Aprendizaje por refuerzo aplicado a los videojuegos cooperativos

Grado en Ingeniería informática

Especialidad en computación

Trabajo de fin de grado

2018

Autor:
Daniel Alcocer Soto

Director:
Javier Béjar Alonso

Departamento:
Ciencias de la Computación

Facultat d'Informàtica de Barcelona (FIB)

Universitat Politècnica de Catalunya (UPC) - BarcelonaTech

Resumen

Los videojuegos que incorporan personajes inteligentes necesitan algoritmos creados específicamente para ellos, lo que implica tener que dedicarles muchas horas de trabajo a cada uno. Además, algunos de estos algoritmos utilizan técnicas de aprendizaje automático las cuales necesitan mucho tiempo para adaptarse a cada juego.

Una manera de agilizar el aprendizaje de estos sería que pudiesen aprender de un humano, el cual ya supiese jugar. Así pues, este proyecto se ha encargado de crear una algoritmo capaz de aprender de las acciones del jugador mientras juegan a un videojuego cooperativo, por lo que el algoritmo también ha podido aprender a realizar estrategias cooperativas simples con el jugador.

Para crearlo se ha combinado un algoritmo de Aprendizaje por Refuerzo llamado Q-learning con una red neuronal. Además, como videojuego se ha utilizado una simplificación de los combates dobles de la saga Pokémon creada des de cero.

El resultado ha sido bastante satisfactorio dado que el algoritmo a conseguido aprender a jugar bastante bien al juego y conoce las mecánicas más básicas de este. Además, se han podido observar tanto comportamientos similares en el modelo que aprende del jugador como comportamientos cooperativos simples en el modelo cooperativo.

Resum

Els videojocs que incorporen personatges intel·ligents necessiten algoritmes creats específicament per a ells, el que implica haver de dedicar moltes hores de treball a cadascun. A més, alguns d'aquests algoritmes utilitzen tècniques d'aprenentatge automàtic les quals necessiten molt de temps per adaptar-se a cada joc.

Una manera d'agilitar l'aprenentatge d'aquests seria que poguessin aprendre d'un humà, el qual ja sabés jugar. Així doncs, aquest projecte s'ha encarregat de crear una algoritme capaç d'aprendre de les accions del jugador mentre juguen a un videojoc cooperatiu, de manera que l'algoritme també ha pogut aprendre a realitzar estratègies cooperatives simples amb el jugador.

Per crear-lo s'ha combinat un algoritme d'Aprenentatge per Reforç anomenat Q-learning amb una xarxa neuronal. A més, com a videojoc s'ha utilitzat una simplificació dels combats dobles de la saga Pokémon creada des de zero.

El resultat ha estat força satisfactori degut a que l'algoritme a aconseguit aprendre a jugar bastant bé al joc i coneix les mecàniques més bàsiques d'aquest. A més, s'han pogut observar tant comportaments similars en el model que aprenen del jugador com comportaments cooperatius simples en el model cooperatiu.

Abstract

Videogames that incorporate intelligent characters need algorithms created specifically for them, which means having to dedicate many hours of work to each one. In addition, some of these algorithms use machine learning techniques which need a lot of time to adapt to each game.

Learning from a human that already knows how to play could be one way to help the learning. So, this project has created an algorithm capable of learning from the player's actions while playing a cooperative videogame, so the algorithm has also been able to learn to perform simple cooperative strategies with the player.

To create it, an Learning Reinforcement algorithm called Q-learning has been combined with a Neural Network. Also, a simplification of the double combats of the Pokémon saga has been developed from scratch and used as a videogame.

The result has been quite satisfactory given that the algorithm has learned to play the game well enough and knows the most basic mechanics of this. In addition, similar behaviors have been observed in models that learn from the player and also simple cooperative behaviors have been observed in the cooperative model.

Agradecimientos

Después de 4 meses de duro trabajo realizando este proyecto, me gustaría agradecer a todas aquellas personas que me han ayudado y apoyado durante este proceso.

Para empezar, me gustaría agradecerles tanto a Antonio Chica como a Javier Béjar, ambos profesores del departamento de computación, por su ayuda en el planteamiento inicial del proyecto. En especial a Javier, el director del proyecto, por los numerosos buenos consejos y aportaciones dadas durante el transcurso del proyecto sobre el desarrollo tanto de la red neuronal como del algoritmo de aprendizaje por refuerzo.

Además, quisiera agradéceles a mis amigos y compañeros universitarios Victor y Joel por sus consejos y apoyo moral durante la realización del proyecto. También me gustaría dar a mis padres mi sincero agradecimiento por su comprensión y apoyo diario durante toda mi etapa académica.

A todos, muchas gracias.

Índice general

Índice de Tablas

Índice de Figuras

Capítulo 1

Contextualización y alcance del proyecto

Este capítulo pretende definir y concretar tanto el tema como el objetivo de este proyecto. Para tal fin, se hará una introducción al tema principal (sección 1.1), donde podremos encontrarnos algunas definiciones básicas (sub-sección 1.1.1), una pequeña explicación del por qué se usan los videojuegos para investigar la inteligencia artificial a modo de contexto (sub-sección 1.1.2) y qué papel juega la cooperatividad en este ámbito (sub-sección 1.1.3). También se incluye las posibles partes interesadas en este proyecto (sub-sección 1.1.4).

También se hablará del estado del arte actual (sección 1.2), donde se mencionarán los principales métodos utilizados para el aprendizaje por refuerzo (sub-sección 1.2.1), y algunos de los algoritmos que pueden tener más importancia en el contexto de este proyecto (sub-sección 1.2.2). Además se realizará una referencia a un logro bastante importante relacionado con el tema del aprendizaje por refuerzo en el campo de los videojuegos (sub-sección 1.2.3).

Además, se hablará de la formulación del problema (sección 1.3). En esta sección se hablará del problema que se intenta solventar (sub-sección 1.3.1) y del objetivo del proyecto (sub-sección 1.3.2).

También se explicará el alcance que tendrá este proyecto (sección 1.4). Esta sección incluye la extensión prevista del proyecto (sub-sección 1.4.1) y los posibles obstáculos que se podrían encontrar (sub-sección 1.4.2).

Para finalizar, el capítulo 1.5 hablará de la metodología que se usará (sub-sección 1.5.1), el método de evaluación (sub-sección 1.5.2) y las herramientas principales que se utilizarán (sub-sección 1.5.3).

1.1. Contextualización

1.1.1. Definiciones básicas de términos y conceptos propios del tema

Antes de empezar, esta sección se encargará de definir los siguientes términos y conceptos debido a que serán utilizados a lo largo del proyecto.

Videojuego cooperativo Un juego cooperativo es una modalidad concreta dentro de los juegos multijugador. Son cooperativos esas modalidades de juegos en los que dos o más jugadores trabajar juntos en equipo para conseguir un objetivo común dentro de una partida, ya sea para resolver puzles o problemas o para acabar con un enemigo común controlado por el propio juego.

Un claro ejemplo puede ser Portal 2[1], en el que dos jugadores tiene que ayudarse mutuamente para superar una serie de habitaciones llenas de rompecabezas. También lo es Left 4 Dead[2], donde un grupo de jugadores tendrán que ponerse a salvo de una horda de enemigos cubriéndose las espaldas mutuamente.

Personaje no jugador, (o NPC, por sus siglas en inglés) Son todos aquellos personajes que no están controlados por el jugador, es decir, que los controla el propio juego con algún tipo maquina de estados o algoritmo de Inteligencia Artificial.

Agente El agente es el ente representativo de la Inteligencia Artificial (o IA). El agente es quien aprende, quien recibe las recompensas y quien toma las decisiones correspondiente al resultado del algoritmo. Este suele trabajar con un **modelo** que le indica como debe actuar en cada situación.

Estado (o state en inglés) Los estados del entorno son la información que el agente tiene sobre el entorno. Estos incluyen toda la información que se considera relevante del entorno y que debe ser conocida por el agente.

Acción (o action en inglés) Las acciones representan las diferentes formas en que el agente puede actuar en el entorno. Estas son capaces interactuar y modificar el entorno, lo cual se ve reflejado en pequeños cambios en el estado.

Aprendizaje por Refuerzo (Reinforcement Learning o RL) El RL es un enfoque del aprendizaje automático inspirado en la manera en la que aprenden los humanos y los animales a tomar decisiones, a través de recompensas (positivas o negativas) recibidas de un entorno.

[1] Videojuego de Valve Corporation, 2011. `http://www.thinkwithportals.com`
[2] Videojuego de Valve Corporation, 2008. `http://www.l4d.com`

En un punto en el tiempo, el **agente** está en un **estado** (s) concreto y decide llevar acabo una **acción** (a) de entre las posibles del estado. Como respuesta, el **entorno** le proporciona una **recompensa** (r) y el **nuevo estado** resultado de haberse llevado a cabo la acción a en el estado s. El siguiente esquema resume el proceso anterior [1].

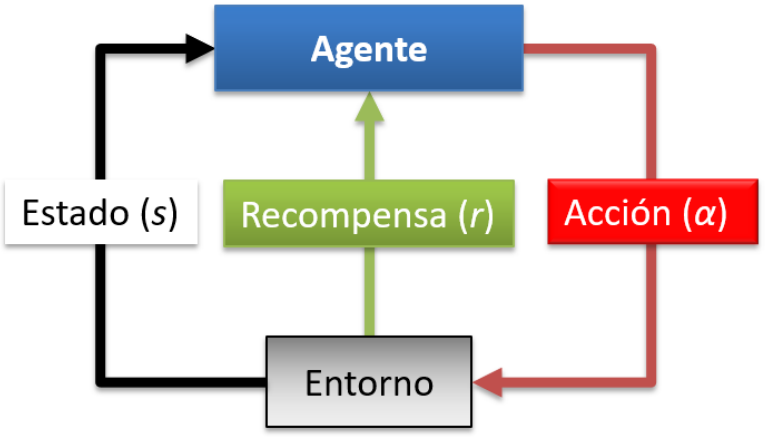

FIGURA 1: Esquema básico del funcionamiento de RL

1.1.2. Por qué usar videojuegos en la investigación de la IA

Los videojuegos ofrecen un entorno ideal para el estudio de la Inteligencia Artificial (IA o AI por sus siglas en inglés). Esto se debe a tres principales factores: su complejidad, su variedad y su popularidad. Además, debido a que cada vez son más complejos, más variados y más populares, hace que se necesiten constantemente nuevas soluciones en IA para enfrentarse a los nuevos retos.

Los videojuegos son atractivos para ser resueltos con la ayuda de la IA porque suponen un problema difícil y complejo, debido al gran tamaño del espacio finito de estados y las numerosas estrategias posibles a realizar. Además, la gran variedad que hay, hace posible que haya retos para todas las áreas de la Inteligencia Artificial como en Machine Learning (Go[3]) o en Tree Search (Ajedrez, Damas) o en Navegación y Pathfinding (StarCraft[4]), entre otros [1].

Además, gracias a la popularidad de los videojuegos, estos pueden ofrecer grandes cantidades de datos sobre las partidas jugadas. Algunos de estos datos están disponibles de forma abierta y se pueden utilizar como fuente de información para el entrenamiento de un agente.

[3] El campeón mundial de Go fue derrotado por AlphaGo. `https://deepmind.com/research/alphago/`

[4] Videojuego de Blizzard Entertainment, 1998. `https://starcraft.com`

A modo de referencia, en Marzo de 2017 se subió un archivo de más de mil millones de partidos del juego Dota 2[5] que se jugaron entre Marzo de 2011 y Marzo de 2016 a el proyecto OpenDota[6] [2].

Un beneficio adicional de los juegos como entorno para el estudio del comportamiento de un agente es que ofrecen un espacio realista pero mucho más conveniente y más barato en comparación, por ejemplo, con la robótica. Esto se debe a que los videojuegos pueden simular las condiciones necesarias y no necesitan de elementos reales que se pueden llegar a estropear.

Gracias la investigación de la IA en los juegos, se han podido idear nuevos algoritmos que logran obtener muy buenos resultados para determinados propósitos, un de ellos es el aprendizaje por refuerzo (o RL por sus siglas en inglés). Fue Arthur Samuel, en 1959, quien ideó un algoritmo para jugar a las Damas. Este aprendió a jugar a las Damas jugando contra sí mismo, auto-aprendiendo a como jugar. A pesar de los recursos computacionales muy limitados de entonces, el algoritmo aprendió a jugar lo suficientemente bien como para vencer a su propio creador [3].

1.1.3. ¿Qué hay de la cooperatividad?

A pesar de todo lo dicho anteriormente, hay un tipo de juego que no se suele utilizar para el estudio de la IA en general, los videojuegos cooperativos. El campo de la cooperatividad o colaboración entre agentes no ha sido el más investigado, pero en la última década a empezado a coger fuerza. En concreto, el aprendizaje por refuerzo de múltiples agentes (o MARL por sus siglas en inglés) tiene un gran potencial en aplicaciones como por ejemplo control de las luces de tráfico, o la conducción autónoma [4] [5].

En el mundo de los videojuegos, las aplicaciones de la colaboración entre agentes también son notables. Se puede lograr aumentar la dificultad de los juegos si, por ejemplo, los NPCs enemigos colaboraran entre ellos de forma inteligente para derrotar al jugador. Y a la vez también para hacerle la vida más fácil al usuario si los NPCs aliados colaboran entre ellos. Y es aquí donde se formula la siguiente pregunta: ¿Y si los NPCs aliados no solo colaboran entre ellos, sino que también colaboran con el propio jugador?

Los videojuegos actuales que ya integran cooperatividad entre los NPCs y el jugador suelen usar IAs especificas, las cuales se construyeron teniendo en cuenta el funcionamiento del juego. Un ejemplo de una IA que tiene como función principal ser la compañera del jugador y ayudarlo es Ellie en The Last of Us[7].

[5] Videojuego de Valve Corporation, 2013. `http://www.dota2.com`

[6] Plataforma de datos de Dota 2 mantenida por la comunidad. `https://www.opendota.com`

[7] Videojuego de Sony Computer Entertainment, 2013. `http://www.thelastofus.playstation.com`

Esta está muy bien conseguida (aunque no es perfecta), pero usa otras técnicas que requieren conocimientos previos del juego, es decir, no aprende desde cero. Algunos de estos conocimientos son el uso de algoritmos de pathfinding o la obligación de permanecer cerca del jugador [6].

1.1.4. Partes interesadas

A continuación veremos las partes que podrían estar interesadas en este proyecto teniendo en cuenta que trata sobre la inteligencia artificial aplicada en los videojuegos.

Beneficiados

Siempre y cuando este proyecto obtenga unos buenos resultados, estos podrían tener gran interés para la comunidad académica por los posibles avances que se puedan producir. Ya sea visto desde el punto de vista de un método de RL que aprende más rápido gracias a la observación de la ejecución correcta de la tarea que realizará el agente, pero hecha por parte de un humano, o desde el punto de vista de la colaboración entre hombre y maquina.

También podría ser beneficiosa para la industria de los videojuegos, ya que podían usar el resultado en sus propios videojuegos con el fin de ahorrarse el tiempo y coste de programar un agente específico para sus juegos (a cambio de dedicar tiempo al entrenamiento del agente), o para mejorar su producto final y así obtener como resultado videojuegos de mayor calidad.

Aunque de entre los dos grupos anteriores quien se beneficiaría más es sin dudas la industria de los videojuegos. Esto se debe principalmente a que si son capaces de crear mejores videojuegos, obtendrían un aumento en las ventas de sus juegos y por tanto obtendrían más beneficio económico de ellos.

Usuario principal

Aun hay una tercera parte involucrada, los jugadores. Ellos serán quienes usen más los resultados del proyecto (si obtiene buenos resultados) ya que son ellos los que utilizarán al final el videojuego que incorpore los resultados del proyecto.

Ellos también obtendrán algo de beneficio, ya que si el juego contiene mejores NPCs, estos podrán ser más útiles durante el transcurso de su partida. Por tanto, la experiencia de juego será mejor y los jugadores disfrutarán más de ésta.

1.2. Estado del Arte

A continuación veremos un poco de la base teórica que existe actualmente sobre el aprendizaje por refuerzo, así como un hito conseguido gracias a esta técnica.

1.2.1. Principales métodos

Antes de poder hablar del estado del arte en el campo del aprendizaje por refuerzo, hay que tener en mente alguna definiciones o clasificaciones de los métodos de RL que se suelen usar para describirlos [1]:

Política (o policy en inglés) Una política es la estrategia que el agente sigue para seleccionar acciones, dado el estado en el que se encuentra.

Off-policy Este tipo de políticas aproxima la política óptima independientemente de las acciones del agente.

On-policy Este tipo de políticas aproxima la política óptima como un proceso vinculado a las acciones del agente, incluidos los pasos de exploración.

Episódico (episodic en inglés) En este tipo de algoritmos de RL, el entrenamiento ocurre fuera del juego en un conjunto finito de instancias de entrenamiento. A la secuencia finita de estados, acciones y recompensas recibidas en cada instancia de entrenamiento se les llama episodio (o episode).

Incremental A diferencia de los episódicos, en estos algoritmos de RL el aprendizaje ocurre durante el juego.

Bootstrapping es un término dentro de RL que clasifica los algoritmos según la forma en que optimizan los valores del estado. Los algoritmos que usan bootstrapping estima qué tan bueno es un estado en función de qué tan bueno creemos que será el próximo estado.

Backup A diferencia del bootstrapping, los algoritmos que usan backup retroceden desde un estado en el futuro hacia el estado actual que queremos evaluar y consideran los valores de los estados intermedios en su estimación del valor del estado. El backup tiene dos propiedades principales: la profundidad (o depth) y la amplitud (o breadth).

Actualmente los principales algoritmos de aprendizaje por refuerzo se basan en tres métodos básicos que se explicará por encima a continuación [1]:

Programación dinámica (o Dynamic programming) Este método requiere conocimiento del modelo del mundo. El modelo del mundo debe de estar formado por una función de probabilidad, *P(s,s',a)*, que indica la probabilidad que hay de ir al estado *s'* desde el estado *s* al realizar la acción *a*, y por una función de recompensa, *R(s,s',a)*, que indica la recompensa obtenida al pasar del estado *s* al estado *s'* realizando la acción *a*. Este método obtiene la política óptima mediante bootstrapping.

Métodos Monte Carlo Los métodos Monte Carlo se basan en el muestreo aleatorio repetido y, a diferencia del método anterior, no requiere conocimiento del entorno ni utilizan bootstrapping. Los algoritmos de este tipo son ideales para el entrenamiento fuera del juego (episódico) y aprenden usando backup.

Temporal Difference learnig (o TD-learning) Al igual que con los métodos Monte Carlo, el conocimiento del modelo del mundo no es necesario y, por lo tanto, se estima. Los algoritmos de este tipo aprenden de la experiencia mediante bootstrapping y variantes de backup. A diferencia de los métodos Monte Carlo, estos suelen ser incrementales.

1.2.2. Principales algoritmos

El algoritmo más utilizado en la investigación de IA en los juegos es **Q-learning** [7]. Q-learning es un algoritmo que utiliza el método TD-learning, off-policy y que aprenden de la experiencia utilizando bootstrapping. Este algoritmo se basa en los valores *Q(s,a)*, los cuales indican como de bueno es escoger la acción *a* estando en el estado *s*. Estos valores se pueden ver como una estimación de la recompensa. De forma simplificada, Q-learning es una simple actualización de los valores Q de forma iterativa [1]. Para la realización de este proyecto se usará una versión de este algoritmo.

Para los juegos de lucha (categoría parecida a la del juego que usaremos en este proyecto) es popular usar el aprendizaje por refuerzo, en particular, el algoritmo SARSA para el aprendizaje on-policy de los valores Q que están representados por aproximaciones lineales y de funciones ANN (Artificial Neural Network) [1]. Un ejemplo puede ser el trabajo que hizo el equipo de Microsoft Research en Tao Feng: Fist of the Lotus[8] [8].

A pesar de esto, este algoritmo no acaba de encajar con el tipo de videojuego que se utilizará para el proyecto, así que no se tendrá en cuenta. Esto se debe a que el juego utilizado será de batallas por turnos en la que los jugadores no se pueden moverse del lugar, a diferencia del Tao Feng.

[8] Videojuego de Microsoft Game Studios, 2003.

Por la parte de los algoritmos pensados para entornos multi-agente, en el 2017 el equipo de DeepMind desarrolló un nuevo algoritmo llamado MADDPG, que permite a los agentes aprender a colaborar y competir entre ellos, y parece que ha obtenido buenos resultados [9] [10]. Pero a pesar de sus buenos rendimientos, la aplicación de este algoritmo en el proyecto sería demasiado complicada para el objetivo principal del proyecto, por lo cual tampoco se utilizará.

1.2.3. Logro importante del RL en los videojuegos

Un logro importante para las IAs que juegan a videojuegos fue en 2014 cuando los algoritmos desarrollados por Google DeepMind[9] aprendieron a jugar varios juegos de la clásica consola de videojuegos Atari 2600, sin ningún conocimiento previo del juego. El resultado fue que logró alcanzar un nivel de habilidad sobrehumano en 29 de los 46 juegos [11].

Las entrada solo eran los píxeles brutos de los elementos visuales del juego, junto con la puntuación, y la salida eran las direcciones del controlador y el botón de disparo [12]. El método utilizado fue una combinación de RL, en concreto Q-learning (que hemos visto en la sub-sección 1.2.2), y redes neuronales profundas, obteniendo como resultado Deep Q-Networks (DQN[10]).

Para este proyecto se intentará imitar la combinación usada de Q-learning y redes neuronales, solo que la entrada no serán los píxeles de pantalla sino valores extraídos del propio juego.

1.3. Formulación del problema

1.3.1. El problema

El proceso de creación de agentes para videojuegos suele suponer la creación de un algoritmo especifico para el videojuego. Esto implica tener que dedicarle horas y esfuerzo a implementar una inteligencia artificial concreta para cada videojuego. Pero tal y como se ha visto en la sección 1.2.3 el aprendizaje por refuerzo permitiría poder crear un único algoritmo que funcionase para distintos juegos.

Además, uno de los problemas de la inteligencia artificial es el tiempo de aprendizaje. Esto afecta sobretodo a algoritmos como los del aprendizaje por refuerzo, donde el algoritmo aprende a base de prueba y error. Una manera de agilizar el aprendizaje del algente sería que este pudiese aprender de un humano, el cual ya supiese jugar.

[9] `https://deepmind.com/`
[10] `https://deepmind.com/research/dqn/`

1.3.2. Objetivo

El objetivo principal de este proyecto será realizar una prueba de concepto en la que se apliquen técnicas de aprendizaje por refuerzo a un NPC aliado con el fin de que aprenda a jugar bien (al menos mejor que el mero azar) a un juego desde cero, pero observando al jugador (que teóricamente ya sabe jugar) para que su aprendizaje sea más rápido y de mejor calidad.

Para alcanzar este objetivo se usará como entorno un videojuego cooperativo. Esto se debe a que en estos tipos de juegos, dos o más jugadores colaboran para poder ganar el juego. Así pues, el jugador y el NPC tendrán el mismo objetivo y podrán realizar las mismas acciones, hecho que facilitará el aprendizaje al algente.

Además habrá un objetivo secundario. Dado que el agente aprenderá a jugar un videojuego cooperativo observando las acciones de otro jugador, se intentará guiar a la inteligencia artificial para que aprenda a desarrollar estrategias cooperativas sencillas en las cuales se tenga en cuenta al propio jugador.

Cabe destacar que para que se pueda llegar a intentar cumplir el segundo objetivo primeramente se tiene que haber conseguido que el primer objetivo se logre satisfactoriamente. Esto se debe a que para lograr un estrategia cooperativa con el jugador, primero debe de ser capaz de poder observarlo y aprender de su comportamiento.

1.4. Alcance del proyecto

1.4.1. Extensión del proyecto

Respecto al alcance del proyecto, se espera que la inteligencia artificial sea capaz de aprender las mecánicas básicas del juego y realizar la acción más apropiada a cada situación a partir de la observación de las acciones de un jugador real. También se espera que aprenda a cooperar con el jugador desarrollando estrategias sencilla para ganar, como por ejemplo atacar al oponente que podría causarle más daño al jugador, atacar al mismo enemigo que el jugador, atacar al enemigo que ha atacado al jugador o atacar al otro enemigo porque prevé que este morirá por la acción del jugador.

En ningún momento se espera crear una inteligencia artificial que sepa jugar al videojuego de una forma perfecta y ganado siempre, pero si que el resultado sea significativamente mejor que la toma de decisiones aleatorias. Tampoco se espera que aprenda estrategias cooperativas complejas dado que no es el objetivo principal del proyecto.

Para realizar el proyecto, se creará desde cero un versión muy simplificada de los combates dobles de la saga Pokémon[11] a modo de entorno donde entrenar al agente. Así pues, el videojuego resultante será un juego de lucha por parejas por turnos con unas dinámicas de juego muy sencillas.

El motivo principal por el cual se creará el juego desde cero es porque de esta manera se podrá tener un control absoluto para facilitar la implementación de la inteligencia artificial. El videojuego también dispondrá de una interfaz gráfica para facilitar la comprensión del estado del juego y facilitar la interacción con el jugador.

En esta versión del juego únicamente se podrá atacar y cada jugador solo dispondrá de un pokémon por batalla. Así que se podía ver como una batalla entre cuatro pokémon de dos contra dos. También se simplificarán los ataques, eliminado efectos secundarios (quemar, paralizar, etc) o acciones que producen esperas de turno, cambios de meteorología, etc. Todo esto con el fin de simplificar el motor del juego y las posibles estrategias que aprendería el agente.

A pesar de esto se mantendrán ciertos aspectos para mantener el juego impredecible, como la posibilidad de hacer un critico (daño extra) con cierta probabilidad, la variabilidad del daño del ataque, las prioridades de los movimientos, la diferenciación de las características de cada tipo de pokémon y la variabilidad de esta entre pokémon de la misma clase.

Teniendo el entorno definido, podemos formular el objetivo principal como que el agente aprenda las mecánicas básicas de pokémon. Estas incluyen las bonificaciones en el daño causado al enemigo dependiendo de la efectividad del tipo, tanto de los pokémon como del movimiento. A modo de referencia, la efectividad de los tipos se puede ver resumida en la tabla que muestra la figura 2, y básicamente es una relación entre los tipos.

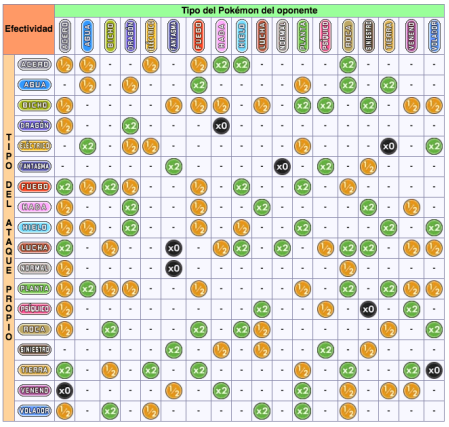

FIGURA 2: Efectividad de los tipos en el juego Pokémon

[11] Videojuego de Game Freak, 1996. `https://www.pokemon.com`

Para lograr estos objetivos se intentará hacer alguna variación de Q-learning (visto en la sub-sección 1.2.2) o similar, adaptándolo para incluir al jugador en el algoritmo.

1.4.2. Obstáculos y riesgos del proyecto

1. **Grado de aprendizaje.** Un de los problemas de RL es el equilibrio adecuado entre el uso, o explotación, del conocimiento aprendido frente a la exploración de nuevos estados no vistos en el espacio de búsqueda. Tanto la selección aleatoria de acciones (sin explotación) como la selección constante de la mejor acción de acuerdo con el modelo (sin exploración) son estrategias que generalmente arrojan malos resultados.

   La solución más usada es e-greedy. De acuerdo con e-greedy, el agente de RL elige con probabilidad $1 - \varepsilon$ (donde $\varepsilon \in [0, 1]$) la acción que cree que devolverá la recompensa futura más alta (explotación), de lo contrario, elige una acción aleatoria (exploración). Sin embargo, encontrar la ε más adecuada puede llegar a suponer un problema y una perdida de tiempo en los casos de las ε que den malos resultados.

