar iterativamente el comportamiento del agente. Los elementos de esta tabla representan las recompensas de todas las acciones realizadas sobre un estado particular del entorno, y su objetivo es que el agente en cuestión aprenda una política que le ayude a elegir qué acción tomar bajo qué circunstancias, eligiendo, cuando ha terminado de aprender, la acción con mayor recompensa en dicho estado (figura 2.2).

Q-Learning se clasifica dentro de los algoritmos *off-policy* y estima el valor resultante de una acción-estado aplicando en principio una política avariciosa (*greedy*).

Una política *greedy* significa que el agente siempre realiza la acción que más recompensa le va a aportar. El problema surge en que, si un agente emplease esta política constantemente, nunca sería capaz de explorar los efectos de otras acciones en los estados del entorno y no podría hallar la acción más favorable en cada estado. Es por esto

TABLA-Q	Acción 1	. . .	Acción M
Estado 1	Valor-Q	. . .	Valor-Q
. . .	. . .	. . .	. . .
Estado N	Valor-Q	. . .	Valor-Q

Figura 2.2: Esquema de una tabla-Q de Q-Learning

por lo que, para elegir la acción *a*, se emplea un valor ϵ ($0 \leq \epsilon \leq 1$) para decidir la acción que va a realizar. Cuanto más próximo a 0 se encuentre el valor ϵ más posibilidades hay de que se elija la acción de manera aleatoria. A lo largo del aprendizaje se va modificando el valor ϵ, empezando en $\epsilon = 0$ y convergiendo poco a poco a $\epsilon = 1$.

2.4.1. Algoritmo

El algoritmo de Q-Learning se basa en la *Ecuación de Bellman*. Esta ecuación define la relación entre un estado (o una pareja acción-estado) con sus sucesores. Aunque se expresa de distintas formas como se aprecia en la sección 2.3.1, la más común en aprendizaje por refuerzo es la representación 2.6.

$$Q(s,a) = \sum_{S',r} p(s',r|s,a)[r + \gamma \cdot maxQ(s',a')] \tag{2.6}$$

$$Q(s,a) = (1-\alpha) \cdot Q(s,a) + \alpha \cdot (r + \gamma \cdot maxQ(s',a')) \tag{2.7}$$

Aún así, si existe certeza (las probabilidades son 1 o 0) se emplea la fórmula de la ecuación 2.7. En esta, el símbolo α representa el índice de aprendizaje, el cual indica la velocidad del aprendizaje. Por ejemplo, si α tuviese un valor de 0, el agente no aprendería, pero, a medida que el valor de α aumenta hasta 1, el agente va aprendiendo a mayor velocidad.

Por otra parte, el valor de γ marca el factor de descuento, situado al igual que el α entre un rango de 0 y 1. Este número se emplea en la propagación, marcando el hecho de que las recompensas recibidas por las acciones más antiguas tienen menor valor que las recibidas por las acciones más actuales. Este valor γ también puede ser interpretado como la probabilidad de tener éxito en cada paso. Si el valor del factor de descuento es cero las recompensas no podrán propagarse entre los distintos estados. El

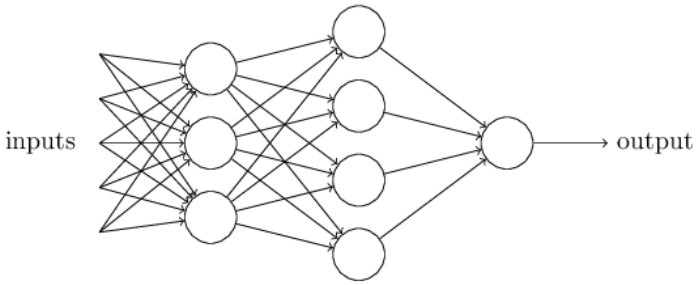


Figura 2.3: Red neuronal formada por neuronas [31]

símbolo *r* expresa la recompensa asociada a realizar la acción *a*, la cual depende del propio entorno, utilizándose para guiar el aprendizaje del agente. En lo que respecta a la expresión *Q(s, a)*, esta especifica el valor-Q que asociado al par estado *s* y acción *a*, mientras que *maxQ(s', a')* representa la máxima recompensa que se puede obtener en el nuevo estado *s'* que ha resultado de realizar la acción *a*.

2.5. Redes neuronales

Las redes neuronales se propusieron por primera vez en 1944 por Warren McCullough y Walter Pitts. Hasta 1969 fueron un campo muy amplio de investigación, pero cayó en decadencia hasta prácticamente su abandono. En los últimos años se ha convertido en una tecnología muy popular, en su mayor parte gracias a que son capaces de resolver problemas que hasta hace poco eran impensables. Esto es debido, principalmente, a la cantidad de datos disponibles, la capacidad de cómputo actual y las nuevas arquitecturas de redes neuronales que han surgido para trabajar con grandes cantidades de datos [30]. Gracias a todo ello, los algoritmos de aprendizaje profundo son los más populares, ya que, a diferencia de muchos de los algoritmos tradicionales de aprendizaje automático, el aprendizaje profundo mejora su rendimiento al poder acceder a un mayor número de datos.

Las redes neuronales son un grupo interconectado de neuronas o nodos compuestos por capas: primero una capa de entrada, seguida de una o más capas ocultas y, finalmente, una capa de salida. Cada uno de los nodos está conectado a los siguientes y tiene un peso asociado a él. Si la salida de un nodo está por encima de un valor especificado, envía datos a la siguiente capa de la red neuronal (figura 2.3).

Cada nodo puede entenderse como un modelo de regresión lineal compuesto por datos de entrada, pesos y salidas. Las entradas se multiplican por sus respectivos pesos y acto seguido pasan por una función de activación, la cual determina la salida de ese nodo.

2.5.1. Perceptrones y funciones de activación

Un perceptrón [28], recibe una serie de entradas y produce una única salida. El número de entradas puede variar, pero lo importante es la introducción de los pesos, los cuales son números asociados a la importancia de cada entrada. La salida del perceptrón es el resultado de aplicar una función de activación a la combinación lineal de las entradas y de los pesos internos del perceptrón.

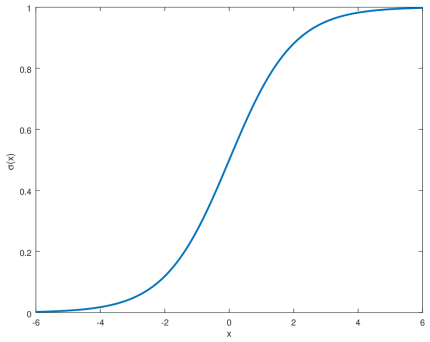
El objetivo es crear una red de perceptrones que sea capaz de aprender a ajustar de forma correcta sus pesos. Por ejemplo, si se asume que esta red es capaz de clasificar si un correo electrónico es *spam* o no, y se asume que se ha encontrado una manera de usar correos como entrada a esta red, se busca cambiar de forma mínima los pesos para intentar que la red mejore en sus predicciones. En otras palabras, se quiere que un cambio pequeño en los pesos de la red de perceptrones implique un pequeño cambio en la salida de esta red, de esta manera los cambios en las salidas no son bruscos.

El problema se encuentra en que se necesita una función de activación adecuada para obtener estos resultados. Al hacer un cambio mínimo en los pesos de la red, puede suceder que la salida de la red cambie de forma drástica dependiendo de la función de activación que se use o si no se usa ninguna. La función de activación es imprescindible para que la salida no sea lineal con respecto a las entradas. Sin función de activación, al apilar capas de neuronas sólo se calcularían combinaciones lineales de combinaciones lineales y eso vuelve a ser otra combinación lineal.

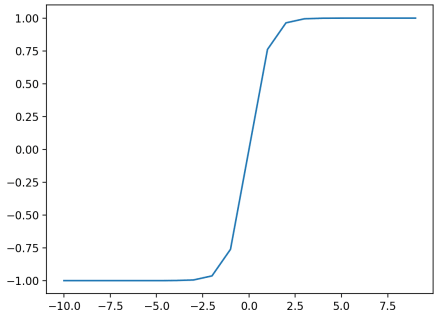
Una de las funciones de activación más usadas es la sigmoide [20]. En estas, un cambio mínimo en sus pesos causa un cambio equivalente en la salida. Al igual que los perceptrones, las neuronas que usan función de activación sigmoide tienen una o más entradas, estas pueden ser cualquier número entre 0 y 1. Además, las neuronas sigmoides tienen asociados unos pesos y un sesgo. La salida de la neurona sigmoide es igual a $\sigma((w \cdot x) + b)$, donde σ es la función sigmoide que viene definida por la ecuación 2.8, y $(w \cdot x)$ representa el sumatorio de las entradas multiplicadas por sus pesos.

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{2.8}$$

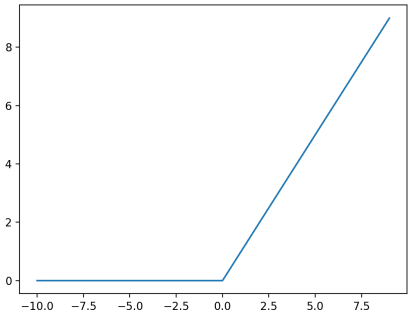
En los últimos años cada vez se ha hecho más popular la función de activación *rectified linear function* (*ReLU*). Es relativamente sencilla, como podemos ver en la figura 2.4c. Se define como $R(z) = max(0,z)$, lo que se traduce a que *ReLU* permite el paso de todos los valores positivos sin cambiarlos y todos los valores negativos se cambian a cero. Actualmente, prácticamente cualquier red neuronal utiliza esta función de activación [17]. Esta popularidad respecto a la función de activación sigmoide se debe a que es computacionalmente más eficiente y resuelve el problema de desvanecimiento de gradiente [18] (en algunos casos, el gradiente se va desvaneciendo a valores muy pequeños, impidiendo eficazmente que el peso pueda cambiar su valor. Esto puede llegar impedir el correcto entrenamiento de una red neuronal).



(a) Función Sigmoide [1]



(b) Función tanh [2]



(c) Función *ReLU* [3]

Figura 2.4: Función *ReLU*, Sigmoide y tanh

2.5.2. Descenso de Gradiente

Las neuronas de una red neuronal pueden aprender a ajustar los costes. Para ello, una opción muy usada es la técnica del descenso de gradiente, la cual se usa principalmente para resolver problemas de minimización.

La idea principal consiste en que existiendo una función J($\bar{\theta}$) se cambien los pesos hasta encontrar el mínimo de la función y converger en este, como se puede ver en la ecuación 2.9, donde α indica la tasa de aprendizaje. Esta tasa se refiere a cómo de rápido será el descenso de gradiente. Si α es demasiado pequeña es muy lento pero si es demasiado grande se podrá fallar en converger.

$$\bar{\theta} := \bar{\theta} - \alpha \cdot \frac{\delta}{\delta \bar{\theta}} \cdot J(\bar{\theta}) \tag{2.9}$$

Por lo tanto, el entrenamiento de una red neuronal consiste en minimizar una función de pérdida *loss* que mide el error que comete la red. También puede ocurrir que se encuentre el mínimo de una función local y no el mínimo global.

[2] https://hvidberrrg.github.io/deep_learning/activation_functions/sigmoid_function_and_derivative.html

[3] https://machinelearningmastery.com/choose-an-activation-function-for-deep-learning

[4] https://machinelearningmastery.com/choose-an-activation-function-for-deep-learning

Hay otros algoritmos que hacen un mejor trabajo que el de descenso de gradiente y muchos de ellos nacen a partir de la misma idea. Uno de estos es el algoritmo de optimización *Adam*, el cual es una extensión de descenso de gradiente, que, además, se ha convertido en el más popular a la hora de usar aprendizaje profundo.

Adam se presentó por Diederik Kingma y Jimmy Ba [21]. El desceno de gradiente clásico mantiene una única tasa de aprendizaje para todas las actualizaciones de los pesos de la red neuronal y esta tampoco varía durante el aprendizaje. Adam se basa en la combinación de otras dos extensiones de descenso de gradiente.

Combina los beneficios tanto de *Adaptive Gradient Algorithm* [26] como de *Root Mean Square Propagation* [23], pero, a diferencia del último, recoge más datos para saber de que manera debería de cambiar el índice de aprendizaje. En concreto *Adam* calcula el promedio móvil exponencial del gradiente y el cuadrado del gradiente. En otras palabras, *Adam* no sólo se fija en cómo varía el gradiente en un momento específico, sino que estudia cómo varía en el tiempo.

Por último, los algoritmos de descenso de gradiente suelen utilizar la retropropagación. Este método primero calcula las derivadas del error de la capa de salida de la red. Después, las propaga hacia atrás pasando por el resto de capas de la red y ajusta los pesos para avanzar en el aprendizaje.

2.5.3. Estructura de una red neuronal

Como se ha mencionado, un perceptrón tiene una única salida que se expande como entrada de cada uno de los perceptrones de la siguiente capa (figura 2.3). De esta manera, si se une un número considerable de capas de perceptrones, se puede tener un sistema de toma de decisiones muy complejo.

El problema que se puede encontrar es que todos estos pesos se tienen que definir y, si se desea crear un sistema medianamente complejo, puede llegar a ser una tarea complicada. Por suerte, existen algoritmos de aprendizaje automático que se pueden usar para que los pesos se ajusten automáticamente y así se puedan crear redes que aprenden a resolver problemas por su cuenta.

Las redes neuronales están formadas por capas de neuronas, de las cuales, usando como referencia la figura 2.3, la capa que se encuentra gráficamente más a la izquierda es denominada capa de entrada, la que se sitúa en el extremo de la derecha se llama capa de salida y las capas intermedias se llaman capas ocultas.

Una red neuronal puede tener una o varias capas ocultas, dependiendo de la complejidad del problema, pero diseñar las capas de entrada y de salida es más sencillo. Haciendo nuevamente uso del ejemplo de la red neuronal que clasifica si un correo es *spam* o no, se podrían crear tantos nodos de entrada como palabras clave aparecen en un correo de *spam* y, por ejemplo, la capa de salida podría estar compuesta de un único nodo. Así pues, si este valor de salida es mayor que un cierto valor, se puede asumir que el correo es de *spam*, en cambio, si es menor que ese valor, se puede asumir lo contrario.

Sin embargo, el diseño de las capas ocultas de una red neuronal resulta más difícil [31]. Primero de todo, hay que decidir el número de capas ocultas que se van a querer en la red neuronal y luego cuantos nodos tendrá cada una de estas, pero no es ciencia exacta cuantas capas y cuantos nodos ocultos debemos añadir para resolver el problema con el que se esté trabajando. En la gran mayoría de ocasiones, suele ser más sencillo probar con distintas combinaciones hasta encontrar con una que resuelva el problema satisfactoriamente. En este trabajo se experimenta con un distinto número de capas ocultas dependiendo del entorno para facilitar el aprendizaje, como se explica más adelante.

2.6. Deep Q-Learning

En 2013, DeepMind presentó una nueva variante del aprendizaje por refuerzo que llamaron Deep Q-Learning [29], la cual podía aprender a jugar a diferentes juegos de la Atari 2600 recibiendo únicamente imágenes de la pantalla como entrada y recompensas cuando cambiaba la puntuación del juego.

Más tarde, Google adquirió DeepMind por 500 millones de dólares. Desde entonces, DeepMind continúa mejorando Deep Q-Learning en entornos altamente complejos como StarCraft II. Además es uno de los muchos algoritmos que combinan el aprendizaje profundo y el aprendizaje por refuerzo [19].

2.6.1. Algoritmo

Deep Q-Learning es un algoritmo de aprendizaje por refuerzo que usa redes neuronales para predecir valores-Q. La red neuronal reemplaza la tabla-Q que usaba Q-Learning (sección 2.4). Igual que en el anterior algoritmo, se cuenta con un conjunto de estados *S* y, para cada uno de estos estados, existe la acción *a*, la cual viene determinada por la máxima salida de la red neuronal o por otras políticas que se verán más adelante. Actualizar un *valor-Q* debe ser equivalente a la retropropagación. Por lo tanto, debemos calcular el valor de la función de pérdida la cuál se puede ver en la ecuación 2.10. Una vez se conoce la pérdida se puede usar descenso de gradiente u otra técnica para minimizarla.

$$Loss = (r + \gamma \cdot maxQ(s', a'; \theta') - Q(s, a; \theta))^2 \tag{2.10}$$

La idea de usar una red neuronal para sustituir la tabla-Q como se puede apreciar en la imagen 2.5. En Q-Learning como ya se ha estudiado, la tabla-Q es una discretización de una función que asocia valores esperados de recompensas a pares de estado y acción. Usar una red neuronal para aproximar esta función permite generalizar lo que ya se ha aprendido, a un par estado y acción que sea nuevo. En otras palabras al usar una red neuronal no se tiene por qué haber explorado todos y cada uno de los pares estado y acción. Esto permite modelar problemas donde el espacio de estados y acciones no se

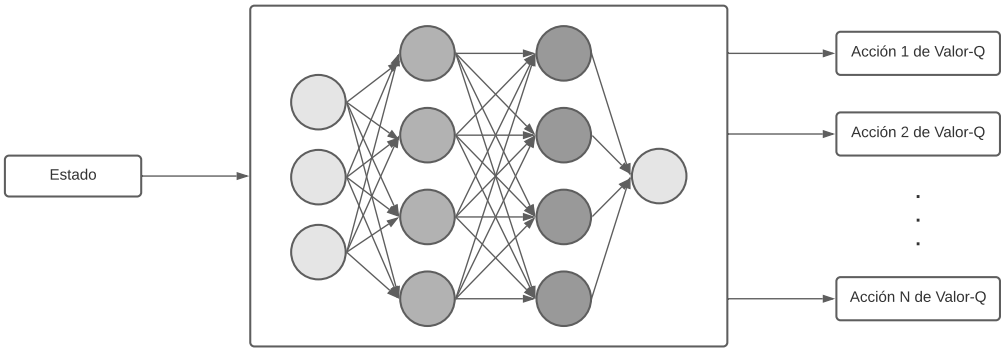


Figura 2.5: Esquema del funcionamiento de la red neuronal en Deep Q-Learning [4]

pueda representar explícitamente en memoria y, aunque se pudiera, sería muy difícil pasar por todos y cada uno de los pares estado y acción.

Si se quisiera modelar un problema en el que se ven envueltas distancias, con Q-Learning se tendrían que especificar todas y cada una de las distancias posibles, en cambio en Deep Q-Learning al usar redes neuronales, la distancia sería la entrada de estas, no habría que especificar de antemano cuales van a ser las distancias y no se tendrían que explorar todas y cada una de ellas por lo que se ha explicado anteriormente.

2.6.2. Divergencia y sobreajuste

La idea de usar una red neuronal para aproximar una función Q no es original de los creadores de Deep Q-Learning, anteriormente se usó esta idea sin mucho éxito porque el algoritmo no llegaba a converger y a veces olvidaba todo lo que había aprendido [29].

Deep Q-Learning recibe como entrada un estado S y devuelve un *valorQ* asociado a cada acción A posible. La novedad de este algoritmo es que usa dos redes neuronales como se puede ver en la imagen 2.6 para evitar divergencia en el aprendizaje. Con la primera red neuronal se calcula el valor de predicción *valorQ(S,A)* y con la segunda, el valor objetivo $(r + \gamma \cdot maxQ(S', A))$ como se puede ver en la ecuación 2.10. Si sólo se emplea una red neuronal para calcular tanto el valor objetivo como el de predicción, se podrían empezar a predecir valores demasiado grandes para una pareja de estado-acción, lo que derivaría en destruir la calidad del aprendizaje y, por lo tanto, se produciría divergencia.

Para resolver este problema, como se ha mencionado anteriormente, se tiene una segunda red neuronal que se encarga de predecir los valores objetivo. Esta segunda red neuronal tiene la misma estructura que la primera, con la diferencia de que sólo se actualizan con frecuencia los pesos de la primera y cada *C* entrenamientos se copian los pesos de la primera red en la segunda, para actualizarla. Esto nos lleva a un entrena-

[4] https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/

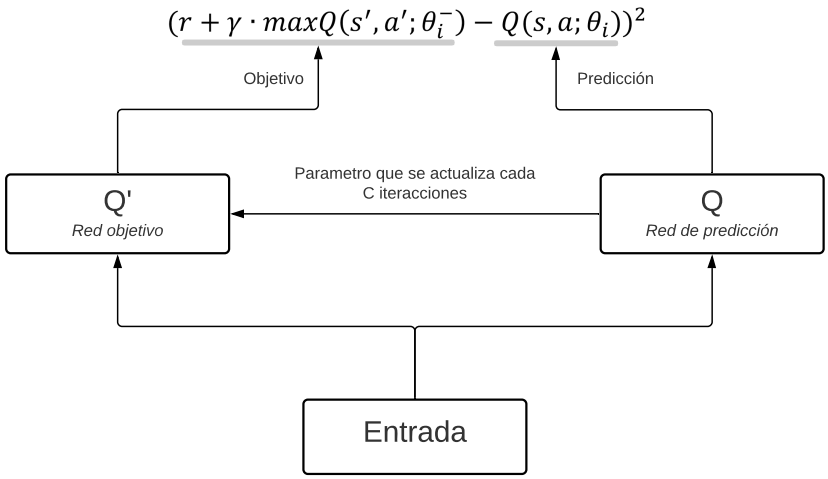


Figura 2.6: Predicción y objetivo[5]

miento más estable y por lo tanto es más probable que converja. El funcionamiento es el siguiente. Primero se inicializan las dos redes neuronales con los mismos pesos aleatorios. Luego se ejecuta la acción *a* desde un estado *s* y se observa el estado *s'*, al que lleva la acción elegida, para calcular la recompensa *r*. Acto seguido, cuando se cuente con estas variables, se almacena esta experiencia *(s, a, s',r)*. Una vez se tienen suficientes experiencias acumuladas, se calculan los *valores-Q(s,a)* de un grupo aleatorio de estas experiencias para finalmente actualizar la red neuronal con el objetivo de que prediga mejor los futuros casos.

La otra medida se conoce por *experience replay*. Como se ha mencionado antes, se elige un grupo de casos aleatorios de los que se tienen almacenados en un *batch* para entrenar a la red neuronal. No se adquieren todos los casos ya que es muy posible que los estados *S* y acciones *A* sean muy parecidos. Esto es algo que puede llevar al agente a no entender el entorno en el que se encuentra, ya que al aprender un número muy elevado de casos similares se traducirá en un sobreajuste de los pesos de la red neuronal.

En resumen, es de vital importancia a la hora del aprendizaje del agente almacenar los conjuntos de estado actual, acción, nuevo estado y recompensa, *(s, a, s',r)* y cuando se tenga un número suficientemente grande de experiencias, empezar a tomar muestras aleatorias de este. También es una idea muy interesante que una vez se han almacenado muchos casos se eliminen poco a poco los más antiguos [32]. Con esto potencialmente se evita que el agente aprenda con una misma experiencia más veces de las necesarias y además indirectamente se prioriza el aprendizaje de experiencias más recientes. También es importante entrenar con un conjunto de casos al mismo tiempo y no con un único caso cada vez. De esta manera, en vez de hacer que la red neuronal se sobreajuste a cada caso, se consigue un aprendizaje más estable gracias al entrenamiento con un lote de casos potencialmente diferentes.

[5]https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/

Capítulo 3

Starcraft II

En el presente trabajo se utiliza el entorno del videojuego de estrategia en tiempo real llamado *StarCraft II* para el desarrollo de jugadores automáticos que utilizan aprendizaje por refuerzo. Este videojuego aporta un complejo entorno y un alto número de estados y posibles acciones. Además, surgen distintas estrategias de combate dependiendo de la raza elegida en dicho juego, originando un desafío a la hora de desarrollar en este entorno jugadores controlados por ordenador que empleen aprendizaje automático.

Para hacer uso del entorno de *StarCraft II*, se emplea el componente desarrollado por *DeepMind* en colaboración con *Blizzard Entertainment*, *PySC2*. Este componente ofrece un entorno de aprendizaje sobre el videojuego de *StarCraft II*, llamado en inglés *"StarCraft II Learning Environment"* (SC2LE).

El entorno de este videojuego ha tenido una gran repercusión en el mundo del aprendizaje automático, siendo capaz de revolucionar este campo [41]. Por otro lado, hay entornos más sencillos dentro de *StarCraft II* llamados minijuegos, proporcionados por *PySC2*, en los cuales se pueden aplicar distintas técnicas de aprendizaje automático para que los jugadores controlados por ordenador logren aprender a realizar un objetivo específico.

3.1. Juegos de estrategia en tiempo real

Los juegos de estrategia en tiempo real o RTS, por sus siglas en inglés, son un género de videojuegos en el cual dos o más jugadores se enfrentan unos a otros en un escenario, evitando ser destruidos hasta que sólo quede uno. La economía en el enfrentamiento permite realizar la gran mayoría de acciones mediante el uso de los recursos distribuidos por el escenario. Por regla general, en todos los RTS se pueden construir edificios usando los recursos recolectados, lo que habilita reclutar unidades controlables que permiten eliminar al adversario. Algunos de los RTS más conocidos son: *Age of Empires* [1], basado en un enfoque histórico que permite al jugador avanzar por distintas épocas históricas y sus sucesivas civilizaciones; y *StarCraft II*, que se

desarrolla en un futuro tecnológico.

3.1.1. Características generales

La característica principal y diferenciadora de los juegos RTS es, como indica su propio nombre, que se desarrollan en tiempo real, es decir, no hay turnos, creando infinidad de posibilidades respecto a juegos de estrategia clásicos como pueden ser el ajedrez o videojuegos de cartas como *HearthStone*.

Los juegos RTS se diferencian además por incluir acciones durativas que tardan cierto tiempo en completarse. Por ejemplo, para reclutar una unidad en un edificio tan sólo es necesario realizar una acción sencilla, pero, para que la unidad pase a estar disponible debe pasar cierto tiempo. Del mismo modo, otras acciones como construir edificios o desarrollar tecnologías también requieren del paso de un periodo de tiempo para ser completadas.

Otra característica fundamental de los juegos de estrategia en tiempo real es la llamada *niebla de guerra*. En estos casos, el escenario se presenta cubierto de niebla, lo que provoca que el jugador sólo pueda conocer lo que está ocurriendo en las zonas del mapa sobre los que tiene visión, ya sea a través de unidades, edificios u otros medios que ofrece el videojuego. Este elemento aporta un mayor grado de complejidad a los RTS por la incertidumbre de no conocer el estado del adversario y la necesidad de explorar continuamente el terreno para ir adquiriendo información sobre el estado de la partida.

Además, los videojuegos en tiempo real crean la posibilidad de ejecutar acciones de manera simultánea. Esto puede desembocar en que dos jugadores realicen acciones incompatibles al mismo tiempo, como crear dos edificios en un mismo lugar. Por ello es necesario incluir en este tipo de videojuegos políticas que permitan resolver estas situaciones.

En un videojuego RTS, el conjunto de estados posibles puede llegar a ser elevado, superando por varios órdenes de magnitud a otros juegos de estrategia por turnos. El elevado número de unidades que puede llegar a haber en una partida al mismo tiempo unido a la amplia libertad a la hora de tomar decisiones provoca que, en este espacio, el factor de ramificación de un árbol de búsqueda pueda llegar a ser inabarcable. Es por ello que el estudio de los RTS se ha erguido como uno de los campos de investigación más interesantes dentro de la inteligencia artificial.

3.1.2. Desarrollo de una partida

Una partida estándar de un RTS tiende a desarrollarse de una manera parecida y, aunque puede haber distinto número de jugadores, para la presente explicación vamos a asumir que sólo existen dos jugadores enfrentados entre sí. El objetivo de los jugadores es destruir por completo la base del jugador enemigo, eliminando la posibilidad de que siga expandiéndose. Al comenzar la partida, ambos jugadores aparecen con un

edificio que les permite crear trabajadores para poder expandirse. Asimismo, la partida termina cuando uno de los jugadores consigue destruir todos los edificios que generan las unidades y no queda vivo ningún trabajador que pueda volver a construirlos.

Lo primero a tener en cuenta al iniciar la partida es el mapa en el que se desarrolla la batalla, pues dependiendo del mismo puede haber, por ejemplo, rampas que nos permitan acceder a una zona alta más fácil de defender. Además, distintos mapas pueden estar más o menos enfocados a ciertas unidades militares. Por otro lado, en función de lo grande que sea el mapa, puede resultar más beneficioso expandir el terreno que se controla.

