E.T.S. de Ingeniería Industrial,
Informática y de Telecomunicación

Gestión de Microgrids mediante Aprendizaje por Refuerzo

Grado en Ingeniería Informática

Trabajo Fin de Grado

Miguel Pagola Barrio

Ángel Sanz Gutiérrez

upna
Universidad Pública de Navarra
Nafarroako Unibertsitate Publikoa

Índice

1. Introducción

1.1 Resumen

La inteligencia artificial se propone como una de las ramas más importantes de la informática y una herramienta fundamental para resolver muchos de los problemas de la humanidad debido a su capacidad de procesar grandes cantidades de datos y crear estrategias que pueden incluso superar a las planteadas por los humanos. Uno de los paradigmas de la IA que puede ser a futuro uno de los más destacados es el aprendizaje por refuerzo, basado en la conducta humana de ensayo-error junto con una recompensa por realizar las acciones en el entorno del problema. El objetivo final es maximizar esta recompensa para conseguir una política (estrategia de comportamiento) óptima.

Probablemente el invento más importante de la humanidad podría considerarse como la electricidad o mejor dicho la capacidad de poder generar energía. Hoy en día todo depende de la electricidad y un gran problema de esto reside en la gestión de las redes eléctricas para satisfacer la demanda y la generación descentralizada de esta, además del gran problema medioambiental que supone. Esto es lo que podríamos llamar una microgrid donde existe un sistema de control y fuentes de demanda y generación de energía.

El trabajo se enfoca en usar algoritmos de aprendizaje por refuerzo específicamente los algoritmos DQN y PPO, siendo ambos algunos de los más reconocidos en este campo, para la optimización de la microgrid y poder así realizar una comparación con uno de los algoritmos base más usados por su sencillez y capacidad llamado Rule Base Control (RBC).

Para cumplir este objetivo se realizará una investigación del funcionamiento de estos algoritmos, así como en su implementación se emplea una de los frameworks más usados para IA como es Pytorch. Además, contamos con la librería Pymgrid para la simulación de la microgrid, esta se hereda de otra llamada Gymnasium de Open AI la cual es fundamental para el aprendizaje por refuerzo. Una vez terminado el análisis observamos que estos algoritmos no suponen una mejora sustancial, llegando a comportarse en cuanto a coste de gestión igual que el RBC después de esto se proponen posibles mejoras para aumentar su rendimiento siendo también necesario un estudio más profundo de que algoritmo son los más adecuados para resolver el problema.

1.2 Palabras clave

- Aprendizaje por Refuerzo
- Microgrids
- Deep Q Learning
- Proximal Policy Optimization
- Pytorch
- Ray Tune
- Gymnasium Open AI

1.3 Objetivo

El objetivo final de este trabajo de investigación y desarrollo consiste en la optimización de una microgrid mediante el campo del Aprendizaje por Refuerzo trabajando específicamente con los algoritmos de Deep Q-Network y Proximal Policy Optimization. Una vez planteado el objetivo final este se divide en objetivos más sencillos, solucionando cada uno de estos pasos intermedios se irá aumentando la complejidad del problema.

El primer paso consiste en aprender sobre el nuevo paradigma con el que estamos trabajando, puesto que es completamente distinto a lo visto anteriormente. Para ello se aprenderá el funcionamiento básico del nuevo paradigma con lo que se consigue abstraerse del funcionamiento e implementación de los algoritmos con lo que se trabaja, posteriormente se estudiarán en profundidad los algoritmos empleados para comprender su funcionamiento interno y como se podría realizar la implementación.

A continuación, será necesario aprender a utilizar con un nivel adecuado todas las herramientas que se usan durante el proyecto, sobre todo enfocándose en las librerías de Gymnasium Open AI, Pytoch, Ray Tune. Y destacando la necesidad de comprender en gran detalle la librería Pymgrid puesto que es la clave para la simulación de la gestión de las microgrids, revisando internamente el código de la gestión para encontrar las posibilidades y limitaciones de la librería.

Una vez completados el primer y segundo subobjetivo se realiza la implementación de los algoritmos DQN y PPO, planteando a su vez mejoras en la optimización de estos y su ejecución en la GPU como método de aceleración de cómputo. Estos a su vez se testean con los entornos disponibles en Gymnasium Open AI simplemente para comprobar su correcta implementación y comprobar las mejoras de optimización.

Finalmente se realiza la búsqueda de los mejores hiperparámetros para cada experimento y escenario planteado, también se realizará un estudio de la manera en que se comporta cada algoritmo en el entrenamiento y una comparación respecto a los algoritmos tradicionales de gestión de microgrids que es donde se observará si existen mejoras o si nos vemos en la necesidad de emplear otros algoritmos o un nuevo enfoque con los algoritmos empleados.

2. Microgrid

Es necesario que conozcamos que significa una microgrid puesto que la idea fundamental es la gestión de estos sistemas eléctricos, una microgrid se define como un sistema de energía a pequeña con un conjunto de componentes clave que funcionan en perfecta sincronía para suministrar la energía necesaria para cumplir la demanda. Este tipo de sistemas puede funcionar de dos maneras distintas: conectado a la red eléctrica general o de forma aislada e independiente de esta (modo isla).

El objetivo de tener una microgrid consiste en mejorar la fiabilidad, eficiencia y la posibilidad de reducir costes de operación para zonas geográficas específicas o para instalaciones tales como campus universitarios, complejos industriales o incluso viviendas particulares si se reducen las cargas de suministro y demanda para ajustarlas a un escenario correcto.

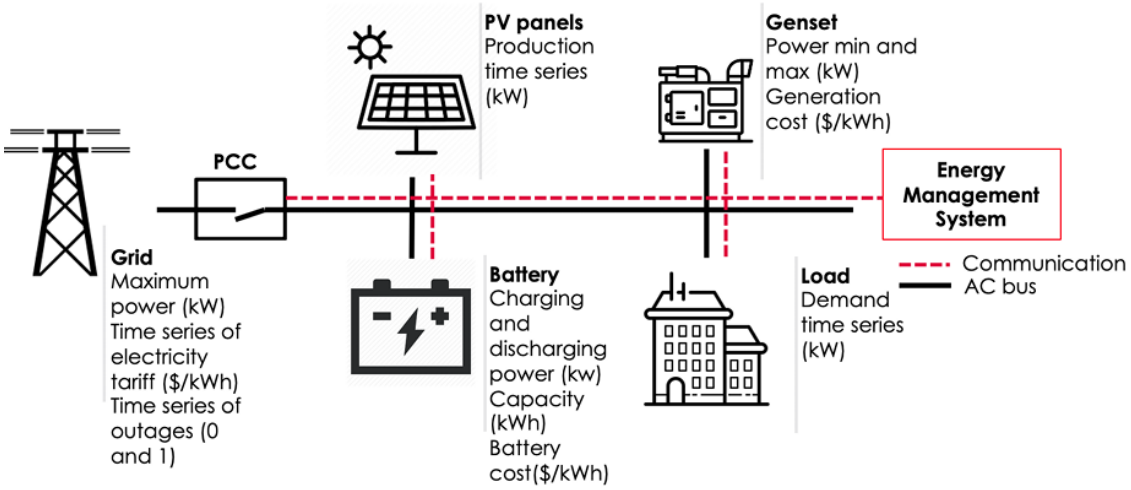

Ilustración 1 Ejemplo de Microgrid de Pymgrid [18]

Podemos mencionar una serie de componentes que pueden estar presentes en una microgrid:

- **Fuentes de generación distribuida:** en este componente englobamos todos los sistemas de la microgrid que produzcan en algún momento energía eléctrica, podemos incluir las energías renovables, generadores y la red eléctrica general.

- **Sistemas almacenamiento:** es posible que la microgrid tenga sistemas de almacenamiento de energía tales como baterías, estas ayudan a cumplir con la demanda sobre todo en sistemas con energía renovables donde existe intermitencia en la producción o en sistemas modo isla.
- **Punto acoplamiento a la red eléctrica general:** se trata del interruptor que permite funcionar a la microgrid conectada a la red eléctrica general o que funcione como isla. Es posible que una microgrid funcione a intervalos en modo isla y luego pase a modo conectado o viceversa.
- **Demanda energética:** la energía que necesitamos cumplir para que la microgrid funcione en condiciones óptimas, será necesario cumplir en todo momento esta demanda.
- **Sistema de control de microgrid:** consiste en el algoritmo o software encargado de gestionar que se satisfaga la demanda de energía, aquí es donde entran los algoritmos tradicionales hasta ahora o los que vamos a proponer mediante el Aprendizaje por Refuerzo en este trabajo.

El problema de las microgrids llega en su momento de gestión. Es decir que fuente de energía usar, cuando usarla, almacenarla o venderla debido a esto supone un problema de optimización de gran complejidad, aquí es donde existe la posibilidad de que el Aprendizaje por Refuerzo sea una buena opción para considerar.

3. Aprendizaje por Refuerzo

Uno de los objetivos fundamentales del trabajo consiste en aprender el funcionamiento de uno de los campos de la Inteligencia Artificial menos conocidos, pero con un gran impacto actualmente en las últimas investigaciones que consiste en el Aprendizaje por Refuerzo principalmente gracias al laboratorio de Google llamado DeepMind el cual es especialista en emplear este campo para resolver problemas de alta complejidad, desde el momento en que presentaron AlphaGo [1].

En el siguiente capítulo presentaremos una introducción a los conceptos sobre el Aprendizaje por Refuerzo, los objetivos que tiene y una explicación teórica detallada de cada algoritmo que usaremos, así como el pseudocódigo de cada algoritmo que hemos implementado desde cero.

3.1 Introducción

Podríamos asemejar este estilo de entrenamiento con el del proceso de humanos y animales de ensayo y error para cumplir objetivos, donde una recompensa positiva o negativa acaba condicionando que se tomen ciertas decisiones según el instante de tiempo en el que estén. Por lo que se trata de un paradigma muy diferente a los más comunes como el supervisado, no supervisado o el auto supervisado, ya que es posible incluso que no contemos con ningún dato específico como tenemos en el resto de los paradigmas. Si no que se busca entrenar un agente (modelo de IA) el cual va a interactuar con un entorno con el objetivo de aprender a maximizar una recompensa acumulada esperada que sirve para establecer cómo de bien funciona el agente.

Necesitaremos por lo tanto establecer qué componentes únicos comprenden el Aprendizaje por Refuerzo y cómo interactúan entre ellos para maximizar la recompensa, tal y como podemos ver en la *Ilustración 1.*

- **Agente:** se trata del modelo de IA que vamos a entrenar, será el encargado de dados los estados del entorno aprender a predecir qué acción es la más adecuada.
- **Entorno:** consiste en todo aquello que se usa para simular el problema con el que estemos trabajando.
- **Estado:** se representa con S_to s y establece todos los valores que especifican la configuración actual del entorno. Este puede ser discreto o continuo según el entorno en el que estemos interactuando.
- **Acción:** se denotan como A_t o a son el conjunto de decisiones que se van a realizar dentro del entorno para completar un paso de tiempo en la simulación. Podrán ser tanto discretas como continuas según con el tipo de entorno con el que trabajemos.
- **Recompensa:** se denotan como R_t o r se obtienen tras simular las acciones en cada estado en que se encuentra el entorno. Consiste en un valor numérico con el cual el agente obtendrá un feedback de cómo de bien funciona la acción seleccionada en el entorno según el estado en el que nos encontramos. Como hemos mencionado, podrán ser tanto positivas (si la acción conlleva un beneficio) como negativas (si la acción constituye una penalización) la magnitud del valor de la recompensa refleja la importancia del feedback.
- **Política:** denotada como π, define a la estrategia global que seguirá un agente para decidir qué acciones se toman. Esta puede llegar a ser estocástica (se otorga una probabilidad de tomar cada acción en el estado) o determinista (donde se especifica la única acción para cada estado), por lo que el agente aprenderá una política que maximice la recompensa acumulada esperada.

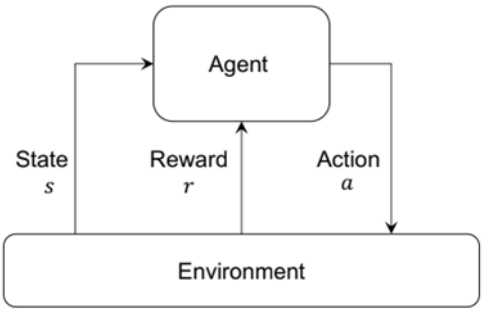

Ilustración 2 Interacción del aprendizaje por refuerzo

El proceso de aprendizaje en los algoritmos de aprendizaje por refuerzo se compone de una serie de pasos secuenciales, donde se busca la mejora de la política:

- El agente recibe el estado s_t y la recompensa acumulada r_t

- Se selecciona una acción a_t usando la política $\pi(a_t \mid s_t)$
- Se ejecuta una simulación (paso de tiempo) en el entorno usando la acción a_t
- Debido a la acción obtenemos un nuevo estado en el entorno definido como s_{t+1}y se calcula la nueva recompensa acumulada de ejecutar la acción $r_{t+1} = R(s_t, a_t, s_{t+1})$
- Se actualiza la política π del agente según el algoritmo que estemos utilizando.
- Se actualiza el nuevo estado y recompensa: $s_t = s_{t+1}$ y $r_t = r_{t+1}$

Este bucle se repite hasta que la simulación con el entorno finaliza, entonces tendremos lo que llamamos un episodio, el cual contendrá tantos pasos de tiempo (*steps*) como el valor t el cual puede variar en cada episodio.

Normalmente se repite este proceso de simulación y aprendizaje de episodios según el número de veces que se quiera entrenar, otras veces en vez de indicar el número total de episodios a usar durante el entrenamiento se especifica el número de simulaciones o steps que se van a usar durante el entrenamiento.

Vamos a trabajar con dos tipos de algoritmo de aprendizaje por refuerzo es conveniente establecer una pequeña clasificación de estos como podemos ver en la *Ilustración 2*.

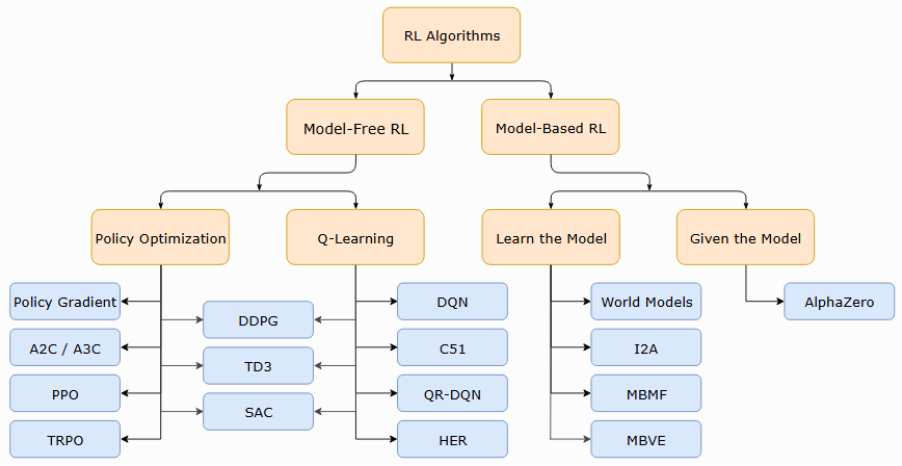

Ilustración 3 Clasificación de algoritmos https://spinningup.openai.com

Nosotros nos vamos a centrar únicamente en la parte izquierda del diagrama, aquellos algoritmos del tipo *Model-Free RL* y específicamente en los algoritmos DQN y PPO. Donde la diferencia principal entre estos dos algoritmos es:

- Q-Learning: se busca aprende primero una función de valor $Q(s, a)$ para cada acción posible en un estado y posteriormente se establece la política, la cual podría ser maximizar esta función.
- Policy Optimization: directamente se aprende la política $\pi(a \mid s, \theta)$, esta se parametriza dentro de una red neuronal y se ajustan los pesos θ de esta para maximizar la recompensa acumulada esperada directamente sin la necesidad de calcular la función de valor para cada acción.