2. **Parámetros del modelo.** Otro problema es el ajuste de parámetros del modelo de RL que se aplique. Dependiendo del modelo los parámetros pueden ser vitales para conseguir un rendimiento aceptable, y normalmente no son fáciles de establecer. Esto debería de poder solucionarse en la fase de experimentación.

3. **Potencia computacional.** Al tratarse de un problema computacional complejo, la potencia computacional de la maquina donde se ejecute el entrenamiento será de gran importancia para obtener buenos resultados en un tiempo aceptable. Si la maquina donde se entrena el agente no dispone de suficiente potencia computacional es posible que el agente resultante dé peores resultados de los que podría llegar a dar debido la limitación temporal del proyecto.

4. **Función de recompensa.** Como se ha explicado en la sección 1.1.1, los algoritmos de RL necesitan para funcionar una recompensa. Una de las mayores dificultades que habrá en el proyecto será definir la función de recompensa para que el agente aprenda correctamente.

5. **Errores de programación.** También hay que tener en cuenta los posibles errores o *bugs* que se produzcan durante la fase de desarrollo del juego y del agente. Dependiendo de la importancia del error, pueden causar una perdida de tiempo considerable.

1.5. Metodología y rigor

1.5.1. Metodología

Dado que este proyecto incluye la creación de un juego para ser el entorno donde entrenar al agente, el desarrollo de este tendrá que posponerse hasta que el prototipo del juego permita incluirlo. Por este motivo, ambas partes (crear el juego y crear el agente) seguirán la metodología *agile*. Esta permitirá generar pequeños prototipos funcionales para poder empezar a realizar pruebas de funcionamiento los más pronto posible. Así se podrá adelantar el momento en el cual se podrá iniciar la creación de agente, el cual es la parte fundamental del proyecto.

También se realizaran una serie de reuniones con el director del proyecto para poder informar sobre el estado del proyecto y poder corregir cualquier problema o desviación del resultado esperado.

1.5.2. Método de evaluación

El método de evaluación para el primero objetivo será con el propio juego. Dado que se pretende hacer un agente que sea mejor que las decisiones aleatorias, solo será necesario comparar los resultados de este con los NPCs aleatorios para ver si los ha superado, si los puede ganar, o simplemente observando si su comportamiento deja de ser aleatorio y comienza ha ser predecible.

Para el segundo objetivo, la cooperatividad con el usuario, el método de evaluación será la propia opinión de jugador. Esto se debe a la dificultad que hay en detectar la cooperatividad y a la objetividad de esta. Si el jugador considera que el agente le está ayudando y ha observado comportamientos cooperativos en el agente, se podrá considerar que se ha cumplido el segundo objetivo.

1.5.3. Herramientas

La principal herramienta utilizada será GitHub, que permite gestionar código de una forma sencilla [13]. Además, permite subir actualizaciones del código con comentarios y revisar las actualizaciones anteriores, lo cual encaja muy bien con la metodología *agile*.

El lenguaje utilizado para programar tanto el juego como el agente será Python. Sobretodo se usarán las librerías de PyGame para el desarrollo del juego y Keras para el desarrollo del agente [14] [15]. También se utilizará la API online pokeAPI para la obtención de la información relevante del juego [16].

Capítulo 2

Planificación temporal

Este capítulo pretende definir la extensión del proyecto en el límite dado. Para tal fin se dividirá el proyecto en diferentes fases y tareas (sección 2.1), se comentarán los recursos de los cuales se dispondrá durante la realización del proyecto (sección 2.2), y se hablará de las posibles alternativas en caso de que aparezcan obstáculos durante el desarrollo del proyecto (sección 2.3).

Finalmente se verá la estimación del tiempo y la gráfica de Gantt correspondiente a la planificación final (sección 2.4) y los cambios sufridos respecto la planificación inicial (sección 2.5).

2.1. Descripción de las tareas

El proyecto se dividirá en diversa fases en las que al final de cada una se logrará un objetivo o se añadirá una nueva funcionalidad al proyecto. Hay dos tipos de fases, las fases de documentación (2.1.1 y 2.1.7), y las fases de implementación del proyecto, que en orden de realización lógico son: la creación el motor del juego (2.1.2), la creación de la interfaz gráfica (2.1.3), la creación del agente RL (2.1.4), su posterior extensión (2.1.5) y su evaluación (2.1.6). En las siguientes sub-secciones se detallará cada una de las fases en las que se ha dividido el proyecto.

2.1.1. Documentación del Hito Inicial

Esta fase es la que esta relacionada con la asignatura GEP. Debido a este hecho, esta fase es totalmente paralelizable con las demás fases debido a que no hay ninguna dependencia con la implementación del proyecto. Las principales tareas de esta fase son:

- Definición del alcance y contextualización
- Planificación temporal
- Gestión económica y sostenibilidad
- Presentación preliminar
- Revisión de las competencias
- Documentación y Presentación

2.1.2. Implementación del motor del juego

Esta fase estará centrada en la creación de los elementos básicos para la ejecución del juego. Una vez finalizada esta fase se tendrá un juego funcional el cual se podrá interactuar con él a través de comandos, y cuyos rivales realizan acciones aleatorias. Esta fase consta de cuatro tareas principales:

- **Investigación de las dinámicas básicas**: En la cual se recogerá la información del funcionamiento básico del juego como el cálculo de daño o la relación entre los tipos de ataque.
- **Diseño del motor**: En la que se ideará como se implementará el motor del juego.
- **Obtención de datos**: La cual se encargara de crear una base de datos con toda la información necesaria de cada Pokémon para el correcto funcionamiento de las funciones básicas que se implementarán del juego.

- **Creación del motor**: Esta tarea se encargará de implementar las dinámicas investigadas en la tarea "Investigación de las dinámicas básicas", y usarán los datos obtenidos en la tarea "Obtención de datos" para dotar al juego de comportamiento similar al original.
- **Testeo y corrección de errores**: El objetivo principal de esta tarea es comprobar el correcto funcionamiento de toda la fase y corregir posible errores que se hayan pasado por alto durante las implementación de las demás tareas.

2.1.3. Implementación de la interfaz del juego

Esta fase estará dedicada a la creación y implementación de la interfaz gráfica del juego. Esta interfaz tendrá como principal cometido facilitar la interacción con el usuario/jugador humano. No se podrá comenzar el desarrollo de esta fase hasta haber acabado la anterior, la implementación del motor del juego.

Esta interfaz tendrá tres funciones principales: visualizar el estado actual de la batalla para facilitar la comprensión del estado actual al jugador, mostrar frases que expliquen las acciones que se están levando a cabo en cada momento, y por último facilitar la selección de la acción que se llevara acabo por parte del jugador. Las siguientes tareas son las que conforman esta fase:

- **Investigación**: Investigar las distintas opciones que existen para poder implementar esta fase.
- **Diseño de la interfaz**: En la que se diseñará como se implementará la interfaz gráfica.
- **Implementación de la visualización de la batalla**: Centrada en implementar la primera función principal de la interfaz.
- **Implementación del dialogo**: Centrada en implementar la segunda función principal de la interfaz.
- **Implementación de la selección de acción**: Centrada en implementar la tercera función principal de la interfaz.
- **Integración con el motor**: Ajuntar el desarrollo de esta fase con la fase anterior.
- **Testeo y corrección de errores**: Centrada en la corrección de errores no detectados durante las tareas de implementación.

2.1.4. Implementación del agente RL

En esta fase se creará una versión básica del agente que usará el aprendizaje por refuerzo para jugar el juego. Esta fase podrá comenzar en cuanto se haya terminado la fase de implementación del motor del juego. Se aplicaran los métodos básicos para crear una versión simple que permita la integración del agente al juego y probar las funciones básicas del agente. Por lo tanto esta fase se puede dividir en las siguientes tareas:

- **Investigación**: Investigar las posibles maneras de implementar un agente de aprendizaje por refuerzo y los diversos métodos que se pueden usar.
- **Diseño del agente**: En la que se ideará como se implementará la primera versión del agente.
- **Implementación del modelo básico**: Implementará un modelo básico que sea capaz de elegir acciones de una forma no aleatoria a partir de información similar a la que el juego proporcionaría.
- **Integración con el juego**: Unirá el modelo creado con el juego, en forma de agente, para que pueda aprender y jugar a este.
- **Testeo y corrección de errores**: Corregirá errores no detectados durante las tareas de implementación.

2.1.5. Extensión del Agente

En esta fase se harán modificaciones al modelo creado en la fase anterior para que sea capaz de aprender del propio jugador, y consiga crear estrategias cooperativas con él. Esta fase podrá empezar a ser realizada en cuanto se acabe la fase anterior, la implementación del agente RL. Las tareas que conforman esta fase son:

- **Diseño del modelo para aprender del jugador**: En la que se ideará como se mejorará el actual modelo para que aprenda del jugador.
- **Implementación del modelo**: Se modificará el modelo básico para que pueda aprender también de las acciones que tome el jugador.
- **Evaluación básica del modelo**: Se llevará a cabo un evaluación simplificada del modelo para compáralo con su antecesor.
- **Diseño del modelo para la cooperación**: En la que se ideará como se mejorará el actual modelo para que aprenda a cooperar con el jugador.

- **Implementación del modelo**: Se modificará el modelo anterior para que pueda intentar realizar acciones cooperativas con el jugador.
- **Evaluación básica del modelo**: Se llevará a cabo una evaluación simplificada del modelo para compararlo con su antecesor.

2.1.6. Evaluación del Agente

Para terminar la parte correspondiente al modelo, se realizará una evaluación completa del agente resultante de la fase anterior para poder sacar conclusiones del proyecto, y por tanto podrá ser llevada a cabo en cuanto se termine la fase anterior. Esta fase se puede desglosar en estas dos tareas:

- **Entrenamiento del modelo final**: Entrenar el modelo final con muchas horas de practica para obtener un modelo lo suficientemente bueno.
- **Análisis de rendimiento y eficacia**: Analizar el resultado del modelo, como por ejemplo medir los combates ganados, comprobar si a aprendido a jugar o a cooperar.

2.1.7. Documentación Final

Para termina con el proyecto se documentará todo el proyecto y se preparará una presentación de los resultados de este. Esta fase al igual que la de la sección 2.1.1 también es paralelizable debido a que no depende de la finalización de ninguna tarea para poder comenzar a realizarse. Básicamente esta última fase consta de las siguientes dos tareas:

- **Documentar el proyecto**: Realizar la documentación del proyecto realizado.
- **Presentación Final**: Crear todo el material necesario para la realización de la presentación final.

2.2. Recursos

Durante la realización de este proyecto esta previsto utilizar los siguientes recursos:

Hardware

- **Ordenador personal de sobremesa**: Intel Core i3-6100 3.7 GHz, 16GB RAM, 120GB SSD + 1TB HDD.
- **Portátil personal**: Intel Core i7-7500U 2.7 GHz, 8GB RAM, 256GB SSD.

Software

- **Windows 10**: El sistema operativo que se usará principalmente durante el proyecto.
- **LaTeX**: El editor principal para la redacción de los documentos de las fases de documentación (2.1.1 y 2.1.7). En concreto se usará el editor online ShareLaTeX[1].
- **GitHub**: El repositorio donde se guardará el código del proyecto.
- **Atom**: El editor de textos que se usará para escribir el código del proyecto.
- **Python**: El lenguaje de programación que se usará para la implementar todo el proyecto.
- **PokeAPI**: API abierta de Pokémon disponible online [16]. Se utilizará en la fase de la creación del motor del juego (2.1.2) para la obtención de datos.
- **PyGame**: Librería de Python utilizada para crear juegos [14]. Se utilizarán básicamente las funciones de visualización de imágenes, con el fin de mostrar los elementos del juego, y las funciones de gestión de eventos, para facilitar la iteración con el usuario durante la fase de la creación. Se utilizará únicamente durante la fase de creación de la interfaz gráfica (2.1.3).
- **Keras**: Librería de Python utilizada para crear modelos de redes neuronales profundas [17]. Se utilizara para implementar el modelo y sus extensiones durante las fases relacionadas con el agente (2.1.4, 2.1.5 y 2.1.6).

2.3. Plan de acción y alternativas

Plan de acción

El plan de trabajo durante la primera mitad del proyecto es trabajar de forma paralela en las fase de documentación del hito inicial y las tres primeras fases de implementación (crear el motor, crear la interfaz y crear el modelo básico). Esto se debe a que son fases sencillas y fácilmente compaginables con la documentación inicial.

La segunda mitad del proyecto se dedicará a mejorar y extender el modelo para poder obtener los resultados deseados. De esta forma durante esta parte del proyecto se puede tener más tiempo para corregir posibles errores y para dedicarle más tiempo al desarrollo del agente, la tarea más compleja del proyecto.

[1] `https://www.sharelatex.com`

Alternativas

El plan alternativo para las fases de la creación del motor del juego y para la implementación de la interfaz en caso de que el problema sea la falta de tiempo previsto, es simplemente aumentar las horas de trabajo dedicadas al día con el fin de poder acabarlo dentro del plazo previsto. En caso de que el problema sea alguna dificultad de implementación el plan alternativo sería simplificar las funcionalidades del juego.

El riesgo de estas fases es realmente bajo ya que la fase de creación del motor del juego se podría ver como la aplicación de las fórmulas del cálculo de daño, y la interfaz del juego ni siquiera es esencial, ya que solo simplifica la evaluación del agente y ayuda a entender el estado del juego. Además, su coste temporal no sería muy elevado, ya que simplificar el juego se puede ver como la eliminación de algunos factores en el cálculo del daño, como por ejemplo los factores aleatorios. Aun así no sería bajo debido al aumento de horas de trabajo y a la perdida de tiempo en la implementación de las funcionalidades eliminadas.

Por otro lado tenemos las fases del agente. En concreto, el plan alternativo para la fase de extensión del modelo consiste en simplificar aun más el videojuego. De esta manera debería ser más fácil poder observar algún tipo de cooperatividad o aprendizaje de las acciones del jugador. Esta alternativa se usaría solo en caso de no poder implementar alguna de las extensiones básicas del juego como la integración con el juego o el aprendizaje a través del jugador, ya sea por falta de tiempo, falta de conocimientos o resultados peores a los mínimos esperados.

El riesgo de estas fases es bastante elevado, ya que es la fase más importante y difícil de realizar del proyecto. Además el plan alternativo tendría un coste temporal bastante elevado debido al tiempo que consume el entrenamiento de un modelo, el cual en caso de tener que rehacer el algoritmo sería tiempo perdido. Aun así, se ha previsto bastante tiempo para su desarrollo como para poder asegurar que se podrá terminar el proyecto a tiempo.

2.4. Diagrama de Gantt y tiempo estimado final

2.4.1. Gráfica de Gantt

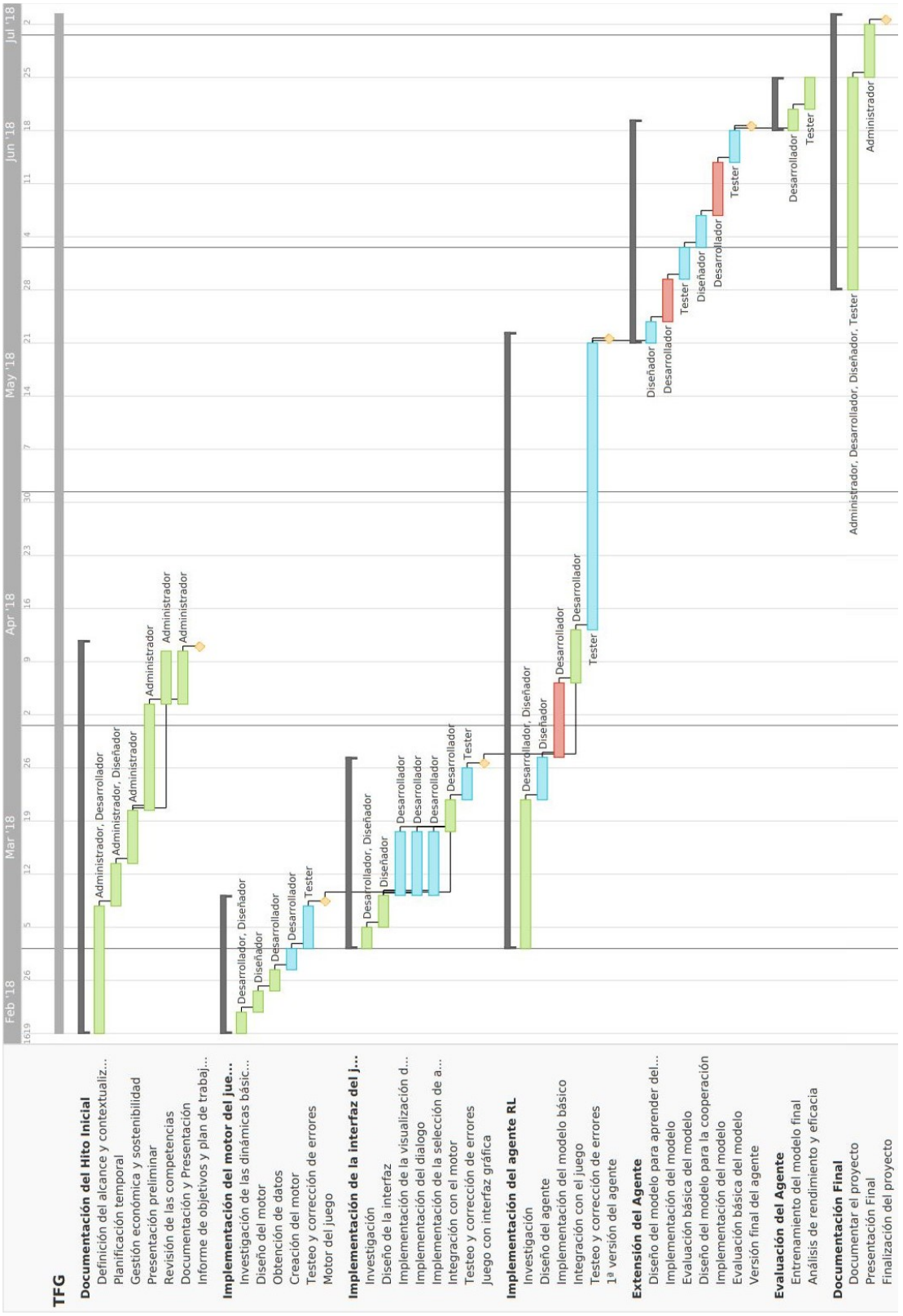

FIGURA 3: Diagrama de Gantt de la planificación final

En la figura anterior, la 3, se puede ver el diagrama de Gantt[2] final de este proyecto. Los colores de cada tarea indican su nivel de riesgo. Así pues, el verde indica bajo riesgo, el azul indica un riesgo medio y el rojo indica un nivel de riesgo alto.

También se pueden apreciar los hitos principales que se han conseguido después de cada fase. Estos son: el informe de objetivos y plan de trabajo GEP, el motor del juego, el juego con interfaz gráfica, la primera versión del agente, la versión final del agente y la finalización del proyecto. No se han incluido las reuniones de seguimiento con el directo del proyecto dado que estas no tenían una fecha especifica.

A continuación, la tabla 1 muestra la parte textual del diagrama de Gantt.

Num.	**Tarea**	**Inicio**	**Fin**	**Precedencia**	**Responsables**
1	**Doc. Inicial**	19/2/18	9/4/18		
1.1	Alcance	19/2/18	6/3/18		Admin, Des
1.2	Planificación	7/3/18	12/3/18	1.1	Admin, Dis
1.3	Gestión	13/3/18	19/3/18	1.2	Administrador
1.4	Presentación	20/3/18	2/4/18	1.3	Administrador
1.5	Competencias	3/4/18	9/4/18	1.3	Administrador
1.6	Doc. y Presentación	3/4/18	9/4/18	1.4	Administrador
2	**Imp. del motor**	19/2/18	6/3/18		
2.1	Investigación	19/2/18	20/2/18		Des, Dis
2.2	Diseño del motor	21/2/18	22/2/18	2.1	Diseñador
2.3	Obtención de datos	23/2/18	26/2/18	2.2	Desarrollador
2.4	Creación del motor	27/2/18	28/2/18	2.3	Desarrollador
2.5	Testeo y corrección	1/3/18	6/3/18	2.4	Tester
3	**Imp. de la interfaz**	1/3/18	23/3/18		
3.1	Investigación	1/3/18	2/3/18		Des, Dis
3.2	Diseño interfaz	3/3/18	7/3/18	3.1	Diseñador
3.3	Imp. vis. batalla	8/3/18	15/3/18	3.2	Desarrollador
3.4	Imp. dialogo	8/3/18	15/3/18	3.2	Desarrollador
3.5	Imp. selección	8/3/18	15/3/18	3.2	Desarrollador
3.6	Integración	16/3/18	20/3/18	2.5, 3.3, 3.4, 3.5	Desarrollador
3.7	Testeo y corrección	21/3/18	23/3/18	3.6	Tester

[2] La cual se ha obtenido gracias a TeamGantt: `https://www.teamgantt.com/`

Num.	Tarea	Inicio	Fin	Precedencia	Responsables
4	**Imp. del Agente RL**	1/3/18	18/5/18		
4.1	Investigación	1/3/18	20/3/18		Des, Dis
4.2	Diseño del agente	21/3/18	26/3/18	4.1	Diseñador
4.3	Imp. del modelo	27/3/18	4/4/18	4.2	Desarrollador
4.4	Integración	5/4/18	11/4/18	3.6, 4.3	Desarrollador
4.5	Testeo y corrección	12/4/18	18/5/18	4.4	Tester
5	**Extensión del Agente**	18/5/18	15/6/18		
5.1	Diseño para aprender	18/5/18	22/5/18	4.4	Diseñador
5.2	Implementación	23/5/18	28/5/18	5.1	Desarrollador
5.3	Evaluación básica	29/5/18	31/5/18	5.2	Tester
5.4	Diseño para cooperar	1/6/18	5/6/18	5.3	Diseñador
5.5	Implementación	6/6/18	12/6/18	5.4	Desarrollador
5.6	Evaluación básica	13/6/18	15/6/18	5.5	Tester
6	**Evaluación**	18/6/18	22/6/18		
6.1	Entrenamiento	18/6/18	19/6/18	5.6	Desarrollador
6.2	Análisis	20/6/18	22/6/18	6.1	Tester
7	**Doc. Final**	28/5/18	29/6/18		
7.1	Documentación	28/5/18	24/6/18	5.6	Adm,Des,Dis,Tes
7.2	Presentación	25/6/18	29/6/18	7.1	Administrador

Cuadro 1: Parte textual del diagrama de Gantt

2.4.2. Tiempo estimado

A continuación, la tabla 2 muestra la duración que se ha estimado que ha durado cada fase del proyecto:

Fase	Tiempo estimado
Documentación del Hito Inicial	90
Implementación del motor del juego	30
Implementación de la interfaz del juego	60
Implementación del Agente RL	110
Extensión del Agente	120
Evaluación del Agente	50
Documentación Final	90
Estimación Total	550

Cuadro 2: Estimación de los tiempos de cada fase

2.5. Evaluación de la planificación

2.5.1. Desviación de la planificación inicial

A mediados de Mayo, aún se estaba finalizando la fase de Implementación del agente RL, en concreto se estaba acabando de realizar la tarea de testeo y corrección de errores, tal y como muestra el diagrama de Gantt de la imagen 3. Esta fase estaba planificada para que solo durará un par de días, y su finalización estaba programada para mediados de Abril, más o menos a la vez que la fase de Documentación Inicial asociada a la signatura de GEP, tal y como se puede observar en al imagen 4, la cual es el diagrama de Gantt de la planificación inicial.

Este retraso fue debido principalmente a algunos errores cometidos durante la implementación del agente que dificultaron el aprendizaje del agente hasta tal punto que siempre realizaba la misma acción fuese cual fuese el estado del juego. Estos errores ya fueron corregidos gracias a la ayuda del director del proyecto, y esta fase fue finalizada hacia el 20 de Mayo, acarreando un retraso de aproximadamente un mes.

A pesar de este retraso, el proyecto no se vio muy afectado. Esto se debe a que para finalizar el proyecto solo hacía falta implementar el aprendizaje del jugador y la cooperación, y evaluar los resultados de estos para poder sacar las conclusiones, lo cual pudo hacerse en el mes que quedaba. En parte, esto se pudo hacer dado que ya se comenzó a pensar como se podrían hacer las fases restantes mientras se intentaba solucionar el problema que causó el retraso.

2.5.2. Cambios en planificación

Así pues, el tiempo dedicado a la fase de implementación del agente RL se alargo unas 50 horas. Por tanto, el tiempo dedicado a esa fase paso de 60 horas estimadas inicialmente, a 110 horas. De igual manera, el tiempo total dedicado al proyecto pasó de ser de 500 horas a aumentar hasta las 550 horas.

Debido al retraso, se han realizado cambios en la planificación. Estos cambios han sido básicamente aumentar las horas de trabajo al día de las tareas restante para poder realizar el mismo trabajo en menos tiempo y así poder terminar el proyecto en el plazo previsto.

La siguiente figura muestra el diagrama de Gantt de la panificación inicial, a modo de referencia para poder observar mejor los cambios entre la planificación inicial y final:

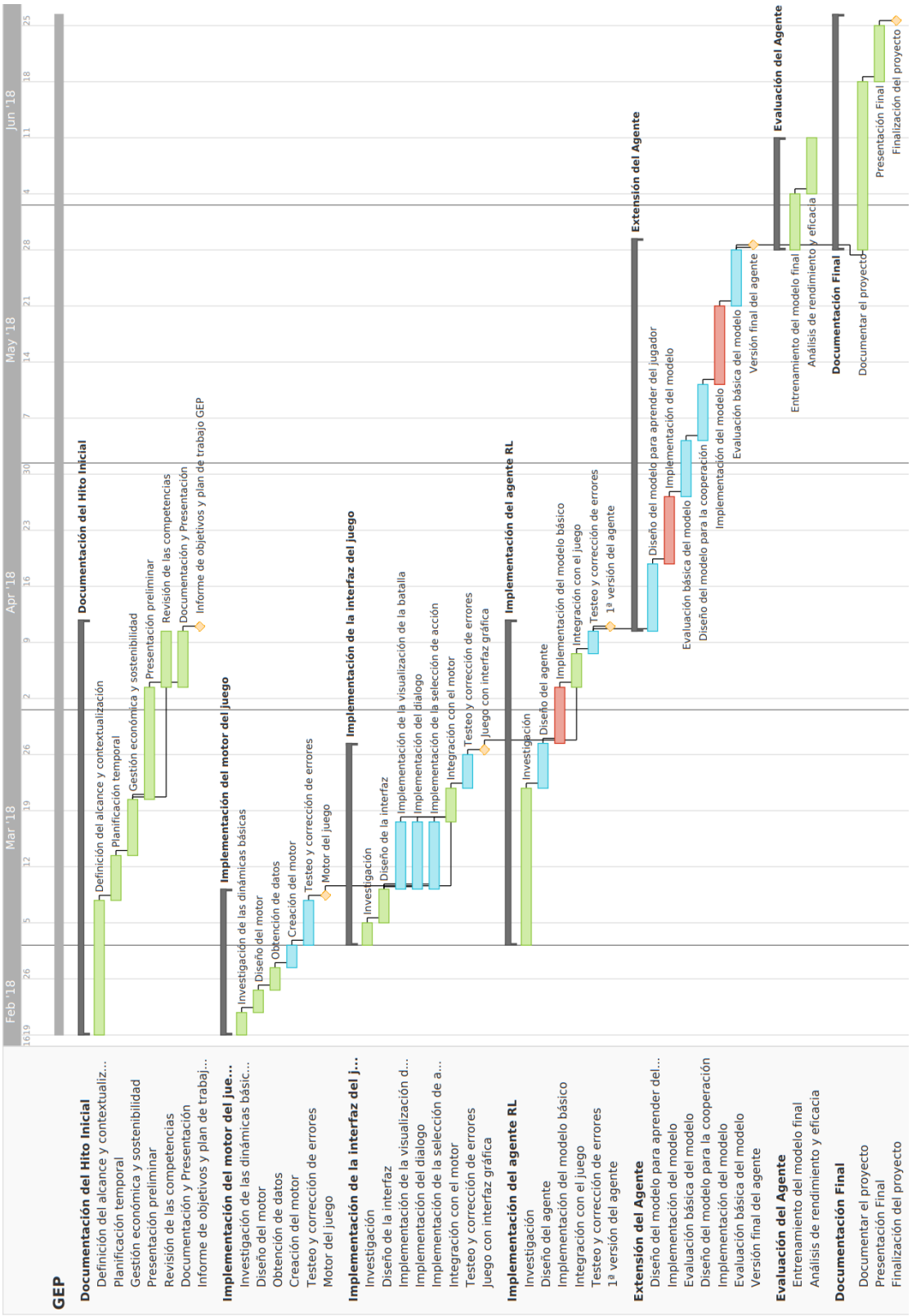

FIGURA 4: Diagrama de Gantt de la planificación inicial

Capítulo 3

Presupuesto y sostenibilidad

Este capítulo pretende detallar la sostenibilidad del proyecto, y dado que es un TFG se entrará en detalle en la sostenibilidad económica durante la fase de PPP (Proyecto Puesto en Producción) mediante una estimación de los costes en la sección 3.1. Esta sección tendrá en cuenta el presupuesto de los costes directos (sección 3.1.1), el presupuesto de los costes indirectos (sección 3.1.2), el presupuesto de los imprevistos (sección 3.1.3) y el presupuesto de la contingencia (sección 3.1.4), acabando con un resumen de el presupuesto total (sección 3.1.5). Además, se hablará de la gestión que se hará de los costes durante el desarrollo de este proyecto (sección 3.1.6).