Al iniciar una partida, el jugador aparece con una base principal y algunos trabajadores en uno de los extremos del mapa. Los trabajadores pueden extraer recursos, uno de los elementos más importantes de cualquier partida ya que, como se ha explicado anteriormente, con estos se pueden construir unidades, otros edificios o más trabajadores para recolectar más rápido nuevos recursos. Por lo tanto, independientemente de la estrategia que se quiera seguir, el foco principal del jugador al principio de la partida ha de ser la creación de trabajadores, pues mantener un nivel elevado de recursos puede ser clave. Además, esto le permite recolectar recursos a mayor velocidad, crear los edificios más básicos y generar las primeras unidades militares.

Por tanto, los recursos son clave para realizar acciones fundamentales dentro de la partida. En primer lugar, en cualquier partida hay que construir algunos edificios esenciales, como pueden ser los que te permiten tener un mayor número de unidades o los que desbloquean la creación de nuevos tipos de soldados, lo que no impide que puedan existir una gran variedad de edificios con otras funciones. Una vez se han adquirido recursos suficientes y construido los edificios pertinentes, la creación de un ejército capaz de destruir la base del adversario es fundamental para lograr el objetivo de la partida.

Cuando el jugador logra una primera base bien establecida, el siguiente paso es explorar el mapa para encontrar la base del enemigo así como nuevos recursos que le permitan, potencialmente, establecer una segunda base. Después de esta primera etapa inicial, comienzan los primeros enfrentamientos entre los jugadores, pues es común enviar unidades a explorar el escenario para encontrar la ubicación de la base enemiga. Este resulta también un buen momento para investigar mejoras militares y económicas que aporten una ventaja sobre el adversario.

Los trabajadores tienen la función de recolectar recursos y llevarlos a la base. Tener una base alejada de la fuente de recursos supone que tengan que recorrer largas distancias, lo que conlleva tanto una pérdida de tiempo como un peligro. Si el jugador decidiese no expandirse a una segunda base, ha de tener en cuenta que el jugador enemigo probablemente sí lo haga, obteniendo recursos el doble de rápido que él. Queda así puesto de manifiesto la importancia de la economía de la partida, pues cuanto mayor es la economía, más fácil es construir estructuras defensivas, mejorar unidades o crear ejércitos consecutivos que se pueden lanzar contra el enemigo.

Si ambos jugadores consiguen resistir los ataques iniciales del otro, se pasa a una etapa en la que se desbloquean todas la unidades avanzadas, centrando todos sus recursos en la creación de ejércitos que les permitan eliminar por completo al enemigo.

Por último, se puede alcanzar una etapa de la partida en la que todos los recursos del escenario hayan sido agotados y resulta por tanto vital no malgastarlos. Finalmente, en algún momento, uno de los jugadores termina siendo eliminado por el otro, dando fin a la partida.

3.1.3. Estrategias de juego aplicadas a RTS

Durante una partida se deben tomar varias decisiones a distintos niveles del juego [33], los cuales son:

- Decisiones estratégicas: Consisten en el conjunto de decisiones que hay que tomar a largo plazo durante una partida y que, por lo tanto, afectarán en una escala mayor. Por ejemplo, en los RTS se pueden mejorar las unidades para que a partir de ese momento todas la unidades creadas reciban esa mejora o se pueden invertir recursos para construir edificios defensivos que mejoren la situación defensiva de nuestra base.
- Decisiones tácticas: Con estas decisiones se decide cómo se quiere aplicar la estrategia en cada momento de la partida, por lo que son decisiones a medio plazo. Por ejemplo, una decisión táctica sería no construir estructuras defensivas en los primeros momentos de la partida ya que se puede decidir que seguir una estrategia de expansión puede ser mucho más efectivo en ese momento.
- Decisiones reactivas: Por último se tienen las decisiones que son de pequeña escala y a corto plazo. Por ejemplo, cómo se mueven las unidades en una pelea o cómo se usan las habilidades activas de las unidades.

Todas estas decisiones deben de tomarse en tiempo real, por lo que se puede apreciar la complejidad que puede llegar a tener un RTS, de lo que se hablará más adelante. En el ámbito profesional de los RTS hay otra manera de clasificar estas decisiones: el microjuego y el macrojuego.

El microjuego se refiere a todas las acciones que son de pequeña escala [27], como el control de unidades (tanto su posicionamiento como el uso de sus habilidades). Un ejemplo muy claro de microjuego es el *kiting*. Este término se refiere a huir al mismo tiempo que se ataca. Si se está luchando con unidades a distancia contra unidades cuerpo a cuerpo un jugador profesional estará alejando a sus unidades a distancia en vez de dejarlas estáticas. Con esto conseguirá recibir menos ataques de la unidad cuerpo a cuerpo enemiga. Este tipo de acciones son lo que diferencia un jugador profesional de uno casual, ya que el jugador profesional al mismo tiempo que realiza una batalla también sigue construyendo edificios, generando más unidades y empleando la técnica *kiting* en el transcurso de la batalla. En cambio, un jugador casual para estar realizando *kiting* en una batalla tiene que poner toda su concentración en realizar esta acción.

El macrojuego abarca todas las acciones que tienen un beneficio a largo plazo, cuya selección afectan al aprendizaje por refuerzo, requiriendo realizar adaptaciones

especificas [46]. Algún ejemplo de este tipo de acciones es mantener un buena reserva de recursos, decidir si es un buen momento para explorar el mapa o si por ejemplo puede ser una buena decisión realizar un ataque a la base del enemigo.

Como se puede ver, el macrojuego se asemeja a las decisiones estratégicas y tácticas. Por otro lado, el microjuego es muy similar a las decisiones reactivas explicadas anteriormente.

Este trabajo se enfocará en el microjuego, donde se puede estudiar con más profundidad la técnica del *kiting* (capítulo 6) que, como se ha dicho anteriormente, puede suponer la diferencia entre salir victorioso de una batalla o no. Al igual que con el microjuego, se dedicará todo un capítulo a la resolución de un problema relacionado con el macrojuego de *StarCraft II* (capítulo 7).

Se pueden considerar al menos tres estrategias muy diferenciadas unas de otras [45, 15, 13].

- Estrategia de mejora: Esta estrategia se considera la básica que cualquier jugador principiante seguirá. Consiste en desarrollar la base principal en la que aparece el jugador al iniciar la partida. A medida que se obtienen recursos el jugador va desarrollando tecnologías que mejoren las unidades, creando nuevos edificios que le permitan crear unidades más avanzadas y, poco a poco, ir construyendo un ejército con el poder suficiente como para poder acabar con el enemigo. Esta estrategia en un principio puede parecer eficaz pero hay muchas maneras de que pueda salir mal.

- Estrategia de *Rush*: En esta ocasión se busca castigar al contrincante continuamente para no dejarle tiempo a respirar. El jugador intenta construir lo antes posible un edificio donde crear unidades ofensivas. Una vez cuente con un pequeño escuadrón de unidades, las manda a buscar la base enemiga y atacarla. De esta forma se buscan varios beneficios. Si el jugador no está muy experimentado puede no tener con qué defenderse de un ataque tan temprano, pudiendo llegar a perder los trabajadores y acabar en un resultado fatal. En cambio, si sí que es capaz de defenderse, tampoco importa ya que en breve estará llegando otra oleada de unidades a su base. De esta manera, hostigando de forma continuada al enemigo, este pierde la oportunidad de realizar muchas otras tareas que son importantes y se obtiene una ventaja considerable contra el rival [40].

- Estrategia de expansión: Por último, nos encontramos con una estrategia que a diferencia de la primera no trata de desarrollar nuestra base principal. Aquí se prioriza establecer lo más rápido posible otras bases en otras zonas del mapa y empezar a recoger recursos en esas nuevas zonas. Así se consigue una ventaja económica importante de una manera extremadamente rápida ya que una vez se tiene al menos una base más donde extraer recursos, se hace al doble de velocidad que nuestro enemigo.

Estas estrategias se contrarrestan en mayor o menor medida entre ellas. Por ejemplo, la estrategia de *Rush* puede molestar mucho a un jugador que intente mejorar su

Figura 3.1: Imagen del juego *StarCraft II*

base lo antes posible, ya que recibir ataques continuamente puede significar la pérdida de edificios y por lo tanto la pérdida de mucho tiempo. En cambio, la estrategia de expansión si se plantea bien puede contrarrestar a la de *Rush*, ya que aunque se pierda un poco de ventaja en la base principal ya se tienen otras que están funcionando en otras áreas del mapa.

Los jugadores profesionales de videojuegos RTS no se ciñen a ninguna de estas estrategias en concreto. Un jugador de alto nivel es capaz de combinar todas estas estrategias al mismo tiempo. Mientras mejora la base envía unidades a buscar y hostigar la base enemiga y también empieza a crear nuevas bases en distintas zonas del mapa.

Este nivel solo se puede encontrar en altas competiciones como los *eSports*, los cuales son deportes electrónicos en los que los jugadores de más alto nivel compiten. Un jugador estándar en un juego RTS realiza de unas 60 a 150 acciones por minuto. Una acción se considera el realizar cualquier movimiento dentro del juego. Seleccionar una unidad sería una acción y ordenar que se mueva a cierto punto sería otra. En el caso del videojuego *StarCraft II*, que es extremadamente complejo y requiere de años de práctica para ser dominado, un jugador profesional realiza entre 400 o 500 acciones por minuto, llegando al pico de 800 acciones por minuto en algunos momentos límites de las partidas.

3.2. StarCraft II

StarCraft II es un videojuego de estrategia en tiempo real (RTS) desarrollado por *Blizzard Entertainment*. Siendo la secuela del juego *StarCraft*, su fecha de lanzamiento fue el 27 de julio de 2010, aunque su desarrollo ha continuado a través de actualizaciones y expansiones hasta el propio 2020. Continúa la historia de su antecesor, cuatro años después del final de *StarCraft: Brood War* (expansión del videojuego original). Narra el destino de las tres razas habitantes del universo de *StarCraft*: los *terran*, humanos exiliados de la Tierra; los *protoss*, una raza tecnológicamente avanzada con poderes

psiónicos; y los *Zerg*, una especie de vida organizada en enjambres (apartado 3.2.2). El juego está dividido en tres capítulos, uno incluido en el juego original y los otros dos añadidos posteriormente mediante expansiones.

El videojuego consiste en el desarrollo de una campaña (para un jugador) en la que se narra la historia del videojuego a la vez que se realizan enfrentamientos contra la inteligencia artificial. Además existe la opción de luchar contra la inteligencia artificial en diversos escenarios y con distintas configuraciones, dando a los jugadores la oportunidad de mejorar sus habilidades.

StarCraft II también cuenta con el modo multijugador, donde el usuario puede poner a prueba su destreza en combate contra otros jugadores en línea. El enfrentamiento por batallas puede darse entre dos o más jugadores. Cada jugador puede elegir a una de las tres razas, las cuales pueden estar repetidas en el enfrentamiento, permitiendo que más de un jugador pueda seleccionar la misma raza sin problemas. En cada batalla se debe estar pendiente de distintas tareas, como son la gestión de recursos, los combates y la creación de edificios y unidades. Cada jugador cuenta con marcadores indicando el número de recursos que tiene en ese momento, los cuales se adquieren con los trabajadores encargados de recogerlos. Con estos recursos se pueden construir nuevos edificios para adquirir nuevos tipos de unidades y aumentar la fuerza del jugador en cuestión, con el objetivo final de crear un ejército con el que poder luchar contra el rival y destruir su base para ganar la partida.

A pesar de su apariencia 3D, *StarCraft II* sigue una lógica 2D. Cuenta con una cámara cenital la cual permite tener una vista de pájaro sobre el campo de batalla. Esta cámara se puede mover con el teclado, mientras que con el ratón se pueden seleccionar tanto unidades como edificios. Las unidades seleccionadas pueden ser movidas y utilizar sus funciones específicas, al igual que con los edificios que cuenten con acciones posibles.

3.2.1. Recursos

StarCraft II cuenta con tres tipos de recursos. Dos de ellos pueden ser recolectados por la unidad trabajadora de cada raza, que es la encargada tanto de recolectarlos como de construir los edificios. Estos recursos son:

- Minerales. Se utilizan para crear todas las estructuras, las unidades y las mejoras, por lo que su extracción es clave en toda estrategia. Estos se encuentran en yacimientos de color azulado, lugar donde se debe enviar a la unidad trabajadora para obtenerlos. Pueden aparecer en distintas zonas del mapa.
- Gas vespeno. Empleado para crear unidades y estructuras más avanzadas, además de ser requerido para todas las mejoras. Se puede encontrar en géiseres, usualmente localizados cerca de los yacimientos de minerales. A diferencia de los minerales, para recolectarlos es necesario crear previamente un edificio en el propio géiser. Una vez creado este edificio, se pueden enviar trabajadores de la misma forma que se hace con los minerales.

- Suministros. Es un recurso especial que no puede ser recolectado y cuya funcionalidad es limitar el número de unidades que podemos tener creadas al mismo tiempo. Para aumentar la cantidad de suministros se debe crear un edificio especial que se encargue únicamente de esta tarea. Cabe remarcar que, aunque estos recursos limitan el número de unidades, no significa que este sea un límite de población, pues cada unidad puede consumir un número distinto de suministros a la hora de ser creados. Cuando se alcanza el tope máximo, la fabricación de unidades se interrumpe hasta que el jugador vuelve a tener suministros disponibles.

3.2.2. Razas

A la hora de empezar una partida, el jugador puede elegir una raza con la que jugar. Cada una de las razas cuenta con unidades y edificios especiales, los cuales se van desbloqueando a lo largo de las partidas, marcando las diferentes estrategias de combate que hay entre ellas. Las posibles razas son las siguientes:

- *Terran*. Representan a la raza humana, descendiente de una expedición de colonización que se lanzó desde la Tierra hace siglos.

  Su modo de juego es sencillo con respecto a las otras razas ya que pueden construir en cualquier parte del terreno. Deben construir edificios para mantener las unidades, reparar estos edificios y reparar las unidades mecánicas. Su peculiaridad reside en la capacidad de mover sus construcciones debido a sus sistemas antigravitatorios.

  Además, esta raza cuenta con una fuerza defensiva poderosa, teniendo edificios en los que pueden introducirse unidades y atacar a los enemigos que se acerquen.

  La unidad *Terran* de la que se hace uso en este trabajo es el *marine*. Esta unidad puede atacar a unidades aéreas y terrestres, gracias a su capacidad de atacar a distancia, y puede usar *Paquetes de Estimulantes*, los cuales mejoran su desempeño en combate durante unos segundos.

- *Zergs*. Esta raza fue creada por una raza antigua y es originaria del planeta Zerus. Son liderados por *La Reina de Espadas* y se caracterizan por ser un enjambre y, por lo tanto, ser numerosos y abrumadores en batalla.

  Su modo de juego es peculiar, pues no pueden edificar en ningún terreno que no sea biomateria *Zerg*. Esta biomateria puede expandirse haciendo uso de criaderos o utilizando a la reina. Además, sus edificaciones no se reparan, sino que se regeneran solas, aunque para construir edificios debes sacrificar a un constructor.

  A pesar de las limitaciones de construcción, es una raza que se expande muy rápidamente y cuyo coste de las unidades es mucho menor en comparación con las otras razas.

- *Protoss*. Es una raza ancestral y tecnológicamente avanzada que se caracteriza por sus poderes psiónicos. Fue creada por una raza antigua y es originaria del planeta *Aiur*.

Su modo de juego también tiene unas características especiales, ya que, al igual que los *zergs*, las unidades de esta raza no pueden edificar en cualquier sitio, sólo en las zonas psiónicas habilitadas por pilones. Sus unidades son muy poderosas y al mismo tiempo más costosas, lo que afectará a la estrategia en esta facción. Además, tanto las unidades como los edificios cuentan con escudos que les protegen del daño y una vez destruido empiezan a perder vida. Además, estos escudos se regeneran con el tiempo.

Las unidades *Protoss* de las que se hace uso en este trabajo son:

- Stalker: gracias a sus ataques a distancia, puede atacar tanto a unidades terrestres como aéreas. Además, puede hacer uso de su habilidad *Traslación* para moverse por el mapa.
- *Zealot*: siendo unidades cuerpo a cuerpo con gran velocidad de ataque, los *zealots* son débiles contra otras unidades que tengan un ataque a distancia, pero poderosos contra los que no.

3.2.3. Estrategias habituales para cada raza

Como ya se conoce, hay tres razas en *StarCraft II* y cada una tiene unas características propias que las hace afines a las distintas estrategias existentes.

- *Terran*: Por las propias características de la raza, en relativamente poco tiempo y sin mucho esfuerzo pueden llegar a tener la base principal muy bien fortificada. Además, por cómo funcionan sus unidades, requiere un nivel alto de microjuego, por lo que la técnica de *kiting* es esencial en esta raza al ser la mayor parte de sus unidades de ataque a distancia. Los *terran* reflejan la jugabilidad clásica de los juegos RTS. Además, al poder defender de forma relativamente sencilla nuestra base, llevar a cabo la estrategia de mejora puede resultar más sencillo que con otras razas.

- *Zerg*: Son opuestos a los *protoss* en cuanto a unidades se refiere. Al ser considerados un enjambre en la historia del videojuego, sus unidades individualmente son muy poco eficientes. Pero es relativamente sencillo tener un número elevado de unidades débiles al poco tiempo de comenzar la partida. Por lo tanto, esta raza es perfecta para estar continuamente hostigando al contrincante, ya que una vez se ‘pierden las unidades que se han enviado para molestar, ya se deben de tener varios grupos más de unidades débiles para continuar con los ataques.

- *Protoss*: esta raza está considerada la raza más sencilla del juego. La estrategia principal de los *protoss* es intentar alargar la partida, pues sus unidades más avanzadas son de las más poderosas de todo el juego. Por lo tanto, al principio de la partida pueden considerarse débiles, ya que sus unidades al ser muy poderosas son costosas en cuanto a recursos y puede ser más complicado formar un ejército considerable en poco tiempo. Esta raza necesita una economía fuerte para poder crear ejércitos imponentes, por lo que es de vital importancia saber expandirse por el mapa.

Aun así, *StarCraft II* está diseñado de tal manera que cualquier raza puede vencer a otra y que, aunque alguna tenga ciertas ventajas claras sobre otra de las razas, es un videojuego que está perfectamente equilibrado dejándolo todo a la habilidad de cada jugador.

3.3. Beneficios para el aprendizaje automático

Los RTS cuentan con entornos muy ricos y complejos para el desarrollo de jugadores automáticos basados en un aprendizaje automático. Esto se debe a varios factores y elementos que nos proporciona estos entornos. Uno de ellos es el extenso mapa del juego, el cual cuenta con la niebla de guerra en las zonas que no han sido exploradas y ofrece un nuevo reto a la hora de programar una inteligencia artificial [10]. Otro elemento es la cámara, la cual observa parcialmente el mapa y puede y debe ser movida por el propio jugador para poder recoger información del terreno de juego.

Además, el género de los RTS plantean problemas multiagente, pues, en cada partida, varios jugadores deben luchar por los recursos del mapa y expandirse por sus propios medios. Pero este problema multiagente también se extiende a bajo nivel, pues cada jugador debe manejar cientos de unidades. Cada una de estas unidades funciona de una manera distinta y tiene distintas estrategias de acción, necesitando colaborar en equipo con otras unidades en más de una ocasión y, en lo que respecta a tomar una decisión, generando un espacio de acciones muy extenso.

Asimismo, muchas de las acciones tomadas por el jugador no tienen un efecto inmediato, sino que se aprecia el impacto de la acción en el entorno más adelante en el tiempo de juego (acciones durativas). Esto genera un rico conjunto de desafíos en la asignación y exploración de recompensas a lo largo del tiempo, planteando un reto en cuanto a cómo enfocar distintos elementos del aprendizaje automático [2].

Si se quiere aplicar todo esto a un algoritmo de inteligencia artificial, surgen retos importantes. Como se ha explicado anteriormente, en los videojuegos RTS hay cientos de unidades que deben controlarse simultáneamente y esto se traduce en que debe de haber algún tipo de coordinación. Para resolver esto se podrían plantear varias soluciones, por ejemplo, en vez de haber un único jugador automático que controle todo (conocido como agente), podría haber varios que controlen aspectos determinados de una batalla. Si se tuviera un agente que se encargue de controlar todo lo relacionado con las batallas y otro que se encargue con todo lo relacionado con las construcciones se podría intentar coordinarlos para que sean muy eficientes. Otra aproximación podría ser tener un agente estratégico que se encargue de dar ciertas ordenes a los agentes menores, los cuales se encargarían de situaciones más específicas.

Como se ha comentado en la sección 3.1.3, un jugador de videojuegos RTS toma decisiones en tres niveles de abstracción distintos, a nivel estratégico, táctico y reactivo o, si se prefiere, a nivel de macrojuego y microjuego. Al igual que con el reto multiagente, este se puede intentar resolver con distintos agentes que controlen cada uno de los distintos niveles en los que hay que tomar decisiones. El agente estratégico podría

decidir si es un buen momento para atacar al enemigo y por debajo un agente reactivo sería el encargado de colocar a las unidades en las batallas y realizar la técnica de *kiting* y así mejorar la eficacia de las unidades en combate.

Al transcurrir los RTS en tiempo real, el algoritmo de inteligencia artificial que se plantee debe de tomar todas estas decisiones en el menor tiempo posible. En un videojuego por turnos un agente tiene mucho más tiempo para tomar una decisión, pero este no es el caso, ya que un pérdida de varios segundos puede suponer una desventaja respecto al adversario pues se requiere una toma de decisiones de forma continua.

Además se plantea el reto de la visibilidad parcial. Al comenzar la partida no se conoce prácticamente nada sobre el estado de la batalla. Cuando poco a poco se va desarrollando la partida, los estados se van volviendo más y más complejos debido a que se descubren nuevas zonas en el mapa. Los jugadores automáticos también deben de estar preparados para reaccionar a situaciones inesperadas. En los RTS, una táctica muy común es intentar conducir una fuerza considerable hasta la base del enemigo, lo que se traduce en un ataque sorpresa al que normalmente se tardará demasiado en reaccionar y por lo tanto pueda significar una derrota.

Como se ha visto anteriormente, la complejidad del juego es inmensa, generando una cantidad de estados posibles en el juego ingente. En una batalla de un RTS puede haber miles de estados distintos. Para poder entrenar a un agente de una gran magnitud se necesita un hardware de un nivel extremadamente alto y una gran cantidad de tiempo de entrenamiento para llegar a un resultado mínimamente digno. Al mismo tiempo, los jugadores profesionales realizan cientos de acciones por minuto. Si se quiere que un agente pueda igualar a estos jugadores, se tendrá que crear un agente pensando que por minuto va a realizar cientos de acciones y cualquier error en estas puede significar la derrota.

Por último, una batalla en el género RTS puede alargarse hasta una hora y esto implica poder encontrar más situaciones en las que el agente no sepa reaccionar. Al haber un gran número de situaciones en una misma partida, el agente debe aprender a aproximar de situaciones parecidas a las nuevas que se encuentre.

Debido a estos beneficios con los que cuenta el género RTS, existen diversas competiciones que permiten a distintos programadores poner a prueba su creatividad y habilidad para desarrollar jugadores automáticos que consigan derrotar a los demás. Desde 2010 se ha usado el videojuego original *StarCraft* en competiciones de jugadores automáticos [11]. A día de hoy sigue funcionando y cada vez hay más participantes, pudiendo encontrar diversos enfoques que se pueden aplicar al entorno de *StarCraft* [44]. También se ha llegado a intentar crear una inteligencia artificial aplicando aprendizaje por refuerzo para una partida completa [35]. En cuanto a StarCraft II, aunque es mucho más reciente ya hay competiciones donde se pueden poner a prueba los bots creados. En concreto, *SC2 AI Arena* es la plataforma que organiza todo lo relacionado con las competiciones. *Blizzard*, la desarrolladora del juego, fue consciente de esta capacidad de *StarCraft II* y, colaborando con *DeepMind*, crearon un complemento para este videojuego, aportando herramientas útiles en el aprendizaje automático, facilitando su uso para los usuarios que quieran crear jugadores automáticos: *PySC2*.

3.4. PySC2

StarCraft y *StarCraft II* cuentan con un gran número de jugadores compitiendo en torneos por más de veinte años. Además, el juego original se emplea para investigaciones sobre inteligencia artificial y los usos del aprendizaje automático, compitiendo anualmente en la competición de bots de AIIDE[1], haciendo que StarCraft siga siendo relevante en nuestros días. Asimismo, el hecho de que haya jugadores que compiten cada día hace que exista una base de datos de partidas muy extensa sobre la que poder aprender, al igual que una larga lista de agentes ya desarrollados.

PySC2 es un componente desarrollado en Python por *DeepMind* en colaboración con *Blizzard* [42] y aporta un entorno que facilita el aprendizaje de jugadores controlados por ordenador que hagan uso de métodos de aprendizaje automático en el videojuego de *StarCraft II*. Debido a esto, *PySC2* es un componente clave en este trabajo, permitiendo a los jugadores acceder a los estados del juego. Este componente cuenta con:

- Una API desarrollada por *Blizzard* que aporta a los desarrolladores e investigadores la posibilidad de manipular el juego. Además, por primera vez ofrecen herramientas para Linux, ya que el juego originalmente no existe en ese sistema operativo.
- Una base de datos de partidas ya jugadas por jugadores reales.
- Un conjunto de simples minijuegos que facilitan el desarrollo de bots que empleen aprendizaje por refuerzo, permitiendo a los desarrolladores probar sus agentes realizando tareas específicas.
- Un documento de *PySC2* que introduce el funcionamiento del entorno y expone los resultados conseguidos por *DeepMind* en cada minijuego, el aprendizaje supervisado haciendo uso de la base de datos y un minijuego completo contra una inteligencia artificial incorporada.

PySC2 aporta una interfaz sencilla y flexible enfocada al aprendizaje por refuerzo en *StarCraft II* (figura 3.2), aportando observaciones y acciones dependiendo del entorno, recompensas basadas en la puntuación del propio motor del juego contra una inteligencia artificial previamente generada; y un conjunto de minijuegos con objetivos específicos que podemos encontrar en partidas de *StarCraft II*. Tal y como se observa en la figura 3.2, esta interfaz permite aislar el juego en distintas capas, separando el espacio de juego en elementos como el tipo de unidad, la vida de las unidades, la visibilidad del mapa mientras se preserva los elementos visuales y espaciales del juego. Además, *PySC2* contiene un conjunto de minijuegos que separan las distintas técnicas y estrategias de *StarCraft II* en entornos más simplificados y fáciles de manipular, idóneos para realizar pruebas (sección 3.4.1), ya que el aprendizaje sobre una partida completa de *StarCraft II* puede llegar a ser complicado.

[1] `http://www.cs.mun.ca/~dchurchill/StarCraftaicomp/`

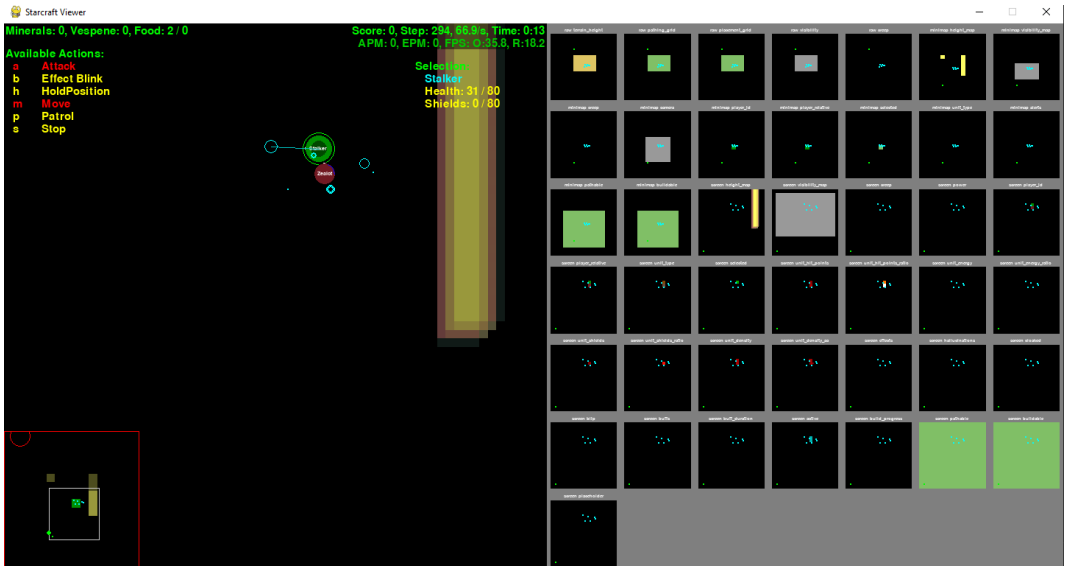


Figura 3.2: Interfaz de *PySC2*

Esta combinación de interfaz y base de datos hace que *PySC2* ofrezca un punto de referencia clave a la hora de programar no sólo algoritmos de aprendizaje por refuerzo, sino también para jugar con gran variedad de áreas del aprendizaje automático, como son la memoria y la predicción de secuencias.

