Universidad de Alcalá
Escuela Politécnica Superior

Grado en Ingeniería Informática

Trabajo Fin de Grado

Desarrollo de Inteligencia Artificial mediante aprendizaje por refuerzo para combates Pokémon

Autor: Marco González Martínez

Tutor: Adrián Domínguez Díaz

<<Año de realización del examen>>

UNIVERSIDAD DE ALCALÁ
Escuela Politécnica Superior

GRADO EN INGENIERÍA INFORMÁTICA

Trabajo Fin de Grado

Desarrollo de Inteligencia Artificial mediante aprendizaje por refuerzo para combates Pokémon

Autor: Marco González Martínez

Tutor/es: Adrián Domínguez Díaz

TRIBUNAL:

Presidente: << Nombre y Apellidos >>

Vocal 1º: << Nombre y Apellidos >>

Vocal 2º: << Nombre y Apellidos >>

FECHA: << Fecha de depósito >>

Contenido

1. Resumen & Abstract

1.1. Resumen

Este proyecto aborda la viabilidad del uso de estrategias de *Deep Reinforcenment Learning* para diseñar un modelo de IA capaz de combatir en combates del videojuego *Pokémon*. En el proyecto se abordan varios ejemplos de entornos más sencillos en los que se evalúa el algoritmo DQN. Como resultado del proyecto se puede observar que el algoritmo DQN demuestra un gran potencial para su uso en entornos de juegos.

1.2. Abstract

This project addresses the feasibility of using Deep Reinforcement Learning strategies to design an AI model capable of fighting in Pokémon video game battles. The project addresses several examples of simpler environments in which the DQN algorithm is evaluated. As a result of the project, the DQN algorithm shows great potential for use in gaming environments.

2. Introducción

En los últimos años el uso de la IA ha experimentado un crecimiento sin precedentes. Esta revolución ha sido impulsada por avances importantes en diversas áreas de la IA como aprendizaje automático (*Machine Learning*, *ML*) y aprendizaje profundo (*Deep Learning*, *DL*). Una técnica dentro del ML es el aprendizaje por refuerzo (*Reinforcement Learning*, *RL*). El aprendizaje profundo por refuerzo (*Deep Reinforcement Learning*, *deep RL* o *DRL*) combina la capacidad para representar datos complejos del DL con la habilidad de optimizar acciones en base a funciones de recompensas del RL. Esta tecnología ha sido capaz de resolver problemas complejos que se consideraban fuera del alcance de los métodos tradicionales de IA.

2.1. Usos y Aplicaciones Recientes de la IA, ML y RL

La inteligencia artificial ha encontrado aplicaciones en una amplia gama de dominios. En la industria del entretenimiento, se utiliza para personalizar recomendaciones de contenido en plataformas como *Netflix* y *Spotify*. En el sector de la salud, se emplea para el diagnóstico de enfermedades a través del análisis de imágenes médicas y la predicción de brotes epidemiológicos. En el ámbito financiero, la IA se utiliza para el análisis de mercados y la detección de fraudes. Además, el aprendizaje por refuerzo ha mostrado su potencial en áreas que requieren la toma de decisiones estratégicas.

Más relacionado con el ámbito del proyecto a desarrollar, los juegos de estrategia, hay una serie de proyectos destacables.

Comenzando con *Deep Blue* de *IBM*, que en 1997 derrotó al campeón mundial de ajedrez, Garry Kasparov. Este evento destacó las capacidades de la IA para calcular y evaluar posiciones en ajedrez. Más recientemente, *AlphaGo* y su adaptación posterior, *AlphaZero*, desarrollados por *DeepMind*, revolucionaron el campo al utilizar un enfoque basado en redes neuronales profundas y *RL*, a diferencia de *Deep Blue*, que dependía en gran medida de una base de datos extensa de partidas. *AlphaZero* aprendió a jugar ajedrez, *Go* y *shogi* desde cero, jugando millones de partidas contra sí mismo y mejorando progresivamente. Estos logros demuestran el potencial del *Deep RL* para descubrir y perfeccionar estrategias complejas de manera autónoma y adaptativa en entornos dinámicos.

Con el crecimiento que ha ido recibiendo el uso de la Inteligencia Artificial con el paso del tiempo han surgido cada vez más herramientas para el desarrollo de sistemas de aprendizaje automático. Entre ellas, tiene relevancia la librería *Gymnasium*, orientada al desarrollo de sistemas de aprendizaje por refuerzo. *Gymnasium* ofrece funcionalidades para diseñar entornos de entrenamiento de agentes, por lo que presenta unas características enriquecedoras para el planteamiento del proyecto.

2.2. Objetivos del proyecto

Siguiendo por la línea de la aplicación de modelos de *Deep RL* para juegos de estrategia, el trabajo planteado consiste en entrenar un agente en el entorno de los combates de los juegos *Pokémon*, un videojuego de "estrategia por turnos" (o *turn-based strategy* en inglés) [1], en los que es muy importante planificar un turno para poder obtener resultados óptimos en la partida. Para esto se aprovechará el simulador en línea de *Pokémon Showdown*.

El entorno del juego consiste en un duelo entre 2 jugadores con 6 personajes cada uno. Los personajes de cada jugador se enfrentan 1 a 1; cuando la salud de un personaje llega a cero, este se considera debilitado. El jugador pierde la partida cuando todos sus personajes están debilitados.

Para lograr que en el mismo entorno exista la mayor cantidad de escenarios posibles se plantea enfrentar 2 jugadores (ambos controlados por el agente) con equipos elegidos aleatoriamente. De esta manera se pretende lograr que el agente verdaderamente aprenda a jugar en cada situación. Si no se incorporase una gran variabilidad del entorno se correría un riesgo de *overfitting* o sobreajuste, es decir, que el agente aprenda a resolver un entorno y no generalice las distintas situaciones que se puede encontrar en un combate de manera adecuada.

Este proyecto pretende alcanzar los siguientes objetivos:

- Diseñar un modelo de Inteligencia Artificial para lograr explorar el potencial del uso de IAs para los videojuegos de estrategia por turnos. El proyecto está orientado a los juegos de *Pokémon*. El proyecto se enfocará en el uso de inteligencias artificiales por refuerzo. El modelo *deep reinforcement learning* se entrenará mediante simulaciones contra sí mismo.

- Adquirir conocimientos teóricos y prácticos de las técnicas de *Deep RL*, aprendiendo sobre la implementación de algoritmos de aprendizaje automático y su aplicación en problemas complejos.
- Estudiar las funcionalidades de la librería *Gymnasium* con entornos clásicos para el aprendizaje de *DRL*, implementando también sistemas de aprendizaje por refuerzo en entornos clásicos.
- Implementar un escenario simplificado del entorno de los combates *Pokémon* en el que poder analizar el funcionamiento de los sistemas de aprendizaje por refuerzo.

3. Estado del arte

Como ya se había mencionado, el uso de redes de IA en juegos de estrategia por turnos ya ha sido explorado en varios proyectos destacables.
Deep Blue, desarrollado por IBM, logró derrotar a un campeón mundial de ajedrez, Garry Kasparov, en 1997. Aunque no utilizaba DRL, *Deep Blue* sentó las bases para la idea de que las máquinas podían competir y superar a los humanos en juegos de estrategia.

Al éxito de *Deep Blue* le siguió *AlphaGo*, que tenía un objetivo parecido, pero un desarrollo más sofisticado. *AlphaGo* estaba diseñado para el juego de mesa *Go*, y consiguió al igual que *Deep Blue*, vencer al campeón mundial de *Go*, Lee Sedol en 2016. En el aspecto técnico, *AlphaGo* poseía características más interesantes que Deep Blue, *AlphaGo* empleaba Redes Neuronales Convolucionales. Para evaluar el estado del juego y Árboles de Montecarlo para evaluar secuencias de movimientos y elegir la óptima. También se entrenó jugando contra versiones anteriores de sí mismo. Aunque *AlphaGo* sigue sin implementar *DRLs* algunas de sus características son muy relevantes en el contexto de las *DRLs*, como la importancia de evaluar el estado del juego, criterios de selección de acciones óptimas o autoentrenamiento [2].

El proyecto más importante de cara al uso de *DRLs*, es *Atari Games*, que popularizó el uso de *Deep Q-Networks*, para entrenar modelos capaces de jugar a juegos clásicos de la consola *Atari*. *Atari Games* fue capaz de superar el rendimiento en varios juegos. Este logro es muy significativo, ya que algunos de estos juegos requerían manejar grandes espacios de estados y una planificación a largo plazo óptima. Además de esto, *Atari Games* usa tecnologías más relevantes de cara a este proyecto. Las redes DQN son redes neuronales profundas que intentan aproximar la función Q, utilizada para estimar el valor que tiene elegir una acción en un momento determinado. Además, usaba técnicas como *Experience Replay* (grabando experiencias del entorno) y Redes de Destino (generando las predicciones del entorno) para poder estabilizar el entrenamiento [3].

Un proyecto aplicado a los juegos de *Pokémon*, desarrollado por *Rempton Games*, usaba varios agentes con distintas "personalidades" (determinadas por sus funciones de recompensa). Este proyecto también empleaba aprendizaje por refuerzo en combates *Pokémon*, por lo que se mantiene más cerca de la idea del proyecto planteado [4].

Existen también estudios que plantean el uso de agentes de aprendizaje por refuerzo en entornos de Pokémon. Por ejemplo, [5] que plantea el juego en combates Pokémon con sistemas como AlphaZero y entrenamiento en entornos de OpenAI Gym. Y el proyecto [6] que basado en las librerías OpenAI Gym y también Stable Baseline 3, utilizando un enfoque de aprendizaje por refuerzo, consigue realizar un agente bastante completo. Capaz de combatir con grandes resultados.

4. Marco teórico

4.1. Machine Learning

El *Machine Learning* es un área de las inteligencias artificiales que tiene como objetivo estudiar como inferir un conocimiento a partir de los datos provistos mediante el uso de sistemas artificiales sin necesidad de ser programado explícitamente para ello [7]. Según la definición formal de Mitchell un sistema ML: "aprende de la experiencia E respecto a un grupo de tareas T y una medida de rendimiento P en dichas tareas. El rendimiento P en las tareas T debería aumentar según aumente la experiencia E" [8].

Los sistemas de aprendizaje automático tienen la capacidad de aprender de los datos, en contraposición con los sistemas de conocimiento experto, que se rigen por una serie de heurísticas dependientes del dominio [9]. Esta característica permite simplificar enormemente el desarrollo de sistemas artificiales de decisión. En los sistemas de aprendizaje automático se depende menos de la intervención humana (generalmente lenta) y permite generalizar reglas que podrían no estar claras o tener una definición formal compleja, ya que las reglas las encontraría la máquina en su proceso de entrenamiento.

Dentro del área del aprendizaje automático se pueden distinguir 3 tipos principales:

- El aprendizaje supervisado construye algoritmos que relacionan los datos de entrenamiento con sus salidas.
- El aprendizaje no supervisado trata de reconocer patrones en los datos que permitan predecir el resultado.
- El aprendizaje por refuerzo (será el más importante para este proyecto) aprende por entrenamiento de prueba y error ante los datos de entrenamiento.

