TRABAJO FIN DE MÁSTER

MÁSTER EN INGENIERÍA INFORMÁTICA

Aprendizaje por refuerzo profundo para redes eléctricas inteligentes

Autor
Pablo Alfaro Goicoechea

Directores
Juan Gómez Romero
Miguel Molina Solana

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

—

Granada, 1 de noviembre de 2022

Aprendizaje por refuerzo profundo para redes eléctricas inteligentes

Autor
Pablo Alfaro Goicoechea

Directores
Juan Gómez Romero
Miguel Molina Solana

Aprendizaje por refuerzo profundo para redes eléctricas inteligentes

Pablo Alfaro Goicoechea

Palabras clave: aprendizaje por refuerzo, aprendizaje por refuerzo profundo, redes neuronales, Grid2Operate, L2RPN

Resumen

El aprendizaje por refuerzo profundo es un campo del aprendizaje automático que combina el aprendizaje por refuerzo y el aprendizaje profundo.

En los problemas de aprendizaje por refuerzo se entrena un agente para realizar una acción. Este entrenamiento lo realiza tomando decisiones sobre un entorno y observando cuales dan mejores resultados con una experimentación de tipo prueba y error. Esto se asemeja a la forma que tenemos las personas de tomar decisiones frente a situaciones nuevas o poco comunes.

Por otro lado, el aprendizaje profundo forma parte de los algoritmos de aprendizaje automático. Estos algoritmos implementan redes neuronales artificiales que se inspiran en el procesamiento y en la transmisión de información en los sistemas biológicos.

En el aprendizaje por refuerzo profundo se entrena un agente como los de aprendizaje por refuerzo, pero en este caso, se utiliza una red neuronal de aprendizaje profundo para tomar una mejor decisión y así conseguir un entrenamiento más eficiente.

En este trabajo se implementará un agente de aprendizaje por refuerzo profundo para experimentar con el problema L2RPN. Esta es una competición que busca una implementación de un agente de aprendizaje por refuerzo que modele los entornos de toma de decisiones secuenciales sobre las operaciones de la red eléctrica en tiempo real.

Deep reinforcement learning for smart grids

Pablo Alfaro Goicoechea

Keywords: deep reinforcement, deep reinforcement learning, deep learning, neural networks, Grid2Operate, L2RPN

Abstract

Deep reinforcement learning is a subfield of machine learning that combines reinforcement learning and deep learning.

In reinforcement learning problems, an agent is trained to perform an action. This training is done by making decisions about an environment and observing which ones give the best results with a trial and error experimentation. This is similar to the way people make decisions in the face of new or unusual situations.

On the other hand, deep learning is part of machine learning algorithms. These algorithms implement artificial neural networks that are inspired by the processing and transmission of information in biological systems.

In deep reinforcement learning, an agent such as reinforcement learning is trained, but in this case, a deep learning neural network is used to make a better decision and thus achieve a more efficient training.

In this work, a deep reinforcement learning agent will be implemented to experiment with the L2RPN problem. This is a competition that seeks an implementation of a reinforcement learning agent that models sequential decision-making environments about power grid operations in real time.

Yo, **Pablo Alfaro Goicoechea**, alumno de la titulación Máster en Ingeniería Informática de la **Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación de la Universidad de Granada**, con DNI **73453001V**, autorizo la ubicación de la siguiente copia de mi Trabajo Fin de Máster en la biblioteca del centro para que pueda ser consultada por las personas que lo deseen.

Fdo: Pablo Alfaro Goicoechea

Granada a 1 de noviembre de 2022.

D. **Juan Gómez Romero**, Profesor del Departamento de Ciencias de la Computación e Inteligencia Artificial de la Universidad de Granada.

D. **Miguel Molina Solana**, Profesor del Departamento de Ciencias de la Computación e Inteligencia Artificial de la Universidad de Granada.

Informan:

Que el presente trabajo, titulado ***Aprendizaje por refuerzo profundo para redes eléctricas inteligentes***,ha sido realizado bajo su supervisión por **Pablo Alfaro Goicoechea**, y autorizamos la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expiden y firman el presente informe en Granada a 1 de noviembre de 2022.

Los directores:

Juan Gómez Romero **Miguel Molina Solana**

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

1.1. Introducción al trabajo

La vida moderna cada vez depende más de la electricidad y por ello las redes eléctricas, encargadas de transportar la electricidad a través de grandes regiones, son cada vez más complejas. Las variaciones en los perfiles de demanda y producción, con la creciente integración de energías renovables, así como la tecnología de redes de alta tensión, constituyen un verdadero desafío para los operadores humanos a la hora de optimizar el transporte eléctrico evitando apagones. Para ayudar a este trabajo se ha creado el desafío L2RPN, Learning to run a Power Network [12]. Con este desafío se busca investigar el potencial de los métodos de inteligencia artificial para trabajar con las operaciones sobre las redes eléctricas. En concreto, el desafío se ha creado para fomentar el desarrollo de soluciones de aprendizaje por refuerzo para problemas clave presentes en las redes eléctricas de nueva generación.

El aprendizaje por refuerzo forma parte de la Inteligencia Artificial, más concretamente de la rama de algoritmos de aprendizaje automático. La Inteligencia Artificial es la combinación de algoritmos que son capaces de representar algunas capacidades del ser humano. Es una de las ramas más jóvenes de la informática. Surgió a partir de algunos trabajos publicados en 1940 que no tuvieron demasiada repercusión, pero en 1950 Alan Turing publicó un trabajo en el que realizó la siguiente pregunta "¿Puede pensar una máquina?", fue entonces cuando se empezó a investigar en profundidad la Inteligencia Artificial. [3]

Desde ese momento y hasta la actualidad, ha sido y sigue siendo una de las ramas de la informática que más se ha investigado y desarrollado. Este tipo de algoritmos están ya extendidos en muchos de las actividades de nuestro día a día. Estos van desde aspectos simples cómo pueden ser los recomendadores de publicaciones en redes sociales o de series en plataformas de streaming como Netflix, hasta algunos más sofisticados, que pueden llegar a crear imágenes artificiales a partir de una descripción en texto (DALL·E) [8] o modificar textos para que parezcan escritos por otros autores [4], el proyecto Maquet en concreto implementa un algoritmo que transforma un texto para que se asemeje a una novela del Capitán Alatriste de Arturo Pérez-Reverte.

1.2. Motivación

Desde que conocí el mundo de la inteligencia artificial y sus posibilidades, en la carrera, me interesé por el tema. Las aplicaciones que tienen este tipo de métodos siempre me han parecido muy atractivas. Conforme avancé en la carrera fui especializándome en estos temas y acabé desarrollando un TFG relacionado con esto. En ese trabajo desarrollé una aplicación para poder evaluar el servicio de Urgencias del Complejo Hospitalario de Navarra, en concreto del área de Traumatología. Para el TFM decidí buscar un tema que también estuviese relacionado con el mundo de la inteligencia artificial. En este caso, sobre un campo que desconocía, el del aprendizaje por refuerzo.

Cuando empecé a investigar llegué al desarrollo más mediático que había en este campo hasta la fecha, el proyecto AlphaGo de DeepMind. En este proyecto consiguieron desarrollar un agente capaz de ganar a los mejores jugadores de ajedrez, shogi (tipo de ajedrez japonés) y Go. El más importante de los tres fue el Go dada la complejidad del juego.

Otro de los proyectos más conocidos fue el de AlphaStar, también de DeepMind. En este caso, el agente estaba entrenado para jugar al juego de ordenador StarCraft II. Este es un juego multijugador complejo que se basa en la gestión de recursos. A diferencia de los juegos de mesa anteriores, en el StarCraft no tienes la información completa del "tablero", este se va descubriendo a medida que avanzas con tus tropas. Aun teniendo en cuenta las dificultades anteriores, el agente consiguió ganar a los jugadores profesionales con un marcador final de 10 a 1. En este tipo de juegos, un aspecto importante es el del número de acciones por minuto. A priori se podría pensar que, al ser un ordenador, el número de acciones sería superior, siendo esto un factor importante para conseguir ganar. Pues esto fue todo lo contrario, los jugadores humanos llegaban a hacer más acciones que el agente. Esto quiere decir que las acciones del agente eran mejores.

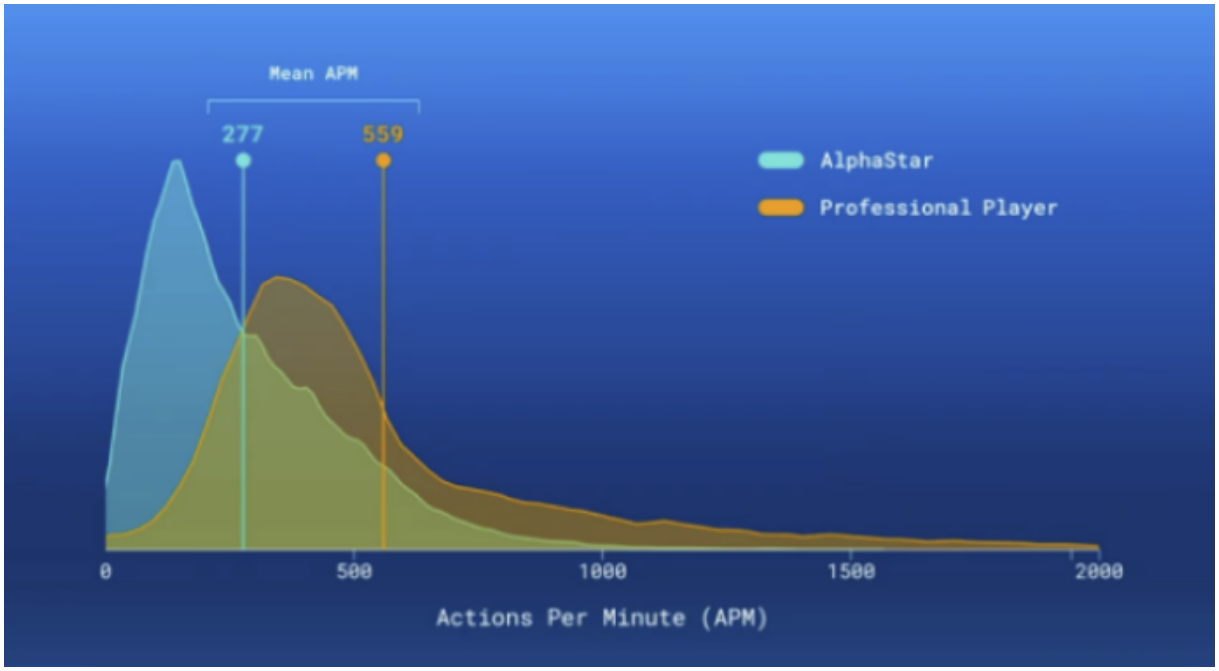


Figura 1.1: Comparación de las acciones por minuto. [18]

Como he comentado previamente, este era un campo que no había abordado aún, por esta razón y viendo las posibilidades que estos métodos ofrecen, me acabé decantando por este tema.

1.3. Objetivos

Los objetivos del trabajo son los siguientes:

1. El objetivo principal es resolver una versión simplificada de el desafío L2RPN. Para ello habrá que desarrollar un agente basado en aprendizaje profundo por refuerzo. Las librerías que más importante que se van a utilizar son Gym y Torch.

2. Formarme y profundizar sobre las técnicas de Aprendizaje por Refuerzo Profundo (DRL), conociendo las diferentes técnicas que existen y la teoría sobre las que se basan.

3. Investigar sobre el desafío L2RPN y experimentar con el framework de Grid2Op con técnicas de Aprendizaje por Refuerzo Profundo.

4. Familiarizarme con las librerías más comunes para hacer desarrollos de DRL.

5. Desarrollar el entorno de pruebas con ayuda de los servicios de la nube para realizar las labores más pesadas. En este caso serían los entrenamientos de los agentes y el servicio será Google Colab.

1.4. Planificación

El proyecto se planificó en 5 bloques de trabajo. El primero está centrado en identificar las tareas y problemas iniciales. Este primer bloque se realizará de la mano del segundo que se basa en estudiar el estado del arte, tanto del desafío L2RPN como de los algoritmos de DRL. Durante el tercer bloque se estudiará la teoría sobre la que se basan las tecnologías que se van a usar y al final del mismo se identificarán los agentes que puede ser más interesante implementar. Durante el cuarto bloque se realizará toda la experimentación y el análisis de los resultados. Finalmente el quinto bloque, el de la documentación, realizará durante todo el proyecto dejando por escrito en esta memoria el recorrido realizado.

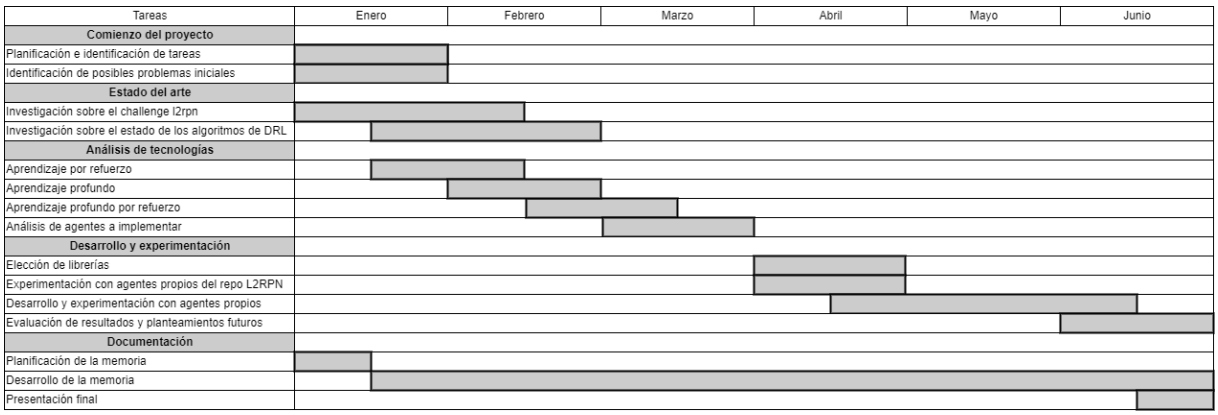


Figura 1.2: Estimación temporal

Para el presupuesto se ha tenido el cuenta el gasto de personal, el gasto en material y algunos gastos complementarios.

Se ha calculado un salario bruto de 13.000€ para el investigador del proyecto por 6 meses. A ese salario se le ha aplicado un 14 % de retención de IRPF y una cotización a la seguridad

social del 6,35 %. Adicionalmente, si se trata de un trabajo facturado por una empresa, habría que considerar los costes de cuota patronal (alrededor de un 30 % del sueldo bruto).

De la parte de material se ha presupuestado un ordenador de sobremesa de 800€ con una pantalla de 200€ y para ratón, teclado y cascos 100€. No se ha aplicado porcentaje de amortización a los equipos, que sería proporcional a la vida útil habitual (por ejemplo, 16 % si asumimos un periodo útil de 3 años).

Finalmente se han dejado 200€ para posibles imprevistos durante la realización del proyecto.

El desglose de todo lo anterior se puede ver en la siguiente imagen:

Gastos elegibles	Importe solicitado
GASTOS DE PERSONAL	13000
Total de gastos de contratación de personal investigador	13.000,00 €
Sueldo bruto (6 meses)	10.267,40 €
IRPF	1.907,10 €
Seguridad Social	825,50 €
Total de gastos de contratación de personal experto	0,00 €
GASTOS DE MATERIAL	1.150,00 €
Total material inventariable	1.100,00 €
Ordenador	800,00 €
Pantalla	200,00 €
Periféricos	100,00 €
Total material fungible	50,00 €
Consumibles	50,00 €
Total de costes de consultoría	0,00 €
GASTOS COMPLEMENTARIOS	200,00 €
Gastos de desplazamiento, viajes, estancias	0,00 €
Gastos de material de difusión, publicaciones, promoción	0,00 €
Gastos de recursos en la nube	0,00 €
Otros Gastos	200,00 €
Gastos inprevistos	200,00 €
TOTAL PRESUPUESTADO	14.350,00 €

Figura 1.3: Presupuesto.

Capítulo 2

Métodos

2.1. Introducción

El aprendizaje profundo por refuerzo forma parte de los modelos de aprendizaje automático. Estas técnicas utilizan modelos de aprendizaje profundo, es decir redes neuronales, en tareas de aprendizaje por refuerzo.