A pesar de los recursos que ofrece *PySC2*, en su día la propia compañía *DeepMind* realizó pruebas con las inteligencias artificiales desarrolladas por ellos mismos, enfrentando a sus mejores bots contra otros agentes competentes en entornos controlados como son los minijuegos y consiguiendo unos resultados positivos y una gran actuación por parte sus agentes. Pero, cuando enfrentaron a su bot más complejo y eficaz contra la inteligencia artificial más sencilla que ofrece el videojuego en una partida real, con todos los recursos del juego disponibles, el resultado dejó mucho que desear [41].

Se dieron cuenta de que, incluso con un agente tan avanzado como el que tenían creado, cuando lo enfrentaban a un entorno más complejo como es una batalla estándar en *StarCraft II* los resultados eran pésimos. Tras años de trabajo sobre esta inteligencia artificial, *DeepMind* aún no han conseguido crear un bot que sea invencible. La última vez que lo mostraron consiguió ganar algunas partidas contra jugadores profesionales, pero otras las perdió, demostrando que todavía no está listo para actuar de forma perfecta ante todas las situaciones.

Intentar realizar una inteligencia artificial sobre una batalla real no sería abarcable ni por tiempo ni por el equipo del que se dispone para desarrollarla. Por esa razón, este trabajo se va a centrar en el desarrollo de pequeños jugadores automáticos haciendo uso de estos entornos controlados, sin extenderse más allá de estos límites, priorizando el correcto funcionamiento de los bots en estos minijuegos.

3.4.1. Minijuegos de PySC2

PySC2 nos ofrece varios minijuegos para poder desarrollar jugadores automáticos más sencillos y específicos, simplificando el entorno. Estos minijuegos son variados, permitiéndonos explorar distintos aspectos del entorno de *StarCraft II* y una variedad de estrategias de combate.

En el caso de este trabajo, se han aprovechado dos minijuegos aportados por *PySC2*, haciendo modificaciones en los mismos para la realización de un mayor número de pruebas. Los minijuegos que se han empleado son los siguientes:

- *Moverse a la baliza*. Cuenta con un mapa en el que se posicionan, en lugares aleatorios, un *marine* y una baliza, la cual se traslada a una posición aleatoria cada vez que el *marine* llega a ella, sumando un punto a la recompensa.
- *Crear* marines. Un mapa que contiene un centro de comandos, ocho zonas de minerales y doce trabajadores. Se deben recoger y realizar las acciones necesarias para poder generar nuevos *marines*, generando todos lo que se pueda antes de que termine el episodio. La recompensa esta compuesta del número total de *marines* creados.

Además de estos minijuegos, la comunidad de PySC2 ha generado un mayor número de minijuegos los cuales pueden ser empleados con igual facilidad que los previamente descritos. De este conjunto, el que se ha empleado es:

- *Derrotar zealots con blinks*. Compuesto de dos *stalkers* situados en un lado aleatorio del mapa y tres *zealots* en el lado opuesto, en una posición también aleatoria. El objetivo es desarrollar un agente que consiga derrotar a los tres *zealots* sin sufrir daños. Se dan cinco puntos de recompensas por cada *zealots* derrotado y se resta un punto por cada *stalker* destruido. El episodio termina cuando el tiempo (2 minutos) expira o cuando los *stalkers* han sido derrotados.

Capítulo 4

Arquitectura

Para el desarrollo de las pruebas se ha llevado a cabo la creación de un código reutilizable con el objetivo de facilitar y agilizar el avance de las pruebas, originando un programa que pueda ser usado con entornos y agentes diferentes de los empleados en este trabajo, siendo independiente de la metodología de aprendizaje automático que se emplea en cada caso.

Para este desarrollo se ha empleado el lenguaje de programación *Python*.

4.1. Módulo Environment

El desarrollo de jugadores automáticos debe producirse sobre un entorno que les otorgue lo necesario para su correcto funcionamiento. En el caso de los agentes que emplean aprendizaje por refuerzo, el entorno debe proporcionarles el estado actual del entorno y la recompensa que debe recibir cada agente según la acción que estos hayan realizado, con el fin de aportarle la capacidad de resolver problemas en un entorno específico. También es importante que exista un medio que permita al jugador mandar instrucciones al entorno. Además, estos jugadores automáticos pueden aprender en entornos distintos, cambiando cómo se manipula cada uno de ellos.

Para ello se ha generado una interfaz de entorno denominada *Environment* (figura 4.1) la cual define los métodos básicos que debe tener un entorno para poder acceder a su información y manipularle. Esta es la interfaz desde la cual deben heredar todas las clases que vayan a implementar un entorno de aprendizaje.

Los métodos básicos definidos son los siguientes, explicando la funcionalidad que debe tener cada uno de ellos:

- *reset*. Restaura el entorno cuando un episodio termina, volviendo al estado inicial.
- *prepare*. Realiza las inicializaciones y preparativos necesarios en el inicio de cada episodio.

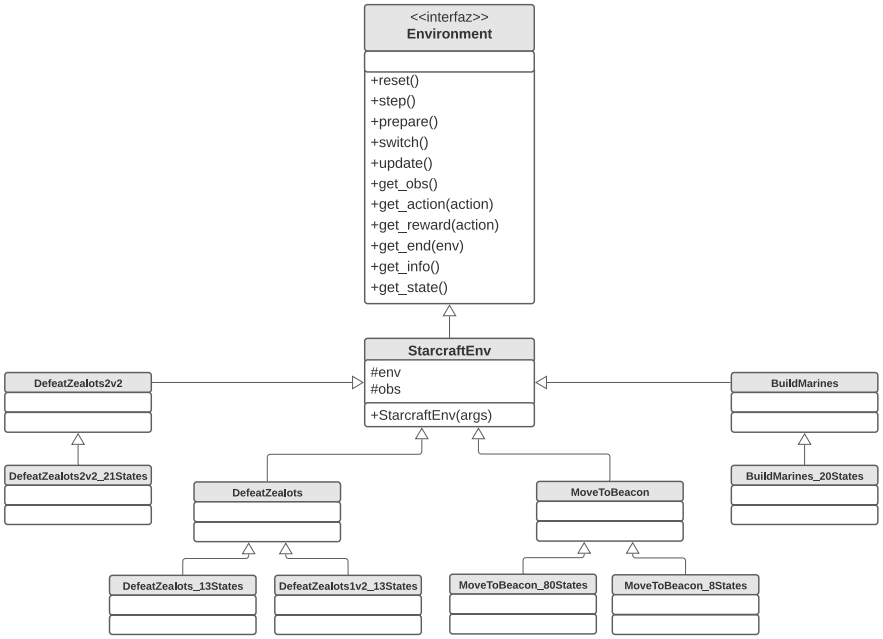


Figura 4.1: Diagrama UML de la interfaz del entorno

- *update*. Recibe como parámetro el tiempo transcurrido desde el último *frame* (*delta time*) y sirve para actualizar valores en cada *tick* del entorno.

- *step*. Permite especificar que acciones se transmitirán a la conexión con el juego.

- *get_info*. Especifica las características del *AbstractAgent* (apartado 4.2) que se vaya a utilizar, dependiendo del entorno en el que se encuentre el jugador automático.

- *get_obs*. Devolver las características del entorno en el *tick* actual, como es el tiempo del juego.

- *get_action*. Cambia la acción ofrecida en el parámetro de entrada *action* a una representación comprensible para la conexión con el juego.

- *get_state*. Devuelve el estado actual del entorno.

- *get_reward*. Accede a la recompensa obtenida en el estado actual al realizar una acción especificada en la variable de entrada *action*.

- *get_end*. Permite especificar razones por las que debe terminar un episodio antes de tiempo.

- *switch*. Debido a la posibilidad de emplear aprendizaje multiagente, cambia el foco del entorno entre los agentes, permitiendo devolver las observaciones desde el punto de vista del agente enfocado.

4.1.1. Clase StarCraftEnv

Utilizando la interfaz del entorno *Environment*, se desarrolla una clase hija denominada *StarCraftEnv* la cual envuelve las acciones básicas de los entornos que hacen uso de *PySC2* (sección 3.4). Es una clase abstracta, y proporciona la funcionalidad básica de conexión con el juego usando *PySC2* y las subclases la usarán para particularizar la conexión con los distintos minijuegos.

La inicialización del entorno se lleva a cabo en el constructor. Este entorno depende del mapa que se va a emplear, buscando en la variable de entrada *args* el nombre del mapa en cuestión. Se almacena la instancia del entorno de *StaCraft II* en una variable *env* para sus futuros usos en la clase.

De los métodos básicos de *Environment* (apartado 4.1), se implementan los siguientes métodos:

- *reset*. Llama al propio método *reset* existente en *PySC2*, el cual cumple con esta función. Al reiniciar el entorno también se actualiza la variable *obs* que contiene la observación del entorno.

- *get_obs*. Devuelve la observación del entorno, almacenada en la variable *obs*. Esta observación se recalcula en los otros métodos del entorno cuando se realiza un cambio en su estado (porque se ha restaurado el entorno o porque se ha realizado una acción)

El resto de métodos de *Environment* no se implementan debido a que su funcionamiento es específico del entorno de *StarCraft II* que se utilice.

4.1.2. Clase del entorno del minijuego

Por cada uno de los minijuegos empleados en este trabajo (capítulos 5, 6 y 7), se ha desarrollado una clase hija de *StarCraftEnv* para controlar el entorno según las características de cada minijuego. Estas clases implementan los métodos de *Environment* que no se han implementado en la clase padre *StarCraftEnv*. Esto se debe a que esos métodos dependen de las necesidades del entorno de cada minijuego, como es la conversión de las acciones.

Cada minijuego puede tener representaciones distintas de algunos métodos dependiendo del agente que lo emplee, como es el caso de *get_info* y *get_state*. Por ejemplo, si dos agentes emplean distintos algoritmos de aprendizaje, se puede querer que la representación del estado o la configuración de este algoritmo, sea distinta. Es por esto por lo que se desarrollan clases hijas a las clases de los minijuegos, que se mencionarán como clases especializadas. Los métodos de *get_info* y *get_state* solo estarán implementados por las clases especializadas.

Para que esto se vea más claro, se va a poner como ejemplo el minijuego *Moverse a la baliza* (capítulo 5). Este minijuego consiste en mover a una unidad hacia el punto marcado, por lo tanto las acciones que van a ser necesarias son el movimiento de la unidad, y estas acciones se comparten sobre todas las especializaciones en este minijuego. Por lo tanto, el método *get_action* (el cual se encarga de las acciones) estará implementado en la clase general del minijuego.

Pero el minijuego *Moverse a la baliza* es utilizado tanto por un agente que emplea Q-Learning como por un agente que usa Deep Q-Learning. Cada uno de estos agentes necesitan una información distinta del entorno dependiendo de sus algoritmos. Por ejemplo, Deep Q-Learning necesita saber los nodos y capas ocultas con las que van a contar sus redes neuronales (apartado 2.6) pero Q-Learning no necesita esta información, sino que necesita saber las dimensiones de su tabla-Q (apartado 2.4).

Del mismo modo, cada uno de estos agentes puede necesitar una representación del estado distinta para emplearlo en sus algoritmos, pudiendo necesitar uno únicamente un número, representando en la tabla-Q el estado específico, y el otro una ristra de números, representando los nodos de entrada de sus redes neuronales. Pero el resto del funcionamiento del entorno es el mismo para cada uno de los agentes, no siendo necesaria cambiar la representación de ningún otro método en las clases especializadas.

4.2. Módulo AbstractAgent

Los jugadores automáticos están controlados por agentes que deciden como deben actuar para alcanzar sus objetivos. Estos agentes trabajan sin la intervención directa de un usuario y sus comportamientos están determinados por las metas que persiguen. Para poder llevar a cabo el desarrollo de estos agentes se ha empleado la interfaz *AbstractAgent*. Esta interfaz define los métodos básicos para desarrollar los comportamientos de cada agente, enfocando su funcionamiento al aprendizaje automático (figura 4.2).

Esta interfaz cuenta con los métodos que son empleados desde el programa principal (apartado 4.3) y que son necesarios para el funcionamiento del agente en el entorno. Estos métodos y la funcionalidad que deben implementar son:

- *prepare*. Preparar las variables del agente necesarias antes del inicio de un episodio. Para llevar a cabo esta preparación, existen dos variables de entrada por si son necesarias: *env*, la clase del entorno en el que se realiza el aprendizaje; y *episode*, el número de la partida que se va a ejecutar.

- *update*. Realizar las acciones necesarias en cada *tick* del entorno, como puede ser actualizar variables, llamar a otros métodos, etc. Para estas actualizaciones se cuenta con dos variables de entrada: el tiempo transcurrido desde el último *frame* (*delta time*) y la clase del entorno (*env*).

- *train*. Llevar a cabo el algoritmo de aprendizaje del que hace uso el agente.

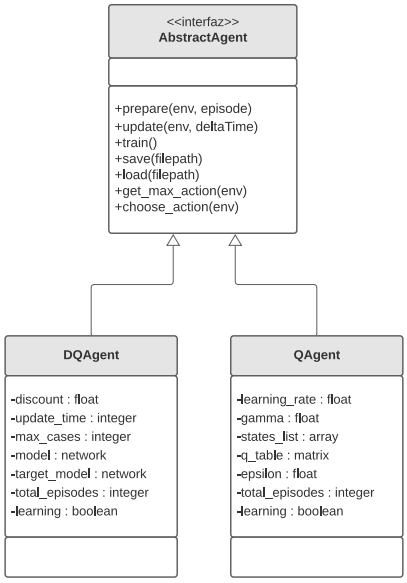


Figura 4.2: Diagrama UML de la interfaz del agente

- *save*. Guardar en un fichero el progreso del aprendizaje. Cuenta con una variable de entrada *filepath* que especifica la ruta en la que guardar el fichero.
- *load*. Cargar un aprendizaje posterior almacenado en un fichero. Tiene una variable de entrada *filepath* que especifica dónde se encuentra el fichero.
- *get_max_action*. Elegir la acción con mayor recompensa en el estado actual e informar al entorno de la acción que se ha elegido. Para ello cuenta con la clase *Environment* como valor de entrada *env*.
- *choose_action*. Elegir la acción que se va a realizar e informar al entorno de la acción que se ha elegido. Para ello cuenta con la clase *Environment* como valor de entrada *env*.

4.2.1. Clase QAgent

Para los agentes que quieran emplear el algoritmo de Q-Learning se ha desarrollado una clase llamada *QAgent*, hija de la interfaz de *AbstractAgent* (figura 4.2). Esta clase genera y rellena la *tabla-Q* que se emplea en Q-Learning y actualiza los valores de la tabla siguiendo su algoritmo (apartado 2.4.1). Esta tabla se genera al mismo tiempo que se va aprendiendo, almacenando las acciones y los estados del entorno según van apareciendo en el aprendizaje.

En el constructor de la clase se establece el valor de las variables que se van a emplear en el algoritmo. Estos valores se consiguen mediante un diccionario llamado *info* que se pasa como variable de entrada. Un ejemplo de estos valores serían el porcentaje de descuento (γ) y el índice de aprendizaje (α) que se emplea en Q-Learning e incluso la lista de acciones posibles en el entorno. Además, este diccionario contiene

otra información útil para el funcionamiento del agente, como es si se necesita realizar un aprendizaje o emplear lo ya aprendido y el número total de episodios que se van a ejecutar.

Esta clase implementa los métodos de la interfaz *AbstractAgent*, los cuales realizan las siguientes acciones siguiendo con el funcionamiento planeado en la interfaz:

- *prepare*. Actualiza el valor ϵ para que vaya convergiendo a $\epsilon = 0$ con el paso de los episodios. Este valor expresa la probabilidad de que se elija una acción aleatoria o la acción que mayor recompensa puede aportar. Esta actualización se consigue haciendo uso del número total de episodios que hay y el número del episodio por el que se va, otorgado por la variable de entrada *episode*. Este valor se va a disminuyendo hasta que el número de episodios ejecutados es mayor a un tercio de los episodios totales, permitiendo que durante un periodo de la sesión de aprendizaje refuerce las acciones ya aprendidas en cada estado.

- *update*. Actualiza cada *tick* de la sesión las variables que van a usarse en el algoritmo y necesitan ser obtenidas por el entorno: el nuevo estado y la recompensa por la acción realizada. Además, se le pide al entorno que se actualice y, si se está aprendiendo, se lleva a cabo el entrenamiento llamando al método *train*. Una vez el método *train* ha realizado su trabajo, se establece el nuevo estado como el estado actual y se elige la acción que se va a realizar. Esta acción se elige con el método *choose_action* si se está llevando a cabo el aprendizaje en esa sesión. En caso contrario, se emplea únicamente el método *get_max_action* para esa tarea.

- *train*. Lleva a cabo el algoritmo de Q-Learning. Cuando se llama a este método, la clase ya contiene la información de las variables básicas para el funcionamiento del algoritmo, pues han sido establecidas anteriormente. Estas variables básicas son: el estado actual, la acción realizada, la recompensa por ejecutar dicha acción y el nuevo estado. Hay que destacar que la tabla-Q del algoritmo se representa mediante una tabla dinámica en la que se van añadiendo estados según se visitan, por lo que el primer paso del método es comprobar si los estados están registrados o no, añadiéndolos en caso negativo. Una vez se sabe que ambos estados están en la tabla, se lleva a cabo el algoritmo explicado en el apartado 2.4.1.

- *save*. Guarda en la ruta especificada por el parámetro de entrada *filepath* el aprendizaje que se ha obtenido.

- *load*. Cargar el aprendizaje almacenado en la carpeta especificada en la variable de entrada *filepath*, inicializando la tabla-Q y la lista de estados registrados a partir de los ficheros que existen en esa carpeta.

- *get_max_action*. Comprueba cuál es la acción con mayor recompensa en el estado actual y la almacena como la acción que se va a ejecutar. Además, informa al entorno (mediante la variable de entrada *env*) que esa es la acción que se va a llevar a cabo para que la ejecute.

- *choose_action*. Verifica si el estado está registrado, añadiéndolo a la tabla-Q en caso contrario. A continuación, comprueba el porcentaje del ϵ para saber si la

acción que se va a realizar debe ser aleatoria o debe ser la que mayor recompensa tiene ese estado, empleando el método *get_max_action* en caso de que no sea aleatorio. Al igual que el método *get_max_action*, si la acción ha resultado aleatoria, se almacena esa acción como la que se va a ejecutar e informa al entorno (mediante la variable de entrada *env*) de que esa es la acción que se desea realizar.

4.2.2. Clase DQAgent

La clase *DQAgent* se desarrolla para todos aquellos agentes que quieran emplear el algoritmo de Deep Q-Learning en su aprendizaje. Esta clase hereda de la interfaz *AbstractAgent* (figura 4.2) y genera las redes neuronales que se emplean en esta técnica de aprendizaje por refuerzo, modificando los pesos de los nodos a medida que se realiza el aprendizaje hasta obtener los pesos finales. Son dos las redes: una que calcula el valor de predicción y otra calcula el valor objetivo. A la segunda red neuronal se le actualizan sus pesos cada un cierto número de aprendizajes, como se explica en en el apartado 2.6.2.

En el constructor de la clase se lleva a cabo la inicialización de sus variables haciendo uso de la variable de entrada *info*, explicada en el apartado 4.2. Los datos necesarios para la creación de las redes neuronales, como son el número de capas y de nodos ocultos, está dispuesta en la variable *info*. Con estos valores se generan las dos redes neuronales mencionadas anteriormente. Del mismo modo, los valores que se emplean en el algoritmo vienen dadas en esa misma variable de entrada, aportando datos como el porcentaje de descuento (γ), el número máximo de casos almacenados para empezar a predecir en la red neuronal y el tiempo que debe pasar entre aprendizajes para actualizar los pesos de la segunda red, entre otros valores.

Al igual que sucedía en la clase *QAgent* (apartado 4.2.1), la variable de entrada *info* del constructor también aporta otros datos útiles para el funcionamiento del agente. Uno de los valores expresa si se debe realizar el aprendizaje en la sesión actual o no, restringiendo el uso de métodos dependiendo de este valor. El segundo valor útil es el número de episodios totales que se van a llevar a cabo en la sesión y se emplea para actualizar el valor ϵ al principio de cada episodio.

DQAgent implementa todos los métodos de su interfaz. Algunos métodos tienen el mismo funcionamiento que las implementaciones de la clase *QAgent* (apartado 4.2.1). Estos métodos son: *prepare*, *update*, *get_max_action* y *choose_action*. Es por esta razón por la que no se van a explicar de nuevo, aunque sí estén implementados.

El funcionamiento de los métodos que difieren con *QAgent* son los siguientes:

- *train*. Realiza el algoritmo de Deep Q-Learning. Para ello, cuenta con las variables básicas del algoritmo gracias a que han sido establecidas en métodos anteriores. Estas variables son: el estado actual, la acción realizada, la recompensa por ejecutar dicha acción y el nuevo estado. Si ha pasado el tiempo especificado anteriormente en el diccionario del constructor, se copian los pesos adquiridos

por la red principal en la red secundaria. El procedimiento de este método empieza comprobando si tenemos un número mínimo de experiencias almacenadas. Si se cumple, se recoge un *batch* con una porción aleatoria de las experiencias que hay almacenadas y se realiza el algoritmo de Deep Q-Learning. Esto se hace así para evitar el posible sobreajuste y divergencia de modelo, como se explica en la sección 2.6.2.

- *save*. Guardar en la carpeta especificada en la variable de entrada *filepath* el aprendizaje llevado a cabo. Este proceso se basa en los pesos que han obtenido las redes neuronales.

- *load*. Cargar el aprendizaje almacenado en la carpeta especificada en la variable de entrada *filepath*, inicializando las redes neuronales dependiendo de los pesos especificados en los archivos de la carpeta.

4.3. Programa principal

Se ha desarrollado un programa el cual ofrece dos opciones para ejecutar un agente: una para llevar a cabo el aprendizaje de los agentes y la otra para observar cómo han aprendido estos agentes. La primera opción va a ser mencionada en este documento como *learn* y la segunda opción como *smart*.

Este programa permite al usuario especificar qué opción de las mencionadas se va a emplear en la sesión, además de otras configuraciones útiles con el uso de *flags*. Algunas de ellas son: el número de episodios que se van a realizar, el número de *steps* que va a tener cada episodio, el nombre y la ruta del fichero de guardado o de carga para el agente y similares. Además, de cara a *learn*, también se ofrece la posibilidad de elegir si se va a cargar información de aprendizaje al modelo, permitiendo que parta de una sesión de un aprendizaje anterior.

Una de las características del programa que puede especificar el usuario es el número de agentes que estarán actuando en la sesión. El programa principal soporta el control de varios jugadores automáticos, generando una lista de agentes la cual recorre en cada momento clave del bucle. Estos momentos claves son aquellos en los que el agente interactúa con el entorno, anunciando al entorno qué agente va a actuar haciendo uso del método *switch* mencionado en el apartado 4.1.

El programa inicia generando el entorno decidido por el usuario. Acto seguido se crean todos los agentes que se desean utilizar en esa sesión, otorgándoles la información obtenida del entorno a cada uno. Con el entorno y los agentes ya inicializados, se comprueba si se quiere cargar de fichero el aprendizaje, ya sea para partir de una sesión anterior o para observar lo que se ha aprendido.

Como se ha mencionado anteriormente, el programa cuenta con dos opciones: *learn* y *smart*. La diferencia entre el uso de una opción u otra reside principalmente dentro de los agentes, pues su única diferencia en el programa es si se carga o no de fichero, cargando en el caso de que se elija la segunda opción o si se especifica que se

cargue cuando se elige la opción *learn*. Esto otorga libertad a cada agente de gestionar su funcionamiento en cada opción de la forma que mejor les convenga.

En el caso de los agentes desarrollados en este trabajo, *QAgent* y *DQAgent* (apartados 4.2.1 y 4.2.2), la desigualdad se produce en el método *update*. En este método, como se ha explicado en los apartados señalados, si se ha elegido la opción *learn* el agente tendrá que realizar su algoritmo del método *train* y decidir entre una acción aleatoria o la que maximice la recompensa. En el caso de elegir la opción *smart*, el algoritmo no se lleva a cabo y siempre se elige la acción con mayor recompensa.

Hay que destacar que el bucle del programa cuenta con ciertas singularidades. La primera es que se realiza es que se realiza un guardado automático de lo aprendido cada número de episodios que especifique el usuario. Esta decisión se ha llevado a cabo para que, en caso de que el programa se interrumpa por un error no controlado en un momento muy avanzado del aprendizaje, el desarrollador no pierda completamente el progreso. Además, esto es igualmente útil si, por alguna razón, el agente realiza su tarea correctamente pero en cierto episodio empieza a desaprender, ya sea por algún tipo de sobreajuste o por algún caso de divergencia, dando la oportunidad de poder usar los datos de lo aprendido hasta el momento correcto sin necesidad de tener que volver a ponerlo a entrenar con la esperanza de conseguir los mismos resultados que anteriormente.

La segunda peculiaridad a destacar es que, en algunos entornos de entrenamiento, se puede desear parar la ejecución de un episodio por razones distintas a que los *steps* del episodio han llegado a su fin. Además, si se sale del episodio de forma abrupta, lo más lógico es que se quiera guardar el efecto de la última acción en el aprendizaje del agente. Por esta razón, se ha hecho posible que el bucle de cada episodio pueda interrumpirse si el agente lo decide, señalando que el episodio debe terminarse debido a lo que el programador especifique. Si esto sucede, se realiza un último entrenamiento del agente, antes de inicializar un nuevo episodio, con esta última acción y la recompensa que obtiene con dicha acción.

Se ha optado por utilizar un valor llamado tiempo por acción. Este valor tiene predeterminado medio segundo y marca cada cuánto tiempo se le va a solicitar al agente una nueva acción a ejecutar. Mientras este tiempo no haya pasado, al entorno se le ordena realizar la misma acción que se le solicitó al agente la última vez que se le preguntó. Esto se implementa por si se quiere evitar la dificultad que aportan las acciones durativas (apartado 3.3) o por otra razón.

Respecto al funcionamiento del bucle general, antes de cada episodio se reinicia el entorno para situarlo en su estado inicial y se prepara tanto al agente como al entorno para el episodio que está por llegar. Por otro lado, si estamos en la opción de aprendizaje, se comprueba si se ha solicitado el fin del episodio, realizando un entrenamiento de los agentes en caso de que así sea. También se comprueba si es el momento de hacer el guardado de seguridad mencionado anteriormente. Una vez se han inicializado todo lo necesario, se pasa al desarrollo de un episodio.