4.2. Reinforcement Learning

Mediante el uso de sistemas de aprendizaje por refuerzo, se pretende lograr que un agente sea capaz de realizar las acciones óptimas en un entorno dinámico maximizando su desempeño. Esta tarea generalmente es abordada usando funciones de recompensas, que "premian" al modelo de IA si sus acciones se acercan al resultado deseado, pero lo "castigan" si se aleja del resultado. En los

sistemas de *Reinforcement Learning*, el modelo es entrenado por el siguiente proceso con un ciclo de interacción entre agente y entorno:

Observación del estado: El agente, toma los datos del estado actual del entorno.
Selección de la acción: Siguiendo la política de establecida, el agente escoge una acción al azar.
Transición de estados: El entorno se actualiza de acorde a la acción realizada por el agente.
Recepción de la recompensa: se comparan los datos de los dos estados del entorno para determinar la recompensa que recibirá el agente.
Actualización de la política: Las políticas del agente se actualizan de acuerdo con la recompensa recibida.

Este proceso se inspira en los procesos de decisión de Márkov (MDP). El fundamento principal de los MDP es que cumplan la propiedad de Márkov. La propiedad de Márkov establece que el estado futuro de un proceso estocástico carece de memoria, lo que significa que sólo depende del estado actual y no de los anteriores. Expresado formalmente de la siguiente manera:

$$P(s_{t+1} | s_0, s_1, \dots, s_t) = P(s_{t+1} | s_t)$$

Figura 1. Propiedad de Márkov.

Respecto a los elementos de un proceso de Márkov consta de un conjunto con un espacio de estados (S), un espacio de acciones (A), la recompensa (R) y una función de transición entre estados (P).

El objetivo de utilizar un proceso de Márkov es encontrar la política (π) capaz de lograr un estado óptimo a largo plazo. Esta política debería indicar con que probabilidad debería seleccionarse cada una de las acciones posibles en cada estado posible.

Para obtener una política óptima, el sistema debe ser capaz de maximizar la función de recompensa total del sistema:

$$R_t = r_{\{t+1\}} + r_{\{t+2\}} + \dots + r_T$$

Figura 2. Función de recompensa total de un sistema de Márkov

La recompensa que recibe el agente de una sistema de aprendizaje por refuerzo debe representar la eficacia de la acción que ha tomado en una situación

determinada. Para ello se utilizarán recompensas positivas, negativas (también llamadas castigo o penalización) y nulas (que se usan para casos neutros). Es conveniente incorporar ambos tipos de recompensas simultáneamente, ya que usando sólo recompensas negativas el sistema evitará los casos desfavorables, pero sin buscar el óptimo, por lo que el resultado tendrá un rendimiento mínimo. Si por el contrario se usan sólo recompensas positivas el sistema podría aprender lento al no penalizar los errores cometidos.

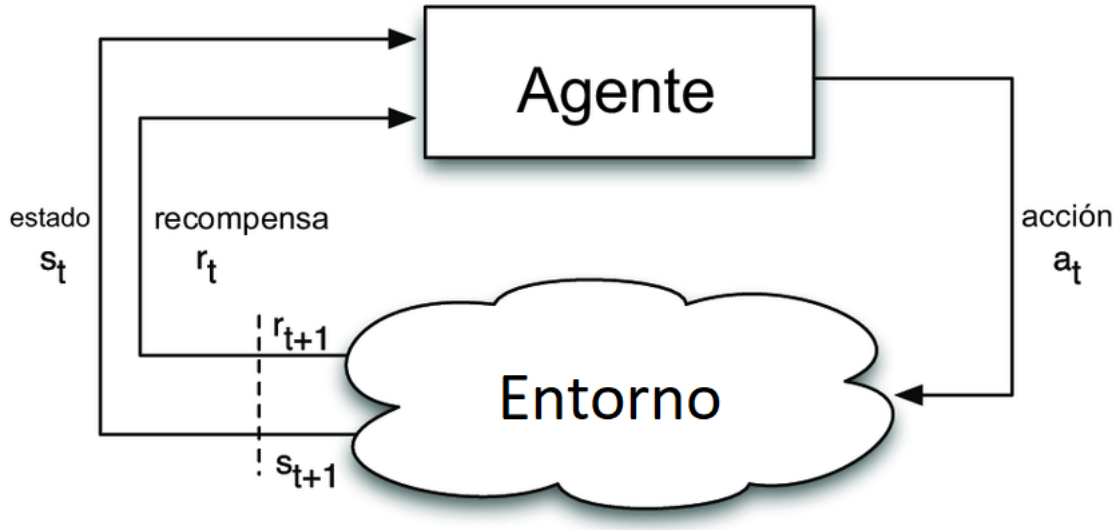

Figura 3. Diagrama de la interacción de los componentes en el entrenamiento de agentes por refuerzo

En un sistema RL el entorno es fundamental, ya que es donde el agente realiza acciones y recibe retroalimentación en forma de recompensas. El agente puede poseer un modelo del entorno (*model-based*), o basarse únicamente en la exploración del entorno (*model-free*). Mediante los algoritmos *model-based* el agente construye un modelo explícito del entorno, es decir, aprende las dinámicas de transición entre estados y las recompensas asociadas a esas transiciones. Una vez que el modelo está construido, el agente puede planificar sus acciones utilizando métodos como la búsqueda en el espacio de estados o la programación dinámica. Por otro lado, los algoritmos *model-free* no intentan modelar el entorno explícitamente. En su lugar, aprenden directamente la política óptima o la función de valor a partir de la experiencia acumulada mediante interacciones con el entorno. Aunque pueden ser más simples y robustos a errores de modelado, suelen requerir más datos de interacción para aprender eficazmente.

El aprendizaje por refuerzo destaca especialmente en casos donde no se conoce el modelo del entorno o es muy complejo de modelar. Siendo los algoritmos *model-free* los más eficaces en esta situación, al no requerir un modelo del entorno.

4.3. Deep Learning

El aprendizaje profundo o DL (del inglés *Deep Learning*) es uno de los campos más populares e importantes de la IA, los sistemas de aprendizaje por refuerzo utilizan RNAs (redes neuronales artificiales). El diseño de las redes neuronales surge de la idea de intentar crear inteligencia artificial en un esquema lo más parecido posible al de la inteligencia natural. En los seres humanos las neuronas están interconectadas entre sí y reciben impulsos por sus dendritas y envían otros impulsos a las neuronas que estén conectadas a su axón [10]. En el contexto de la computación, esta analogía se consigue plasmar con nodos (típicamente llamados neuronas) que reciben información de otras neuronas y crean, a partir de los valores de entrada, un nuevo valor que enviarán a la siguiente capa de neuronas. La salida de cada neurona se obtiene de la siguiente manera:

$$a = f(W \cdot P + b)$$

Figura 4. Función de salida de una neurona artificial.

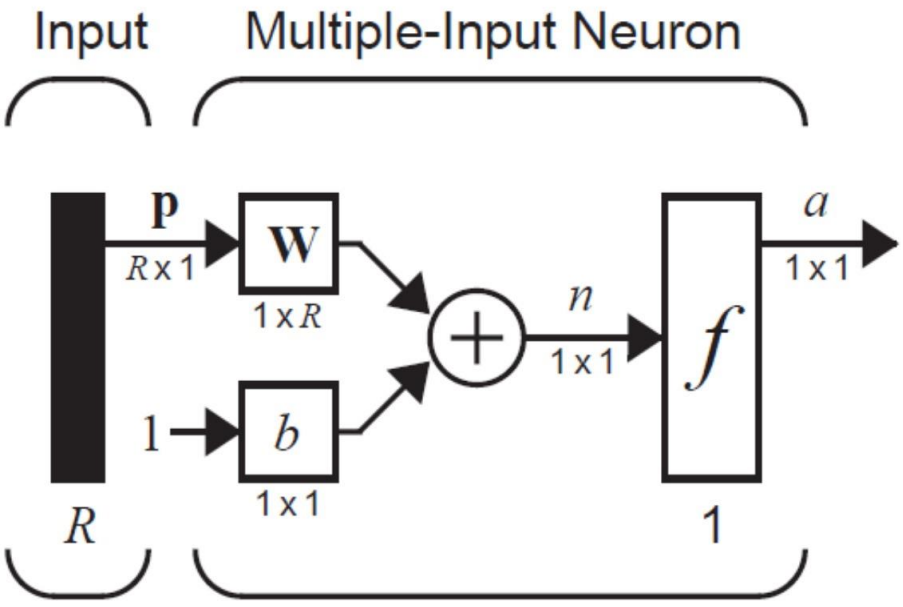

Figura 5. Representación de una neurona artificial de entrada múltiple.

Donde a es la salida de la neurona, f la función de activación. Que ajusta la salida de la neurona a los umbrales requeridos. W es el vector de pesos, de dimensiones 1xR, y P el vector de entradas, de dimensiones Rx1; R es el número de entradas de la neurona. Cabe destacar que, debido a las dimensiones de W y, el resultado de su producto será una matriz 1x1 que se interpreta como un número. La b, es el sesgo (o *bias*) de la neurona. Los pesos y el sesgo de la neurona se actualizan durante la fase de entrenamiento para lograr adaptarse a los datos. La función de activación puede variar dependiendo del contexto, algunos de los tipos más comunes son *linear*, *hard-limit* o *log-sigmoid*.

Las neuronas se organizan en capas de manera que la información se transmita desde las neuronas en la primera capa hasta la capa final. Los datos entran a la red neuronal por la capa de entrada (*input layer*), esta capa debe acomodarse al formato de los datos, y habrá una neurona por cada dato que se introduzca a la red. La salida de la *input layer*, se transmite hasta las capas ocultas (*hidden layer*), en esta capa tanto la entrada como la salida quedan dentro de la red. Una red neuronal puede llegar a contener varias capas ocultas aumentando la complejidad de la red, pudiendo mejorar el rendimiento en algunos casos. Aunque en muchos casos incrementar el número de capas ocultas o el número de neuronas en ellas mejora los resultados, ya que aporta un cálculo más sofisticado al problema; no siempre será conveniente, por el aumento que supone a la carga computacional y además el riesgo de sobreajuste. Por último, las salidas generadas por la capa oculta son interpretadas por la capa de salida, que se encarga de adaptar los datos al formato requerido por el resultado final. [11]

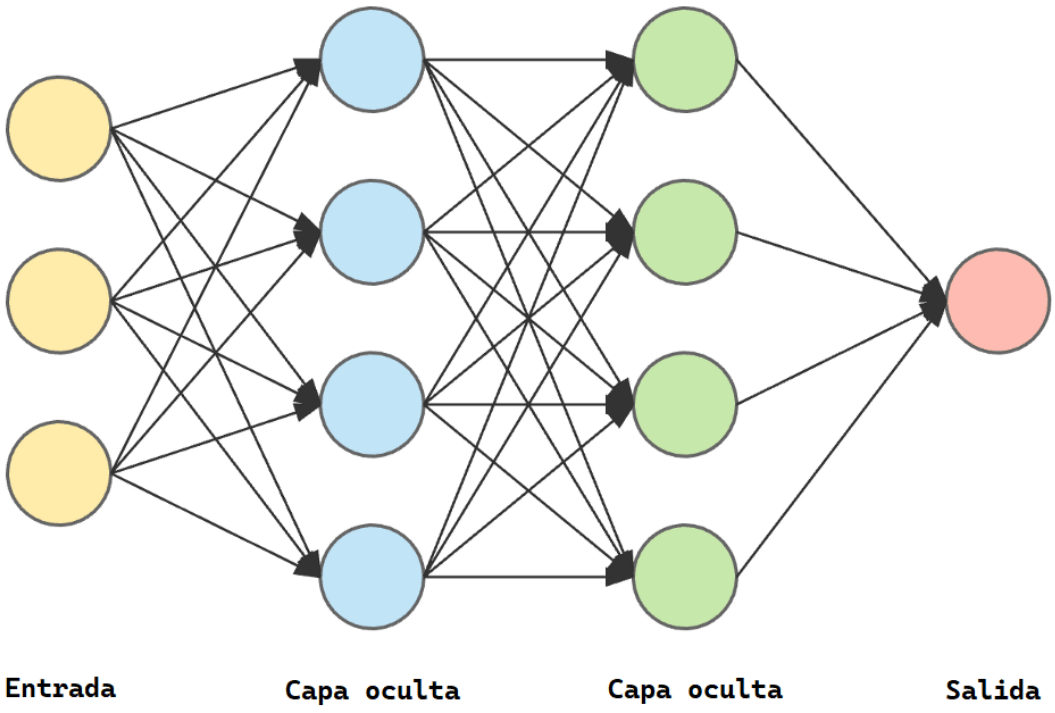

Figura 6. Diagrama de una red neuronal multicapa.