Cada vez está más extendido el uso de técnicas de aprendizaje por refuerzo. Por ejemplo, se está utilizando en comunicaciones y redes, como las de Internet de las cosas e incluso en redes de vehículos aéreos no tripulados. El aprendizaje por refuerzo ha conseguido buenos resultados cuando los entornos con los que ha trabajado han sido pequeños. Es decir, cuando la cantidad de decisiones posibles o acciones diferentes era reducida y por tanto el espacio de estados era también reducido. Sin embargo, al trabajar con problemas a gran escala los espacios de acciones y de estados aumentaba considerablemente haciendo que no se consiguiese dar una solución en un tiempo razonable. Es en este tipo de situaciones cuando es necesario trabajar con redes neuronales y por tanto, con técnicas de aprendizaje profundo por refuerzo [16].

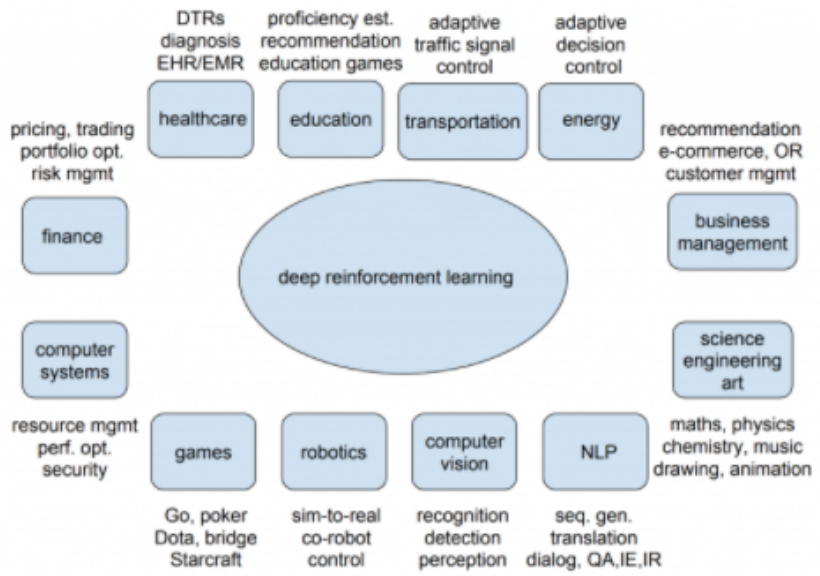


Figura 2.1: Aplicaciones de DRL

Otras aplicaciones conocidas han sido la del sistema AlphaZero de DeepMind [5], capaz de aprender a jugar a al ajedrez, al shogi (tipo de ajedrez japonés) y al Go, llegando a ganar a los campeones mundiales de estos juegos. Los motores de ajedrez tradicionales basan su estrategia en calcular miles de escenarios diseñados por jugadores capacitados con el fin de adelantarse a los posibles escenarios. Con el aprendizaje por refuerzo profundo el enfoque es totalmente diferente, se descarta el factor humano y se implementa un agente que empieza a aprender desde una posición de juego aleatorio, sin tener un conocimiento incorporado que muestre las reglas básicas del juego. Este agente aprende de cada partida que juega, ajustando los parámetros de la red neuronal para elegir los movimientos más ventajosos en cada momento. Según DeepMind, AlphaZero necesitó aproximadamente 9 horas para aprender el ajedrez, 12 horas para el shogi y 13 días para el Go.

2.2. Aprendizaje por refuerzo

El aprendizaje por refuerzo es el más común en la naturaleza. Un individuo interactúa con el entorno del cual obtiene información de las relaciones causa efecto. De esta manera, con los resultados obtenidos de las acciones realizadas, se busca una estrategia para conseguir un objetivo.

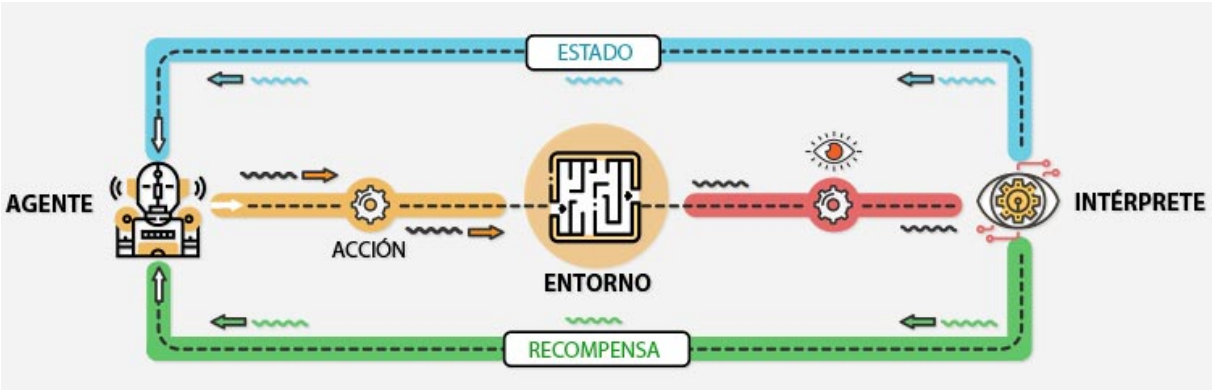


Figura 2.2: Agente de RL [6]

En la Imagen 2.2 se pueden observar los elementos que conforman un algoritmo de aprendizaje por refuerzo. En la imagen aparece un agente que observa el entorno y a través de un intérprete comprueba el estado actual del entorno. Cuando realiza una acción sobre ese entorno el intérprete le indica el estado del entorno tras realizar la acción y la recompensa. Esta recompensa indica lo cerca que está de cumplir su objetivo.

Una estrategia común a la hora de abordar estos problemas es la del Proceso de decisión de Markov(MDP). La característica de esta estrategia es que al cambiar de un estado al siguiente a través de la realización de una acción, el nuevo estado sólo dependerá del estado anterior y de la acción realizada, no de los estados o acciones llevadas a cabo anteriormente. Así, para obtener una estrategia adecuada para resolver el MDP, se considera inicialmente que todas las decisiones tienen la misma probabilidad y, a medida que se van realizando acciones en el entorno, se va actualizando la valoración que se tiene de las acciones para cada estado por los que se pasa. La forma en que se decide qué acción tomar y cómo actualizar las valoraciones es lo que distingue a los algoritmos de aprendizaje por refuerzo.

En las siguientes imágenes se puede observar el desarrollo de un problema en el que se busca que el agente llegue a la posición superior izquierda o a la inferior derecha. Tras realizar una serie de iteraciones se obtiene una política que indica las acciones óptimas para cada estado.

- (1ª evaluación de la política)

0	-1	-1	-1
-1	-1	-1	-1
-1	-1	-1	-1
-1	-1	-1	0

- (2ª evaluación de la política)

0	-1.75	-2	-2
-1.75	-2	-2	-2
-2	-2	-2	-1.75
-2	-2	-1.75	0

- (3ª evaluación de la política)

0	-2.4	-2.9	-3
-2.4	-2.9	-3	-2.9
-2.9	-3	-2.9	-2.4
-3	-2.9	-2.4	0

- (10ª evaluación de la política)

0	-6.1	-8.4	-9.0
-6.1	-7.7	-8.4	-8.4
-8.4	-8.4	-7.7	-6.1
-9.0	-8.4	-6.1	0

Figura 2.3: Inicio del proceso [7]

- (Actualización de la política)

	←	←	←↓
↑	←↑	←↓	↓
↑	↑→	↓→	↓
↑→	→	→	

Figura 2.4: Estado final [7]

Volviendo al ejemplo del ajedrez, el agente sería el "jugador máquina", el entorno sería el tablero, la acción mover una pieza, el estado sería la disposición de las piezas en un turno y la recompensa indicaría lo cerca que está de ganar. El intérprete codificaría la disposición del tablero de tal forma que el agente comprenda cómo se encuentra la partida. De está forma conoce que piezas puede mover en cada momento. En el caso de la recompensa, habrá una función que calcule en cada momento lo cerca que está de ganar. Para definir esta función se necesita conocimiento experto dado que es una parte clave del algoritmo.

2.2.1. Elementos del Aprendizaje por Refuerzo

En un algoritmo de aprendizaje por refuerzo existen cuatro elementos principales: la política, la función de recompensa, las funciones de valoración de estados y acciones y, de manera opcional, un modelo para el entorno.

2.2.1.1. Política

La política determina la manera en la que el agente toma las decisiones en cada momento, se puede considerar como la relación de los estados con las acciones que pueden ser escogidas, intentando determinar en cada caso las más óptimas. En el ajedrez, la política indicaría que pieza se ha de mover en cada caso y a que posición se debe mover.

La política se define con una función que devuelve una de las posibles acciones según el entorno en cada momento. Esta función no tiene por qué ser compleja. Un ejemplo de esto podría ser una función que escogiese la acción de manera aleatoria. Esta función, en general, no suele dar buenos resultados. Esta función se debe adaptar al problema que se quiera abordar, pudiendo llegar a establecer redes neuronales para para problemas que así lo requieran.

Según la salida que genere la política, se pueden clasificar en dos tipos: las estocásticas y las deterministas. Las estocásticas son aquellas que devuelven un listado que indica por cada acción posible la probabilidad de llevarlas a cabo en la siguiente iteración. En el caso de las deterministas, la salida es directamente la acción que el agente debe ejecutar a continuación.

2.2.1.2. Funciones de valoración

Existen varias funciones que permiten describir, evaluar y mejorar el comportamiento de la política: la función acción-valor, la estado-valor y la acción-ventaja. De forma general, todas ellas suelen expresarse en términos de esperanza matemática $\mathbb{E}$, dada la naturaleza estocástica de las acciones y de las entornos.

Función acción-valor

El objetivo de la función acción-valor es valorar la posible respuesta a que el agente realice una acción a en un estado s, teniendo en cuenta que a continuación se seguirá aplicando la política determinada π. Si la función es adecuada, se convierte en una herramienta útil para elegir la mejor acción en cada estado y con ello mejorar las políticas. Esta función trata de estimar la recompensa acumulada de una política π después de haber realizado una acción a en un estado s y continuar hasta el estado terminal según la política. Esta función se define con la siguiente ecuación:

$$q_\pi(s,a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a]$$

$$q_\pi(s,a) = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s, A_t = a]$$

Función estado-valor

La función estado-valor se utiliza para comparar las diferentes políticas. Esta función resulta de gran utilidad puesto que, una de las claves para conseguir un agente que resuelva los problemas de la mejor manera posible es tener una política óptima.

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s]$$

La función anterior se utiliza para calcular la recompensa acumulada para unos estados concretos guiándose con una política (π). Esta función también se puede reescribir de la siguiente manera:

$$q_\pi(s,a) = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s]$$

Con esta nueva reformulación es posible comparar diferentes políticas. Para poder hacerlo, se elige un problema concreto y se parte siempre del mismo estado inicial.

$$\pi > \pi' \Leftrightarrow v_\pi(s) \geq v_{\pi'}(s)$$

El objetivo ideal en la búsqueda de la política sería encontrar la óptima. Si se quiere resolver un problema de este tipo, se deberá buscar la política que obtenga unos resultados en la función estado-valor lo más cercanos a la supuesta política óptima.

$$\pi_* \geq \pi' \forall \pi'$$

Esta función estado-valor expresada de manera recursiva es conocida como la ecuación de Bellman:

$$\begin{aligned} v_\pi &= \mathbb{E}[G_t|S_t = s] \\ &= \mathbb{E}[R_{t+1} + \gamma G_{t+1}|S_t = s] \\ &= \mathbb{E}[R_{t+1} + \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s] \end{aligned}$$

La recompensa acumulada es la suma de las recompensas que se van obteniendo en cada estado hasta el final del episodio. Además, se tiene en cuenta un valor de descuento exponencial γ. Si se expresase esta recompensa acumulada como la suma de la primera recompensa más las siguientes multiplicadas por el factor de descuento anterior, volveríamos a obtener la función estado valor, obteniendo de esta forma una función recursiva.

Finalmente se puede concluir que, si se da con la política óptima, las funciones estado-valor y acción-valor serán iguales para la acción óptima que se estime en cada estado. Esto sucede ya que partimos del supuesto de que siempre elige la mejor acción:

$$\begin{gathered} v_{\pi*} = max(q_{\pi*}(s,a)) \\ a \in A(s) \end{gathered}$$

2.2.1.3. Recompensa

La función de recompensa indica lo cerca que está el sistema de cumplir con el objetivo. Después de cada acción el agente obtiene una recompensa nueva que será un valor numérico que normalmente se busca maximizar. De esta manera el agente aprende que resultados resultan beneficiosos y cuales resultan negativos. Para el problema de el ajedrez, esta función de recompensa se podría calcular por ejemplo con una combinación de la ficha que comes, las fichas que atacas, cuantas fichas tuyas son atacadas, las fichas que están defendidas...

2.2.1.4. Función de valor

En estos métodos la optimalidad se define para una política si se logra con ella un mejor rendimiento esperado en cualquier estado inicial. Una función de valor V de un estado $s \in S$ dada una política π de denota como V^π. Esta función indica el valor esperado de la recompensa R en el estado s.

$$V^\pi = e[R|s, \pi]$$

Para obtener esa optimalidad es útil definir también valores para las acciones.

$$Q^\pi(s, a) = E[R|s, a, \pi]$$

En este caso R es la recompensa está asociada a un estado s, a una acción inicial a, siguiendo con una política π. De esta manera si alguien nos da Q para una política óptima, siempre podemos optar por acciones óptimas simplemente eligiendo la acción con el valor más alto en cada estado.

Los algoritmos que aplican el método de Monte Carlo y los basados en diferencias temporales aplican los métodos de la función valor.

2.2.1.5. Modelo

El modelo del entorno es un elemento con el que el agente puede estudiar a qué estados se puede llegar realizando determinadas acciones y en cada paso se actualiza su conocimiento del modelo. Los métodos que utilizan modelos son los llamados basados en modelos y los que no los usan son agentes libres de modelos. Estos segundos son los más comunes, son los llamados agentes de prueba y error.

2.2.1.6. Exploración y explotación

Los problemas que afrontan los algoritmos de RL generalmente son de gran complejidad al tratar con espacios de acciones y estados grandes. Por estas razones, afrontar este tipo de problemas con un planteamiento iterativo se hace una tarea inviable.

Dado lo anterior, se plantea una disyuntiva a la hora de mejorar la política del agente: se explota el conocimiento ya adquirido o se exploran nuevas posibilidades. Explorando el cono-

cimiento adquirido se realizaran las mejores acciones de las que se conoce su resultado. Por el contrario, si se exploran nuevas posibilidades se realizarán acciones que a priori son subóptimas pero que pueden generar nuevos estados de los que poder aprender y desde los que se puede llegar a mejores resultados futuros. La virtud está en el término medio y, en este, caso lo mejor es optar por un equilibrio entre las dos estrategias.

Una estrategia de las más utilizadas que combina los dos planteamientos anteriores es la exploración aleatoria. El agente generalmente explotará sus conocimientos pero tiene una pequeña posibilidad de realizar una acción aleatoria cada vez que tiene que tomar una decisión. Este planteamiento se le conoce también como -Greedy. Hay variantes de este planeamiento en las que se juega con esta probabilidad de tomar decisiones aleatorias haciendo que la exploración sea más frecuente.

2.2.2. Algoritmo de Aprendizaje por Refuerzo

Existen diferentes tipos de algoritmos de aprendizaje por refuerzo. El siguiente algoritmo muestra cómo funciona aplicando una política definida.

Algorithm 1: Algoritmo básico

```
Inicializar entorno;
Inicializar política;
Inicializar funValor;
recompensaAcumulada = 0;
while no finalice el episodio do
    acción = política.elegirAcción(entorno);
    entorno.aplicar(acción);
    recompensa = entorno.recompensa();
    recompensaAcumulada = recompensaAcumulada + recompensa;
end
```

Este algoritmo tiene ya una política definida que es la encargada de indicar las acciones que se llevarán a cabo en cada momento. Empieza inicializando el entorno, la política y la función de valor, además establece a cero la recompensa acumulada obtenida. Comprueba que no se ha terminado el episodio, un episodio es uno de los intentos o en el caso de juegos partidas que realiza. Esta comprobación la hace comparando si la recompensa acumulada es igual a la función de valor o estudiando si se ha terminado el episodio, este segundo caso será cuando se haya llegado al objetivo o se haya fracasado y no se pueda continuar. Para cada iteración se empieza eligiendo la acción que se va a realizar observando el entorno en cada momento, se realizan los cambios de la acción sobre el entorno y se actualizan las recompensas con el nuevo entorno.