3.2 Exploración vs Explotación

Nos podemos ya preguntar el cómo elegir las acciones con el agente para poder satisfacer el dilema de la exploración respecto a la explotación, puesto que es necesario que el agente pruebe nuevas acciones nunca exploradas antes para descubrir la recompensa que ofrecen, pero al mismo tiempo también es necesario seleccionar aquellas acciones ya exploradas para maximizar la recompensa. Por lo que será necesario un proceso inteligente para compaginar ambas estrategias.

Una de las estrategias más usadas, sobre todo en algoritmo DQN, es la epsilon-greedy policy donde se compaginan una selección aleatoria de acciones (exploración) y una que maximiza la recompensa de acciones pasadas (explotación). Se establece un parámetro denominado como ε (épsilon) el cual decae con el paso del entrenamiento para seleccionar una estrategia u otra con el objetivo de que al principio del entrenamiento nos centramos más en la exploración y conforme avance este seleccionemos más asiduamente la estrategia de explotación ya que existirá un conjunto lo suficientemente grande de acciones exploradas.

En otros tipos de algoritmo como podría ser el PPO este problema ya tiene solución desde el planteamiento principal del algoritmo, ya que al emplear una política estocástica existe desde el inicio la opción de explorar acciones con una probabilidad baja de ser seleccionadas.

3.3 Objetivos

Como ya hemos mencionado anteriormente el objetivo fundamental de estos algoritmos consiste en la optimización de la política $\pi^*(s)$ para los estados. Es decir, el maximizar la recompensa acumulada esperada para un número de episodios.

Para ello tendremos que saber primero el proceso de cálculo de la recompensa de un episodio (τ):

$$R(\tau) = \sum_{i=0} \gamma^i * r_{i+1}$$

Se establece un hiperparámetro llamado factor de descuento (γ) con un rango de valor entre 0 y 1, el cual establece la importancia de las recompensas a futuro. Si se establece como un valor cercano 0, únicamente se da importancia a las recompensas inmediatas mientras que un valor cercano a 1 da importancia a que el agente considere recompensas a futuro.

Para conseguir esta optimización de la recompensa acumulada esperada, nos podemos ayudar de dos conceptos fundamentales en el aprendizaje por refuerzo que son la función de valor de estado y la función de valor de estado-acción:

- **Función de valor de estado $V(s)$:** se usa para cuantificar cómo de bueno es un estado. Representa la recompensa acumulada esperada si comenzamos en el estado s y seguimos una política determinada π.

$$V^{\pi}(s) = E_{\pi}[R(\tau) \mid s_1 = s]$$

- **Función de valor de estado-acción $Q(s, a)$:** se usa para cuantificar lo bueno que es realizar una acción en un estado. Representa la recompensa acumulada esperada de empezar en un estado s, tomar una acción determinada a y seguir la política π.

$$Q^{\pi}(s, a) = E_{\pi}[R(\tau) \mid s_1 = s, a_1 = a]$$

Es necesario que mencionemos que no todos los algoritmos del campo del aprendizaje por refuerzo usarán directamente estas fórmulas para calcular la recompensa acumulada esperada, esto dependerá de la familia a la que pertenezcan, tal y como hemos visto en el apartado 2.1. Puesto que el algoritmo DQN si hará uso de al menos una de estas funciones directamente para calcular la política, mientras que PPO establece la política directamente sin la necesidad de estas funciones intermedias.

3.4 DQN

Como hemos podido ver en la *Ilustración 1* del apartado 2.1 Introducción el algoritmo DQN pertenece a la familia de algoritmos Model Free y también a la subclase Q-Learning tratándose de un método off-policy, este representa un gran hito en el campo del aprendizaje por refuerzo puesto que se trató del primer método desarrollado capaz de aprender una política desde una entrada de alta dimensión y a su vez demostró que se podrían usar las redes neuronales para aproximar las funciones de valor reemplazando así las tablas de valor o funciones lineales mediante la extracción de características clave.

3.4.1 Origen

El algoritmo DQN [2] fue desarrollado en el año 2013 por el laboratorio DeepMind para dominar juegos de la Atari 2600, pero desafortunadamente no fue hasta el 2015 que su publicación una revisión en la revista Nature [3] que obtuvo un gran reconocimiento y se popularizó su uso como método efectivo en el campo del aprendizaje por refuerzo.

Se trata de una evolución del algoritmo Q-Learning [4], algoritmo de aprendizaje por diferencia temporal, y su mejora usando aproximación lineal. Para ello se plantea usar redes neuronales que aprendan las "aproximaciones lineales" solucionando así la selección manual de características para estas aproximaciones y la asunción de que las funciones de valor Q son lineales. También vamos a destacar dos características fundamentales que mejoran el rendimiento:

- **Replay Memory (experience replay):** se emplea de almacenamiento para las transiciones (s, a, r, s') que se generan durante el entrenamiento. En vez de ir aprendiendo en orden de la generación de las transiciones, estas se van almacenando en un *replay buffer* de un tamaño fijo y posteriormente se realiza un muestreo aleatorio con las que entrenará el agente. Se emplea este método para solucionar principalmente dos problemas de aprender de transiciones secuenciales, que son: datos correlacionados (si se entrena con transiciones secuenciales estas están altamente correlacionadas por lo que al realizar el entrenamiento con lotes de transiciones aleatorias se evita la correlación reduciendo así la varianza de las actualizaciones), eficiencia en los datos (al poder usar una misma transición más de una vez en el entrenamiento) y solución a posibles mínimos locales (puesto que se realiza un promedio de las transiciones no se priorizan aquellas transiciones que maximicen la recompensa evitando así mínimos locales y los bucles de retroalimentación)

- **Target Network:** aunque esta mejora no se propuso en el artículo original del 2013 sino en el de la publicación de Nature 2015. Se emplean dos redes neuronales, la principal que se actualiza en cada lote de la *Replay Memory* y la segunda red (target network), las cuales tendrán la misma arquitectura y los pesos se actualizarán usando los de la primera red neuronal cada cierto número de pasos. Esta segunda red neuronal se usa para obtener los valores de la función Q del siguiente estado, obteniendo así un entrenamiento más estable.

3.4.2 Fundamentos matemáticos

Si tomamos como punto de partida la función de valor de estado-acción $Q(s, a)$ que hemos definido en el apartado 2.3 Objetivos y dado que el objetivo que tenemos con los algoritmos de aprendizaje por refuerzo es encontrar la política óptima que maximice la

recompensa acumulada esperada. Con esto se transforma la ecuación anterior en la ecuación de Bellman:

$$Q^*(s,a) = R(s,a) + \gamma * \ Q^*(s´,a´)$$

Donde se expresa que para resolver el problema usando el algoritmo DQN se debe calcular los valores de la función $Q(s,a)$ la cual es igual a la recompensa $R(s,a)$ de realizar la acción a en el estado s más el valor de la función $Q(s´,a´)$, que consiste en el valor Q de ejecutar la mejor acción posible $a´$ en el siguiente estado $s´$, multiplicado por el factor de descuento (γ). Esta acción a se elige mediante la política epsilon-greedy.

Hemos mencionado anteriormente en el algoritmo DQN se emplean dos redes neuronales, la principal se encarga de calcular los valore de la función $Q(s,a)$ y la segunda red neuronal (*target network*) calcula $Q(s´,a´)$ con el objetivo de cumplir con la función de Bellman, la cual se usa para obtener el valor de la predicción objetiva (y). Dado esto ahora podemos aplicar una función de pérdida en este caso se usa el Mean Squared Error (MSE) entre el valor Q obtenido de la red neuronal principal y la predicción objetiva obtenida con la función de Bellman, con lo que ya podemos aplicar el descenso por gradiente para actualizar los pesos de las redes neuronales.

$$Loss \ = \ (y \ - \ Q(s_i, a_i))^2$$

En el siguiente esquema *Ilustración 4* de podrá ver cómo funciona el entrenamiento del algoritmo se verá más claro el funcionamiento de ambas redes neuronales, que devuelven y para que se usa cada predicción:

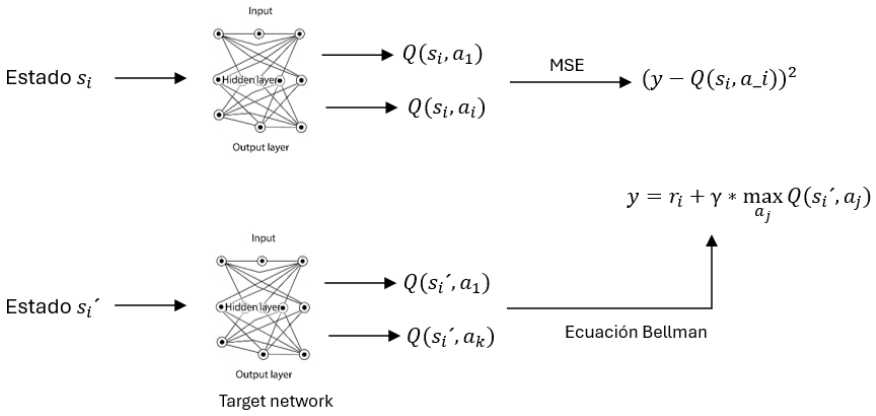

Ilustración 4 DQN

3.4.3 Pseudocódigo

A continuación, veremos el pseudocódigo, el cual será útil para visualizar cómo se ha programado el algoritmo. Salvando que no se mostrarán muchas de las eficiencias en el código y la configuración usada para el entrenamiento con la GPU. Pero alguna de las mejoras propuestas son la ejecución de todo el proceso de entrenamiento en la GPU, para ello es

necesario que la información de las interacciones con el entorno se almacene en un buffer alocado en la GPU además de todo el proceso de entrenamiento de las redes neuronales y los cálculos de las funciones objetivo. Otra característica añadida para mejorar la eficiencia en la ejecución únicamente con GPU consiste en la precisión mixta donde se emplean distintas precisiones de bits en los cálculos. Por ejemplo, los parámetros de las redes neuronales (la parte "menos" crítica del proceso) se puede realizar con números de 16 bits y las partes críticas emplean una precisión de 32 bits (esto se realiza sobre todo para el cálculo de la función de pérdida y el proceso de backward).

$\theta \rightarrow$ random values; $\theta^{target} \rightarrow \theta$
for episode $= 1 \ldots$ M do
  $s \rightarrow$ initial state
  for $t = 1 \ldots T$ do
    with probability ε select random action a_t
    otherwise, select $a_t \rightarrow$ armax $Q(s, a_i, \theta)$
    $s_t', r_t \rightarrow$ results of execute action on the enviroment
    store transition (s_t, a_t, r_t, s_t') in Replay Memory
    if transitions stored $>$ minibacth:
      sample random batch (s_j, a_j, r_j, s_j') from Replay Memory

$$y_j \rightarrow \begin{cases} r_j & \text{if } s_j' \text{ is terminal state} \\ r_j + \gamma * \max_{a'} Q(s_j', a', \theta^{target}) & \text{otherwise} \end{cases}$$

      perform gradient descent to update θ with loss $= \left(y_j - Q(s_j, a_j, \theta)\right)^2$
      every k steps: $\theta^{target} \rightarrow \theta$
  end for
end for

3.5 PPO

Hemos podido ver en la *Ilustración 1* del apartado 2.1 Introducción el algoritmo DQN pertenece a la familia de algoritmos Model Free y también a la subclase Policy-Optimization tratándose de un método on-policy. Este algoritmo es uno de los más populares y empleados en el campo del aprendizaje por refuerzo debido a gran rendimiento, convergencia durante el entrenamiento además de corregir ciertos problemas que aparecen en otros algoritmos tales como ser efectivo en un gran espacio dimensional para las acciones, las cuales pueden ser tanto discretas como continuas y finalmente aprende políticas estocásticas, por lo que no es necesario implementar métodos de exploración vs explotación.

3.5.1 Origen

El algoritmo PPO (Proximal Policy Optimization) [5] fue presentado en el 2017 por OpenAI este se trata de una evolución de otro algoritmo creado en el 2015 llamado TRPO (Trust Region Policy Optimization) [6]. PPO hereda la idea fundamental de los métodos Actor-Critic, como el Advance Actor Critic (A2C) donde contamos con dos estructuras que definen el comportamiento del algoritmo: un Actor el cual controla la política y un Critic que mide lo buenas que son las acciones.

El algoritmo PPO supone una gran ventaja respecto al TRPO, aprovechando las características relevantes del TRPO tales como la eficiencia en los datos y su rendimiento además añade optimizaciones de alto nivel aplicando un recorte de la función objetivo y realizar varias épocas (epochs) de optimización sobre los datos muestreados:

- **Función de Objetivo Recortada (Clipped Surrogate Objective Function):** se sustituye la rigurosa restricción sobre la divergencia de Kullback-Leibler del algoritmo TRPO por una modificación de la función objetivo reduciendo a su vez la complejidad. Esta nueva función restringe que las actualizaciones en la política excedan un límite inferior y superior con el objetivo de desincentivar cambios en la relación de probabilidad de la política actualizada respecto a la anterior.

- **Múltiples epochs:** PPO permite realizar varias veces el entrenamiento sobre un mismo lote de datos, tal y como se realiza en el entrenamiento tradicional de las redes neuronales, con este método se mejora la eficiencia de los datos y por ende el rendimiento que podemos obtener con el algoritmo.

3.5.2 Fundamentos matemáticos

Puesto que el algoritmo PPO es de la clase Policy Gradient necesitamos entender en primer lugar la idea fundamental de este tipo de algoritmos, para ello como hemos mencionado una de las características fundamentales es la parametrización de la política π_θ mediante una red neuronal con lo que obtenemos la distribución de probabilidad sobre las acciones:

$$\pi_\theta(s) = P[A|s, \theta]$$

Dada la política, lo que intentamos conseguir es una mejora en su rendimiento mediante el ascenso del gradiente (*gradient ascent*) y para ellos definimos una función objetivo $J(\theta)$ que consiste en la recompensa acumulada esperada, ahora está en el Policy Gradient se define como:

$$J(\theta) = E_{\pi}[R(\tau)]$$
$$= \sum_{\tau} P(\tau \mid \theta) * R(\tau)$$

Ya sabemos que $R(\tau)$ es la recompensa acumulada del episodio τ y $P(\tau \mid \theta)$ es la probabilidad del episodio τ dependiendo de los parámetros θ, si descomponemos esta función obtenemos que:

$$P(\tau \mid \theta) = \prod_{t=0} P(s_{t+1}|s_t, a_t) * \pi_\theta(a_t|s_t)$$

Debido a que nuestro objetivo es maximizar la función objetivo $J(\theta)$ usamos para ello el ascenso del gradiente, el cual tiene el objetivo contrario al descenso por gradiente, comúnmente usado para el entrenamiento del Deep Learning donde el objetivo es minimizar la función objetivo.

$$\theta \leftarrow \theta + \alpha * \nabla_\theta J(\theta)$$

Ahora para poder obtener la $\nabla_\theta J(\theta)$ aplicamos el Policy Gradient Theorem con lo que finalmente obtenemos que la derivada es:

$$\nabla_\theta J(\theta) = E_{\pi_\theta}[\nabla_\theta \ log \ log \ (\pi_\theta(a_t \ |s_t)) * A_t \ ]$$

Esta consiste en la derivada usada como función objetivo en el Policy Gradient, la cual se define como:

$$L^{PG}(\theta) = \ E_t[log \ log \ \pi_\theta((a_t \ |s_t) * \ A_t \ ]$$

donde definimos A_t como el estimador que mide lo buenas que son las acciones (a_t) en el estado (s_t) comparado con la acción promedio o el valor esperado. Este valor se puede calcular como $A_t = Q(s_t, a_t) - V(s_t)$ y si este valor es positivo significa que la acción en el estado es mejor que el resto de las acciones (promedio esperado) y si el valor es negativo significa que la acción es peor de lo esperado.