El desarrollo de un episodio sucede de la siguiente manera (figura 4.3). Primero se pregunta si el agente desea terminar el episodio. Si es así, se realiza lo mencionado

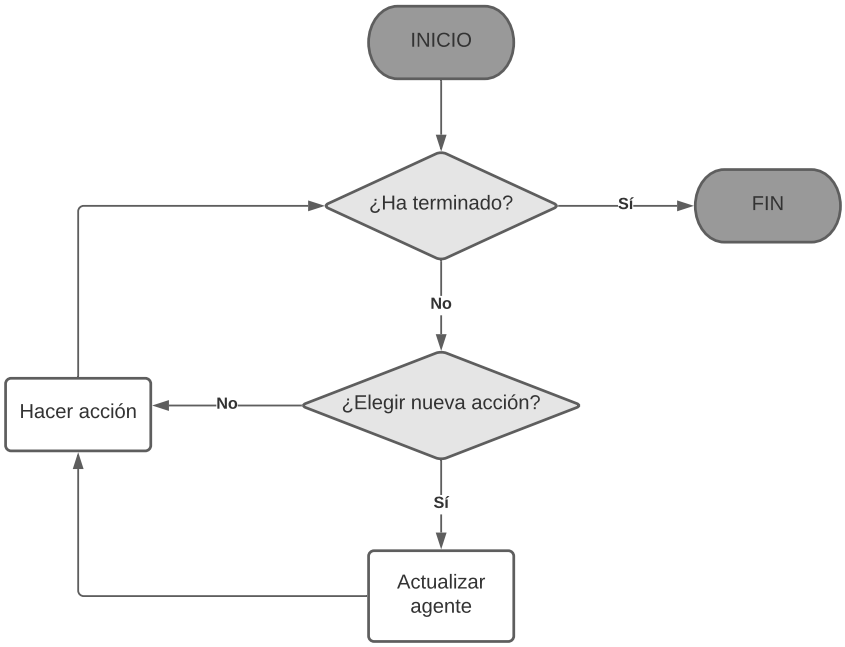


Figura 4.3: Diagrama de flujo del bucle de un episodio

con anterioridad, saliendo del bucle en ese mismo instante. Una vez se confirma que continuamos con el episodio, comprobamos si es el momento de realizar una nueva acción y, en caso afirmativo, se le pide al agente que se actualice, encargándose él internamente de todos los pasos que necesite para poner en orden sus variables y obtener la acción. En esta actualización se le ofrece al agente el entorno, como se menciona en el apartado 4.2, para que pueda especificarle la acción que se va a realizar y pueda adquirir los datos que necesite del entorno para su funcionamiento. En el caso de que no sea el momento para elegir una nueva acción, se realiza el *step* del entorno, sin interferencia del agente más allá de la pregunta inicial de si se ha terminado el episodio.

Capítulo 5

Moverse a la baliza

5.1. Descripción del minijuego

El minijuego que se elige para realizar este experimento es *Moverse a la baliza*, que consiste en hacer que el agente aprenda a dirigirse a un punto específico, denominado como baliza, que se posiciona en un punto aleatorio. La elección de este minijuego por encima de otros se debe a que es el minijuego recomendado para comenzar a usar *PySC2* y se considera uno de los minijuegos fundamentales (figura 5.1).

Figura 5.1: Entorno del minijuego *Moverse a la baliza*

El objetivo del minijuego es mover la unidad al punto marcado en el mapa. Cada vez que se llega a la baliza se recibe un punto. Además, una vez se llega a la baliza, esta reaparece en una posición aleatoria en cualquier punto del mapa.

El minijuego tiene una duración de 2 minutos de tiempo real y una persona debería poder alcanzar de 16 a 22 balizas en total. Este número varía dependiendo de la suerte que se tenga, es decir, si al coger una baliza esta reaparece en una posición muy lejana se tiene que recorrer una distancia mucho mayor que si reaparece en un punto más cercano a la posición de la unidad.

Para una persona este minijuego no debería de suponer un reto muy elevado. Primero, se tiene que seleccionar la unidad aliada y una vez seleccionada hay que moverla hasta ese punto. Una vez se alcanza la baliza se tiene que repetir el proceso, pero esta vez en una nueva posición.

5.1.1. Acciones

El agente desarrollado cuenta únicamente con 8 acciones, siendo éstas moverse en las cuatro direcciones básicas y las diagonales.

Inicialmente, en cada paso *step* del juego el agente elegía una de las posibles acciones dependiendo de la tabla-Q y, en el siguiente *step*, elegía otra de las posibles acciones de la misma forma que antes. Esto resulta en que el agente prácticamente no tiene tiempo para desarrollar la acción seleccionada. Como se ha explicado anteriormente, en *StarCraft II* las acciones son durativas y para evitar la complejidad de este factor en el primer minijuego, se ha optado por no elegir una nueva acción por cada nuevo *step*. Una vez se selecciona una acción, se espera medio segundo para elegir otra acción distinta. El agente repite la misma acción durante el tiempo en el que la acción no varía.

5.1.2. Estados

La primera impresión de cómo se deben plantear los estados en este minijuego son todas las posibles combinaciones de posiciones del jugador y de la baliza. Es decir, la representación de un estado serían cuatro números, la posición en la x e y de la unidad y la posición en la x e y de la baliza. Esto es difícil y costoso de representar en una tabla, además de que no se puede generalizar lo aprendido en un punto del mapa a cualquier otro. Por si esto no fuese suficiente, como ya se ha explicado en el apartado 2.4, Q-Learning asocia los valores esperados de una recompensa a pares estado y acción y por lo tanto no va a poder aproximar un nuevo estado con lo aprendido de los anteriores.

Si se plantearan así los estados, parece altamente improbable que se vayan a poder explorar todos y cada uno de estos posibles estados. Esto significa que si una vez terminado el aprendizaje se llega a un estado no explorado en el aprendizaje, la tabla Q no tendría un valor asociado correcto a ese par estado y acción, lo que significa que el agente no sabría que hacer en esa situación.

Por todo esto, para plantear los estados se decide observar el mapa como un entorno dividido en distintos rangos de ángulos. De esta forma, el estado en el que se

Figura 5.2: Visión de los estados del juego en el minijuego *Moverse a la baliza*

encuentra el agente depende del rango en el que se encuentra el ángulo que forma nuestro agente con la baliza. Este ángulo se mueve en el rango de 0º a 360º, por lo que, para reducirlo a sólo 8 estados, decidimos dividir cada estado en un rango de 45º. Al ser una representación tan simple se pueden encontrar problemas como el que se explica a continuación.

Si se supone que el agente se encuentra en el estado 8, es decir, con la baliza arriba a la derecha de éste (figura 5.2), y está muy cerca de la baliza, esto significa que, tanto al realizar la acción de que el agente se mueva hacia arriba, arriba a la derecha y a la derecha, se llega hasta la baliza. En el caso de que la acción realizada haya sido moverse hacia la derecha, el agente recibe una recompensa por esta. Por lo tanto, si nos encontramos en el estado 8, el agente ha aprendido que es una buena decisión moverse hacia la derecha.

Ahora se va a suponer que la baliza se ha movido a una nueva posición y que el agente se encuentra en el estado 8. Debido a la recompensa anteriormente recibida, el agente decide moverse hacia la derecha, lo que le lleva al estado 1. En este momento, si el agente decide realizar la acción de moverse hacia la izquierda le devolverá al estado 8 y, por como funciona Q-Learning, se propagará la recompensa. Esto significa que el agente en este momento, cuando se encuentra en el estado 8, se moverá a la derecha y, en el 1, se moverá de nuevo hacia la izquierda, quedándose en un bucle sin avanzar hacia la baliza en ningún momento.

Una vez se entiende el problema que puede surgir con esta representación, hay que entender cuál es la razón, ya que en un principio parecía que se podía asociar perfectamente un estado con una acción. El problema es que no es lo mismo realizar una acción cuando se está en un estado y cerca de la baliza que realizar esa misma acción cuando el agente se encuentra en ese mismo estado y muy alejado de la baliza. Para resolver este problema se ha optado por, además de esta representación de los

ángulos, representar a qué distancia se encuentra el agente de la baliza.

5.1.3. Recompensas

Para la recompensa del agente, se ha optado por probar 2 planteamientos distintos.

La primera recompensa viene dada por la diferencia de la distancia entre el agente y la baliza en el estado *S* y la distancia entre el agente y la baliza en el siguiente estado *S'*. Esto significa que si el agente se acerca a la baliza este recibe una recompensa positiva y si se aleja recibe una recompensa negativa. Como es de esperar, al recompensar cada acción el agente aprende muy rápidamente a acercarse a la baliza y por lo tanto consigue llegar a su posición en relativamente poco tiempo.

La segunda recompensa es la que ofrece el propio entorno de PySC2, la cual es 1 si se ha llegado a la baliza y 0 si no se ha conseguido llegar. Con esta recompensa si se puede comprobar si el agente es capaz de propagar la recompensa. Cuando llegue por primera vez a la baliza, de manera aleatoria, esa será la única acción que tendrá una recompensa positiva. Debido al funcionamiento del algoritmo, la recompensa se propagará poco a poco a las acciones que llevan de nuevo al estado recompensado. Si por ejemplo, se llegara a la baliza en el estado 1, significa que la baliza se encontraba encima del agente y que estaba muy cerca. Si en el futuro el agente se encuentra en el estado 2 y de este estado pasara al 1, la recompensa se habrá propagado debido a que esa acción le ha llevado a un estado recompensado.

Como es de esperar, con esta segunda recompensa el agente tarda más en aprender a acercarse a la baliza. Porque, como ya hemos explicado, la mayoría de las acciones no se ven recompensadas y la recompensa tarda tiempo en propagarse.

5.2. Q-Learning

Para aplicar la representación del estado a Q-Learning (sección 2.4.1) se realiza de la siguiente manera. La distancia no puede ser un número arbitrario ya que los estados deben de poder representarse en una tabla. Por lo tanto, se ha decido que los estados del 0 al 9 representen que la baliza se encuentra encima del agente, siendo el estado 0 muy cercano a la baliza y el 9 muy lejano. Luego, los estados del 10 al 19 representan que la baliza se encuentra arriba a la izquierda y las distancias se representan de la misma forma (figura 5.3). El resto de ángulos siguen este mismo patrón hasta llegar al estado 79.

Se ha aplicado el algoritmo de Q-Learning con un índice de aprendizaje α igual a 0.8 y un factor de descuento γ igual a 0.95. Para crear la tabla del algoritmo se tiene que tener en cuenta el número de estados. Así pues, se decide representar con 80 estados: 8 por cada ángulo y 10 indicando las distancias. Por lo tanto, la tabla tiene 80 filas para los estados y 8 columnas indicando las acciones.

Figura 5.3: Representación del estado aplicada a Q-Learning

5.3. Deep Q-Learning

Se realizan varios experimentos con las herramientas que nos ofrece PySC2 y usando el algoritmo de Deep Q-Learning.

La mayor diferencia respecto a Q-Learning se ve reflejada en los estados, ya que ahora no se cuenta con una tabla para representar los pares estado y acción si no con una red neuronal, lo cual implica ciertas diferencias explicadas a continuación.

Se tienen 8 estados que representan cada uno de los ángulos que forman el agente con la baliza. Por otro lado, hay que representar la distancia que gracias a la red neuronal no tenemos que especificar de antemano, ya que el valor de entrada de cada neurona puede representar la distancia. Una opción sería crear la red neuronal con 80 neuronas de entrada, de esta manera, se asimilaría mucho más a lo que se ha hecho anteriormente con Q-Learning. Sin embargo, esto sería desaprovechar las ventajas que nos ofrece una red neuronal al no tener que ser la entrada de cada neurona un 0 o un 1.

Por todo esto, se ha decidido que haya sólo 8 neuronas de entrada, cada una representando uno de los ángulos que forma el agente con la baliza. Cuando el agente se encuentre en uno de esos ángulos, la neurona asociada se activa con un número entre el 0 y el 1. Si está muy cerca de la baliza, será un número muy cercano a 1 y si se encuentra muy alejado será un número muy cercano al 0.

El modelo en cuestión está formado por una única capa oculta, la cual contiene 50 nodos. Tanto la capa de entrada como la oculta tendrán activación *ReLU*. Se ha probado también con 2 capas ocultas, pero la única diferencia es que tarda más tiempo en llegar al resultado deseado.

En cuanto al resto de características, se ha optado por guardar un total de 512

experiencias, y una vez se llega a este número descartar las 64 más antiguas. Por otra parte, a la hora de entrenar a la red neuronal se recogen 64 experiencias aleatorias del resto que se tienen almacenadas. Por último, cada 5 entrenamientos se le copian los pesos de la red neuronal que hace la predicción a la red neuronal que calcula los valores objetivos.

5.4. Comparativa

Una vez se han realizado ambos entrenamientos se obtienen distintos resultados. Como se puede ver en la figura 5.4, el agente de Q-Learning necesita un amplio número de episodios para llegar a recoger un número elevado de balizas por episodio. Esta y el resto de gráficas enfocadas a la evolución del aprendizaje muestran la media de recompensa obtenida cada 50 episodios. Con cada uno de los agentes con distinto tiempo de aprendizaje, se juegan 10 episodios para sacar la media de la recompensa obtenida.

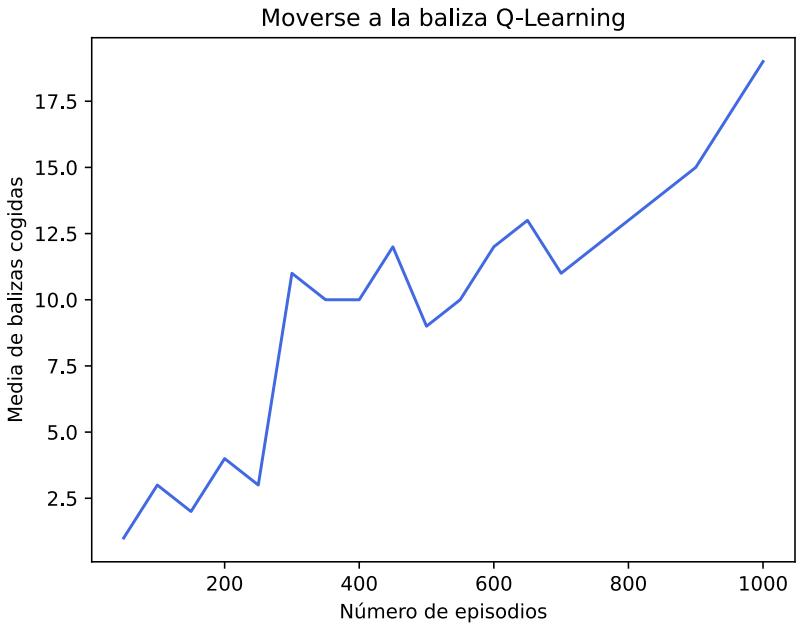


Figura 5.4: Evolución de la media de balizas alcanzadas con la segunda recompensa Q-Learning

Como ya se ha estudiado, Q-Learning necesita explorar todos los pares de estado y acción. Esto significa la imposibilidad de aproximar nuevos estados, así que para obtener unos buenos resultados hacen falta alrededor de 1000 capítulos. Cada capítulo dura exactamente 2 minutos, lo que significa que el agente de Q-Learning necesita unos 2000 minutos para recoger 19 balizas por episodio. Como se explicó en la sección 5.1, un humano es capaz de recoger de unas 16 a 22 balizas, por lo que se puede considerar que el agente está realizando un gran trabajo. Esto significa que para casos reducidos como el que se ha experimentado, Q-Learning ofrece unos resultados relativamente buenos.

Por otro lado tenemos los resultados en de este mismo experimento en Deep Q-Learning, que se pueden ver en la figura 5.5. Lo primero a tener en cuenta es que hace falta un número mucho menor de capítulos para llegar a resultados similares. A partir del epsiodio 250 ya supera el umbral de lo que podría hacer un humano, sin embargo con Q-Learning necesita al menos 900 episodios.

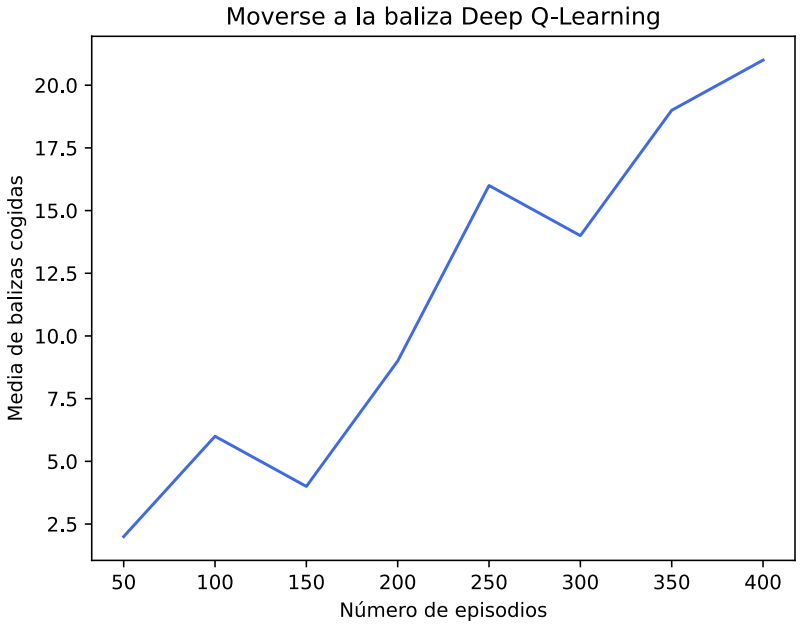


Figura 5.5: Evolución de la media de balizas alcanzadas con la segunda recompensa Deep Q-Learning

Esto se debe a que Deep Q-Learning es capaz de aproximar nuevos estados con los aprendidos anteriormente, esto resulta en un aprendizaje muchos más rápido al usar redes neuronales. En otras palabras, no hace falta explorar todos los estados. Si por ejemplo la red neuronal aprende que cuando la segunda neurona se active con un valor de 1 debe moverse hacia la izquierda, al activarse esta misma neurona con un valor de 0.9 es muy probable que sea capaz de generalizar el caso y la acción que prediga sea moverse hacia la izquierda también.

Por último, se cuenta con dos métricas más para estudiar como ha ido evolucionando el modelo a lo largo del aprendizaje. Estas sólo se muestran para el algoritmo de Deep Q-Learning, ya que evalúa dos aspectos de la red neuronal entrenada y las ofrece *Keras*. Estas métricas se calculan cada *step* del aprendizaje a diferencia de las gráficas que muestran la evolución del aprendizaje.

Primero se tiene una gráfica que muestra cómo van variando los resultados de la función de pérdida en la figura 5.6, que evalúa la desviación entre las predicciones realizas por la red neuronal y los valores reales de las observaciones utilizadas durante el aprendizaje. Como se puede apreciar, a lo largo del aprendizaje la pérdida va disminuyendo poco a poco hasta estabilizarse.

La segunda gráfica muestra la tasa de aciertos del modelo (figura 5.7), que indica si es capaz de predecir la acción correcta. En este caso la gráfica es de carácter ascendente,

llegando a un 95 % de precisión, lo cuál significa que el modelo rinde a un gran nivel.

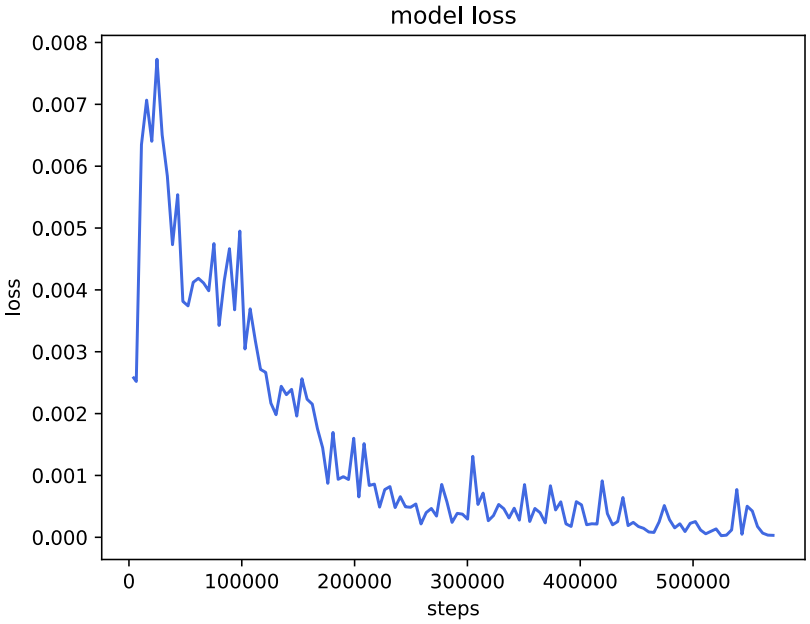


Figura 5.6: Pérdida del modelo *Moverse a la baliza*

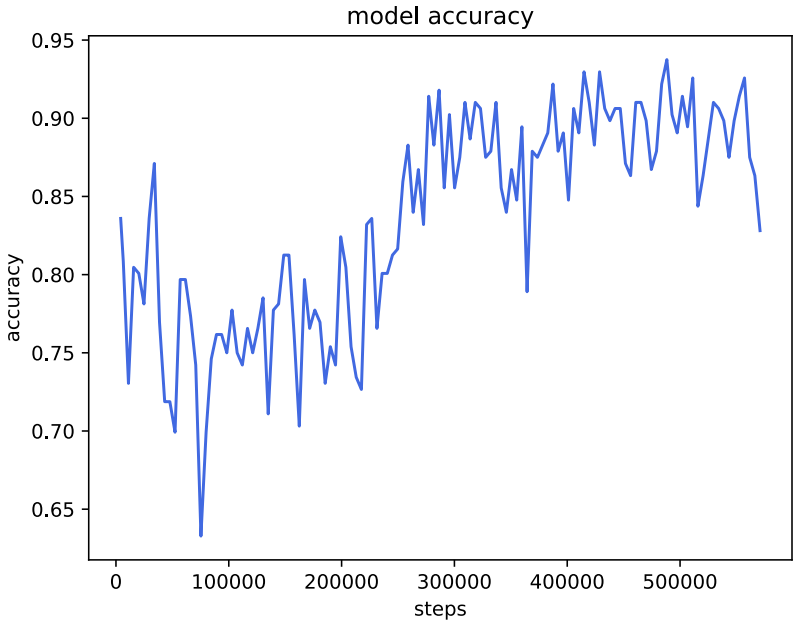


Figura 5.7: Tasa de aciertos del modelo *Moverse a la baliza*

Capítulo 6

Derrotar zealots con blinks

En lo que respecta al segundo minijuego *Derrotar* zealots *con* blinks, a medida que se van realizando más pruebas, se van efectuando un mayor número de modificaciones sobre el minijuego original, surgiendo tres entornos con dificultad progresiva. Todos ellos están enfocados a la toma decisiones a nivel reactivo o el microjuego, como se explica en la sección 3.1.3

- *Derrotar* zealots *con* blinks *uno contra uno*. Se simplifica el minijuego original de tal manera que sea un combate entre dos unidades: un stalker contra un *Zealot*.
- *Derrotar* zealots *con* blinks *uno conta dos*. Utilizando el mismo entorno que el minijuego origen, presenta un combate de uno contra dos: un stalker contra dos *zealots*.
- *Derrotar* zealots *con* blinks *dos contra dos*, cuyo tipo de enfrentamiento ya es mencionado en el propio nombre, un dos contra dos, enfrentando, por razones de funcionamiento, un stalker y un *roach* contra dos *zealots*. Siendo su antecesor *Derrotar* zealots *con* blinks *uno conta dos*, ambas unidades deben tomar la decisión de a qué enemigo priorizar, coordinándose en este caso entre ellas y atacando y moviéndose al mismo tiempo, teniendo en cuenta las posiciones tanto de los enemigos como de su aliado.

6.1. Uno contra uno

En este minijuego se controla a una unidad de la facción *Protoss*, llamada *stalker*, que puede disparar a distancia, como se menciona en la sección 3.2.2. Como enemigo nos encontramos con un *zealot*, también de la raza Protoss, el cual ataca cuerpo a cuerpo.

El objetivo principal de este minijuego es disparar y mantener las distancias con el enemigo hasta derrotarlo todas las veces posibles en dos minutos, sin morir ninguna vez. Por lo tanto, este minijuego plantea varios retos. Por ejemplo, no se puede huir siempre en la misma dirección ya que el escenario es pequeño y llega un momento en el

Figura 6.1: Entorno del minijuego *Derrotar zealots con blinks* en uno contra uno

que el agente se encuentra contra paredes, por lo que se tiene que cambiar de dirección. Luego, el agente se encuentra en desventaja, ya que, aunque el enemigo es cuerpo a cuerpo, si sólo se realiza la acción de atacar, el enemigo acaba derrotando al *stalker* antes que él al *zealot*. Por útimo, si el enemigo muere este reaparece con toda la vida y nuestra unidad continúa con la misma vida, en cambio si muere la unidad aliada se termina la partida.

El *kiting* es una de las técnicas más usadas en combates por jugadores profesionales en *StarCraft II*, como se describió en el capítulo 3.1.2. Desde el punto de vista humano no es nada fácil de realizar debido a que requiere experiencia. Para un jugador principiante, la estrategia sería seleccionar a la unidad y ordenar que ataque a la unidad enemiga de inmediato. Esto no serviría de nada pues, aunque el *stalker* tiene más vida que la unidad enemiga, el *zealot* tiene una velocidad de ataque mucho mayor a la del anterior, es decir, ataca un número mayor de veces por segundo. Mientras el *zealot* no tiene que esperar ni un 1 segundo para volver a atacar, la unidad aliada debe esperar 1.5 segundos para poder realizar su ataque de nuevo. Todo se traduce en que si únicamente se ordena atacar a la unidad enemiga, el combate duraría poco y no saldría victorioso.

Por otro lado, está el comportamiento de la unidad enemiga, el cual es relativamente simple. El *zealot* permanece quieto hasta que alguien se acerque a su posición y, una vez se entre en un radio alrededor suyo, este persigue indefinidamente a la unidad que ha entrado a su perímetro para intentar eliminarla.

La clave de este enfrentamiento es el rango de ataque de ambas unidades. La única ventaja de la que dispone el *stalker* sobre el *zealot* es que el primero puede atacar desde una distancia mucho mayor, lo que implica la posibilidad de realizar la técnica de *kiting*.

Un jugador profesional podría resolver el minijuego realizando esta técnica, que consiste en atacar a la unidad enemiga, esperar a que la animación de ataque termine, alejarse del enemigo todo lo posible hasta tener el ataque disponible de nuevo y repetir hasta eliminar a la unidad. Esta técnica puede resultar complicada ya que si se clica antes de tiempo para alejarse de la unidad enemiga se puede cancelar el ataque, por lo

tanto, hay que tener experiencia sobre la animación de ataque del *stalker*. Además, en cuanto se tiene el ataque disponible debe de usarse de nuevo lo antes posible, ya que hay que eliminar el mayor número de *zealots* en el tiempo establecido.

Por último, eliminar un *zealot* a una persona debería de llevarle entre 25 y 30 segundos. El minijuego dura dos minutos, por lo que un buen jugador debería de poder eliminar de tres a cuatro zealots. El reto del minijuego varía dependiendo de la experiencia del jugador en este tipo de situaciones, si no ha empleado la técnica de *kiting* en este u otro videojuego le puede resultar extremadamente difícil llegar a eliminar a un solo *zealot*, sin embargo para un jugador con mayor experiencia sería un reto de una dificultad normal.

En consecuencia de ello, el agente debe aprender a realizar un *kiting* perfecto, para lo cual necesita ciertos estados y acciones que se ven a continuación.

6.1.1. Acciones

Las acciones que se han determinado en este entorno son moverse y atacar. El movimiento cuenta con las 8 direcciones elegidas anteriormente para *Moverse a la baliza* (capítulo 5): moverse en las cuatro direcciones básicas y moverse en cualquiera de las cuatro diagonales.

En lo que respecta al ataque, debido a la existencia de un único enemigo en el minijuego, el agente tiene una única acción: atacar al enemigo. Esto hace que el número de acciones posibles para el agente sean 9.

Al igual que sucedía con *Moverse a la baliza*, las acciones de atacar son durativas, ya que el disparo tarda cierto tiempo en llegar hasta la unidad enemiga y, por lo tanto, ese disparo no se ve reflejado hasta que pasan varios *steps* de juego. De nuevo se evitará la complejidad que suponen las acciones durativas realizado lo explicado en la sección 5.1.1.

6.1.2. Estados

Si se intenta pensar en la información que necesita un humano para poder realizar la técnica de *kiting*, lo primero que haría falta sería saber la posición del enemigo respecto a la de la unidad aliada.