2.2.3. Métodos de toma de decisión

Como se ha podido observar, la dificultad de estos algoritmos radica en tomar las mejores decisiones con respecto a las acciones que se deben realizar en cada momento. Estas acciones tienen que ser las que nos conduzcan a mayores recompensas acumulativas. Para realizar esta labor existen diferentes métodos. [26]

2.2.3.1. Criterio de optimalidad

Dentro de este criterio existen dos opciones: con un mapa de acciones o con una función de valor de estado. Con un mapa de selección de acciones la política se modela de la siguiente manera:

$$\pi : A \times S \to [0, 1]$$
$$\pi(a, s) = Pr(a_t = a | s_t = s)$$

Este mapa de políticas devuelve una probabilidad de realizar cada una las acciones en un estado concreto. También hay políticas de tipo no probabilístico.

El otro tipo de método de criterio de optimalidad es el de la función de valor de estado. La función de valor $V_\pi(s)$ calcula el rendimiento esperado empezando en un estado s, que en la primera acción será el estado original, y utilizando una política π.

$$V_\pi(s) = E(R)$$
$$R = \sum_{t=0}^{\infty} \gamma^t r_t$$

La variable R se denota como return y es la suma de las futuras recompensas aplicándoles un descuento γ. Este descuento es un valor que está entre $[0, 1)$, de esta manera las recompensas de estados cercanos tienen un mayor peso que las más lejanas. La política elegida será la que de un máximo rendimiento esperado.

2.2.3.2. Fuerza bruta

Este enfoque requiere que para cada política posible se muestreen los resultados y de todas ellas se escoja la que tenga un mayor retorno esperado.

Un problema con esto es que la cantidad de políticas puede ser grande o incluso infinita y la otra es que la varianza de los rendimientos puede ser grande, lo que requiere muchas muestras para estimar con precisión el rendimiento de cada política.

Estos problemas pueden mejorarse si asumimos alguna estructura y permitimos que las muestras generadas a partir de una política influyan en las estimaciones realizadas para otras. Los dos enfoques principales para lograr esto son la estimación de la función de valor y la búsqueda directa de políticas.

Método de Monte Carlo

El concepto de este método es ejecutar una gran cantidad de episodios con lo que se obtiene un gran número de trayectorias posibles de estados y acciones. Finalmente se calcula el promedio de las recompensas acumuladas para cada estado.

Algorithm 2: Psudo-código de Monte Carlo

```
Para cada i = 0 hasta númeroEpisodios:
    Calcular trayectorias
    Actualizar (q) función acción-valor
    Obtener política mejorada a partir de esta nueva función
Calcular el promedio de recompensas acumuladas
```

La recompensa acumulada se calcula o bien obteniendo la media de los valores de recompensa para cada estado y acción, o bien eligiendo directamente el primer valor de recompensa para cada estado y acción.

Estos algoritmos tienen algunas inconvenientes:

- Puede llevar mucho tiempo alcanzar una política óptima.
- La actualización de las políticas es ineficiente, sólo se actualiza el camino por el que pasa.
- Cuando las trayectorias tienen una alta varianza, la convergencia será lenta.
- Sólo se puede aplicar a problemas episódicos.
- Sólo se puede aplicar en problemas con un conjunto de estados pequeño y por tanto finito.

Método de diferencias temporales(TD)

Estos métodos estiman la función valor en cada uno de los estados que encuentran en un episodio siguiendo una política en concreto, a diferencia de los de Monte Carlo que esperan al final del episodio para hacer la estimación. Este proceso puede ser problemático porque podría evitar la convergencia en una solución.

Algorithm 3: Psudo-código de TD

```
Para cada i = 0 hasta númeroEpisodios:
    Hasta que termine el Episodio:
        Escoger la acción a realizar según la política
        Realizar la acción
        Actualizar (q) función acción-valor
```

Estos métodos son aconsejables para procesos de decisión de Markov en los que la función de valor y la recompensa cambian en el tiempo si necesidad de una alta complejidad computacional. Al tener estas ventajas, la mayoría de algoritmos actuales de aprendizaje por refuerzo aplican estos métodos.

El método tabular TD(0) [27] es uno de los más simples. Este método estima la función de valor de estado para un MDP en un estado finito.

2.2.3.3. Búsqueda directa de la política

Este método se centra en hacer una búsqueda de un subconjunto dentro del espacio de políticas. Los métodos de este tipo están dentro de la categoría de optimización estocástica. Los de esta categoría son métodos de optimización que generan y utilizan variables aleatorias.

Este tipo de métodos consisten en resolver el problema de optimización dentro del espacio de políticas. El criterio de optimización tiene en cuenta los retornos obtenidos desde cada posible estado inicial. Cualquier técnica de optimización es válida para buscar una política óptima, pero hay que tener en cuenta que el criterio de optimización puede ser no derivable y puede contener óptimos locales. Si se quiere garantizar que la solución obtenida sea global, resulta necesario emplear algoritmos de búsqueda global en lugar de aquellos basados en el cálculo de gradiente.

Los problemas de búsqueda directa son generalmente considerados más complicados de resolver que los búsquedas de funciones de valor. El coste computacional suele ser mayor, así como la cantidad de muestras necesarias para llegar a la convergencia.

2.2.3.4. Q-Learning

El método de Q-Learning está dentro del grupo de algoritmos libres de política. En este tipo de algoritmos la estimación de la política y la política de comportamiento pueden ser distintas.

Para calcular las estimaciones de cada una de las trayectorias de cada episodio se usa la siguiente función:

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

Q-learning aprende una función de acción-valor (Q), que se aproxima directamente a la función acción-valor óptima, q, independientemente de la política seguida. De esta manera, Q-learning siempre convergerá en la política óptima.

Algorithm 4: Algoritmo Q-Learning

Inicializar cada entrada Q(s,a) de forma arbitraria, para todo s$\in$ $Syparatodoa \in A(s)$;
PARA cada episodio:
    Elegir el estado inicial de s;
    MIENTRAS no acabe el episodio:
        Elegir una acción dada la política de Q en el estado s.
        Realizar la acción a.
        Guardar los valores de recompensa r.
        Q(s,a) = Q(s,a) + $\alpha[r + \gamma max_{a'} Q(s',a') - Q(s,a)]$
        Actualizar el estado actual con el nuevo estado.
    FIN MIENTRAS
FIN PARA

2.2.4. Tipos de aprendizaje por refuerzo

2.2.4.1. Aprendizaje por refuerzo asociativo

Las tareas de aprendizaje por refuerzo asociativo combinan las facetas de las tareas de autómatas de aprendizaje estocástico y las tareas de clasificación de patrones de aprendizaje supervisado. En las tareas de aprendizaje por refuerzo asociativo, el sistema de aprendizaje interactúa en un circuito cerrado con su entorno.

2.2.4.2. Aprendizaje por refuerzo profundo

Este enfoque amplía el aprendizaje por refuerzo mediante el uso de una red neuronal profunda y sin diseñar explícitamente el espacio de estados. Como se ha comentado previamente, el trabajo de aprendizaje de juegos ATARI de Google DeepMind aumentó el interés sobre el aprendizaje por refuerzo profundo.

2.2.4.3. Aprendizaje por refuerzo inverso

En el aprendizaje por refuerzo inverso (IRL), no se proporciona ninguna función de recompensa. En cambio, la función de recompensa se infiere dado un comportamiento observado por un experto. La idea es imitar el comportamiento observado, que a menudo es óptimo o cercano al óptimo.

2.2.4.4. Aprendizaje por refuerzo seguro

El aprendizaje por refuerzo seguro (SRL) se puede definir como el proceso de políticas de aprendizaje que maximizan la expectativa de retorno en problemas en los que es importante garantizar un rendimiento razonable del sistema o respetar las restricciones de seguridad durante los procesos de aprendizaje o implementación.

2.3. Aprendizaje profundo

En el aprendizaje profundo se usan estructuras lógicas que se asemejan en mayor medida a los sistemas nerviosos de los mamíferos. Estas estructuras llamadas neuronas forman capas de unidades de proceso que se especializan en detectar determinadas características existentes en los elementos de entrada. Estos elementos de entrada pueden ir desde conjuntos de datos comunes, audios, textos, imágenes, vídeos... La visión artificial es una de las áreas donde estos métodos proporcionan una mejora considerable en comparación con algoritmos más tradicionales.

2.3.1. Redes Neuronales

Una red neuronal artificial está compuesta de una colección de nodos conectados llamados neuronas artificiales. Cada conexión entre neuronas, como las sinapsis en un cerebro biológico, puede transmitir una señal a otras neuronas. Una neurona artificial recibe una señal, la procesa y puede mandar una señal a las neuronas conectadas a ella. Esta señal es un número real y la salida de cada neurona se calcula mediante alguna función no lineal resultado de la suma de sus entradas. Las neuronas tienen un peso que se ajusta a medida que avanza el aprendizaje. El peso aumenta o disminuye el valor de la señal en una conexión. Las neuronas pueden tener un umbral, sólo si la señal agregada cruza ese umbral se envía la señal. Normalmente, las neuronas se agregan en capas. Diferentes capas pueden realizar diferentes transformaciones en sus entradas. Las señales viajan desde la primera capa (la capa de entrada) hasta la última capa (la capa de salida), posiblemente después de atravesar las capas varias veces. [25]

De manera más técnica, las neuronas de una capa están conectadas a las de la siguiente capa con un peso $w_{i,j}$. Cada neurona tiene a su vez un valor de entrada x y una función de activación $f(x)$, que se verán a continuación. El resultado de la función de activación es $y = f(x)$, este es el valor que se transmite a las siguientes neuronas. [13]

El primer nodo que se ve en la imagen es la entrada de los datos. Este nodo no trata los datos, únicamente envía la información a la primera capa de neuronas. El valor que se reciben estas neuronas se calcula con la siguiente fórmula:

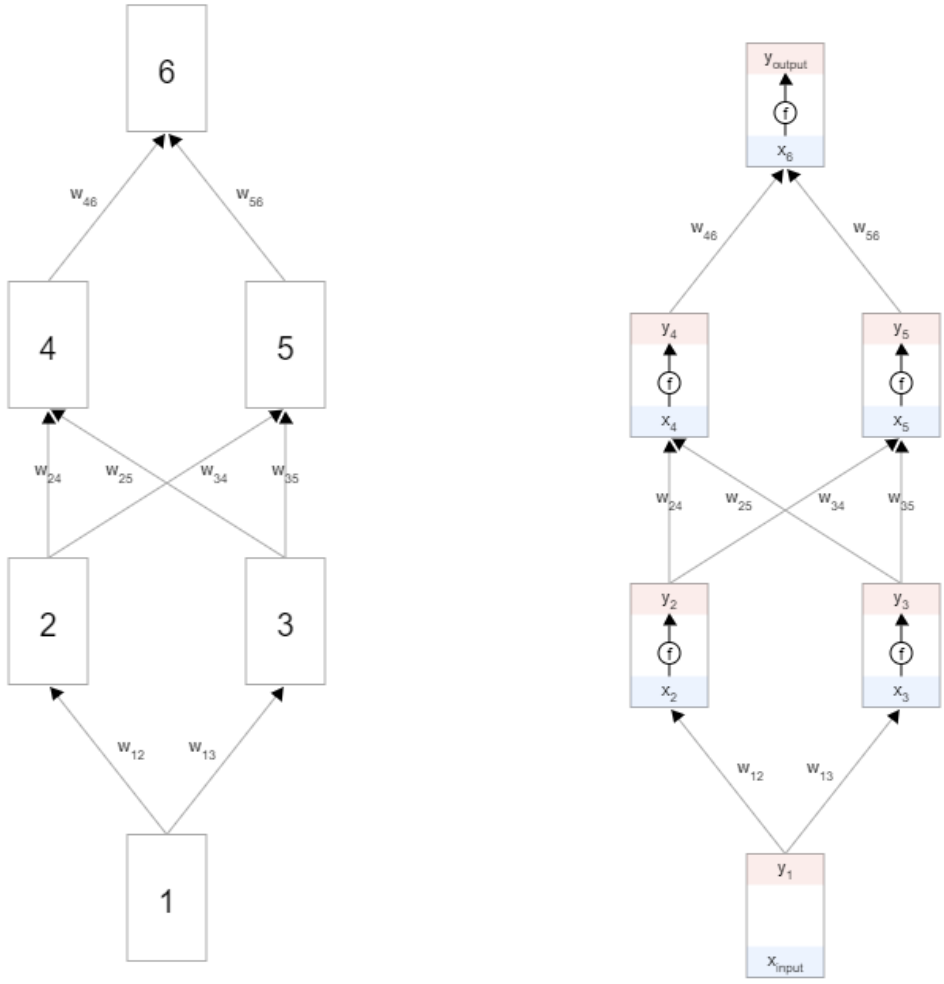
$$x_j = \sum_{i \in j} w_{i,j} y_i + b_j$$

El parámetro b_j es un parámetro de coste independiente. A continuación cada neurona calcula su y correspondiente con la función de activación que se le haya indicado. Esto se realiza hasta llegar a la última neurona. Este último resultado será la salida de la red.

2.3.2. Funciones de activación

Cada neurona procesa la entrada que recibe, esto lo realiza mediante la función de activación. Estas pueden ser de muchos tipos y determina cuando la neurona se trasmite la información o no en función de las entradas que recibe junto con su ponderación. Cada una de las capas que conforman la red neuronal tiene una función de activación que permitirá reconstruir o predecir. Las siguientes funciones de activación suelen ser las más comunes:

- **Lineal:** esta función permite que la salida sea igual que la entrada. Esta función puede ser útil si a la salida se requiere una regresión lineal y de esta manera a la red neuronal que se le aplica la función generará un valor único.



(a) Pesos de la red neuronal (b) Entradas y salidas

Figura 2.5: Cálculo de la salida de una red neuronal

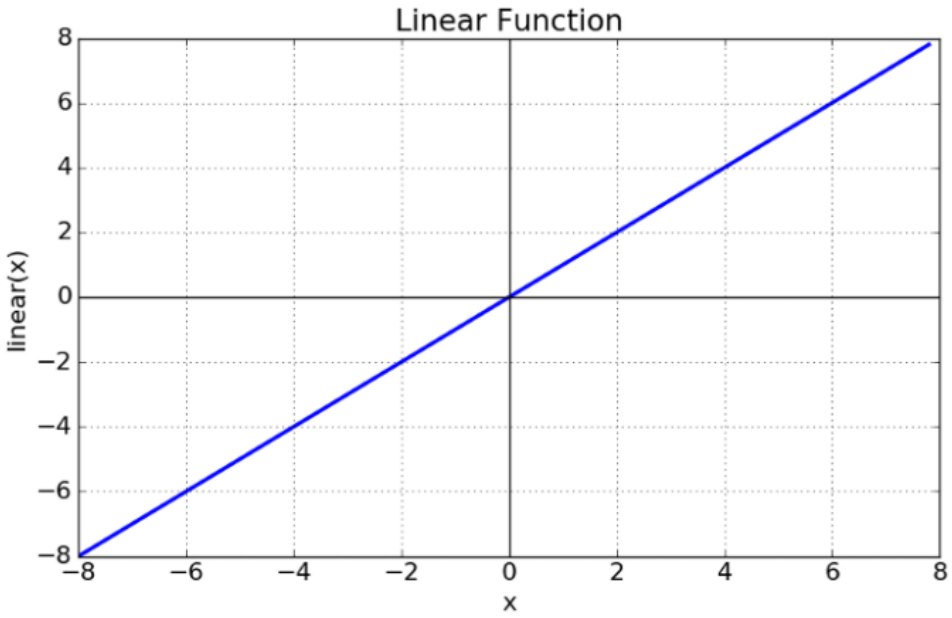


Figura 2.6: Función lineal

- **Umbral:** si el valor de entrada es menor que el umbral la salida será 0 mientras que si es mayor o igual el resultado será 1. Este tipo de funciones se utilizan cuando las salidas tienen que ser categóricas.