Las redes neuronales pueden ser de varios tipos que pueden ser útiles para diferentes aplicaciones. Las redes neuronales *feedforward* son las más básicas, donde la información se mueve en una sola dirección, hacia la capa de salida. Las redes neuronales convolucionales (*CNN*) se utilizan principalmente en el procesamiento de imágenes y videos, aprovechando su capacidad para reconocer patrones espaciales. Las redes neuronales recurrentes (*RNN*) son adecuadas para secuencias y series temporales, gracias a su capacidad para retener información a través de estados internos.

Para entrenar una red neuronal la actualización de pesos y sesgos en redes neuronales se realiza mediante un proceso llamado retropropagación (*backpropagation*). La retropropagación consiste en a partir de una métrica que mida el error, propagar ese error calculado desde el final de la red al inicio, para variar los pesos y *bias* de cada neurona respecto a ese error. La métrica de error típicamente usada es el error cuadrático medio (MSE). Modificando estos valores se pretende que la red aprenda de las entradas de los datos y consiga reducir el error medido. Para cumplir este objetivo se pueden utilizar varios algoritmos para actualizar los pesos, siendo el Descenso por Gradiente y su variante Descenso por Gradiente Estocástico los algoritmos más comunes. El

algoritmo descenso por gradiente, se basa en calcular la derivada de la función de pérdida en puntos determinados, con el fin de encontrar el punto donde el valor de la función de pérdida es mínimo absoluto. Uno de los problemas que puede tener este algoritmo es encontrar un mínimo local, y que no consiga reducir el error cometido a partir de un umbral. La ventaja principal que incorpora el SGD es inyectar ruido en la actualización de pesos, para evitar caer en mínimos locales de la función de pérdida.

4.4. Generalización y *overfitting*

Uno de los problemas más graves que pueden ocurrir durante el entrenamiento de un sistema de aprendizaje automático es el sobreajuste (u *overfitting*, por su nombre en inglés). El *overfitting* se da cuando una red neuronal minimiza demasiado la función de pérdida para cierto conjunto de datos de entrenamiento y adapta los pesos de la red para conseguir distinguir sólo esos datos. Si esto llega a ocurrir, cuando la red se exponga a datos nuevos no será capaz de obtener la salida correcta. Para evitar el sobreajuste, es importante utilizar una cantidad de datos muy amplia y heterogénea, además de evitar entrenar demasiado la red. Otras técnicas que se podrían incorporar para evitar *overfitting* son regularización L2 (evitar pesos de neuronas grandes), *Dropout* (anular algunas neuronas en ocasiones durante el entrenamiento), *early stopping* (detener el entrenamiento si el rendimiento empeora), o validación cruzada (entrenar el modelo con los datos divididos en varias partes). Estas características son cruciales para poder obtener una red correctamente generalizada.

4.5. Algoritmos *off-policy* y *on-policy*

En el aprendizaje por refuerzo, los algoritmos se clasifican en *off-policy* y *on-policy* según la manera en que aprenden y actualizan sus políticas. Los algoritmos *off-policy* aprenden una política óptima usando datos generados por una política diferente, llamada política de comportamiento. Esto permite que la política de comportamiento explore el entorno mientras la política objetivo busca ser óptima. La principal ventaja de los algoritmos *off-policy* es que permiten mayor exploración y pueden utilizar experiencias previas almacenadas. Esto mejora la eficiencia en el uso de datos y facilita el aprendizaje a partir de simulaciones o registros históricos, lo que es especialmente útil en entornos donde la recopilación de nuevos datos puede ser costosa o lenta.

Por otro lado, los algoritmos *on-policy* aprenden la política óptima evaluando y mejorando la misma política que usan para interactuar con el entorno. Esto significa que la política de comportamiento es la misma que se optimiza. La ventaja de los algoritmos *on-policy* radica en su simplicidad y su capacidad para adaptarse mejor a entornos dinámicos, ya que la política aprendida siempre refleja la política utilizada durante la interacción. Esto asegura una coherencia entre la política aprendida y la política aplicada, lo que puede ser crucial en entornos donde las condiciones cambian rápidamente y la política debe ajustarse en consecuencia.

4.6. Explotación y exploración

En el contexto del aprendizaje por refuerzo, la diferencia entre los conceptos de explotación y exploración son fundamentales para el desarrollo de políticas de toma de decisiones óptimas.

La exploración consiste en ampliar el conocimiento del agente sobre el entorno en el que opera. Este proceso implica que el agente pruebe nuevas combinaciones de estado-acción, lo que puede aumentar las posibilidades de obtener un posible beneficio a largo plazo. Al explorar, el agente busca descubrir recompensas más valiosas que no se habrían encontrado si solo se centrara únicamente en las acciones conocidas. En esencia, la exploración permite al agente mejorar su comprensión del entorno, identificando oportunidades y estrategias que podrían proporcionar mayores recompensas a largo plazo.

Por otro lado, la explotación refiere al uso exhaustivo de las combinaciones estado-acción conocidas, que tengan una alta recompensa esperada. La explotación se logra aumentando las recompensas a corto plazo que recibe el agente. El problema que tiene la explotación es que es muy sensible a aprender de recompensas subóptimas, que podría limitar encontrar la combinación de acciones más beneficiosas a largo plazo [12].

Para lograr un agente eficiente es importante encontrar el equilibrio adecuado entre exploración y explotación. Es decir, un comportamiento que descubra suficientes respuestas estado-acción diferentes y que consiga ponerlas a prueba lo suficiente como para determinar si son buenas o no. Algoritmos como *ε-greedy* incorporar una elección aleatoria entre explorar y explotar en cada iteración. Esta elección está regulada por la probabilidad ε que determina la probabilidad de explorar, y *1-ε* la probabilidad de explotar. Con el valor de épsilon se puede

controlar el equilibrio entre exploración y explotación. Algunas variantes del algoritmo *ε-greedy* pueden variar el valor de ε durante la ejecución. Esta característica permite realizar mucha exploración al principio del entrenamiento (cuando es más necesario descubrir nuevas acciones), y según avanza el entrenamiento aumentar la explotación para determinar cuál de las combinaciones de acciones más beneficiosas es la óptima [13]

4.7. Modelos de recompensa

En el aprendizaje por refuerzo, existen dos enfoques principales para establecer la función o modelo de recompensa: proporcionar *feedback* solo cuando se alcanza un estado deseado o hacerlo continuamente en cada transición de estados.

La recompensa dispersa (*sparse reward*) está presente en sistemas en los que el agente sólo recibe una recompensa al alcanzar el estado deseado, generalmente un estado terminal. Por ejemplo, en un juego de mesa o de estrategia, el agente podría recibir +1 punto por ganar la partida y -1 punto por perderla. Este método presenta el problema de que, si las recompensas son aplicadas tras alcanzar el estado terminal, dificulta que el agente sea capaz de identificar las acciones que contribuyeron a alcanzar el estado deseado. Una alternativa para resolver este problema es la incorporación de *reward-shaping*, que consiste en añadir recompensas auxiliares en algunos estados intermedios, que puedan acercarse al objetivo del estado terminal deseado. Esta técnica tiene la desventaja de que esas recompensas intermedias son modeladas por humanos, lo que puede introducir sesgos y restringir la capacidad del agente de explorar estrategias.

Por otro lado, la recompensa densa (o *dense reward*) proporciona *feedback* al agente en cada transición entre estados. Esto aporta al agente información directamente sobre la utilidad que tienen las acciones elegidas. Con este modelo de recompensa, se consigue que el agente logre aprender de manera más ágil. Sin embargo, en algunos entornos es difícil diseñar funciones de recompensa continuas; además, de la misma forma que con el *reward-shaping*, se puede introducir un sesgo humano. Estas características pueden limitar la capacidad de aprendizaje del agente.

4.8. *DQN*

El algoritmo *Deep Q-Network* (*DQN*) es una de las técnicas más importantes del aprendizaje por refuerzo que combina *Q-learning* con redes neuronales profundas para manejar problemas con espacios de estado y acción grandes y continuos. *DQN* fue desarrollado por *DeepMind* [14] para el proyecto *Atari Games*.
Las DQN se inspiran en el funcionamiento del algoritmo *Q-learning*, que trata de aprender una función de valor *Q (s, a)* que estima la recompensa esperada de tomar una acción a en un estado s. La actualización de *Q-learning* se basa en la ecuación de Bellman:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma\, max_{a'}\, Q(s',a') - Q(s,a)]$$

Figura 7. Ecuación de Bellman

Donde a es la tasa de aprendizaje, r es la recompensa recibida, γ es el factor de descuento y *s'* es el nuevo estado.

El uso de redes neuronales junto con el algoritmo *Q-learning* permite aprender una aproximación de la función *Q(s,a)* en lugar de registrar cada par de estado y acción. Esto consigue reducir enormemente la cantidad de datos almacenada por el modelo, sobre todo en espacios de estados muy grandes, en los que este algoritmo consigue un gran desempeño. Además de esto, las *DQN* también pueden generalizar la función para predecir pares de estado-acción visitados si la aproximación de la función Q consigue generalizar adecuadamente.