Se usa la idea del ángulo que forma el aliado con el enemigo para poder indicarle al agente en que dirección se encuentra dicho enemigo (figura 5.2). Por lo tanto, se dispondrá de 8 números los cuales tomarán un valor continuo del 0 al 1. Esto sirve para indicar la distancia a la que se encuentra el enemigo, siendo 0 si éste está muy lejos y 1 si está muy cerca. De esta manera se conseguirá que el agente entienda que no debe estar cerca del enemigo en ningún caso, ya que siempre que reciba daño estará muy cerca del *zealot*.

Para una persona puede resultar obvio que no es bueno acercarse a las paredes ya que probablemente nunca vaya a dejarse acorralar en uno de los límites del mapa. Siempre que el jugador empiece a acercase a una de las cuatro paredes este empezará a cambiar su dirección. En el caso de un jugador automático, si el enemigo estuviera a la izquierda de la unidad aliada, esta se movería hacia la derecha hasta encontrarse contra una pared, pero igualmente seguiría moviéndose hacia la derecha y no entendería por qué de repente se acerca a la unidad enemiga, lo que resultaría en un aprendizaje fallido. Por lo tanto, el agente debe de hacer algo similar a lo que haría un humano y no dejarse acorralar.

Se ha optado por incluir 4 números que van a tomar valores discretos de 0 o 1. Cada uno de ellos indica si se está pegado a una pared o no, ya que solo se tienen limitaciones por arriba, por la izquierda, por abajo y por la derecha. De esta manera, el agente poco a poco aprenderá que si selecciona la acción de moverse a la derecha con el valor que representa la pared de la derecha activo, esto le llevará a un estado en el que el enemigo está mucho más cerca. En cambio, si explora las acciones de moverse hacia arriba o hacia abajo sí conseguirá llegar a un estado en el que al menos la distancia con el enemigo se mantenga.

Por último, el *stalker* no puede disparar de forma continua. Una vez realiza un ataque, tiene que pasar cierto tiempo hasta que este pueda volver a disparar. La intención es que aprenda a alejarse de el *zealot* durante este tiempo en el que no puede atacar. Para ello, se usa un último valor discreto de 0 o 1 indicando si el agente puede disparar o si todavía tiene que esperar para que el ataque esté disponible de nuevo. Como se ha explicado anteriormente, es de vital importancia que cuando el ataque esté disponible el agente realice esta acción, ya que es un minijuego a contrarreloj. Si no se hubiera alejado lo suficiente, el agente deberá evaluar los beneficios de recibir un golpe y atacar al enemigo respecto a limitarse a huir y atacar en el futuro.

6.1.3. Recompensas

A la hora de especificar las recompensas se han probado dos opciones distintas. La primera de ellas se basaba en unas recompensas más especificas, guiando al agente sobre aquello que se quería que realizase, y la segunda, la definitiva, centrada en recompensas más generales, poniendo en práctica el funcionamiento del algoritmo de Deep Q-Learning (sección 2.6.1).

En un principio se emplearon recompensas específicas para cada una de las situaciones posibles por las que podía pasar el agente. Por ejemplo, se premiaba al agente si decidía acercarse estando demasiado lejos del enemigo y si contaba con la posibilidad de atacar, y se le castigaba en caso de que en esas circunstancias decidiese alejarse. Además, en las situaciones en las que se encontraba en rango de ataque se empleaban unas recompensas similares, castigándolo si decidía acercarse y premiándolo si decidía atacar. Como añadido, también se recibían recompensas dependiendo del daño que se le realizaba al enemigo y del daño que recibía el *stalker*, castigando si la diferencia de daño ejecutado entre ambas unidades era mayor para el *zealot* y premiando si la diferencia favorecía al *stalker*.

Aun así, estas recompensas eran guiadas y se llegaba a unos resultados positivos en poco tiempo pero sin aprovechar las ventajas del algoritmo.

Las recompensas que con las que se analizan los resultados son castigar al agente si ha sufrido daños entre una acción y otra y premiarle si se ha realizado daño al enemigo. Con estas dos únicas recompensas, el agente puede aprender a realizar la técnica de *kiting* sin ningún tipo de problemas. No se le recompensa ni por alejarse del enemigo ni por evitar paredes ni tampoco por acercarse al enemigo cuando está demasiado lejos como para poder atacar.

6.1.4. Configuración del algoritmo

En este experimento y los siguientes solo se usa Deep Q-Learning. Lo primero que se explica es el modelo que ha utilizado el agente para poder llegar a estos resultados.

Este modelo cuenta con dos capas de neuronas ocultas, ya que el problema que se quiere solucionar en este caso es mucho más complejo, la primera tiene 100 nodos y la segunda tiene 125 nodos. Tanto la capa de entrada como ambas capas ocultas están activadas con la función *ReLU*. Sin embargo, la capa de salida no tiene ningún tipo de activación, es decir, la salida de la red neuronal son valores-Q sin ningún tipo de modificación.

Por otro lado, las especificaciones en el algoritmo de Deep Q-Learning han sido las siguientes. Cada 10 entrenamientos a la red neuronal que calcula los valores objetivos, se le copian los pesos de la red neuronal que hace la predicción. Se almacenan como máximo 1024 experiencias y se eliminan un total de 128 experiencias cuando se llega a este límite. De estas experiencias, se eligen 128 experiencias aleatorias con las que se entrena la red neuronal cada 80 *steps*.

6.1.5. Análisis

Gracias a la propagación de la recompensa que realiza el algoritmo de Deep Q-Learning, estas recompensas permiten que el agente aprenda a realizar la mecánica de *kiting*, golpeando al enemigo cuando se encuentra a una distancia segura y con posibilidad de atacar y huyendo cuando la situación no es la ideal. Además, aprende que acercarse demasiado al enemigo es un problema y que acercarse a las paredes puede llevarlo a una situación crítica.

Se puede comprobar la eficacia del agente en la gráfica 6.2. Como se puede comprobar, en las primeras fases del entrenamiento el agente es incapaz de eliminar al enemigo. Después, poco a poco es capaz de empezar eliminarlo una vez y en muy pocos episodios consigue una mejora sustancial hasta llegar a eliminar cuatro veces al *zealot* por cada episodio.

Para entender este repentino cambio en la gráfica hay que explicar qué es lo que va ocurriendo durante el entrenamiento del agente. En los primeros episodios, el agente

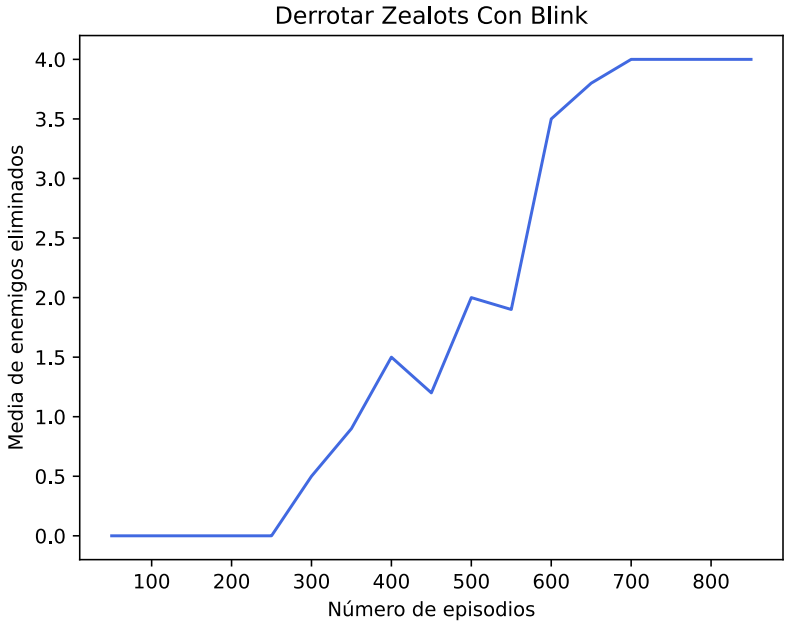


Figura 6.2: Evolución de la media de enemigos eliminados por episodio

poco a poco aprende a hacer *kiting* en alguna de las ocho direcciones. El problema surge debido a que cuando se acerca a las paredes tarda mucho más en llegar a aprender a alejarse de estas. Los primeros picos de conseguir eliminar una o dos unidades reflejan cuando aprende a hacer *kiting* en una de las cuatro paredes. En el momento en el que consigue hacerlo en las cuatro, el *stalker* nunca llegar a morir y puede llegar a eliminar cuatro veces a la unidad enemiga. Tarda aproximadamente 25 segundos en poder hacerlo y en algunas ocasiones llega incluso a quitarle la mitad de la vida al quinto *zealot*

De nuevo, se cuenta tanto con la gráfica de pérdida del modelo (figura 6.3) como de la tasa de aciertos (figura 6.4). En la primera, podemos comprobar como la pérdida tiende a la baja, con algunos picos que pueden estar asociados a los momentos donde empieza a aprender a cómo evitar los límites del escenario. Al contrario, la tasa de aciertos cada vez tiene un valor mayor, llegando a acercase a más de 0.9.

En definitiva, se concluye que el agente ha aprendido con éxito la técnica de *kiting* y se puede afirmar que su rendimiento es prácticamente inmejorable. Poder eliminar a 5 enemigos es imposible por falta de tiempo, lo que significa que el agente está llegando al mejor resultado que es capaz. Cabe destacar que en *StarCraft II* casi nunca se controlan unidades individualmente, si no que se emplea la técnica de *kiting* con un grupo completo de unidades y este mismo agente no está limitado a controlar una única unidad, es decir, podría controlar todo un ejército de *stalkers* con el mismo rendimiento.

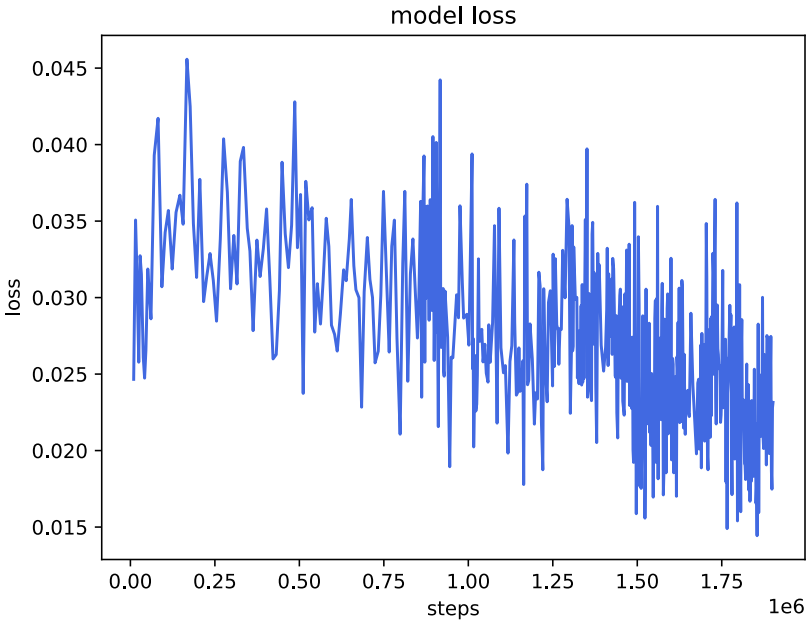


Figura 6.3: Pérdida del modelo uno contra uno

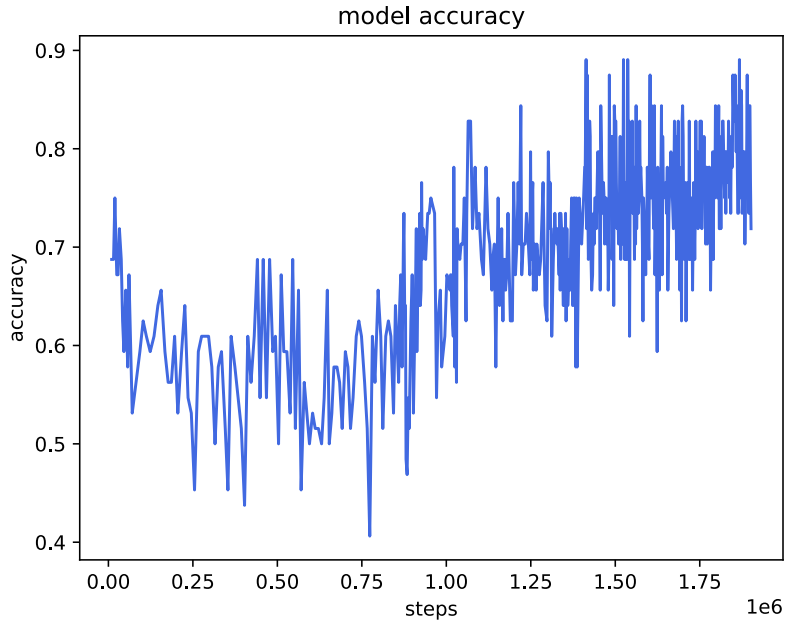


Figura 6.4: Tasa de aciertos del modelo uno contra uno

Figura 6.5: Entorno del minijuego *Derrotar zealots con blinks* uno contra dos

6.2. Uno contra dos

El objetivo principal de este minijuego es derrotar el mayor número de veces a los enemigos sin ser derrotado. Una vez que todos los enemigos han sido derrotados, se genera un grupo nuevo de enemigos a derrotar mientras que la unidad aliada se mantiene con la misma cantidad de vida, originando una batalla infinita hasta que el aliado es derrotados o hasta que el tiempo termina.

En la primera prueba, uno contra uno (sección 6.1), se enfrentaban en un espacio controlado un *stalker* y un *zealot*, controlando el agente a la primera unidad. Ahora, en esta variante del minijuego la unidad del agente, el *stalker*, debe enfrentarse a dos de las unidades enemigas, teniendo que tomar la decisión de qué acción realizar considerando a los dos enemigos en cada estado del juego. Cabe destacar que aunque las pruebas se han realizado empleando únicamente dos enemigos, se puede aumentar el número de *zealots* en el minijuego sin necesidad de modificar el código del agente.

En cuanto al enfoque humano, la situación es bastante similar a la del anterior experimento. La diferencia principal es que en esta ocasión el daño que potencialmente recibirá el *stalker* es el doble y se pueden dar situaciones en las que verse acorralado sea mucho más sencillo.

Es importante que a la hora de realizar el ataque se seleccione a la unidad enemiga con menos vida, ya que, como se explica en la sección 3.2.2, los *protoss* tienen un escudo que se va regenerando con el tiempo y cuanto menos daños reciba más posibilidades hay de que se regenere.

Por último, la mayor complicación es que, al tener que enfrentarse a dos enemigos, es sencillo que estos puedan acorralar al *stalker*. Las esquinas son un lugar peligroso en este enfrentamiento debido a que la mitad de las direcciones en una esquina están bloqueadas y en el resto pueden estar colocados los dos enemigos. Un jugador intentaría evitar acercarse a cualquier esquina y en general a las paredes.

6.2.1. Acciones

Las acciones son moverse en las 8 direcciones ya mencionadas: las 4 básicas y las 4 diagonales. Además, otra de las acciones clave para el *kiting* y para llevar a cabo el objetivo de este minijuego es el ataque, el cual de usarse cuando el agente pueda realizar el ataque. Respecto al uno contra uno, ahora hay dos posibles objetivos. Así pues, se opta por que siempre se seleccione al enemigo con menos vida.

6.2.2. Estados

Reciclando la idea de los estados en *Moverse a la baliza*, se emplea la idea de los ángulos (figura 5.2) para localizar la posición de los enemigos, disponiendo de 8 números que representan estos ángulos.

Cada uno de estos números toma un valor de 0 a 1 representando la distancia que hay entre el agente y los enemigos, modificando el valor de los números según los ángulos en los que se encuentre cada *zealot* del *stalker*. El 0 representa un enemigo lejano y el 1 cercano. En el caso de que algún enemigo coincida con otro en el ángulo, se guarda el valor del enemigo que se encuentre más cercano. Por lo tanto, a diferencia del anterior experimento, se activa más de una neurona para indicar que hay dos enemigos al mismo tiempo.

6.2.3. Recompensa

En cuanto a las recompensas, son las mismas que en el experimento anterior. Están detalladas en la sección 6.1.3.

6.2.4. Configuración del algoritmo

La configuración del modelo utilizado para obtener este aprendizaje ha sido la siguiente. De nuevo se cuenta con 2 capas de nodos ocultos, sin embargo, al tener una mayor complejidad en los estados posibles, la primera capa cuenta con 125 nodos y en la segunda 150. La única capa que no cuenta con función de activación *ReLU* es la capa de salida, por lo que se obtendrán valores-Q sin ningún cambio. Las especificaciones del algoritmo de Deep Q-Learning no se han visto modificadas respecto al anterior experimento, por lo que se pueden encontrar en la sección 6.1.5.

6.2.5. Análisis

Una vez se ha comprendido lo que supone enfrentarse a dos enemigos en vez de a uno, se puede pasar a analizar los resultados obtenidos.

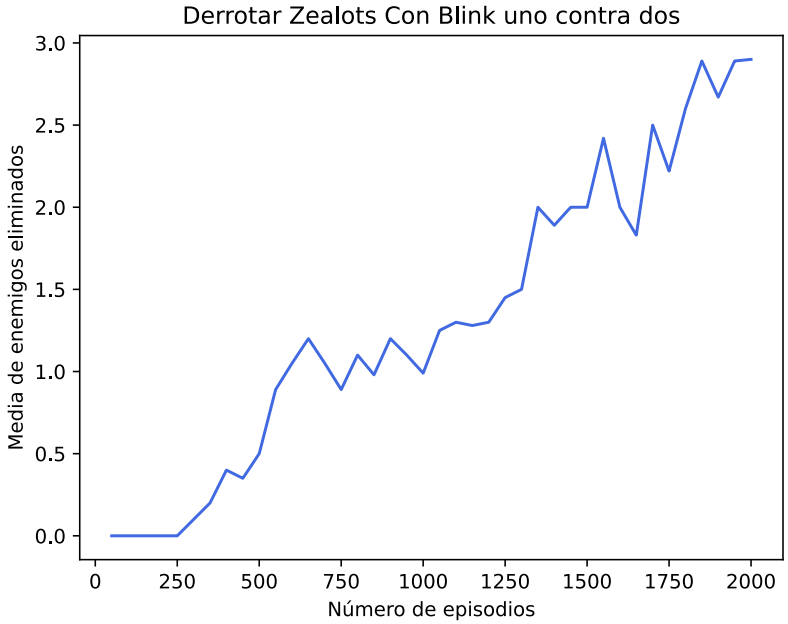


Figura 6.6: Evolución de la media de enemigos eliminados por episodio

La figura 6.6 muestra como ha aprendido el agente a luchar contra dos enemigos. Lo primero que se puede observar es que el agente nunca llega a eliminar a cuatro enemigos a diferencia del anterior minijuego. Esto se debe a que pelear contra dos enemigos y luego contra otros dos, es mucho más complicado que pelear contra uno individualmente cuatro veces seguidas.

Por otro lado, para llegar a estos resultados se han necesitado 1300 capítulos más que en el anterior experimento. En total se han necesitado alrededor de 2000 capítulos, que en tiempo real son 66 horas entrenando. Esto se debe al amplio número de nuevos posibles estados respecto a los que se podían encontrar anteriormente. Al estar en una batalla contra dos unidades al mismo tiempo no solo va a tener que aprender los casos en los que se encuentra en una de las paredes, si no que cuando esté rodeado por estos enemigos tendrá que encontrar cuales son las mejores opciones para huir de ellos.

Para que este agente sea capaz de eliminar a un solo enemigo de manera estable hace falta el mismo tiempo de entrenamiento que con el anterior agente para que consiguiera eliminar a cuatro. Doblando el tiempo de entrenamiento, empieza a ser capaz de eliminar a los dos enemigos iniciales, lo cual es un desafío. Para comprobar esta dificultad, se realizó una prueba con varias personas, incluyendo los autores de este trabajo, para confirmar cuántos enemigos eran capaces de eliminar. Ninguna de las personas que jugaron el minijuego fueron capaces de eliminar a los dos enemigos iniciales, teniendo varios intentos para repetir el minijuego.

A partir de los 2000 episodios, el agente llega a su mejor marca, la cual es eliminar a los dos enemigos iniciales y conseguir eliminar a uno de los dos que reaparecen. Cabe destacar que, al igual que en el anterior experimento, cuando reaparecen los dos enemigos la unidad aliada mantiene la misma vida que tuviera. Sabiendo que cuando consigue eliminar a los dos primeros le suele quedar muy poca vida, es entendible que

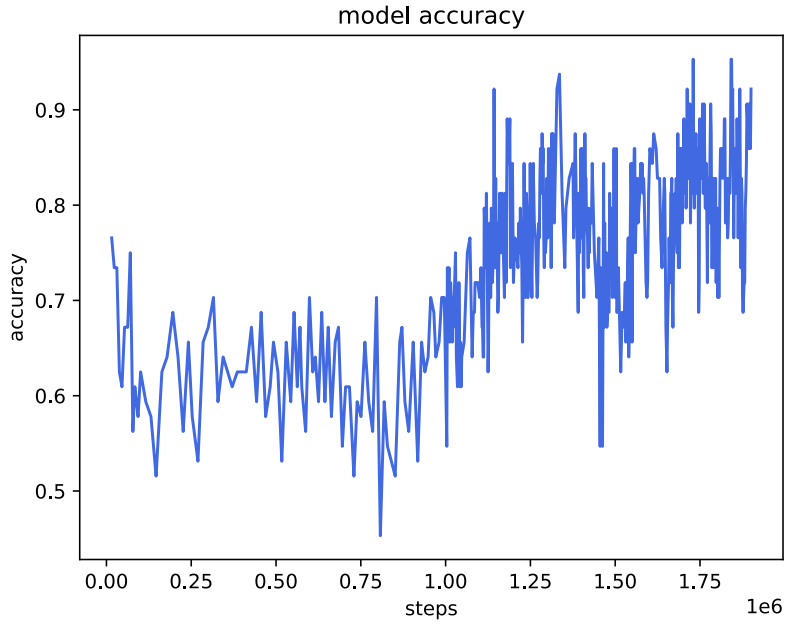


Figura 6.7: Tasa de aciertos del modelo

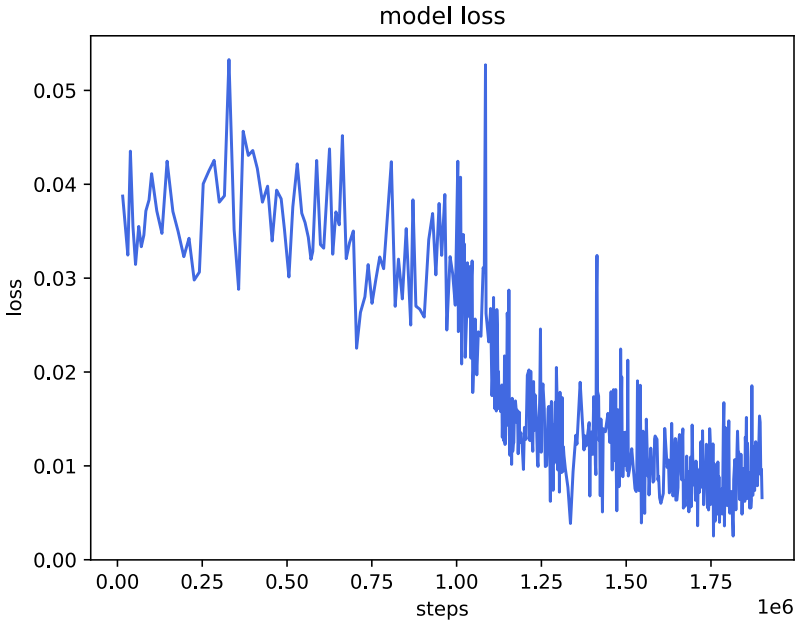


Figura 6.8: Pérdida del modelo

sólo sea capaz de eliminar a tres unidades.

De igual manera se pueden observar las gráficas que muestran cómo ha evolucionado el modelo. En la figura 6.7 se puede comprobar que hasta la mitad del entrenamiento la tasa de aciertos del modelo es bastante baja e incluso va empeorando. Esto sigue así hasta llegar al momento donde el agente empieza a aprender las distintas situaciones que se dan en el minijuego, llegando a una tasa de aciertos bastante buena.

Por último, en la figura 6.8 podemos ver cómo la pérdida desciende drásticamente en el mismo momento en el que la precisión empieza a subir, por lo que podemos concluir que el modelo tiene un buen rendimiento durante el entrenamiento.

En conclusión, el agente es capaz de eliminar a tres unidades enemigas después de un arduo entrenamiento. Teniendo en cuenta que eliminar a los dos enemigos iniciales ya es algo complicado, se puede decir que el agente hace un muy buen trabajo enfrentándose a los dos enemigos. Aunque el agente siga entrenando por muchos más capítulos, no se ha conseguido que elimine a dos tandas de dos enemigos sin morir o antes de que se acabe el tiempo debido a que si el agente es muy agresivo pierde demasiada vida y en la segunda ronda muere. Sin embargo, si juega muy defensivo, alejándose mucho de los enemigos para poder disparar sin recibir daño, pierde demasiado tiempo y el episodio termina antes de conseguir eliminar a tres enemigos. Por lo tanto, el agente parece haber encontrado un equilibrio entre agresividad y el estilo más defensivo.

6.3. Dos contra dos

Dos contra dos es otra de las variantes del ya explicado minijuego *Derrotar zealots con blinks* (sección 3.4.1). Se mantiene el mismo objetivo del minijuego original, pero a diferencia de las pruebas realizadas anteriormente, en esta variante se cuenta con dos *zealots* enemigos que se enfrentan a dos unidades aliadas. Esto genera una situación diferente de las pruebas antiguas, provocando un planteamiento distinto.

Anteriormente se necesitaba un agente que controlase la única unidad *stalker* existente en el entorno, centrándose en el aprendizaje de esta unidad y en sus acciones. Ahora se cuentan con dos unidades aliadas, controlada cada una por un agente distinto los cuales deben aprender cada uno por su cuenta cuáles son las estrategias idóneas en cada estado del juego, teniendo en cuenta a la otra unidad aliada y surgiendo la necesidad de coordinarse entre ellas.

Debido al funcionamiento del componente *PySC2* (sección 3.4), las unidades de los agentes deben ser distintas, pues no se puede distinguir entre las unidades de un mismo grupo de unidad. Se ha optado por usar como la segunda unidad aliada a una unidad *roach* debido a su capacidad para disparar a distancia y a la similitud de velocidad de movimiento que tiene con la otra unidad, el *stalker*.

Una persona seleccionaría a ambas unidades y realizaría la técnica de *kiting* con las dos unidades seleccionadas al mismo tiempo, pero en ningún caso controlaría a

cada unidad de manera independiente ya que sería prácticamente imposible, por lo que *StarCraft II* ofrece la opción de controlar a más de una unidad al mismo tiempo. Pero si cada unidad enemiga siguiera a una unidad aliada distinta, una persona no podría realizar *kiting* con cada una de las unidades independientemente y esto puede ser una desventaja.

A diferencia de un jugador, los dos agentes pueden explorar todas las estrategias que deseen, moverse sin dejar mucho espacio entre ellos, separarse completamente para también separar a los enemigos y distintas estrategias que se podrán apreciar más adelante.

En cuanto a la dificultad, para un jugador es similar a la que se pudo encontrar en el experimento de uno contra dos. En cambio, para los agentes tener que aprender a coordinarse y encontrar las mejores opciones puede resultar mucho más complicado. Cabe destacar que la segunda unidad aliada, el *roach*, tiene características similares a las del *stalker* que ya conocemos pero es más débil respecto a este.

Por último, se puede observar como en la figura 6.9 el escenario está cubierto por una capa grisácea. Esto es la biomateria *zerg* que se explica en la sección 3.2.2, ya que la segunda unidad aliada pertenece a la raza de los *zerg* y las características de la unidad mejoran dependiendo de si esta biomateria está presente o no.

Figura 6.9: Entorno del minijuego *Derrotar zealots con blinks* dos contra dos

6.3.1. Acciones

Con el ánimo de experimentar el nivel de coordinación que pueden llegar a tener ambos agentes, se ha decidido ampliar las acciones respecto al minijuego de *Derrotar zealots con blinks uno contra dos* (sección 6.2.1).

Anteriormente, los agentes podían moverse en las 8 direcciones y atacar al enemigo con menor vida. Pero ahora se dan situaciones en las que los enemigos se separan y cada unidad aliada tiene cerca a uno de estos, por lo que podría no ser una buena situación para atacar al enemigo con menos vida.