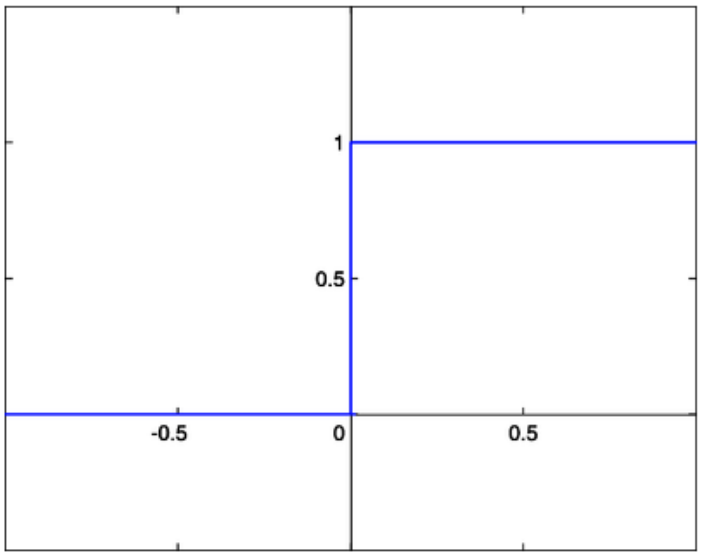


Figura 2.7: Función umbral

- **Sigmoide:** la salida que devuelve está en un rango entre 0 y 1, esta salida es iterpretada como una probabilidad. Esta función se suele usar en la última capa de problemas de clasificación para dos clases.

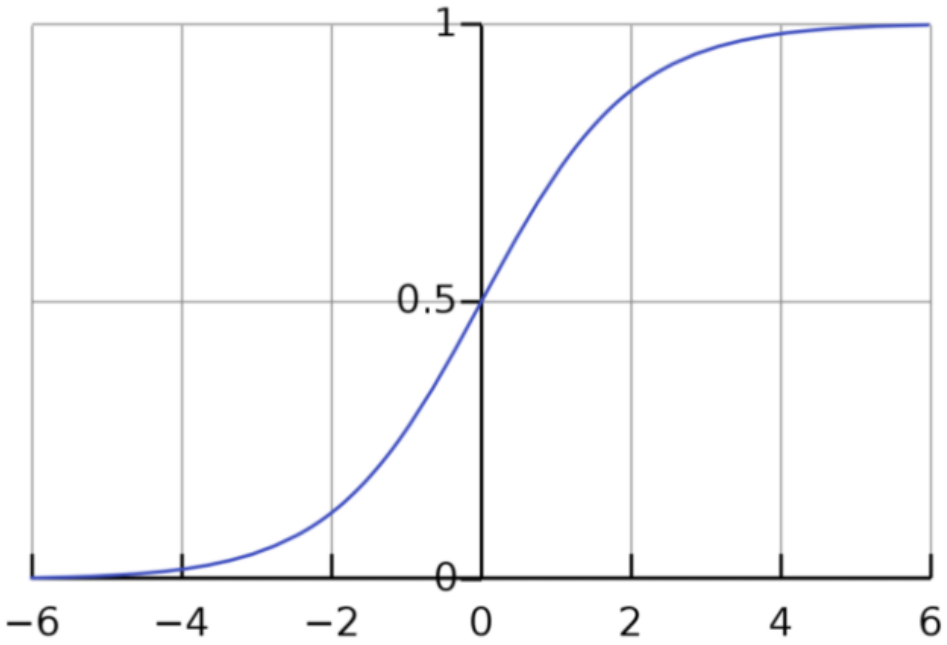


Figura 2.8: Función simoide

- **Hiperbólica:** la salida es un rango de valores entre -1 y 1.

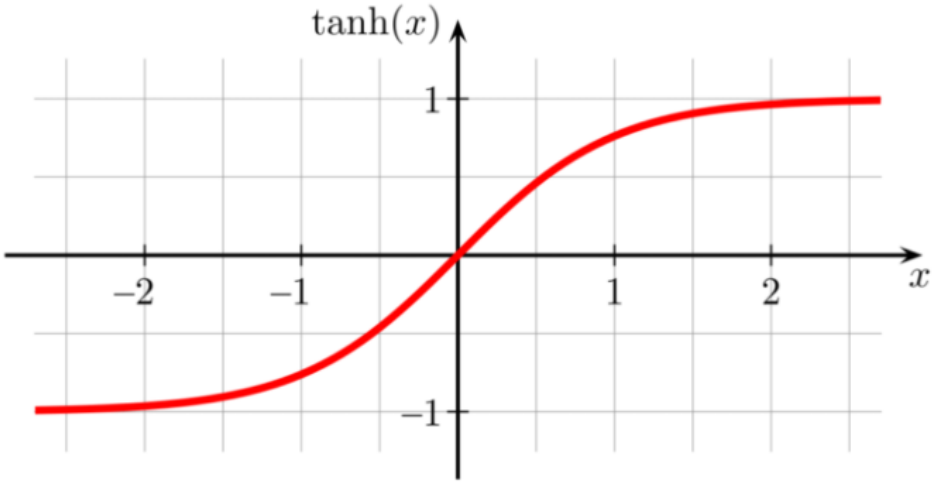


Figura 2.9: Función hiperbólica

- **Relu:** es una de las más utilizadas porque permite un aprendizaje rápido de los parámetros. Los valores de entrada negativos tienen una salida igual a 0 mientras que si el valor de entrada es positivo devuelve el mismo valor.

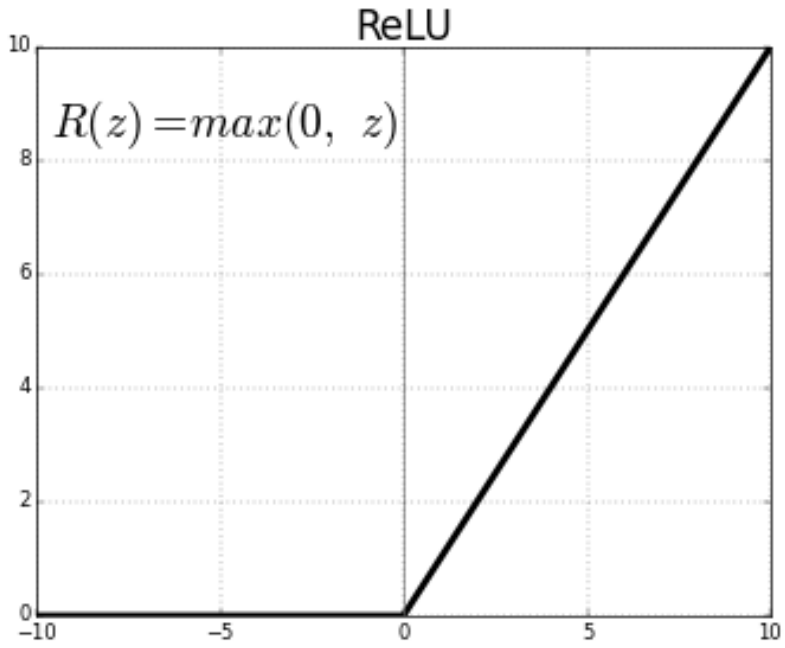


Figura 2.10: Función relu

- **Softmax:** esta es una función utilizada en problema de clasificación multiclase. La salida asigna una probabilidad de pertenencia a cada una de las clases, las sumas de estas probabilidades es igual a 1.

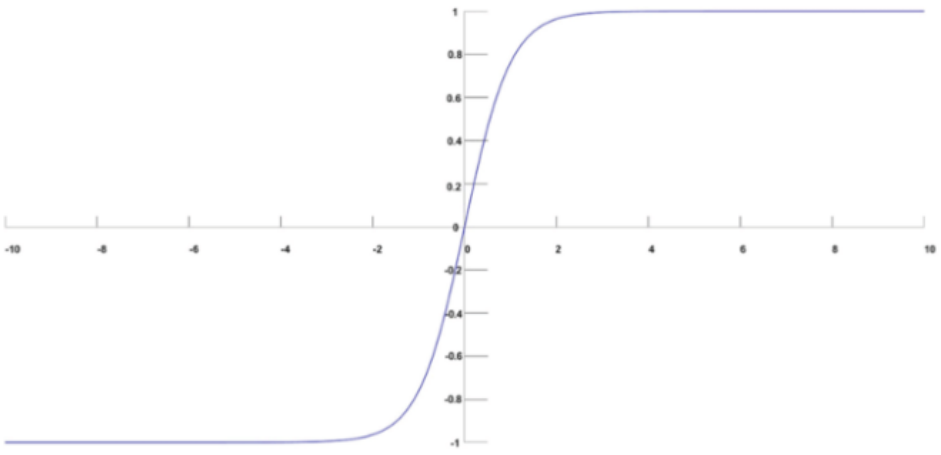
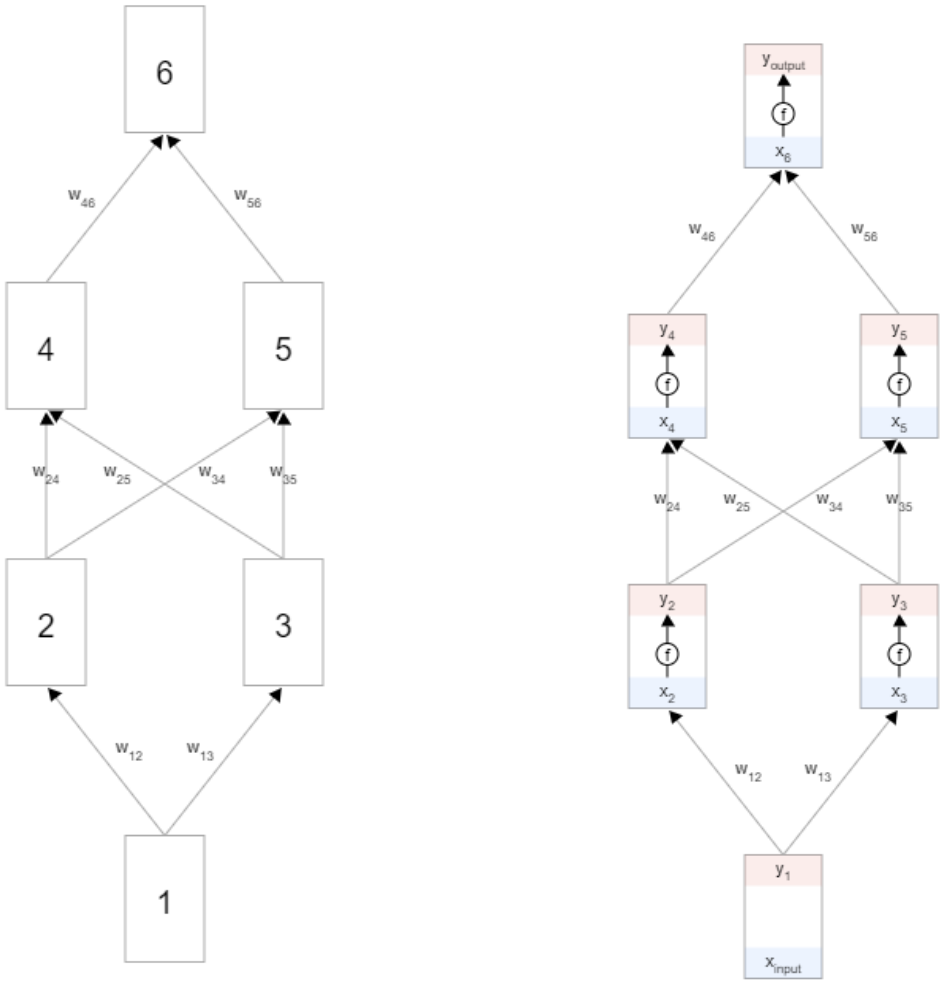


Figura 2.11: Función softmax

2.3.3. Entrenamiento

Uno de los métodos de entrenamiento más comunes y utilizados es la propagación inversa. [13]



(a) Pesos de la red neuronal

(b) Entradas y salidas

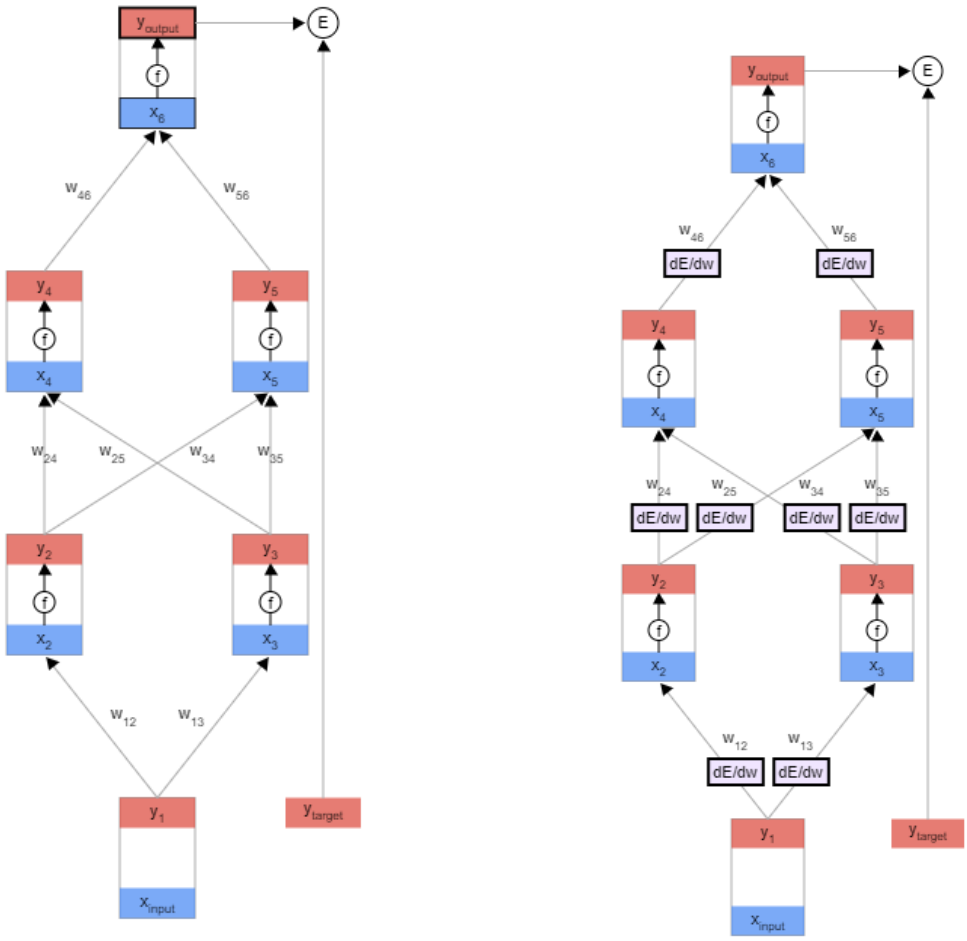
Figura 2.12: Entrenamiento de una red 1

Para poder evaluar que la elección de pesos en las conexiones es la correcta se utiliza una función de error. El objetivo es que para todos los valores iniciales de x el valor final de y que genera nuestra red sea el real. Una función de error comúnmente utilizada es la siguiente:

$$E(y_{output}, y_{target}) = 1/2(y_{output} - y_{target})^2$$

Durante el entrenamiento, los pesos de la red se actualizan después de que la red realice cada predicción. El valor de los pesos se actualiza por medio de una propagación inversa. Para hacer esto se calcula la variación del error respecto a la ponderación de los pesos en cada caso. La manera de calcular los nuevos valores de los pesos es mediante una derivada del error:

$$w_{i,j} = w_{i,j} - \alpha \frac{dE}{dw_{i,j}}$$



(a) Predicción

(b) Actualización de los pesos

Figura 2.13: Entrenamiento de una red 2

Otras medidas que se calculan son la variación del error respecto a la entrada x y la salida y en cada neurona: $\frac{dE}{dx}$ y $\frac{dE}{dy}$. El diferencial del error respecto a y_{final} se puede obtener para la última capa con la siguiente ecuación:

$$\frac{dE}{dy_{predicción}} = y_{predicción} - y_{real}$$

Con el calculo anterior realizado se puede obtener $\frac{dE}{dx}$ usando la regla de la cadena:

$$\frac{dE}{dx} = \frac{dy}{dx}\frac{dE}{dy} = \frac{d}{dx}f(x)\frac{dE}{dy}$$

En la ecuación anterior $\frac{d}{dx}f(x) = f(x)(1 - f(x))$, de esta forma se puede realizar el cálculo. El siguiente paso es calcular:

$$\frac{dE}{dw_{i,j}} = \frac{dx_i}{dw_{i,j}}\frac{dE}{dx_j} = y_i\frac{dE}{dx_j}$$

El último paso es calcular el valor de $\frac{dE}{dy}$ para la nueva capa de neuronas.

$$\frac{dE}{dy_i} = \sum_{j\in i}\frac{dx_j}{dy_i}\frac{dE}{dx_j} = \sum_{j\in i} w_{i,j}\frac{dE}{dx_j}$$

Este proceso se repite hasta actualizar todos los pesos de la red neuronal.