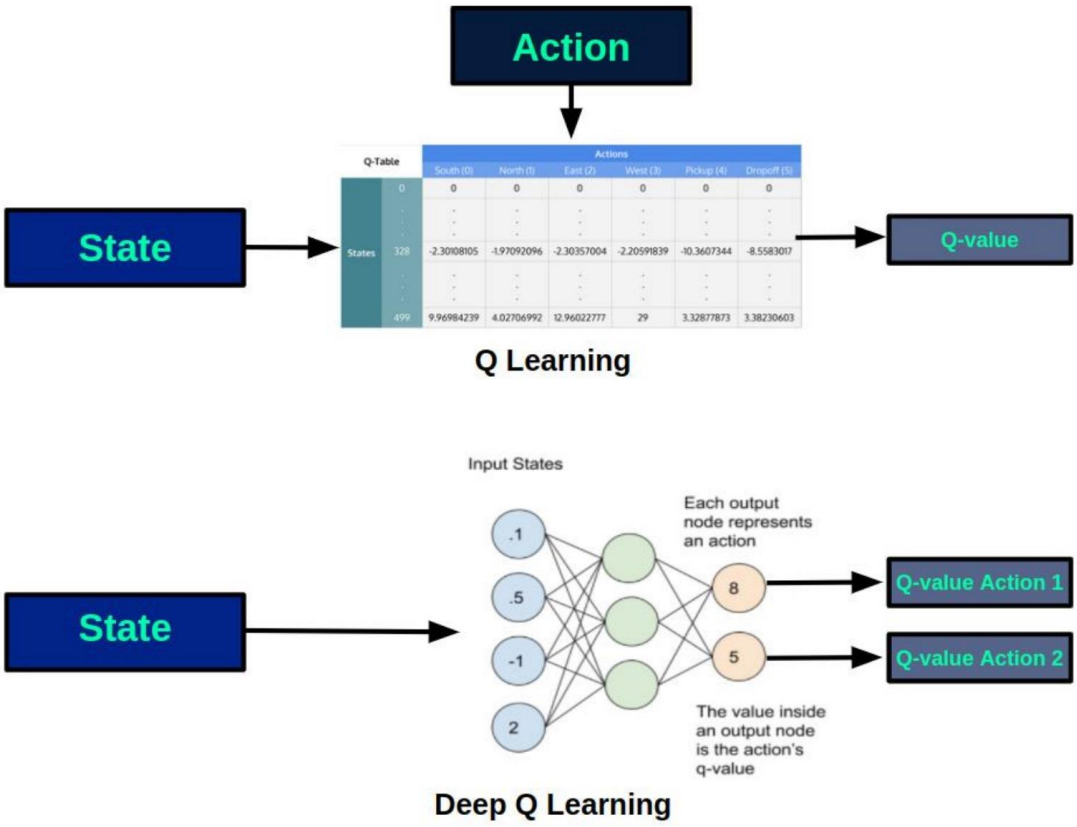

Figura 8. Diagrama comparativo entre los algoritmos Q-Learning y Deep Q-Learning.

Una de las principales innovaciones de *DQN* es el uso de una memoria de repetición de experiencias. En lugar de actualizar los pesos de la red neuronal con cada nueva experiencia en secuencia, *DQN* almacena estas experiencias en una memoria. Esta memoria contiene transiciones de la forma *(s, a, r, s')*, que son posteriormente muestreadas aleatoriamente para entrenar la red. Este enfoque tiene varios beneficios. En primer lugar, rompe la correlación entre experiencias consecutivas, lo que conduce a una mayor estabilidad y eficiencia en el entrenamiento. Al entrenar con un conjunto más diverso de experiencias, el agente puede aprender representaciones más robustas del entorno. Además, la repetición de experiencias permite un uso más eficiente de los datos, ya que cada experiencia puede ser reutilizada múltiples veces durante el entrenamiento. También incorpora una red neuronal adicional llamada red de objetivo (o *target network* en inglés). En el *Q-learning* tradicional, se utiliza la misma red neuronal para estimar tanto los valores actuales como los futuros, lo que puede llevar a inestabilidad y oscilaciones en el entrenamiento. DQN resuelve este problema utilizando dos redes neuronales: la red principal y la red de objetivo. La red

principal se encarga de seleccionar las acciones, mientras que la red de objetivo se utiliza para proporcionar los valores Q necesarios para la actualización.

5. Herramientas utilizadas

5.1. *Pokémon Showdown*

Pokémon Showdown es una plataforma en línea que permite a los usuarios simular combates de *Pokémon* en tiempo real. Este simulador ofrece la posibilidad de probar el funcionamiento de los combates *Pokémon* desde el navegador sin necesidad de utilizar los juegos o las consolas originales.

Pokémon Showdown cuenta con una gran base de datos donde almacena la información referente a las especies de *Pokémon*, movimientos, habilidades, objetos y otros disponibles en el juego; datos que son útiles a la hora de determinar el curso del combate. Esta información se actualiza regularmente con el fin de reflejar los cambios más recientes que presentan los últimos juegos oficiales de *Pokémon*. *Pokémon Showdown* permite a sus usuarios crear equipos personalizados eligiendo entre todos los *Pokémon* disponibles, configurando sus estadísticas, habilidades y movimientos para adaptarse a diferentes estrategias de combate. Esta flexibilidad permite recrear casi cualquier situación posible de combate.

Pokémon Showdown ofrece una gran variedad de formatos de juego, en los que puede variar los personajes permitidos, la cantidad de éstos que se usan en el combate, y otros detalles del reglamento, estando disponibles modos de juego traídos de las diferentes generaciones de los juegos de *Pokémon*. La plataforma también cuenta con un sistema de clasificaciones que ordena a los jugadores en un *ranking* según su desempeño en combates.

La herramienta *Pokémon Showdown* proporciona un entorno de simulación donde se pueden probar los modelos de ML generados para el proyecto. Con su uso se pretende lograr:

Recopilación de datos: De las batallas simuladas en *Pokémon Showdown* será necesario recopilar datos para poder entrenar los modelos de DL. Estos datos, deberán contribuir a definir los elementos de Márkov: estados, acciones, recompensas y transiciones entre estados.

Definición de Estados y Acciones: En un combate Pokémon, el estado puede incluir información sobre los Pokémon activos, sus puntos de salud, habilidades, movimientos disponibles, así como el estado del entorno (como

efectos de clima o trampas en el campo). Las acciones comprenden los movimientos seleccionados, los cambios de Pokémon o el uso de objetos. Es importante decidir qué información conocerá el agente, ya que esto determinará que patrones será capaz de identificar.

El modo de uso típico de *Pokémon Showdown*, es a través de su servidor principal, este servidor permite entablar combates con jugadores de todo el mundo; sin embargo, *Pokémon Showdown* también ofrece la opción de instanciar un servidor local, permitiendo a los usuarios jugar en un ámbito segregado del resto de jugadores. Esto es especialmente útil para proyectos de desarrollo de agentes, ya que el servidor principal tiene limitaciones en su uso para evitar sobrecargas. Eliminar la dependencia en servidores externos consigue evitar este inconveniente. Para instanciar un servidor local, primero se debe clonar el repositorio oficial de *Pokémon Showdown* desde *GitHub* y luego instalar las dependencias necesarias utilizando *Node.js*. El servidor local puede ser ejecutado en un entorno propio, permitiendo la simulación y el entrenamiento de agentes en combates *Pokémon* sin necesidad de conexión a internet. Este enfoque asegura una mayor capacidad de experimentación al tratarse de un entorno controlado [15].

5.2. *Poké-env*

Para poder aprovechar la herramienta de *Pokémon Showdown*, es necesaria una interfaz encargada de la interacción entre algoritmos de inteligencia artificial y el entorno de combate de *Pokémon*. Para esto, se utilizará *Poké-env*, una librería de *Python* que incorpora varias utilidades para facilitar el diseño de agentes que combatan en *Pokémon Showdown*. *Poké-env* permite una integración sencilla con el entorno de batalla, proporcionando funciones para gestionar las acciones del agente y la comunicación con el servidor.

Con la biblioteca *Poké-env*, los agentes se crean como instancias de clases hijas de la clase *Player*. Esta clase permite definir la política que seguirá el agente durante los combates sobrecargando el método *choose_move()* en el que se debe desarrollar la lógica necesaria para que el agente seleccione una acción en cada turno. Este método recibe un objeto de tipo *AbstractBattle* el cual representa el estado actual del combate. Para un agente que utilice aprendizaje por refuerzo será necesario crear el espacio de observaciones a partir de los datos de la batalla recibidos por este objeto, para ello se sobrecargará el método

embed_ battle(), para que convierta los datos de la batalla en un vector que representa la observación del entorno [16].

5.3. *Gymnasium*

Gymnasium es una librería de *Python* dedicada al desarrollo y evaluación de algoritmos de aprendizaje por refuerzo. También posee una *API* para comunicar los algoritmos de aprendizaje con los entornos, además de un set de entornos ya fabricados. *Gymnasium* surge como un *fork* de la librería *Gym*, estas dos librerías se han convertido en protagonistas a la hora de desarrollar experimentos con RL.

La mayor parte del trabajo con *Gymnasium* se destina a la clase *Env*, que representa el entorno de simulación. Esta clase presenta varios atributos y métodos importantes:

- *action_ space* (modelo de acciones): este atributo determina el espacio de acciones que puede realizar el agente para hacer avanzar el estado del entorno. Sobre este atributo se puede utilizar la función *sample*() que muestrea una de las acciones disponibles de manera aleatoria.

- *observation_ space* (modelo de observaciones): establece el formato del espacio de observaciones, que puede ser discreto o continuo. Es un atributo esencial que define los tipos de datos y los rangos de valores que el agente puede percibir del entorno.

- *reward_ range* (rango de recompensas): este atributo define los límites mínimo y máximo de las recompensas que pueden asignarse en el entorno. Por defecto, su valor es $[-\infty, +\infty]$, lo cual es más informativo que funcional, ya que no siempre es fácil determinar estos límites en un experimento.

- *step()*: esta función avanza el estado del entorno según la acción especificada y devuelve una tupla con cuatro elementos:

  - *observation* (observaciones): un vector que proporciona información relevante sobre el entorno al agente, acorde al formato establecido en el *observation_ space*.

  - *reward* (recompensa): la recompensa inmediata obtenida por realizar una acción en el estado actual, expresada como un número real.

- *done* (finalizado): un valor booleano que indica si el episodio ha terminado, es decir, si el agente ha alcanzado un estado terminal.

- *info* (información): un diccionario que puede incluir información relevante sobre el entrenamiento, útil para logs, depuración o diagnósticos. Esta información no es accesible para el agente durante el entrenamiento.

- *reset()*: reinicia el estado del entorno, restaurando todos los campos a sus valores iniciales. Permite introducir una semilla para inicializar el generador de números aleatorios del entorno. Este generador es opcional, y se puede usar un sistema propio si es necesario. Además, si el entorno es configurable, acepta un diccionario de configuración. En este experimento, se reinicia el entorno según la semilla y se registran las métricas del episodio para análisis posterior. Este método devuelve una tupla con dos elementos:

- *observation* (observaciones): la observación inicial del agente en el estado inicial, con el mismo formato que el *observation_space*.

- *info* (información): similar al *info* devuelto por el método *step()*.

- *render()*: representa el entorno y el agente durante el entrenamiento. Este método admite varios modos de representación:

- *human*: para ser renderizado en una pantalla visualizable por un humano. De la misma manera que se suelen representar las imágenes de un videojuego.

- *rgb_array*: devuelve una matriz de dimensiones WxHx3, donde cada submatriz WxH representa el valor de color para cada píxel en rojo (R), verde (G) y azul (B) entre 0 y 255.

- *ansi*: para mostrar en una terminal, devuelve una cadena de caracteres que representa el estado observable, puede incluir caracteres de escape para colores y saltos de línea.

[17]

A continuación, se describe un ejemplo de cómo se puede utilizar *Gymnasium* para entrenar un agente en el entorno *Cart-Pole*. Un entorno simple en el cual, el objetivo es mantener un poste en posición vertical controlando un carro que lo sujeta:

Espacio de Acciones

El espacio de acciones es un array de tipo *ndarray* con forma (1,) que puede tomar los valores {0, 1}, indicando la dirección de la fuerza fija aplicada al carrito:
0: Empujar el carrito hacia la izquierda.
1: Empujar el carrito hacia la derecha.

Espacio de Observaciones

El espacio de observaciones es un array de tipo ndarray con forma (4,) cuyos valores corresponden a las siguientes posiciones y velocidades:

Núm	Observación	Mínimo	Máximo
0	Posición del carrito	-4.8	4.8
1	Velocidad del carrito	-∞	∞
2	Ángulo del poste	-0.418... rad (-24°)	0.418... rad (24°)
3	Velocidad angular del poste	-∞	∞

Figura 9. Tabla de datos del espacio de observaciones del entorno Cart-pole de Gymnasium.