Si, por ejemplo, se está en la situación en la que las dos unidades aliadas están separadas y cada una tiene la atención de un enemigo, si solo existiera la acción de atacar a la unidad con menos vida, a una de las dos unidades aliadas se le estaría obligando a recorrer una gran distancia para poder atacar e incluso puede que tuviera que pasar por delante del enemigo que tiene más cerca para poder llegar a la unidad enemiga con menos vida.

Por esto y con la intención de que los agentes encuentren las mejores estrategias posibles, se añade una nueva acción relacionada con el ataque. Además de poder atacar a la unidad que tenga menos vida, cada agente puede atacar a la unidad enemiga más cercana. Gracias a esto, si se encontrasen en la situación planteada anteriormente, cuando ambos agentes están separados pueden evaluar si es un buen momento para atacar a la unidad con menos vida y tener que desplazarse o si es mejor atacar a la unidad más cercana.

Por otra parte, al ser dos agentes independientes se espera que tomen las decisiones correctas en situaciones en las que favorece no estar controlados por un mismo jugador. Por ejemplo, si los enemigos consiguen colocarse entre las dos unidades aliadas y se tiene las dos seleccionadas, todas la posibles decisiones serían negativas. Supongamos que las unidades están dispuestas de la siguiente manera. En una línea vertical, en la parte superior se encuentra una unidad aliada, por debajo en esa misma línea se encuentra los dos enemigos y aún más abajo se encuentra la segunda unidad aliada. Las decisiones de mover ambas unidades hacia arriba o hacia abajo van a resultar en una gran cantidad de daño recibida por parte de una de las dos unidades, en cambio, huir hacia la derecha o la izquierda resultaría en un daño parecido en las dos unidades aliadas, ya que probablemente un enemigo atacase a una unidad aliada y el otro enemigo a la otra unidad. Si cada unidad está controlada independientemente por un agente, se busca que en esta situación de ejemplo cada una de las unidades huya en direcciones opuestas, ya que esta es la única manera de que ambas puedan salir de esa situación sin recibir daño.

6.3.2. Estados

Como se ha explicado anteriormente, dos contra dos cuenta con dos unidades aliadas las cuales se enfrentan en conjunto a dos enemigos (*zealots*). Estas unidades deben aprender a coordinarse entre ellas para conseguir el objetivo del minijuego y esta coordinación no es posible si no saben nada la una de la otra, como se va a explicar en la siguiente situación.

Se supone que tanto el *roach* como el *stalker* se encuentran pegados el uno al otro en el centro del mapa, el primero a la derecha del segundo, y ambos *zealots* están situados a la derecha de las unidades aliadas. Por como están colocados los enemigos,

el *stalker* no ve la necesidad de huir debido a que aún están los enemigos a una distancia segura, pero el *roach* sí necesita escapar. En este caso, esta segunda unidad intenta huir por la izquierda debido a la proximidad de los enemigos pero no puede porque le está bloqueando el camino el *stalker*. Debido a esto los enemigos terminan acercándose al *roach* y empiezan a atacarle.

Para solucionar problemas como este, en los estados de cada agente se añaden 8 variables continuas entre 0 y 1. Estas variables emplean el mismo método de ángulos utilizado con los enemigos pero, en vez de representar las posiciones de los *zealots*, en esta ocasión estas variables representan la posición en la que se encuentran el resto de aliados. Además, son variables continuas debido a que no solo expresan la dirección sino también la distancia de ese aliado respecto a la unidad del agente.

Con esta modificación, el problema mencionado anteriormente queda resuelto y ambas unidades aliadas pueden evitar obstaculizarse entre ellas, fomentando la coordinación en sus movimientos.

6.3.3. Recompensas

Las recompensas de este minjuego siguen la esencia del minijuego *Derrotar zealots con blinks* (sección 6.1.3), añadiendo una pequeña modificación. Ahora, además de conseguir recompensa cuando se daña al enemigo y disminuir la recompensa si la unidad ha sido herida, el agente se lleva dos puntos más de recompensa si se ha conseguido derrotar a uno de los enemigos. Esta modificación es necesaria para favorecer la coordinación entre los agentes.

Debe de quedar claro que las recompensas que reciben los agentes son independientes para cada uno de ellos, al igual que ocurre con las acciones y los estados. Si las recompensas se compartieran entre los dos agentes podrían ocurrir situaciones no deseadas y por lo tanto el aprendizaje podría verse comprometido como se explica a continuación.

Se supone que uno de los dos agentes por alguna razón ha llegado a un punto bastante avanzado del aprendizaje mientras que el segundo agente aún necesita mucho más tiempo para llegar a ese estado. Si las recompensas fueran compartidas entre ambos, el agente aventajado estaría generando recompensas muy positivas constantemente y eso significa que el segundo agente se podría estar viendo recompensado por cualquier tipo de acciones ya fuesen buenas o malas. De la misma forma, el agente que sí lo está haciendo bien recibiría las recompensas negativas en acciones que son correctas.

Para evitar este tipo de situaciones que se pueden dar y podrían dificultar el aprendizaje, cada uno de los agentes recibe recompensas independientemente de lo que haya recibido el otro. Esto significa que los agentes tienen que explorar las distintas situaciones intentando maximizar su recompensa, lo que idealmente se debería de traducir en que los agentes realicen acciones coordinadas.

6.3.4. Configuración del algoritmo

Cada uno de los agentes tiene un modelo idéntico, para que en ese sentido no haya ninguna diferencia. En este caso, se contará con 2 capas de nodos ocultas, la primera tiene ciento 150 y la segunda 200. Se ha aumentado el número de nodos respecto al anterior minijuego debido a la modificación del número de estados posibles. A todas las capas de nodos menos a la de salida se les aplica la función de activación *ReLU*. En esta ocasión, las características del algoritmo de Deep Q-Learning sí se han visto modificados.

Las especificaciones del algoritmo Deep Q-Learning para obtener estos resultados han sido las siguientes. Cada 15 episodios se actualizan los pesos de la red neuronal objetivo con los pesos de la red de predicción. Además, se almacenan 1052 experiencias y se eliminan 128 al llegar al límite establecido y se eligen de forma aleatoria 128 experiencias para entrenar a la red neuronal.

6.3.5. Análisis

Lo primero que se debe tener en cuenta es que la unidad que controla el agente número dos, el *roach*, es mucho más débil que su compañero el *stalker*. En la mayoría de partidas ambos enemigos se enfocan en alguno de los dos agentes. Cuando ocurre esto, el agente que está siendo perseguido emplea la técnica de *kiting* mientras el otro agente, al no estar en peligro, ataca todas la veces que puede.

La otra opción es que cada uno de los enemigos se fije en cada uno de los agentes, lo que suele resultar en que cada uno de estos emplea la técnica de *kiting* independientemente contra cada uno de ellos.

En la gráfica 6.10 se puede apreciar el número de enemigos que eliminan los dos agentes por cada episodio. A diferencia del anterior experimento en el que el agente no llegaba a eliminar nunca a 4 enemigos, en esta ocasión se puede observar que, aunque no consigue hacerlo de manera estable, sí llega a eliminar a 4 en algunas ocasiones. Esto se debe a la disparidad de situaciones que se dan en las distintas partidas.

Se puede apreciar la cantidad de episodios necesarios para llegar a estos resultados, que han sido 4000. Prácticamente dos veces lo necesitado en el anterior minijuego, siendo en tiempo real 133 horas. Si se aumenta el tiempo de entrenamiento no se llega a mejores resultados. Se observa que para eliminar a dos enemigos de manera estable se necesitan cerca de 2000 episodios. Esto está infludo por la debilidad de la unidad *roach* en comparación con el *stalker*, lo que implica que muere mucho más rápido que este.

Es importante tener en cuenta que al eliminar a los dos enemigos, si una de las unidades aliadas ha muerto, esta no reaparece. Solo reaparecen los enemigos. Sabiendo que normalmente la unidad *roach* suele morir en el primer combate, se puede entender que solo en algunas ocasiones son capaces de eliminar a cuatro enemigos.

Por otro lado, al igual que en anteriores minijuegos se tienen las gráficas de tasa

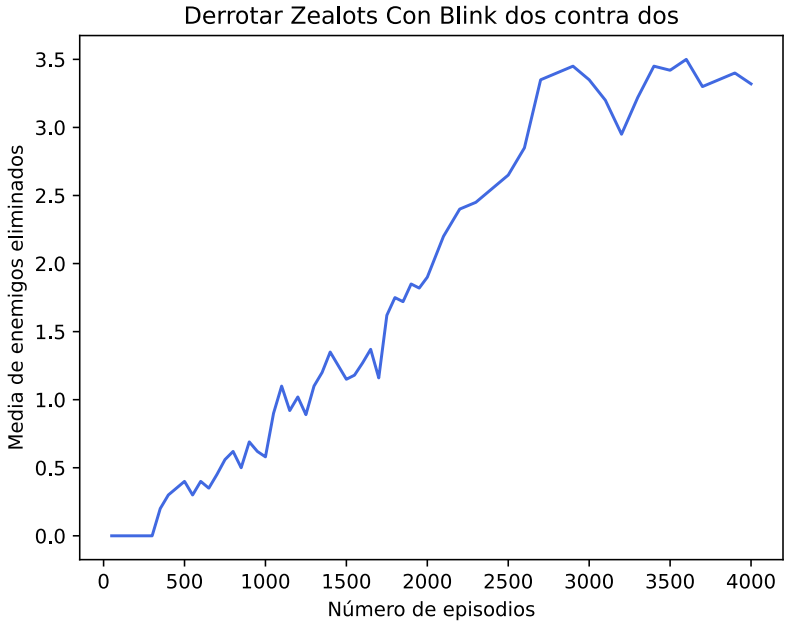
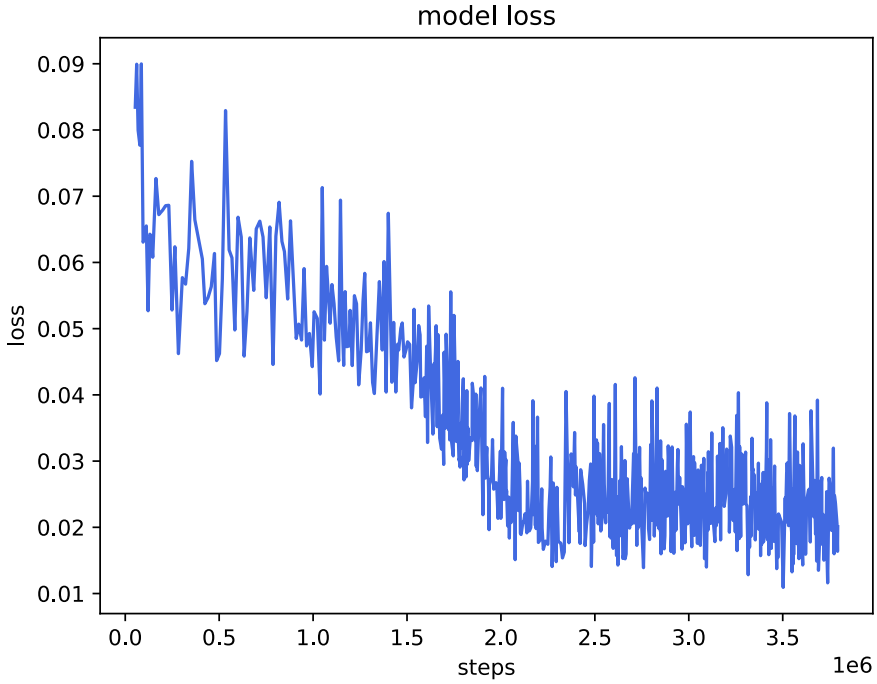
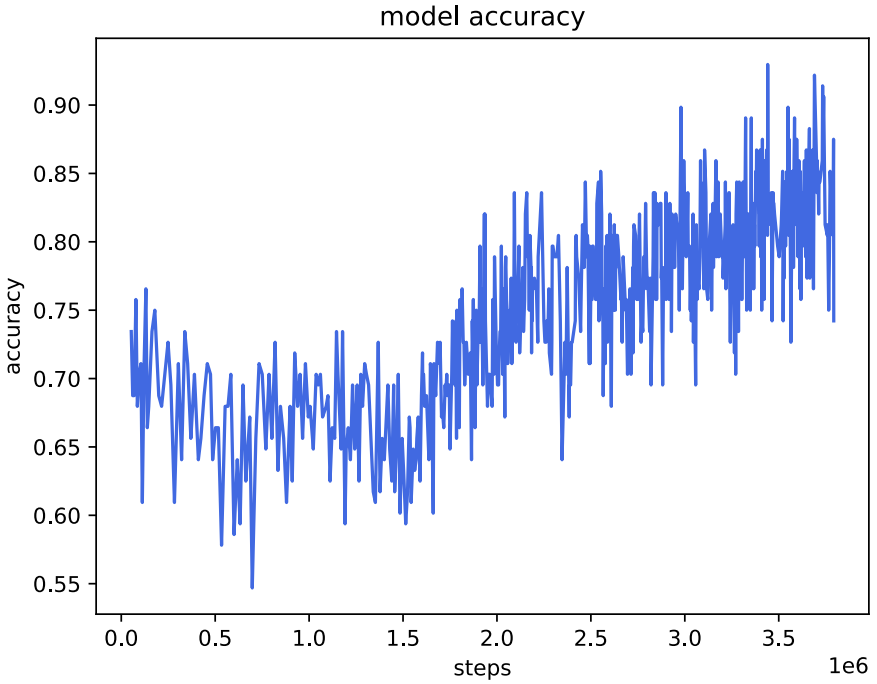


Figura 6.10: Evolución de la media de enemigos eliminados por episodio

de aciertos y de pérdida, solo que en este caso son dos ya que hay dos agentes, 6.11, 6.12. Se puede observar cómo las dos gráficas de pérdida evolucionan de forma similar al igual que con las dos gráficas de tasa de aciertos.

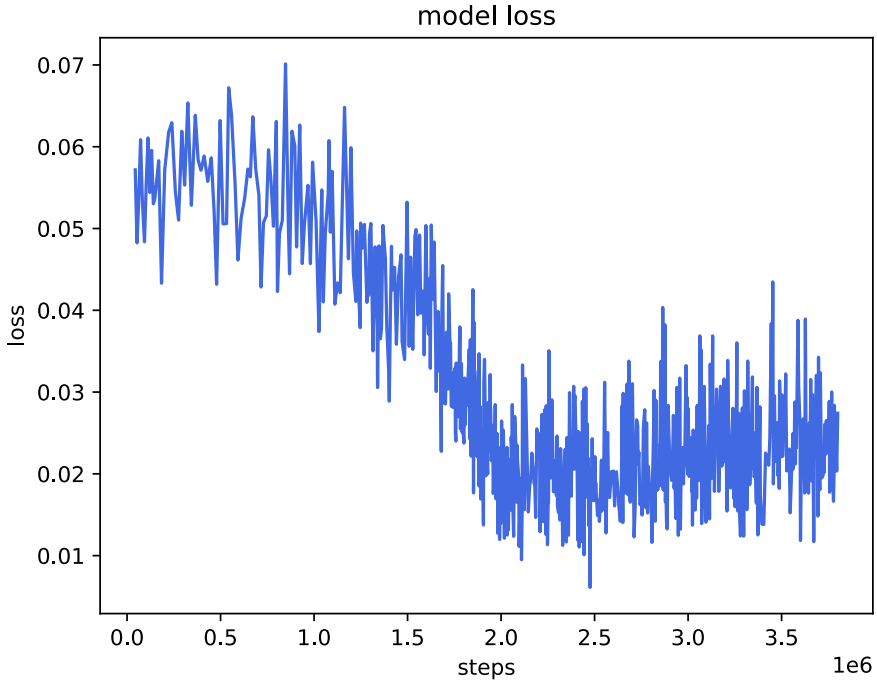


(a) Pérdida del modelo

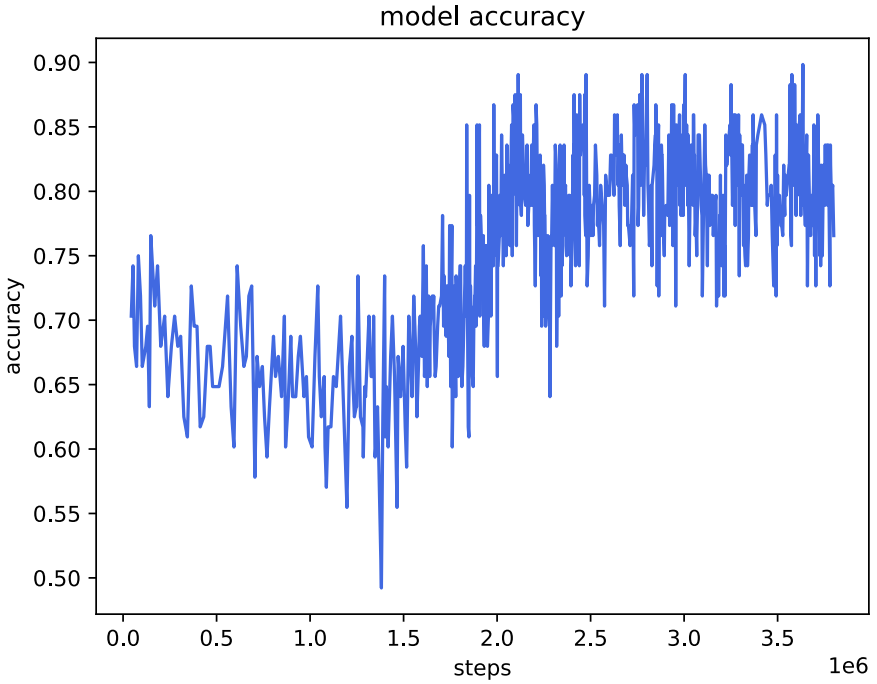


(b) Tasa de aciertos del modelo

Figura 6.11: Pérdida y tasa de aciertos del agente *stalker* en dos contra dos



(a) Pérdida del modelo



(b) Tasa de aciertos del modelo

Figura 6.12: Pérdida y tasa de aciertos del agente *roach* en dos contra dos

Capítulo 7

Construir marines

Figura 7.1: Entorno del minijuego *Construir* marines

Derrotar zealots con blinks era un claro ejemplo de un entorno en el que se necesitaban decisiones reactivas. En cambio, en este último experimento, *Construir* marines, se explorará un entorno más complejo que los anteriores en el que es necesaria la toma de decisiones estratégicas y, por lo tanto, se entra en el ámbito de macrojuego (sección 3.1.3).

En este minijuego, el objetivo principal es construir el mayor número de *marines* posibles, como indica su nombre. Se dispone de quince minutos hasta que este acabe. El aumento de tiempo respecto a los anteriores entornos se debe al aumento de complejidad para alcanzar el objetivo. Todos los elementos implicados en una partida real de *StarCraft II* a la hora de construir *marines* están presentes. Se comienza con un centro de mando y doce trabajadores que están asignados a la recogida de minerales desde el inicio. Por lo tanto, se tiene que tener en cuenta el número de minerales que se tienen y además emplear los suministros, que son uno de los tres recursos de *StarCraft II* e que indican el límite de población (sección 3.2.1).

La única manera de aumentar el número de minerales es recolectándolos. Por otro lado, si se quieren aumentar los suministros es necesario construir uno de los edifi-

cios disponibles, los depósitos de suministros. También es necesaria la construcción de barracones para poder reclutar a los *marines*.

Como se ha explicado anteriormente, las decisiones estratégicas son importantes en este minijuego. Primero, un jugador debe de saber que para construir barracones al menos es necesario tener un depósito de suministros. Una vez se tiene un barracón se puede empezar a construir *marines*, aunque en principio cuantos más barracones haya construidos más *marines* se pueden crear paralelamente.

El número de depósitos de suministros está limitado a 16 mientras que el número de barracones se limita a 8. El número de trabajadores que se pueden crear no está limitado. Al inicio de la partida se debe de tomar la decisión de crear más trabajadores para recolectar más minerales, aunque una vez haya 16 recolectores ya no se podrán añadir más.

Después viene la fase de construcción. En este momento se deben de tomar varias decisiones importantes, las cuales consisten en crear muchos trabajadores para poder construir edificios paralelamente, seleccionar a los trabajadores que están recolectando y darles una nueva tarea construyendo o tener pocos trabajadores que construyan poco a poco todos los edificios. Puede parecer que la mejor opción es construir paralelamente la mayor cantidad de edificios, pero eso significaría un gran coste en la creación de trabajadores y un gran aumento en la población, lo que se traduce en menos espacio para los futuros *marines*.

Por otro lado, una vez se tienen construidos todos los barracones, hasta que no quede pocos espacios de población no parece que sea necesario seguir construyendo depósitos de suministros, ya que construir los 16 es muy costoso y es muy difícil llegar a ocupar todos los espacios de población que se han añadido.

Por lo tanto, hace falta tomar una serie de decisiones estratégicas que pueden llegar a ser complicadas. La decisión que parece ser más clara es la de tener a 16 trabajadores recolectando minerales continuamente. Todo el minijuego se basa en la construcción o creación de unidades y para ello es necesario una gran cantidad de minerales, por lo que tener a menos trabajadores recolectando se traduce en menos *marines* construidos en el futuro.

En cuanto a la construcción de edificios las distintas opciones tienen sus ventajas y desventajas. Lo que parece claro es que tener 8 barracones construidos cuanto antes, debería ser la mejor opción ya que se podría empezar a crear 8 *marines* en paralelo. El problema es que para tener los 8 barracones construidos lo antes posible es necesario construirlos también en paralelo y esto significa que o bien habría que sacrificar trabajadores que estén recolectando o crear un gran número de trabajadores que solo estén enfocados en la construcción. El problema, es que una vez esos trabajadores han terminado sus labores de construcción, no tendrán ninguna tarea que realizar y solo servirán para aumentar la población.

En conclusión, para un jugador las decisiones estratégicas a tomar pueden ser bastante complejas y difíciles de acertar, ya que es sencillo llegar a ciertos momentos en los que queden pocos minerales o haya muchos trabajadores ocupando la población.

7.1. Acciones

El objetivo de este minijuego es crear el mayor número de *marines* posibles en el tiempo indicado. Para ello, se ha decidido crear una serie de acciones abstractas que el agente pueda elegir. Estas acciones pueden estar compuestas por más de una acción en lo que respecta a *StarCraft II*.

La primera acción, es mandar a un trabajador inactivo a recolectar minerales. Como ya se sabe, es importante mantener un número elevado de minerales, por ello se deber permitir al jugador automático poder ordenar a los trabajadores recolectar minerales. Específicamente se trata de los trabajadores inactivos ya que, si no es un trabajador inactivo, estará recolectando minerales o bien construyendo un edificio y a los últimos no se les puede cortar su tarea. Esta acción está compuesta por la selección del trabajador inactivo y por la orden de ir a un mineral a recolectar. En el caso de que no haya trabajadores inactivos o de que se haya alcanzado el límite de 16 trabajadores recolectando, esta acción no puede realizarse.

La siguiente acción será crear un depósito de suministros con un trabajador inactivo. Es necesario tanto para poder construir barracones como para poder aumentar la población, por lo que esta acción debe de poder realizarse en todo momento. En este caso, la acción está compuesta por la selección del trabajador inactivo y la acción de construir un depósito de suministros en un lugar en específico. Esta acción no puede realizarse si no hay ningún trabajador inactivo o si no se tiene el número necesario de minerales.

Para las construcciones de distintos edificios, se ha optado por almacenar cuáles son las posiciones en las que los distintos edificios son construidos. Esto significa que el lugar de construcción de los edificios no es arbitrario y se ha establecido conociendo la limitación de construcción de 16 depósitos de suministros y 8 barracones.

La próxima acción sirve para construir barracones con trabajadores inactivos. Los barracones son los edificios donde se pueden entrenar los *marines*, por lo que es necesario poder construirlos. La acción se compone por la selección de un trabajador inactivo y la orden de construir un barracón en un lugar específico. En este caso la acción no puede realizarse si no hay ningún trabajador inactivo, no hay suficientes minerales o si no hay al menos un depósito de suministros construidos.

Las dos acciones siguientes son construir depósitos de suministros y barracones respectivamente, con la diferencia de que en este caso los trabajadores seleccionados no están inactivos si no que recolectan minerales. Estas acciones se añaden ya que se puede llegar a un estado en el que no se puede avanzar. Si la población llega al límite y todos los trabajadores están recolectando minerales nunca se podría crear un nuevo depósito de suministros y tampoco se podrían crear nuevos trabajadores, resultando en no poder avanzar más en el minijuego.

Por otro lado están las acciones de crear trabajadores y de crear *marines*. Ambas son necesarias para llegar a un buen resultado. Con los trabajadores se pueden recolectar minerales a una mayor velocidad y construir los edificios y con los *marines* se

llega al objetivo deseado. La acción de construir un trabajador no puede realizarse si no quedan suficientes minerales o si no hay suficiente espacio en la población. Para construir *marines* se le añade la condición de que haya al menos un barracón que no esté ya reclutando *marines*.

Por último, se añade una acción que no hace nada. Esto sirve para muchos casos en los que puede ser mejor no hacer nada antes de gastar más recursos. Esta acción se puede realizar en cualquier momento. En las situaciones en las que no se puede realizar alguna de las acciones seleccionadas, se realiza la acción de no hacer nada.

7.2. Estados

Para este experimento se ha intentado equilibrar la representación del estado de tal manera que la complejidad esta en un punto medio entre poder representar todos los estados importantes y que ese espacio de estados posibles no sea tan grande como para que haya algún problema al explorarlos todos de forma correcta.

Primero, hay un valor que indica el número de trabajadores inactivos que hay es ese momento. Con esto se pretende conseguir que el jugador automático sea capaz de diferenciar en qué momentos puede o no realizar acciones con estos trabajadores y que decida si es buena idea crear más.

También se tiene un valor que indica el número de trabajadores que están recogiendo minerales. Como se ha explicado anteriormente, el número de minerales es lo que genera la posibilidad de realizar la mayoría de acciones, por lo que parece lógico que el jugador automático maximice el número total de trabajadores que recogen minerales.

Por otro lado, es importante tener en cuenta cuántos espacios libres quedan en la población. En los momentos en los que no queda ningún espacio libre no se puede crear ninguna unidad, ya sea un *marine* o un trabajador, por lo que es importante que el jugador automático sea capaz de detectar cuando le quedan pocos espacios y así poder construir depósitos de suministros y elevar el límite de población.

Al igual que con el número de espacio libres de población, es muy importante conocer el número de minerales disponibles. Dependiendo de la cantidad de minerales se pueden realizar las distintas acciones de creación de unidades y de construcción de edificios. Si se tiene en cuenta el valor anterior, que indica el número de trabajadores recolectando minerales, el jugador automático asociaría que al tener un mayor número de trabajadores la cantidad de minerales recogidos aumenta cada *step*.

En los videojuegos *RTS* es muy común encontrarse con acciones durativas (sección 3.1.1) y en este minijuego están muy presentes. Las acciones durativas suponen un reto a la hora del aprendizaje por refuerzo, ya que al realizar estas acciones no se encuentra el cambio en el entorno en el siguiente *step*. Por ejemplo, si se decide realizar la acción de construir un depósito de suministros, la próxima vez que se observe el entorno ese edificio no estaría construido. Lo importante es que para la mayoría de acciones durativas sí que hay un cambio en la observación del entorno. Por ejemplo, si

se construye un edificio se puede observar cómo en la observación del entorno se indica que ese edificio se está construyendo.

Debido a todo esto, se decide ampliar la representación del estado para que se indique la transición en las acciones durativas. En el caso de la construcción de los depósitos de suministros, hay un valor que indica que ese edificio está en construcción. Al igual que con los depósitos de suministros, por cada una de las acciones durativas se tiene un valor que indica si esa acción se está realizando. Por lo tanto, hay dos valores que indican si se están construyendo los dos posibles edificios y otras dos que indican si se está construyendo alguna de las dos posibles unidades.

Por otro lado, hay un valor que indica si ya se ha construido al menos un depósito de suministros. Esto es importante ya que si no hay construido ningún depósito de suministros no se puede construir ningún barracón. Además, hay otro valor que indica el número de barracones que hay construidos hasta el momento.