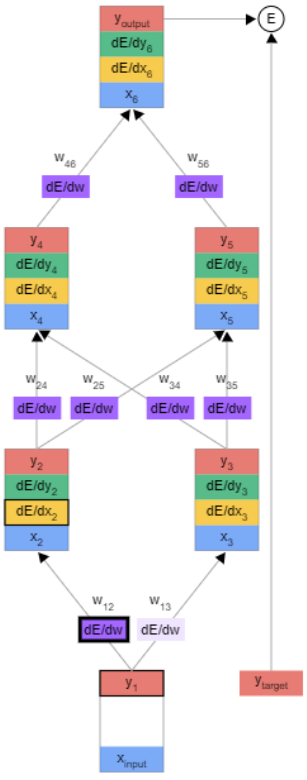


Figura 2.14: Todos los pesos actualizados

2.3.4. Convolutional Neural Network

Las redes neuronales convolucionales son las que permiten realizar clasificaciones de imágenes. Estas se utilizan en imagen médica, en la conducción de los coches autónomos y ,en general, en muchas de las tareas que lleven a cabo identificación de patrones en imágenes.

La entrada de estás redes son los píxeles de las imágenes. Cada píxel está codificado en una escala de 0 a 255 si es en escala de grises. En el caso de que sea a color cada píxel guarda 3 valores referentes al color rojo, al color verde y al color azul, estos 3 valores a su vez estarán en una escala de 0 a 255. Estos datos se normalizan antes de empezar. Este proceso convierte los valores de 0 a 255 a valores de 0 a 1 para que la red los pueda tratar de una manera más eficiente.

2.3.4.1. Capa de convolución

El procesado distintivo de este tipo de redes neuronales son las convoluciones. Este proceso consiste en recorrer la imagen tomando grupos de píxeles cercanos y operando sobre ellos. La forma de operar es calculando el producto escalar de la matriz de píxeles cercanos con una matriz del mismo tamaño llamada kernel. Con este kernel se recorre toda la imagen generando una nueva matriz. Este kernel actúa a modo de filtro y la matriz que genera muestra las características de la imagen que se busquen.

Generalmente en esta capa se aplican diferentes filtros, también llamados canales, lo que genera una matriz por cada característica de la imagen se busque. Por ejemplo, se podría usar un kernel para obtener los bordes horizontales de una imagen y otro para los verticales.

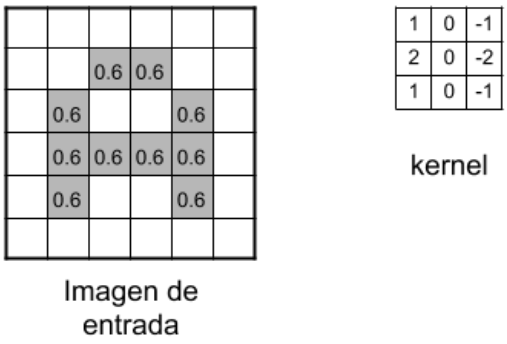


Figura 2.15: Ejemplo de imagen y kernel. [1]

2.3.4.2. Subsampling

El proceso anterior de convolución genera gran cantidad de datos. Para una imagen de 42 x 42 píxeles si se le aplican 32 de 3 x 3 kernels generas 32 matrices de 40 x 40. Si se volviera a aplicar otra capa de convolución directamente al resultado anterior se ve que los datos rápidamente serían inmanejables por su volumen. Para evitar el problema del tamaño de los datos se lleva a cabo un proceso de reducción de los datos después de cada capa de convolución. El método de reducción o Subsampling más utilizado es el de Max-Pooling.

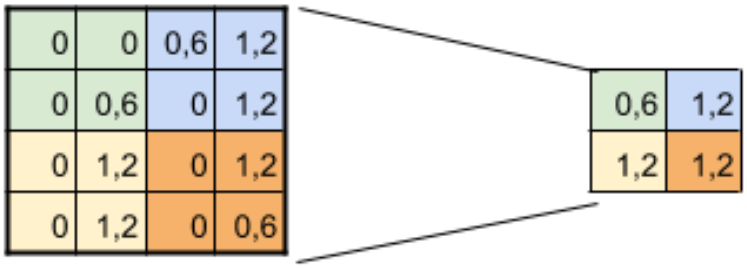


Figura 2.16: Max-Pooling. [1]

El método de Max-Pooling consiste en recorrer las matrices seleccionando agrupaciones de datos adyacentes, similar al recorrido del kernel. Por cada agrupación de datos se selecciona el de mayor valor. Si se realiza recorriendo en agrupaciones de 4 datos, como en la imagen, se reduce entre 4 el número de datos con los que se trabaja. En este proceso se pierde cierta información. Esta perdida de información puede parecer algo negativo, pero tiene ventajas además de la evidente disminución del volumen de datos. Otra ventaja es que se reduce el sobreaprendizaje al eliminar algunos detalles, esto hace que la red neuronal generalice mejor y sea más robusta ante nuevas entradas.

2.3.4.3. Capa Flatten

El trabajo de la capa Flatten es el de concatenar los resultados anteriores en un único vector de características. La capa de entrada de la parte densa tendrá una neurona por cada una de estas características.

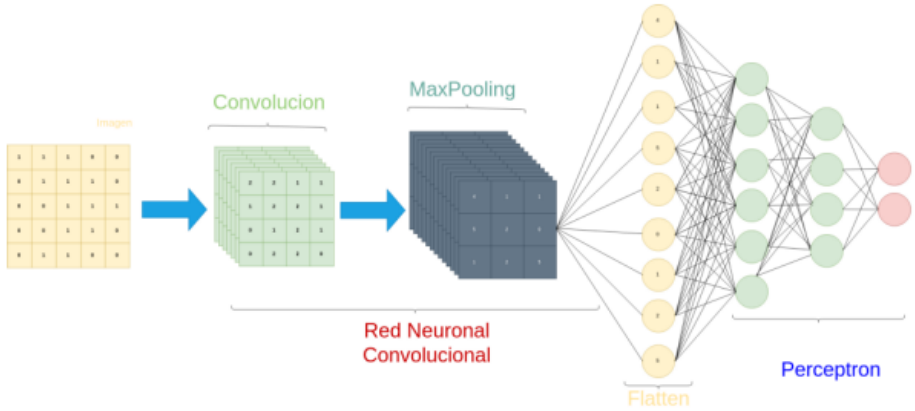


Figura 2.17: Adaptación de convolución a la parte densa de una red.

Esta capa se aplica después de una serie de capas convoluciones con sus respectivos subsamplings. El resultado de esta serie de convoluciones es una gran cantidad de canales de pequeña dimensión con características aisladas y muy concretas. Para poder unir estas características a la parte densa de la red se utiliza la capa de Flatten como se ha explicado.

2.4. Aprendizaje por refuerzo profundo

Previamente se han explicado los fundamentos del Aprendizaje por refuerzo, estos fundamentos también se aplican a los métodos de Aprendizaje por refuerzo profundo. Como se ha comentado, la característica distintiva de este tipo de métodos es que aplican técnicas de aprendizaje profundo en la búsqueda de las políticas.

2.4.1. Métodos basados en valor: Deep Q Learning

El método de Q Learning simple se basa en que se conoce la recompensa esperada de cada acción en cada situación. De esta manera el agente siempre elegirá la acción que mayor recompensa total le otorgue [10].

La idea principal de Q-learning es que predice el valor de un par estado-acción y luego compara esta predicción con las recompensas acumuladas observadas en algún momento posterior. A continuación, actualiza los parámetros de su algoritmo para que la próxima vez mejore las predicciones. Esa fue una implementación particular y simple de una clase más amplia de algoritmos Q-learning que se describe mediante la siguiente regla de actualización:

Con la siguiente ecuación se conoce el valor de Q en el estado s, realizando la acción a. El valor de $r(s, a)$ es la recompensa inmediata, a la que se le suma la recompensa esperada en el siguiente estado s'. El parámetro γ se utiliza como factor de descuento para las recompensas de estados futuros.

$$Q(s, a) = r(s, a) + \gamma max_a Q(s', a)$$

Cómo se puede ver, las recompensas futuras van decreciendo de importancia:

$$Q(s, a) \rightarrow \gamma Q(s', a) + \gamma^2 Q(s'', a)...\gamma^n Q(s'^n, a)$$

En la práctica, los valores iniciales de Q se establecen de forma arbitraria y con la experiencia acaba convergiendo hacia la política óptima. Como se ha comentado previamente, las recompensas futuras tienen menos peso que las más cercanas. Esto se produce por la forma de calcular el retorno R. Esto se muestra en la siguiente fórmula dónde se vé el método para realizar la actualización de los valores de Q:

$$Q(s, a) \leftarrow Q(s_t, a_t) + \alpha[R_{t+1} + \gamma max_a Q(s_{t+1}, a) - Q(s_t, a_t)]$$

Este planteamiento es muy potente, acaba generando una hoja de ruta para el agente dónde le indica exactamente que acciones debe realizar en cada momento. Este método tiene dos principales problemas:

- Cuando el entorno del problema tiene una cantidad de estados y de acciones grande la tabla de valores de Q es inmanejable por la cantidad de memoria que requiere.
- Teniendo la misma situación que en el punto anterior, la cantidad de tiempo necesaria para explorar todos los estados de la tabla de Q sería demasiado grande.

El planteamiento de Deep Q Learning intenta solucionar los problemas anteriores. Para ello utiliza una red neuronal que aproxima el valor de Q. Estas redes almacenan todas las experiencias

por la que ha pasado el agente, determinan la siguiente acción con la salida con un valor máximo para Q. La función de perdida es, en este caso, es el error cuadrático medio del valor de Q que predice la red y el valor de Q real.

$$E = (r + \gamma max_{a'} Q(s', a'; \theta_i') - Q(s, a; \theta_i))^2$$

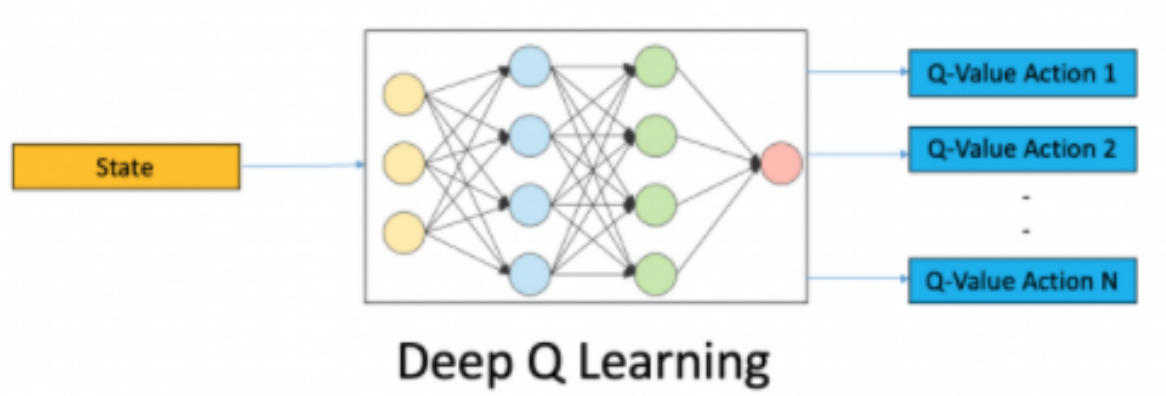


Figura 2.18: Red de Q Learning

El valor de θ es el valor de los pesos de las neuronas. El problema que surge es que no se conoce el valor objetivo real. Por lo tanto se necesita aprender a mapear para la entrada y para la salida. Dado que podría haber mucha divergencia entre estos dos valores, se utilizan dos redes neuronales para calcular cada valor.

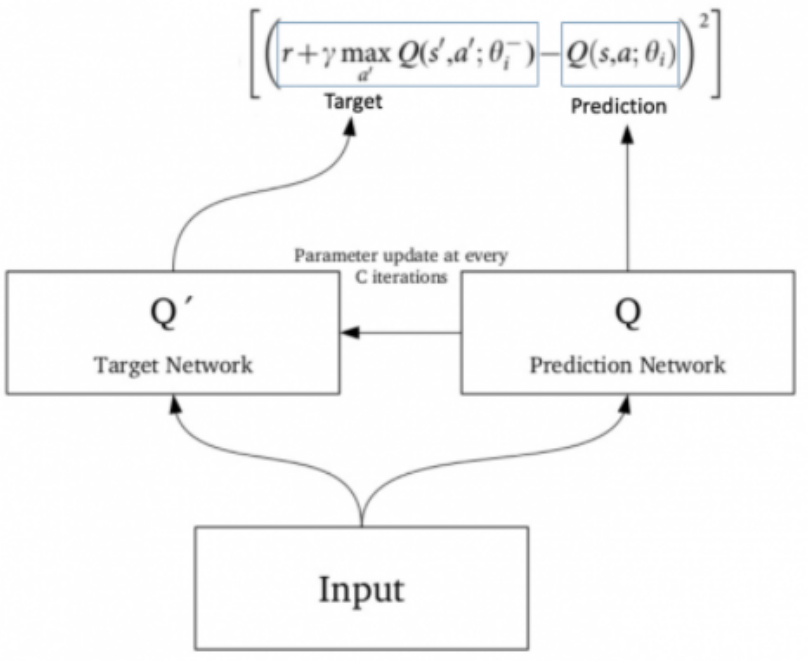


Figura 2.19: Las 2 redes de Q Learning

Después de cada iteración en la red de predicción se copian los parámetros en la red que calcula el valor objetivo de Q. Esto hace que las predicciones de la red objetivo sean más precisas.

Teniendo lo anterior en cuenta, el proceso que se sigue para entrenar el modelo es el siguiente:

1. Se le pasa a la red el estado s. Esta red nos devolverá los valores de Q para todas las posibles acciones.
2. Se selecciona la acción a usando una política greedy de probabilidad. Las acciones con un mayor valor de Q tendrán más posibilidades de ser escogidas.
3. Se realiza la acción a en el estado s y se pasa al siguiente estado s' en el que se recibe una recompensa s'. Estos datos se almacenan en lotes: ¡s,a,r,s'¿
4. Con estos lotes se calcula la pérdida usando la función mostrada anteriormente.
5. Se realiza el descenso por gradiente para ajustar los pesos de la red.
6. Después de un número indicado de iteraciones, se copian los pesos de la red real en la red objetivo.
7. Se repite el proceso hasta completar los episodios de entrenamiento.

2.4.2. Métodos basados en la política

En algoritmos de aprendizaje por refuerzo profundo basados en política $\pi(a, s)$ en lugar de buscar un valor Q, cómo se ha mostrado previamente, se busca aproximar la función de política. Esta política indica la acción que se debe realizar en cada estado. En lugar de entrenar una red para generar valores de las acciones, se entrena para que genere la probabilidad de realizar cada acción.

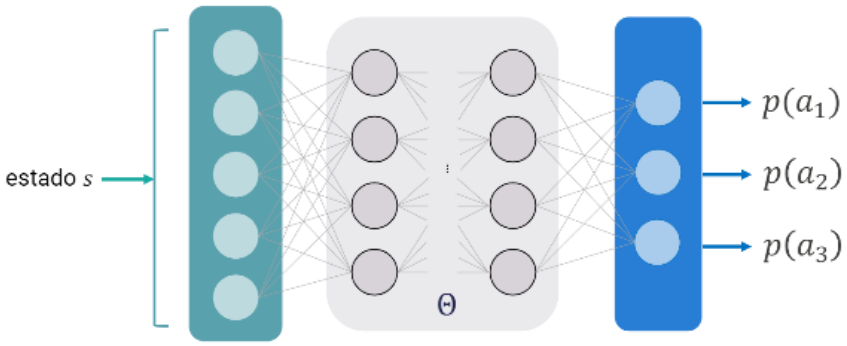


Figura 2.20: Red de modelos basados en la política [19]

Un algoritmo que aplica estos métodos es Hill Climbing. Este es un algoritmo iterativo que busca los pesos para lograr una política óptima. El algoritmo busca llegar a la cima de la colina, que en este caso será el valor de los pesos para la red que devuelvan las mejores acciones en cada caso. En concreto, para DRL se hace una adaptación de este algoritmo. La explicación que se realiza a continuación es la del algoritmo adaptado.