Recompensas

Dado que el objetivo es mantener el poste en posición vertical durante el mayor tiempo posible, se asigna una recompensa de +1 por cada paso dado, incluyendo el paso final [18].

Ejemplo de visualización:

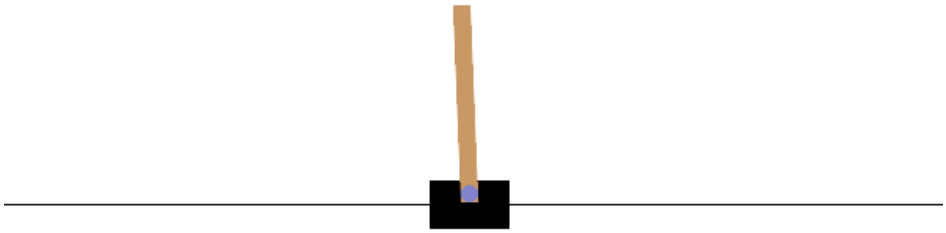

Figura 10. Animación del entorno Cart-pole de Gymnasium.

5.4. *PyTorch*

PyTorch es una biblioteca diseñada para entrenar modelos de aprendizaje automático.
PyTorch ha ganado rápidamente popularidad tanto en la comunidad académica como en la industria debido a su flexibilidad, eficiencia y facilidad de uso. A continuación, se detallan sus características y ventajas principales:

5.4.1. Características Principales

Tensor Computation:
PyTorch proporciona estructuras de datos multidimensionales llamadas *tensors*, que son similares a los *arrays* de *NumPy*. Sin embargo, a diferencia de los *arrays* de *NumPy*, los *tensors* de *PyTorch* permiten realizar operaciones en *GPU* para acelerar significativamente los cálculos, lo que es crucial para el entrenamiento de modelos de aprendizaje profundo.

Soporte para Paralelización:
Para entrenar modelos en grandes conjuntos de datos y aprovechar múltiples *GPUs*, *PyTorch* incluye herramientas para la paralelización y la distribución de

datos y modelos. Esto facilita el escalado de proyectos de aprendizaje profundo para manejar grandes volúmenes de datos y modelos complejos.

Construcción de Redes Neuronales Dinámicas:
Una de las características más destacadas de *PyTorch* es su enfoque dinámico para la construcción de redes neuronales, conocido como definición por ejecución (*define-by-run*). Esto permite construir y modificar redes neuronales de manera flexible y sobre la marcha, y es especialmente útil para el entrenamiento de modelos iterativos que aprendan a base de experimentar el entorno, pues se pueden mostrar gráficas del aprendizaje del agente mostrando su evolución a lo largo de los episodios del entrenamiento.

5.4.2. Funciones Importantes en *PyTorch*

nn.Sequential:
nn.Sequential es un contenedor en *PyTorch* que permite la construcción de modelos de redes neuronales de una manera sencilla y organizada. Este contenedor facilita la secuenciación de capas o módulos que componen la red neuronal, ejecutándolos en el orden en que se añaden al contenedor. Es especialmente útil para arquitecturas de redes que son puramente secuenciales, es decir, donde los datos fluyen de una capa a la siguiente sin bifurcaciones ni combinaciones complejas de capas.

Se muestra un ejemplo de una red neuronal básica creada extendiendo la clase *Sequential*:

```python
import torch.nn as nn

# Definir una red secuencial
model = nn.Sequential(
    nn.Linear(10, 20),    # Capa lineal de 10 a 20 neuronas
    nn.ReLU(),            # Función de activación ReLU
    nn.Linear(20, 10),    # Capa lineal de 20 a 10 neuronas
    nn.ReLU(),            # Otra función de activación ReLU
    nn.Linear(10, 1)      # Capa lineal final de 10 a 1 neurona
)
```

Figura 11. Fragmento de código para crear una red neuronal en PyTorch.

model.train():
Esta función es la función básica que se utiliza para entrenar una red neuronal de *PyTorch* a partir de unos datos determinados. Durante el entrenamiento, la red actualizará sus pesos según el optimizador que esté utilizando.

model.test():
La función *test()* calcula la salida del modelo ya entrenado frente a unos datos de entrada. Esta función permite evaluar el modelo entrenado en base a los resultados que genera.

Funciones de Activación:
PyTorch proporciona una variedad de funciones de activación en el módulo *torch.nn.functional*. Algunas de las funciones más relevantes son:

- *ReLU* (*Rectified Linear Unit*): Es una función lineal rectificada que devuelve cero para valores negativos y el mismo valor para valores positivos.
- *Sigmoid*: Convierte los valores de entrada en un rango de (0,1), útil para problemas de clasificación binaria donde se necesita una salida de probabilidad.
- *Tanh* (Tangente hiperbólica): Similar a la función *sigmoid*, pero con un rango de salida de $(-1,1)$. Es útil en redes neuronales para normalizar datos.

Optimizadores:
PyTorch proporciona optimizadores estándar como *SGD* y *Adam*, entre otros. Estos optimizadores permiten ajustar los pesos de la red neuronal durante el

entrenamiento de acuerdo con el valor obtenido de la función de pérdida para optimizar la convergencia del modelo.

Criterios de Pérdida:
Para calcular la pérdida durante el entrenamiento, *PyTorch* ofrece varios criterios de pérdida como *nn.CrossEntropyLoss*, *nn.MSELoss*, etc.

Entrenamiento y Evaluación:
Los bucles de entrenamiento y evaluación se definen explícitamente, lo que proporciona una mayor transparencia y control sobre el proceso de entrenamiento. Esta característica permite que *PyTorch* sea flexible para entrenar distintos tipos de modelos de aprendizaje automático.

6. Metodología

Para la realización del proyecto, se ha configurado un servidor local de *Pokémon Showdown* donde probar el agente. Este entorno permite ejecutar simulaciones de manera ágil y continua.

Para poder hacer uso del servidor, se ha diseñado un agente heurístico, para verificar la conectividad con el servidor de simulaciones. El uso de un agente basado en reglas simples predefinidas, ha servido como una primera etapa para asegurar que la comunicación entre el agente y el servidor de *Pokémon Showdown* funcione correctamente.

Antes de entrenar el agente de combates Pokémon, se han probado entornos más sencillos facilitados por *Gymnasium*, en los que se desarrollan las técnicas de aprendizaje por refuerzo de forma más controlada. Estos entornos permiten experimentar con distintas configuraciones, facilitando la identificación de estrategias eficaces antes de aplicarlas a un entorno más complejo como *Pokémon Showdown*. Además, los entornos más sencillos han servido para analizar los resultados obtenidos en esos entornos, sirviendo como un conocimiento previo de cara al desarrollo futuro.

Una vez probados los entornos de *Gymnasium*, se intenta acercar el planteamiento al problema de los combates *Pokémon*. Para esto, se ha implementado un entorno simplificado que simula las batallas de *Pokémon* para poder probar así la capacidad de aprendizaje de un agente.

Aunque el uso de *DRL* para *Pokémon Showdown* fue planteado inicialmente como el objetivo principal, una de las dependencias necesarias para este enfoque no funcionaba correctamente. En concreto, el error ocurría al importar módulos de *Keras* desde *Tensorflow*. A pesar de probar varias soluciones y versiones de los paquetes, el problema persistió. Resolver este error habría requerido un tiempo considerable de desarrollo adicional para resolverlo. Por lo tanto, aunque la metodología de *DRL* fue estudiada y planificada, su implementación completa no fue posible en la cota temporal del proyecto.

7. Desarrollo

7.1. Instanciación del servidor local

El primer paso del desarrollo del proyecto ha sido instanciar un servidor local de Pokémon Showdown, el cual está planteado para ejecutar las simulaciones de los agentes diseñados. Como ya se mencionó anteriormente, se escoge la opción de utilizar un servidor local en lugar de uno público debido a limitaciones de recursos.

Para instanciar el servidor local, se debe clonar el repositorio oficial desde *GitHub*. Es necesario contar con *Node.js* instalado previamente, ya que el servidor se ejecuta en este entorno. Después es necesario instalar las dependencias con el comando `npm install`; con esto, ya se podría iniciar el servidor con `node pokemon-showdown`. Esto permite ejecutar combates de manera autónoma y controlar el entorno, facilitando pruebas y desarrollo sin depender del servidor principal, lo que es crucial para evitar limitaciones y garantizar un entorno personalizado para el entrenamiento de agentes.

7.2. Diseño de agentes con *Poké-env*

Una vez está disponible el servidor local, se crea un agente encargado de seleccionar las acciones óptimas en las batallas *Pokémon*. Como ejemplo básico, se implementa un agente que recibe del entorno los valores de potencia de los movimientos disponibles y selecciona el de mayor potencia, intentando infligir en cada turno el máximo daño posible. Esto se hace sobrecargando el método *choose_move()* de la clase *Player* de *Poké-env* para que escoja la acción en base a su potencia.

El agente diseñado se puede conectar al servidor para que juegue en combates. A continuación, se muestra una animación en la que se ve el agente funcionando en un servidor propio de *Pokémon Showdown*:

Figura 12. Animación del agente jugando un combate en servidor local.

También se puede comprobar la efectividad de este agente, en este caso se prueba contra otro agente que escoge las acciones de manera aleatoria.

```
Max damage player won 93 / 100 battles [this took 16.583915 seconds]
```

Figura 13. Salida de la evaluación de dos agentes enfrentados.

En este caso probando con 100 batallas el agente de daño máximo ha ganado al agente de selección aleatoria en 93 de ellas. El criterio de mayor daño a infligir es un buen punto de partida, pero es una estrategia que puede mejorarse bastante. Para mejorar las decisiones del agente, se podría utilizar una red neuronal que aprenda a predecir la acción óptima en cada estado del juego.

7.3. Diseño de agentes de aprendizaje por refuerzo

Para probar el aprendizaje por refuerzo utilizando *Gymnasium* se han utilizado varios entornos en los que probar agentes entrenados por medio del algoritmo *DQN*. La implementación se lleva a cabo utilizando las utilidades de la biblioteca *PyTorch* para diseño de redes neuronales. El entorno de *Cliff Walking* presente en la propia librería de *Gymnasium*, el cual consiste en un pequeño juego en el que el personaje debe moverse hasta el final sin caer por acantilados; el entorno de *Frozen Lake*, similar al anterior, pero añadiendo dificultad por el riesgo de que el personaje resbale por el hielo; y además, un entorno de batallas

Pokémon con características simplificadas. Desarrollado como parte del proyecto mediante el uso de las funcionalidades de la librería *Gymnasium* para el diseño de entornos.

7.4. *Cliff Walking*

El entorno *Cliff Walking* representa un pequeño juego que toma lugar en un tablero de 4x12 casillas. El jugador empieza en la casilla [3, 0] y debe desplazarse hasta la casilla [3, 11]. Las casillas [3, 1 ... 10] representan acantilados por los que el jugador se caerá si se sitúa sobre ellos, haciéndole volver a la casilla de inicio. Un intento permanece activo mientras el jugador no alcance la meta, por lo que si el jugador cae por un acantilado seguirá en el mismo intento, pero tendrá que repetir el progreso desde el principio.