Hemos mencionado que PPO aplica una serie de mejoras a la función objetivo para corregir problemas asociados a la estabilidad en el aprendizaje, con esto se define una nueva función objetivo llamada *Clipped surrogate objective function* (L^{CLIP}):

$$L^{CLIP}(\theta) = \ E_t[min(r_t(\theta) * A_t, clip(r_t(\theta), 1 - \varepsilon, 1 + \varepsilon) * \ A_t \ )]$$

Nos encontramos con un nuevo parámetro llamado ratio de probabilidad ($r_t(\theta)$) el cual define la relación de probabilidad entre la política actual (π_θ) y la política antigua ($\pi_{\theta_{old}}$), se calcula de la siguiente manera: $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$. Podemos establecer que si este valor $r_t(\theta) > 1$ la acción (a_t) en el estado (s_t) es más probable en la política actual y si $0 < r_t(\theta) < 1$ la acción es menos probable en la política actual que en la antigua.

Ahora pasaremos a explicar la parte sin el recorte establecido en la función L^{CLIP} a la cual también se le puede llamar $L^{CPI}(\theta)$:

$$L^{CPI}(\theta) = r_t(\theta) * A_t = E_t[\frac{\pi_\theta(s_t)}{\pi_{\theta_{old}}(s_t)} * A_t]$$

Y también es necesario que detallamos la parte que tiene el recorte en la función L^{CLIP} puesto que se trata de una de las características más importantes del algoritmo PPO para evitar grandes actualizaciones de la política:

$$clip\ [(\mathrm{r_t}(\theta), 1-\varepsilon, 1+\varepsilon) * A_t]$$

con esto conseguimos penalizar los cambios del ratio que se alejan del intervalo, se define en el artículo original el valor de $\varepsilon = 1$ con lo que el ratio sólo podrá fluctuar en el intervalo entre 0.8 y 1.2

La *Ilustración 4* muestra la función L^{CLIP} la cual crea una gráfica de donde se aplica el recorte ε para emplear una parte de la función o la otra correspondiendo a la zona que se encuentre, tendremos 6 posibles casos para estudiar *Tabla 1*.

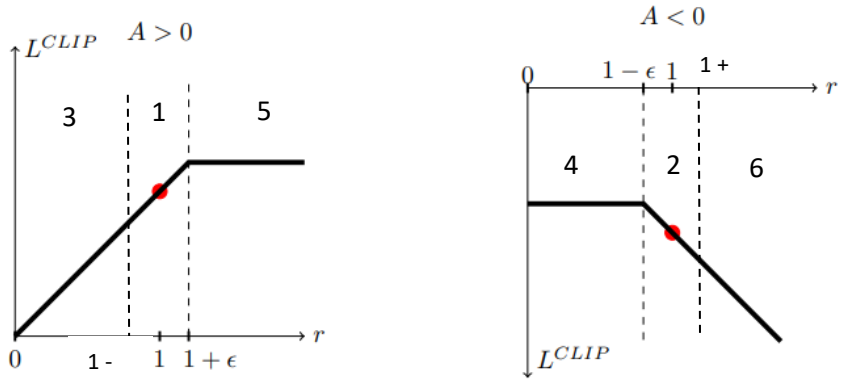

Ilustración 5 Visualización L^{CLIP}

$r_t(\theta) > 0$	A_t	Retorno del min	Aplica recorte	Signo Función	Gradiente
$r_t(\theta) \in [1-\varepsilon, 1+\varepsilon]$	+	$r_t(\theta) * A_t$	no	+	✓
$r_t(\theta) \in [1-\varepsilon, 1+\varepsilon]$	-	$r_t(\theta) * A_t$	no	-	✓
$r_t(\theta) < 1-\varepsilon$	+	$r_t(\theta) * A_t$	no	+	✓
$r_t(\theta) < 1-\varepsilon$	-	$(1-\varepsilon) * A_t$	si	-	no
$r_t(\theta) > 1-\varepsilon$	+	$(1+\varepsilon) * A_t$	si	+	no
$r_t(\theta) > 1-\varepsilon$	-	$r_t(\theta) * A_t$	no	-	✓

Tabla 1 Casos estudio L^{CLIP}

- Caso 1 y Caso 2: el $r_t(\theta)$ se encuentra dentro del rango por lo que no se aplica el recorte de la función objetivo.
    - Caso 1: $A_t > 0$, la acción es mejor que el resto de posibles acciones por lo que se aumenta la probabilidad de seleccionar la acción en ese estado.
    - Caso 2: $A_t < 0$, la acción es peor que el promedio del resto de acciones por lo que tenemos que disminuir la probabilidad de elegir la acción.
- Caso 3 y Caso 4: donde el $r_t(\theta) < 1 - \varepsilon$ la acción es menos probable en la política actual que en la antigua.
    - Caso 3: $A_t > 0$, tenemos que aumentar la probabilidad de elegir la acción, puesto que se trata de una buena decisión a tomar, pero al tener menor probabilidad de ser elegida en la política actual es necesario mejorarla.
    - Caso 4: $A_t < 0$, no queremos disminuir la probabilidad de elegir la acción puesto que esta se encontraría fuera del límite inferior, por lo que además se aplicará el recorte de la función L^{CLIP} y a su vez al ser el gradiente igual a 0 no se actualizarán los pesos.
- Caso 5 y Caso 6: donde el $r_t(\theta) > 1 + \varepsilon$ la acción es más probable en la política actual que en la antigua.
    - Caso 5: $A_t > 0$, no aumentamos la probabilidad de elegir la acción, ya que esta se encontraría fuera del límite superior, por lo que además se aplicará el recorte de la función L^{CLIP} y a su vez al ser el gradiente igual a 0 no se actualizarán los pesos.
    - Caso 6: $A_t < 0$, la acción es peor que el promedio del resto de acciones por lo que disminuimos la probabilidad de esta.

Finalmente, para calcular la función objetivo del algoritmo PPO usando la estructura Actor-Critic debemos combinar varias funciones objetivo más junto con la que hemos mencionado hasta ahora:

- *Clipped surrogate objective function* (L^{CLIP}):
- *Value Loss Function* (L^{VF})
- *Entropy bonus* (S)

$$L_t^{CLIP+VF+S}(\theta) = E_t[L^{CLIP}(\theta) - c1 * L^{VF} + c2 * S[\pi_\theta](s_t)]$$

De la función final podemos definir también varios parámetros fundamentales:

- $c1$ y $c2$ que son coeficientes para establecer importancia en las otras funciones objetivo que se han combinado
- $L^{VF} = (V_\theta(s_t) - V_t^{targ})^2$ se trata de una función de pérdida
- $S[\pi_\theta](s_t)$

Hasta ahora sabemos calcular todas las funciones y parámetros involucrados en el algoritmo PPO, salvo el valor de la ventaja (A_t) que para ellos usaremos el método Generalized Advance Estimation (GAE):

$$A_t = -V(s_t) + r_t + \gamma * r_{t+1} + \cdots + \gamma^{T-t+1} * r_{T-1} + \gamma^{T-t} * V(s_t)$$

Y además si usamos la versión truncada de la función anterior cuando $\lambda = 1$ y sabiendo que $\delta_t = r_t + \gamma * V(s_{t+1}) - V(s_t)$

$$A_t = \delta_t + (\gamma * \lambda) * \delta_{t+1} + \cdots + (\gamma * \lambda)^{T-t+1} * \delta_{T-1}$$

Cabe destacar que en el artículo original del PPO establecen los valores de los parámetros:

- Discount (γ) = 0.99
- GAE parameter (λ) = 0.95

3.5.3 Pseudocódigo

A continuación, veremos el pseudocódigo, el cual será útil para visualizar cómo se ha programado el algoritmo. Salvando que no se mostrarán muchas de las eficiencias en el código y la configuración usada para el entrenamiento con la GPU. Y se realizan también las mismas mejoras propuestas en el apartado

Además, para la implementación nos ayudamos de la información que proporciona OpenAI en OpenAI Baselines [7] y el blog de 37 Implementations detail [8] ambas fuentes proporcionan una explicación detallada del funcionamiento del algoritmo y el cómo se puede implementar.

Tenemos que destacar claves importantes del entrenamiento del algoritmo que difieren del resto de algoritmo convencionales:

- Existen N actores que trabajan en paralelo para recolectar las interacciones con el entorno, a la hora de la implementación lo que se crea es una serie de copias del mismo entorno que se ejecutan en paralelo para interactuar con el agente.
- Se recolectan tantas interacciones como se especifique con el valor de T
- Se optimiza la política con un lote de datos de tamaño $M \leq N * T$ tantas veces como se a especificado en las epochs K

A la hora de implementar el algoritmo PPO existen grandes diferencias en cómo programarlo, pero todas cumplen el objetivo de forma correcta, primero detallamos ciertos parámetros necesarios para comprender la implementación realizada:

- total_timesteps: interacciones totales realizadas en el entorno.

- num_envs: como hemos mencionado se usan varios entornos en paralelo para la recopilación de las interacciones.
- num_steps: número de instantes en el tiempo para recolectar del entorno.
- num_epochs: número de épocas para la actualización de las redes neuronales del agente

Entonces si tenemos un total_timesteps de 2000000, además el número de entornos es igual a 4 y el valor de num_steps del rollout es 1000. Poemas calcular el número de veces que se actualizan los parámetros de la política:

$$num\_updates = total\_timesetps/(num\_envs * num\_steps) = 2000000/(4 * 1000) = 500$$

$for\ update\ in\ range(num\ updates)$:
  $for\ rollout\ step\ in\ range(num\ rollout\ steps)$:
    select a_t from agent
    $s_t', r_t \rightarrow$ results of execute action on the enviroment
    store transition (s_t, a_t, r_t, s_t') in Rollout Buffer
  $end\ for$
  $compute\ advantages\ gae\ of\ transition\ from$ Rollout Buffer
  $for\ epoch\ in\ range(num\ epochs)$:
    $for\ start\ in\ range(0, minibatch\ size)$:
      $get\ data\ from\ Rollout\ Buffer$
      $calculate\ policy\ gradient\ loss$
      $calculate\ values\ loss$
      $calculate\ entropy\ loss$
      $calculate\ finla\ PPO\ loss$
      perform gradient ascent to update agent policy
    $end\ for$
  $end\ for$
$end\ for$

4. Herramientas

En el siguiente apartado realizaremos un pequeño estudio de las herramientas usadas para el desarrollo e implementación del proyecto con el fin de comprender el funcionamiento de estas, por qué se han usado y como se han configurado para poder reproducir exactamente los mismos experimentos realizados.

4.1 Python

Se trata del lenguaje principal empleado para el desarrollo del proyecto, específicamente la versión 3.12, ante posible duda de porque usar este lenguaje de programación y no otros que son específicos para el análisis estadístico o el cálculo matemático tales como R y MATLAB o lenguajes de programación generales como C++ y JAVA.

Python se ha convertido en el lenguaje de programación más usado en el último año el éxito se relaciona con el creciente uso de Inteligencia Artificial y la Ciencia de datos, según el informe Octoverse 2024 [9] publicado por GitHub, a su vez explican un mayor uso de herramientas relacionadas con el desarrollo de Inteligencia Artificial tales como Jupyter Notebook. Otras características de gran relevancia para la elección del lenguaje de programación son las siguientes:

- **Legibilidad:** probablemente Python se trate de uno de los lenguajes de programación más fáciles de comprender debido a su sintaxis simple y fácil de leer, muy parecida al inglés. Se trata de una característica muy apreciada en el software con alta complejidad como podrían ser algunos de los algoritmos de Deep Learning o Reinforcement Learning.
- **Comunidad:** tratándose actualmente del lenguaje más popular como hemos mencionado anteriormente, Python cuenta con una gran cantidad de personas realizando sus propios proyectos de inteligencia artificial desarrollándose así una gran cantidad de librerías, documentación técnica y ayudas sobre distintos campos de esta, incluyendo la explicación e implementación de la gran mayoría de algoritmos principales.
- **Soporte para aceleración gráfica o GPU:** dada la gran cantidad de datos y la complejidad de los algoritmos que se usan en el entrenamiento de los modelos de inteligencia artificial el tiempo de cómputo puede suponer un gran problema si estos se ejecutan en la CPU. Desde el 2012 que se usó por primera vez la aceleración mediante GPU para entrenar el modelo AlexNet, es común el uso de esta tecnología para reducir el tiempo de ejecución, Python proporciona capas de abstracción y librerías para soportar esta tecnología.
- **Rendimiento:** aunque se trate de un lenguaje interpretado y su rendimiento se ve empeorado en comparación con un lenguaje compilado, dado que Python ha sido programado en C, donde muchas funciones existentes o de ciertas librerías se pueden compilar directamente su rendimiento es considerable.

- **Ecosistema de librerías y frameworks:** probablemente la mejor característica que tiene Python es la gran cantidad de librería y frameworks desarrollados para IA tanto por la

comunidad como por los laboratorios/empresas punteras de IA, aquellas librerías más destacadas que se emplean durante el proyecto son las siguientes:

- Numpy: para el cálculo vectorial y matricial de forma sencilla y eficiente. [10]
- Pandas: estructuración de datos complejos en formato DataFrame y lectura/escritura de archivos. [11]
- Matplotlib: librería usada para dibujar gráficas y representar datos [12].
- Pytorch: framework empleado para el desarrollo de los algoritmos de IA, más adelante se verá en detalle.

4.2 Anaconda

Anaconda se trata de la distribución de código abierto desarrollada para los lenguajes de programación de Python y R ampliamente usada en ciencia de datos y aprendizaje automático.

Una de sus ventajas principales consiste en la gestión de paquetes o librerías y entornos de desarrollo donde se pueden crear entornos independientes para manejar las dependencias específicas de cada uno de estos entornos sin problemas de incompatibilidad entre ellos. Además de ser útil para encapsular todos los paquetes en un mismo entorno y poder reproducir los experimentos de manera sencilla, para ello se usa Conda (el gestor de paquetes o librerías de Anaconda) el cual se encarga de que las dependencias y versiones de los paquetes sean compatibles o el propio gestor de Python llamado Pip el cual es compatible con Anaconda.

Otras ventajas del uso de Anaconda en el desarrollo del proyecto son las siguientes:

- **Sistema de paquetes:** ya hemos mencionado que se gestiona de forma automática las dependencias y versiones de los paquetes instalados en los entornos de forma independiente.
- **Instalación directa de frameworks de IA:** en nuestro caso la instalación del framework Pytorch para la implementación del proyecto se puede realizar de forma sencilla empleando Pip. Cuenta con la forma de poder gestionar la versión específica, el Sistema Operativo y los paquetes de soporte para el uso de distintas plataformas de cómputo con sus versiones.
- **IDE:** Anaconda funciona de manera independiente del IDE que se esté empleando por lo que existe una gran libertad para desarrollar proyectos. En nuestro caso usamos JupyterNotebook altamente usado para el desarrollo e investigación en el campo de la Inteligencia Artificial en el cual se crean celdas independientes de ejecución para el código lo cual nos permite ejecutar el código y el resultado inmediatamente. A su vez, la principal fortaleza consiste en la capacidad de agrupar tanto código como análisis de los resultados y gráficas en el mismo documento, en el cual los resultados obtenidos se pueden quedar guardados para visualizar posteriormente sin la necesidad de ejecutar de nuevo el código.

- **Comunidad y ayuda:** cuenta con una gran comunidad de personas y una documentación extensa, detallada y de código abierto sobre el uso y solución de problemas de Anaconda, las librerías y frameworks más usados en el desarrollo de proyectos de Inteligencia Artificial.

4.3 Pytorch

Pytorch [13] se trata de uno de los frameworks de código abierto para el desarrollo de Inteligencia Artificial más usados en el mundo, el cual ha sido desarrollado y publicado por Meta AI (antiguamente llamado Facebook AI Research) pensado para el ámbito de la investigación y se trata del framework elegido por los laboratorios y compañías que desarrollan los algoritmos y librerías de Inteligencia Artificial para crear el código de estos y posteriormente publicarlo. Los algoritmos DQN y PPO originales mencionados anteriormente fueron implementados usando Pytorch.

La base fundamental de Pytorch son los Tensores los cuales son el método para representar la información que deseemos, estos se consideran una matriz multidimensional con lo que se pueden representar como tensores tanto números escalares como vectores y matrices, similares en forma y sintaxis para realizar las operaciones entre tensores a los usados en la librería Numpy. Pudiéndose estos ejecutar en la GPU para optimizar el tiempo de ejecución del modelo de Inteligencia Artificial.