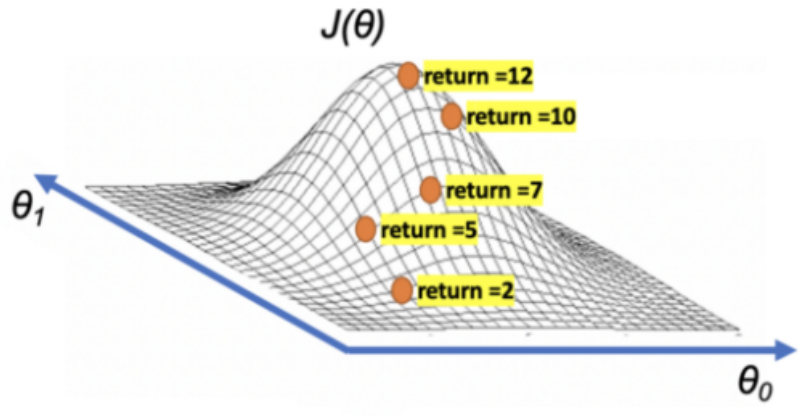


Figura 2.21: Red de modelos basados en la política [19]

El retorno que se obtiene es la estimación de la inclinación G para unos pesos determinados. Este valor puede cambiar para unos mismos pesos en episodios distintos. Esta estimación no puede ser perfecta por la aleatoriedad del entorno, pero se aproxima lo suficiente para poder

resolver el problema. El pseudocódigo sería el siguiente:

Algorithm 5: Algoritmo Hill Climbing

```
Inicializar política π con θ aleatorio;
Inicializar θ_mejor con valores de θ;
Inicializar G_mejor = 0;
while no se acaben los episodios do
    realizar un episodio con θ_actual;
    if G_actual > G_mejor then
        θ_mejor = θ_actual;
        G_mejor = G_actual;
    end
    θ_actual = θ_mejor modificado con ruido
end
```

Otro de estos métodos es el de REINFORCE [19]. Este método se basa en trayectorias en vez de en episodios, de este modo se consigue que pueda buscar políticas óptimas tanto para tareas episódicas como para continuas. La manera de calcular el gradiente descendiente para una red basada en la política, como es la de la imagen 2.21, empleando el método de REINFORCE sería usando las siguientes expresiones:

$$U(\Theta) = \sum_{x} P(x;\Theta)R(x)$$

$$\nabla_\Theta U(\Theta) \approx \frac{1}{K}\sum_{i=1}^{K}\sum_{t=1}^{n} \nabla_\Theta \log_{\pi\Theta}(A_t^{(i)}|S_t^{(i)})R(x^{(i)})$$

La función de $U(\Theta)$ es la que se trata de maximizar. La segunda ecuación es la del gradiente de $U(\Theta)$. El algoritmo genera K trayectorias $x^{(i)}$ de longitud n utilizando la política π_Θ. A continuación, calcula la recompensa R^i para cada $x^{(i)}$. Finalmente, actualiza los pesos Θ para estimar el gradiente de $U(\Theta)$.

Algorithm 6: Algoritmo de REINFORCE. [24]

```
Inicializar los parámetros de θ;
while no converja do
    Obtener la trayectoria t usando π_θ
    Se estima el retorno para la trayectoria t calculada;
    Se utiliza la trayectoria t para estimar el gradiente;
    Se actualizan los parámetros de θ
end
Se devuelve π
```

2.4.3. Métodos actor-critic

Estos métodos son una combinación de los dos anteriores. Los métodos basados en valor van estimando el valor de recompensa, pero tienen el problema de que esas estimaciones se hacer a partir de otras estimaciones previas por lo introducen un sesgo en el aprendizaje. Por otro lado, los métodos basados en política actúan realizando las acciones que conduzcan a un buen resultado final. Estos métodos tienen el problema de que puede haber trayectorias que lleven a un mal resultado final puede contener buenas acciones.

Los algoritmos de actor-critic implementan dos redes: una llamada actor que es similar a la de REINFORCE y una de critic similar a la de DQN. El actor selecciona la mejor acción en cada estado, mientras que el critic realiza una estimación de la recompensa acumulada a partir del estado en el que se encuentra. Esta implementación tiene dos principales ventajas, acelera el aprendizaje y evita los sesgos que podíamos tener en DQN.

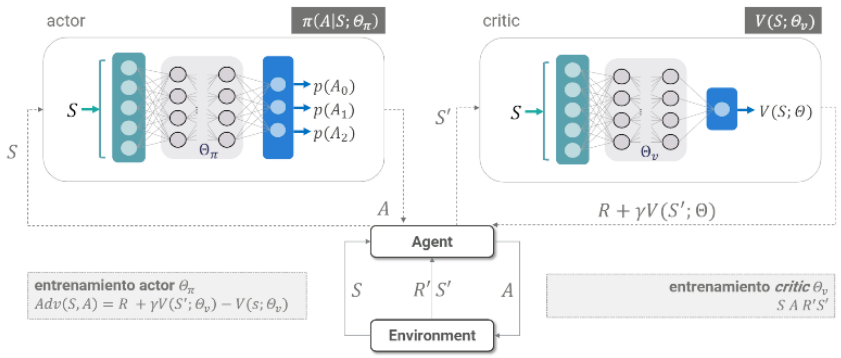


Figura 2.22: Redes de actor-critc [19]

El actor recibe un estado S y devuelve una acción A al agente. El agente realiza la acción A y para al estado S'. Este estado S' lo recibe el critic, el cual realiza una estima de la recompensa acumulada esperada. Este valor es el que le llega al agente.

Capítulo 3

Caso de uso y Herramientas

3.1. El desafío L2RPN: gestión de redes eléctricas

Las redes eléctricas son infraestructuras cada vez más complejas de las cuales la vida moderna depende críticamente. Las variaciones en la demanda y producción, con la creciente integración de energías renovables y la complejidad de las tecnologías de las actuales redes eléctricas, constituyen un verdadero desafío para los operadores a la hora de optimizar el transporte eléctrico evitando apagones. Con el objetivo de investigar el potencial de los diferentes métodos de inteligencia artificial para permitir la adaptabilidad en las operaciones sobre la red eléctrica, se ha diseñado el desafío L2RPN. Este desafío fomenta el desarrollo de soluciones de aprendizaje reforzado para problemas clave presentes en las redes eléctricas de nueva generación.

Para alcanzar los objetivos climáticos, las redes eléctricas dependen cada vez más de la generación de energía renovable y eso provoca una descentralización en su producción. Las herramientas tradicionales utilizadas por los ingenieros eléctricos para resolver los problemas de red se están quedando cada vez más obsoletas.

Existen algunas innovaciones prometedoras y aplicaciones recientes de técnicas de aprendizaje automático que pueden ofrecer soluciones a este desafío. Sin embargo, existía una limitación provocada pro la falta de entornos utilizables, datos, redes y simuladores. El desafío "Learning to run a Power Network" (L2RPN) es una serie de competiciones que modelan los entornos de toma de decisiones secuenciales de las operaciones de la red eléctrica en tiempo real. Estas competiciones están respaldadas por el ecosistema de código abierto GridAlive [22] y el framework de Grid2op [21].

El desafío de L2RPN tenía como objetivo ser un punto de referencia para soluciones al problema de las operaciones de red complejas fácilmente disponibles a través del framework Grid2Op. El marco Grid2Operate ha creado conciencia sobre el desafío más allá de la comunidad del sistema de energía. Además, los desafíos presentan una oportunidad para que la comunidad de IA pruebe los avances recientes. Estos avances podrían encontrar aplicaciones en otros problemas del mundo real yendo más allá de entornos de juego. Otro objetivo es concienciar a la comunidad del sistema de energía sobre las innovaciones y el potencial de los algoritmos de IA y aprendizaje por refuerzo para resolver los desafíos de la red.

3.2. Grid2Operate

Grid2Operate es el entorno de simulación de código abierto se basa el desafío L2RPN, se ejecuta con el backend del simulador de red eléctrica de pandapower e implementa la API de OpenAI Gym. Grid2Operate modela los conceptos sobre los que se realizan las operaciones del mundo real. Estos conceptos son los que se utilizan para probar algoritmos de control avanzados.

Aunque el problema de controlar los sistemas de energía es nuevo para la comunidad de IA, se ha intentado garantizar que los usuarios puedan interactuar con él de una manera familiar. El framework permite una fácil manipulación de la red eléctrica utilizando el marco de aprendizaje por refuerzo OpenAI Gym. El marco Grid2Operate viene con múltiples entornos ya disponibles para pruebas. Estos entornos varían en tamaño (desde un sistema de inicial que contiene 5 subestaciones, hasta sistemas con 118 subestaciones) y dificultad.

Además del material anterior, también ponen a disposición de los usuarios algunos scipts de ejemplo. El código de referencia es código abierto, disponible para realizar importaciones por parte de los participantes. [21]

Es un framework basado en Python, para poder desarrollar, entrenar o evaluar las actuaciones de un "agente" que actúa sobre un powergrid de diferentes formas. Es modular y se puede utilizar para entrenar al agente de aprendizaje por refuerzo o para evaluar los rendimientos de algoritmos de control. Abstrae la modificación de una red eléctrica y usa esta abstracción para calcular, por ejemplo, los fallos en cascada resultantes de la desconexión de las líneas eléctricas.

El objetivo de grid2operate es modelar la "toma de decisiones secuencial"que podrían tomar los operadores humanos, por ejemplo, cambiando la configuración de algunas "subestaciones".

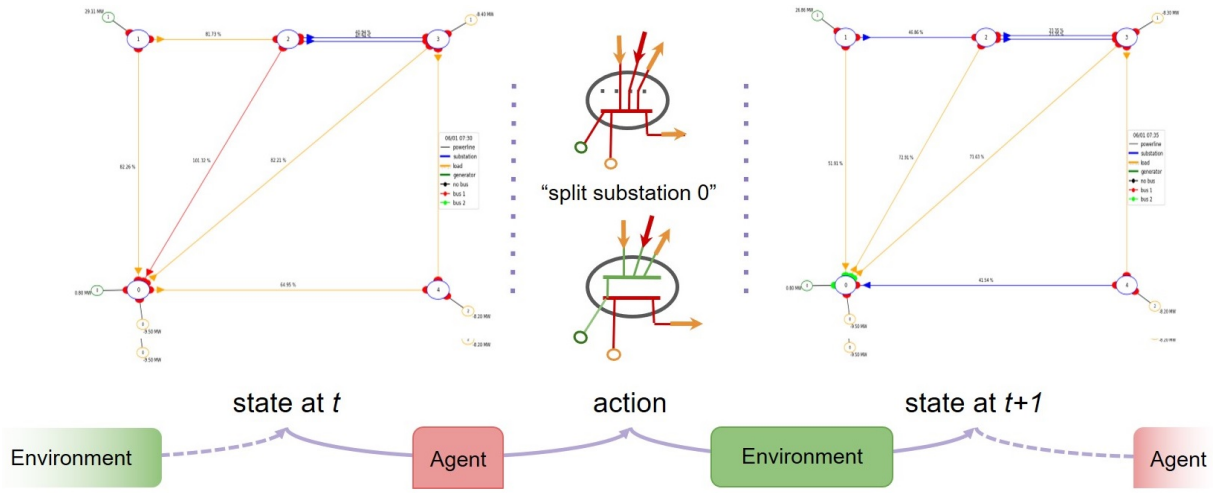


Figura 3.1: Operación sobre un entorno de red. [11]

El agente puede actuar sobre:

- Las inyecciones:
  - La potencia de producción de los generadores activos.
  - El voltaje de los generadores.
  - El consumo de energía de los centros de consumo.
  - El consumo de energía reactiva de los centros.
- El estado de las líneas eléctricas:
  - Conectada
  - Desconectada
- La configuración de las subestaciones

Para entender lo anterior, hay que conocer previamente cómo están compuestas las redes que conforman los entornos. Fijándonos en los entornos de la imagen anterior, se puede apreciar, si pensamos en el entorno cómo si fuese un grafo, que los nodos son las subestaciones y las aristas las redes de electricidad. Estos nodos o subestaciones están numerados y contienen información de 3 tipos de edificios ligados a ellos:

- Las plantas generadoras de electricidad.
- Los consumidores. Estos serían los núcleos urbanos o polígonos industriales por ejemplo.
- Las zonas de almacenamiento de electricidad.

Además de lo anterior, en la leyenda se observa unos elementos llamados buses. Los enlaces a las subestaciones por parte de los edificios o de las redes de electricidad se realiza mediante un bus. A estos se les puede asignar el bus 1 o el bus 2. Las conexiones que tengan asignado el mismo bus estarán en contacto.

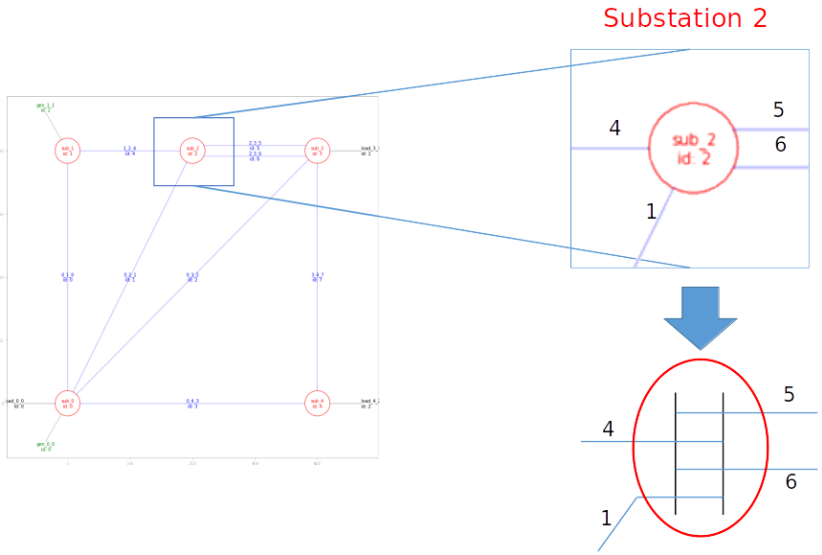


Figura 3.2: Esquema de subestación

En la imagen anterior se puede ver una subestación a la que le llegan las redes 1, 4, 5 y 6. A continuación se muestra el esquema de switches que se tendría en este caso.

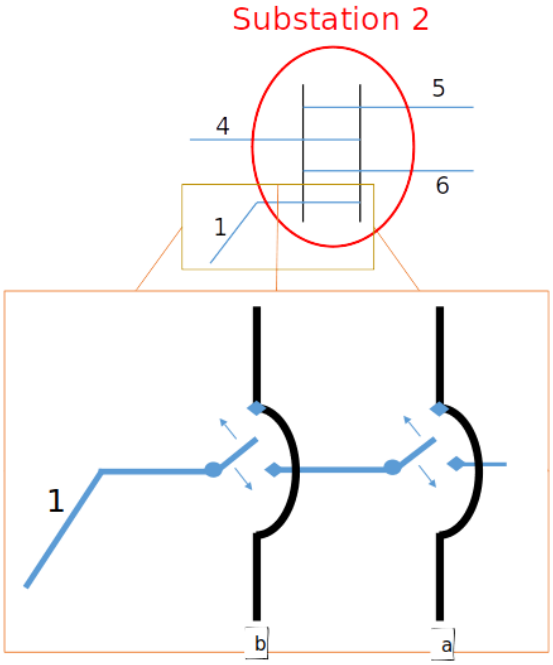
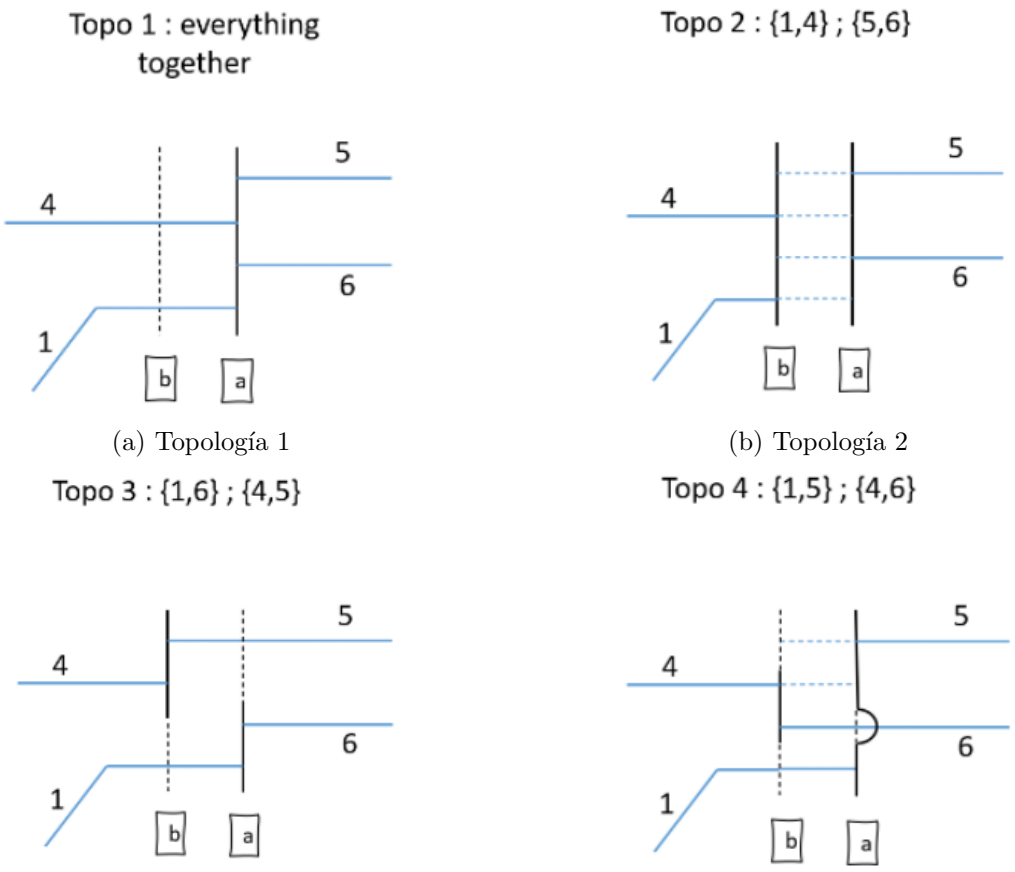


Figura 3.3: Switches de subestación

Viendo la imagen anterior, se puede ver que cada conexión a la subestación se puede hacer a través del bus 1, el 2 o ninguno si se quiere desconectar una línea.