Espacio de acciones:
Para controlar al personaje en el juego, hay 4 acciones discretas muy básicas de movimiento. Cada acción está asociada a un número, para simplificar la comunicación del agente con el entorno.
0: Movimiento hacia arriba
1: Movimiento hacia la derecha
2: Movimiento hacia abajo
3: Movimiento hacia la izquierda

Espacio observable:
El único estado que se puede recibir del entorno es la posición del personaje. Como el juego se acopla en una malla bidimensional, el espacio de estados es discreto. Las dimensiones de la malla son 4x12. La cantidad de estados observables sería 48, pero existen algunas restricciones: El personaje no puede estar en un acantilado, pues al pisar un acantilado, el personaje es desplazado instantáneamente hasta la posición inicial. Hay 10 casillas de acantilado en el entorno que no son un estado observable posible. Además, llegar a la casilla final implica la compleción del juego, por lo que el agente debe dejar de actuar. Teniendo estas restricciones en cuenta, el entorno tiene 37 estados distintos, que dependen de la posición del personaje. El estado de la posición se representa con un número que será el resultado de la operación:

$$current\_row \cdot ncols + current\_col$$

Figura 14. Expresión del cálculo del estado del entorno Cliff Walking.

En este caso *ncols*, el número de columnas, es fijo a 12.
A continuación, se muestra una imagen representando el entorno, con los números de casillas , los acantilados y la posición final.

Figura 15. Esquema de la numeración de las casillas del entorno Cliff Walking.

Recompensas:
En este entorno el objetivo es llegar al destino en el menor número de pasos. Con una simple observación humana es claro que el mejor camino sería llegar a través del camino junto al acantilado como se muestra en la figura:

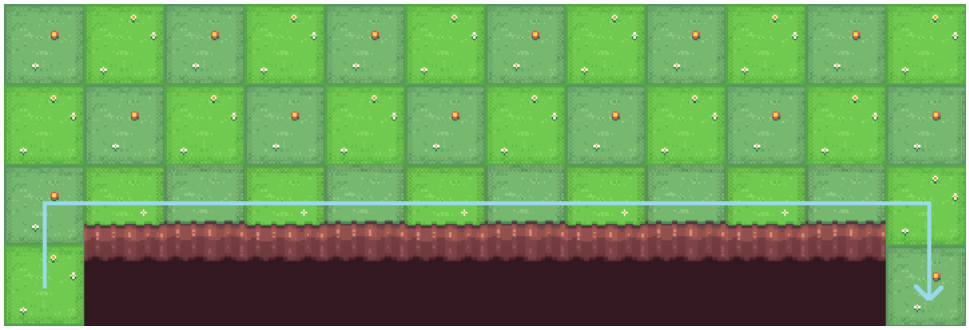

Figura 16. Esquema de la ruta óptima del entorno Cliff Walking.

Para lograr este objetivo, se incorpora un castigo por cada vez que el personaje realice un paso. De esta forma el agente se entrenará con un modelo de recompensa dispersa (sólo recibe la recompensa al final de cada episodio), de otra forma al utilizar sólo recompensas negativas, podría no aprender que debe llegar al punto final para lograr el objetivo, además de que podría penalizar mucho la acción de movimiento a la derecha evitando la acción principal de la ruta a la que debe converger.
[19]

Entrenamiento y resultados:

Para entrenar un modelo capaz de cumplir el objetivo del entorno, se ha hecho uso de un sistema de aprendizaje por refuerzo mediante el algoritmo *DQN*. La red neuronal utilizada tiene una estructura de [12, 8, 4] neuronas en sus capas. Una red apropiada para un entorno de 48 estados y 4 acciones disponibles. Otros hiperparámetros relevantes son el la tasa de actualización del *épsilon-greedy*, fijada a 0.994 y la tasa de aprendizaje de la red, fijada a 0.001. Para entrenar la red se realizaron un total de 600 entrenamientos, obteniendo los siguientes resultados:

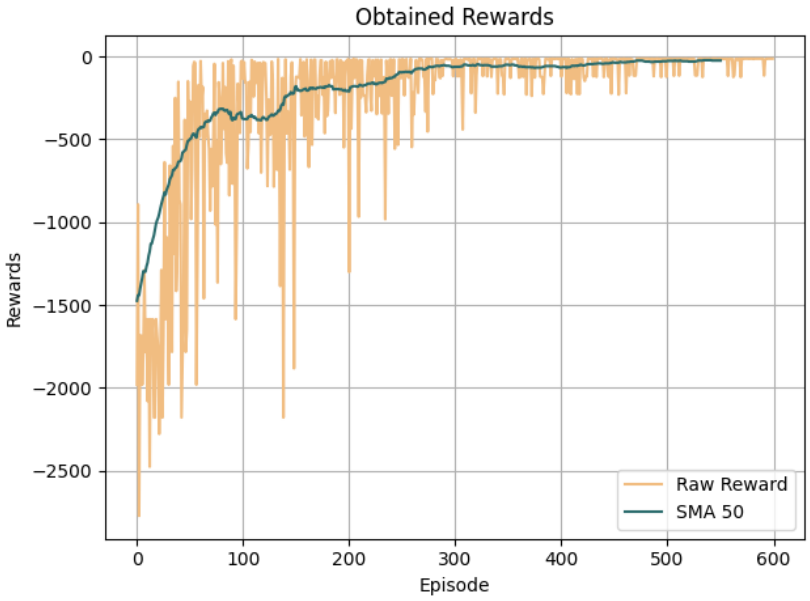

Figura 17. Gráfica de la evolución de las recompensas en el entorno Cliff Walking.

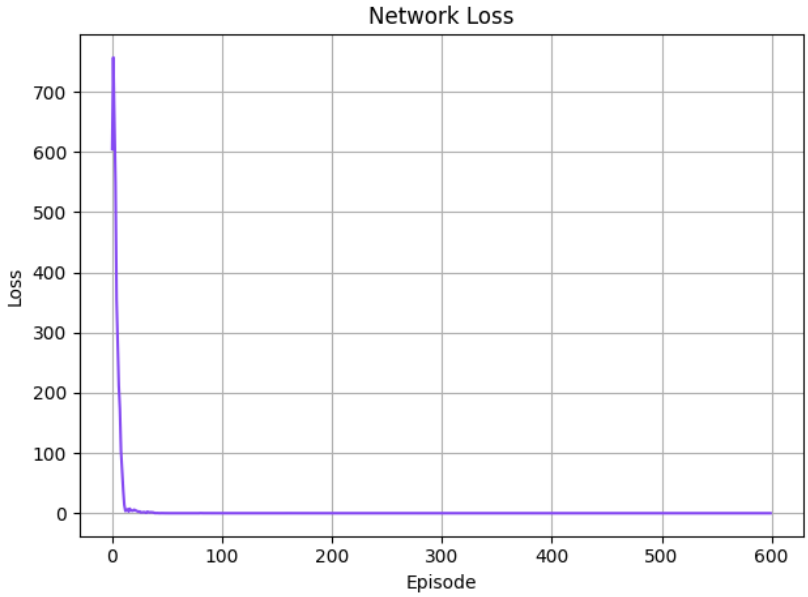

Figura 18. Gráfica de la evolución de la pérdida de la red en el entorno Cliff Walking.

En estas gráficas se puede observar que el agente tiene una recompensa media muy baja durante los primeros episodios, además en los primeros episodios el agente recibe recompensas muy distintas, esto es fruto de la estrategia *épsilon-greedy*, que comienza el entrenamiento asignando una alta probabilidad a escoger las acciones de manera aleatoria. A partir del episodio 200 el agente ya ha aprendido bastante, pues la mayoría de episodios se quedan muy cerca de la recompensa 0 (la recompensa óptima es obtener -13, que se logra al hacer la ruta óptima explicada anteriormente); previsiblemente, los pocos episodios a partir de aquí que no hacen la ruta óptima se ejecutan con activaciones del *épsilon*, que hace escoger alguna de las acciones del episodio de manera aleatoria. En los últimos episodios, el modelo ha convergido casi por completo, y ya no hay irregularidades, porque el valor de *épsilon* se reduce hasta llegar a valores muy cercanos a 0. La función de perdida de la red, evoluciona más rápido que las recompensas, con los 10 primeros episodios, consigue reducir enormemente el error, y en el episodio 100 ya consigue minimizar el error todo lo posible.

7.5. *Frozen Lake*

El entorno *Frozen Lake* es similar a *Cliff Walking*. El personaje debe llegar desde un punto inicial hasta la meta, sin caerse por los agujeros del lago de

hielo. En este entorno, la distribución de los agujeros en el lago es irregular, por lo que el modelo tendrá que aprender mejor a diferenciar las casillas que son desventajosas de las válidas. Además, este nuevo entorno tiene una característica de dificultad añadida, que hace que el personaje pueda desplazarse en una dirección perpendicular a la deseada debido al deslizamiento con la capa de hielo del lago.

Espacio de acciones:
El espacio de acciones es el mismo que en *Cliff Walking*, con la asignación de números a acciones:
0: Movimiento hacia arriba
1: Movimiento hacia la derecha
2: Movimiento hacia abajo
3: Movimiento hacia la izquierda

Espacio observable:
El único estado que se puede recibir del entorno es la posición del personaje, al igual que en Cliff Walking. Este entorno se basa en una malla bidimensional de la misma manera que el anterior, aunque las dimensiones de la malla son distintas, ahora la malla será de tamaño 4x4. La cantidad de estados observables sería 16, antes de restar los agujeros del lago y la posición final. Para representar la posición como un único entero se usa el mismo cálculo que en el entorno anterior:

$$current\_row \cdot ncols + current\_col$$

Figura 19. Expresión del cálculo del estado del entorno Frozen Lake.

En este caso *ncols*, el número de columnas, es fijo a 4.
A continuación, se muestra una imagen representando el entorno, con los números de casillas y la posición final.

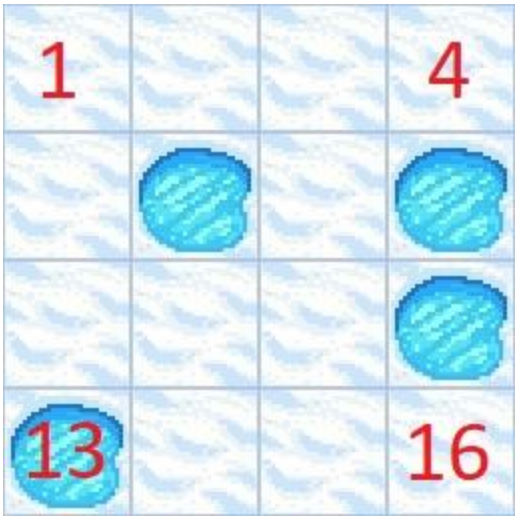

Figura 20. Esquema de la numeración de las casillas del entorno Frozen Lake.

Recompensas:
Al igual que en el entorno anterior, el objetivo es llegar al destino en el menor número de pasos. Observando el mapa, se puede deducir que el menor número de pasos necesarios serían 6. Para lo que existirían 3 rutas posibles, pero no necesariamente son las 3 igual de buenas, porque si el personaje llega a resbalarse por el hielo podría caer por los huecos del lago. Sin embargo, por probar recompensas distintas al entorno anterior, ahora se otorgarán recompensas en función de cómo acaba el nivel. Otorgando una recompensa positiva si se alcanza la meta, y ninguna recompensa si el episodio acaba debido a que el personaje se cae por un agujero del lago. Esta decisión condiciona el experimento a utilizar recompensa dispersa, ya que no existe una recompensa asociada a un paso intermedio.
[20]

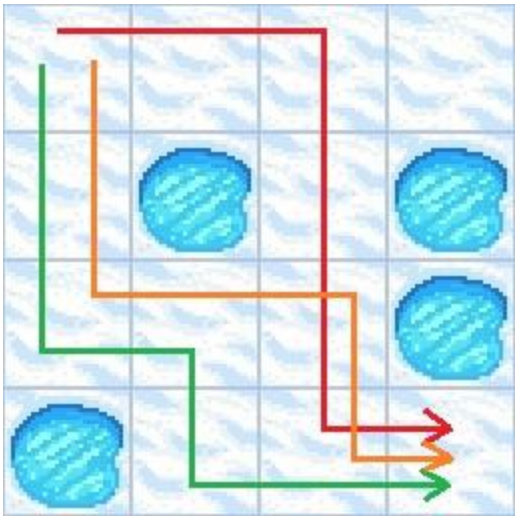

Figura 21. Esquema de la ruta óptima del entorno Frozen Lake.