A su vez otra de las grandes características que lo diferencia del resto de frameworks conocidos es la capacidad de crear grafos computacionales dinámicos lo cual se usa para representar el modelo de inteligencia artificial creado, es decir se crea un grafo de flujo de cómo se comportan los datos en los diferentes nodos con sus respectivas operaciones matemáticas y las dependencias entre los distintos niveles/capas de procesamiento de la red. La diferencia entre los grafos estáticos empleados por otros frameworks consiste en aplazar la creación de los cálculos y las dependencias hasta el momento de la ejecución del modelo en tiempo real, por lo que no se crean una vez diseñado y compilado el modelo esto permite que se modifique en tiempo de ejecución la estructura y cálculos del grafo, lo cual permite por ejemplo que el número de capas de una red neuronal se adapte al número de características de los datos de entrada o que según el tipo de datos el modelo se comporte de maneras específica.

Se trata de una característica realmente útil a la hora de modificar partes del código de un modelo de forma independiente sin afectar al resto del modelo o sin tener que volver a reiniciar y compilarlo.

4.4 CUDA y cuDNN

El entrenamiento de los modelos de Inteligencia Artificial supone un coste computacional y de tiempo muy grande debido a la complejidad de los modelos, a la cantidad de hiperparámetros y al número de datos usado para el entrenamiento. Antiguamente se realizaba este entrenamiento en la CPU limitando así la capacidad para crear modelos más grandes y complejos capaces de mejorar su rendimiento y eficiencia para resolver problemas con mayor precisión, también el uso de la CPU aumenta el tiempo de entrenamiento e inferencia puesto que está limitada la capacidad de paralelización y cantidad de la memoria para poder disponer de los datos directamente en la RAM sin tener que cargarlos cada cierto tiempo desde la unidad de almacenamiento.

Pero la mayoría de los problemas mencionados se solucionan en el 2012 con el uso de la aceleración gráfica de las GPUs para el entrenamiento de AlexNet [14], a partir de este momento se vuelve parte de la literatura y una práctica común el uso de la GPU para el entrenamiento e inferencia de los modelos de Inteligencia Artificial, gracias a su gran capacidad de paralelización que proporcionan, la cantidad de RAM o VRAM, la cual se va aumentando cada año para soportar modelos más grandes, y la disponibilidad de usar más de una GPU en paralelo las cuales son capaces de trabajar virtualmente como una sola y acelerar y dividir la cantidad de datos que recibe cada una. Además de que las GPUs están optimizadas para realizar las operaciones matriciales necesarias para el funcionamiento de los modelos, ya que son las mismas operaciones necesarias para el manejo de gráficos, que es la tarea inicial para la que fueron diseñadas.

Lo que nos lleva al uso de CUDA (Compute Unified Device Architecture) [15] para la implementación del código de los algoritmos de Inteligencia Artificial, CUDA se trata de una API publicada por NVIDIA en el año 2007 la cual incluye el compilador y las herramientas necesarias para poder codificar algoritmos usando la GPU y beneficiarse así de la paralelización de estas. Esta herramienta se puede usar en varios lenguajes de programación tales como C/C++, Python, Fortran y JAVA.

Una vez que tenemos las herramientas necesarias para desarrollar algoritmos que se puedan ejecutar en la GPU podríamos centrarnos en las operaciones específicas empleadas en el desarrollo de modelos de Inteligencia Artificial, aquí es donde entra cuDNN (CUDA Deep Neural Network Library) [16] una librería diseñada para la mayoría de las operaciones necesarias en el campo de la Inteligencia Artificial, esta librería proporciona implementaciones altamente optimizadas de las operaciones más usadas en las redes neuronales profundas tales como operaciones convolucionales, multiplicación de matrices, normalización, softmax y las operaciones de atención. Esto nos permite junto con el uso de CUDA reducir el tiempo de entrenamiento.

4.5 Gymnasium Open AI

Gymnasium [17] se trata de una librería de aprendizaje por refuerzo fundamental para el desarrollo y el testeo de algoritmos, desarrollada en Python y mantenida por la Fundación Farma desde el 2021 como sucesor de la librería Gym desarrollada por OpenAI. Se mantiene la misma API y sintaxis, pero mejorando el desarrollo y la cantidad de actualizaciones realizadas para el mantenimiento e implementación de nuevas características. Gymnasium se encarga de uno de los conceptos fundamentales del aprendizaje por refuerzo que es el de la creación y simulación de los entornos con los que interactúa el agente de Inteligencia Artificial, con esta librería somos capaces de abstraernos de la implementación específica y las características únicas de cada entorno con el que queramos trabajar.

A continuación, vamos a explicar conceptos básicos de cómo funciona la sintaxis de esta librería y estándares a cualquier entorno que queramos usar:

- Clase gymnasium.Env es el componente central de esta librería. De cualquier entorno que creemos usando Gymnasium o de cualquiera que creemos ya que heredará de este. Se destacan varios atributos y métodos clave para establecer la información del entorno y la simulación/funcionamiento durante el entrenamiento y la inferencia:

  La información más importante de un entorno se representa con los siguientes parámetros:

  - **Action:** qué acción va a realizar el agente en el entorno.
  - **Observation:** representa el nuevo estado del entorno, el cual se obtiene después de ejecutar una acción.
  - **Reward:** la recompensa/feedback que establece lo bien o mal que ha sido la ejecución del agente en el entorno.
  - **Terminated:** cuando el entorno finaliza naturalmente, es decir el problema con el que trabajamos tiene una condición de finalización la cual se puede alcanzar siempre. Por ejemplo, ganar o perder una partida, un choque en un simulador de conducción.
  - **Truncated:** se establece cuando el entorno finaliza gracias a una condición externa al problema. Por ejemplo, se alcanza un límite de tiempo (duración del episodio).
  - **Info:** información adicional del entorno que puede servir para diagnosticar problemas y mejorar el análisis de los resultados.

  Los métodos que se usan para interactuar con el entorno y que son esenciales en el bucle de aprendizaje del agente son los siguientes:

  - **Reset():** es el método usado para reiniciar el entorno al final de cada episodio. Se emplea para que este vuelva al estado inicial el cual puede ser aleatorio o

determinista si el problema con el que trabajamos lo requiere además es necesario llamar a este método al inicio del entrenamiento.

- Retorno: se devuelve una tupla que contiene la siguiente información (observation, info) en este caso los valores de observation serán los valores del estado inicial.

- **Step():** se trata del método más crítico para el aprendizaje por refuerzo. Sirve para ejecutar un paso de tiempo en el entorno, es decir dada una acción seleccionada por el agente este método realiza la simulación lógica del comportamiento de esta en el entorno con el que trabajamos.
  - Retorno: se devuelve también una tupla que contiene los siguientes valores (observation, reward, terminated, truncated, info)
- **Close():** este método se usa una vez terminado el entrenamiento o la inferencia con el entorno. Sirve para la limpieza y la liberación de memoria de los recursos usados por el entorno, si este también dispone de funciones gráficas se cerrarán.

Existen ciertos valores que sirven para comprender y almacenar cierta información útil para el funcionamiento y la correcta validación de las acciones y observaciones del entorno se representa con los siguientes atributos:

- **Action_space:** consiste en un atributo de tipo gymnasium.spaces.Space el cual corresponde a las acciones válidas del entorno. Todas las acciones que se seleccionen y ejecuten deben estar comprendidas en este espacio de valores. Este atributo también define la capa de salida de la red neuronal que representa al agente.
- **Observation_space:** de forma análoga, indica el tipo y el rango de valores de las observaciones del entorno. A su vez determina también la capa de entrada de la red neuronal del agente.

- Necesitamos detallar en más profundidad ciertos aspectos mencionados en los atributos de la clase gymnasium.Env tales como el action_space y el observation_space puesto que la información que contienen se representa con un tipo de dato que no es común a los lenguajes de programación:

El tipo de dato conocido como gymnasium.spaces.Space lo proporciona Gymnasium para poder establecer la interacción con el entorno de forma correcta, es decir se usa para validar que las acciones sean las indicadas y la estructura para representar las observaciones del entorno. Además de servir de ayuda para poder representar la complejidad de un problema de forma sencilla y sin la necesidad de conocer al detalle el funcionamiento lógico del entorno.

Gymnasium proporciona 4 tipos básicos de Spaces, pero nos vamos a centrar únicamente en 2 de ellos que serán los disponibles posteriormente en la librería que simulará el comportamiento de la microgrid de nuestro problema:

- **Discrete:** se trata de un espacio de valores con elementos finitos. Es un subconjunto finito de números enteros consecutivos de forma que se puede representar de la siguiente manera: $\{a, a+1, \dots, a+n-1\}$.
  Donde a representa el valor de la primera acción y n el número de posibles acciones. Un ejemplo sería el Space de acciones para empujar un elemento respecto al eje horizontal. Lo definiríamos como Discrete(2) donde:
  0 → empujar a la izquierda
  1 → empujar a la derecha

- **Box:** representa un rango de valores infinito comprendido en $R\text{^}n$ se debe especificar a su vez el límite inferior y superior del rango de valores con lo que podemos representar los siguientes rangos $[a, b], (-\infty, b], [a, \infty)$ y $(-\infty, \infty)$ también deberemos establecer el tamaño del Space el cual se establece usando la sintaxis de Numpy.
  Con esto podemos establecer 2 formas básicas para representar el espacio siendo tipo Box:

$$Box(low = -1.0, high = 2.0, shape = (3, 4), dtype = np.float32)$$

Con esto se establece que se trata de una matriz de tamaño 3x4 donde todos sus valores estarán comprendidos entre -1.0 y 2.0

$$Box(low = np.array([-1.0, -2.0]), high = np.array([2.0, 4.0]), dtype = np.float32)$$

En este caso el tamaño se establece usando la dimensión del array usado en el límite inferior y superior. Por lo que será un array de dimensión 2 donde el primer valor está comprendido entre -1.0 y 2.0, el segundo valor estará entre -2.0 y 4.0

4.6 Pymgrid

Ahora debemos explicar la librería principal usada para la simulación de la microgrid, llamada Pymgrid [18]. La cual a su vez hereda de Gymnasium para poder crear los entornos que usaremos en el entrenamiento de los algoritmos de aprendizaje por refuerzo.

Será necesario una descripción de todos los módulos establecidos en la librería para la simulación de distintos elementos que pueden comprender una microgrid, así como una explicación del funcionamiento lógico de la librería para comprender cómo se simula la obtención de energía, la demanda, como se suple está usando las distintas fuentes disponibles

y cómo se visualiza el comportamiento y resultados de la microgrid. También detallaremos los dos entornos disponibles los cuales hacen referencia al uso de acciones discretas o continuas.

Gracias a Pymgrid podemos simular componentes de la microgrid, tales como las fuentes de energía renovables, baterías, generadores diésel, la red eléctrica general y finalmente el módulo que simula la demanda. Cabe destacar que no vamos a mencionar todos los atributos que comprenden las clases establecidas para los módulos, sólo aquellos que son fundamentales conocer para la simulación:

- **RenewableModule**: simula una fuente de energía renovable, los ejemplos suelen ser la fotovoltaica y la eólica. Sirve únicamente como productor de energía y se trata de un módulo no controlable, por lo que no puede recibir acciones. A su vez los atributos que definen este módulo son:
    - *time_series*: consiste en los valores que simularán la producción de energía en kWh. Se establece como un array de tamaño (n_steps,).
    - *forecaster*: se usa para poder cierto número de steps en el futuro además de poder establecer cierta variabilidad a los datos. Puede tener varios valores:
        - *float*: se usa como la desviación estándar para una función de ruido gaussiano con media cero.
        - *oracle*: pronóstico perfecto en el futuro, sin ninguna modificación del time_series.
        - *callable*: una función personalizada por el usuario, la cual debe contener ciertos parámetros necesarias para definirla, tales como: val_c (valor actual del time_series), val_{c+n} (el valor de time_series n steps en el futuro) y finalmente n (número de steps a mirar)
        - *none*: no se usa un forecaster.
    - *forecaster_horizon*: el número de steps en el futuro usado por forecaster. Se establece a 0 si no se usa forecaster.

  Cabe destacar una limitación importante de la librería y es que la energía producida por este módulo siempre se usa para suplir la demanda de energía. Solo se podrá usar para cargar baterías o venderla a la red general si previamente se ha cubierto toda la demanda y existe un exceso de esta posteriormente.

- **BatteryModule:** se trata de la clase que sirve para la simulación de baterías. Esta puede servir como productor o almacenamiento de energía. Por lo que se trata de un módulo controlable en el cual se pueden realizar acciones (las que aprenderán los algoritmos de aprendizaje por refuerzo). A su vez los atributos que definen este módulo son:
    - *min_capacity*: capacidad mínima de la batería en kWh.
    - *max_capacity*: capacidad máxima de la batería en kWh.
    - *max_charge*: capacidad máxima que se puede cargar de la batería en un solo paso en kWh.
    - *max_discharge*: capacidad mínima que se puede descargar de la batería en un solo paso en kWh.

- *efficiency*: eficiencia de la carga/descarga de la batería establecida en forma de porcentaje. Como ejemplos si realizamos $100 - efficiency$ obtenemos la cantidad de energía desperdiciada que no podrá ser usada para cargar/descargar la batería.
- *battery_cost_cycle*: el coste monetario (penalización) marginal en el proceso de carga y descarga.
- *init_charge*: carga inicial de la batería.
- *init_soc*: carga inicial en formato de porcentaje. Si ya se ha establecido previamente el parámetro init_charge, se descarta el uso de init_soc.

- **GensetModule:** simula el funcionamiento de un generador diésel. Esta sirve únicamente como productor de energía y también consiste en un módulo controlable. A su vez los atributos que definen este módulo son:
  - *running_min_production*: la producción mínima de energía del generador en kWh.
  - *running_max_production*: la producción máxima de energía del generador en kWh.
  - *genset_cost*: el coste monetario (penalización) de producción de energía del generador. Puede tener varios valores:
    - float: se trata del coste por unidad de energía. Por lo que el coste marginal se calcula como: $genset\_cost * producción$
    - callable: función personalizada por el usuario donde se le pasa la producción actual en un instante y devuelve el coste final.
  - *co2_per_unit*: la producción de CO2 por cada unidad de energía producida.
  - *cost_per_unit_co2*: el coste monetario (penalización) de cada unidad de CO2 producida por el generador.

- **GridModule:** consiste en la red eléctrica general. La cual puede servir como importador y exportador de energía por lo que se trata también de un módulo controlable. A su vez los atributos que definen este módulo son:
  - *max_import*: la máxima capacidad de importación de energía en cada instante de tiempo.
  - *max_export*: la máxima capacidad de exportar energía en cada instante de tiempo.
  - *time_series*: se trata de un array el cual puede tener 3 o 4 características. Si es de tamaño (n_steps, 3) estas características representan el *import_price*, *export_price*, *co2_per_kwH* y si sus dimensiones son (n_steps, 4) hay que añadir a las características anteriores otra nueva llamada *grid_status* que establece si la red eléctrica está disponible en ese instante de tiempo para importar o exportar energía.
  - *forecaster*: se usa para poder cierto número de steps en el futuro además de poder establecer cierta variabilidad a los datos. Puede tener varios valores:
    - *float*: se usa como la desviación estándar para una función de ruido gaussiano con media cero.