Además, también se hablará de los costes finales del proyecto en la sección 3.2, la cual incluye una sección donde se comentarán los costes producidos por la desviación de la planificación inicial (sección 3.2.1), a aparte de la sección de los costes finales propiamente dichos (sección 3.2.2).

Por otra parte, en la sección 3.3, se realizará un reflexión sobre la sostenibilidad de este proyecto tanto en la área económica (sección 3.3.1), como en la área ambiental (sección 3.3.2) así como en la área social (sección 3.3.3).

3.1. Estimación de Costes

Esta sección está dedicada a realizar una estimación del coste económico que tendrá el proyecto. Se tendrán en cuenta los costes directos, los costes indirectos, de los imprevistos y de la contingencia.

A demás, se dedicará una sección a la propuesta de diferentes mecanismos para el control de la desviación respecto el presupuesto estimado.

3.1.1. Costes directos

Los costes estimados de está sección son los relacionados con los recursos detallados en la sección de recurso del apartado anterior. También se incluirá el coste asociado a los recursos humanos que necesitará el proyecto, los cuales principalmente son los salarios del administrador del proyecto, del diseñador y desarrollador de software y del *tester* del programa.

Recursos Hardware

En la tabla 3 que hay a continuación se detallan los costes estimados de los elementos hardware que se utilizarán durante el desarrollo de este proyecto. La amortización se ha hecho suponiendo un uso de 8 horas diarias solo entre semana, es decir que para una vida útil de 4 años se aproxima que las horas reales con las que se puede trabajar con ese hardware son unas 8000.

Producto	**Precio**	**Vida Útil**	**Tiempo de uso**	**Amortización**
Ordenador de sobremesa	800 €	4 Años	340 h	36 €
Portátil personal	600 €	4 Años	160 h	12 €
Total	-	-	-	48 €

Cuadro 3: Presupuesto para los recursos hardware

Recursos Software

Como se puede apreciar en la tabla 4, el coste estimado para software es de 0 € dado que todo el software utilizado es *open source*, a excepción de Windows 10, cuyo precio va incluido en el precio de los ordenadores previamente mencionados.

Producto	**Precio**	**Vida Útil**	**Amortización**
Windows 10	0 €	-	0 €
LaTeX	0 €	-	0 €
GitHub	0 €	-	0 €
Atom	0 €	-	0 €
Python	0 €	-	0 €
PokeAPI	0 €	-	0 €
PyGame	0 €	-	0 €
Keras	0 €	-	0 €
Total	-	-	0 €

Cuadro 4: Presupuesto para los recursos software

Recursos Humanos

En la tabla 5 se especifica el tiempo estimado que se dedicará a cada actividad en función de si se realizará siendo el administrador del proyecto, el diseñador, el desarrollador del software o el *tester*. Esta tabla será útil para el cálculo del coste de los recursos humanos.

Fase	**Administrador**	**Diseñador**	**Desarrollador**	***Tester***
Documentación Inicial	70 h	10 h	10 h	0 h
Imp. del motor	0 h	5 h	20 h	5 h
Imp. de la interfaz	0 h	10 h	35 h	15 h
Imp. del Agente RL	0 h	20 h	20 h	20 h
Extensión del Agente	0 h	40 h	50 h	30 h
Evaluación del Agente	0 h	0 h	25 h	25 h
Documentación Final	50 h	10 h	25 h	5 h
Total	120 h	95 h	185 h	100h

Cuadro 5: Estimación de los tiempos para cada papel

Esta estimación se ha basado en las subtareas de cada actividad, mencionadas en la sección 2.1 y en el diagrama de Gantt 4. Por lo tanto, este cálculo se ha realizado teniendo en mente la planificación inicial.

El coste estimado para los recursos humanos resultado de la estimación de las horas trabajadas de cada papel se puede observar en la tabla 6 que hay a continuación. Para el cálculo se ha utilizado una aproximación del promedio del salario de distintas ofertas de trabajo encontradas online.

Papel	**Horas**	**Precio/Hora**	**Salario**
Administrador	120 h	50 €/h	6.000 €
Diseñador	95 h	35 €/h	3.325 €
Desarrollador	185 h	30 €/h	5.550 €
Tester	100 h	20 €/h	2.000 €
Total	-	-	16.875 €

Cuadro 6: Presupuesto para los recursos humanos

Total del coste directo

Finalmente, una vez vistas todas las tablas anteriores (3, 4 y 6), el presupuesto final asociado a los costes directos del proyecto se pueden resumir en la tabla 7.

Recurso	**Coste**
Hardware	48 €
Software	0 €
Humano	16.875 €
Total	16.923 €

Cuadro 7: Costes directos

3.1.2. Costes indirectos

En la tabla 8 podemos ver el coste indirecto de este proyecto que básicamente incluye la energía consumida por los ordenadores y la cuota de Internet.

Para calcular la energía consumida se ha estimado que la potencia del ordenador de sobremesa y del portátil es de 290 W y 45 W respectivamente. Para calcular esta estimación se ha utilizado una calculadora de potencia online[1] para el ordenador de sobremesa, y una *review*[2] para la estimación del consumo del portátil.

Además, el tiempo de uso de cada uno se estima en 340 h y 160 h respectivamente. Esto se debe a que se prevé que se usará más el ordenador de sobremesa que el portátil. Al igual que las horas dedicadas a los recursos humanos, las horas de uso del hardware se ha calculado teniendo en mente la panificación inicial.

En cuanto al precio de la energía, se ha aproximado a 0,12 €/kWh dado que se ha realizado el promedio de los valores encontrados en una web online[3].

[1] `https://outervision.com/power-supply-calculator`
[2] `http://hexus.net/tech/reviews/laptop/99145-medion-akoya-s3409-ultrabook/?page=2`
[3] `https://tarifaluzhora.es/`

Por la parte de coste del Internet se ha aproximado de una tarifa online[4] a 20 €/mes, debido a que el coste real esta contabilizado en un pack que incluye otros servicios como el teléfono.

Producto	**Precio**	**Cantidad**	**Coste**
Energía	0,12 €/kWh	105,8 kWh	12,70 €
Internet	20 €/mes	5 meses	100 €
Total	-	-	112,70 €

Cuadro 8: Costes indirectos

3.1.3. Costes de los imprevistos

En la tabla 10 podemos ver la estimación del coste de las posibles horas extras que se podrían llevar a cabo debido a posibles problemas, obstáculos, retrasos o modificaciones no previstas durante el desarrollo del proyecto.

Papel	**Horas**	**Precio/Hora**	**Salario**
Administrador	10 h	50 €/h	500 €
Diseñador	10 h	35 €/h	350 €
Desarrollador	20 h	30 €/h	600 €
Tester	10 h	20 €/h	200 €
Total	50 h	-	1.650 €

Cuadro 9: Costes de las horas extras

Causa	**Riesgo**	**Importe**	**Coste**
Horas extras	25 %	1.650 €	412,50 €

Cuadro 10: Costes de los imprevistos

3.1.4. Costes de las contingencias

Dado el nivel de detalle que muestra el diagrama de Gantt del capítulo anterior (sección 2.4), se ha fijado un porcentaje de contingencia del 15 % el cual se aplicará a la suma de los costes directos y los costes indirectos (secciones 3.1.1 y 3.1.2). La estimación final del coste de las contingencias se muestra en la tabla 11 que hay a continuación.

[4] `http://www.movistar.es/particulares/tienda/comparador-tarifas-internet/`

Tipo de coste	**Estimación**
Coste directo	16.923 €
Coste indirecto	112,70 €
Suma directo y indirecto	17.035,70 €
Contingencia (15 %)	2.555,40 €

Cuadro 11: Costes de las contingencias

3.1.5. Costes final

Una vez vistas todas las secciones anteriores, el precio estimado final del proyecto queda reflejado en la siguiente tabla.

Tipo de coste	**Estimación**
Directo	16.923 €
Indirecto	112,70 €
Imprevistos	412,50 €
Contingencia	2.555,40 €
Total	20.003,60 €

Cuadro 12: Estimación de coste final

3.1.6. Gestión del presupuesto

Como ya se ha mencionado anteriormente, el principal problema que puede surgir en el transcurso del proyecto es la falta de tiempo. Esto se debe a que es muy improbable que se necesite variar los recursos hardware o software mencionados anteriormente. Además, en caso de necesitar algún otro recurso software seguramente sería *open source* al igual que los ya mencionados, y por tanto no causaría cambios en la estimaciones hechas.

Tanto las posibles horas extras como los improbables imprevistos en otros factores están contemplados en la estimación de los costes para imprevistos y los costes de contingencia. Aun así, para poder realizar un control de los costes del proyecto se utilizarán indicadores al final de cada fase y para todos los costes. De esta forma se puede evaluar dónde se han producido las desviación, porqué y de qué importe hablamos en cada caso de desviación. Así se podrá adelantarse a posibles problemas relacionados con el presupuesto. Así pues se utilizarán los siguientes indicadores:

$$Desviación\_coste = (Coste\_Estimado - Coste\_Real) \cdot Horas\_Reales$$
$$Desviación\_horas = (Horas\_Estimadas - Horas\_Reales) \cdot Coste\_Estimado$$

Así mismo también se utilizarán al final del proyecto unos indicadores para calcular los desvíos totales. Estos indicadores son los siguientes:

$$Desviación\_total\_coste = Total\_coste\_Estimado - Total\_coste\_Real$$
$$Desviación\_total = Total\_presupuesto - Total\_Real$$

3.2. Coste Final

3.2.1. Coste de la desviación temporal

Tal y como se comentaba en la sección 2.5, se produjo una desviación la cual se calcula que fue un retraso de unas 50 horas. Estas habrían sido realizadas tanto por el *tester* como por el desarrollador de software. Así pues, el precio de esta desviación en la planificación, en cuanto a recursos humanos y costes indirectos se refiere, se puede ver reflejada en la siguiente tabla:

	Cantidad	**Precio**	**Coste**
Desarrollador	10 h	30 €/h	300 €
Tester	40 h	20 €/h	800 €
Energía	12.05 kWh	0.12 €/kWh	1,45 €
Total	-	-	1.101,45 €

Cuadro 13: Coste de la desviación temporal

La cantidad de kWh se ha calculado teniendo en cuenta que se utilizó el portátil 10h y las otras 40h se utilizó el ordenador de sobremesa.

Hay que tener en cuenta que estos 1.101,45€ no se deben añadir al presupuesto como un nuevo gasto, sino que debe ser contemplados dentro de la partida reservada para los imprevistos (de 412,50€) y la partida de contingencia (de 2.555,40€). Eso si, se deberán de tener en cuenta como un coste más al calcular el cose final del proyecto dado que es una desviación de la estimación de este.

3.2.2. Coste Final

Así pues, los costes reales una vez finalizado el proyecto son los costes estimados para los costes directos (los ordenadores y los salarios de los empleados principalmente), los indirectos (la energía consumida y el uso de Internet), y los gastos producidos por la desviación temporal mencionada en la sección 2.5, los cuales afectaron principalmente al salario del *tester* como al del desarrollador de software, y a la energía consumida por los recursos hardware.

Así pues, en la siguiente tabla se puede ver los costes finales del proyecto.

	Presupuesto	**Desviación**	**Coste Final**
Ordenador de sobremesa	36 €	-	36 €
Portátil personal	12 €	-	12 €
Software	0 €	-	0 €
Salario Administrador	6.000 €	-	6.000 €
Salario Diseñador	3.325 €	-	3.325 €
Salario Desarrollador	5.550 €	300 €	5.850 €
Salario *Tester*	2.000 €	800 €	2.800 €
Energía	12,70 €	1,45 €	14,15 €
Internet	100 €	-	100 €
Total	17.035,70 €	1.101,45	18.137,15 €

Cuadro 14: Coste final del proyecto

3.3. Sostenibilidad

Esta sección esta dedicada a la evaluación de la sostenibilidad del proyecto en la área económica, la área ambiental y la área social. Dado que el proyecto es para un TFG, se hablará de la fase de PPP (Proyecto Puesto en Producción), aunque también se hará una reflexión sobre la posible vida útil del proyecto.

3.3.1. Sostenibilidad económica

Este documento ya tiene los costes del proyecto detallados en la sección 3.1 donde se tienen en cuenta los costes materiales, humanos e indirectos.

Sería complicado reducir el coste de la estimación propuesta debido a que la mayor parte del presupuesto esta dedicada a pagar las horas de desarrollo del proyecto, las cuales es posible que se tengan que aumentar por algún imprevisto.

Se podría intentar ahorrar algo de dinero utilizando unos recursos hardware más baratos, pero eso provocaría un aumento en el tiempo de entrenamiento del modelo y por tanto también de la horas de trabajo, por lo que no es una buena opción.

El coste estimado del proyecto puede ser elevado, pero teniendo en cuenta que una de las partes interesadas es la industria de los videojuegos, la estimación ya no parece tan elevada. Además, en lo referente a la parte económica, la industria de los videojuegos es la que saldría ganando, tal y como se menciona en la subsección 1.1.4.

Además, si el proyecto obtiene buenos resultados, se podría llegar a usar para cualquier juego que usase un agente que tenga que coopera con el jugador. Solo sería necesario hacer algunas modificaciones y entrenarlo para que pudiese adaptarse a cada juego. Esto evitaría tener que realizar un agente personalizado para el juego, lo que reduciría los costes de producción del videojuego.

3.3.2. Sostenibilidad ambiental

Como se ha podido ver en el apartado 2.2, el único recurso utilizado que afecta al medio ambiente es la electricidad consumida por los ordenadores personales. Tal y como se ve en la subsección 3.1.2 esta no es muy elevada.

Para reducir el consumo eléctrico se podría trabajar solo con el portátil personal, el cual es el que menos consume de los dos. Pero esta opción aumentaría de forma significativa el coste económico dado que aumentaría el tiempo necesario para entrenar el modelo debido a la menor potencia de cálculo del portátil. Por este motivo se ha decidido consumir un poco más de energía a cambio de reducir el coste económico.

Otra manera de reducir el consumo eléctrico sería reutilizando código. En este aspecto tenemos dos grandes bloques: el juego y el modelo. Por la parte del modelo, dado que el objetivo principal es crear uno de nuevo, lo único reutilizable es código base para crearlo, que en nuestro caso es la librería de Keras [15]. Por otra parte tenemos el juego, el cual se creará desde cero. Si se utilizara algún juego ya creado, la dificultad que habría para implementar el agente por encima requeriría mucho más tiempo de desarrollo y por tanto consumiría más energía.

En caso de que el proyecto consiga unos buenos resultados y se utilizara para la creación de nuevos videojuegos, se podría reducir el tiempo de desarrollo y por tanto el consumo de energía. Aun así hay que tener presente que la fase de creación del agente sería una parte pequeña del desarrollo de un juego, y que visto en conjunto la reducción de energía del proceso no sería muy importante.

3.3.3. Sostenibilidad social

A nivel personal este proyecto aportará conocimiento y experiencia sobre el aprendizaje por refuerzo y redes neuronales. A la vez aportará conocimiento y experiencia sobre la gestión y creación de proyectos.

Respecto a las soluciones ya existentes este proyecto ni mejora ni empeora la cualidad de vida de las partes interesadas de este proyecto. A la vez no existe una necesidad real del proyecto dado que el objetivo es mejorar algo ya existente y funcional.

3.3.4. Matriz de sostenibilidad

A modo de conclusión de esta sección, en la figura 15, se encuentra la matriz de sostenibilidad. Dado que este proyecto es para un trabajo de final de grado, en la matriz solo aparece la columna de la fase de PPP (Proyecto Puesto en Producción).

Tipo	**PPP**
Ambiental	7
Económico	8
Social	7

Cuadro 15: Matriz de sostenibilidad

Capítulo 4

El juego

Este capítulo pretende describir el juego desarrollado, el cual es utilizado como entorno de entrenar para el agente. Así pues, para empezar se comentará por que se ha decidido hacer el juego basado en Pokémon (sección 4.1), y luego se explicará tanto las mecánicas básicas del juego (sección 4.2) como la manera de obtener la información relativa al juego (sección 4.3) y el procedimiento utilizado para simular una batalla (sección 4.4). Además se explicará la utilidad y las partes de la interfaz gráfica creada (sección 4.5).

4.1. Elección del juego

El proyecto esta dedicado a crear un agente capaz de interactuar con un aliado humano dentro de un videojuego. Por este motivo es importante decidir que juego será la base donde entrenar al agente.

El juego elegido debe de permitir la cooperatividad de tal forma que ambos jugadores puedan realizar las mismas acciones, para que así el agente pueda aprender fácilmente de las acciones del compañero. Además, el juego debería de ser sencillo para que el agente sea capaz de aprender a jugarlo pero manteniendo cierto nivel de dificultad para si poder observar como mejora el agente dependiendo del algoritmo utilizado. Como última condición, el juego deberá de poder proporcionar un acceso fácil a los datos de la partida, para que el agente los pueda utilizar, y además ser fácilmente modificable para poder añadir el agente implementado.

Dadas estas condiciones, se ha decidido que el juego será creado des de cero, para poder tener un gran control de este. Respecto a la cooperatividad, se ha elegido que el jugo será del estilo de luchas, en el que se puedan realizar batallas en equipo. Finalmente, para que el juego sea sencillo se elegirá un juego que este pensado para poder ser jugado por niños.

Una saga de juegos que encaja con esa definición son los videojuegos de la saga Pokémon. La mayoría de estos juegos son del tipo RPG (role-playing game) con mecánicas realmente complicadas con el objetivo de poder ofrecer un sinfín de estrategias a los jugadores más experimentados, pero con unas mecánicas básicas fáciles de aprender.

Pero dado el tiempo limitado del proyecto, y que el proyecto se centra en el desarrollo del agente, solo se desarrollará la parte básica del juego, es decir los combates. Estos además también serán simplificados, de forma que se intentará tener un buen balance entre la dificultad y el tiempo de desarrollo.

Para concretar un poco más, y para aquellos que sepan un poco del tema, la versión implementada contiene los pokémon introducidos hasta la 4ª generación (unos 490 distintos), correspondientes a las versiones Diamante, Perla y Platino. Pero la información de los datos utilizados corresponden a la última versión (Sol y Luna) debido ha que esta información se extrae de una API actualizada (como veremos en la sección 4.3).

4.2. Mecánicas del juego

Como ya se ha mencionado, el juego creado se centra en las batallas doble de pokémon, las cuales se han simplificado por motivos de tiempo de implementación principalmente.

Para hacerlo se han eliminado alguna de la mecánicas que daban lugar a estrategias más complejas, como por ejemplo los efectos en el estado de un pokémon (congelado, paralizado, envenenado, ...) y los efectos secundarios que provocaban ciertos ataques, como provocar cambios en el clima, aumentar o disminuir características de los pokémon en combate, además de la posibilidad de usar objetos curativos o de relevar a los pokémon, entre otras.

Lo que si se ha conservado ha sido la mecánica de daño principal la cual es bastante compleja por si sola, tal y como se verá en la sección 4.4.4. Esta mecánica tiene en cuenta información tanto del pokémon que ataca como de que recibe del daño, así como del propio ataque. Esta información incluye valores y atributos como el ataque, la defensa, el tipo o la potencia del ataque, los cuales veremos más a fondo en las secciones 4.2.1, 4.2.2 y 4.2.3 que hay a continuación.

4.2.1. Los Tipos

Los tipos son atributos tanto de los movimiento (que se explicarán en la sección 4.2.2) como de los pokémon (que veremos en la sección 4.2.3). Hay un total de 18 tipos diferentes, los cuales son: **Acero, Agua, Bicho, Dragón, Eléctrico, Fantasma, Fuego, Hada, Hielo, Lucha, Normal, Planta, Psíquico, Roca, Siniestro, Tierra, Veneno y Volador**.

Entre ellos existen relaciones de ventajas y debilidades en diferentes grados que crean una mecánica fundamental en el juego. Cuando se elije que ataque se realizará, es muy impotente tener en cuenta cual es la relación que tiene el tipo del ataque con los tipos del pokémon al cual se quiere atacar, dado que el daño causado puede variar en gran medida dependiendo de la relación que se de entre los tipos.

Esta relaciones a veces están fundamentadas en la realidad como por ejemplo el agua es fuerte contra el fuego, o que el fuego es eficaz contra el hielo. En otras ocasiones las relaciones son un poco más extrañas o sacadas de refranes como por ejemplo, que el tipo roca sea eficaz contra el tipo volador viene del refrán 'matar a dos pájaros con una sola piedra'.

En cambio, otras veces la relaciones son más difíciles de explicar, dado que algunas solo están para equilibrar el juego como por ejemplo el tipo Hada que se introdujo solo para debilitar el tipo Dragón. Por esto, y al hecho de que los pokémon pueden tener más de un tipo (combinando la efectividad de estos como se explicará en la sección 4.4.4), muchas veces hasta que no se prueba un ataque en combate no se sabe si un tipo será efectivo o no contra ese pokémon, por lo que esta mecánica suele ser incierta la primera vez que te enfrentas a un nuevo pokémon.

Con tal de ayudar a aprender las relaciones entre tipos, en la tabla[1] 5 que se muestra a continuación resume las relaciones entre ellos. Las filas de esta tabla son el tipo del movimiento, y las columnas son el tipo del pokémon que recibe el ataque. Dentro de la tabla se encuentra la relación de efectividad entre los tipos indicada con un color y su correspondiente factor multiplicador (el cual se usa para el cálculo del daño final, tal y como se explicará en la sección 4.4.4).

Efectividad		Tipo del Pokémon del oponente																
		ACERO	AGUA	BICHO	DRAGÓN	ELÉCTRICO	FANTASMA	FUEGO	HADA	HIELO	LUCHA	NORMAL	PLANTA	PSÍQUICO	ROCA	SINIESTRO	TIERRA	VENENO	VOLADOR
TIPO DEL ATAQUE PROPIO	ACERO	½	½	-	-	½	-	½	x2	x2	-	-	-	-	x2	-	-	-	-
	AGUA	-	½	-	½	-	-	x2	-	-	-	-	½	-	x2	-	x2	-	-
	BICHO	½	-	-	-	-	½	½	½	-	½	-	x2	x2	-	x2	-	½	½
	DRAGÓN	½	-	-	x2	-	-	-	x0	-	-	-	-	-	-	-	-	-	-
	ELÉCTRICO	-	x2	-	½	½	-	-	-	-	-	-	½	-	-	-	x0	-	x2
	FANTASMA	-	-	-	-	-	x2	-	-	-	-	x0	-	x2	-	½	-	-	-
	FUEGO	x2	½	x2	½	-	-	½	-	x2	-	-	x2	-	½	-	-	-	-
	HADA	½	-	-	x2	-	-	½	-	-	x2	-	-	-	-	x2	-	½	-
	HIELO	½	½	-	x2	-	-	½	-	½	-	-	x2	-	-	-	x2	-	x2
	LUCHA	x2	-	½	-	-	x0	-	½	x2	-	x2	-	½	x2	x2	-	½	½
	NORMAL	½	-	-	-	-	x0	-	-	-	-	-	-	-	½	-	-	-	-
	PLANTA	½	x2	½	½	-	-	½	-	-	-	-	½	-	x2	-	x2	½	½
	PSÍQUICO	½	-	-	-	-	-	-	-	-	x2	-	-	½	-	x0	-	x2	-
	ROCA	½	-	x2	-	-	-	x2	-	x2	½	-	-	-	-	-	½	-	x2
	SINIESTRO	-	-	-	-	-	x2	-	½	-	½	-	-	x2	-	½	-	-	-
	TIERRA	x2	-	½	-	x2	-	x2	-	-	-	-	½	-	x2	-	-	x2	x0
	VENENO	x0	-	-	-	-	½	-	x2	-	-	-	x2	-	½	-	½	½	-
	VOLADOR	½	-	x2	-	½	-	-	-	-	x2	-	x2	-	½	-	-	-	-

FIGURA 5: Efectividad entre tipos

Dependiendo del color del indicador podemos saber como de efectivo será un ataque. Así pues, el color verde indica que el ataque será efectivo, el color naranja indica que será poco efectivo, el color negro indica que no producirá ningún daño y los que están marcados con un guión "-" indican que tendrán una efectividad normal. Por ejemplo, si el ataque es de tipo Dragón, este no causará daño a los pokémon de tipo Hada, será poco efectivo contra los de tipo Acero y será muy efectivo contra los de tipo Dragón.

[1] Tabla obtenida de [18]

4.2.2. Los Movimientos

Los movimientos son los distintos ataques que puede realizar un pokémon. Como ya se ha mencionado en el apartado anterior, cada ataque tiene un tipo asociado el cual determina en gran medida cuanto daño provocará este al enemigo. Sin embargo, este no es el único atributo que caracteriza a un movimiento y que tiene un gran efecto en el combate. Los atributos que caracterizan a un movimiento son: el **tipo**, la **clase de daño**, el **poder/potencia**, los **puntos de poder/PP**, la **precisión**, la **prioridad** y el **ratio de crítico**. Aunque en la versión original del juego tienen algún atributo más, para esta implementación se ha elegido conservar solo estos, dado que son los más importantes y fundamentales para el combate.

La clase de daño de un movimiento indica si un movimiento causa daño físico o especial, lo que puede hacer variar el daño final dependiendo del pokémon que lo usa y del que lo recibe (tal y como se explicará en la sección 4.4.4).

El poder o potencia del movimiento es el daño básico del ataque, el cual oscila entre valores de 10 y de 250. Aunque la verdad es que es bastante difícil encontrar movimientos con potencia superior al 100 (solo 68 de 395 movimientos) y aun más difícil si son superiores a 160 (solo 3: *v-create* (180 de potencia), *self-destruct* (200) y *explosion* (250)).

Además, cuanto más potencia tenga un movimiento, más probable es que tenga menos puntos de poder (o PP para abreviar) los cuales indican cuantas veces se puede usar ese movimiento. Esto se puede apreciar en el hecho de que de los 68 movimientos que tienen solo 5 PP, 58 de ellos tienen una potencia superior o igual a 100.

La precisión es la probabilidad de acertar el ataque, o visto desde otro punto de vista, de realizar el ataque. Normalmente es del 100 %, pero en algunos movimientos puede llegar a bajar hasta el 70 % (en casos muy extremos hasta el 50 % como *inferno*, *zap-cannon* y *dynamic-punch*). La reducción de la precisión suele estar relacionada con una alta potencia con el objetivo de mantener equilibrado el juego. Por esto, de los 21 movimientos que tiene una precisión inferior o igual al 80 %, 19 movimientos tienen una potencia superior o igual a 80.

En cuanto a la prioridad del ataque, este es un valor que crea un orden ente los movimientos haciendo que haya un relación de prioridad en la realización de un ataque. De forma similar, el ratio de crítico también es un valor que permite identificar a aquellos movimientos con más probabilidades de realizar un ataque crítico. Esta parte se verá más a fondo en la sección 4.4.2 y 4.4.4 respectivamente dado que son atributos más bien orientados al funcionamiento interno del juego.