(a) Topología 1

(b) Topología 2

(c) Topología 3

(d) Topología 4

Figura 3.4: Diferentes topologías de conexión

En la imagen anterior se pueden ver 4 topologías distintas para ejemplificar el uso de los buses. En este caso, los buses son el **a** y el **b**. En la primera topología se puede ver que las cuatro redes están conectadas al bus **a** y por lo tanto están conectadas todas entre sí. En la 2, se ha asignado al bus **a** las redes 5 y 6, mientras que al **b** se le han asignado las redes 1 y 4. En esta topología se puede ver que las redes están conectadas 2 a 2. Finalmente, en las topologías 3 y 4 se puede ver que pasa algo similar que en la 2, demostrando así las posibilidades que existen de conectar las redes a las subestaciones.

Conociendo lo anterior, se pueden explicar las acciones que se pueden hacer sobre estas conexiones.

- Indicar un bus. Con esta acción le indicas en que bus quieres que esté.
- Cambio de bus. Le indicas que quieres que cambie al bus contrario, independientemente del bus en el que se encuentre.
- Desconectar. Esto sólo se aplica a las líneas que conectan subestaciones.
- Conectar. No importa el bus, sólo se aplica a las líneas que conectan subestaciones.

3.3. OpenAI Gym

Gym es un conjunto de herramientas para desarrollar y comparar algoritmos de aprendizaje por refuerzo.

Gym es una biblioteca de Python de código abierto para desarrollar y comparar algoritmos de aprendizaje por refuerzo al proporcionar una API estándar para comunicarse entre los algoritmos de aprendizaje y los entornos, así como un conjunto estándar de entornos que cumplen con esa API. Desde su lanzamiento, la API de Gym se ha convertido en el estándar de campo para hacer esto.

No hace suposiciones sobre la estructura de su agente y es compatible con cualquier biblioteca de cálculo numérico, como TensorFlow o Theano. Esta biblioteca es una colección de problemas de prueba (entornos) que puede utilizar para desarrollar sus algoritmos de aprendizaje por refuerzo. Estos entornos tienen una interfaz compartida, lo que le permite escribir algoritmos generales.

3.4. OpenAI Baselines

OpenAI Baselines [17] es un conjunto de implementaciones complejas de algoritmos de aprendizaje por refuerzo.

Estos algoritmos hacen más fácil para la comunidad de investigación replicar, refinar e identificar nuevas ideas. También permite crear líneas de investigación sobre las que basar futuros trabajos. La implementación de DQN y sus variantes están documentadas en diversos artículos que publican.

Esta librería puede ser una base alrededor de la cual se puedan agregar nuevas ideas y usar como una herramienta para comparar un nuevo enfoque con otras implementaciones existentes.

Los principales motivos por los que se ha elegido a OpenAI baselines es su comunidad activa, la documentación y la incorporación de ejemplos sencillos, lo cuál facilita mucho el empezar a trabajar con sus algoritmos.

Sus implementaciones están desarrolladas en Python. Este es un lenguaje muy potente, con gran capacidad a la hora de hacer desarrollos en campos de IA. Otras características importantes son la legibilidad del código y lo fácilmente modificable que resulta.

La compatibilidad con entornos Gym y la gran variedades de adaptaciones de otros entornos hace que sea muy flexible lo cuál le permite trabajar con una gran cantidad de problemas, incluso de algunos que no han sido desarrollado por la propia OpenAI.

En cuanto al estado del arte, cabe destacar de entre todos sus desarrollos el del famoso DALLE 2. Una IA capaz de generar imágenes a partir de una descripción utilizando el lenguaje natural. [9]

3.5. Keras

Keras es una biblioteca de Redes Neuronales de Código Abierto escrita en Python. Es capaz de ejecutarse sobre TensorFlow, Microsoft Cognitive Toolkit o Theano.Está especialmente diseñada para posibilitar la experimentación en más o menos poco tiempo con redes de Aprendizaje Profundo.

Keras es el marco de aprendizaje profundo más utilizado entre los 5 equipos ganadores de Kaggle. Esto es debido a que Keras facilita la ejecución de nuevos experimentos, permite probar más ideas que la competencia y de forma más rápida.

Está construido sobre TensorFlow 2. Keras es un marco de trabajo sólido en la industria, que puede escalar a grandes grupos de GPU de manera sencilla.

3.6. Google Colab

Google Colab es un producto de Google Research. Permite a cualquier usuario escribir y ejecutar código de Python en el navegador. Es especialmente adecuado para tareas de aprendizaje automático, análisis de datos y educación. Desde un punto de vista más técnico, Colab es un servicio alojado de Jupyter Notebook que no requiere configuración y que ofrece acceso gratuito a recursos informáticos, como GPUs.

Todos los cuadernos de Colab se almacenan en Google Drive o puedes cargarlos desde GitHub. Los cuadernos de Colab se pueden compartir igual que cualquier archivo de Google.

Para poder ofrecer recursos informáticos de forma gratuita, Colab necesita tener flexibilidad para ajustar los límites de uso y la disponibilidad de hardware sobre la marcha. Los recursos disponibles en Colab pueden variar con el tiempo para adaptarse a las fluctuaciones de la demanda, así como para atender el crecimiento general y otros factores.

Los tipos de GPU disponibles varían a lo largo del tiempo. Esto es necesario para que Colab pueda ofrecer recursos de forma gratuita. Las GPU disponibles en Colab suelen ser los modelos Nvidia K80, T4, P4 y P100. En Colab no puedes seleccionar el tipo de GPU al que te conectas.

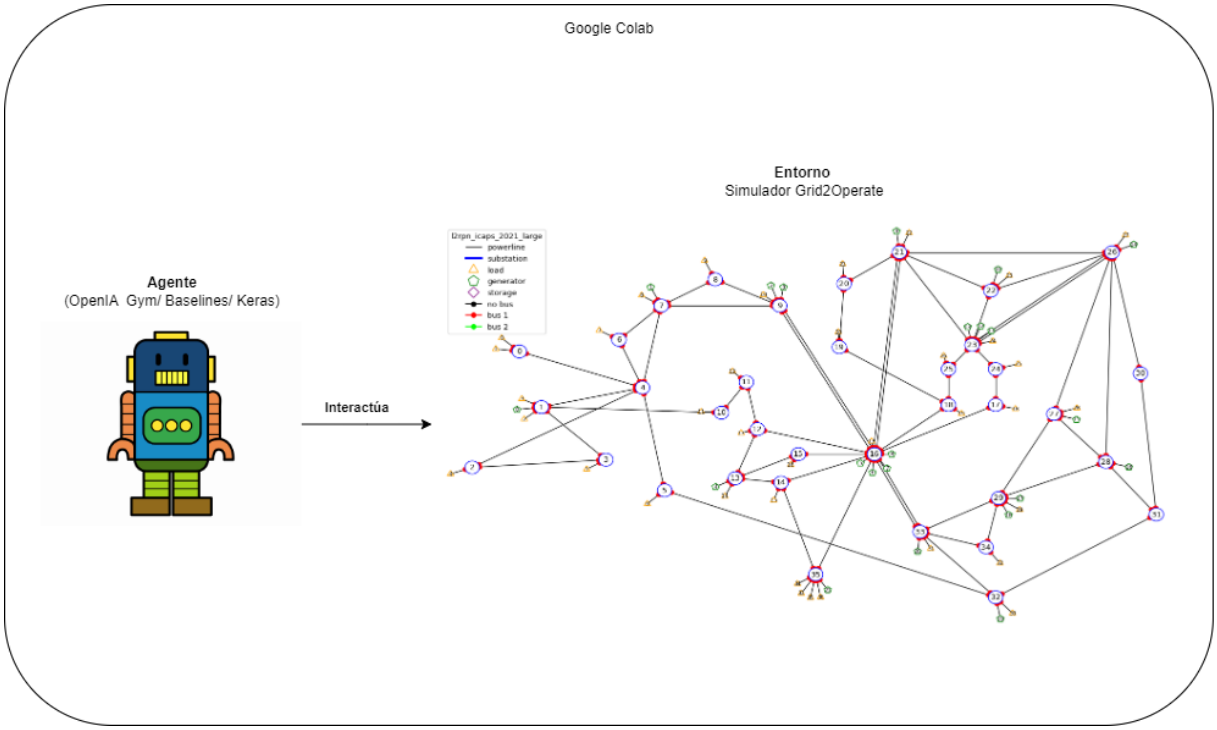
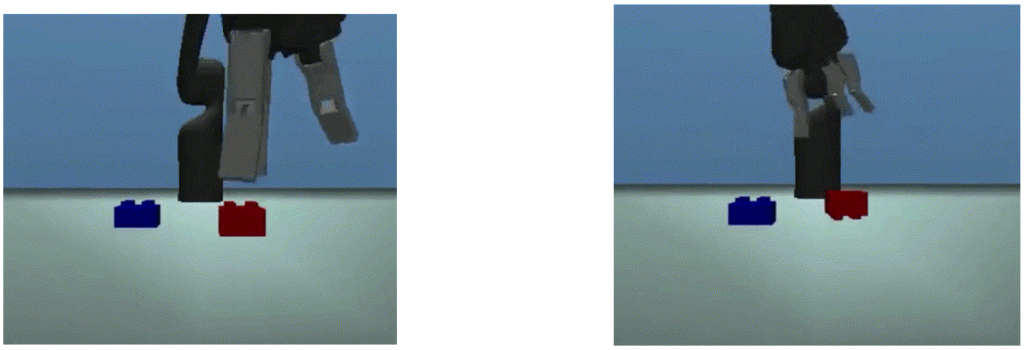


Figura 3.5: Diagrama de las herramientas utilizadas

3.7. Specification gaming

Specification gaming hace referencia a un suceso o comportamiento que satisface el objetivo sin lograr el resultado esperado [23]. Este es un fenómeno conocido dentro de los algoritmos de aprendizaje por refuerzo. Como se ha descrito, esto sucede cuando un agente cumple con los requisitos pero no de la manera deseada.

En el anterior artículo de deepmind se muestra una situación que ejemplifica perfectamente este fenómeno. Se busca que un agente monte una pieza de lego sobre otra. La función de recompensa se establece dándo una mayor recompensa en función de la altura a la que se encuentre la cara inferior de una de las piezas. ¿Que hace el agente? Simplemente le da la vuelta a la pieza roja como se ve en las imágenes inferiores:



(a) Situación inicial (b) Situación final

Figura 3.6: Comportamiento del agente

Como se puede intuir por el ejemplo anterior, este comportamiento está provocado por una definición incorrecta o poco precisa de la función de recompensa. Este es un problema complejo que, en muchas ocasiones, no se puede prever. Dentro de este trabajo nos encontramos con este problema. En el apartado de experimentación se hablará más a fondo de los motivos que lo habían provocado

Capítulo 4

Metodología

4.1. Planteamiento de la experimentación

Para la parte de experimentación se empezó trabajando con los cuadernos del apartado "Getting started"del repositorio de Grid2Operate [21]. Este primer paso sirvió para entender mejor el funcionamiento del simulador y para revisar la teoría anteriormente descrita en el apartado 3.2.

A continuación, se usó implementaciones de agentes de DRL de la librería Stable Baselines. La documentación del simulador indicaba que el entorno era compatible con los agentes de esta biblioteca. Pronto se vio que no era así y que la tarea de poder hacerlos compatibles era compleja. Tras un largo trabajo de adaptación de los entornos se pudo experimentar con algunos de los agentes de esta librería.

Los últimos experimentos fueron con una implementación propia de DQN. También se implementó un pequeño oponente que saboteaba la red para crear la necesidad de hacer cambios sobre esta.

Tras estos experimentos se vio que la recompensa del simulador por defecto no era muy adecuada. En el caso de que no se hiciesen cambios en la red, la recompensa era muy elevada. Si se realizaba un cambio y mejoraba el estado, la recompensa aumentaba muy poco y si empeoraba con el cambio, la recompensa se reducía drásticamente. Esto provocaba que el agente aprendiese a mantener la red sin cambios ante el peligro de tomar una decisión errónea. Teniendo lo anterior en cuenta, el final de la experimentación se centró en modificar la implementación de DQN. Se tuvieron que hacer algunas adaptaciones sobre el código inicial para poder llegar a una implementación que se adecuase al problema y al simulador.

Todos los agentes se han entrenado todos de la misma manera con 2000 iteraciones. Hay que tener en cuenta que para cada entorno hay que entrenar un agente distinto y no se pueden reutilizar porque las acciones que se pueden realizar sobre la red son distintas y la redes se comportan de manera diferente. La validación se ha realizado lanzando ejecuciones con los agentes entrenados de 1000 iteraciones.

4.2. Experimentación inicial

El primer acercamiento al problema fue trabajar con los cuadernos disponibles en el repositorio del desafío para familiarizarme con el funcionamiento del simulador [20]. En estos cuadernos se hace un repaso de las funcionalidades del simulador y se explica cierta teoría acerca de las redes eléctricas con el fin de que los usuarios tengan una visión más profunda del tema. En estos cuadernos se puede ver que dejan a disposición de los usuarios diferentes entornos que van desde los más simples, que sirven para empezar a familiarizarse con el entorno, hasta los más complejos, que son los usados en la competición. Además de lo anterior, implementan diferentes agentes con los que poder experimentar y que pueden servir de punto de partida partida para futuras implementaciones.