        - *oracle*: pronóstico perfecto en el futuro, sin ninguna modificación del time_series.
        - *callable*: una función personalizada por el usuario, la cual debe contener ciertos parámetros necesarias para definirla, tales como: val_c (valor actual del time_series), val_{c+n} (el valor de time_series n steps en el futuro) y finalmente n (número de steps a mirar)
        - *none*: no se usa un forecaster.
    - *forecast_horizon*: el número de steps en el futuro usado por forecaster. Se establece a 0 si no se usa forecaster.
    - *cost_per_unit_co2*: coste monetario (penalización) de producir una unidad de CO2

- **LoadModule:** se emplea para la simulación de la demanda de la energía en la microgrid, consiste en un módulo no controlable, por lo que no puede recibir ninguna acción. A su vez los atributos que definen este módulo son:
    - *time_series*: consiste en los valores que simularán la demanda de energía en kWh en cada instante de tiempo. Se establece como un array de tamaño (n_steps,).
    - *forecaster*: se usa para poder cierto número de steps en el futuro además de poder establecer cierta variabilidad a los datos. Puede tener varios valores:
        - *float*: se usa como la desviación estándar para una función de ruido gaussiano con media cero.
        - *oracle*: pronóstico perfecto en el futuro, sin ninguna modificación del time_series.
        - *callable*: una función personalizada por el usuario, la cual debe contener ciertos parámetros necesarias para definirla, tales como: val_c (valor actual del time_series), val_{c+n} (el valor de time_series n steps en el futuro) y finalmente n (número de steps a mirar)
        - *none*: no se usa un forecaster.
    - *forecast_horizon*: el número de steps en el futuro usado por forecaster. Se establece a 0 si no se usa forecaster.

Y la clase **Microgrid** es la principal de todas ya que agrupa a los posibles módulos que hemos mencionado para poder realizar la simulación final, así como establecer ciertos parámetros para definir qué hacer en situaciones de sobre generación de energía y la posibilidad de establecer funciones de recompensas personalizadas a la hora de entrenar los algoritmos de aprendizaje por refuerzo. Los atributos de esta clase son:

- *modules*: consiste en una lista con los módulos que van a definir la microgrid, se puede establecer un nombre personalizado para cada módulo.
- *add_unbalanced_module*: se añade un módulo auxiliar llamado UnbalancedEnergyModule el cual gestiona si existe un exceso de demanda no cumplida o generación.

- *loss_load_cost*: define el coste monetario (penalización) por unidad de energía en kWh no satisfecha, se emplea como penalización por incumplir la necesidad de demanda de energía en cada instante de tiempo. Este atributo entra en funcionamiento únicamente cuando no se pueda comprar más energía de la red eléctrica (límite *max_import* del módulo GridModule alcanzado) o si no está disponible la red eléctrica en el instante de tiempo.
- *overgeneration_cost*: define el coste monetario (penalización) por unidad de energía en kWh generada en exceso, se emplea para penalizar la generación de energía que no se va a poder usar. Este atributo se activa si no se puede vender más energía a la red eléctrica (límite *max_export* del módulo GridModule alcanzado) o si no está disponible la red eléctrica en ese instante de tiempo.
- *reward_shaping_func*: puede consistir en una función personalizada de cómo se calcula la recompensa en cada instante de tiempo según los módulos que comprenden la microgrid. Si no se establece ninguna función el método para calcular la recompensa esta se calcula como la suma de cada módulo, más adelante veremos un ejemplo detallado.

Como sabemos una de las claves que diferencian al aprendizaje por refuerzo del resto de campos de la Inteligencia Artificial consiste en la introducción de una recompensa para establecer lo bien que funciona el modelo, y puesto que estamos trabajando con un tipo de entorno muy complejo es necesario establecer cómo se calcula esta recompensa, salvo que nosotros creemos una función personalizada establecida en el atributo *reward_shaping_func* de la clase Microgrid. Digamos que tenemos los siguientes valores en la *Tabla 2* de los atributos como ejemplo:

Energía importada	40 kWh	Coste importación: $40 * 0.20 = -8$
Precio importación	0.20 €/kWh	
Energía exportada	10 kWh	Coste exportación: $10 * 0.10 = -1$
Precio exportación	0.10 €/kWh	
CO2 por kWh	0.5 Kg	Coste emisiones CO2: $40 * 0.5 * 0.05 = -1$
Coste por CO2	0.05 €/Kg	
Falta de demanda	10 kWh	Coste por falta demanda: $10 * 10 = -100$
Coste falta de demanda	10 €/kWh	
Sobre generación	5 kWh	Coste por sobre generación: $5 * 2 = -10$
Coste de sobre generación	2 €/kWh	

Tabla 2 Cálculo de recompensa microgrid

Cabe destacar que en la tabla que hemos propuesto existen incompetencias, puesto que no se puede dar el caso de existir una falta de energía y al mismo tiempo sobre generación de energía, esto lo hemos realizado con el objetivo de comprender al detalle cómo se calculan todos los costos, a la hora de realizar los experimentos solo existirá la falta de demanda o la sobre generación.

Finalmente, la recompensa final se calcula como la suma de todos los costes anteriores más el coste marginal de cargar/descargar las baterías, el cual normalmente se establece a cero. Y como podemos comprobar los costos y por tanto la recompensa final en cada instante de tiempo estarán siempre en negativo. Entonces el objetivo de los algoritmos que vamos a usar para optimizar el control de la microgrid consistirá en maximizar este valor, puesto que se trata de conseguir optimizar el coste final de gestionar la microgrid, si este valor se establece en negativo significa que hemos tenido pérdidas monetarias y si es positivo hemo tenido beneficios gestionando la energía de la microgrid.

Pymgrid proporciona varios escenarios de microgrids que se pueden usar directamente, puesto que ya tienen todos los datos de la Microgrid establecidos. Estos datasets simulan un año entero con un paso de tiempo de una hora, lo que consiste finalmente en 8760 simulaciones totales.

Puesto que consisten en muchos escenarios y contamos con limitaciones de tiempo y hardware vamos a seleccionar 2 escenarios distintos para trabajar *Tabla 3*. Cada escenario presenta características distintas, únicamente comparten la disponibilidad de una conexión con la red eléctrica general por lo que podrán importar y exportar energía. La diferencia principal sería la existencia de un generador diésel, el objetivo es simular un escenario que podríamos asociar a una vivienda residencial (el que no contiene el generador) y otro escenario que sea similar a lo que podríamos encontrar en la industria o sistemas de más demanda energética.

ID	Avg Load	Avg PV	Max Battery	Min Battery	Max Genset	Min Genset	Max Grid Import	Max Grid Export
0	483.87	160.37	1452.00	290.40	-	-	1920.00	1920.00
22	58584.64	18478.25	292924.00	58584.80	80372.70	4465.15	160744.00	160744.00

Tabla 3 Escenarios

Existen también ciertos parámetros que son comunes a ambos escenarios con lo que trabajaremos:

- Forecast: oracle (predicción a futuro sin ninguna modificación de los datos)
- Forecast_horizon: tiene en ambos un valor de 23 para los módulos de carga, energía fotovoltaica y la red eléctrica general. Con esto en cada instante de tiempo contamos con la información de la microgrid actual y con los datos de los módulos 23 instantes de tiempo por delante, por lo que al final trabajamos con todos los datos de simulación de un mismo día a la vez.
- La red eléctrica general está disponible siempre para cada instante de tiempo y además el precio de exportar energía a esta se establece siempre a cero, por lo que se obtiene beneficio de vender exceso de energía.

En el contexto del aprendizaje por refuerzo con el que estamos trabajando necesitamos mencionar los entornos que proporciona Pymgrid los cuales heredan de Gymnsium para su simulación y cumple con los estándares convencionales para operar los algoritmos en estos. Como hemos mencionado se proporcionan dos tipos distintos de entorno dependiendo si las acciones que soportan estos son discretas o continuas, llamados DicreteMicrogridEnv y ContinuosMicrogridEnv:

DicreteMicrogridEnv: entorno discreto donde se implementan listas de prioridades como las posibles acciones de la microgrid, donde se especifica el orden de ejecución de los módulos. Ya hemos mencionado en el apartado 3.5 Gymnasium Open AI existen ciertos valores necesarios conocer tales como action_space y el observation_space**:**

- action_space: si en la microgrid existen n módulos que sirven como fuente de energía, quitando los módulos de energías renovables. El número de acciones posibles se calcula: $num_{acciones} = n! * acciones\_cada\_modulo$
- observation_space: para el cálculo del número de observaciones nos debemos fijar primeramente en las posibles claves que se indican en cada instante de la simulación de la microgrid y además también nos tenemos que fijar en el valor forecast_horizon puesto que añade valores a futuro en aquellas claves que funcionan con series temporales.
  Por ejemplos si tenemos que las claves que se indican en la microgrid son: net_load, load_current, renewable_current, soc, current_charge, import_price_current, export_price_current, co2_per_kwh, grid_status_current por defecto se especifican estas 9 claves para cada instante de la simulación, ahora si estamos trabajando con un forecast debemos añadir a las claves por defecto aquellas claves que se comportan igual que series temporales que son: load_forescat, renewable_forecast, import_price_forecast, export_price_forecast, co2_per_kwh_forecast y grid_status_forecast.

  Por lo que el cálculo general se realiza de la siguiente manera: 9 + forec ast_horizon * 6

ContinuosMicrogridEnv: entono continuo donde las acciones representan los porcentajes de carga/descarga de los módulos que funcionan como fuente de energía en la microgrid. Por lo que el action_space y el observation_space se establece:

- action_space: si en la microgrid existen n módulos que sirven como fuente de energía, quitando los módulos de energías renovables. En este tipo de entorno las acciones no se representan con un número entero si no con en tipo de dato llamado Box, ya que con este se pueden representar todos los posibles valores de las acciones en la microgrid. Las acciones quedan: $acciones = Box(0.0, 1.0\ (n), float)$

- observation_space: se calcula exactamente de la misma forma que en el entorno discreto.

5. Experimentación

Llegamos a una de las partes más fundamentales del trabajo donde realizaremos la investigación de uso de los algoritmos propuestos en el problema de optimizar la gestión de los dos escenarios con los que estamos trabajando. Únicamente nos centraremos en trabajar con el entorno discreto para ambos algoritmo y escenarios, puesto que trabajar con el escenario continuo supone una complejidad grande y no contamos con tiempo ilimitado y recursos suficientes.

En la *Ilustración 5* podemos ver un pequeño flujo del trabajo realizado para cada experimento planteado y algoritmo usado. Con esto podremos comprender en un vistazo el flujo que se ha cumplido durante la fase de experimentación. Primero se plantea el experimento donde se usará la microgrid de uno de los escenarios planteados ya sea sin realizar ninguna modificación o planteando alguna para observar posteriormente cómo se comporta el algoritmo, posteriormente se realiza la búsqueda de los mejores hiperparámetros del algoritmo para el experimento planteado y finalmente se realiza un pequeño análisis del mejor modelo encontrado para ver el comportamiento en la optimización de la microgrid y su comparación con los algoritmo tradicionales.

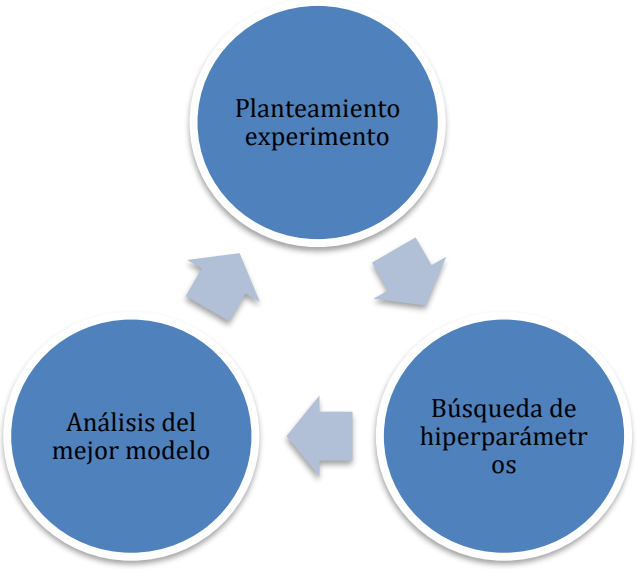

Ilustración 6 Flujo de trabajo durante la experimentación

Puesto que estamos trabajando con dos algoritmos distintos vamos a separar cada experimento en dos partes, en cada parte se usará el algoritmo DQN o PPO. Además, es necesario añadir que cada pequeña mejora que podamos conseguir respecto a los algoritmos tradicionales o a iteraciones anteriores durante el entrenamiento supone una gran ventaja

debido a que con el gran coste que supone la gestión de los escenarios planteados podríamos ahorrar una gran cantidad de dinero.

Debemos mencionar ahora el proceso para realizar la búsqueda de los mejores hiperparámetros para cada modelo, para ello nos ayudamos con la librería Ray Tune [19] donde se puede automatizar la búsqueda de hiperparámetros mediante un algoritmo de búsqueda eficiente y se establece el espacio de búsqueda de aquellos hiperparámetros que queramos establecer. A continuación, detallamos las características clave que proporciona Ray Tune y la configuración que hemos usado:

- **Ejecución distribuida:** Ray Tune permite establecer el hardware con el que contamos en nuestra máquina, donde especificaremos el número total de núcleos de la CPU y el total de GPUs disponibles. Una vez se sabe esto, se puede establecer los recursos disponibles para cada entrenamiento con lo que se puede distribuir el entrenamiento según nuestros recursos y sin duda la característica más importante es la posibilidad de entrenar varios modelos al mismo tiempo sobre la misma GPU, para ellos simplemente es necesario saber si el modelo no ocupa mucho espacio y poder establecer un porcentaje de uso de la GPU para cada entrenamiento.
- **Selección algoritmo búsqueda:** aquí es donde podemos encontrar los algoritmos de búsqueda como RandomSearch o el GridSearch, pero en nuestro caso usamos uno mucho más complejo que proporciona Ray Tune llamado Optuna. Este se trata de un wrapper sobre un framework de búsqueda de hiperparámetros se trata de un tipo algoritmo de optimización Bayesiana donde la idea principal de esta familia consiste en emplear los resultados anteriores para condicionar la búsqueda a regiones más prometedoras del espacio de búsqueda. Específicamente Optuna realiza al principio un comportamiento similar a la búsqueda aleatorio para poder recopilar información y posteriormente crea un modelo probabilístico donde se divide aquellos hiperparámetros que obtienen buenos resultados en las métricas y aquellos que no, por lo que se construye un modelo de probabilidad para aprender la distribución de hiperparámetros que podrían obtener un buen rendimiento. Con lo que en las futuras búsquedas se quiere una combinación de hiperparámetros que tenga alta probabilidad de pertenecer al grupo con buenos resultados pasados y baja probabilidad al peor grupo.
- **Selección del planificador:** con el fin de acelerar la búsqueda general se establece un planificador llamado Asynchronous Successive Halving Algorithm (ASHA) el cual se encarga de detener de forma temprana los entrenamientos que muestran un peor rendimiento.
- **Análisis de resultados:** una vez realizada toda la búsqueda se podrá ver un resumen de cada combinación de hiperparámetros y las métricas establecidas para evaluar el mejor modelo encontrado.
- **Establecer número de búsquedas:** Ray Tune nos permite establecer cuántas combinaciones de hipeparámetros queremos usar en la búsqueda.

Pasamos a continuación a presentar las acciones y la PriorityList que tiene asignada cada una en el escenario 0, puesto que estamos trabajando con un entorno discreto.

- Acción 0: PriorityList [module = (battery, 0, (module = (grid, 0)]
- Acción 1: PriorityList [module = (grid, 0), (module = (battery, 0)]

También presentamos las acciones y su PriorityList para el escenario 22:

- Acción 0: PriorityList [(module = (genset, 0, action=0), (module = (battery, 0), (module= (grid, 0)]
- Acción 1: PriorityList [(module = (genset, 0, action=0), (module= (grid, 0), (module = (battery, 0)]
- Acción 2: PriorityList [(module = (genset, 0, action=1), (module = (battery, 0), (module= (grid, 0)]
- Acción 3: PriorityList [(module = (genset, 0, action=1), (module= (grid, 0), (module = (battery, 0)]
- Acción 4: PriorityList [(module = (battery, 0), (module = (genset, 0, action=0), (module= (grid, 0)]
- Acción 5: PriorityList [(module = (battery, 0), (module = (genset, 0, action=1), (module= (grid, 0)]
- Acción 6: PriorityList [(module = (battery, 0), (module= (grid, 0), (module = (genset, 0, action=0)]
- Acción 7: PriorityList [(module = (battery, 0), (module= (grid, 0), (module = (genset, 0, action=1)]
- Acción 8: PriorityList [(module= (grid, 0), (module = (genset, 0, action=0), (module = (battery, 0)]
- Acción 9: PriorityList [(module= (grid, 0), (module = (genset, 0, action=1), (module = (battery, 0)]
- Acción 10: PriorityList [(module= (grid, 0), (module = (battery, 0), (module = (genset, 0, action=0)]
- Acción 11: PriorityList [(module= (grid, 0), (module = (battery, 0), (module = (genset, 0, action=1)]

5.1 Experimento de los escenarios con RBC

Pymgrid proporciona uno de los algoritmos más usado para la gestión de microgrids, se trata del algoritmo Rule Based Control (RBC) donde lo que se busca es crear una serie de reglas para aplicar a la gestión de una microgrid. Esta regla se crea desde un principio, el algoritmo no aprende ninguna de estas reglas como puede ocurrir con los algoritmos de aprendizaje por refuerzo.