En cuanto al aprendizaje de los movimientos por parte de los pokémon, los movimientos que puede aprender un pokémon son un rasgo distintivo de este por lo que están predefinidos, es decir, los pokémon no pueden realizar cualquier movimientos sino los que pueda aprender según el tipo de pokémon que sea. Aunque en general, y a modo de guía, un pokémon tiene más probabilidades de aprender movimientos del mismo tipo que los suyos.

Esto provoca que los movimientos tengan un desequilibrio en cuanto a cuantos pokémon pueden usarlo. Algunos son aprendidos por la mayoría de ellos, em concreto hay 8 movimientos especialmente comunes que pueden ser usados por más de 300 pokémon. Estos son: *headbutt* (349 pokémon), *mud-slap* (359), *double-edge* (376), *facade* (476), *secret-power* (476), *round* (476), *hidden-power* (477) y *snore* (481).

Sin embargo, hay 28 movimientos que solo pueden ser aprendidos por un único pokémon, lo cual les convierte en su movimiento característico. Estos pokémon suelen ser realmente poderoso, y los movimientos suelen tener mucho poder de ataque y pocos PP. En concreto estos movimientos son: *power-trip*, *steamroller*, *storm-throw*, *high-horsepower*, *psystrike*, *volt-tackle*, *throat-chop*, *liquidation*, *triple-kick*, *aeroblast*, *psychic-fangs*, *mist-ball*, *luster-purge*, *origin-pulse*, *precipice-blades*, *dragon-ascent*, *doom-desire*, *psycho-boost*, *attack-order*, *mystical-fire*, *chatter*, *rock-wrecker*, *roar-of-time*, *spacial-rend*, *magma-storm*, *shadow-force*, *seed-flare* y *judgment*.

4.2.3. Los Pokémon

Los pokémon son los luchadores de este juego. Estos tienen distintos atributos que les permiten diferenciarse entre ellos. Todo pokémon tiene **1 o 2 tipos**, su **nivel** y un conjunto de **6 características**, las cuales son: la **vida o puntos de salud (HP) máximos**, el **ataque**, la **defensa**, el **ataque especial**, la **defensa especial** y la **velocidad**.

Como ya se ha podido ver en la sección 4.2.1, los tipos de un pokémon juegan un papel fundamental en el combate, determinando como de efectivos serán los ataques. Pero también esta el nivel, con valores entre 1 y 100, que afecta a valor final del ataque, la defensa, la velocidad y la vida máxima de un pokémon (tal y como se verá en la sección 4.4.1). El nivel es un atributo suficientemente importante como para que una diferencia de unos 20 niveles entre pokémon resulte casi imposible de equilibrar.

Cada pokémon tiene un número limitado de puntos de salud, el cual varia entre los distintos pokémon. Estos se ven reducidos cada vez que el pokémon recibe un ataque, y si llega a cero el pokémon queda debilitado, es decir que a perdido el combate y no puede continuar realizando movimientos.

En cuanto al ataque y al ataque especial, son atributos similares que indican el ataque del pokémon al utilizar movimientos de la clase de daño físico y especial respectivamente. De forma similar pasa con los atributos de defensa y defensa especial, que indican la protección que tiene el pokémon al recibir ataques de la clase de daño físico y especial respectivamente. Por último, la velocidad del pokémon indica como de veloz es al realizar un movimiento, lo que sirve para determina quien atacara primero. Estos atributos se verán más a fondo en la sección 4.4.2.

A parte de estas características, los pokémon pueden realizar **4 movimientos** distintos durante un combate, a pesar de poder aprender muchos más. Esto significa que un mismo pokémon puede tener diferentes movimientos entre combates. Además, si el tipo del movimiento coincide con alguno de los dos tipos de pokémon, se obtiene una bonificación en el daño producido en el enemigo, tal y como veremos en la sección 4.4.4

Hay un total de 482 pokémon diferentes en esta implementación de juego y estos se pueden dividir en 4 generaciones (correspondientes a la versiones reales del videojuego) de tal manera que mantienen más o menos un equilibrio entre ellas en cuanto a tipos y características se refiere. Es decir, que entre generaciones hay más o menos la misma cantidad de pokémon "fuertes" y "débiles" dado que los videojuegos originales se crearon para mantener un equilibrio en los combates a cada nueva generación que introducían. En la primera generación se incluye los 146 primeros pokémon, en la segunda los 97 siguientes, en la tercera los 132 siguientes y en la cuarta los 107 restantes.

Aparte, algunos pokémon son similares entre ellos debido principalmente a que uno es la "evolución" del otro. La evolución es un termino de los juegos original en los cuales un pokémon evoluciona a otro obteniendo un aumento muy significativo de sus características, y cambiando de forma. Dado este hecho, los pokémon se pueden clasificar en evoluciones finales y no finales, donde los primeros tienen mejores características en general que los primero.

Además de esta clasificación, hay 38 pokémon conocidos como legendarios por tener características realmente elevadas. Esto implica que son difíciles de vencer en combates aunque estos sean justos o equilibrados (mismo nivel, tipos con relaciones de efectividad similar ...). Estos pokémon en concreto, tienen un promedio superior al 95 de sus 6 características y son los siguientes: *Articuno*, *Zapdos*, *Moltres*, *Mewtwo* y *Mew* en la primera generación, *Raikou*, *Entei*, *Suicune*, *Lugia*, *Ho-Oh* y *Celebi* en la segunda generación, *Regirock*, *Regice*, *Registeel*, *Latias*, *Latios*, *Kyogre*, *Groudon*, *Rayquaza*, *Jirachi* y *Deoxys* en la tercera generación, y *Uxie*, *Mesprit*, *Azelf*, *Dialga*, *Palkia*, *Heatran*, *Regigigas*, *Giratina*, *Cresselia*, *Manaphy*, *Darkrai*, *Shaymin* y *Arceus* en la cuarta generación.

Además, en cada generación también hay otros pokémon que superan los 95 puntos de promedio de características, los cuales son: *Dragonite* en la primera generación, *Tyranitar* en la segunda generación, *Slaking*, *Salamence* y *Metagross* en la tercera generación, y *Garchomp* en la cuarta generación. Estos últimos no son conocidos como legendarios ya que en el juego original esta categoría implicaba que también eran difíciles de obtener, y estos últimos no lo eran.

4.3. PokéAPI

Con el fin de poder implementar las mecánicas previamente explicadas en la sección 4.2, se ha utilizado la API online Pokéapi [16]. En concreto, se ha accedido a la versión 2, la cual esta en BETA pero contiene más información y esta mejor estructurada. Esta API es de solo consumo y se ha accedido a ella usando el método HTTP GET. Además, esta API no requiere autenticación por lo que el acceso a los datos ha sido muy sencillo.

Los datos que dispone esta API están actualizados a la 7ª generación, y dado que este proyecto pretende realizar una versión de la 4ª generación, esta nos podrá proporcionar toda la información que necesitaremos. Pero también hay que decir que el juego no se comportará exactamente como lo hacía la 4ª generación original, dado que algunas mecánicas han cambiado con el fin de equilibrar el juego. Por ejemplo, la 7ª generación introduce un nuevo tipo, el Hada, cambiando así el tipo de algunos ataques y pokémon de las generaciones anteriores.

La API obtiene sus datos principalmente de la *Pokedex Veekun* y de *Bulbapedia* [19]. *Bulbapedia* es una enciclopedia muy completa que contiene información sobre Pokémon donde cualquier persona puede contribuir, al estilo de Wikipedia pero con contenido exclusivamente relacionado con Pokémon.

4.3.1. Información disponible y la información usada

PokéAPI contiene información muy completa y bien interconectada sobre los videojuegos de Pokémon. Por este motivo, esta contiene mucha información que para la realización de este proyecto no servirá para nada. Por ejemplo, la información sobre las bayas, los encuentros con pokémon, las evoluciones, los juegos, los objetos, los lugares, ciudades y pueblos, habilidades, huevos, naturalezas, ratios de crecimiento, especies... son datos que no se utilizan para nada en el proyecto pero si que están disponibles en esta API.

Para la realización de este proyecto solo es necesario acceder a la información relativa a las tres mecánicas explicadas anteriormente, es decir, los tipos, los movimientos, y los pokémon. Pero como hemos dicho anteriormente, la API es muy grande y de estos datos contiene información innecesaria, así que se debe realizar un filtrado de esta.

En lo referente al **tipo**, solo interesa la relación de efectividad que tiene con los demás tipos. En concreto, solo nos fijaremos en las relaciones de daño de un tipo hacia otra, ya que la relación de daño que le infligen los demás tipos hacia este es información duplicada y fácilmente obtenible con la otra relación. Además otros datos como los pokémon que tiene el tipo o los movimientos de ese tipo no se usarán para el proyecto, dado que esa información estará disponible en los propios movimientos y pokémon.

Por la parte de los **movimientos**, nos interesan los atributos descritos en la subsección 4.2.2, las cuales son: poder/potencia, PP, tipo, precisión, prioridad, clase de daño y ratio de crítico. Los demás datos que la API nos proporciona no nos serán de ayuda para implementar el juego, como descripción y objetivos.

Además, solo se utilizarán aquellos movimientos que pertenecen a la saga original del juego, por lo que la búsqueda de los movimientos se realiza hasta el movimiento número 719, dado que los siguientes ya son de otros tipos de juego. Tampoco se usarán aquellos movimientos que no causen daño, como paralizar, reflejo..., ni aquellos que tengan menos de 5 PP, dado que indican que son un movimiento especialmente poderoso introducido en la última generación que se podía usar bajo circunstancias muy especiales y controladas por un tipo de mecánica que no se ha conservado para esta implementación.

Por último, por la parte del **pokémon** solo nos fijaremos en sus características, sus tipos, los movimientos que puede aprender, y en sus *sprites*, los cuales son las imágenes de los pokémon que se mostrarán en la interfaz gráfica. Así pues, solo se tendrá en cuenta los datos mencionados anteriormente, y no se tendrán en cuenta otros datos que proporciona la API, como peso, altura, formas o habilidades.

Además también se le aplicará un filtro con el fin de evitar a aquellos pokémon que no pueden aprender 4 o más movimientos con el fin de simplificar la implementación del juego. Si se permitiese tener menos de 4 movimientos, las acciones escogidas tanto por de forma aleatoria como las escogidas por un agente deberían tener en cuenta eso. También se ha reducido el la búsqueda de información de los pokémon a la 4ª generación (como se menciona en 4.1) por lo que solo se buscará en los primeros 493.

Respecto a los movimientos que los pokémon pueden aprender, estos deben cumplir las condiciones dichas anteriormente para los movimientos. Además, los movimientos están clasificados según como pueden aprenderse en el juego original, por lo que hay una distinción entre por subir de nivel, por el uso de un objeto, u otros medios. Se ha decidido que se usarán todos los movimientos, ya que solo con los aprendidos por nivel muchos pokémon se quedaban con pocos movimientos.

4.3.2. Sistema de ficheros

Para acabar, hay que mencionar que la propia PokéAPI recomienda que si se va a acceder de forma constante a la API, se debería almacenar la información localmente para evitar fallos por exceso de peticiones. Dado que este proyecto tiene la intención de crear un agente, el cual se entrenará con miles de partidas las cuales necesitan esta información constantemente, se ha decidido guardar la información necesaria de forma local en tres ficheros separados. Estos están en formato JSON el cual permite guarda y cargar objetos del tipo diccionario. Los ficheros creados se han llamado typeDB.json, moveDB.json y pokeDB.json, los cuales guardan la información descrita en esta sección sobre los tipos, los movimientos y los pokémon respectivamente.

En concreto, la información contenida en el fichero typeDB.json tiene como llave el nombre del tipo en inglés, y como valor otro diccionario con llaves *half_damage_to*, *no_damage_to* y *double_damage_to*, cuyos valores son las listas de los nombre de los tipos que el tipo de la llave tiene esa relación de efectividad con ellos. En la siguiente imagen se puede ver la estructura descrita:

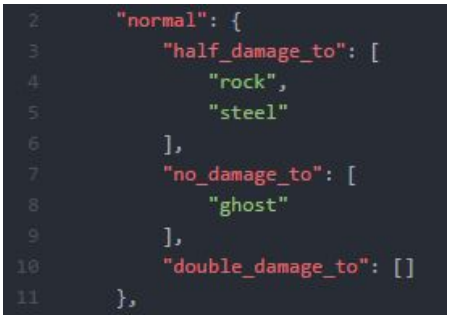

FIGURA 6: Formato de guardado de los tipos

Para el fichero moveDB.json, las llaves son los nombres de los movimientos en inglés, cuyo valor es otro diccionario con las llaves: *power*, *pp*, *type*, *accuracy*, *priority*, *damage_class* y *crit_rate*. Cada una de esta llaves tiene un valor numérico asociado, a excepción de type y damage_class cuyos valores son strings, tal y como se puede apreciar en la siguiente imagen.

FIGURA 7: Formato de guardado de los movimientos

En cuanto al fichero pokeDB.json, este tiene como llave de acceso el nombre del pokémon, y como valor tiene otro diccionario con las llaves *types*, *stats*, *moves* y *sprites*. El objeto correspondiente a la llave *types* es una lista de 1 o 2 string con el nombre de los tipos del pokémon. De forma similar el objeto correspondiente a la llave *moves* es un lista con los distintos nombres de los movimientos que puede aprender ese pokémon. En cuanto al contenido de la llave *sprites*, este es un diccionario con llaves *front* y *back* que contienen el nombre de la imagen correspondiente al *sprite* del pokémon de frente y de espaldas respectivamente.

Por último, la llave *stats* contiene un diccionario cuyas llaves son las 6 características de un pokémon en inglés (*speed*, *special-defense*, *special-attack*, *defense*, *attack* y *hp*). Estas llaves contienen el valor base de la característica forma numérica.

El formato de la información guardada sobre un pokémon se puede ver en la imagen 8 que hay a continuación. Esta imagen a sido recortada entre las lineas 16 y 52 para reducir su tamaño y así poder leer mejor su contenido.

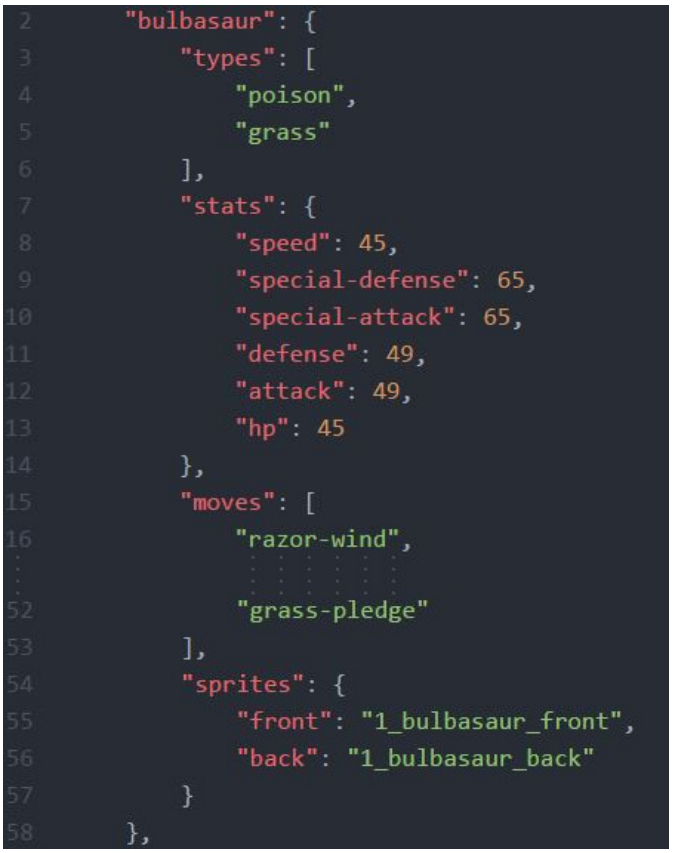

Figura 8: Formato de guardado de los pokémon

4.4. Motor del juego

El motor del juego se encarga de ejecutar la lógica del juego. De esta manera el juego puede ir cambiando de estado mientras se ejecuta el juego. A diferencia de la interfaz gráfica, que se explicará en la sección 4.5, el motor del juego solo realiza los cálculos necesarios para que el juego pueda avanzar.

Esta sección esta dedicada a explicar las funciones básicas del motor del juego, es decir de la ejecución de una partida, lo que incluye un cálculo previo de las características finales de los pokémon, el cálculo de las decisiones de los pokémon y el cálculo del daño provocado.

4.4.1. Características finales de un pokémon

Antes de empezar a explicar el desarrollo de una partida, hay que mencionar que los datos relativos a las características de los pokémon obtenidos a través de la pokéAPI (tal y como se explica en 4.3.1), son los valores base, es decir que son generales y no dependen del nivel del pokémon en concreto.

Con el fin de dar a estos valores más valor dependiendo del nivel del pokémon, a cada característica se le debe aplicar la siguiente fórmula:

$$Característica = \left(\frac{Nivel}{100} \times \left(Stat\_Base \times 2 + IV\right)\right) + 5 \tag{4.1}$$

$$HP = \left(\frac{Nivel}{100} \times \left(Stat\_Base \times 2 + IV\right)\right) + Nivel + 10 \tag{4.2}$$

Además del nivel, también se le añade una componente para dar un valor individual (individual value o IV) a cada pokémon. Esta componente es aleatoria con valores entre 0 y 31, y se ha preparado para que se pueda "apagar" fácilmente, en cuyo caso todos los IV tendrían un valor de 15. De esta manera, se permite eliminar un factor aleatorio y facilitar el aprendizaje del agente en caso de que se considerase oportuno.

Como se puede ver en la fórmula 4.2, hay una fórmula especial para el cálculo de la característica de la vida máxima, la cual solo varía respecto la fórmula general 4.1 en sumarle 5 puntos y el nivel del propio pokémon.

Hay que mencionar que ambas formulas han sido adaptadas a las condiciones del juego implementado, simplificando las formulas utilizadas des de la 3ª generación en los videojuegos originales, las cuales están disponibles en *Bulbapedia* [19].

4.4.2. Desarrollo de una batalla

Como ya se ha mencionado, el juego implementado se basará en las batallas dobles de la saga de videojuegos Pokémon. En estas batallas intervienen 4 pokémon, dos contra dos, en un combate por turnos donde cada pokémon realiza un ataque a uno de sus dos enemigos, y gana aquel bando que haya conseguido derrotar a todos los pokémon de bando contrario. Pero a diferencia de las batallas en los juegos originales, el juego implementado no permite la realización de cambios de pokémon durante el combate ni usar objetos que, por ejemplo, permitan curar las heridas de estos. Esto se debe a que de esta manera se simplifica el motor del juego y las mecánicas que deberá aprender el agente o modelo de RL que se implemente.

Toda batalla empieza con cuatro pokémon con su salud al máximo y con todos sus PP (puntos de poder, ver sección 4.2.2) de sus movimientos al máximo. Entonces se procede a realizar un turno tras otro hasta que se haya terminado la batalla o hasta que se hayan agotado todos los PP de todos los movimientos de todos los pokémon vivos (dado que entonces no se podría acabar nunca la batalla), o hasta haber realizado una cierta cantidad de turnos. Se ha puesto este límite debido a que en algunas ocasiones las partidas jugadas por algunos de los agentes implementados llegaban a un punto muerto en el que se centraban en atacar a un enemigo muerto (por lo tanto no gastan PP) y el enemigo no podía hacer nada por que no le quedaba ningún movimiento con PP.

En cada turno, cada entrenador **decide que movimiento** usará su pokémon y contra quien lo usará. Claro esta, solo los pokémon aun con vida podrán decidir que acción realizarán en el siguiente turno. Esta decisión se realizará dependiendo de que tipo de entrenador tenga el pokémon, los cuales se explicarán en la sección 4.4.3.

Una vez que se conoce que acciones van ha ser tomadas, se **decide el orden** por el cual van ha ser realizadas. El orden de ejecución de los movimientos depende tanto de la prioridad del ataque escogido como en la propia velocidad del pokémon. El factor con más peso es la prioridad del movimiento. Esto significa que solo si dos movimientos tienen la misma prioridad se decidirá quien realiza primero el ataque basándose en la velocidad del pokémon atacante.

Una vez decidido el orden se procede a **atacar** al enemigo escogido con el movimiento elegido, quitándole tanta vida al enemigo como daño haya provocado el ataque. El daño provocado depende de muchos factores, y se explicará con más detalles en la sección 4.4.4. Si se le quita toda la vida a un pokémon este pasa a estar debilitado y no podrá realizar ninguna acción. Por lo tanto aun habiendo decidido una acción, es posible que no se llegué ha realizar dicha acción por que el pokémon que la realizaba se ha debilitado antes de poder realizar su acción.

4.4.3. Entrenadores

Con el fin de poder decidir que acción debe tomar el pokémon en cada turno, este "consulta" a su entrenador. Este entrenador únicamente se ocupa de decidir que movimiento debe usar su pokémon y contra quien lo usará. Así pues, estos entrenadores sirven para poder dar un comportamiento a los pokémon, por lo que estos serán la herramienta que se usará para poder modelar el comportamiento de los pokémon.

Dado que todos los pokémon tiene 4 movimientos, y que hay dos enemigos, la acción se codifica en un espacio de 4×2. Esto significa que por cada turno, cada entrenador debe elegir una acción de entre las 8 posibles acciones. Esta elección, dependerá del tipo de entrenador que tenga el pokémon.

Hay tres tipos distintos de entrenadores. Los entrenadores más sencillos que hay son los **entrenadores aleatorios**, los cuales simplemente eligen al azar un movimiento y un enemigo. También existen los **entrenadores controlados por el modelo de RL**, los cuales decide la acción que realizará su pokémon dependiendo de la predicción de su modelo de RL. El modelo de este entrenador se explicará más a fondo en el capítulo 5 dedicado al agente implementado.

Finalmente existen los **entrenadores controlados por un jugador**, los cuales simplemente deciden la acción a partir de la entrada proporcionada por un jugador. Este entrenador es especial dado que necesita la entrada de datos para tomar su decisión. Además, también proporciona una interfaz gráfica para facilitar la elección al jugador, como se verá en la sección 4.5.

4.4.4. El ataque

Un ataque se produce solo entre dos pokémon, el atacante y el enemigo, y solo se utiliza un movimiento. Estos tres elementos influyen en el cálculo del daño final en gran medida y hay muchos atributos que influyen en su valor final e incluso en si se realizará o no.

Además, también aparece cierto factor aleatorio que puede afectar tanto a la cantidad del daño final como a si se realizará el ataque o no. Este factor aleatorio se ha añadido con el objetivo de poder dificultar un poco el aprendizaje de agente, y por tanto, también se a preparado un mecanismo para poder "apagarlo" en caso de que se dificulte demasiado.

Para empezar, el hecho de que se pueda realizar un ataque o no viene determinado por tres factores: que el enemigo siga vivo, que el movimiento aun se pueda usar (es decir que le quede al menos 1 PP), y finalmente de un factor aleatorio que depende de la propia precisión del movimiento.

Respecto a la precisión hay un detalle a tener en cuenta y es que esta puede tener un valor nulo (None), el cual equivale a que no falla nunca o a ser del 100 %. Este valor se debe a una diferenciación presente en el juego original entre el valor nulo y el 100 % de precisión que no se ha conservado al hacer esta implementación.

Si el ataque se realiza, se consumirá 1 PP del movimiento para poder realizarse, y luego se procederá a disminuir la vida del enemigo en tantos HP como indique el daño calculado. Hace falta mencionar que tanto si el movimiento se realice como si no, el turno se contabiliza igualmente, por lo que utilizar un movimiento sin PP o atacar a un enemigo ya debilitado puede verse como un turno desaprovechado.

Para poder determinar el daño que produce un ataque se utiliza la siguiente fórmula, la cual se ha obtenido de Bulbapedia [19]:

$$Daño = \left(\frac{(0{,}2 \times Nivel + 1) \times Potencia \times Ataque}{25 \times Defensa} + 2 \right) \times Modificadores \tag{4.3}$$

$$Modificadores = Efectividad \times Bonificación \times Crítico \times Variabilidad \tag{4.4}$$

En la fórmula 4.3, podemos observar que se utilizan tanto el nivel del pokémon atacante como la potencia del movimiento. También podemos ver como se utiliza tanto el ataque del pokémon atacante como la defensa del pokémon enemigo. Para estos valores se utilizarán las características de ataque y defensa si el movimiento es de la clase de daño físico, o las características de ataque especial y defensa especial si el movimiento es de la clase de daño especial. Hace falta recordar que estos valores son el resultado de haberles aplicado la fórmula 4.1 como se explica en la sección 4.4.1.

En cuanto a los modificadores, son valores que hacen variar el daño final en gran medida y que se deberían tener en cuenta en la elección de las acciones, especialmente la Efectividad y la Bonificación. Tanto el Crítico como la Variabilidad son valores que dependen del azar y por lo tanto son más difíciles de predecir y tener en cuenta en el momento de la elección del movimiento.

Respecto a la **Efectividad**, esta se calcula utilizando los factores multiplicativos vistos en la tabla 5 de la sección 4.2.1. Dado que los pokémon pueden tener más de un tipo, para el cálculo de la efectividad final del movimiento se realiza la acumulación multiplicativa del daño. Así pues, el valor de la Efectividad solo puede tener 6 valores, los cuales son 0, 0.25, 0.5, 1, 2 y 4.

Esto implica que si un movimiento de tipo Dragón ataca a un pokémon enemigo que tenga los tipos Acero y Dragón, la efectividad de este sería el resultado de multiplicar la efectividad entre Dragón-Acero y Dragón-Dragón, por lo que la efectividad final sería de $0{,}5 \times 2 = 1$.

Otros ejemplos serían que un movimiento de tipo Fuego atacase a un pokémon de tipos Hielo y Planta, dando como efectividad final un valor de $2 \times 2 = 4$, o que un movimiento de tipo Eléctrico atacase a un pokémon de tipos Agua y Tierra, dando como efectividad final un valor de $2 \times 0 = 0$.

Respecto a la **Bonificación**, es un factor que premia el uso de movimientos del mismo tipo que el del pokémon que lo usa, añadiendo un 50 % más del daño. Esto significa que la bonificación tendrá un valor de 1,5 si ambos tipos son el mismo, y 1 si son diferentes.

En cuanto al **Crítico**, este depende del atributo *ratio de crítico* del movimiento utilizado. Tal y como se ha visto en la sección 4.3.2, más concretamente en la figura 7, este atributo se codifica con números enteros comenzando des del cero. Este número representa la probabilidades que hay de que dicho movimiento realice un golpe crítico al atacar al enemigo. Esto significa que el movimiento causará un 50 % más de daño de lo habitual, es decir que el crítico tomará el valor de 1,5 con dicha probabilidad, o 1 en caso contrario.

Más concretamente, para el valor de 0 de *ratio de crítico* hay 1 entre 24 de realizar un golpe crítico, para el valor de 1 hay 1 entre 8, para el valor de 2 es del 50 % y para valores más altos es del 100 %.

Para finalizar, el valor de la **Variabilidad** es un valor aleatorio entre 0,85 y 1. El motivo por el cual se añade este modificador es para dificultar el hecho de predecir el daño final y para asegurarse que dos ataques realizados en las mismas circunstancias tendrán resultados ligeramente distintos. Pero tal y como se ha mencionado anteriormente, todo factor aleatorio puede "apagarse" si en algún momento así se desea, ya sea para facilitarle al modelo el aprendizaje del juego o por motivos de experimentación.

4.5. Interfaz del juego

A pesar de que el proyecto únicamente necesitaría el motor del juego para generar las partidas y el agente que implementase el modelo RL ideado, se ha añadido una interfaz gráfica para cuando es un jugador real el que decide las acciones de un pokémon con el fin de poder visualizar mejor el estado actual del videojuego y facilitarle la toma de decisiones al jugador. Así pues, esta interfaz permite a los jugadores indicar las acciones que hayan decidido de forma fácil y visual, a la vez que permite mostrar el estado actual del juego de una forma más visual y entendible de lo que podría hacerse desde la consola.

La interfaz tiene dos modos de visualización, uno básico el cual visualiza una interfaz similar a la de los videojuegos originales, y otro más complejo el cual visualiza información relativa al aliado para que sea más fácil analizar el comportamiento de este cuando está controlado por un agente.