Por último, hay tres valores que representan si es posible crear *marines*, construir barracones o construir depósitos de suministros. De esta manera, es más sencillo para el jugador automático detectar los estados en los que puede o no realizar estas acciones.

Aunque esta representación del estado es bastante elaborada, puede dar problemas ya que faltan ciertas posibilidades que pueden darse durante el minijuego. Por ejemplo, pueden construirse edificios en paralelo y al sólo tener un único valor que representa si se está construyendo un barracón o no, al construir otro más en paralelo el estado no varía, salvo por el valor que indica la cantidad de minerales.

Una mejor representación del estado incluiría una variable por cada uno de los edificios posibles, indicando si están construidos o no. También habría que aumentar la cantidad de variables que indican si un edificio está en fase de construcción, así se tendría en cuenta la posibilidad de construir edificios en paralelo. Al igual que con los edificios habría que indicar el número de barracones que se están usando para construir *marines*. El problema de esta representación es la cantidad de estados posibles que puede haber, llegando a ser un problema el tiempo necesario para la exploración de todos estos estados.

7.3. Recompensas

Se ha optado por dar una recompensa de 1 al agente cuando se crea un *marine*, de 0 si no se ha creado ningún *marine* y una recompensa negativa al realizar acciones cuando no puede. El problema surge cuando al realizar la acción de crear un *marine* no se crea instantáneamente, sino que tiene que pasar cierto tiempo hasta que termina de crearse (acción durativa). Esto significa que la acción que se ve recompensada puede ser o no la acción de crear un *marine*.

En este minijuego, es muy importante la propagación de la recompensa. La finalidad es que la recompensa se propague a la acción de crear *marine*. Con los estados de transición que se han explicado anteriormente se intenta facilitar que se pueda propagar

la recompensa de forma correcta.

En el caso de la creación de *marines*, al existir el estado que indica si uno se está construyendo o no, el estado que se ve indica que se estaba construyendo un *marine*. Lo importante es que la única acción que lleva al estado en el que se está construyendo un *marine* es la de crearlos, por lo que se puede propagar la recompensa sin problemas. Si no existiera el estado que indica que un *marine* está siendo reclutado, nunca podría propagarse la recompensa a la acción de crearlos ya que esta no cambiaría nada en el estado.

En concreto, se penalizan las acciones de crear *marines* o trabajadores y de construir depósitos de suministros o barracones cuando estas acciones no puedan realizarse, ya sea por falta de minerales u otras razones. Estas recompensas se dan con el ánimo de que el jugador automático sea capaz de aprender en que momentos no es una buena idea realizar acciones que no implican nada.

7.4. Configuración del algoritmo

Debido al aumento de complejidad en la representación del estado, se ha optado por tener un total de 4 capas de nodos ocultos. La primera capa tiene 100 nodos y el resto tiene 20 nodos más que la anterior. Todas ellas tienen la función de activación *ReLu*.

En cuanto a la configuración del algoritmo de *Deep Q-Learning*, cada 15 episodios se actualizan los pesos de la red neuronal objetivo con los pesos de la red de predicción. Se almacenan 2000 experiencias y se eliminan 256 al llegar al límite anterior. Se eligen de forma aleatoria 256 experiencias para entrenar a la red neuronal.

7.5. Análisis

Como se puede comprobar en la imagen 7.2, el mayor número de *marines* construidos que se han conseguido ha sido alrededor de 12. Se han necesitado un total de 850 capítulos para llegar a este resultado. Como se explicó al comienzo, este minijuego dura 15 minutos, por lo que en tiempo real se han necesitado 212 horas. Aún así no se ha llegado a un gran resultado ya que una persona puede llegar a construir alrededor de unos noventa *marines*.

El jugador automático consigue llegar a construir un barracón, pero una vez hay uno construido suele construir *marines* hasta llegar a quedarse sin población. Los dos principales problemas son que tarda demasiado en construir un nuevo depósito de suministros para aumentar la población y que, al no construir habitualmente más de un barracón, nunca hay creación paralela de *marines*. Además tampoco suele tener el máximo número de trabajadores recolectando minerales, por lo que no consigue tener una buena economía.

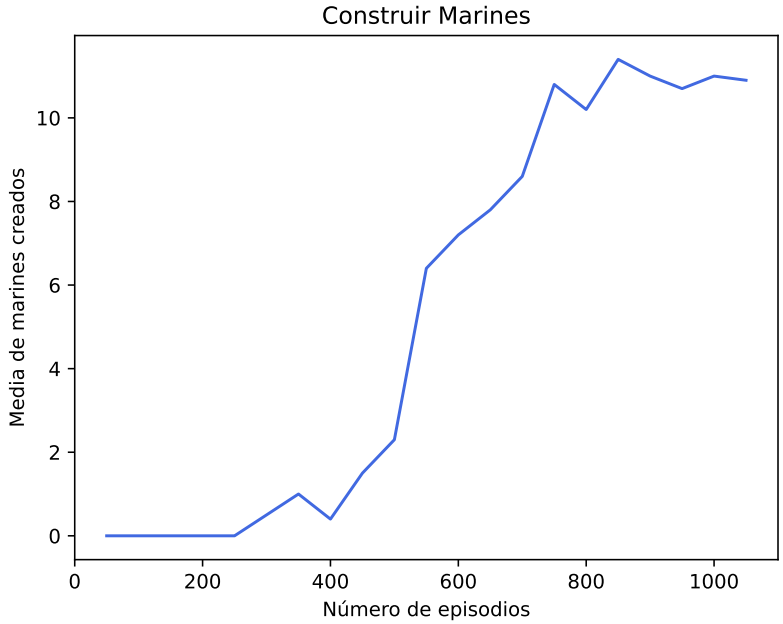


Figura 7.2: Evolución de la media de *marines* construidos por episodio

En la figura 7.3 se puede comprobar cómo la pérdida del modelo va bajando a medida que avanzan los episodios, pero hay picos constantes con los que se puede comprobar cómo el modelo no llega a estabilizarse durante el aprendizaje. Así mismo, en la figura 7.4 se puede comprobar cómo la tasa de aciertos varía continuamente sin llegar a estabilizarse en ningún momento.

Estos resultados principalmente se deben a que es necesario mucho más tiempo para llegar a explorar el espacio de estados de manera correcta. Además, las recompensas deben de propagarse desde los estados finales del minijuego hasta los iniciales, por lo que el tiempo necesario para que la recompensa se propague a lo largo de todos los estados es muy elevado.

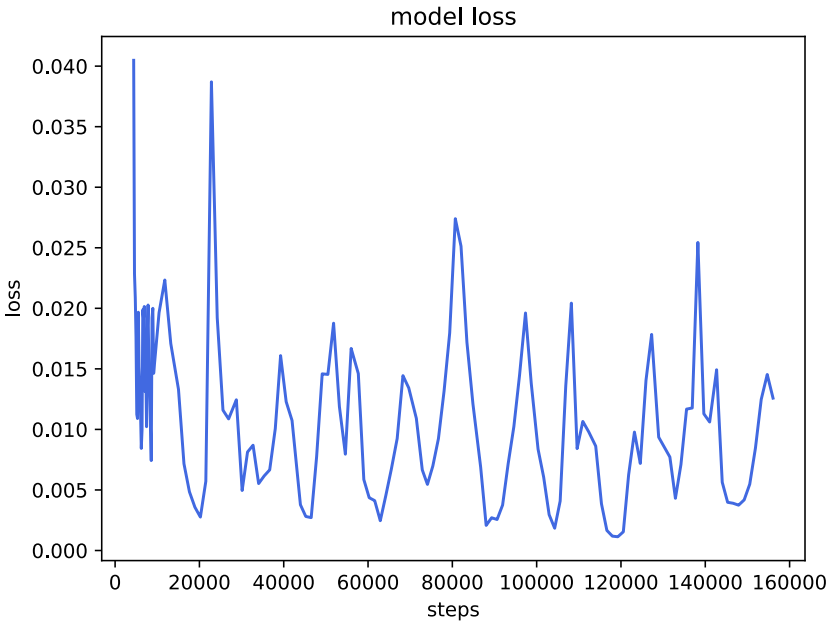


Figura 7.3: Pérdida del modelo de *Construir* marines

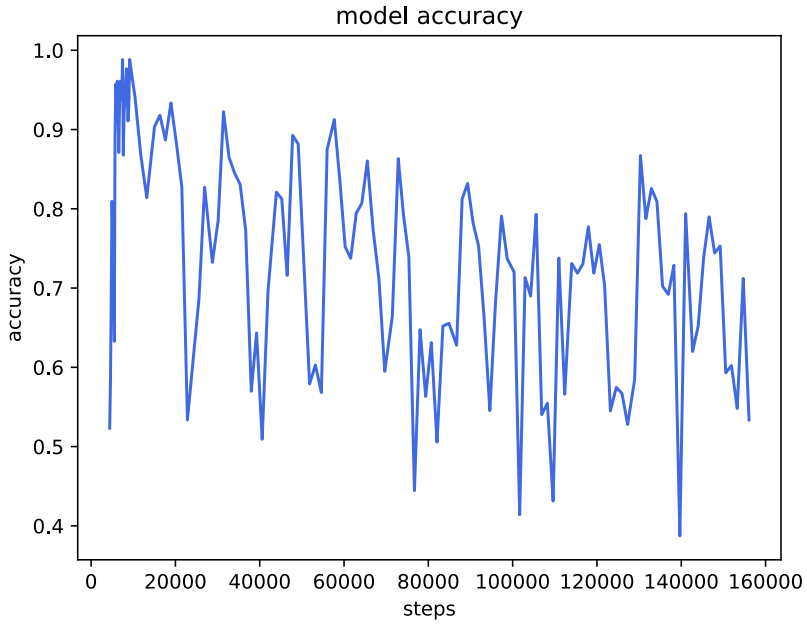


Figura 7.4: Tasa de aciertos del modelo de *Construir* marines

Capítulo 8

Conclusiones

El presente Trabajo Fin de Grado se ha centrado en el ámbito del aprendizaje automático. En primer lugar, se llevó a cabo un estudio en profundidad sobre anteriores usos del aprendizaje automático en *StarCraft II* y otros entornos. Este estudio inicial condujo a tomar la decisión de explorar el área del aprendizaje por refuerzo, en concreto los algoritmos de Q-Learning y Deep Q-Learning. Así, el trabajo se ha enfocado hacia la investigación del funcionamiento de estas técnicas concretas de aprendizaje por refuerzo. Tras explorar la utilidad de las distintas APIs que permiten acceder al entorno de *StarCraft II* para el desarrollo de jugadores automáticos, se acabó eligiendo el entorno de *PySC2*, el cual está desarrollado en el lenguaje Python. Por ello, se estudió cómo funciona esta API y las distintas herramientas que se pueden usar en Python para la implementación de las técnicas de aprendizaje por refuerzo. En segundo lugar, se llevó a cabo un estudio de los distintos niveles en los que se toman decisiones en videojuegos *RTS* como *StarCraft II*, con objeto de entender cuáles deberían de ser las distintas acciones que los agentes tendrían que tomar en los diferentes entornos.

Tras esta fase de estudio inicial, se realizó una primera toma de contacto con el aprendizaje por refuerzo, desarrollando un jugador automático que emplea el algoritmo de Q-Learning en el entorno del minijuego *Moverse a la baliza*. Se eligió este minijuego para esta primera prueba debido a su menor complejidad, lo que facilitó una mejor comprensión de la funcionalidad del aprendizaje por refuerzo.

Después de obtener resultados positivos en este primer experimento, se tradujo el problema de Q-Learning al algoritmo de Deep Q-Learning, permitiendo establecer las bases del aprendizaje por refuerzo profundo para minijuegos de mayor complejidad. Para facilitar este desarrollo, durante esta fase también se profundizó en el estudio del funcionamiento de las redes neuronales y sus configuraciones, ya que Deep Q-Learning emplea las redes neuronales en su algoritmo.

En la realización de estas primeras pruebas se observaron las limitaciones de Q-Learning, necesitando una representación de los estados más sencilla respecto a Deep Q-Learning y, por tanto, siendo difícil de emplear en entornos complejos. Esta limitación se debe a las restricciones que supone representar los estados y sus recompensas en una tabla. Al utilizar una tabla, cada fila o columna debe simbolizar un estado, re-

quiriendo una gran cantidad de memoria si se quiere emplear un entorno complejo. A diferencia de esta técnica, Deep Q-Learning se basa en el uso de redes neuronales para analizar los estados, aportando una mayor flexibilidad a la hora de representar los estados del entorno. Esta ventaja se debe a que las redes neuronales aceptan una ristra de números como valores de entrada, contando con una mayor libertad a la hora de decidir cuántos números representan el estado y de qué forma.

A partir de estos resultados se pasó a realizar nuevos experimentos, probando el funcionamiento del algoritmo Deep Q-Learning sobre distintas configuraciones del entorno del minijuego *Derrotar zealots con blinks*, cuyo objetivo se enfoca a la toma de decisiones reactivas. Se desarrolló un jugador automático para cada una de las tres variantes de este minijuego: uno contra uno, un combate simple para una primera toma de contacto; uno contra dos, siendo necesario tener mayor control sobre los elementos del entorno y los enemigos; y dos contra dos, que plantea un problema multiagente.

En la última variante, la prueba del enfrentamiento de dos contra dos, la complejidad del desarrollo aumentó debido al problema multiagente, ya que fue necesario realizar un programa que soportara el funcionamiento de dos jugadores automáticos independientes que actúan en el mismo entorno y de forma simultánea. Los resultados obtenidos fueron satisfactorios y según lo esperado. Por un lado, cada uno de estos jugadores automáticos aprendió a realizar la técnica de *kiting* sin problemas, demostrando la efectividad de Deep Q-Learning para la toma de decisiones reactivas. Por otro lado y en cuanto al problema multiagente, ambos jugadores automáticos también consiguieron aprender lo esperado. Así, en las situaciones en las que los dos están vivos, no se obstaculizan entre ellos y, cuando solo queda uno con vida, son capaces de actuar igual que el jugador de uno contra dos. Por lo tanto, quedó demostrado que la situación de que dos jugadores aprendan al mismo tiempo en un mismo entorno no tiene por qué suponer un problema, siempre y cuando en la representación del estado se añadan ciertas variables que permitan a cada uno ser consciente del otro.

El último experimento consistió en el desarrollo de un jugador automático en el entorno del minijuego *Construir marines*. Para el desarrollo de este jugador también se empleó el algoritmo de Deep Q-Learning y, en este caso, el reto tiene que ver con la complejidad del entorno, pues su objetivo es construir marines, lo que requiere cumplir con muchos requisitos previos y otros que van apareciendo dinámicamente. En este minijuego aparecen por primera vez las acciones durativas, que pueden apreciarse en muchos otros RTS. Estas acciones no cambian el estado del entorno de forma instantánea, lo que plantea nuevos retos en cuanto al aprendizaje por refuerzo. A diferencia de lo que ocurría con el anterior minijuego, en este experimento se debían tomar decisiones estratégicas o a largo plazo.

Para las acciones durativas se ha comprobado que una buena solución es la creación de estados de transición. Los resultados obtenidos en este experimento no fueron los esperados, ya que el rango de estados posibles resultó demasiado amplio. Para poder alcanzar un mejor resultado, sería necesario un mayor tiempo de entrenamiento y una mejor infraestructura hardware. Por lo tanto, se puede concluir que la utilización del algoritmo de Deep Q-Learning en espacios de estados muy amplios requiere un gran tiempo de aprendizaje para alcanzar un resultado óptimo.

Por último, para la realización de todos estos experimentos se creó una arquitectura reutilizable. Para ello, se estableció un diagrama de UML para organizar la comunicación y las dependencias entre las clases y clases abstractas. Una vez diseñada la organización, se llevó a cabo el desarrollo de las clases: una para el entorno y otra para los agentes. Además se realizó un programa para su uso, que permite configurar desde el propio comando de ejecución aspectos básicos como el número de episodios, el número de agentes, el nombre del fichero de carga o de guardado, sin necesidad de modificar el código. El desarrollo de esta arquitectura se explica en profundidad en el capítulo 4. Esta arquitectura se ha utilizado con cada uno de los agentes y entornos utilizados en este trabajo, mostrando una buena eficacia en todos los casos. Por tanto, se puede concluir que la arquitectura diseñada ha cumplido con su función, permitiendo integrar distintos tipos de agentes y entornos según la necesidad del programador, siempre y cuando estos empleen las interfaces *Environment* y *AbstractAgent* desarrolladas. Como aspecto negativo en relación con la arquitectura desarrollada, debemos mencionar la necesidad de modificar la línea del código del programa principal (*main.py*) que especifica el agente y el entorno que se va a emplear.

8.1. Revisión de los objetivos

Para el presente TFG se habían planteado cuatro objetivos. El desarrollo del trabajo realizado ha permitido abordar y cumplir adecuadamente con los mismos, tal y como se describe a continuación.

1. En relación con el primer objetivo planteado, se ha profundizado en el estudio del uso de técnicas de aprendizaje por refuerzo para el desarrollo de jugadores automáticos en los videojuegos RTS. En concreto, se ha realizado un estudio en profundidad de los algoritmos de aprendizaje por refuerzo más empleados, especialmente de Q-Learning y Deep Q-Learning que son los que se han aplicado. Realizando este estudio se ha apreciado la influencia de los procesos de decisión de Markov en ambos algoritmos, por lo que se ha profundizado en el funcionamiento de esta propiedad. Además, para comprender mejor cómo funciona el aprendizaje por refuerzo profundo (Deep Q-Learning), se han estudiado las redes neuronales que son los modelos computacionales que se emplean en este algoritmo. En el capítulo 2 se recogen los principales conocimientos relacionados con estos aspectos estudiados.

2. De acuerdo con lo planteado en el segundo objetivo, se han investigado los retos que supone la creación de jugadores automáticos en juegos RTS, especialmente se ha estudiado en detalle el entorno de *StarCraft II* y su funcionamiento. Tal y como indica su nombre, un reto de los juegos RTS es el hecho de que las decisiones deben tomarse en tiempo real, teniendo en cuenta el conocimiento parcial que se dispone del mapa debido a la niebla de guerra. Las acciones que se llevan a cabo influyen en el juego a largo plazo, siendo necesario realizar una reflexión previa para saber cómo van a influir las decisiones actuales en momentos posteriores de la partida. Además, los juegos RTS cuentan con un gran número de unidades diferentes, siendo necesaria una coordinación entre ellas y surgiendo el problema

multiagente. Finalmente, las decisiones que se deben tomar en estos juegos tienen tres niveles de abstracción distintos, aumentando la complejidad del aprendizaje de jugadores automáticos. Todas los conocimientos adquiridos sobre los juegos RTS se recogen en el capítulo 3.

3. Para dar respuesta al tercer objetivo, se ha desarrollado un código reutilizable que ha permitido probar distintos algoritmos de aprendizaje en diferentes entornos. El funcionamiento de la arquitectura de este código reutilizable se describe en el capítulo 4. La arquitectura desarrollada ha permitido intercambiar de forma sencilla el entorno en los que se han realizado las distintas pruebas y los algoritmos que se han usado. Asimismo, con este código resulta muy sencillo añadir nuevos entornos y algoritmos, ya que se parte de clases abstractas que tienen la estructura que se debe seguir para añadirlos.

4. Finalmente y de acuerdo con lo planteado en el cuarto objetivo, se ha estudiado el funcionamiento de los algoritmos Q-Learning y Deep Q-Learning para la resolución de distintos minijuegos de dificultad creciente. Así, se comenzó con el problema menos complejo para poner a prueba los conocimientos adquiridos, tanto en Q-Learning como en Deep Q-Learning. Una vez realizada la primera toma de contacto, se usó un entorno más complejo empleando el segundo algoritmo. Este entorno se modificó en varias pruebas para generar retos distintos y evaluar mejor el funcionamiento de Deep Q-Learning, llegando a afrontar con éxito los problemas multiagente. Finalmente, partiendo de una mayor comprensión del funcionamiento de este algoritmo, se realizó una prueba en un entorno en el que juegan un papel importante las acciones durativas. La ejecución de las distintas pruebas realizadas y las conclusiones obtenidas en cada una de ellas se presentan en los capítulos 5, 6 y 7.

8.2. Trabajo futuro

Hay diversas líneas de trabajo en las que se debería profundizar para dar continuidad a los resultados obtenidos en la investigación realizada.

Primero, es necesario seguir investigando para resolver el problema de cómo conseguir una implementación exitosa de los algoritmos utilizados cuando el espacio de estados posibles es muy amplio. Para ello, se podría aumentar la complejidad de la representación del estado del minijuego de Construir Marines. De esta manera, se podría entrenar con más tiempo hasta alcanzar un resultado óptimo. Como se explica en la sección 7.2, para aumentar la complejidad de los estados hay que tener en cuenta ciertas posibilidades como la creación de los distintos elementos en paralelo.

Otra línea de trabajo futura es seguir poniendo a prueba las técnicas de aprendizaje por refuerzo con minijuegos de mayor nivel de complejidad. Como se explica en los capítulos 5 y 6, en los dos primeros minijuegos que se han utilizado en este trabajo, las decisiones que se toman son a nivel reactivo, mientras que en el último minijuego son decisiones a nivel estratégico. Un minijuego que combinara estos dos tipos de tomas de decisiones sería una buena forma de avanzar en esta línea de investigación. La solución

en este caso se podría plantear de distintas maneras: un único agente que se encargue de tomar todas las decisiones o un agente para cada nivel de toma de decisiones. De esta manera, los agentes podrían entrenarse por separado y ver los resultados al combinarlos.

También es necesario avanzar en una línea de investigación que aborde cómo solucionar la toma de decisiones en partidas completas de *StarCraft II*, que presentan una mayor complejidad. Una forma de hacerlo sería crear un agente que sea capaz de tomar decisiones muy abstractas a un nivel estratégico. El estado de la partida habría que simplificarlo y las posibles decisiones también deberían estar restringidas, ya que en caso contrario el espacio de estados y acciones sería demasiado extenso como para poder explorarlo con el tiempo y equipo disponible. Si se logra desarrollar un agente que sea capaz de tomar esas decisiones estratégicas, se podrían desarrollar más agentes para realizar las decisiones elegidas. Estos agentes podrían buscar la mejor manera de ejecutar las decisiones estratégicas y, por lo tanto, se encargarían de tomar decisiones a corto plazo. Los agentes se podrían entrenar por separado y unirlos una vez se logre un resultado óptimo. Además los escenarios específicos para cada agente se podrían recrear en el editor de mapas de *StarCraft II*.

Finalmente, aprovechando la arquitectura creada, sería interesante probar diferentes técnicas de aprendizaje automático y comparar su rendimiento con las utilizadas hasta el momento. Estas nuevas técnicas se podrían aplicar a los minijuegos con los que ya se han obtenido resultados positivos y comparar cuáles son las ventajas de unas técnicas respecto a otras.

Conclusion

This Bachelor's Degree Final Proyect has focused on the field of machine learning. First of all, a study is conducted regarding the uses of machine learning on *Starcraft II* and other similar environments. This initial study led to the decision to explore reinforcement learning field, more specifically, Q-Learning and Deep Q-Learning algorithms. Thus, the work has focused on the investigation the behaviour of these reinforcement learning techniques. After exploring the usefulness of different APIs that allows access to *Starcraft II* environment for automatic players development, ended up choosing *PySC2* component, implemented in Python language. Next, the functionality of this API and the diverse libraries and tools that can be used with Python for the implementation of reinforcement learning techniques was studied . On second place, a study was conducted of the different levels of decision-making in RTS videogames, such as *Starcraft II*, was carried out in order to understand what actions agents must take in differents environments.

After this initial study phase, a first contact with reinforcement learning was made, developing an automatic player that uses the Q-Learning algorithm in the environment of the *Move to beacon* minigame. This minigame was chosen for this first test due to its lower complexity, which allowed a better understanding of the reinforcement learning functionality.

After obtaining positive results in this first experiment, the Q-Learning problem was translated into the Deep Q-Learning algorithm, allowing to establish the foundations of deep reinforcement learning for more complex minigames. To facilitate this development, during this phase the study of the operation of neural networks and their configurations was also deepened, since Deep Q-Learning uses neural networks in its algorithm.

In carrying out these first tests, the limitations of Q-Learning were observed, requiring a simpler representation of the states with respect to Deep Q-Learning and, therefore, being difficult to use in complex environments. This limitation is due to the limitations of representing states and their rewards in a table. When using a table, each row or column must symbolize a state, which requires a large amount of memory if you want to use a complex environment. Unlike this technique, Deep Q-Learning is based on the use of neural networks to analyze the states, providing greater flexibility when representing the states of the environment. This advantage is due to the fact that neural networks accept a string of numbers as input values, having greater freedom when deciding how many numbers represent the state and in what way.

Based on these results, new experiments were carried out, testing the operation of the Deep Q-Learning algorithm in different settings of the *Defeat zealots with blinks* minigame, whose objective is focused on making reactive decisions. An automatic player was developed for each of the three variants of this minigame: one against one, a simple combat for a first contact; one against two, being necessary to have a greater control over the elements of the environment and the enemies; and two against two, posing a multi-agent problem.

In the last variant, the two-against-two match test, the complexity of the development increased due to the multi-agent problem, since it was necessary to carry out a program that supported the operation of two independent automatic players acting in the same environment simultaneously. The results obtained were satisfactory and as expected. On the one hand, each of these automatic players learned to perform the kite technique without problems, demonstrating the effectiveness of Deep Q-Learning for reactive decision making. On the other hand, and regarding the multiagent problem, both automatic players also managed to learn what was expected. Thus, in situations where both are alive, they do not hinder each other and, when only one is left alive, they can act the same as the one-on-two player. Therefore, it was shown that the situation that two players learn at the same time in the same environment does not have to be a problem, as long as certain variables are added in the representation of the state that allow each one to be aware of the situation of the other.

The last experiment consisted of developing an automatic player in the *Build marines* minigame environment. Deep Q-Learning algorithm was also used for the development of this automatic player and, in this case, the challenge has to do with the complexity of the environment, since its objective is to build marines, which requires meeting many prerequisites and others that appear dynamically. Enduring actions first appear in this minigame, which can be seen in many other RTS videogames. These actions do not instantly change the state of the environment, posing new challenges in reinforcement learning. Unlike the previous minigame, this experiment requires making long-term or strategic decisions.

For enduring actions, a good solution has proven to be the creation of transition states. The results obtained in this experiment were not as expected, since the range of possible states was too wide. To achieve a better result, more training time and a better hardware infrastructure would be necessary. Therefore, it can be concluded that the use of the Deep Q-Learning algorithm in very wide state spaces requires a long learning time to achieve an optimal result.

Finally, to carry out all these experiments, a reusable architecture was created. To do this, a UML diagram was established to organize communication and dependencies between classes and abstract classes. Once the organization was designed, the classes were developed: one for the environment and another for the agents. In addition, a program was created for its use, which allows configuring basic aspects such as the number of episodes, the number of agents or the name of the loading and saving file from the execution command itself. The development of this architecture is explained in depth in chapter 4. This architecture has been used with each of the agents and environments used in this work, showing good efficiency in all cases. Therefore, it can be concluded that the designed architecture has fulfilled its function, allowing the integration of

different types of agents and environments according to the needs of the programmer, as long as they use the *Environment* and *AbstractAgent* interfaces developed. As a negative aspect in relation to the architecture developed, we must mention the need to modify some lines of the main program code (*main.py*) that specifies the agent and the environment to be used.

Review of objectives

For the current TFG, four objectives have been set. The development of the work carried out has allowed them to be approached and carried out properly, as described below.

1. Regarding the first objective, the use of reinforcement learning techniques for the development of automatic players in RTS videogames has been studied in depth. Specifically, an in-depth study has been carried out of the most widely used reinforcement learning algorithms, especially Q-Learning and Deep Q-Learning, which are the ones that have been applied. Carrying out this study, the influence of the Markov decision processes in both algorithms has been appreciated, which is why the operation of this property has been deepened. In addition, to approach a better understand on how deep reinforcement learning (Deep Q-Learning) works, neural networks have been studied, which are the computational models used in this algorithm. Chapter 2 contains the main knowledge related to these aspects studied.