De las 3 rutas que se distinguen, la verde sería la óptima, pues sólo pasa por 2 casillas en las que el efecto de "resbalamiento" podría hacer caer al personaje a un agujero, la ruta naranja, pasa por 4 casillas críticas, y la roja por 3.

Entrenamiento y resultados:
Para el modelo en este entorno, se ha utilizado un modelo *DQN* con los mismos hiperparámetros que en el anterior, pues se trataba de un entorno de objetivo y dimensiones simlares. Red neuronal de estructura [12, 8, 4], tasa de actualización del *épsilon-greedy* de 0.994 y tasa de aprendizaje de la red de 0.001. Para entrenar la red se realizaron un total de 800 entrenamientos, obteniendo los siguientes resultados:

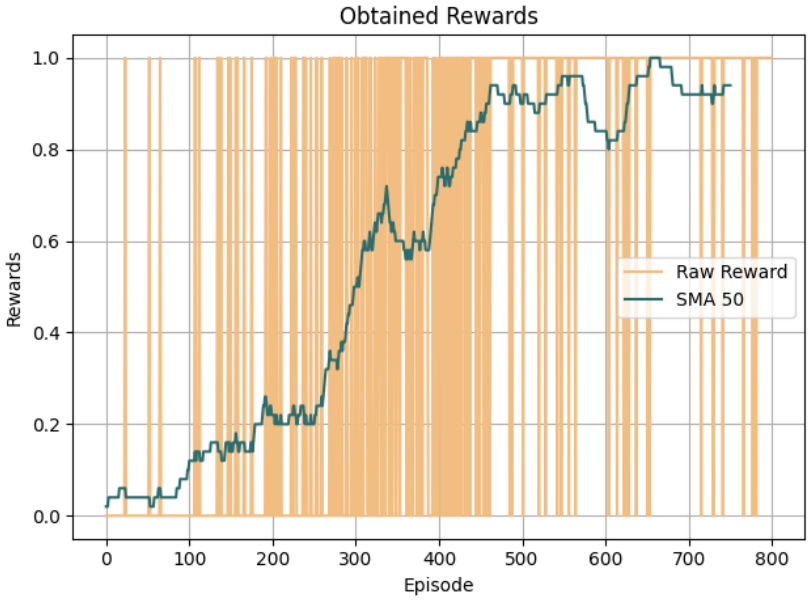

Figura 22. Gráfica de la evolución de las recompensas en el entorno Frozen Lake.

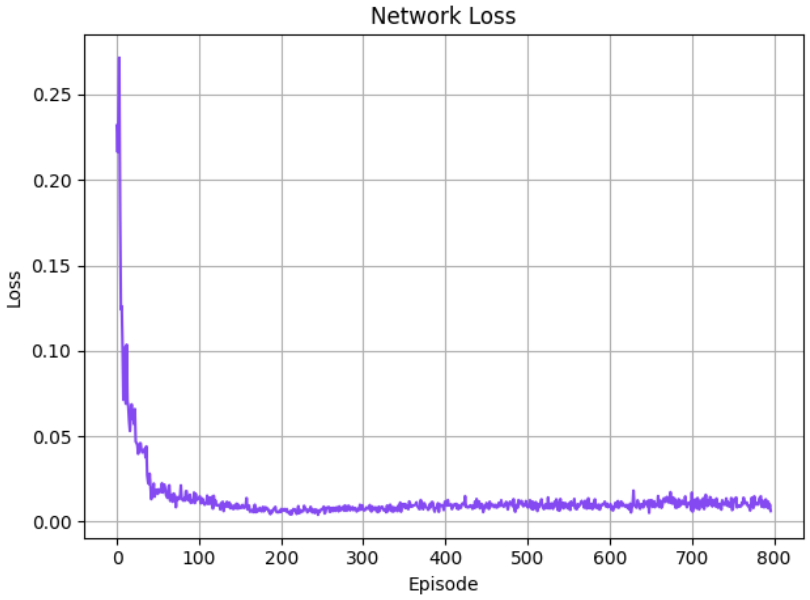

Figura 23. Gráfica de la evolución de la pérdida de la red en el entorno Frozen Lake.

En este ejemplo las recompensas oscilan continuamente, ya que las únicas recompensas posibles son 0 (si se fracasa en completar el juego) ó 1 (si se supera con éxito). De todas formas, el agente logra aprender y se puede apreciar la

variación de épsilon, cuando es alto, hay muchas ejecuciones de recompensa 0, cuando es mediano las recompensas oscilan mucho, y cuando el épsilon disminuye, las mayor parte de las ejecuciones consiguen superar el nivel obteniendo recompensa de 1. Por otro lado, la función de pérdida, sí se minimiza muy rápidamente, cuando la red neuronal ya tiene un ejemplo en el que se supera el nivel, consigue aprender de ese ejemplo y reducir enormemente el valor.

7.6. Entorno de batallas *Pokémon* simplificado

Antes de implementar un agente de combate en un entorno real de *Pokémon*, se plantea evaluar un entorno que simule un entorno semejante al de las batallas *Pokémon*, pero simplificando algunas características, con el fin de facilitar el desarrollo del entorno y la capacidad de aprendizaje del agente. Dicho entorno ha sido desarrollado utilizando *Gymnasium* para definir el entorno.

En este entorno, en lugar de utilizar todos los elementos Pokémon existentes se han seleccionado 4 de ellos y se han modificado algunas ventajas elementales entre ellos:

Elemento del Pokémon defensor				
Elementos atacante:	Normal	Psíquico	Siniestro	Lucha
Normal	Daño x1	Daño x1	Daño x1	Daño x1
Psíquico	Daño x1	Daño x1	Daño x0	Daño x2
Siniestro	Daño x1	Daño x2	Daño x1	Daño x0.5
Lucha	Daño x2	Daño x0.5	Daño x2	Daño x1

Figura 24. Tabla de la ventaja elemental del entorno de combates Pokémon

Ambos *Pokémon* contarán con 100 puntos de salud y podrán utilizar movimientos que infligirán 10 puntos de daño (antes de aplicar modificaciones). En cada turno, el *Pokémon* de cada equipo elegirá un movimiento y ambos se efectuarán al mismo tiempo. El primer *Pokémon* en caer debilitado pierde la partida, si ambos caen debilitados en el mismo turno, se considera un empate.

Espacio de acciones:
Se han definido como acciones los distintos movimientos que puede realizar cada *Pokémon*. En este entorno, independientemente del elemento que tenga el

personaje, podrán utilizar un ataque de cada elemento. Según la siguiente asignación de ataques a números:
0: Ataque de tipo Normal
1: Ataque de tipo Psíquico
2: Ataque de tipo Siniestro
3: Ataque de tipo Lucha

Espacio observable:
El espacio que se puede observar del entorno, es la salud restante de cada personaje (un valor entre 0 y 100) y el elemento de cada personaje (representado por un número de 0 a 3). Por las cantidades de daño que se pueden infligir, la salud restante sólo pueden ser números múltiplos de 5. Por lo que para calcular el número de estados posibles del entorno sería tener en cuenta los 21 estados de salud distintos para cada personaje y los 4 estados de elemento distintos para cada personaje. Esto deja una cantidad total de 7056 estados distintos en el entorno, mayor que en los entornos evaluados anteriormente.

Recompensas:
Como el principal objetivo de este agente es infligir daño al rival, la única recompensa que se ha incorporado es 1 punto de recompensa por cada punto de daño infligido al rival. Además, si utiliza un movimiento ineficaz (es decir, que inflija 0 puntos de daño), el agente recibirá un castigo de 10 puntos. No se incorpora ninguna penalización por recibir daño porque según el entorno, el agente no tiene una manera efectiva de minimizar el daño recibido. En este caso se usaría recompensa dispersa, pues la recompensa se le otorga al agente al final, pero incorpora *reward-shaping*, pues la recompensa no se decide por alcanzar el estado final, sino por hitos intermedios (dañar al rival). A la hora de obtener las recompensas, el agente recibirá mayores recompensas si derrota al rival rápidamente, esto se ha realizado dividiendo sus recompensas entre el número de turnos empleados en el episodio y multiplicando por 5 (cantidad de turnos óptima). Esta normalización de las recompensas es interesante, para evitar que el rival tenga oportunidad de ganar a nuestro agente, si tarda más turnos en ganar previsiblemente recibirá más daño, lo que podría hacer perder al agente.

Entrenamiento y resultados:
Para el entrenamiento del agente en este nuevo entorno de batallas *Pokémon*, se ha definido una red neuronal de configuración [64,64,4], al encontrarnos con un entorno de una gran cantidad de estados. Como tasa de actualización del *épsilon-greedy* se ha utilizado en 0.999 y la tasa de aprendizaje de la red se ha

mantenido en 0.001. El entrenamiento de la red ha utilizado 20000 episodios dejando los siguientes resultados:

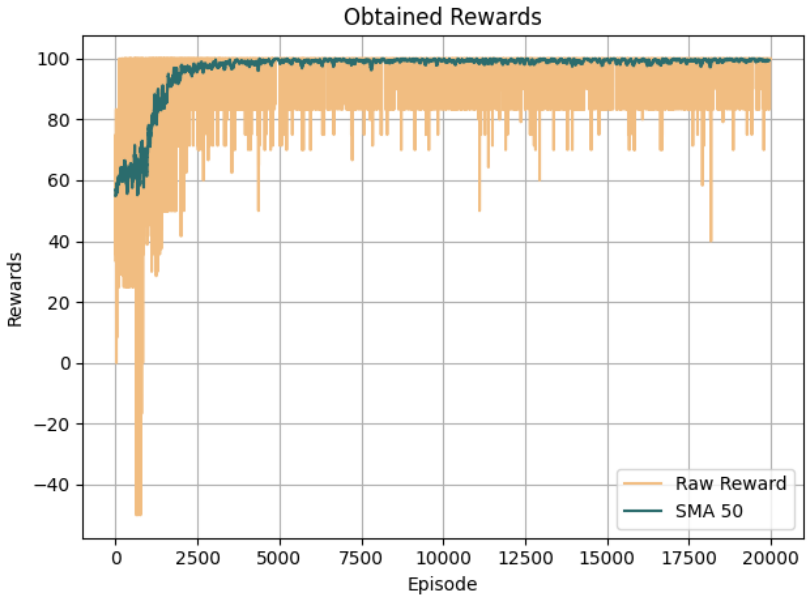

Figura 25. Gráfica de la evolución de las recompensas en el entorno de combates Pokémon simplificado.