4.5.1. Elementos básicos del juego

En el modo básico de visualización se muestra exclusivamente la información relativa a el juego que debería ser conocida por el jugador para poder tomar sus decisiones. Esta es la misma información que se muestra en los juegos originales mientras se esta en la pantalla principal del combate.

La interfaz básica se puede dividir en dos partes, una que muestra el combate y la información relacionada con el desarrollo de este, y la otra que muestra la información relativa a los movimientos que se pueden elegir para tomar la decisión más adecuada en cada turno. Además, cuenta con un sistema de interacción integrado para poder recibir la acción decidida por el jugador.

El Combate

En cuanto a la parte que muestra la **información del combate**, esta muestra para cada pokémon que esta en batalla su nombre, su nivel y su barra de vida. Esta también muestra el *sprite* de cada pokémon con vida para que sean fácilmente reconocibles y a la vez saber quien sigue vivo, dado que no se muestran para los pokémon eliminados. En la siguiente imagen se puede observar como la interfaz gráfica representa el inicio de un combate.

Figura 9: Representación de la batalla en la interfaz gráfica

Los dos pokémon que están en la parte superior, los que se ven frontalmente, son los enemigos. Tanto su vida como su nombre y nivel se pueden ver en los rectángulos grises ubicados en la esquina superior izquierda. El rectángulo inferior corresponde al enemigo izquierdo mientras que el rectángulo superior al enemigo derecho.

De forma similar, los pokémon de la parte inferior de la imagen 9, los que se ven de espaldas, son el pokémon del jugador y su aliado. Los rectángulos grises de la esquina inferior derecha representan la vida y el nombre de cada un de ellos. El rectángulo inferior corresponde al aliado derecho mientras que el rectángulo superior al pokémon izquierdo, el cual es el pokémon del jugador.

Hay que mencionar que, a diferencia de los videojuegos originales, la representación de la batalla no muestra ningún tipo de animación, ya sea cuando se mueren los pokémon o cuando se realizan movimientos. Esto se debe a que no aporta ningún tipo de información visual, ni para la evaluación del agente ni para la elección de la acción a realizar.

En la figura 9 también se puede ver que la interfaz gráfica incluye un **cuadro de dialogo** en el que se muestran los sucesos ocurridos en el combate. En este se indican hechos como los ataques realizados (indicando el pokémon que lo ha realizado, el movimiento utilizado y el pokémon que recibe el ataque), la efectividad de los ataques realizados, y los pokémon que se debilitan. Aparte de esta información, el cuadro de dialogo también se encarga de informar al jugador de cuando es su turno de elegir la acción a realizar para ese turno, o de indicar la finalización de la batalla, mostrando que bando a ganado.

Además, la mayoría de la información que se muestra en este cuadro también se muestra en la terminal. Esto se hace porque el cuadro de dialogo muestra la información de forma temporal, y aveces es necesario tener que repasar la información de los eventos sucedidos en una batalla. Esto puede ser de gran utilidad en caso de no haber podido leer la información del cuadro de dialogo por culpa de algún descuido, o simplemente para poder ver el resumen de la batalla realizada. A continuación se puede ver una imagen de como se muestra la información en la terminal.

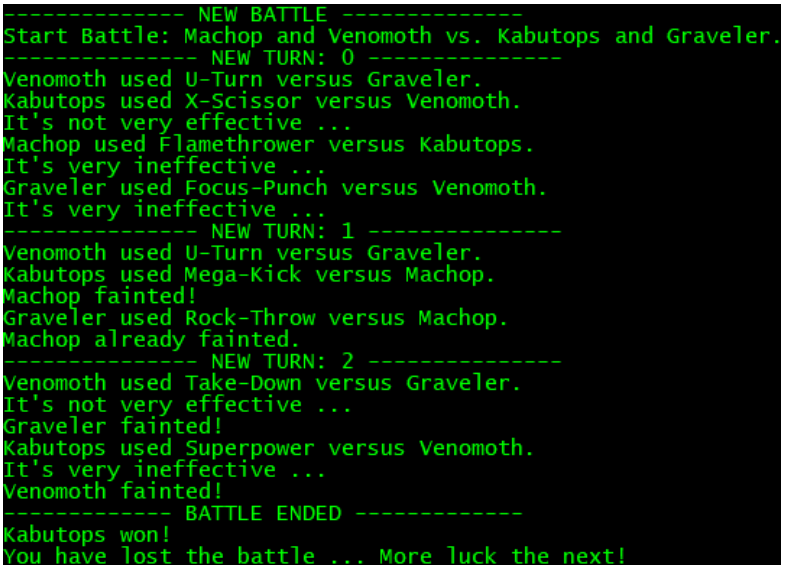

FIGURA 10: Eventos de la batalla mostrados en la consola

La Selección

Por otra parte está la **visualización de los movimientos**. Esta permite visualizar la información más relevante de los movimientos del pokémon del jugador para facilitarle la elección de la acción.

En esta parte de la interfaz se muestran los cuatro movimientos del pokémon mostrando su nombre, la cantidad de PP que le quedan al movimiento y el tipo de este. Además, cada movimiento se encuentra dentro de un recuadro cuyo color depende del tipo del movimiento par así ser fácilmente identificable. Esta manera de mostrar los movimientos, al igual que la cantidad de información mostrada, es muy similar a la forma en que se muestran en los videojuegos originales.

En la siguiente imagen se puede ver como la interfaz creada representa los movimientos del pokémon controlado por el jugador y muestra la información de cada uno de ellos.

FIGURA 11: Interfaz de la selección de movimiento

Una vez que el jugador ya ha decidido que movimiento usar, debe indicar contra que pokémon enemigo lo usará. Para tal fin, la interfaz proporciona una **pantalla de selección de enemigo**. Esta es muy similar a la anterior y permite al jugador decidir contra quien usará el movimiento seleccionado anteriormente. Esta simplemente nuestra el nombre del enemigo en cada botón distribuidos tal y como aparecen en la pantalla de la batalla.

Además incluye un botón de retroceso por si el jugador se arrepiente de la elección de movimiento que hubiese hecho en la pantalla anterior. La imagen 12 que hay a continuación muestra la pantalla de sección del enemigo.

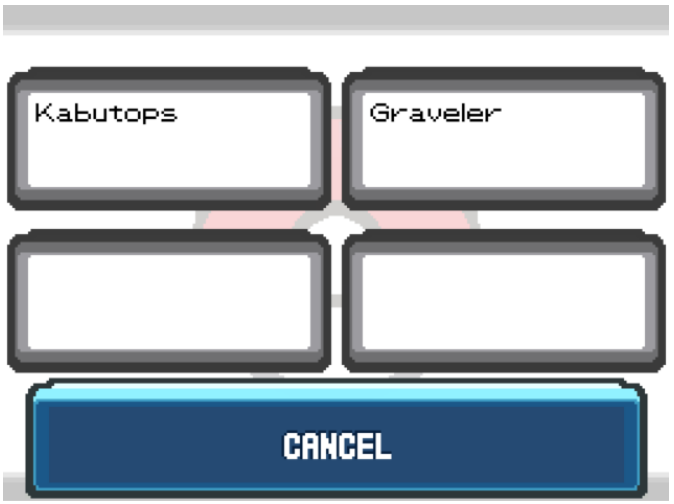

FIGURA 12: Interfaz de la selección del objetivo

La Interacción

La parte de la interfaz de selección sirve también de entrada de información para el juego, así que el jugador pude indicar que movimiento a decidido que debe realizar su pokémon al igual que contra que enemigo quiere usarlo. Esto se hace seleccionando el botón correspondiente con el ratón o con el selector.

En concreto, el selector se desplaza a través de los botones con las fechas del teclado, y se selecciona el botón concreto pulsando la tecla *enter* o *espacio*. Además, el selector recuerda la acción seleccionada del último turno.

Aparte, se pueden pulsar tanto la tecla *enter* o *espacio* como pulsar el botón izquierdo del ratón para poder saltarse el tiempo de espera de los diálogos de la batalla.

4.5.2. Información de apoyo

Debido a que el proyecto tiene como objetivo evaluar las acciones que realiza el agente creado, se han añadido más pantallas de visualización de datos para poder proporcionar más información de la que da la de los juegos originales.

Para tal propósito se han añadido dos pantallas de apoyo, una que muestra más información relativa al combate y otra que proporciona información sobre los movimientos del aliado.

En primer lugar tenemos la **pantalla de información** relevante de los pokémon que hay en combate. Esta se encarga de mostrar para cada pokémon tanto su nombre como su tipo y sus características así como su vida actual y vida máxima en forma numérica.

De esta manera se puede comprobar de forma fácil que enemigo es más fuerte, tiene más vida o cual es la clase de daño que le puede provocar más daño. De igual manera de pueden comprobar estas cosas en los pokémon aliado para saber con que hay que tener cuidado. En la siguiente imagen se muestra como la interfaz gráfica muestra esta información.

FIGURA 13: Visualización información de apoyo de la batalla

Por otro lado también se ha añadido otra pantalla de **visualización de movimientos para el aliado** similar a la ya explicada en la subsección 4.5.1 con algunos cambios.

Con la intención de poder identificar el motivo de las elecciones del agente aliado, a esta pantalla se le ha añadido unos marcadores de colores que indican la efectividad de cada movimiento para ambos enemigos, además de incluir el factor efectividad de estos. De esta forma es más fácil identificar si hay algún movimiento que sea especialmente efectivo contra alguno de los enemigos. Además también se ha modificado la pantalla para poder mostrar tanto la potencia del movimiento como su precisión así como la clase de daño que provoca cada movimiento.

Aparte, dado que puede darse el caso de que el jugador pueda necesitar también de esta ayuda, se ha creado un modo que permite que se visualice la información extra de los movimientos en la pantalla de selección de movimientos de jugador.

El formato de esta nueva pantalla de visualización de movimientos queda tal y como se muestra en la siguiente imagen.

FIGURA 14: Visualización de los movimientos con información extra

Para finalizar, una vez vistas todas las pantallas por separado, en la siguiente imagen se puede ver toda la interfaz gráfica al inicio de una batalla.

FIGURA 15: Interfaz gráfica completa

Capítulo 5

El agente

En este capítulo se expondrán las principales características que tiene el agente desarrollado. Para tal fin, se describirá el diseño general utilizado para crearlo, (sección 5.1), y como este aprende a jugar y decide las acciones más apropiadas en cada turno (sección 5.2).

Además, también se describirán los tres tipo de modelos implementados (5.3) donde se explicará como se ha hecho para que el agente aprenda de las acciones del jugador y aprenda a colaborar con él.

5.1. Diseño

Con el fin de poder controlar la toma de decisiones de un pokémon, tal y como se menciono en el apartado 4.4.3, existen entrenadores capaces de decidir tanto los movimientos como los objetivos de los pokémon. Estos entrenadores se utilizarán para crear al agente que utilice el modelo de RL creado en este proyecto.

El modelo de RL creado en este proyecto se dedicará a predecir cual es la mejor acción que puede realizar el pokémon en cada turno. Para ello tendrá que elegir entre las ocho posibles acciones (cuatro movimientos por dos objetivos). Esto lo hace escogiendo la acción que predice que le aportará más recompensa en ese turno. Por lo tanto, primero se debe predecir la recompensa que le dará el juego en el estado actual al realizar cada acción.

Con el objetivo de predecir dicha recompensa, el agente implementado utiliza un algoritmo de Aprendizaje por Refuerzo (o Reinforcement Learning, RL) adaptado al problema, el cual utiliza redes neuronales (o Neural Network, NN) como apoyo en el proceso. Así pues, el agente solo debe consultar la predicción predicha por el modelo de RL a partir del estado actual del juego, y escoger aquella que tenga la recompensa mayor. De esta manera sabrá que movimiento y objetivo debe hacer el pokémon para ese turno.

5.1.1. Aprendizaje por Refuerzo

El algoritmo de aprendizaje por refuerzo es un tipo de algoritmo de aprendizaje automático que tiene como objetivo encontrar la política de selección de la acción que maximice la recompensa a largo plazo utilizando el estado actual del entorno (en nuestro caso el juego) y un sistema de recompensas que proporciona el entorno.

La **función de recompensa** se ha decidido que sea simplemente el daño provocado, dado que es lo que se pretende maximizar. Además, este es nulo cuando se hacen acciones no deseadas como atacar a un enemigo muerto, utilizar un movimiento sin PP o atacar con un movimiento a un pokémon cuya relación de eficacia de los tipos es nula. También se reduce cuando la relación entre tipos es poco efectiva y se aumenta cuando es muy efectiva, cosa que es deseable si se pretende que el agente aprenda las relaciones de tipo. Así pues, con esta función de recompensa el modelo debería ser capaz de aprender fácilmente a evitar estas acciones.

En concreto, para este proyecto se utilizará el algoritmo de RL llamado Q-learning, el cual ya fue explicado en la sección 1.2.2. A modo de resumen, este algoritmo se basa en predecir los valores *Q(a,s)* los cuales representan la recompensa obtenida en el estado s al utilizar la la acción a.

La manera en que este algoritmo predice los valores Q es bastante sencilla. Básicamente se utiliza una tabla de tamaño de número de estados posibles por número de acciones posibles, y con cada par de s y a obtenidos de las muestras de ejemplo, se actualizan los valores de estos en la tabla. La forma en que se actualizan estos valores se verá en más detalle en la sección 5.2.1.

Como ya se ha mencionado anteriormente, la versión de este algoritmo utilizada para este proyecto utiliza redes neuronales como apoyo en su algoritmo. De hecho, la red neuronal sustituye a la tabla de valores Q debido a que el elevado tamaño del conjunto de estados posibles hace imposible manejarla. De esta forma se pretende utilizar un algoritmo similar al Deep-Q-Networs (DQN) desarrollado por Google DeepMind explicado en la sección 1.2.3.

Así pues, en vez de tener que consultar una tabla para poder realizar una predicción, solo se debe consultar a la red neuronal. Esta ofrece ciertas ventajas como que puede aprender ciertas regalas de juego para predecir valores no observados anteriormente.

5.1.2. Red Neuronal

Las redes neuronales son una herramienta muy útil tanto para la clasificación como para la predicción de valores numéricos. Gracias a esto último, se usará la red neuronal para predecir la recompensa de cada acción a partir del estado del juego. Es decir, se sustituirá la tabla de valores Q del algoritmo de aprendizaje por refuerzo por una red neuronal con el fin de que la red sea la que predice la recompensa y la que aprenda las mecánicas del juego al extrapolar la información que se le proporciona.

De esta manera la tabla de valores Q quedaría englobada en la red neuronal, pero teniendo esta una forma de aprendizaje más flexible. Así pues, el principal objetivo de la red neuronal utilizada en este proyecto será encontrar la función que dado un estado del juego, deduzca la recompensa para cada acción del pokémon.

En concreto, la red neuronal utilizada se ha obtenido gracias a la librería Keras, la cual se ejecuta por encima de TensorFlow, el *framework* de código abierto para el aprendizaje automático de Google [15] [20].

Las redes neuronales intentan imitar en cierta manera el funcionamiento del celebro humano, por tanto estas se pueden ver como un conjunto de capas de neuronas con conexiones entre las capas. Así pues, las redes neuronales se pueden definir por el número de capas, las cantidad de neuronas por capa y la función de activación de cada capa. Para este proyecto se ha implementado una red neuronal cuyas capas son densas, es decir que tienen conexión con todas la neuronas de la siguiente capa.

Además se a preparado la red para poder ser configurada a voluntad con el fin de poder personalizar la red y así adaptarla al problema. En concreto se ha preparado la red para poder dar la flexibilidad de poder elegir tanto el número de capas como el número de neuronas de estas, así como el tipo de función de activación de estas. A pesar de esto, es necesario cumplir ciertos requisito.

El primero de ellos es que la primera capa de neuronas tenga la misma cantidad de neuronas que de valores de entrada, los cuales son la codificación del estado, tal y como se explicará en la sección 5.1.3. También es necesario que la última capa contenga la misma cantidad de neuronas que de acciones que se quieran predecir, en este caso ocho. Además también hay que procurar que esta capa dé una salida directa del valor predicho por lo que se ha decidido fijar la función de activación de la última capa a una función de activación lineal.

Por último, se ha decidido que se utilizará el método *Adam* (Adaptive Moment Estimation) como optimizador de los valores de la red, utilizando como métrica la precisión, y como función para cuantificar el error cometido se utiliza el promedio del cuadrado de la diferencia (*mse*).

5.1.3. Codificación del estado

El estado actual del juego es la representación del juego en el turno actual, la cual esta formada por distintas variables numéricas que caracterizan al estado del juego, como por ejemplo los pokémon en batalla o la vida de estos. Como ya se ha mencionado, la red neuronal recibe como entrada el estado actual del juego, por lo que se tendrá que adaptar el estado en la forma apropiada y con las variable adecuadas para que se puedan introducir los datos en la red. Para tal fin el estado del juego debe ser codificado con valores numéricos normalizados.

Que información del estado actual hay que pasarle a la red para conseguir el mejor resultado es algo complicado de saber a priori. Una información básica que se debería proporcionar es que pokémon enemigos están vivos y cuales no, así como que movimientos puede usar el pokémon que controlamos y cuales no. Esto se debe a que así podrá aprender cuando debe de evitar usar un movimiento al cual ya no le quedan PP o evitar atacar a los pokémon ya debilitados.

También se le debería de proporcionar los tipos de los pokémon enemigos como los del pokémon controlado así como los de cada movimiento del propio pokémon. Esto se debe al importante papel que desarrollan las relaciones entre tipos en el combate. Los tipos, al ser una palabra, deberán de ser codificados en forma de *one-hot* para poder ser variable de entrada de la red.

Esta codificación simplemente genera un vector de números del tamaño igual a la cantidad de tipos diferentes y coloca un 1 en la posición correspondiente al tipo a codificar mientras que en las demás posiciones coloca 0.

Aparte de las variables mencionadas anteriormente, otras como la potencia del movimiento, las características de cada pokémon o la vida restante de estos podrían ser de gran utilidad para que la red funcione apropiadamente. Pero las variable a utilizar para la codificación del estado se decidirán después de la realización de algunas pruebas experimentales, las cuales se detallarán en la sección 6.1.

5.2. Funcionamiento

Una vez introducido el diseño del modelo RL, se puede explicar con mayor facilidad el funcionamiento de este tanto a la hora de aprender como a la hora de tomar decisiones.

5.2.1. Aprendizaje

El aprendizaje del modelo RL se realiza a través de sesiones de entrenamiento. En estas sesiones se obtienen los datos de una cantidad elevada de batallas (por ejemplo 10.000) las cuales se juegan de **forma aleatoria** con el fin de sacar el máximo provecho de la exploración realizada. De estos datos se obtienen para cada turno el estado actual del juego codificado, el estado siguiente del juego codificado y la recompensa obtenida en ese turno.

Una vez que se dispone de estos datos, se calcula para cada turno el valor Q asociado al estado s y a la acción a elegida en ese turno, tal y como indica la siguiente fórmula:

$$Q(s,a) = Q(s,a) + \alpha\big[r - Q(s,a) + \gamma max(Q(s',a'))\big] \tag{5.1}$$

Donde α es el ratio de aprendizaje (o *learning rate*) el cual determina en que medida la nueva estimación remplazará a la antigua, γ es el ratio de descuento o (*discounting rate*) el cual pondera la importancia de las recompensas futuras, r es la recompensa obtenida en ese turno y $max(Q(s',a'))$ es la recompensa máxima predicha para el siguiente estado. Tanto la α como la γ son parámetros los cuales necesitan de experimentación para poder encontrar su valores óptimos, pal y como se verá en la sección 6.2.2.

Para funcionar correctamente, esta fórmula necesita de varias iteraciones con el fin de poder converger en la recompensa esperada final. Por este motivo se ha creado un nuevo parámetro, llamado *epochs_RL_fit*, que decidirá cuantas veces se repetirá el proceso de actualización de los valores $Q(s,a)$.

Para utilizar la fórmula 5.1 descrita anteriormente se necesita tanto leer como modificar el valor de *Q(s,a)*. La estructura encargada de manejar esta información podría ser una simple tabla en la que la escritura y la lectura del valor se realizarían a través de la indexación de *s* y *a*. Sin embargo, como ya ha sido mencionado anteriormente en la sección 5.1.1, en este proyecto no se utilizan tablas sino redes neuronales, por lo que el acceso a los datos se complica un poco.

Con el fin de **obtener el valor *Q(s,a)***, es necesario realizar una predicción del estado *s*, y obtener la recompensa correspondiente a la acción *a*. El hecho de que sea una predicción y no el valor real previamente calculado provoca que haya cierta variación en la recompensa, pero esta es muy leve debido a la gran precisión de la red neuronal.

Para poder **modificar el valor *Q(s,a)*** que hay en la red neuronal, es necesario entrenarla, llamando a la función *fit* que la propia librería de Keras proporciona. Esta función solo necesita la codificación de los estados y los valores *Q(s,a)* previamente calculados para entrenar a la red. Además, permite definir ciertos parámetros como el número de repeticiones del entrenamiento (*epochs*), el tamaño de datos por actualización de valores (*batch_size*) y la proporción de datos dedicada a la validación (*validation_split*), de los cuales solo *epochs* tiene una influencia significativa en el resultado del entrenamiento.

Así pues, una vez realizado este proceso, la red neuronal habrá actualizado sus pesos y habrá ajustado estos a los datos proporcionados, siguiendo el proceso de actualización dictado por el algoritmo de aprendizaje por refuerzo.

Hay que tener en cuenta que este proceso tiene un elevado coste temporal, dada la gran cantidad de datos que necesita para entrenarse, los cálculos que se realizan para cada dato, y las iteraciones que se realizan sobre estos. Con el objetivo de reducir ese tiempo, se ha conseguido que el proceso de entrenamiento de la red se realice en la **GPU**.

5.2.2. Toma de decisiones

Una vez que el modelo ha sido entrenado, para saber que decisión debe tomar el pokémon es tan sencillo como preguntárselo a la red neuronal, la cual, como ya se ha dicho, es la encargada de predecir los valores *Q(s,a)*, y por tanto de predecir las recompensas que generará cada acción.

Primeramente hay codificar el estado actual del juego para que la red neuronal pueda predecir las recompensas para cada acción. Una vez conocida las recompensas para cada acción, y dado que el objetivo del juego es conseguir la máxima recompensa, solo hay que elegir aquella acción con mayor recompensa predicha. De esta forma, cuanto mejor sea la red neuronal prediciendo las recompensas de cada movimiento mejor actuará el pokémon en consecuencia.

5.3. Tipos de Modelo

Tal y como se decía en la sección 1.3.2, el proyecto pretende crear una inteligencia artificial capaz de aprender de un jugador real y cooperar con él. Para tal fin se han desarrollado tres tipos de modelos de RL: uno básico que aprende a jugar solo, uno que aprende a jugar observando al jugador, y otro que aprende a cooperar con el jugador.

Los tres modelos tienen un diseño y funcionamiento similar, siguiendo lo descrito en las secciones 5.1 y 5.2. Pero los modelos que tienen en cuenta al jugador tienen pequeñas variaciones para incluirlo en sus algoritmos.

5.3.1. Modelo Básico

Por la parte del modelo básico, el cual se a llamado ***BasicModel***, no hay mucho que comentar debido a que es la implementación tal cual se explica en las secciones 5.1 y 5.2.

Solo cabe destacar que durante su entrenamiento este modelo puede ser aplicado a todos los pokémon que están en la batalla, dado que no depende de ningún compañero. Esto provoca que con la ejecución de un solo turno se puedan obtener 4 muestras diferentes, una por cada punto de vista, por lo que este modelo puede tener una aceleración en su proceso de aprendizaje. De esta forma se reduce el tiempo de entrenamiento al tener que generar menos batallas para obtener el mismo número de muestras.

Además, este modelo se utilizará para encontrar lo valores óptimos durante la mayor parte de la experimentación descrita en la sección 6. Esto se debe a la posibilidad que se acaba de mencionar de poder reducir el tiempo del aprendizaje.

5.3.2. Como aprende del jugador

Con el objetivo de que el algoritmo sea capaz de aprender del jugador, este debe fijarse en las acciones que toma el jugador en cada turno. Por tanto, se ha modificado el algoritmo básico para que en vez de procesar solamente la información proveniente del pokémon aliado, también procese la proveniente del jugador. Al resultado de este algoritmo se le ha llamado ***LearnerModel***.

Además, dado que se supone que el jugador ya conoce las mecánicas del juego y que sabe jugarlo bien, se a creado un nuevo parámetro que hace la misma función que la variable alfa de la fórmula 5.1, el ratio de aprendizaje, para ser usado solo en los datos provenientes del jugador en vez de utilizar la antigua. Esta nueva variable se ha llamado *learning rate player*, y debría de ser igual o superior a *learning rate* con el objetivo darle más importancia a los datos obtenidos del jugador.

Con el fin de entrenar este modelo, es necesario que el aliado de este ya sepa jugar al juego, y que durante el entrenamiento tome las acciones basadas en su conocimiento. Por lo tanto, durante el entrenamiento de un modelo *LearnerModel* es necesario que su aliado sea un agente previamente entrenado, como por ejemplo un *BasicModel*. No se entrenará con un jugador humano dado que para aprender necesitaría muchas batallas y dado el tiempo limitado del proyecto no sería viable.

Además, de forma similar a lo que pasa con el *BasicModel*, el tiempo de entrenamiento de este modelo se puede reducir debido a que se puede utilizar la pareja de modelos tanto en el bando aliado como en el bando enemigo. Aun así, la reducción del tiempo no sería tan grande como en el anterior modelo.

5.3.3. Como aprende a cooperar

Al igual que en la sección anterior, para hacer que el algoritmo tenga en cuenta las decisiones del jugador e intente colaborar con él, es necesario modificar el algoritmo anterior para que a demás de fijarse en las acciones que realiza el jugador, también intente aprender que acción debería realizar para obtener un mejor recompensa en función de la acción tomada por el jugador. Así pues, como resultado de las modificaciones que se han llevado a cabo, se ha creado un nuevo modelo el cual se ha llamado ***CoopModel***.

Con el fin de poder realizar esta modificación, se han realizado cambios tanto en la forma de codificación del estado como la estructura de la red neuronal, así como en la forma de decidir la acción ha realizar y la función de recompensa.

Por la parte de la **codificación del estado**, este ha tenido que ser modificado para poder incluir más información del aliado. En concreto, se le añadirá la codificación que se crea oportuna tanto para el pokémon del jugador como para cada uno de sus movimientos dependiendo de los resultados obtenidos durante la experimentación. De esta forma la red dispondrá de la información necesaria para prever el resultado de la combinación de ambos ataques.

Debido a el cambio de la codificación del estado, también se ha modificado **la propia estructura de la red neuronal** para poder tratar las nuevas variables de entrada. Para hacerlo simplemente se han añadido más neuronas a la primera capa de la red. Además, también se ha modificado la salida de la red neuronal para poder obtener las predicciones según la acción tomada por el jugador. Así pues, de tener una salida de 8 neuronas, una por acción que puede realizar un pokémon, ha pasado ha tener $8 \times 8 = 64$ neuronas. Esto se debe a que se quiere predecir la recompensa obtenida por cada acción que puede realizar el pokémon dependiendo de la acción que haya tomado el pokémon del jugador.

En cuanto a la **forma de decidir** la acción más acertada, se ha debido de alterar ligeramente para poder tener en cuenta la acción del jugador. Así pues, para este modelo primeramente se escogen las recompensas predichas para las acciones en las que la acción escogida por el jugador se llevaba a cabo, y luego se elige la mejor de esta.

Por último, la **función de recompensa** a tenido que ser modificada para incluir la recompensa del jugador. Así pues, en este modelo la recompensa de cada acción se calcula sumando las recompensas individuales de las acciones del aliado y del jugador. De esta manera se puede combinar la recompensa de ambos, obteniendo como resultado una función de recompensa que devuelve el daño combinado. Esta función debería de permitir aprender muy fácilmente cuando se debe atacar a un mismo enemigo o cuando no, debido a que es probable que alguno de los ataques realizados elimine al enemigo, y el siguiente falle en consecuencia, obteniendo menos recompensa en esos casos.

En cuanto al modo de entrenamiento de este modelo, es idéntico al modelo anterior, el *LearnerModel*, debido a que debe interactuar con un modelo con experiencia para poder aprender de él.