2. In accordance with what was stated in the second objective, the challenges posed by the creation of automatic players in RTS games have been investigated, especially the *StarCraft II* environment and its operation. As the name implies, a challenge in RTS games is the fact that decisions must be made in real time, taking into account the partial knowledge that is available of the map due to the fog of war. The actions that are carried out influence the game in the long term, being necessary to carry out a previous reflection to know how the current decisions will influence in later moments of the game. In addition, RTS games have a large number of different units, requiring coordination between them and the multi-agent problem arises. Finally, the decisions to be made in these games have three different levels of abstraction, increasing the complexity of machine learning. All the knowledge acquired about RTS games is collected in chapter 3.

3. To respond to the third objective, a reusable code has been developed that has made it possible to test different learning algorithms in different environments. The functionality of the architecture of this reusable code is described in chapter 4. The architecture developed has made it possible to easily exchange the environment in which the different tests have been carried out and the algorithms used. Also, with this code it is very easy to add new environments and algorithms, since it starts from abstract classes that have the structure that must be followed to add them.

4. Finally, and in accordance with the provisions of the fourth objective, the operation of the Q-Learning and Deep Q-Learning algorithms for solving different minigames of increasing difficulty has been studied. Thus, we start from the less complex problem to test the knowledge acquired, both in Q-Learning and Deep Q-Learning. Once the first contact was made, a more complex environment was used using the second algorithm. This environment was modified in several tests to generate different challenges and better evaluate the operation of Deep Q-Learning, successfully addressing multi-agent problems. Finally, from a better understanding of the operation of this algorithm, a test was carried out in an environment in which durable actions play an important role. The execution of the different tests carried out and the conclusions obtained in each of them are presented in chapters 5, 6 and 7.

Future work

There are various lines of work that should be deepened to give continuity to the results obtained in the research carried out.

First, it is necessary to continue research to solve the problem of how to achieve a successful implementation of the algorithms used when the space of possible states is very wide. To do this, you could increase the complexity of the representation of the state in the *Build marines* minigame. This way, you could train longer until you reach an optimal result. As explained in section 2, to increase the complexity of the states it is necessary to take into account certain possibilities such as the creation of the different elements in parallel.

Another line of future work is to continue testing reinforcement learning techniques with minigames of a higher level of complexity. As explained in chapters 5 and 6, in the first two minigames that have been used in this work, the decisions made are at a reactive level, while in the last minigame they are decisions at a strategic level. A minigame that combines these two types of decision-making would be a good way to advance this line of research. The solution in this case could be presented in different ways: a single agent who is responsible for making all decisions or one agent for each decision-making agent. In this way, agents could train separately afnd see the results when combined.

It is also necessary to advance in a line of research that addresses how to solve decision making in complete *StarCraft II* games, which present greater complexity. One way to do this would be to create an agent that is capable of making very abstract decisions at a strategic level. The state of the game must be simplified and the possible decisions must also be restricted, since otherwise the state and the action space would be too extensive to be explored with the time and the available equipment. If it is possible to develop an agent who is capable of making those strategic decisions, more agents could be developed to carry out the chosen decisions. These agents could find the best way to execute the strategic decisions and, therefore, they would be the ones in charge of making short-term decisions. Agents could train separately and join together once an

optimal result is achieved. Additionally, agent-specific scenarios could be recreated in the *StarCraft II* map editor.

Finally, taking advantage of the architecture created, it would be interesting to test different machine learning techniques and compare their performance with those used so far. These new techniques could be applied to minigames with which positive results have already been obtained and compare the advantages of some techniques over others.

Contribución: Miriam Leis Baltanás

La primera parte del trabajo abarca el periodo de investigación previa sobre el entorno de *Starcraft II*, la API (Interfaz de programación de aplicaciones) que se va a emplear para acceder a él y las técnicas de aprendizaje automático que se han aplicado hasta el momento en este juego.

La idea inicial del proyecto consistió en desarrollar jugadores automáticos para *StarCraft*. Se realizaron investigaciones conjuntas sobre APIs que nos permitiesen acceder a la información del juego, decantándonos por utilizar BWAPI[1] y creando juntos un código que lo emplease. Pero cuando se intentó añadir un programa que nos permitiese ejecutar repetidas veces un agente para entrenarle (*StarcraftAITournamentManager*) empezaron a surgir problemas, volviendo a investigar juntos qué API podríamos utilizar en *StarCraft* y llegando a la conclusión de que era inviable, buscando opciones con *StarCraft II*.

Investigando APIs para *StarCraft II* encontramos *PySC2*. Para asegurarnos de su viabilidad, yo en concreto leí varios artículos de anteriores trabajos empleando este complemento [41, 14]. También busqué varios repositorios de desarrollos de jugadores automáticos llevados a cabo por otros programadores, permitiéndome entender cómo *PySC2* se comunica con *StarCraft II* y cómo se accede a la información del entorno. Además, cuando me encontraba con partes del código complejas, investigaba el código fuente de *PySC2*[2] para entenderlo mejor.

Cuando terminamos nuestras investigaciones, nos reunimos, compartimos los conocimientos adquiridos y pusimos en práctica lo que habíamos aprendido desarrollando un programa básico que emplease *PySC2*, poniéndolo en práctica con los jugadores automáticos que habíamos investigado.

Con un entorno ya funcional en el que trabajar, empiezo a estudiar las bases del aprendizaje automático y el tipo de técnicas que existen, centrándome en el aprendizaje por refuerzo. Además, reúno ideas investigando anteriores usos del aprendizaje automático en *Starcraft II* leyendo artículos de trabajos anteriores [3, 33]. Cuando llegamos a una conclusión, nos reunimos para discutir qué técnica es la más favorable, decidiendo emplear aprendizaje por refuerzo y, más específicamente, Deep Q-Learning. Yo me centro en estudiar más en profundidad la funcionalidad del aprendizaje por refuerzo, leyendo una serie de libros al respecto [38, 6, 9], y, al ver la relación de esta técnica

[1] https://bwapi.github.io/

[2] https://github.com/deepmind/pysc2

con los procesos de decisión de Markov, procedo a investigar sobre esta propiedad [5, 34, 39]. A su vez examino el algoritmo de Q-Learning debido a que su funcionamiento está muy ligado a Deep Q-Learning, examinando el mecanismo de ambos algoritmos y comparando sus similitudes y diferencias.

La segunda parte del trabajo consiste en el desarrollo de las pruebas de los algoritmos de aprendizaje por refuerzo en distintos entornos de *Starcraft II*, estudiando los resultados obtenidos en cada una de ellas.

Nos dividimos la implementación de un agente en un entorno básico que utilice el algoritmo de Q-Learning, tocándome desarrollar la implementación del algoritmo. Con esto desarrollado, decidimos que *Moverse a la baliza* es el minijuego de *PySC2* indicado para empezar las pruebas. A partir de aquí, todos los entornos van a seguir la misma metodología de desarrollo, pensando primero de forma conjunta los parámetros del minijuego (cómo se van a representar los estados, qué acciones se van a poder realizar y qué recompensas deberíamos establecer), repartiéndonos las tareas de implementación, turnándonos entre nosotros el proceso de aprendizaje (consume muchos recursos del ordenador) y analizando y sacando conclusiones juntos de lo aprendido.

Así pues, realizamos los pasos comentados en el minijuego *Moverse a la baliza*, repartiéndonos después la implementación de cada uno de estos parámetros. Por mi parte, implemento la representación de las acciones y de las recompensas.

Al terminar la prueba con Q-Learning, planteamos la implementación de un agente en un entorno básico que emplee el algoritmo de Deep Q-Learning. En este caso, mi tarea es desarrollar el agente. Una vez las bases de Deep Q-Learning ya están implementadas, volvemos a reunirnos para plantear *Moverse a la baliza* con este algoritmo. En concreto yo me encargo de la configuración del algoritmo.

A continuación, llevamos a cabo la siguiente prueba con Deep Q-Learning usando el mapa del entorno de *Derrotar zealots con blinks*. Este minijuego se edita varias veces para realizar distintas pruebas. Yo personalmente me encargo de configurar la segunda prueba: *Derrotar zealots con blinks uno contra dos*. Con cada una de las pruebas en cada entorno se van a seguir los mismos pasos mencionados anteriormente. En la prueba de uno contra uno, mi tarea es generar la representación de los estados y de la configuración del algoritmo; en la segunda, me encargo de la configuración del algoritmo en este entorno y en la última, mis tareas son representar las acciones y estados acordados. En el caso de *Derrotar zealots con blinks dos contra dos*, este minijuego necesita establecer un sistema que permita entrenar a dos agentes simultáneamente. Para ello, nos reunimos y acordamos una solución juntos.

En la parte final de implementación, yo soy la encargada de generar la arquitectura reutilizable de la que se habla en este trabajo. Empiezo estableciendo el diagrama UML, para tener claras las comunicaciones y dependencias entre clases, y continúo desarrollando poco a poco cada módulo: *Environment*, para los entornos; y *AbstractAgent*, para los agentes. Para terminar, implemento el programa que haga uso de ambos módulos y permita los problemas multiagentes.

Por último se redacta la memoria. En ella plasmamos tanto las investigaciones

realizadas del aprendizaje por refuerzo y sus algoritmos como las investigaciones de los juegos RTS y de *Starcraft II*. Además, plasmamos el funcionamiento de la arquitectura y el desarrollo de las distintas pruebas.

Primero de todo, en el capítulo de introducción me encargo de contar el inicio, narrando nuestra motivación en el proyecto, y expongo los objetivos que habíamos establecido en el trabajo (apartado 1.1).

En la parte de investigación me encargué de escribir sobre aquello que yo había investigado en profundidad. En el capítulo 2, escribo aquello que es concerniente al aprendizaje por refuerzo (apartado 2.2), a los procesos de decisión de Markov (apartado 2.3) y al algoritmo de Q-Learning (apartado 2.4). Sobre el capítulo 3, mi tarea es escribir la introducción al propio capítulo y explicar el componente de *PySC2* (apartado 3.4).

Por parte de los capítulos de implementación, escribo el capítulo 4, el cual explica el funcionamiento y las características de la arquitectura desarrollada. Respecto a los capítulos de las pruebas, en el capítulo 5 explico las representaciones decididas para las acciones, los estados y las recompensas en el minijuego *Moverse a la baliza* (apartados 5.1.1, 5.1.2 y 5.1.3). Finalmente, en el capítulo 6 desarrollo los mismos apartados que en el anterior pero relacionados con las distintas configuraciones del minijuego *Derrotar zealots con blinks*.

Contribución: Pablo Joaquín Rodríguez Hidalgo

El trabajo comenzó con la investigación de entornos en los que poder crear jugadores automáticos para *StarCraft*. Ambos llegamos a la conclusión de que el entorno de BWAPI[3] era la mejor opción. Después se continuó implementado un programa para probar como funcionaba BWAPI entre los dos. Se llegaron a ciertos problemas al intentar añadir la herramienta para enfrentar a distintos jugadores automáticos repetidas veces, *StarcraftAITournamentManager* y se decidió que podía ser una mejor opción desarrollar el trabajo en *StarCraft II* con un entorno más reciente y con una mejor documentación.

En este momento se decidió usar la herramienta de *PySC2*. Cada uno investigó por su cuenta el funcionamiento de *PySC2*. Yo en particular estudié el código fuente[4] y la documentación de la herramienta para entender su funcionamiento más a fondo. Ambos llegamos a la conclusión de que *PySC2* parece una buena opción y que podemos continuar con el desarrollo del trabajo. Por lo tanto, se implementa un primer programa que usa *PySC2* y se prueban varios jugadores automáticos ya implementados.

Una vez decidida la herramienta, yo investigué las estrategias habituales que se usan en videojuegos RTS, leyendo artículos como [15] o como [40]. También investigo a que niveles se toman decisiones en RTS con artículos como [33] o [27]. Por último, investigo las distintas peculiaridades de *StarCraft II* respecto al género de los *RTS*.

El siguiente paso, es investigar qué técnicas de aprendizaje automático se suelen usar para resolver este tipo problemas. En esta caso, nos dividimos la investigación. Yo me encargué de investigar cómo funcionan los algoritmos genéticos. Después ponemos en común lo que hemos estudiado y vemos que es más común el uso de aprendizaje por refuerzo.

De nuevo, ambos investigamos anteriores aplicaciones del aprendizaje automático en RTS y específicamente en *StarCraft* y *StarCraft II*. Yo en concreto, me leí varios artículos como [35] o [46]. Una vez ponemos las ideas en común, decidimos que uno de los algoritmos que más nos atrae es Deep Q-Learning. Para ello, primero optamos por familiarizarnos con el algoritmo de Q-Learning que comparte algunas ideas, pero es más sencillo que el anterior.

[3] https://bwapi.github.io/

[4] https://github.com/deepmind/pysc2

Primero estudio a fondo el algoritmo de Q-Learning. Una vez estoy familiarizado con su funcionamiento, implemento un agente funcional en un entorno básico que usa el algoritmo de Q-Learning implementado por mi compañera. Después, estudio a fondo el funcionamiento del algoritmo de Deep Q-Learning. Para ello, primero estudio el funcionamiento de las redes neuronales con la ayuda de este libro [31]. Por último, estudio en profundidad el algoritmo de Deep Q-Learning y sus ventajas respecto a Q-Learning.

Cuando tenemos un entorno funcional y hemos estudiado el funcionamiento de los algoritmos, se pasa a realizar el primer experimento. Este se realiza en el minijuego de Moverse a la baliza. Primero, cada uno diseña por su cuenta cuales deben de ser los parámetros del agente (representación de los estados, acciones a realizar y recompensas a establecer) del agente y después ponemos las ideas en común. Una vez sabemos cuáles serán los parámetros del agente, yo me encargo de implementar la configuración del algoritmo y la representación del estado.

Por otro lado, se realizan estas mismas tareas pero con Deep Q-Learning. En este caso yo me encargo de implementar el algoritmo usando las librerías de *TensorFlow* y *Keras*, y después, se implementa el agente del minijuego de *Moverse a la baliza* usando Deep Q-Learning. Para ello, yo me encargo de implementar la representación del estado respecto al nuevo algoritmo. Por último, se analizan los resultados de ambos aprendizajes y se estudian las ventajas que aporta Deep Q-Learning sobre Q-Learnig.

A continuación, busco un nuevo entorno que se enfoque en el microjuego de *StarCraft II*. El minijuego elegido es *Derrotar a zealots con blinks*. Para los siguiente experimentos se usará este minijuego, editándolo y aumentando su dificultad gradualmente. Por lo tanto, me encargo con el editor de mapas de *StarCraft II* de modificar el minijuego de tal manera que se enfrente una unidad aliada contra una enemiga.

De nuevo, cada uno diseña los parámetros del agente para este experimento por su cuenta y después se pone en común para decidir con cuales parámetros será entrenado. Después se implementa el agente y en este caso, yo me encargo de implementar las acciones y las recompensas de este. Por último se entrena y se analizan los resultados obtenidos.

En cuanto a los minijuegos de uno contra dos y de dos contra dos se divide el trabajo de forma similar a las anteriores. En la variante de uno contra dos, yo me encargo de modificar la representación del estado. En cambio, en la variante de dos contra dos, me encargo de editar el escenario con el editor de mapas de *StarCraft II* e implemento la configuración del algoritmo para los dos agentes que habrá en este experimento. Por último, se obtienen los resultados y se analizan entre los dos.

Por último, yo me encargo de encontrar un último minijuego que se enfoque en el macrojuego de *StarCraft II*. El minijuego es Construir marines y me encargo de todo lo necesario para poder realizar el experimento. Primero, diseño la representación del estado del agente, las acciones que podrá realizar y las recompensas recibidas. Después, implemento todo esto para poder realizar el entrenamiento. Al no conseguir unos buenos resultados por falta de tiempo, diseño una nueva representación del estado muy simplificada respecto a la anterior, de nuevo, entreno al agente, obtengo los resultados

y los analizo.

Una vez se han realizado todos los experimentos, falta la memoria. En la parte de introducción, yo me encargo de escribir el apartado que describe los objetivos 1.1 y también escribo el apartado relacionado con la estructura del documento 1.3.

Respecto a la parte de investigación del proyecto, primero en el capítulo 2 se explica todo lo relacionado con el aprendizaje por refuerzo. Yo escribo todo lo relacionado con la explicación del algoritmo de Deep Q-Learnig 2.6 y también escribo la sección con la introducción a redes neuronales 2.5. En el capítulo 3 hay una intrucción a *StarCraft II*. En este caso yo me encargo de escribir la explicación de todo lo relacionado con los videojuegos RTS *sec:ent:rts*, la sección en la que se explica el videojuego *StarCraft II* 3.2 y los beneficios que aporta este entorno al aprendizaje automático 3.3.

Por otro lado, en el primer experimento que se detalla en el capítulo 5, yo me encargo de escribir la descripción del minijuego 5.1, la configuración del algoritmo de Q-Learning 5.2, la configuración del algotirmo de Deep Q-Learning 5.3 y por último el análisis de los resultados obtenidos 5.4. En el capítulo 6, se explican los experimentos realizados con el minijuego de *Derrotar a zealots con blinks* y yo me encargo de escribir la descripción de los 3 experimentos, las distintas configuraciones aplicadas a los algoritmos y los análisis de todos los resultados obtenidos.

Para terminar, me encargo de escribir el capítulo 7 por completo. En él se describe el experimento que se va a realizar, los parámetros del agente y el análisis de los resultados obtenidos.

Bibliografía

[1] *Age of Empires Franchise*. URL: https://www.ageofempires.com/.

[2] Arthur Allshire, Roberto Martín-Martín, Charles Lin, Shawn Manuel, Silvio Savarese y Animesh Garg. *LASER: Learning a Latent Action Space for Efficient Reinforcement Learning*. Cornell University Library, 2021.

[3] Per-Arne Andersen, Morten Goodwin y Ole-Christoffer Granmo. "Deep RTS: A Game Environment for Deep Reinforcement Learning in Real-Time Strategy Games". En: (2018).

[4] Entertainment Software Association. *Essential Facts about the Computer and Video Game Industry*. 2019. URL: https://www.theesa.com/wp-content/uploads/2021/03/ESA_Essential_facts_2019_final.pdf.

[5] Nicole Bäuerle y Ulrich Rieder. *Markov Decision Processes with Applications to Finance*. Springer, Berlin, Heidelberg, ene. de 2011. ISBN: 978-3-642-18323-2. DOI: 10.1007/978-3-642-18324-9.

[6] Boris Belousov, Hany Abdulsamad, Pascal Klink, Simone Parisi y Jan Peters. *Reinforcement Learning Algorithms: Analysis and Applications*. Ene. de 2010. ISBN: 978-3-030-41187-9. DOI: 10.1007/978-3-030-41188-6.

[7] David M. Bourg y Glenn Seemann. *AI for Game Developers*. O'Reilly Media, Inc, 2004. ISBN: 0-596-00555-5.

[8] M. Buro y T. Furtak. "RTS Games as Test-Bed for Real-Time AI Research". En: *International Joint Conference on Artificial Intelligence 2003*. unpublished, 2003.

[9] Lucian Busoniu, Robert Babuska, Bart De Schutter y Damien Ernst. *Reinforcement Learning and Dynamic Programming Using Function Approximators*. 1.ª ed. CRC Press, Inc., 2010, pág. 270. ISBN: 9781439821091.

[10] Ho Chul Cho, Hyun Soo Park, Chang Yeun Kim y Kyung Joong Kim. "Investigation of the Effect of Fog of War in the Prediction of StarCraft Strategy Using Machine Learning". En: *Comput. Entertain.* 14.1 (dic. de 2016), pág. 16. DOI: 10.1145/2735384.

[11] David Churchill. *AIIDE StarCraft AI Competition*. 2010. URL: http://www.cs.mun.ca/~dchurchill/starcraftaicomp/history.shtml.

[12] "Deep Blue se Impone en la Primera Partida a Kasparov". En: *ABC, MADRID* (feb. de 1996). URL: https://www.abc.es/archivo/periodicos/abc-madrid-19960211-94.html.

[13] Kyriakos Efthymiadis y Daniel Kudenko. "Using Plan-Based Reward Shaping to Learn Strategies in StarCraft: Broodwar". En: *2013 IEEE Conference on Computational Inteligence in Games (CIG)*. 2013, págs. 1-8. DOI: 10.1109/CIG.2013.6633622.

[14] Islam Elnabarawy, Kristijana Arroyo y Donald C. Wunsch II au2. *StarCraft II Build Order Optimization using Deep Reinforcement Learning and Monte-Carlo Tree Search*. Cornell University Library, 2020.

[15] Abdelrahman Elogeel, Andrey Kolobov, Matthew Alden y Ankur Teredesai. "Selecting Robust Strategies in RTS Games via Concurrent Plan Augmentation". En: *International Foundation for Autonomous Agents and Multiagent Systems* 1 (ene. de 2015), págs. 155-162.

[16] Peter Flach. *Machine Learning: The Art and Science of Algorithms That Make Sense of Data*. Cambridge University Press, 2012. ISBN: 1107422221.

[17] Ian Goodfellow, Yoshua Bengio y Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[18] Sepp Hochreiter. "The Vanishing Gradient Problem During Learning Recurrent Neural Nets and Problem Solutions". En: *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* 6 (abr. de 1998), págs. 107-116. DOI: 10.1142/S0218488598000094.

[19] Niels Justesen, Philip Bontrager, Julian Togelius y Sebastian Risi. "Deep Learning for Video Game Playing". En: *IEEE Transactions on Games* 12.1 (2020), págs. 1-20. DOI: 10.1109/TG.2019.2896986.

[20] Joe Kilian y Hava T. Siegelmann. "On the Power of Sigmoid Neural Networks". En: *Proceedings of the Sixth Annual Conference on Computational Learning Theory*. New York, NY, USA: Association for Computing Machinery, 1993, págs. 137-143. ISBN: 0897916115. DOI: 10.1145/168304.168321. URL: https://doi.org/10.1145/168304.168321.

[21] Diederik P. Kingma y Jimmy Ba. "Adam: A Method for Stochastic Optimization". En: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. Ed. por Yoshua Bengio y Yann LeCun. 2015.

[22] Raph Koster. *Theory of Fun for Game Design*. Paraglyph Press, 2005, pág. 244. ISBN: 9781449314996.

[23] Thomas Kurbiel y Shahrzad Khaleghian. *Training of Deep Neural Networks based on Distance Measures using RMSProp*. Cornell University Library, ago. de 2017.

[24] Raul Lara-Cabrera, Carlos Cotta y Antonio Fernández-Leiva. "A review of computational intelligence in RTS games". En: *2013 IEEE Symposium on Foundations of Computational Intelligence (FOCI)*. Abr. de 2013, págs. 114-121. DOI: 10.1109/FOCI.2013.6602463.

[25] Yann LeCun, Y. Bengio y Geoffrey Hinton. "Deep Learning". En: *Nature* 521 (mayo de 2015), págs. 436-44. DOI: 10.1038/nature14539.

[26] Wenjie Li, Zhaoyang Zhang, Xinjiang Wang y Ping Luo. *AdaX: Adaptive Gradient Descent with Exponential Long Term Memory*. Cornell University Library, 2020.

[27] Siming Liu, Sushil J. Louis y Christopher Ballinger. "Evolving effective micro behaviors in RTS game". En: *2014 IEEE Conference on Computational Intelligence and Games*. 2014, págs. 1-8. DOI: 10.1109/CIG.2014.6932904.

[28] Marvin Lee Minsky, Seymour Papert y Seymour Papert. *Perceptrons: An Introduction to Computational Geometry*. Cambridge, MA, USA: MIT Press, 1969, pág. 258.

[29] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra y Martin Riedmiller. *Playing Atari with Deep Reinforcement Learning*. Cornell University Library, dic. de 2013.

[30] Hai Nguyen y Hung La. "Review of Deep Reinforcement Learning for Robot Manipulation". En: *2019 Third IEEE International Conference on Robotic Computing (IRC)*. 2019, págs. 590-595. DOI: 10.1109/IRC.2019.00120.

[31] Michael A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2018. URL: http://neuralnetworksanddeeplearning.com/.

[32] Hao Yi Ong, Kevin Chavez y Augustus Hong. *Distributed Deep Q-Learning*. Cornell University Library, 2015.

[33] Santiago Ontañon, Gabriel Synnaeve, Alberto Uriarte, Florian Richoux, David Churchill y Mike Preuss. "RTS AI Problems and Techniques". En: *Encyclopedia of Computer Graphics and Games*. Ed. por Newton Lee. Cham: Springer International Publishing, 2015, págs. 1-12. ISBN: 978-3-319-08234-9. DOI: 10.1007/978-3-319-08234-9_17-1. URL: https://doi.org/10.1007/978-3-319-08234-9_17-1.

[34] Martijn Otterlo y Marco Wiering. "Reinforcement Learning and Markov Decision Processes". En: *Reinforcement Learning: State of the Art* (ene. de 2012), págs. 3-42. DOI: 10.1007/978-3-642-27645-3_1.

[35] Zhen-Jia Pang, Ruo-Ze Liu, Zhou-Yu Meng, Yi Zhang, Yang Yu y Tong Lu. *On Reinforcement Learning for Full-length Game of StarCraft*. Cornell University Library, sep. de 2018.

[36] Stuart J Russell y Peter Norvig. *Artificial Intelligence : A Modern Approach*. Ed. por Inc. Prentice-Hall. Vol. 281 of Studies in Computational Intelligence. 2014. ISBN: 9781292024202.

[37] Adam Summerville, Sam Snodgrass, Matthew Guzdial, Christoffer Holmgård, Amy K. Hoover, Aaron Isaksen, Andy Nealen y Julian Togelius. "Procedural Content Generation via Machine Learning (PCGML)". En: *IEEE Transactions on Games* 10.3 (2018), págs. 257-270. DOI: 10.1109/TG.2018.2846639.

[38] Richard S. Sutton y Andrew G. Barto. *Reinforcement Learning: An Introduction*. Ed. por MA : The MIT Press Cambridge. 2.ª ed. Vol. 83 of Studies in Computational Intelligence. 2018. ISBN: 9780262039246.

[39] Csaba Szepesvári y Michael L. Littman. *Generalized Markov Decision Processes: Dynamic-Programming and Reinforcement-Learning Algorithms*. Brown University, nov. de 1997.

[40] Budianto Teguh, Oh Hyunwoo, Ding Yi y Long Zi. "An Analysis on the Rush Strategies of the Real-Time Strategy Game StarCraft-II". En: *The 31st Annual Conference of the Japanese Society for Artificial Intelligence, 2017*. unpublished, 2017.

[41] Oriol Vinyals, Timo Ewalds, Sergey Bartunov, Petko Georgiev, Alexander Vezhnevets, Michelle Yeo, Alireza Makhzani, Heinrich Küttler, John Agapiou, Julian Schrittwieser, John Quan, Stephen Gaffney, Stig Petersen, Karen Simonyan, Tom Schaul, Hado Van Hasselt, David Silver, Timothy Lillicrap, Kevin Calderone y Rodney Tsing. *StarCraft II: A New Challenge for Reinforcement Learning*. Cornell University Library, ago. de 2017.

[42] Oriol Vinyals, Stephen Gaffney y Timo Ewalds. *DeepMind and Blizzard Open StarCraft II as an AI Research Environment*. Ago. de 2017. URL: https://deepmind.com/blog/announcements/deepmind-and-blizzard-open-starcraft-ii-ai-research-environment.

[43] Christopher J.C.H Watkins y Peter Dayan. "Technical Note: Q-Learning". En: *Machine Learning* 8 (mayo de 1992), págs. 279-292. DOI: 10.1023/A:1022676722315.

[44] Stefan Wender y Ian Watson. "Applying reinforcement learning to small scale combat in the real-time strategy game StarCraft:Broodwar". En: *2012 IEEE Conference on Computational Intelligence and Games (CIG)*. 2012, págs. 402-408. DOI: 10.1109/CIG.2012.6374183.

[45] Shuo Xiong e Hiroyuki Iida. "Attractiveness of Real Time Strategy Games". En: *2014 2nd International Conference on Systems and Informatics, ICSAI 2014*. Vol. 12. Ene. de 2015, págs. 271-276. DOI: 10.1109/ICSAI.2014.7009298.

[46] Sijia Xu, Hongyu Kuang, Zhi Zhuang, Renjie Hu, Yang Liu y Huyang Sun. *Macro Action Selection with Deep Reinforcement Learning in StarCraft*. Cornell University Library, 2018.