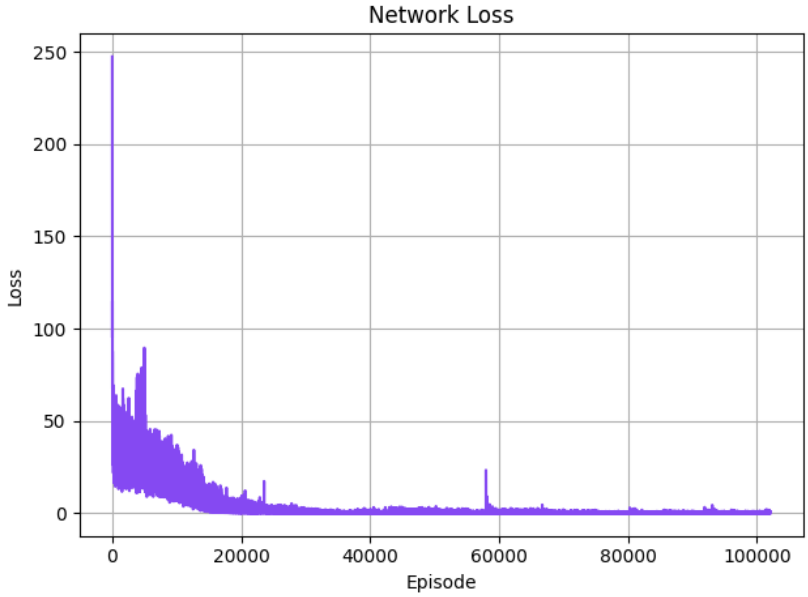

Figura 26. Gráfica de la evolución de la pérdida de la red en el entorno de combates Pokémon simplificado.

En este experimento, el agente consigue ir maximizando las recompensas aproximándose bastante al valor 100 (la recompensa máxima). Cerca del episodio 2500 ya se estabiliza en una recompensa media en torno a 98 y con los episodios posteriores consigue mejorar la recompensa y eliminar bajadas puntuales de la recompensa. Con la función de pérdida de la red ocurre algo similar, sólo que en este caso consigue minimizar más las irregularidades ocasionales.

```
Wins: 100/100
Moves: uneffective: 0, not effective: 1, neutral: 8, supereffective: 499
```

Figura 27. Salida del resumen de simular 100 combates del agente.

Haciendo un ejemplo probando 100 combates contra el entorno haciendo que el rival escoja movimientos aleatorios. El agente consigue ganar o empatar todos los combates (podría variar con las ejecuciones). Y conseguir una distribución de movimientos utilizados en la que priman los movimientos supereficaces. Con estos resultados, se puede concluir que el agente consigue aprender con éxito las reglas del combate y el siguiente paso sería escalar la complejidad del entorno utilizando el entorno de los juegos originales.

7.7. Diseño de agente en el entorno *Showdown*

Los combates simulados en *Pokémon Showdown* siguen de una manera mucho más fiel la lógica de los juegos originales de Pokémon, en comparación al entorno simplificado anterior deben tenerse en cuenta algunas diferencias importantes del entorno, como la existencia de 18 tipos elementales en lugar de 4, Pokémon que pueden poseer hasta 2 de estos en lugar de 1, varios *Pokémon* en cada equipo.

El diseño del agente de aprendizaje por refuerzo para el entorno de *Pokémon Showdown* se plantea utilizando el ejemplo presente en el proyecto *Poké-env* . Dicho ejemplo emplea las librerías *TensorFlow* (para el desarrollo de la red neuronal), y *Gymnasium* (para definir el entorno dentro del paradigma del aprendizaje por refuerzo) [21].

Al intentar ejecutar el ejemplo del proyecto en el entorno de desarrollo, se encontró el siguiente error:

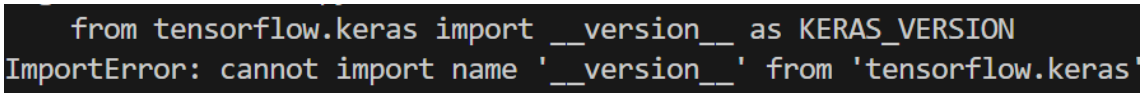

Figura 28. Salida del error del ejemplo de Poké-env.

Este error indica que el módulo tensorflow.keras no contiene un atributo o nombre llamado __version__. La línea de código intenta importar un nombre inexistente, lo cual genera un ImportError. Este tipo de error es común cuando hay cambios en la API de una librería o cuando la documentación utilizada no está actualizada. Este análisis apunta a que el problema se encuentre en las versiones requeridas de los distintos módulos, ya que las notificadas en el propio proyecto de *Poké-env* no eran correctas. Sin embargo, después de probar con diferentes versiones de los paquetes involucrados, el error persistió.

8. Conclusiones

El proyecto tuvo como objetivo principal diseñar un modelo de Inteligencia Artificial que explorara el potencial del uso de IA en videojuegos de estrategia por turnos, específicamente en el contexto de los juegos de *Pokémon*. A través de este trabajo, se buscó aplicar técnicas de aprendizaje por refuerzo profundo (*Deep Reinforcement Learning*, *DRL*) para entrenar un modelo capaz de mejorar su desempeño mediante simulaciones de combates contra sí mismo.

A lo largo del proyecto, se lograron adquirir conocimientos teóricos y prácticos sobre las técnicas de *DRL*, lo cual incluyó la implementación de algoritmos de aprendizaje automático y su aplicación en problemas complejos. Este proceso de aprendizaje no solo fue teórico, sino que se reforzó con la práctica a través de la implementación de sistemas de aprendizaje por refuerzo en entornos clásicos proporcionados por la librería *Gymnasium*. Esto permitió una comprensión más profunda de los fundamentos y desafíos asociados con el aprendizaje por refuerzo.

En la fase práctica, se estudiaron las funcionalidades de la librería *Gymnasium*, implementando y experimentando con entornos clásicos de aprendizaje por refuerzo. Estos ejercicios iniciales fueron fundamentales para establecer una base sólida antes de abordar el entorno más complejo de los combates *Pokémon*.

Posteriormente, se implementó un escenario simplificado de combates *Pokémon* para analizar el funcionamiento de los sistemas de aprendizaje por refuerzo. Esta simplificación permitió identificar y resolver problemas en un entorno controlado antes de intentar aplicarlo a un entorno más complejo como *Pokémon Showdown*.

No obstante, aunque se planificó el uso de *DRL* en el entorno de *Pokémon Showdown*, uno de los principales problemas fue la incompatibilidad de ciertas dependencias necesarias, lo que impidió la implementación completa dentro del marco temporal del proyecto. A pesar de este obstáculo, la metodología de *DRL* fue extensamente estudiada y se hicieron avances significativos en la planificación y diseño del agente de combate.

Como trabajo a futuro se propone completar la implementación del uso de sistemas de aprendizaje *DRL* en combates *Pokémon*. Una opción para lograrlo podría ser implementar el agente del entorno de *Pokémon Showdown* con distintas librerías. También se podría expandir eventualmente el entorno

simplificado hasta que contuviese todas las características del entorno de *Pokémon Showdown*.

En resumen, el proyecto proporcionó una valiosa experiencia en el diseño y desarrollo de modelos de IA para videojuegos de estrategia por turnos, consolidando conocimientos tanto teóricos como prácticos en técnicas de DRL y demostrando el potencial y las limitaciones actuales de estas tecnologías en aplicaciones complejas como los combates Pokémon.

9. Referencias

9.1. Bibliografía

[1] «Videojuego de estrategia por turnos,» Wikipedia, [En línea]. Available: https://es.wikipedia.org/wiki/Videojuego_de_estrategia_por_turnos.

[2] «AlphaGo,» Wikipedia, [En línea]. Available: https://es.wikipedia.org/wiki/AlphaGo.

[3] «Atari Games,» Papers With Code, [En línea]. Available: https://paperswithcode.com/task/atari-games.

[4] «Programming AI for Pokémon Showdown,» Rempton Games YouTube, [En línea]. Available: https://www.youtube.com/watch?v=C1KpQc9cWmM.

[5] M. L. Sanchez, Learning complex games through self play, UPC, Barcelona, España, 2018.

[6] L. P.-B. Ballester, Reinforcement learning para videojuegos, UA, Alicante, España, 2023.

[7] I. Olmeda, Introduction to Machine Learning, Material de asignatura Inteligencia Artificial. GII. Departamento de Ciencias de la Computación, UAH, Alcalá de Henares, España..

[8] T. M. Mitchell, Machine Learning, Nueva York, Estados Unidos: McGraw-Hill, 1997.

[9] «Expert Systems in AI: Bridging Human Expertise and Machine Intelligence,» SAP Community, [En línea]. Available: https://community.sap.com/t5/technology-blogs-by-sap/expert-systems-in-ai-bridging-human-expertise-and-machine-intelligence/ba-p/13705873#:~:text=The%20key%20difference%20between%20a,directly%20from%20data%20through%20training..

[10] I. Olmeda, Introduction to Deep Learning, Material de asignatura Inteligencia Artificial. GII. Departamento de Ciencias de la Computación, UAH, Alcalá de Henares, España..

[11] L. M. Bergasa, Introduction to the Neural Networks, Material de asignatura Sistemas de Control Inteligente. GII. Departamento de Electrónica, UAH, Alcalá de Henares, España..

[12] «¿Cuál es la importancia del equilibrio entre exploración y explotación en el aprendizaje por refuerzo?,» EITCA Academy. [En línea]. Available: https://es.eitca.org/artificial-intelligence/eitc-ai-arl-advanced-reinforcement-learning/introduction-eitc-ai-arl-advanced-reinforcement-learning/introduction-to-

reinforcement-learning/examination-review-introduction-to-reinforcement-learning/what-is-the-significance-of-the-exploration-exploitation-trade-off-in-reinforcement-learning/

[13] García-Ferreira, «Explotación o Exploración: El Gran Dilema,» García-Ferreira, [En línea]. Available: https://www.garcia-ferreira.es/explotacion-o-exploracion-el-gran-dilema/.

[14] V. M. e. al., «Human-level control through deep reinforcement learning,» Nature, vol. 518, no. 7540, pp. 529-533, 2015. [En línea]. Available: https://storage.googleapis.com/deepmind-media/dqn/DQNNaturePaper.pdf.

[15] G. Luo, «Pokémon Showdown: Pokémon battle simulator,» [En línea]. Available: https://github.com/smogon/pokemon-showdown.

[16] H. Sahovic, «Poke-env: pokemon AI in python,» [En línea]. Available: https://github.com/hsahovic/poke-env.

[17] «Gymnasium Documentation: Env.,» Farama, [En línea]. Available: https://gymnasium.farama.org/api/env/#.

[18] «Gymnasium Documentation: Cart Pole,» Farama, [En línea]. Available: https://gymnasium.farama.org/environments/classic_control/cart_pole/.

[19] «Gymnasium Documentation: Cliff Walking,» Farama, [En línea]. Available: https://gymnasium.farama.org/environments/toy_text/cliff_walking/.

[20] «Gymnasium Documentation: Frozen Lake,» Farama, [En línea]. Available: https://gymnasium.farama.org/environments/toy_text/frozen_lake/.

[21] H. Sahovic, «poke-env/examples,» [En línea]. Available: https://github.com/hsahovic/poke-env/blob/master/examples/rl_with_new_open_ai_gym_wrapper.py.

9.2. Índice de Figuras

9.3. Archivos de código fuente

Los archivos de código fuente empleados en este proyecto se encuentran en el repositorio de *GitHub*: MarcoGonzalezM/TFG-MarcoGonzalezM

ESCUELA POLITECNICA
SUPERIOR

Universidad
de Alcalá