Capítulo 6

Experimentos

Este capítulo se detallarán los distintos experimentos realizados para encontrar la mejor codificación del estado (sección 6.1) y los realizados al agente para encontrar los valores óptimos de los principales parámetros ajustables (sección 6.2). Además se realizara una evaluación del funcionamiento y de los resultados de los modelos creados con el fin de sacar las conclusiones del proyecto (sección 6.3).

6.1. Variables del estado

Tal y como se ha mencionado en la sección 5.1.3, averiguar que variables deben formar parte de la codificación del estado no es una tarea fácil dado que la inclusión de variables que no aporten información puede causar que sus predicciones empeoren. Así pues, se realizarán una serie de experimentos con el fin de encontrar que conjunto de variables es el más indicado.

6.1.1. Procedimiento

Para el conjunto de experimentos explicados en esta sección, se realizará un entrenamiento de **1.000 batalla**, cuyos pokémon participantes serán decididos de forma aleatoria pero con la misma semilla en cada entrenamiento, de esta forma. Su evaluación se realizará de forma similar, solo que con otra semilla diferente a la del entrenamiento pero común para cada evaluación y con el doble de batallas, es decir **2.000**. Como detalle cabe mencionar que el promedio de turnos por batalla durante la evaluación de los modelos es de unos 1,7 turnos por batalla, así que en cada entrenamiento se podría decir que se entrena al modelo con unas 1.700 muestras.

También hay que mencionar que tanto en el entrenamiento como en la evaluación, se han generado los pokémon para que tengan una variación de ± 2 niveles teniendo como base el nivel 50. Así pues los pokémon generados tendrán niveles entre 48 y 52, los cuales se mantendrán constantes entre ejecuciones debido a la semilla de la aleatoriedad común. De esta manera el algoritmo se verá forzado a tener en cuenta el nivel de cada pokémon en sus cálculos.

El motivo por el cual se realizará con la misma semilla es que de esta manera se puede repetir el tipo de combate con los mismos pokémon, y por lo tanto la cantidad de combates "desequilibrados" (debido a que se hayan generados pokémon más fuertes que otros) será la misma en cada ejecución. De esta manera se podrán comparar los resultados de una manera más precisa sin tener tantas variables aleatorias que puedan alterar el resultado.

A pesar de esto, se seguirá conservando la aleatoriedad del propio juego (en la probabilidad de critico, las variaciones del ataque, la posibilidad de fallar el ataque) para que el modelo intente adelantarse a la aleatoriedad utilizando alguna de las variables del estado, como por ejemplo la precisión de los movimientos, y para que tenga las mismas condiciones en las que se jugaría en una partida real.

El modelo utilizado será el **BasicModel** dado que de esta manera se podrá acelerar un poco el aprendizaje del modelo, utilizando cuatro modelos durante el entrenamiento como se explica en la sección 5.3.1. Además de que para la decisión de qué variables utilizar en la codificación del estado no importa si el modelo aprende del jugador o no. Pero al contrario que en el entrenamiento, durante la evaluación del modelo se realizarán combates donde los aliados serán controlados por dos entrenadores que utilicen el modelo ha evaluar, mientras que los dos rivales estarán controlados por un entrenador cuyas acciones son aleatorias.

Además, el modelo se entrenará utilizando solamente pokémon de la **primera generación**, mientras que será evaluado con los pokémon de las **generaciones 2, 3 y 4** para poder comprobar que realmente a aprendido algo, es decir que ha generalizando el comportamiento que ha observado.

Por último, hay que mencionar que el modelo se entrenará 10 veces durante la ejecución completa de su entrenamiento. En este caso significa que cada 100 batallas el modelo será entrenado tal y como se explica en la sección 5.2.1. Esto se debe a que de esta manera el modelo puede reajustar sus pesos más seguido.

Los demás parámetros utilizados son: una red neuronal con una capa oculta de 20 neuronas y función de activación *relu*, un entrenamiento realizado con *epochs* = 10, *batch_size* = 32, *validation_split* = 0,2 y *epochs_RL_fit* = 10, y unos valores de $\gamma = 0{,}2$ y $\alpha = 0{,}75$ para la fórmula del RL.

Con el fin de poder compararlos, se tendrá en cuenta el porcentaje de victorias del modelo. Además, dado el factor aleatorio del juego, se tendrá en cuenta el promedio de unas cuantas evaluaciones. En concreto se realizarán 2 evaluaciones debido al tiempo que tarda en realizarse cada una (aproximadamente unos 20 minutos).

6.1.2. Experimentación

Primeramente comenzaremos codificando las variable que, tal y como se ha comentado en la sección 5.1.3, se consideran imprescindibles para tener un modelo de **referencia**. Para codificar a un pokémon, estas son sus **tipos** y si están **debilitados o no**. Por la parte de los movimientos, se codificarán los **tipos** de cada movimiento y si se pueden **utilizar o no**. Así pues, se codificarán 3 pokémon, los dos enemigos más el propio pokémon (dado que en la función de recompensa el pokémon del jugador no interviene), y los 4 movimientos del propio pokémon.

Se realizaran distintos experimento con el fin de que la red neuronal aprenda distintas partes del juego. Esto se realizará añadiendo a la codificación del estado básico variables relacionadas con el daño provocado, la vida de los enemigos, el orden de realización de los ataques y variables que son capaces de determinar parte del azar del juego.

Variables de daño

En el primer experimento se probará de añadir tanto la **potencia** de los movimientos como el **nivel** de los pokémon, así como el **ataque** y la **defensa** de estos, tanto la física como la especial, más la **clases de daño** del movimiento. De esta manera, en el primer experimento se le estará proporcionado a la red neuronal todos los datos que necesita para realizar el cálculo general del daño tal y como vimos en la fórmula 4.3. Hay que mencionar que tanto la potencia como el nivel, el ataque y la defensa, al ser valores numéricos, se probará de normalizarlos para que la red neuronal pueda tratarlos de forma equitativa.

En la siguiente tabla se pueden observar los experimentos realizados más relevantes sobre las variables del daño.

Experimento	**Tamaño del estado**	**Porcentaje de victorias**
Referencia (Básico)	187	58,23 %
Sin normalizado	214	59,63 %
Todo normalizado	214	59,87 %
Sin normalizar la potencia	214	65,71 %
Sin potencia	210	56,83 %
Solo potencia no norm.	191	66,75 %
Combinación At, Def norm	208	**67,93 %**
Combinación At, Def (Local Norm)	208	67.75 %
Combinación At, Def sin nivel	205	64,83 %

Cuadro 16: Resultados para las variables de daño

Como aclaración, el experimento *Combinación At, Def norm* solo añade las dos clases de defensas a los enemigos mientras que para el aliado solo añade las dos clases de ataque. Esto se hace dado que los valores del ataque de los enemigos no aparecen en la fórmula de daño que vimos en la ecuación 4.3, y de igual manera pasa con los valores de la defensa del aliado.

Además, en el experimento *Combinación At, Def (Local Norm)* la normalización de los valores de ataque y defensa se realizaron de forma local, es decir teniendo como máximo el máximo de la batalla y no el máximo global. Como se puede observar, este es ligeramente peor, y sumado al hecho de que tarda más en calcular el máximo, esta opción se ha descartado.

Como podemos ver en la tabla 16, el hecho de normalizar los valores no tiene un gran impacto en los resultados, pero la normalización de la variable potencia hace perder mucha información, dado que al no normalizarla se ganan casi un 6 % más de batallas. Esto se puede deber a que esta variable tiene mucha influencia en la fórmula del daño, y esta íntimamente ligada con el movimiento a elegir. Por otro lado se ha comprobado que esta variable contribuye mucho a la manera en que trabaja la red, ya que sin ella la red no sabe que hacer con las demás variables y empeora sus resultados respecto a la referencia en casi un 2 %.

Por otra parte, se ha podido observar que las características de ataque y defensa aportan menos información si se utilizan para todos los pokémon que si se utiliza la defensa para los enemigos y el ataque para el aliado. Esto se debe seguramente a que la red no sabría que hacer con el ataque de los enemigos ni la defensa del aliado en sus cálculos para predecir al recompensa debido a que estos no influyen en el dako causado. Por último, se ha podido observar que el nivel del pokémon ayuda en un 3 % debido a que este factor si se usa en la fórmula del daño.

Así pues se decide que la codificación del daño, realizada como en el experimento *Combinación At, Def norm* será la utilizzada para las variabel de daño, y además se usará como nueva referencia.

Variables de vida

Para el segundo experimento se le ha añadido más detalles de la vida de los pokémon enemigos. En concreto se le ha proporcionado el porcentaje de **vida restante**, y la **vida máxima** de estos normalizada. De esta forma se pretende que el modelo aprenda que enemigos tienen la suficientemente poca vida como para ser eliminados de un solo golpe.

En la siguiente tabla se pueden observar los experimentos más relevantes realizados sobre las variables de la vida.

Experimento	Tamaño del estado	Porcentaje de victorias
Referencia (Mejor Var. Daño)	208	67,93 %
Porcentaje y vida máx. no norm.	212	64,38 %
Porcentaje y vida máx. norm.	212	65,65 %
Solo porcentaje	210	65,65 %
Solo vida máx. norm.	210	63,23 %
Referencia (Básico)	187	58,23 %
Básico + porcen. y vida norm.	191	58,70 %
Básico + Porcentaje	189	**59,28 %**

Cuadro 17: Resultados para las variables de vida

Como podemos ver en la tabla 17, es evidente que añadir variables de la vida no es útil para la red neuronal. También podemos ver como la presencia de la vida máxima normalizada parece no tener efecto en el resultado de las batallas ganadas.

Dado que no esta claro cual es la mejor representación de las variables de la vida, se han realizado dos experimentos extras utilizando las variables básicas y el porcentaje de vida y la vida máxima normalizada, y otro con las variables básicas más el porcentaje de vida.

Estos experimentos, los dos últimos de la tabla 17, han mostrado que es ligeramente mejor utilizar solo el porcentaje de vida como la representación de la variable de vida. Como observación, hay que mencionar que sin utilizar las variables de daño la variable de vida si contribuyen a mejorar el aprendizaje de la red neuronal, pero solo un 1 %.

Variables de orden

Para el tercer experimento se le ha proporcionado a la red información relativa al orden de ejecución de los movimientos. En concreto se le ha añadido la **velocidad** normalizada de cada pokémon y la **prioridad** de ataque de cada movimiento de forma normalizada. De esta forma se pretende que el modelo intente conocer que enemigo podrá atacarle antes de que el pueda realizar algún movimiento.

En la tabla 18 se pueden observar los experimentos realizados sobre las variables del orden más relevantes.

Experimento	**Tamaño del estado**	**Porcentaje de victorias**
Referencia (Mejor Var. Daño)	208	67,93 %
Sin normalizar	215	67,33 %
Todo normalizado	215	66,94 %
Solo velocidad norm.	211	67,75 %
Solo velocidad no norm.	211	64,75 %
Solo prioridad norm.	212	66,78 %
Solo prioridad no norm.	212	67,33 %
Vel. norm. y prioridad no norm.	215	65,95 %
Referencia (Básico)	187	58,23 %
Básico + velocidad norm.	190	**60,48 %**
Básico + prioridad no norm.	191	59,53 %

Cuadro 18: Resultados para las variables del orden

Como podemos ver en la tabla anterior, las variables de velocidad no ayudan a la red, pero tampoco acaban de perjudicarla de una forma significativa. Respecto a la normalización de las variables hay que mencionar que aunque la velocidad normalizada y la prioridad sin normalizar por separado son las mejores opciones de la forma de normalizar, juntas empeoran los resultados obtenidos al normalizarlo todo y los de no normalizar nada.

Dado que tanto la velocidad normalizada como la prioridad sin normalizar obtienen resultados muy similares, se han realizado los dos últimos experimentos de la tabla 18 para decidir cual es la mejor representación del orden. Un vez realizados, podemos observar que utilizar la velocidad normalizada es mejor, por lo que será la variable representativa del orden.

Variables de azar

Para el último experimento se pretende que la red neuronal aprenda a predecir el factor aleatoria del juego que viene provocado por ciertas variable. Así pues se le proporcionará la **precisión** de cada pokémon y el **factor de crítico** de cada movimiento, ambos en su formato de porcentaje de probabilidad.

En la tabla 19 se pueden observar los experimentos realizados sobre las variables del azar más relevantes.

Experimento	Tamaño del estado	Porcentaje de victorias
Referencia (Mejor Var. Daño)	208	67,93 %
Precisión y crítico	216	67,30 %
Solo precisión	212	64,30 %
Solo crítico	212	64,65 %
Referencia (Básico)	187	58,23 %
Básico + precisión y crítico	195	**59,25 %**

Cuadro 19: Resultados para las variables de azar

Como se puede ver en la tabla 19, la incorporación de la precisión y de la probabilidad de critico en la codificación del estado hace que se reduzca un poco los resultados obtenidos. También se puede ver que ambas variables son necesarias para no empeorar drásticamente los resultados obtenidos. A pesar de esto, tal y como se puede apreciar en la última prueba realizada, ambas variables permiten mejorar un 1 % respecto al la referencia básica. Así pues se cogerán ambas variables como representativas del azar.

6.1.3. Resultado

En la siguiente tabla podemos observar los resultados de cada experimento, así como el tamaño del estado codificado.

Experimento	Tamaño del estado	Porcentaje de victorias
Referencia (Básico)	187	58,23 %
Básico + Var. Daño	208	67,93 %
Básico + Var. Vida	189	59,28 %
Básico + Var. Orden	190	60,48 %
Básico + Var. Azar	195	59,25 %

Cuadro 20: Resultados de la experimentación del estado

Donde para las variables de daño se ha decidido que serán representadas por la potencia sin normalizar y la clase de daño de cada movimiento, el nivel normalizado de los pokémon enemigos y del propio pokémon, así como el ataque físico y el ataque especial del propio pokémon normalizados y la defensa física y la defensa especial de los pokémon enemigos normalizadas.

Por parte de las variables del la vida, se ha comprobado que es mejor que esta este representado por el porcentaje de vida de cada enemigo únicamente.

Algo similar pasa con las variables del orden, las cuales es mejor que solo estén representadas por la velocidad normalizada de los pokémon enemigo y del propio pokémon. Por último, se ha podido verificar que tanto la precisión como la probabilidad de crítico son buenas representantes de las variables de azar.

A pesar de que por separado estas variable aumentaban las partidas ganadas con respecto a las variables básicas, la combinación de estas no aporta mejores resultados que la utilización de las variables de daño y las básicas. Tal y como se ha comentado antes, esto es debido a que la red neuronal es capaz de aproximar muy bien la función de recompensa la cual básicamente esta formada por el daño realizado. Además, la red es muy susceptible a variables que no aportan información para predecir la función deseada, y por consiguiente normalmente empeora al añadir otro tipo de variables.

Así pues, se ha decidido que la codificación del estado este formada por las **variables básicas y las variables de daño**. Por lo cual el estado codificado tendrá **208 variables**, las cuales serán la entrada de la red neuronal. A modo de ejemplo, la siguiente imagen muestra la codificación de un movimiento y la de un pokémon:

```
Codificación de un movimiento:
[1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 1, 40, 1, 0]

Codificación de un pokémon:
[0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0,
 0, 0.5, 0.21304347826086956, 0.2826086956521739]
```

FIGURA 16: Ejemplo de codificación de un movimiento y de un pokémon

Hay que tener en cuenta un detalle importante el cual no se ha comentado antes, y es que el modelo usado de referencia, con solo las variables básicas, ya supera el 50 % de batallas ganadas, lo que significa que el modelo creado ya es mejor que el mero azar antes incluso de acabar de optimizar los parámetro.

6.2. Parámetros del modelo RL

Tal y como se ha mencionado a lo largo de las secciones 5.1 y 5.2, el modelo RL implementado tiene muchas variables y parámetros los cuales se han dejado libres para poder encontrar su valor óptimo y así adaptar el modelo al juego. Entre ellos tenemos la estructura de las capas de la red neuronal, el entrenamiento de esta, así como los parámetros de la fórmula 5.1 utilizada para el calculo de los valores *Q(s,a)*, y el parámetro añadido para hacer la extensión del modelo, el *Learning_rate_player*. Además, también se experimentará con la variable *epochs_fit* del entrenamiento de la red neuronal.

El procedimiento será idéntico que el explicado en la sección 6.1.1, solo que esta vez ya se conoce que variables codificarán el estado. A modo de recordatorio, los valores por defecto de los parámetros que pretendemos encontrar en esta sección son: una red neuronal con una capa oculta de 20 neuronas y función de activación *relu*, un entrenamiento realizado con *epochs_fit* = 10 y *epochs_RL_fit* = 10, y unos valores de $\gamma = 0{,}2$ y $\alpha = 0{,}75$ para la fórmula del RL.

6.2.1. Estructura de la red neuronal

Para poder decidir la estructura de la red neuronal se realizarán dos experimentos. En el primero se decidirá cual es la mejor función de activación que se puede utilizar para las capas intermedias, y el segundo decidirá como deben ser las capas de la red neuronal, es decir el número de capas y el número de neuronas de cada capa.

Función de activación

Para empezar, la primera prueba probará distintas funciones de activación con el objetivo de encontrar aquella que dé mejores resultados. Cabe mencionar que las funciones de activación softmax y softsign no se han usado debido a que no pueden ser utilizadas en regresión lineal ya que normalizan los valores y por lo tanto se pierde el valor que se pretendía calcular/predecir. En la siguiente tabla se pueden encontrar los resultados obtenidos en esta experimentación.

Función	**Porcentaje de victorias**
Relu (Referencia)	**67,93 %**
Selu	65,23 %
Elu	62,70 %
Linear	63,70 %
Sigmoid	65,83 %
Tanh	66,68 %
softplus	65,75 %

Cuadro 21: Resultados para la función de activación

Tal y como se puede observar en la tabla anterior, la mejor función de activación encontrada es la relu (Rectified Linear Unit), la que se había utilizado como referencia. Esta función simplemente filtra los valores negativos y los deja a cero.

Las capas

Una vez decidido que la mejor función de activación es la relu, se realizarán una serie de pruebas para determinar tanto la cantidad de neuronas como la cantidad de capas de la red neuronal. En la siguiente tabla se puede encontrar los resultados obtenidos.

Estructura	**Porcentaje de victorias**
1 capa, 20 neuronas (Referencia)	67,93 %
1 capa, 30 neuronas	66,38 %
1 capa, 10 neuronas	**68,33 %**
1 capa, 8 neuronas	67,58 %
2 capa, 10, 20 neuronas	64,75 %
2 capa, 8, 10 neuronas	64,83 %
3 capa, 8, 10, 20, neuronas	59,03 %

CUADRO 22: Resultados para las capas de la red neuronal

Como se puede observar en la tabla 22, la mejor opción es tener una única capa intermedia con 10 neuronas. También se puede apreciar que cuando aumenta el número de capas peores resultados da el modelo. Esto pude ser causado por que la función de recompensa que queremos aproximar es suficientemente sencilla como para que no se necesite una estructura muy compleja para predecirla.

Así pues una vez vistos los resultados de las tablas 21 y 22 se ha decidido que la estructura de la red neuronal sea una capa de entrada de datos de 208 neuronas con una función de activación relu, una capa intermedia de 10 neuronas con función de activación relu, y por último una capa de salida de 8 neuronas cuya función de activación es linear.

En la siguiente imagen se muestra un esquema que muestra la estructura final de la red neuronal.

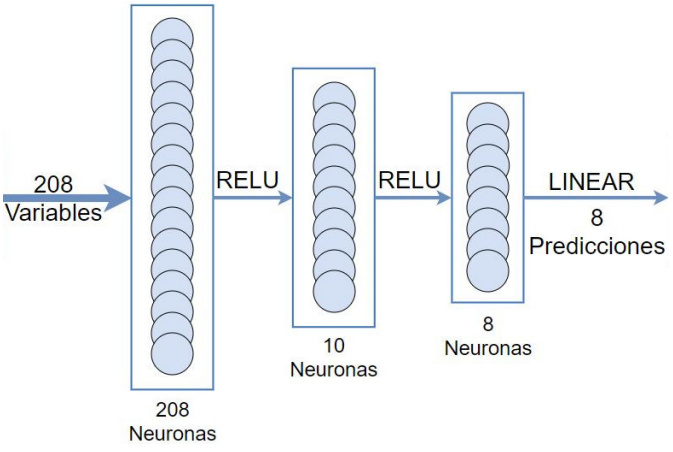

FIGURA 17: Esquema final de la red neuronal

6.2.2. Parámetros del algoritmo RL

A continuación se realizarán una serie de experimentos con el fin de determinar los tres parámetros utilizados en el algoritmo de reinforcement learning. Es decir los dos parámetros que aparecen en el cálculo de los valores *Q(s,a)*, tal y como se vio en la fórmula 5.1, es decir el parámetro *learning rate* (α) y el parámetro *discounting rate* (γ), y también el parámetro que decide cuantas veces hay que aplicar dicha fórmula, es decir *epochs_RL_fit*.

Learning rate

Con el fin de encontrar el valor más apropiado para el parámetro *learning rate*, se realizarán un conjunto de pruebas variado su valor entre 0,60 y 0,90 en incrementos de 0,05. En la siguiente tablas se muestran los resultados de estas pruebas.

Learning rate	**Porcentaje de victorias**
0,75 (Referencia)	**68,33 %**
0,60	65,45 %
0,65	65,93 %
0,70	66,55 %
0,80	66,48 %
0,85	67,88 %
0,90	65,88 %

CUADRO 23: Resultados para el parámetro learning rate

Como podemos observar en la tabla anterior, el valor que ha dado mejores resultados es valor de referencia de 0,75. Así pues se fijará este valor como valor definitivo de este parámetro.

Discounting rate

Para el otro parámetro de la formula del cálculo de los valores *Q(s,a)*, el *discounting rate*, se probarán valores más bajos, entre 0,05 y 0,30, en incrementos de 0,05. La siguiente tabla muestra los resultados obtenidos.

Discounting rate	**Porcentaje de victorias**
0,20 (Referencia)	68,33 %
0,05	65,98 %
0,10	**69,03 %**
0,15	67,13 %
0,25	67,08 %
0,30	66,78 %

CUADRO 24: Resultados para el parámetro discounting rate

Como se puede apreciar, el mejor valor para este parámetro es 0,10, así pues este valor será el valor final del parámetro *discounting rate*.

Epochs RL

Para finalizar con esta sección, se realizarán las pruebas oportunas para encontrar el valor óptimo de del parámetro *epochs_RL_fit*, el cual decide cuantas veces se aplica la fórmula de actualización de los valores *Q(s,a)* (fórmula 5.1) por cada muestra que recibe el modelo.

Epochs_RL_fit	**Porcentaje de victorias**
10 (Referencia)	**69,03** %
5	67,38 %
8	67,95 %
12	66,80 %
15	66,53 %

CUADRO 25: Resultados para el parámetro epochs_RL_fit

Tal y como se puede apreciar en la tabla anterior, el valor que proporciona mejores resultados es el valor de referencia, es decir cuando *epochs_RL_fit* tiene un valor de 10.

Así pues, los valores finales para los parámetros del algoritmo RL son: ***learning rate* = 0,75, *discounting rate* = 0,10 y *epochs_RL_fit* = 10**.

6.2.3. El entrenamiento de la red

Una vez definidos los parámetros finales del algoritmo de RL, se procederá a definir el parámetro del entrenamiento de la red neuronal, el *epochs_fit*.

Epochs

Para encontrar el valor óptimo para el parámetro *epochs_fit*, se han realizado un conjunto de pruebas cuyos resultados se muestran en la tabla 26 que hay a continuación.

Epochs de la red	**Porcentaje de victorias**
10 (Referencia)	**69,03 %**
8	63,88 %
5	58,88 %
12	64,18 %
15	65,65 %
20	65,80 %

Cuadro 26: Resultados para el parámetro epochs_fit

Como podemos ver, el resultado de las pruebas realizadas indica que el mejor valor para el parámetro epochs es 10.

6.2.4. Parámetros de los modelos extendidos

Una vez encontrados los mejores valores para los parámetros del modelo básico, aun falta por encontrar le parámetro definido para poder utilizar las extensiones del modelo, es decir el parámetro *Learning_rate_player* de los modelos *LearnerModel* y *CoopModel*. Además, debido a las modificación realizadas, también se experimentará con las capas del modelo *CoopModel*.

Learning rate player

A modo de recordatorio, el parámetro learning rate player fue añadido con el objetivo de poder dar más importancia a las muestras obtenidas del jugador para así poder aprender mejor su comportamiento. En la siguiente tabla se muestran los resultados de la experimentación de este parámetro.

Learning rate player	**Porcentaje de victorias**
70 %	54,73 %
75 %	**62,35 %**
80 %	55,04 %

Cuadro 27: Resultados para el parámetro learning rate player

Como se puede apreciar en la tabla 27, los mejores resultado se obtienen al utilizar un *learning rate player* de 0,75, el mismo que el del *learning rate* del modelo básico. Esto se debe a que darle más importancia a las muestras obtenidas del jugador no ha ayudado a que el modelo aprenda mejor.

También se puede ver como el porcentaje de victorias a disminuido respecto al modelo básico, pero esto se debe a que este modelo no pretende logran un alto rendimiento sino aprender los movimientos del compañero. Así pues, difícilmente conseguirá igualar o superar el porcentaje de victorias del modelo del cual aprende, pero si que será capaz de aprender de la manera en la que juega el aliado, lo cual se evaluará en la sección 6.3.1. Además, hace falta tener en mente que se ha entrenado con un modelo que no sabe jugar perfectamente al juego, debido a la falta de tiempo para realizar el entrenamiento con un jugador real.

Capas de la red del modelo CoopModel

Por otra parte, dado que el modelo *CoopModel* necesita cambiar la cantidad de valores que recibe de la entrada de 208 a 336 y los que devuelve de salida de 8 a 64, también se ha realizado un pequeño experimento para averiguar la configuración de las capas óptima para este modelo.

Capas de CoopModel	**Porcentaje de victorias**
1 capa, 10 neuronas	51,93 %
1 capa, 50 neuronas	55,05 %
1 capa, 65 neuronas	**56.68 %**
1 capa, 70 neuronas	56,25 %
1 capa, 80 neuronas	54,98 %
2 capa, 70, 65 neuronas	56,43 %

Cuadro 28: Resultados para las capas de CoopModel

Como se puede ver, la mejor configuración para las capas del modelo *CoopModel* es utilizar una única capa de 65 neuronas. Cabe destacar que este resultado es similar que el del modelo básico, dado que ambos utilizan una sola capa intermedia de unas cuantas pocas neuronas más que la de sus capas de salida.

6.3. Evaluación/Comparación

Una vez que se han encontrado todo los valores óptimos de los parámetros, hay que evaluar los tres modelos para ver si se ha conseguido realizar el objetivo de cada uno de ellos. Estos objetivos son los de superar el azar (para el modelo básico), aprender del aliado (para el modelo *LearnerModel*) y aprender a cooperar (para el modelo *CoopModel*).

Como ya se mencionó al final de la sección 6.1.3, el modelo básico es capaz de superar al azar dado que al final de la experimentación se ha conseguido obtener un porcentaje de victorias del 69.03 %. En cuanto a los demás modelos, se realizarán un conjunto de pruebas para averiguar si han logrado alcanzar su objetivo satisfactoriamente.

Para tal fin se realizaran dos pruebas, una por cada modelo, en las cuales intervendrá un jugador real para poder realizar una evaluación de sus decisiones. Por la parte de la prueba realizada para el modelo *LearnerModel* se intentará observar comportamientos comunes entre este y el modelo básico utilizado durante el entrenamiento de los modelos *LearnerModel* y *CoopModel*, el cual se ha llamado *Support*. Por la parte de la prueba realizada para el modelo *CoopModel* se intentará observar comportamientos cooperativos simples con el jugador y ver como le afectan los cambios de decisión del jugador ha las decisiones del modelo.

6.3.1. Evaluación del LearnerModel

Para la evaluación del *LearnerModel*, primeramente se realizarán 5 batallas en las que el aliado será el modelo llamado *Support*. Luego se realizarían otras 5 batallas, las cuales serán jugadas por los mismos pokémon (usando la misma semilla tal y como se explicaba en la sección 6.1.1), solo que esta vez el aliado estará controlado por el modelo *LearnerModel*. Así pues, una vez jugadas las batallas, se podrán comparar las acciones elegidas por ambos modelos dado que han sido escogidas en circunstancias similares.