Para el calculo de las reglas a aplicar se crea una PriorityList que consiste en una lista donde los módulos a aplicar están en orden de importancia, este orden se calcula según el coste marginal que tenga cada módulo de la microgrid, quedando así los módulos con menor coste marginal en primer lugar para aplicar (tienen más prioridad/importancia). Se tiene que agotar toda la energía que puede producir un módulo de la PriorityList para poder pasar a gestionar el siguiente módulo.

Dado los escenarios con los que estamos trabajando vamos a ver como se optimizan estos usando el algoritmo RBC y que PriorityList tiene cada uno:

- Escenario 0:

  PriorityList [(module = (battery, 0), (module = (grid, 0)]
- Escenario 22:

  PriorityList [(module = (battery, 0), (module = (grid, 0), (module = (genset, 0, action=0]

Una vez ejecutado el algoritmo sobre los escenarios podemos observar como ha resultado la estrategia en la gestión de la microgrid:

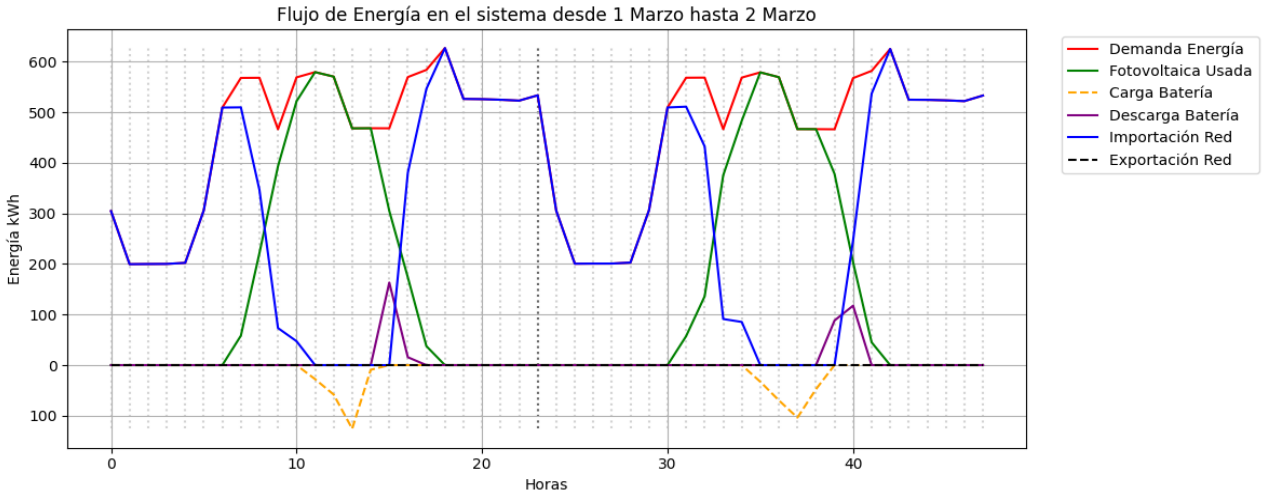

Ilustración 7 Resultado de la estrategia RBC en escenario 0

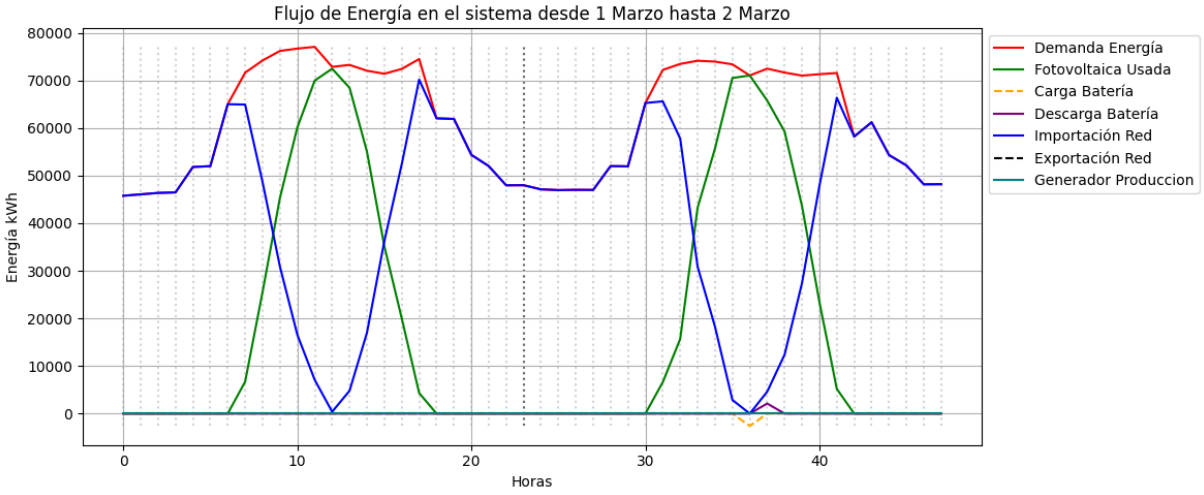

Ilustración 8 Resultado de la estrategia RBC en escenario 22

Una curiosidad ha mencionar sobre la *Ilustración 8* de como resulta al aplicar la estrategia del algoritmo RBC en el escenario 22. Es que, aunque existe la posibilidad de usar un generador para producir energía este no llega a producir nada, esto se puede relacionar con la PriorityList del mismo algoritmo puesto que el generador se encuentra como el último módulo a aplicar.

5.2 Experimento de los escenarios con DQN

En esta parte del trabajo emplearemos los escenarios planteados 0 y 22 sin ninguna modificación en su configuración y simplemente ver cómo se comportan los algoritmos para solucionar este problema respecto a los algoritmos convencionales.

Podemos detallar el espacio de búsqueda para el algoritmo DQN, este mismo espacio se usa para ambos escenarios, para realizar la búsqueda de los mejores hiperparámetros el algoritmo de búsqueda se centrará en los siguientes valores como candidatos:

- memory size: 200000 y 100000
- exploration decay: 0.99995, 0.99999
- hidden layers: [32], [64], [32,32], [64,64]
- train episode: 10, 20, 25
- target update freq: 1000, 4000, 8000

En la siguiente tabla podemos ver las combinaciones de hiperparámetros seleccionados por el algoritmo de búsqueda, para el escenario 0 *Tabla 4*:

batch size	experience decay	hidden layers	lr	memory size	target updated	train episode	total reward	mean reward
1024	0.99999	64, 64	0.001	100000	8000	10	-966027	-110.29
1024	0.99995	64	0.001	100000	8000	25	-957385	-109.303
1024	0.99995	32	0.001	100000	1000	20	-959534	-109.548
1024	0.99995	64	0.001	100000	8000	20	-956360	-109.186
1024	0.99995	64	0.001	100000	1000	25	-957964	-109.369
1024	**0.99995**	**64**	**0.001**	**200000**	**4000**	**25**	**-956130**	**-109.159**
1024	0.99999	32	0.001	200000	8000	20	-960274	-109.633
1024	0.99995	32, 32	0.001	100000	1000	20	-958006	-109.374
1024	0.99995	32, 32	0.001	100000	8000	25	-958006	-109.247
1024	0.99995	32, 32	0.001	100000	1000	25	-997841	-113.922
1024	0.99999	64, 64	0.001	100000	1000	25	-960515	-109.66
1024	0.99995	64	0.001	100000	1000	20	-957552	-109.322
1024	0.99999	64, 64	0.001	200000	4000	10	-966022	-110.289
1024	0.99999	64	0.001	200000	4000	10	-965799	-110.264
1024	0.99995	64	0.001	200000	4000	10	-956246	-109.173

Tabla 4 Búsqueda hiperparámetros DQN con escenario 0

Una vez encontrado el mejor modelo se entrena el algoritmo DQN con los mejores hiperparámetros encontrados para poder ver las gráficas de recompensa respecto a los episodios:

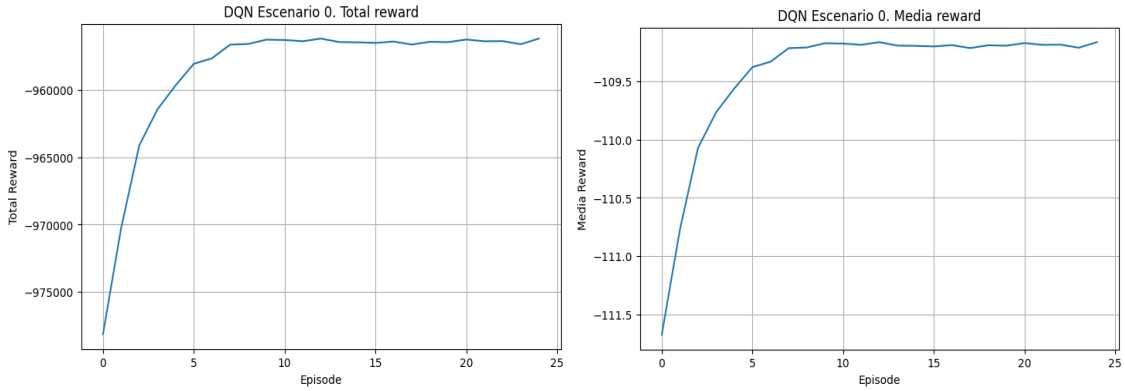

Ilustración 9 Gráficas de entrenamiento DQN en escenario 0

En la *Ilustración 9* podemos comprobar como es el proceso de aprendizaje del mejora modelo encontrado en la búsqueda de hiperparámetros, podemos comprobar que a partir del episodio 5 se produce un estancamiento de las recompensas, es decir que durante los primeros episodios si se comprueba que el algoritmo aprende a optimizar la política y posteriormente se estanca el aprendizaje, esto puede deberse a que se alcanza con el algoritmo actual la convergencia máxima que se ha quedado estancado en un mínimo local.

Es posible que no se pueda conseguir ninguna mejora más allá, puesto que igual la máxima eficiencia para la optimización de la microgrid ya se haya alcanzado.

Una vez entrenado el algoritmo podemos ver como se comporta la política aprendida en la gestión de la microgrid:

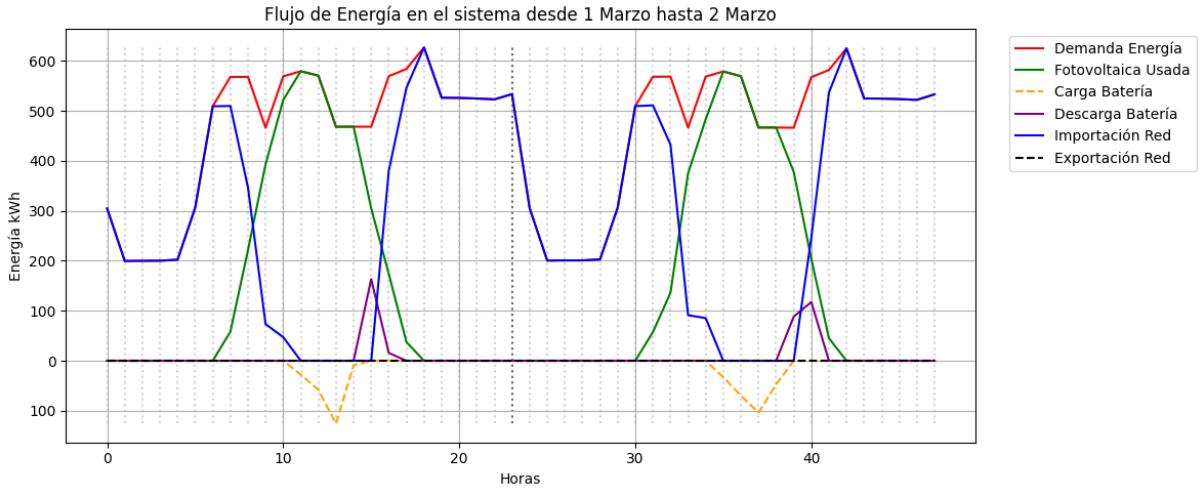

Ilustración 10 Resultado de la estrategia DQN en escenario 0

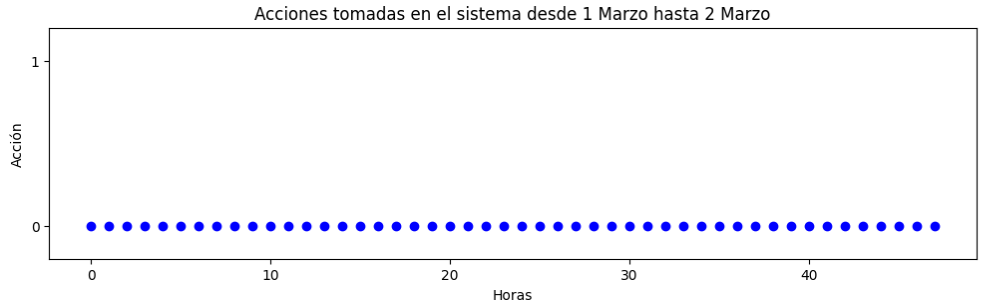

Ilustración 11 Selección de las acciones del DQN en el escenario 0

La *Ilustración 10* es exactamente la misma que podemos encontrar en la estrategia aplicada por el algoritmo RBC en el mismo escenario, esto se podrá comprobar más adelante cuando analicemos el coste monetario de gestión de ambos algoritmos. Además, podemos observar que en la *Ilustración 11* se aplica siempre la acción 0 que corresponde con aplicar primero la red eléctrica y luego usar la batería, que corresponde exactamente con la PriorityList del algoritmo RBC para el escenario 0.