A continuación se mostrarán los resultados de las batallas. Estos resultados están recogidos en las imágenes de las trascripciones en la consola de la partida, la del modelo *Support* a izquierda y la del modelo *LearnerModel* a la derecha. Aparte, para poder entender más fácilmente el contenido de las transcripciones, también se proporciona la imagen de la interfaz gráfica al inicio de cada partida.

También hace falta recordar que tal y como se explicaba en la sección 4.5.1, el aliado, y por tanto el pokémon cuyas acciones queremos fijarnos, es el pokémon situado en la parte inferior derecha, cuyo nombre se encuentra en el rectángulo gris inferior izquierdo. Además su nombre es el segundo en aparecer en la lista que aparece al principio de cada transcripción (en el mensaje "*Start Battle: ...*").

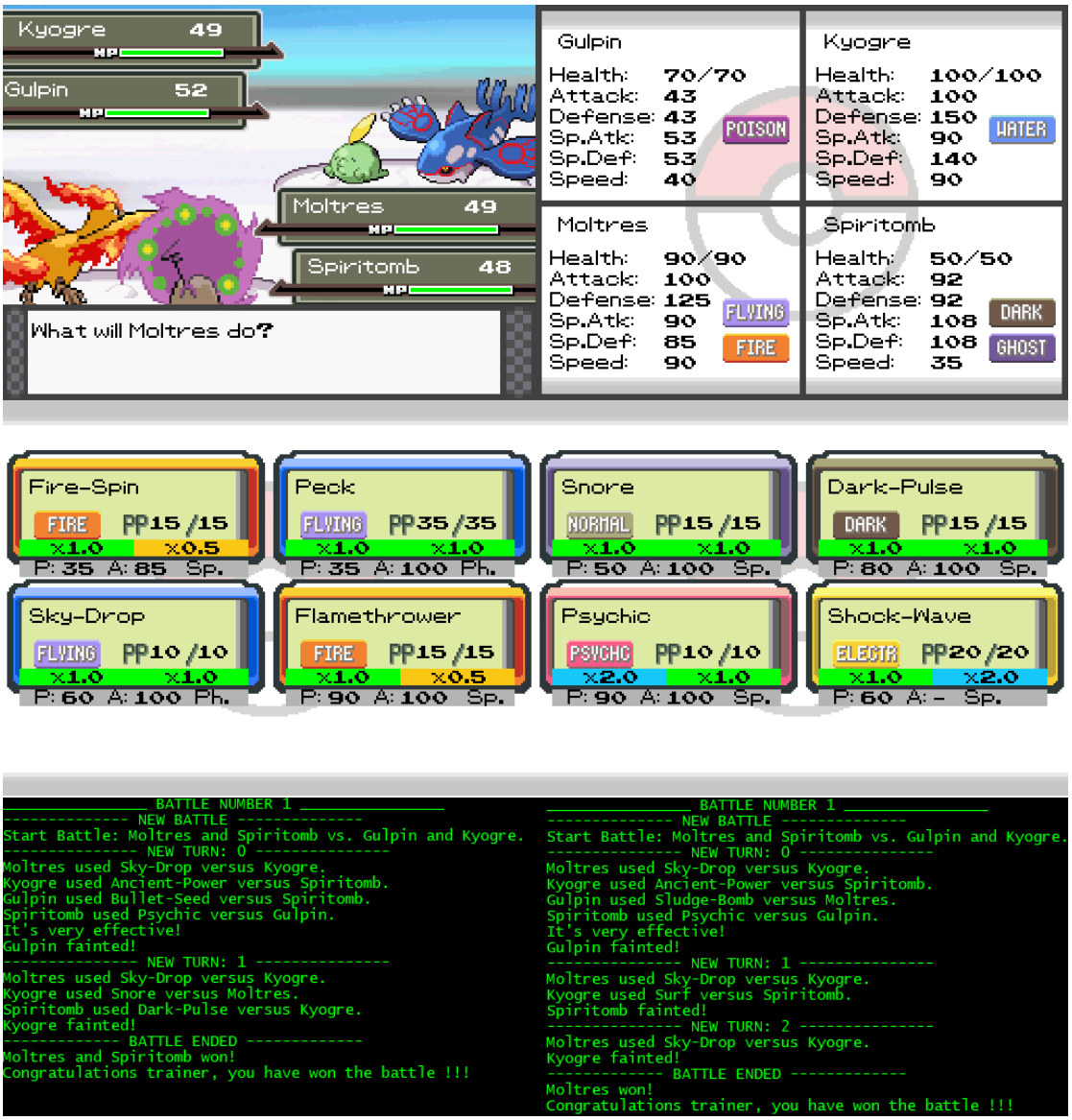

FIGURA 18: Comparación de la 1ª batalla de la evaluación de LearnerModel

En la primera batalla, tal y como podemos observar en la figura 18, ambos modelos han elegido realizar la misma acción (*Psychic* contra *Gulpin*) en el turno 0, la cual es probablemente su mejor opción. Por desgracia, en el siguiente turno el modelo *LearnerModel* ha sido vencido y no se ha podido comprobar si realizaría la misma acción que el modelo *Support*.

En cuanto a la elección de movimientos, en la primera partida podemos observar que ha utilizado tanto *Psychic* como *Dark-Pulse*, sus dos mejores ataques. Además, el primero lo ha realizado contra el enemigo contra el cual el movimiento es eficaz (aumentando el daño provocado). En cuanto al segundo movimiento, lo ha escogido seguramente debido a que al no tener la ventaja de la efectividad, el movimiento *Dark-Pulse* era la mejor opción al contar con la bonificación de tipo.

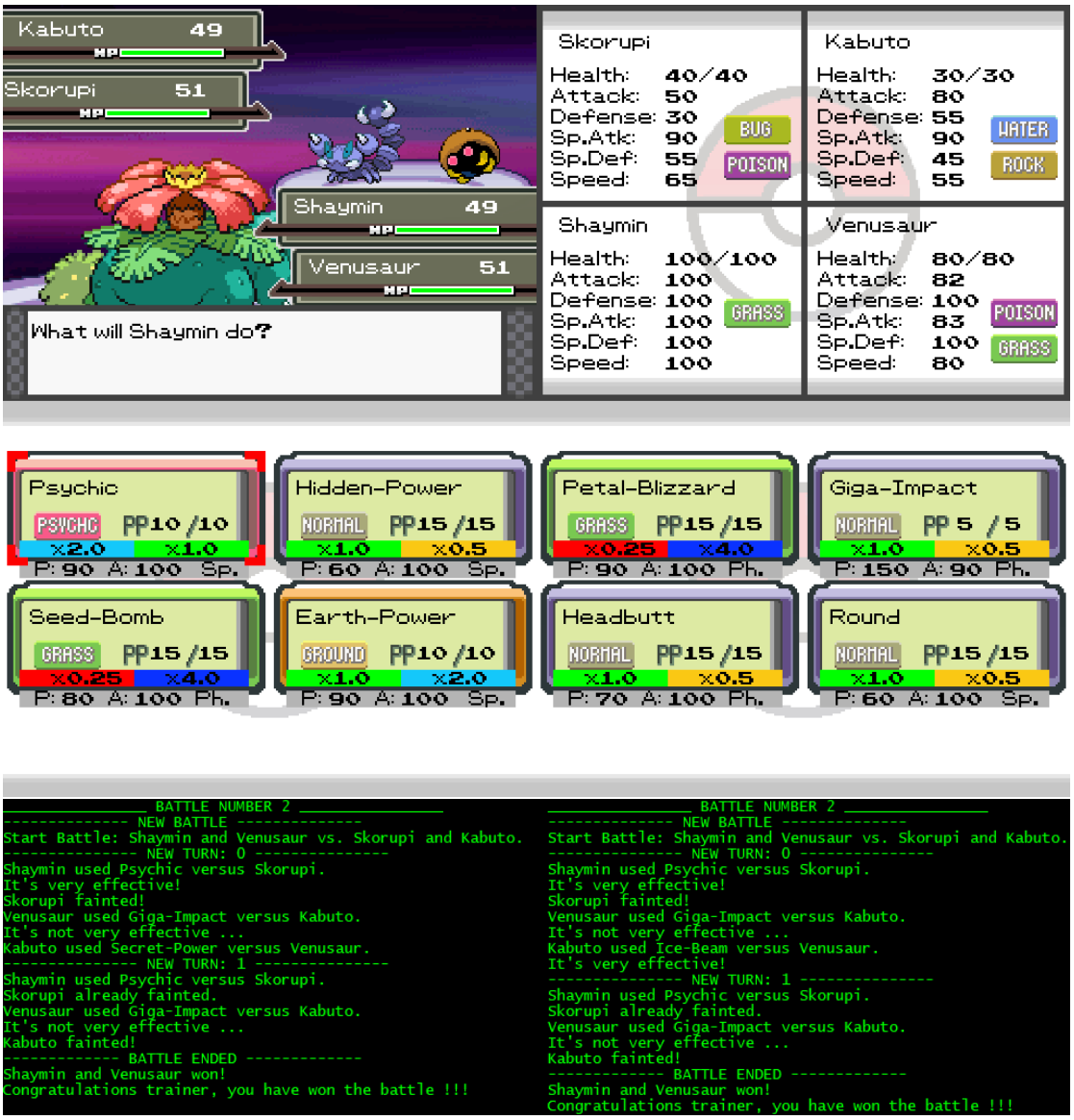

FIGURA 19: Comparación de la 2ª batalla de la evaluación de LearnerModel

Lo mismo sucede en la segunda batalla, la cual se puede ver en la imagen 19, ambos modelos han elegido realizar la misma acción (*Giga-Impact* contra *Kabuto*) en ambos turnos, la cual no es su mejor opción, dado que no es muy efectivo contra *Kabuto*, y tienen un efectividad normal con *Skorupi*, aparte de que *Petal-Blizzard* es mucho más efectivo contra *Kabuto*.

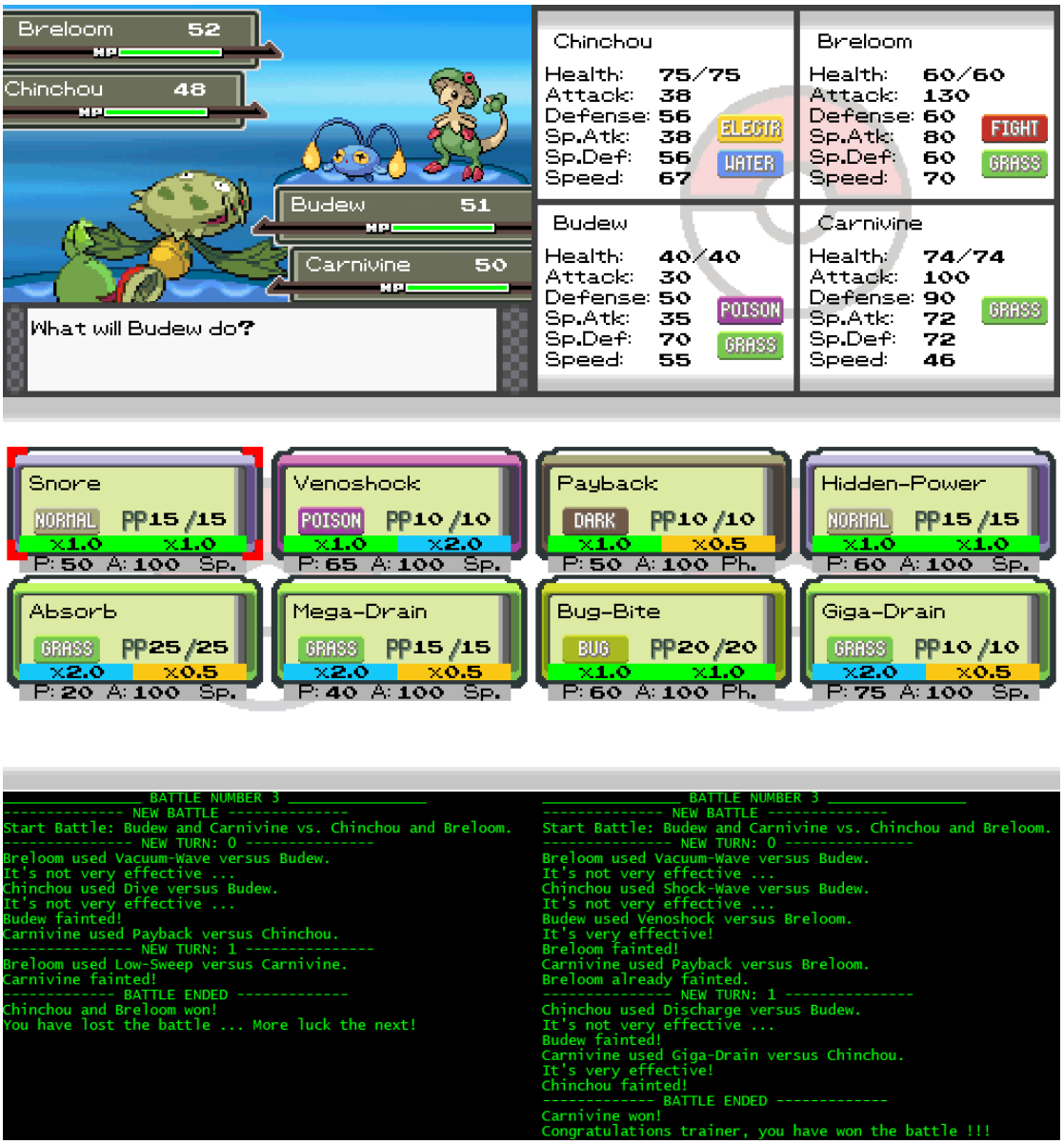

FIGURA 20: Comparación de la 3ª batalla de la evaluación de LearnerModel

En la tercera batalla, la cual se puede ver en la figura 20, a pesar de que ambos modelos han decidido realizar el mismo ataque, han decidido realizarlo contra enemigos diferentes. Hay que destacar que aunque el modelo *LearnerModel* haya atacado a un pokémon debilitado, al principio del turno (donde se deciden las acciones) si que estaba vivo por lo que no podía saberlo y actuar en consecuencia. Como se puede ver en el siguiente turno decide atacar al enemigo que aun esta vivo.

En cuanto a las acciones, el modelo *LearnerModel* sin duda ha escogido la peor opción posible en el primer turno, pero ha escogido la mejor opción posible para el segundo turno. Esto puede deberse a una falta de entrenamiento.

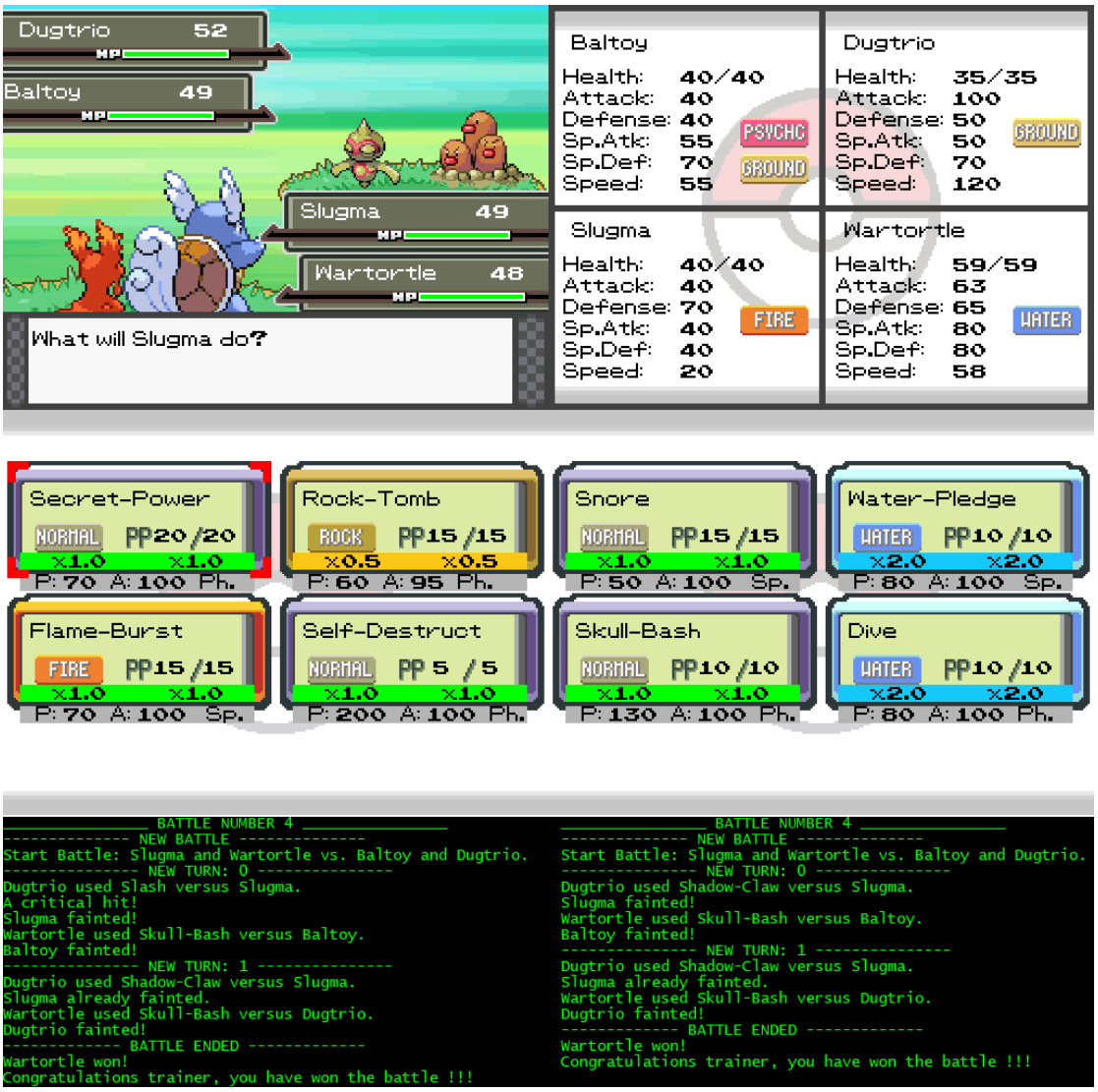

FIGURA 21: Comparación de la 4ª batalla de la evaluación de LearnerModel

En la cuarta batalla, la cual se puede ver en la imagen 21, vuelven ha realizar el mismo ataque, *Skull-Bash* contra *Baltoy* en el primer turno, y contra *Dugtrio* en el segundo, dado que *Baltoy* fue eliminado en el primer turno.

A pesar de que tiene dos ataques de tipo Agua que son eficaces con ambos, ha decidido utilizar *Skull-Bash*, lo que hace pensar que ambos modelos se guían más por la potencia del ataque que por la relaciones de tipos. Aunque esto es solo una suposición dado que el valor final del daño es difícil de saber a priori, y podría ser que *Skull-Bash* causará más daño que los demás.

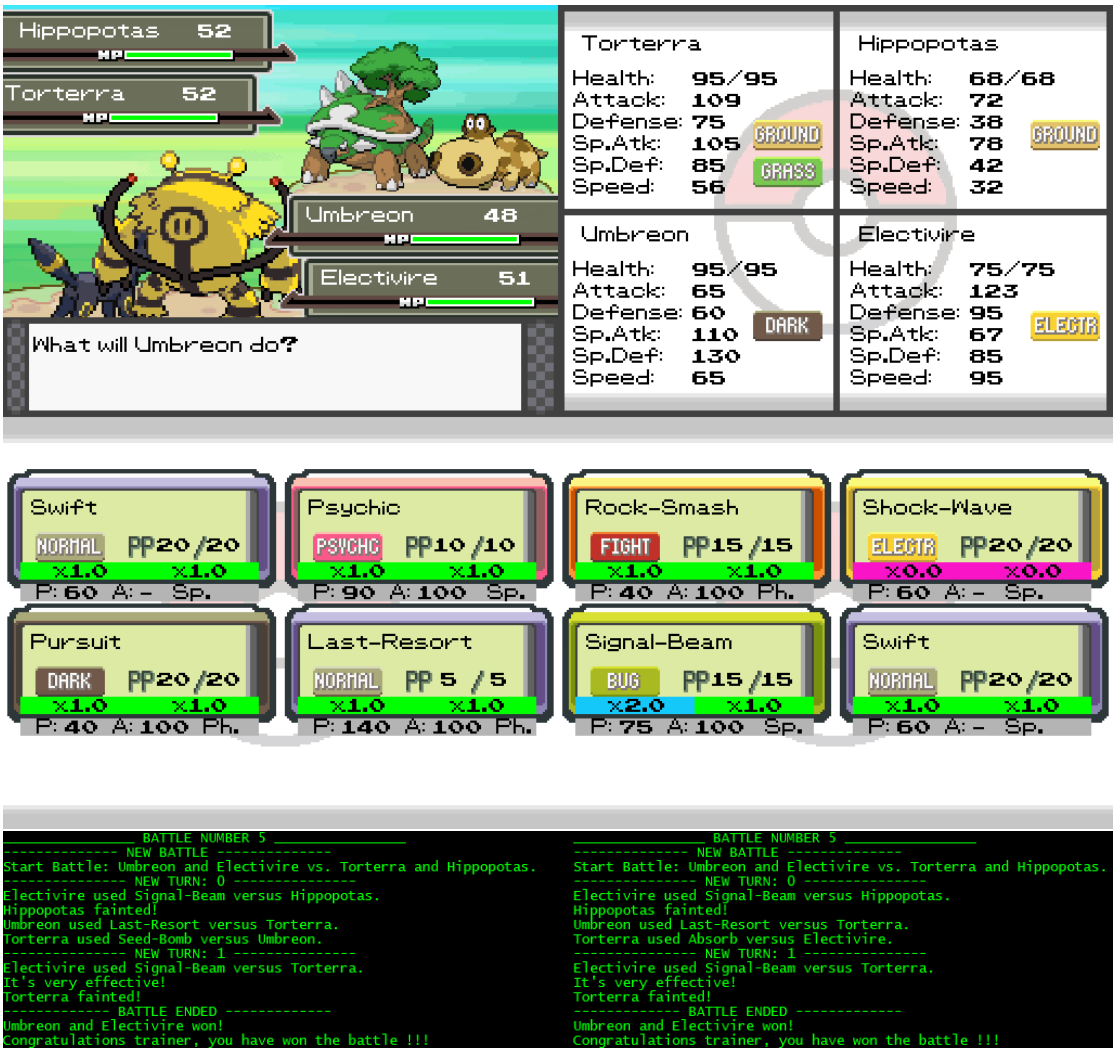

FIGURA 22: Comparación de la 5ª batalla de la evaluación de LearnerModel

Por último, en la imagen 22 se muestra la quinta batalla donde se puede ver que sucede una situación muy parecida a la anterior dado que ambos modelos realizan el ataque *Signal-Beam* contra *Hippopotas* en el primer turno, y contra *Torterra* en el segundo, eliminando al enemigo con cada ataque.

Pero a diferencia del caso anterior, la elección de movimientos es la mejor que podían realizar dado que es el que tiene más potencia de ataque y además cuenta con que el tipo es efectivo contra el enemigo *Torterra*.

Como conclusión se puede decir que en general el modelo *LearnerModel* ha decidido realizar las mismas acciones que el modelo *Support* a pesar de que en algunos casos ambos tenían otras acciones igualmente validas o incluso mejores. Además, las veces que no coincidían exactamente las acciones si que coincidían en la elección del movimiento. Por este motivo, se podría decir que el modelo *LearnerModel* a conseguido aprender el comportamiento del modelo *Support*.

Aun así, para poder asegurar con firmeza que el modelo *LearnerModel* ha aprendido el comportamiento sería necesario realizar más evaluaciones con distintos modelos entrenados teniendo a un jugador como aliado y que este fuese variando su comportamiento. Pero debido al tiempo requerido para realizar las batallas con un jugador real, la cantidad de batallas mínimas necesarias para que el modelo aprenda y el tiempo limitado del proyecto no se ha podido llevar a cabo dicha evaluación.

6.3.2. Evaluación del CoopModel

Para la evaluación del modelo *CoopModel* se procederá de forma similar. Se realizarán 5 batallas en las que el aliado será el modelo *CoopModel*. Luego se realizarán otra vez las mismas 5 batallas, solo que esta vez se cambiarán las decisiones que tome el jugador. De esta manera se podrá observar como cambian, o no, las decisiones del aliado, y así poder sacar conclusiones sobre si su comportamiento es o no cooperativo.

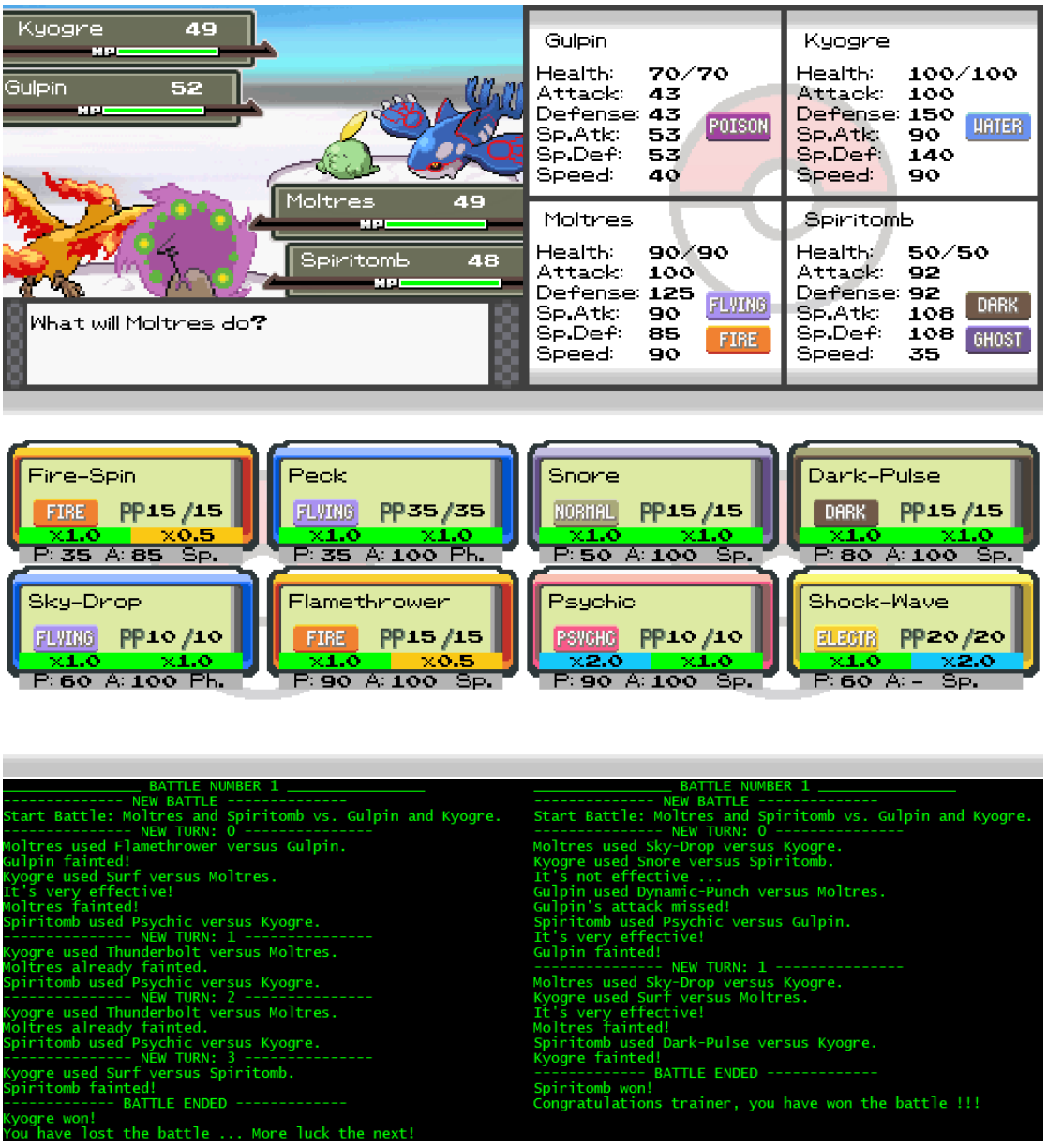

FIGURA 23: Comparación de la 1ª batalla de la evaluación de CoopModel

En la primera batalla que se muestra en la imagen 23, se puede observar que entre las dos batallas la principal diferencia, en cuanto a la toma de decisiones, es que el jugador ha decidido cambiar de enemigo. Esto ha su vez ha provocado que el modelo *CoopModel* también haya cambiado de objetivo debido a que el modelo ha previsto que el enemigo sería eliminado por el ataque del jugado, y ha optado por atacar al otro enemigo.

En cuanto a la elección de movimientos, por lo que hace a la primera partida ha sido constante, y ha utilizado *Psychic* el cual es una muy buena opción dado que es el movimiento con más potencia. En cuanto a la segunda partida, ha utilizado *Psychic* el cual es efectivo contra *Gulpin*, pero cuando ha lo ha debilitado, a pasado a atacar al otro enemigo usando *Dark-Pulse*, dado que este cuenta con una bonificación de $\times 1,5$.

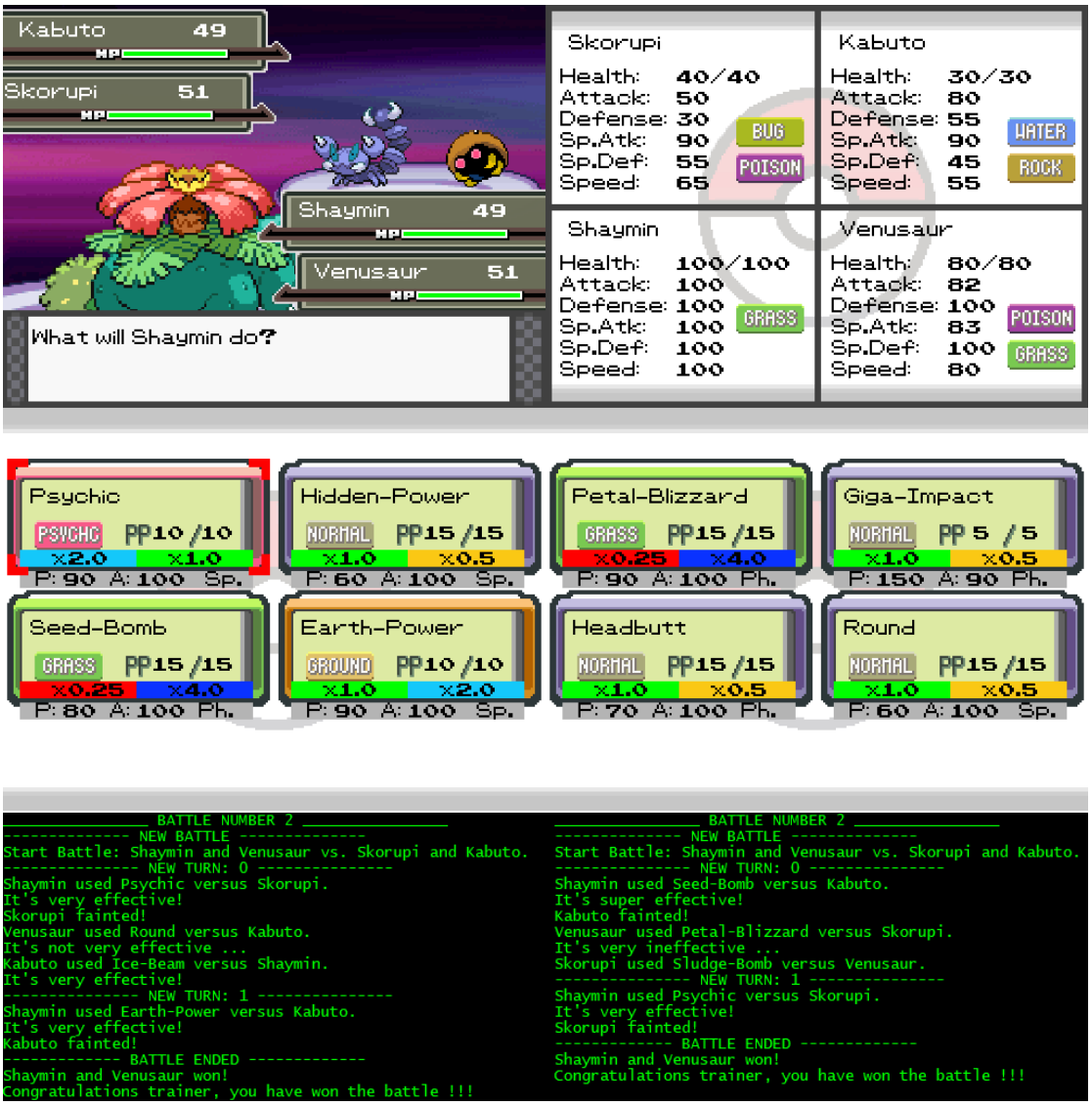

FIGURA 24: Comparación de la 2ª batalla de la evaluación de CoopModel

En la imagen 24 se puede ver como en la segunda batalla sucede algo parecido a la anterior. El cambio de acción del jugador hace que el aliado cambie de acción también. Pero en esta batalla la elección de movimiento que hace el aliado no es muy buena. En la primera partida ha escogido utilizar el movimiento con menos potencia, y en la segunda el movimiento con peor efectividad.

Una posible explicación para esto último es que el modelo haya aprendido a que debe utilizar un movimiento del mismo tipo que el usado por el jugador, pero esto es altamente improbable debido a que no hay indicios anteriores de este comportamiento. Además no es una estrategia muy útil, como se acaba de comprobar, ni se creó al modelo con tal finalidad.

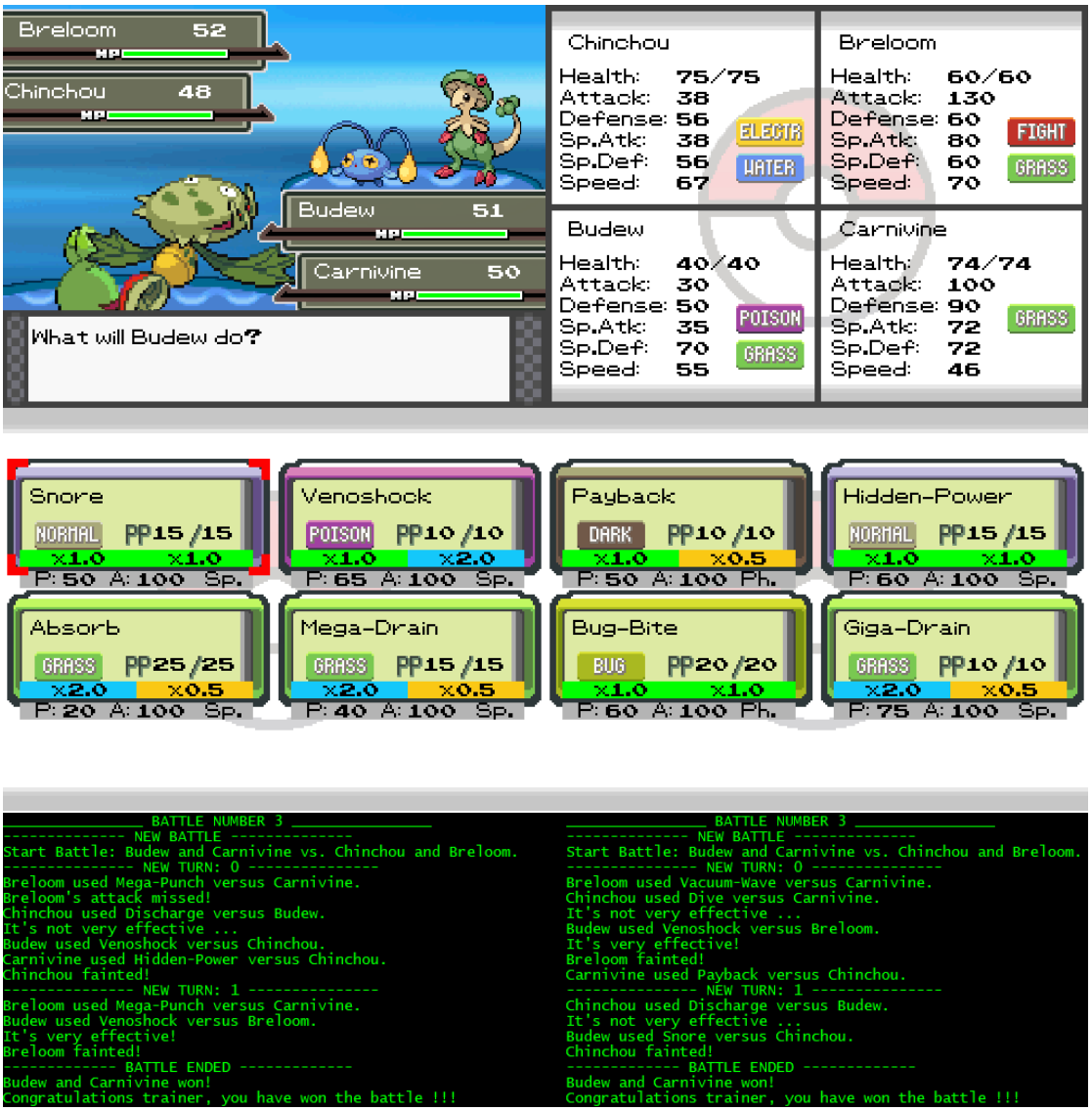

FIGURA 25: Comparación de la 3ª batalla de la evaluación de CoopModel

En la tercera batalla, que se puede ver en la figura 25, en la primera partida tanto el jugador como el aliado han atacado al mismo enemigo, y este ha sido eliminado en consecuencia. Por el contrario, en la segunda partida el aliado si ha atacado al enemigo contrario. Dado que la diferencia entre las dos partidas es la muerte del enemigo de un solo golpe, el comportamiento del modelo *CoopModel* se puede deber a que en la primera partida haya previsto que el enemigo no moriría por el ataque de jugador, y ha considerado que sería mejor atacar al mismo enemigo para poder debilitarlo fácilmente.

La elección de movimientos del modelo es muy mejorable dado que para la primera partida ha usado *Hidden-Power* que no cuenta ni con bonificación de tipo ni una buena efectividad con el enemigo elegido, a diferencia del movimiento *Giga-Drain* que a demás tiene más potencia. Para la segunda partida ha elegido el peor movimiento posible (*Payback*) ya que este es el que tiene menos potencia.

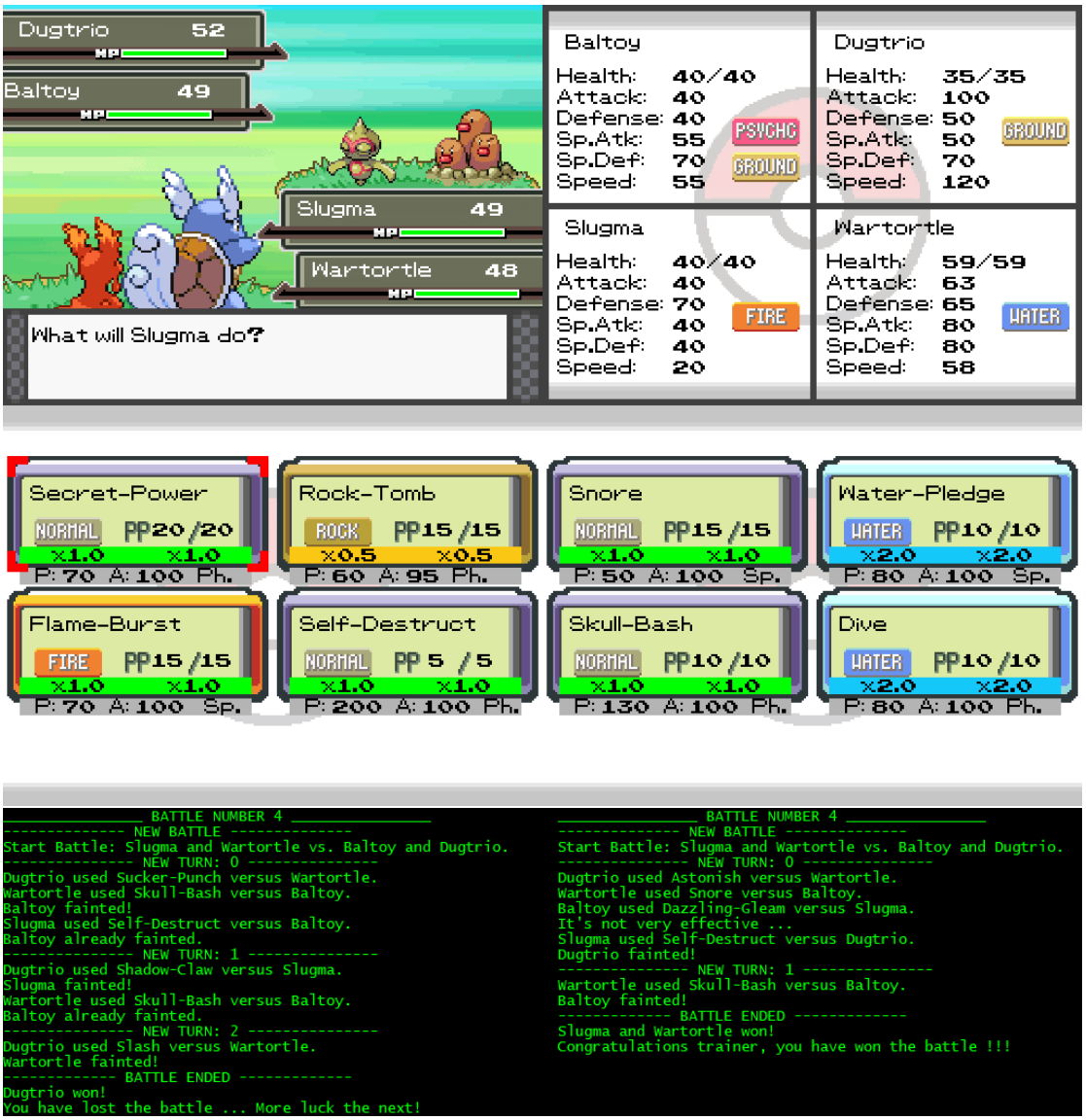

FIGURA 26: Comparación de la 4ª batalla de la evaluación de CoopModel

Respecto a la cuarta batalla que se puede ver en la imagen 26, en la primera partida se puede apreciar que a pesar de que el jugado ha realizado uno de los movimientos con más potencia que hay (*Self-Destruct*) el aliado a considerado que el enemigo no moriría y le ha atacado también. En este caso el enemigo se ha debilitado con el ataque del aliado y en consecuencia el ataque del jugador a fallado.

En cuanto a la elección de movimientos, el aliado ha utilizado bastante el movimiento *Skull-Bash* a pesar de tener dos movimientos que tenían tanto bonificación del tipo como efectividad contra los enemigos. Aunque si nos fijamos en la potencia veremos que hay una diferencia de 50 puntos, por lo que es probable que *Skull-Bash* cause más daño que los demás. A pesar de esto, en la segunda partida ha utilizado el movimiento *Snore* el cual es definitivamente la pero opción.

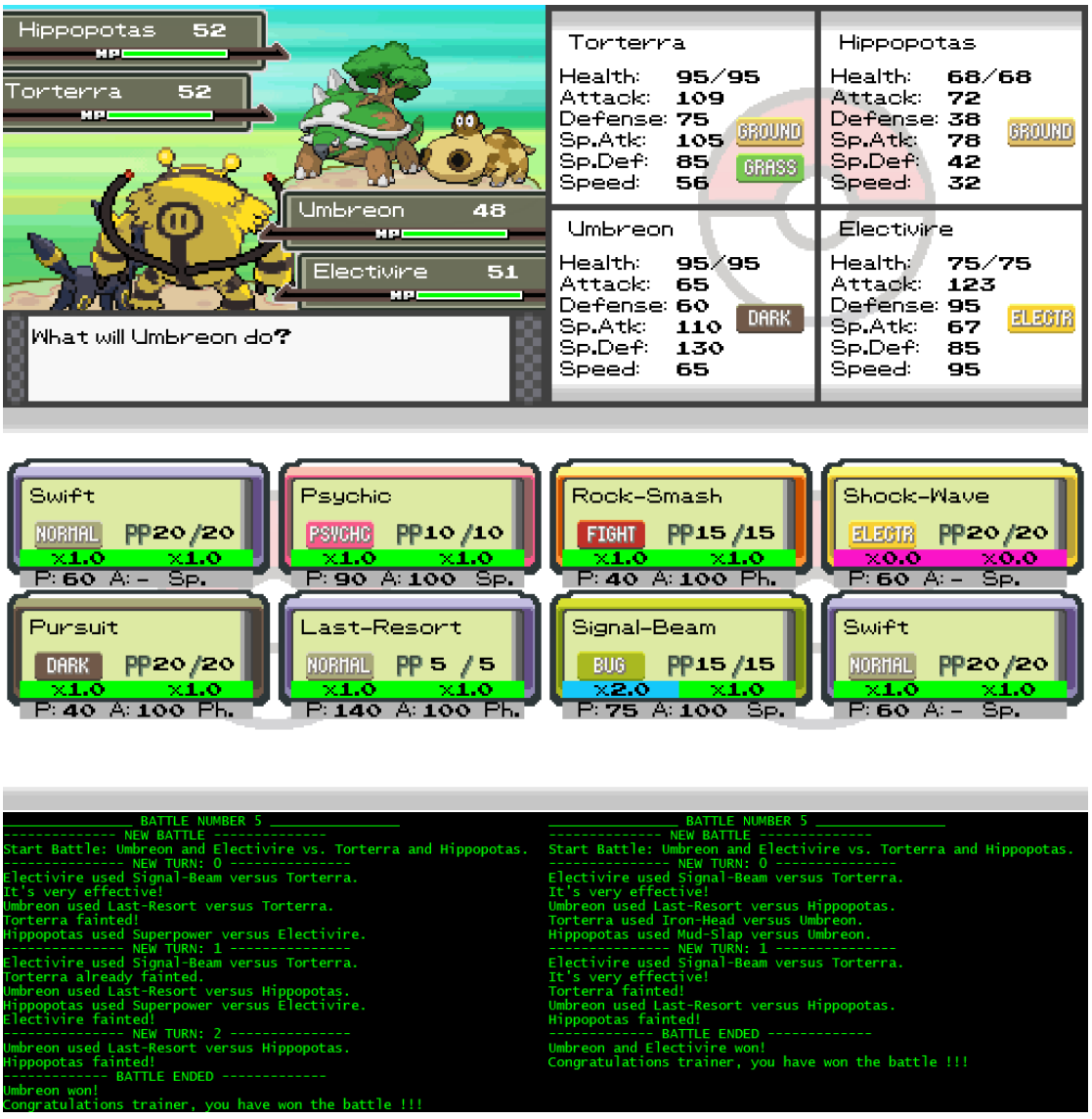

FIGURA 27: Comparación de la 5ª batalla de la evaluación de CoopModel

Por último, en la quinta batalla, que se encuentra en la imagen 27, se puede ver que sucede una situación muy parecida a la de la batalla número tres. En la primera batalla, la elección del enemigo al cual el jugador atacará no le impide al aliado atacar al mismo enemigo, dado que posiblemente haya previsto que el enemigo no se debilitaría con un solo ataque. En consecuencia el enemigo *Torterra* es derrotado en el primer turno gracias a los dos ataques recibidos.

Al igual que pasaba en la quinta batalla de la sección anterior, la elección de movimientos por parte del aliado es constante, pero a la vez es la mejor opción que tiene el aliado contra los dos enemigos. Esto se debe a que es el ataque más potente de los que dispone el aliado, además de ser eficaz contra uno de los enemigos.

Como conclusión se puede decir que este modelo tiene fallos similares que los anteriores modelos en cuanto a elección de movimientos, pero que este tiende a variar un poco más estos durante el combate, con las ventajas y desventajas que esto pueda ocasionar.

Por la parte de la elección de enemigos, este modelo es bastante mejor, dado que es más flexible a las elecciones del jugador y intenta deducir si es necesario atacar al mismo enemigo o no de una forma bastante satisfactoria.

Capítulo 7

Conclusiones

Este capítulo pretende resumir las conclusiones a las que ha llegado este proyecto (sección 7.1), así como la conclusiones personal sobre la realización de este (sección 7.2). Finalmente se ha dedicado una sección a hablar de posibles trabajos futuros relacionados con este proyecto (sección 7.3).

7.1. Conclusiones del proyecto

En lo referente al videojuego creado, se considera que funciona de forma satisfactoria debido a que cumple perfectamente con su objetivo principal, el de ser un entorno para el agente fácil de jugar pero difícil de predecir con exactitud. Esto se debe principalmente a que la cantidad de acciones a elegir en cada turno no es muy elevada, y a la complejidad de la fórmula 4.3 sobre el cálculo del daño producido en la que intervienen distintos factores.

En cuanto a la interfaz gráfica, esta también cumple perfectamente su función principal, la de proporcionar información de una forma visual y clara. También se ha podido comprobar que es fácil de utilizar y facilita la elección de la acción por parte del jugador.

Por la parte de la red neuronal se ha podido apreciar que a pesar de evaluarse con pokémon distintos a los que aparecían en su entrenamiento, esta ha sido capaz de extrapolar información de las muestras para dar buenos resultados en su evaluación, hecho que no podría haber pasado sino se hubiese utilizado una red neuronal. Así mismo también se ha demostrado que la red no necesita una estructura muy compleja para asimilar correctamente la función de recompensa del juego.

En cuanto a los modelos creados hay que mencionar que se ha conseguido cumplir con el requisito mínimo de ser mejor que el azar. Por la parte del aprendizaje a través del jugador, se han podido observar que este modelo tenía comportamientos similares al modelo utilizado como aliado en su entrenamiento.

En cuanto al modelo que aprende a cooperar con el jugador, se ha podido observar que en general este tiende a atacar al compañero del enemigo que es atacado por el jugador, dado que prevé que el enemigo atacado por el jugador será debilitado en ese turno. También ataca al mismo enemigo que el jugador cuando prevé que el enemigo no se debilitará de un solo golpe, aunque aveces falle su previsión.

Ambos modelos parecen actuar tal y como deberían, pero como ya se ha comentado anteriormente, para una comprobación más exhaustiva se debería realizar el entrenamiento de los modelos de forma manual, jugando un jugador humano. Pero esto conllevaría demasiado tiempo teniendo en cuenta el tiempo limitado del proyecto.

Por la parte de la elección de movimientos, los modelos suelen elegir bastante bien el más indicado, pero algunos de los fallos que cometen son graves. Este hecho seguramente este relacionado con una falta de entrenamiento en el cual pudiesen aprender mejor el funcionamiento de estos, pero debido al tiempo limitado del proyecto se ha decidido no entrenar a los modelos durante largos periodos de tiempo.

7.2. Conclusiones personales

Por la parte personal, este proyecto ha sido un reto complicado con el que he adquirido experiencia y conocimientos sobre las redes neuronales y el aprendizaje por refuerzo, temas de los cuales tenia muy poco conocimiento. Además, la experiencia obtenida al realizar un proyecto de tal envergadura sin duda me será útil en el futuro.

Aparte, la experiencia de trabajar con las redes neuronales ha sido muy interesante debido a que la capacidad de estas para predecir o clasificar valores me ha impresionado mucho. Además, la facilidad de uso que proporciona la librería *Keras* ha hecho que la creación de la red neuronal fuese muy sencilla. Aun así no todo ha sido fácil con las redes neuronales, ya que sus resultados pueden variar enormemente con pequeños cambios en su estructura, además de que pueden llegar a necesitar mucho tiempo de entrenamiento para obtener resultados aceptables.

Además, el desarrollo de la interfaz gráfica fue una experiencia diferente dado que no he tenido muchas oportunidades de trabajar en programas gráficos que tuviesen una funcionalidad tan importante detrás. Aparte ha sido bastante cómodo de realizar gracias a la librería *PyGame* que gestiona gran parte del trabajo y proporciona mucha flexibilidad para poder implementar casi cualquier cosa que se desee.

7.3. Trabajos futuros

Dada la limitación temporal del proyecto, hay ciertos aspectos que han quedado fuera de la extensión de este proyecto que podrían mejorar el modelo creado, como por ejemplo un entrenamiento con muchísimas más muestras obtenidas de partidas realizadas con jugadores reales.

Además, también se podría intentar idear una función de recompensa que tuviese en cuenta más factores con el objetivo de poder guiar mejor al algoritmo de aprendizaje por refuerzo hacia la acción más correcta.

Por otra parte, se podría intentar utilizar el modelo creado para jugar a otros videojuegos. De esta forma se podía comprobar si este modelo es capaz de aprender a jugar a otros videojuegos como lo es el algoritmo DQN, en el cual se ha basado el algoritmo creado.

Por último, se podría intentar crear un modelo que tuviese en cuenta todas las variables del estado, modificando la estructura de la red a una de más compleja que requeriría de un elevado número muestras de entrenamiento. Así quizás la red neuronal sería capaz de predecir con más exactitud la recompensa obtenida.

Referencias

[1] Georgios N. Yannakakis and Julian Togelius. *Artificial Intelligence and Games*. Springer, 2018. http://gameaibook.org.

[2] The OpenDota Blog: Data dump (march 2011 to march 2016), 2017. URL https://blog.opendota.com/2017/03/24/datadump2/.

[3] Arthur L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of research and development*, pages 210–229, 1959. URL http://ieeexplore.ieee.org/document/5392560/.

[4] Bram Bakker, Shimon Whiteson, Leon Kester, and Frans C. A. Groen. *Traffic Light Control by Multiagent Reinforcement Learning Systems*, pages 475–510. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. ISBN 978-3-642-11688-9. doi: 10.1007/978-3-642-11688-9_18. URL https://doi.org/10.1007/978-3-642-11688-9_18.

[5] Shai Shalev-Shwartz, Shaked Shammah, and Amnon Shashua. Safe, multi-agent, reinforcement learning for autonomous driving. *CoRR*, abs/1610.03295, 2016. URL http://arxiv.org/abs/1610.03295.

[6] Steven Rabin. *Ellie: Buddy AI in The Last of Us*, pages 431–442. A K Peters/CRC Press, 2015. ISBN 9781482254792. URL http://www.gameaipro.com/GameAIPro2/GameAIPro2_Chapter35_Ellie_Buddy_AI_in_The_Last_of_Us.pdf.

[7] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8 (3):279–292, May 1992. ISSN 1573-0565. doi: 10.1007/BF00992698. URL https://doi.org/10.1007/BF00992698.

[8] Thore Graepel, Ralf Herbrich, and Julian Gold. Learning to fight. In *Proceedings of the International Conference on Computer Games: Artificial Intelligence, Design and Education*, pages 193–200, January 2004. URL https://www.microsoft.com/en-us/research/publication/learning-to-fight/.

[9] DeepMind: Learning to cooperate, compete, and communicate, 2017. URL https://blog.openai.com/learning-to-cooperate-compete-and-communicate/.

[10] Ryan Lowe, Yi Wu, Aviv Tamar, Jean Harb, Pieter Abbeel, and Igor Mordatch. Multi-agent actor-critic for mixed cooperative-competitive environments. *CoRR*, abs/1706.02275, 2017. URL `http://arxiv.org/abs/1706.02275`.

[11] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, pages 529 – 533, 2015. URL `https://storage.googleapis.com/deepmind-media/dqn/DQNNaturePaper.pdf`.

[12] DeepMind: Deep reinforcement learning, 2016. URL `https://deepmind.com/blog/deep-reinforcement-learning/`.

[13] Github. `https://github.com/`.

[14] Pete Shinners. Pygame. `http://pygame.org`, 2011.

[15] François Chollet et al. Keras github. `https://github.com/keras-team/keras`, 2015.

[16] Paul Hallett. Pokéapi, the restful pokémon data api. `https://pokeapi.co`, 2014.

[17] Keras. `https://keras.io/`.

[18] Wikidex. `http://es.pokemon.wikia.com`.

[19] Bulbapedia. `https://bulbapedia.bulbagarden.net`, 2005.

[20] Tensorflow. `https://www.tensorflow.org/`.