En la siguiente tabla podemos ver las combinaciones de hiperparámetros seleccionados por el algoritmo de búsqueda, para el escenario 22 *Tabla 5*:

batch size	experience decay	hidden layers	lr	memory size	target updated	train episode	total reward	mean reward
1024	0.99999	64, 64	0.001	100000	8000	10	-6.93261e+07	-7914.84
1024	**0.99995**	**64**	**0.001**	**100000**	**8000**	**25**	**-4.44142e+07**	**-5039.7101**
1024	0.99995	32	0.001	100000	1000	20	-4.47972e+07	-5114.42
1024	0.99995	64	0.001	100000	8000	20	-4.47972e+07	-5114.42
1024	0.99995	64	0.001	100000	1000	25	-4.47879e+07	-5113.36
1024	0.99995	64	0.001	200000	4000	25	-4.47879e+07	-5113.36
1024	0.99999	32	0.001	200000	8000	20	-5.4577e+07	-6230.96
1024	0.99995	32, 32	0.001	100000	1000	20	-4.47972e+07	-5114.42
1024	0.99995	32, 32	0.001	100000	1000	20	-4.47972e+07	-5114.42
1024	0.99995	32, 32	0.001	100000	8000	25	-4.47879e+07	-5113.36
1024	0.99995	32, 32	0.001	100000	1000	25	-4.47879e+07	-5113.36
1024	0.99999	64, 64	0.001	100000	1000	25	-5.04677e+07	-5761.81
1024	0.99995	64	0.001	100000	1000	20	-4.47972e+07	-5114.42
1024	0.99999	64, 64	0.001	200000	4000	10	-6.93261e+07	-7914.84
1024	0.99999	64	0.001	200000	4000	25	-5.06409e+07	-5781.59
1024	0.99995	64	0.001	200000	4000	25	-4.47879e+07	-5113.36
1024	0.99995	64	0.001	200000	8000	25	-4.47879e+07	-5113.36
1024	0.99995	64	0.001	100000	8000	25	-4.47879e+07	-5113.36

Tabla 5 Búsqueda hiperparámetros DQN con escenario 22

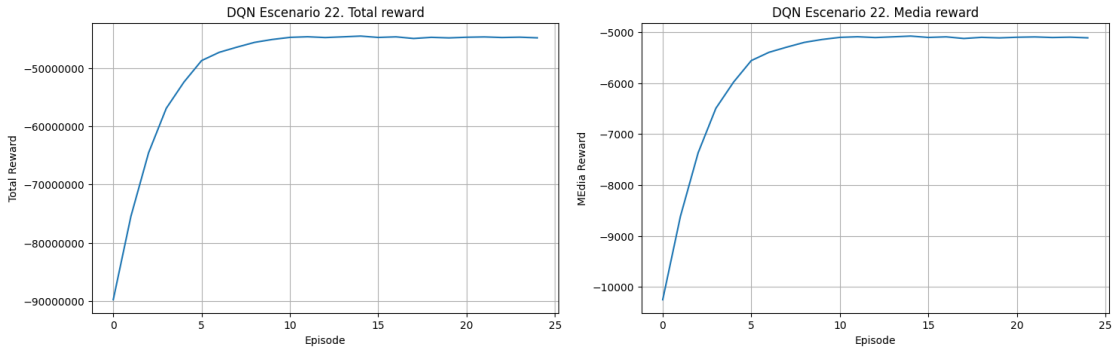

Ilustración 12 Gráficas de entrenamiento DQN en escenario 22

En la *Ilustración 12* comprobamos que ocurre exactamente el mismo proceso que con el escenario 0.

Una vez entrenado el algoritmo podemos ver cómo se comporta la política aprendida en la gestión de la microgrid:

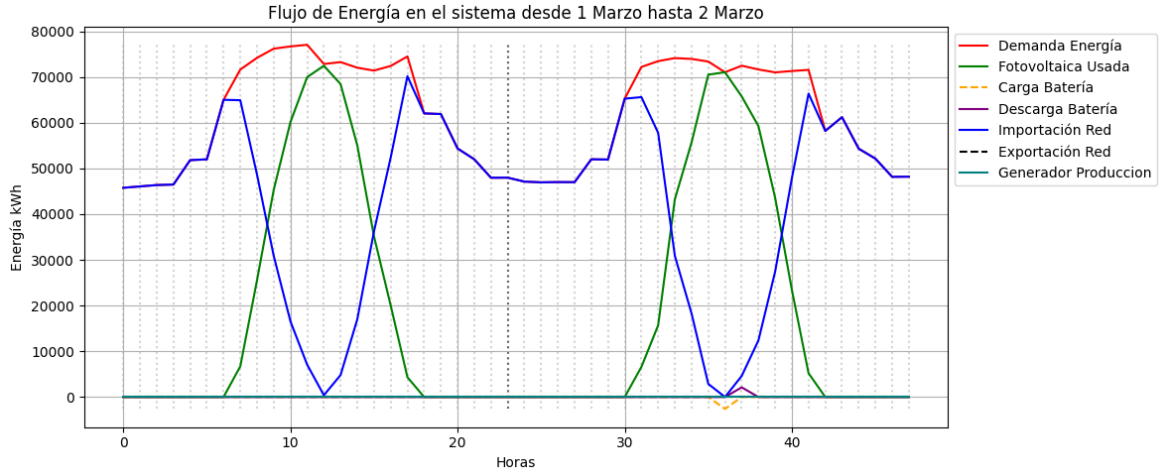

Ilustración 13 Resultado de la estrategia DQN en escenario 22

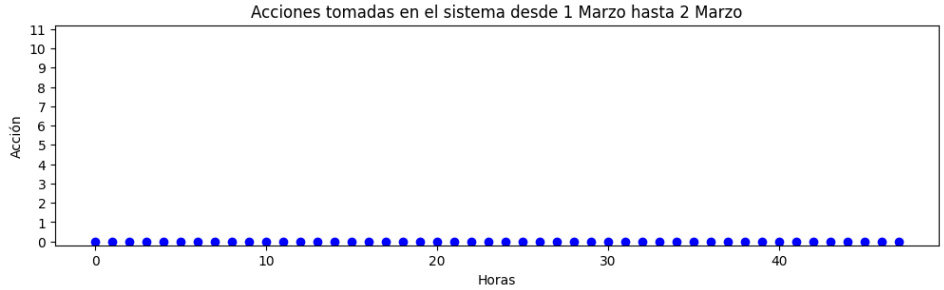

Ilustración 14 Selección de las acciones del DQN en el escenario 22

Volvemos a comprobar que al entrenar el algoritmo DQN en el escenario 22 tanto la *Ilustración 13* coincide con la grafica del algoritmo RBC para el escenario 22. Y en la *Ilustración 14* vemos que se aplica siempre la acción 0 de las 12 posibles acciones, que se trata al final de la misma idea que aplica la acción del RBC en el escenario 22. Puesto que el DQN aplicar PriorityList [(module = (genset, 0, action=0), (module = (battery, 0), (module= (grid, 0)] donde la acción 0 del modulo del generador (genset) significa apagarlo para no generar energía con él, y en el RBC se aplica PriorityList [(module = (battery, 0), (module = (grid, 0), (module = (genset, 0, action=0] la única diferencia es el orden de apagar el generador, pero los módulos que producen energía están exactamente en el mismo orden.

En la siguiente tabla (*Tabla 6*) podremos ver los resultados finales de aplicar el algoritmo DQN para el escenario 0 y 22 realizando una comparación con el algoritmo RBC.

	Rule Base Control	**DQN**	**Porcentaje de mejora**	**Beneficio de coste**
Escenario 0	Coste total: 956059.6622 Coste medio: 109.15	Coste total: 956130.4997 Coste medio: 109.1597	-0,007413 %	-70.88 €
Escenario 22	Coste total: 44142821.0118 Coste medio: 5039.71	Coste total: 44142821.0118 Coste medio: 5039.7101	aprox. 0 %	aprox. 0 €

Tabla 6 Análisis resultados DQN

Como podemos comprobar con el algoritmo DQN se consigue igualar los resultados obtenidos con el algoritmo tradicional RBC. Puesto que el porcentaje de la comparación del algoritmo DQN respecto al RBC en el escenario 0 aunque se indica que la mejora en negativa puesto que cuesta más la optimización del DQN esta puede ser considerada irrelevante al ser tan baja (la nueva optimización cuesta aproximadamente 70€ más anuales)

Más adelante en el apartado 6. Conclusiones detallaremos las posibles causas y mejoras que se podrían ejecutar para aumentar el rendimiento del algoritmo DQN.

5.3 Experimento de los escenarios con PPO

A continuación, nos centraremos en el entrenamiento y estudio con el algoritmo PPO tanto para el escenario 0 y el 22:

Podemos detallar el espacio de búsqueda para el algoritmo PPO, este mismo espacio se usa para ambos escenarios, para realizar la búsqueda de los mejores hiperparámetros el algoritmo de búsqueda se centrará en los siguientes valores como candidatos:

- total timesteps: 10*8760, 20*8760, 25*8760
- num rollout steps: 1024, 2048
- num epochs: 10, 15
- minibatch size: 32, 64
- learning rate: 0.001, 0.01
- hidden layers critic: [32], [64], [128], [64,64]
- hidden layers actor: [32], [64], [128], [64,64]

Si nos fijamos hemos establecido que el número total de interacciones con el entorno se establece como un número de episodios multiplicado por la dimensión de cada episodio, el cual siempre es 8760, esto se realiza para que el número de episodios del algoritmo PPO coincida con los del algoritmo DQN.

En la siguiente tabla podemos ver las combinaciones de hiperparámetros seleccionados por el algoritmo de búsqueda, para el escenario 0:

hidden layers actor	**hidden layers critic**	**lr**	**mini batch size**	**num epochs**	**num rollout steps**	**total time steps**	**total reward**	**mean reward**
64, 64	128	0.001	64	15	1024	175200	-958620	-109.444
64, 64	32	0.01	64	10	2048	87600	-985257	-112.485
128	128	0.001	32	10	1024	175200	-960877	-109.702
64, 64	**128**	**0.01**	**32**	**10**	**2048**	**87600**	**-956060**	**-109.152**
128	32	0.01	64	15	2048	175200	-962121	-109.844
64, 64	64, 64	0.001	32	10	2048	219000	-961493	-109.772
64, 64	128	0.001	32	15	2048	87600	-959509	-109.546
128	128	0.001	64	15	1024	87600	-959475	-109.542

128	64	0.01	32	10	1024	219000	-956060	-109.152
64	32	0.001	32	15	2048	219000	-961627	-109.787
32	128	0.01	64	15	1024	87600	-971090	-110.868
64	128	0.01	32	10	2048	219000	-990334	-113.065
32	32	0.01	64	10	2048	175200	-996054	-113.718
64	64, 64	0.01	32	10	2048	87600	-974202	-111.223
32	64	0.01	32	10	1024	219000	-963113	-109.957
32	64	0.01	32	10	1024	219000	-963113	-109.957
32	64	0.01	32	10	1024	219000	-963113	-109.957
128	64	0.01	32	10	1024	219000	-956060	-109.152
128	64	0.01	32	10	1024	219000	-956060	-109.152
128	64	0.01	32	10	1024	87600	-958333	-109.411

Tabla 7 Búsqueda hiperparámetros PPO con escenario 0

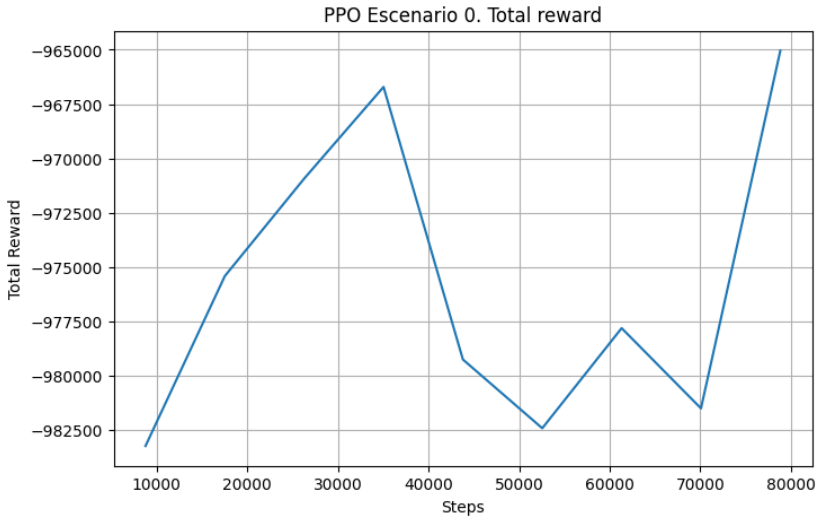

Ilustración 15 Gráfica de entrenamiento PPO en escenario 0

Como podemos ver en la *Ilustración 15* que muestra la media de las recompensas de cada step en el entrenamiento, como hemos mencionado el algoritmo PPO se entrena respecto al número total de interacciones con el entorno, por ello podemos comprobar que en la gráfica y la *Tabla 7* el mejor algoritmo se entrena hasta 87600 steps totales que sería igual a entrenar durante 10 episodios completos. Podríamos realizar con parte de la misma configuración un

entrenamiento más extenso, aunque durante la búsqueda de hiperparámetros este caso no ha sido seleccionado por el algoritmo de búsqueda.

Una vez entrenado el algoritmo podemos ver cómo se comporta la política aprendida en la gestión de la microgrid:

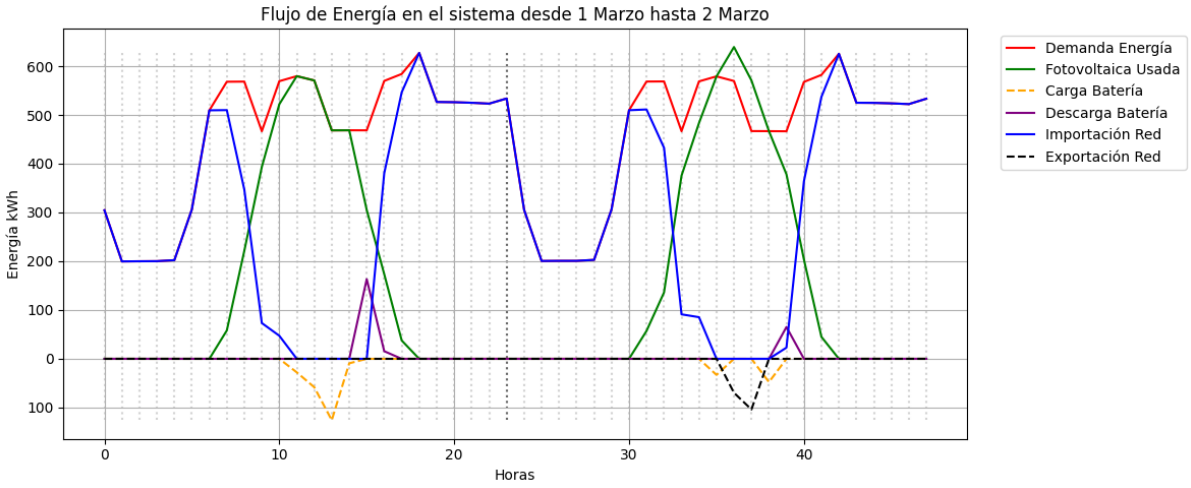

Ilustración 16 Resultado de la estrategia PPO en escenario 0

Ilustración 17 Selección de las acciones del PPO en el escenario 0

La *Ilustración 16* es muy parecida a la que encontramos en el algoritmo RBC y DQN del escenario 0, algunas cosas que cambian es la exportación de energía entre las horas 35 y 38 en vez de almacenar toda esa energía en la batería como ocurre en los otros algoritmos. Y en cuanto a las acciones de la *Ilustración 17* vemos que, si existe una gran diferencia respecto a las acciones que se aplican en el algoritmo DQN que acaba resultando en ser lo mismo que el algoritmo RBC, aunque esto no afecta al coste económico en la comparación

En la siguiente tabla podemos ver las combinaciones de hiperparámetros seleccionados por el algoritmo de búsqueda, para el escenario 22:

hidden layers actor	hidden layers critic	lr	mini batch size	num epochs	num rollout steps	total time steps	total reward	mean reward
64, 64	128	0.001	64	15	1024	175200	-5.75986e+07	-6575.94
64, 64	32	0.01	64	10	2048	87600	-6.28533e+07	-7175.85
128	128	0.001	32	10	1024	175200	-8.70955e+07	-9943.55
64, 64	128	0.01	32	10	2048	87600	-879988e+07	-10046.7
128	32	0.01	64	15	2048	175200	-2.21421e+08	-25279.3
64, 64	64, 64	0.001	32	10	2048	219000	-7.87422e+07	-8989.86
64, 64	128	0.001	32	15	2048	87600	-739994e+07	-9.132.85
128	128	0.001	64	15	1024	87600	-8.20644e+07	-9369.15
128	64	0.01	32	10	1024	219000	-4.4937e+07	-5130.38
64	32	0.001	32	15	2048	219000	-5.99163e+07	-6840.54
32	128	0.01	64	15	1024	87600	-6.20618e+07	-7085.49
64	**128**	**0.01**	**32**	**10**	**2048**	**219000**	**-4.48785e+07**	**-5123.7**
32	32	0.01	64	10	2048	175200	-1.15356e+08	-13170
64	64	0.01	32	10	1024	21900	-8.01739e+07	-9153.32
32	64	0.01	64	10	1024	175200	-9.71186e+07	-11087.9
64	64	0.01	32	10	1024	219000	-8.01739e+07	-9153.32
64	64	0.01	32	10	1024	219000	-8.01739e+07	-9153.32
64	64	0.01	32	10	1024	219000	-8.01739e+07	-9153.32
64	64	0.01	32	10	1024	219000	-8.01739e+07	-9153.32
64	64,64	0.01	32	10	2048	87600	-1.28151+e08	-14630.8

Tabla 8 Búsqueda hiperparámetros PPO con escenario 22

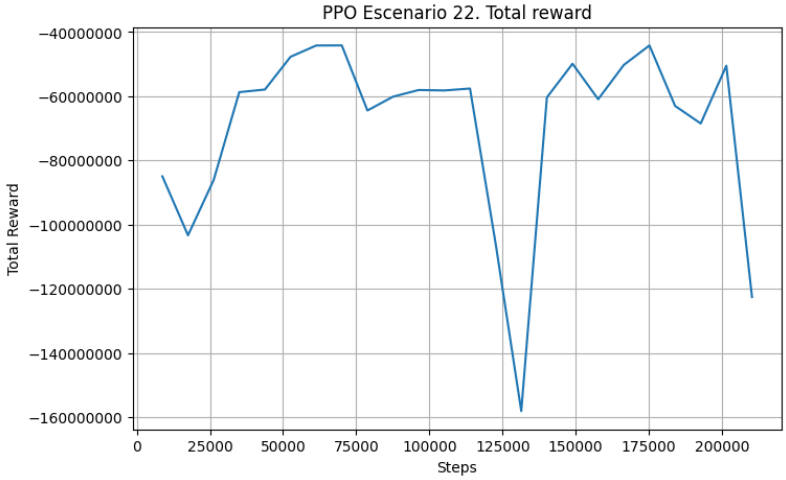

Ilustración 18 Gráfica de entrenamiento PPO en escenario 22

Como podemos ver en la *Ilustración 18* que muestra las recompensas de cada step en el entrenamiento. Podemos comprobar que en la gráfica y la *Tabla 8* el mejor algoritmo se entrena hasta 219000 steps totales que sería igual a entrenar durante 25 episodios completos en este caso el proceso de aprendizaje es más variable

Nos damos cuenta de que el la *Ilustración 18* se puede obtener un mejor resultado si simplemente modificamos el número de total_steps para en entrenamiento y dejamos fijos el resto de hiperparámetros que ya ha encontrado el algoritmo de búsqueda, por ejemplo si se entrena con 60000 steps podemos encontrar una mejora del rendimiento. Donde ahora pasamos a conseguir:

- Total reward: -44142821.011
- Mean reward: -5039.7101

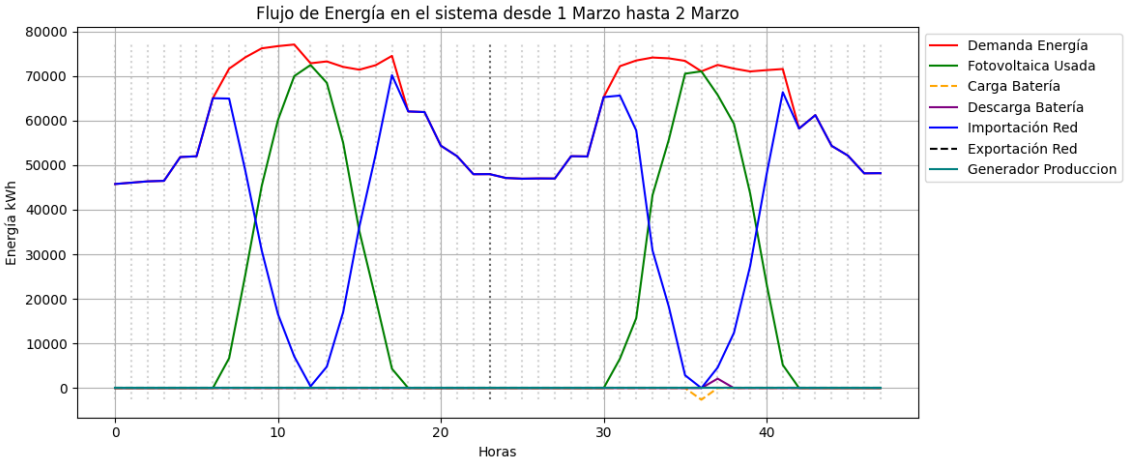

Ilustración 19 Resultado de la estrategia PPO en escenario 22

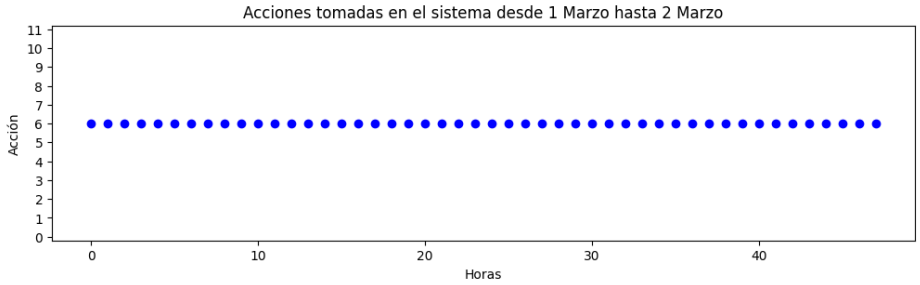

Ilustración 20 Selección de las acciones del PPO en el escenario 22

En la *Ilustración 19* comprobamos que la estrategia que se emplea obtiene los mismos resultados que en algoritmo RBC, lo podremos comprobar posteriormente cuando comparamos los resultados monetarios de esta comparación. Y la acción seleccionada durante esta prueba *Ilustración 20* vemos que se aplica siempre la acción 6 que consiste en aplicar la PriorityList [(module = (battery, 0), (module= (grid, 0), (module = (genset, 0, action=0)] que resulta ser exactamente la misma acción que aplica el algoritmo RBC por lo que es correcto que se obtengan los mismos resultados en ambos algoritmos.

En la siguiente tabla podremos ver los resultados finales de aplicar el algoritmo PPO para el escenario 0 y 22 realizando una comparación con el algoritmo RBC.

	Rule Base Control	**PPO**	**Porcentaje de mejora**	**Beneficio de coste**
Escenario 0	Coste total: 956059.6622 Coste medio: 109.15	Coste total: 956059.6622 Coste medio: 109.15	aprox 0 %	-0.34 €
Escenario 22	Coste total: 44142821.0118 Coste medio: 5039.71	Coste total: 44142821 Coste medio: 5039.7	0 %	0 €

Tabla 9 Análisis resultados PPO

En el análisis del PPO vuelve a ocurrir lo mismo que con el algoritmo DQN, se consigue igualar los resultados obtenidos con el algoritmo base RBC.

6. Conclusiones

Dado el final del trabajo nos vemos en la necesidad de hacer una recapitulación de los objetivos cumplidos y proponer futuras mejoras para aumentar el rendimiento de los algoritmos planteados o proponer algoritmos distintos que se adapten más al problema con el que trabajamos.

Se han cumplido varios de los subobjetivos propuestos inicialmente, puesto que se ha realizado un proceso de comprensión de cómo funciona el campo del aprendizaje por refuerzo puesto que previo a este trabajo no se tenía un amplio conocimiento, las ventajas que ofrece y las limitaciones que pueden afectar al correcto funcionamiento de los algoritmos.

Luego se ha estudiado en profundidad los algoritmos propuestos para comprender su funcionamiento y su implementación desde cero, este subobjetivo si se cumple con gran satisfacción puesto que se ha llegado a comprender el funcionamiento interno de cada algoritmo además de poder implementarlos desde cero incluso añadiendo mejoras de eficiencia para su entrenamiento. La implementación se realiza de forma correcta puesto que en las pruebas de testeo y con las microgrids funciona de manera correcta.

Finalmente, a la hora de realizar la comparación de los algoritmos respecto al algoritmo base que usamos como ya nos hemos dado cuenta en el apartado 5. Experimentación nos damos cuenta que no vemos una mejora optimización en cuanto al coste de la microgrid usando los algoritmos de aprendizaje por refuerzo, para llegar a comprender en profundidad la causa de estos problemas nos tendríamos que obtener más conocimientos sobre el aprendizaje por refuerzo y sus limitaciones para resolver problemas además de investigar en cada paso de tiempo porque es mejor usar las reglas establecidas por el algoritmo base que las acciones que se pueden obtener al aplicar los algoritmo planteados.

Algunas posibles mejoras que se podrían implementar para mejorar la optimización de los escenarios de las microgrids podrían ser las siguientes:

- Aumentar el espacio de búsqueda de los hiperparámetros puesto que se trata de la idea más básica de implementar en un principio y se podría conseguir una pequeña mejora en el rendimiento de ambos algoritmos esto podría suponer una mejora sustancial respecto al coste de gestionar la microgrid. Esta no se ha propuesto desde un principio puesto que, aunque se han realizado muchos esfuerzos en la optimización de los algoritmos estos requieren gran tiempo de entrenamiento y además de contar con un hardware limitado para realizar estos experimentos.

- Proponer otros algoritmos relacionados con el aprendizaje por refuerzo, como ya sabemos para cada conjunto de datos y problemas debemos encontrar el algoritmo que se adapte a este. Igual los algoritmos que hemos propuesto desde un principio no eran los más adecuados para la optimización de las microgrids. En artículo de Pymgrid ya se puede observar que ciertos algoritmos de aprendizaje por refuerzo no son los más adecuados para este tipo de problemas, puesto que para obtener una mejora respecto al RBC se usa un tipo de algoritmo que junta el Q-learning y Árboles de Decisión.

Architecture	Metric (k$)	MPC	Rule-based	Q-learning	Q-learning + DT
All	Mean cost	11,643	19,265	389,234	**13,385**
	Total cost	291,086	481,636	9,730,870	**334,624**
Genset only	Mean cost	19,722	57,398	337,385	**24,777**
	Total cost	78,890	229,593	1,349,543	**99,109**
Grid only	Mean cost	8,150	**8,372**	383,105	8,524
	Total cost	73,352	**75,350**	3,447,945	76,718
Grid + Genset	Mean cost	19,107	**22,327**	480,107	22,376
	Total cost	57,322	**66,982**	1,440,322	67,130
Weak grid	Mean cost	9,058	12,190	388,118	**10,185**
	Total cost	81,522	109,711	3,493,059	**91,666**

Ilustración 21 Comparación algoritmos en Pymgrid

- Nos podemos dar cuenta que al trabajar con microgrids podemos suponer que existe una relación entre los datos debido a su factor temporal, es lo llamado una serie temporal y es posible que los algoritmos propuestos de forma original no soporten de manera óptima trabajar con estos datos. Para ello se puede sustituir las redes neuronales profundas de los algoritmos DQN y PPO por otro tipo de arquitectura que soporte las secuencias temporales.

  Las alternativas más destacadas son las Redes Neuronales Recurrentes (RNN) puesto que se trata del tipo de arquitectura más básico para trabajar con series temporales, pero en particular con arquitecturas más complejas como las Long Short-Term Memory (LSTM) [20] y las Gated Recurrent Unit (GRU) [21], este tipo de redes neuronales están diseñadas específicamente para identificar y aprender las dependencias temporales de los datos debido a la memoria que este tipo de arquitecturas dispone. Al integrar estas redes neuronales se consigue tomar decisiones considerando no solo el estado actual de la microgrid, sino también la tendencia y los patrones históricos.

- Se podría actualizar el motor de la simulación de la librería microgrid para conseguir más flexibilidad con los módulos de energías renovables, ya habíamos mencionado que se usa primero toda la energía proporcionada por estos módulos para cubrir la demanda, si se eliminase esta restricción igual se podrían obtener mejores estrategias de gestión donde se carguen por ejemplo las baterías cuando la compra de energía a través de la

red eléctrica general sea barata y así poder usar la energía acumulada cuando la compra sea más cara. En contraparte esto añade más complejidad al problema por lo que sigue siendo posible que los algoritmos propuestos no obtengan mejoras respecto al algoritmo base usado en la comparación.

La conclusión final de este trabajo reside en la realización de que no todos los algoritmos de aprendizaje por refuerzo son adecuados para todos los problemas, existe una gran dependencia con los datos y el problema que estamos solucionando, cabe destacar que con una capacidad de cómputo y tiempo ilimitado sería necesario para obtener buenos resultados.

7. Bibliografía

[1] Silver, D., Schrittwieser, J., Simonyan, K. *et al.* (2017). Mastering the game of Go without human knowledge. *Nature* **550**, 354–359 https://doi.org/10.1038/nature24270

[2] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing atari with deep reinforcement learning. https://doi.org/10.48550/arXiv.1312.5602

[3] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., 60 Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., & Hassabis, D. (2015). Human-level control through deep reinforcement learning. Nature, 518(7540), 529–533. https://doi.org/10.1038/nature14236

[4] Watkins, C.J., Dayan, P. Technical Note: Q-Learning. Machine Learning 8, 279–292 (1992). https://doi.org/10.1023/A:1022676722315

[5] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal Policy Optimization Algorithms. *arXiv*. https://doi.org/10.48550/arXiv.1707.06347

[6] Schulman, J., Levine, S., Abbeel, P., Jordan, M. I., & Moritz, P. (2015). Trust Region Policy Optimization. *arXiv*. https://doi.org/10.48550/arXiv.1502.05477

[7] OpenAI, "Baselines — high-quality implementations of RL algorithms," GitHub repository, 2017. [Online]. Available: https://github.com/openai/baselines

[8] "The 37 Implementation Details of Proximal Policy Optimization," ICLR Blog Track (implementation checklist), Mar. 2022. [Online]. Available: https://iclr-blog-track.github.io/2022/03/25/ppo-implementation-details/

[9] GitHub Staff, "Octoverse: AI leads Python to top language as the number of global developers surges," *GitHub Blog*, 29-Oct-2024. [Online]. Disponible en: https://github.blog/news-insights/octoverse/octoverse-2024/

[10] Harris, C.R., Millman, K.J., van der Walt, S.J. et al. _ Array programming with NumPy_. Nature 585, 357–362 (2020). DOI: 10.1038/s41586-020-2649-2

[11] The pandas development team. (2025). pandas-dev/pandas: Pandas (v2.3.1). Zenodo. https://doi.org/10.5281/zenodo.15831829

[12] Hunter, J. D. (2007). Matplotlib: A 2D graphics environment. *Computing in Science & Engineering*, *9*(3), 90–95. https://doi.org/10.1109/MCSE.2007.55

[13] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., … Chintala, S. (2019). *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. https://arxiv.org/abs/1912.01703

[14] Krizhevsky, A., Sutskever, I., & Hinton, G. (2012). ImageNet Classification with Deep Convolutional Neural Networks. *Neural Information Processing Systems*, *25*. https://doi.org/10.1145/3065386

[15] NVIDIA. 2007. CUDA Technology; http://www.nvidia.com/CUDA

[16] Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B., & Shelhamer, E. (2014). *cuDNN: Efficient Primitives for Deep Learning*. https://arxiv.org/abs/1410.0759

[17] Towers, M., Kwiatkowski, A., Terry, J., Balis, J. U., Cola, G. D., Deleu, T., Goulão, M., Kallinteris, A., Krimmel, M., KG, A., Perez-Vicente, R., Pierré, A., Schulhoff, S., Tai, J. J., Tan, H., & Younis, O. G. (2024). *Gymnasium: A Standard Interface for Reinforcement Learning Environments*. https://arxiv.org/abs/2407.17032

[18] Henri, G., Levent, T., Halev, A., Alami, R., & Cordier, P. (2020). *pymgrid: An Open-Source Python Microgrid Simulator for Applied Artificial Intelligence Research*. https://arxiv.org/abs/2011.08004

[19] R. Liaw, E. Liang, R. Nishihara, P. Moritz, J. E. Gonzalez y I. Stoica, (2018) "Tune: A Research Platform for Distributed Model Selection and Training," https://arxiv.org/abs/1807.05118

[20] Hochreiter, S., & Schmidhuber, J. (1997). Long Short-Term Memory. *Neural Computation*, *9*, 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

[21] Cho, K., van Merrienboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., & Bengio, Y. (2014). *Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation*. https://arxiv.org/abs/1406.1078