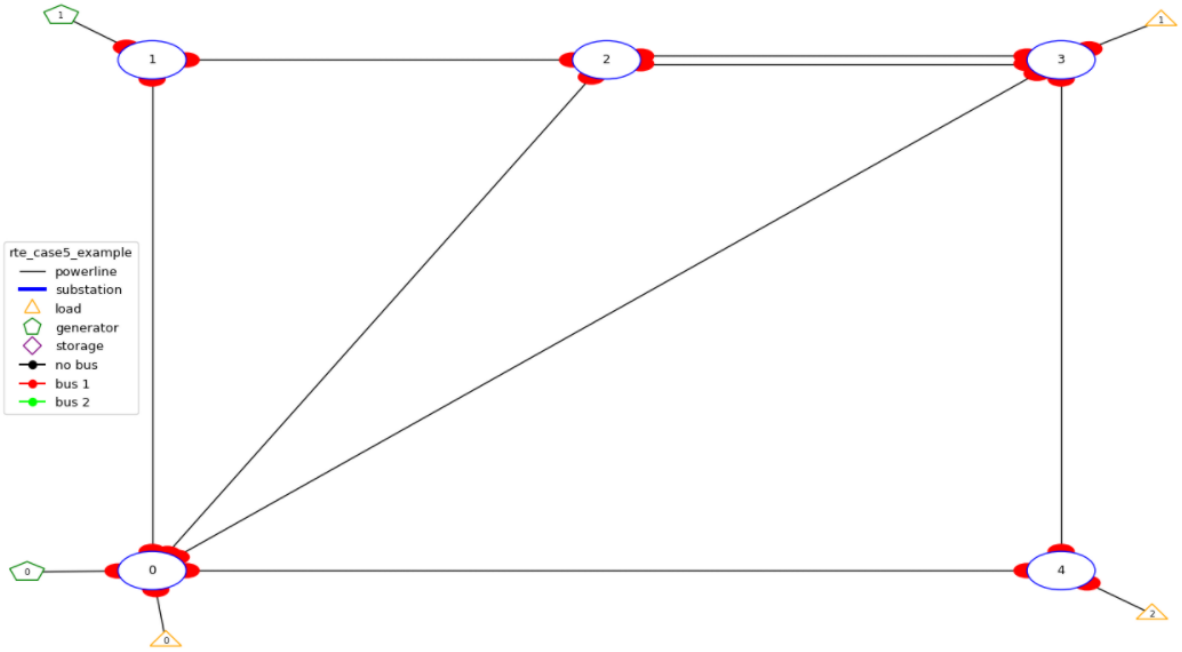


Figura 4.1: Entorno de prueba: *rte case5 example*

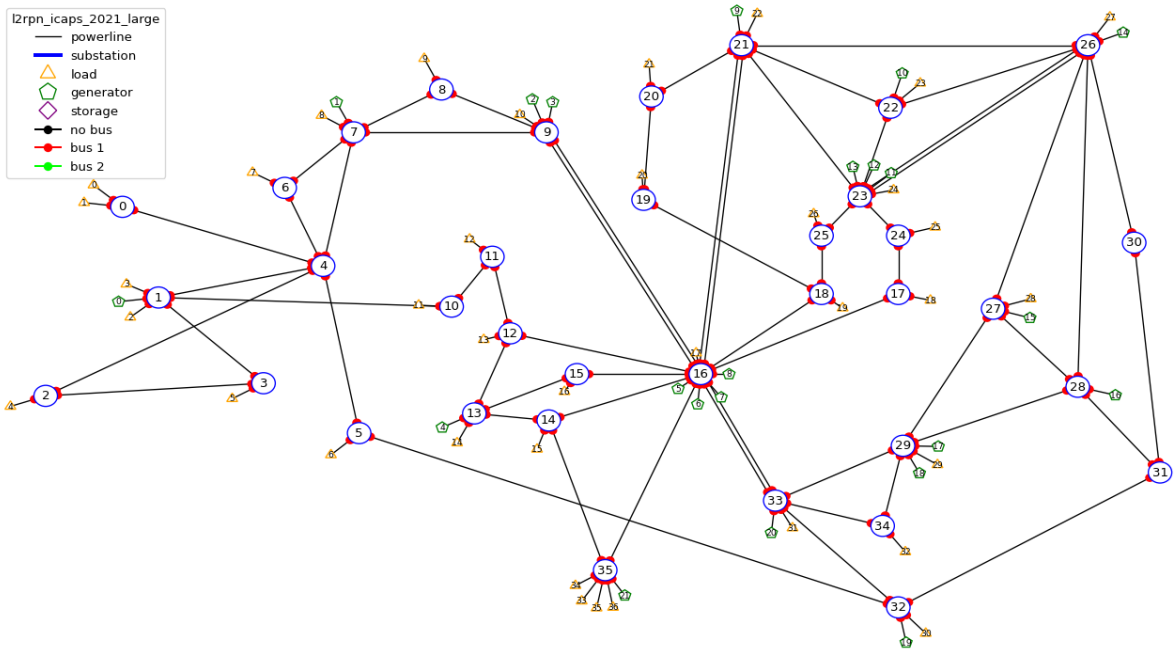


Figura 4.2: Entorno de competición: *l2rpn icaps 2021 large*

Como se puede ver en las imágenes anteriores, los entornos pueden llegar a ser bastante complejos. Esto provoca que la cantidad de acciones posibles sobre la red aumenta haciendo el problema más complejo.

4.3. Experimentación con Stable Baselines

Después del acercamiento anterior, los experimentos fueron con los agentes de Stable Baselines[14]. A priori, la documentación oficial del desafío y del simulador indicaba que se podía usar agentes de esta librería de manera directa. Esto no era cierto, hubo que realizar varios ajustes para poder utilizar estos agentes.

Las primeras opciones que probamos fue construir un wrapper para el entorno. De esta manera cambiaríamos los elementos que daban problemas. Con este wrapper implementado, empezó a dar problemas la clase Space de gym. Para solucionarlo probamos a construir un wrapper para la clase Space y con ello cambiar los elementos problemáticos, de manera similar a la clase Env del entorno. Con los dos wrappers implementados vimos que los problemas continuaban y decidimos optar por otro tipo de soluciones.

El primer paso que tuvimos que hacer fue compatibilizar el entorno con el estándar de Gym. Esta parte la tienen ya desarrollada por lo que pudimos utilizar esa implementación[2]. Tanto el espacio de observaciones como el de acciones es mixto. Esto quiere decir que tiene tanto variables continuas como binarias o discretas. Esto en Gym se representa con el tipo de spaces denominado Dict.

Básicamente se trata de un diccionario donde viene el nombre de los datos y tipos tales como: spaces.Box, spaces.Discrete, spaces.MultiDiscrete, etc. Esto entra dentro del estándar de gym, el problema es que no tiene soporte para Stable-Baselines 3. Recientemente incorporaron soporte de compatibilidad para el espacio de observaciones de este tipo. Por eso hubo que hacer la instalación usando:

```
!pip install git+https://github.com/DLR-RM/stable-baselines3
```

pese a estar en la rama master del repositorio. Haciendo:

```
!pip install stable-baselines3
```

no se incorpora este soporte. Para que el espacio de observaciones funcionase correctamente también hubo que cambiar el tipo de red a MultiInputPolic. Con esto el espacio de observaciones está cubierto, pero sigue sin haber soporte para espacios de acciones basados en diccionarios mixtos.

Después de solucionar los problemas, los experimentos realizadas fueron con agentes que implementaban los algoritmos de DRL de PPO y A2C. Estos agentes eran los únicos compatibles con el tipo de entorno del desafío.

4.4. Experimentación con implementación propia de DQN

4.4.1. Adaptación de DQN

Para el desarrollo de los experimentos contamos con una implementación inicial de uno de una asignatura de uno de los tutores de este trabajo, Juan Gómez Romero [19]. Partiendo de esta implementación inicial, hubo que adaptar el entorno para que el agente pudiera actuar sobre él.

Hubo que convertir el espacio de acciones, el espacio de observaciones. Dentro de la librería de Grid2Op encontramos una serie de funciones que nos ayudaron con esta tarea [15]. La transformación que hubo que realizar es pasar de unos diccionarios a simplemente un entero. Cada una de las posibilidades de formación de esos diccionarios se traduce en un entero. A continuación se muestra uno de los espacios de observaciones originales (los de acciones son similares):

```
1  _shunt_bus:Box([-2147483648], [2147483647], (1,), int32), _shunt_p:Box([-inf], [inf],
       (1,), float32), _shunt_q:Box([-inf], [inf], (1,), float32), _shunt_v:Box([-inf], [inf
       ], (1,), float32), a_ex:Box([0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
        0.], [inf inf inf inf inf inf inf inf inf inf inf inf inf inf inf inf inf inf inf inf
2   inf inf], (20,), float32), a_or:Box([0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
       0. 0. 0.], [inf inf inf inf inf inf inf inf inf inf inf inf inf inf inf inf inf inf inf
3   inf inf], (20,), float32), actual_dispatch:Box([-150. -200.  -70.  -50. -300.], [150.
       200.  70.  50. 300.], (5,), float32), attention_budget:Box([0.], [inf], (1,),
       float32), current_step:Box([-2147483648], [2147483647], (1,), int32), curtailment:
       Box([0. 0. 0. 0. 0.], [1. 1. 1. 1. 1.], (5,), float32), curtailment_limit:Box([0. 0.
        0. 0. 0.], [1. 1. 1. 1. 1.], (5,), float32), curtailment_limit_effective:Box([0. 0.
        0. 0. 0.], [1. 1. 1. 1. 1.], (5,), float32), day:Discrete(32), day_of_week:Discrete
       (8), delta_time:Box([0.], [inf], (1,), float32), duration_next_maintenance:Box([-1
       -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1], [2147483647 2147483647
       2147483647 2147483647 2147483647 2147483647 2147483647
4   2147483647 2147483647 2147483647 2147483647 2147483647 2147483647
5   2147483647 2147483647 2147483647 2147483647 2147483647 2147483647
6   2147483647 2147483647], (20,), int32), gen_margin_down:Box([0. 0. 0. 0. 0.], [ 5. 10.
       0.  0. 10.], (5,), float32), gen_margin_up:Box([0. 0. 0. 0. 0.], [ 5. 10.  0.  0.
       10.], (5,), float32), gen_p:Box([-231.01 -231.01 -231.01 -231.01 -231.01], [381.01
       431.01 301.01 281.01 531.01], (5,), float32), gen_p_before_curtail:Box([-231.01
       -231.01 -231.01 -231.01 -231.01], [381.01 431.01 301.01 281.01 531.01], (5,),
       float32), gen_q:Box([-inf -inf -inf -inf -inf], [inf inf inf inf inf], (5,), float32
       ), gen_theta:Box([-180. -180. -180. -180. -180.], [180. 180. 180. 180. 180.], (5,),
       float32), gen_v:Box([0. 0. 0. 0. 0.], [inf inf inf inf inf], (5,), float32),
       hour_of_day:Discrete(24), is_alarm_illegal:Discrete(2), line_status:MultiBinary(20),
        load_p:Box([-inf -inf -inf -inf -inf -inf -inf -inf -inf -inf -inf], [inf inf inf
       inf inf inf inf inf inf inf inf], (11,), float32), load_q:Box([-inf -inf -inf -inf -
       inf -inf -inf -inf -inf -inf -inf], [inf inf inf inf inf inf inf inf inf inf inf],
       (11,), float32), load_theta:Box([-180. -180. -180. -180. -180. -180. -180. -180.
       -180. -180. -180.], [180. 180. 180. 180. 180. 180. 180. 180. 180. 180. 180.], (11,),
        float32), load_v:Box([0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.], [inf inf inf inf inf inf
       inf inf inf inf inf], (11,), float32), max_step:Box([-2147483648], [2147483647],
       (1,), int32), minute_of_hour:Discrete(60), month:Discrete(13), p_ex:Box([-inf -inf -
       inf -inf -inf -inf -inf -inf -inf -inf -inf -inf -inf -inf -inf -inf -inf -inf -inf
7   -inf -inf -inf -inf -inf -inf], [inf inf inf inf inf inf inf inf inf inf inf inf inf inf
        inf inf inf inf inf
8   inf inf], (20,), float32), p_or:Box([-inf -inf -inf -inf -inf -inf -inf -inf -inf -inf -
       inf -inf -inf -inf -inf
9   -inf -inf -inf -inf -inf -inf], [inf inf inf inf inf inf inf inf inf inf inf inf inf inf
        inf inf inf inf inf
10  inf inf], (20,), float32), q_ex:Box([-inf -inf -inf -inf -inf -inf -inf -inf -inf -inf -
       inf -inf -inf -inf -inf
11  -inf -inf -inf -inf -inf -inf], [inf inf inf inf inf inf inf inf inf inf inf inf inf inf
        inf inf inf inf inf
12  inf inf], (20,), float32), q_or:Box([-inf -inf -inf -inf -inf -inf -inf -inf -inf -inf -
       inf -inf -inf -inf -inf
13  -inf -inf -inf -inf -inf -inf], [inf inf inf inf inf inf inf inf inf inf inf inf inf inf
        inf inf inf inf inf
14  inf inf], (20,), float32), rho:Box([0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
```

```
    0. 0. 0.], [inf inf inf inf inf inf inf inf inf inf inf inf inf inf inf inf inf inf
15 inf inf], (20,), float32), target_dispatch:Box([-150. -200.  -70.  -50. -300.], [150.
    200.  70.  50. 300.], (5,), float32), thermal_limit:Box([0. 0. 0. 0. 0. 0. 0. 0. 0.
    0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.], [inf inf inf inf inf inf inf inf inf inf inf inf
    inf inf inf inf inf inf
16 inf inf], (20,), float32), theta_ex:Box([-180. -180. -180. -180. -180. -180. -180. -180.
     -180. -180. -180. -180.
17 -180. -180. -180. -180. -180. -180. -180. -180.], [180. 180. 180. 180. 180. 180. 180.
    180. 180. 180. 180. 180. 180. 180.
18 180. 180. 180. 180. 180. 180.], (20,), float32), theta_or:Box([-180. -180. -180. -180.
    -180. -180. -180. -180. -180. -180. -180. -180.
19 -180. -180. -180. -180. -180. -180. -180. -180.], [180. 180. 180. 180. 180. 180. 180.
    180. 180. 180. 180. 180. 180. 180.
20 180. 180. 180. 180. 180. 180.], (20,), float32), time_before_cooldown_line:Box([0 0 0 0
    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0], [10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
    10 10 10 10], (20,), int32), time_before_cooldown_sub:Box([0 0 0 0 0 0 0 0 0 0 0 0 0
     0], [0 0 0 0 0 0 0 0 0 0 0 0 0 0], (14,), int32), time_next_maintenance:Box([-1 -1
    -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1], [2147483647 2147483647
    2147483647 2147483647 2147483647 2147483647 2147483647 2147483647
21 2147483647 2147483647 2147483647 2147483647 2147483647 2147483647
22 2147483647 2147483647 2147483647 2147483647 2147483647 2147483647
23 2147483647 2147483647], (20,), int32), time_since_last_alarm:Box([-1], [2147483647],
    (1,), int32), timestep_overflow:Box([-2147483648 -2147483648 -2147483648 -2147483648
     -2147483648 -2147483648
24 -2147483648 -2147483648 -2147483648 -2147483648 -2147483648 -2147483648
25 -2147483648 -2147483648 -2147483648 -2147483648 -2147483648 -2147483648
26 -2147483648 -2147483648], [2147483647 2147483647 2147483647 2147483647 2147483647
    2147483647
27 2147483647 2147483647 2147483647 2147483647 2147483647 2147483647
28 2147483647 2147483647 2147483647 2147483647 2147483647 2147483647
29 2147483647 2147483647], (20,), int32), topo_vect:Box([-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
    -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
30 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
31 -1 -1 -1 -1 -1 -1 -1 -1], [2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
     2 2 2 2 2 2
32 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2], (56,), int32), v_ex:Box([0. 0. 0. 0. 0. 0. 0. 0.
     0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.], [inf inf inf inf inf inf inf inf inf inf inf
    inf inf inf inf inf inf inf
33 inf inf], (20,), float32), v_or:Box([0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
    0. 0. 0.], [inf inf inf inf inf inf inf inf inf inf inf inf inf inf inf inf inf inf
34 inf inf], (20,), float32), was_alarm_used_after_game_over:Discrete(2), year:Discrete
    (2100)
```

4.4.2. Experimentos DQN

Los primeros experimentos se centraron en entrenar el agente en los mismos entornos que con Stable Baselines. Durante estos experimentos pudimos identificar algunos problemas de memoria que hicieron imposible ejecutar los experimentos con los entornos más grandes. En el apartado de resultados se entrará más en detalle en este problema.

El desarrollo que se hizo a continuación fue el de modificar la función de recompensa para intentar mejorar los resultados y por otro lado, se ajustó el entrenamiento para no aprender de según que actuaciones. Estas modificaciones se realizaron por un problema de specification gaming que estaba provocando que la acción que escogiesen los agentes fuese la de no modificar la red. Se entrará más en detalle en el apartado de resultados de Stable Baselines.

Capítulo 5

Resultados

5.1. Experimentación

5.1.1. Resultados con Stable Baselines

Estos han sido los resultados de los primeros experimentos. En ellos hemos utilizado los agentes PPO y A2C. Al ser un entorno continuo en el que no se llega a un estado final con el que se pueda dar por completado el episodio, la manera de observar los rendimientos es observar las recompensas en cada instante y ver como evoluciona. De esta manera se obtiene una traza de la eficiencia de las acciones de los agentes.

Los entrenamientos se llevaron a cabo en 4 entornos diferentes: rte case14 redisp, rte case14 realistic, wcci test, l2rpn wcci 2022. A continuación se puede ver una imagen de cada uno de estos entornos.

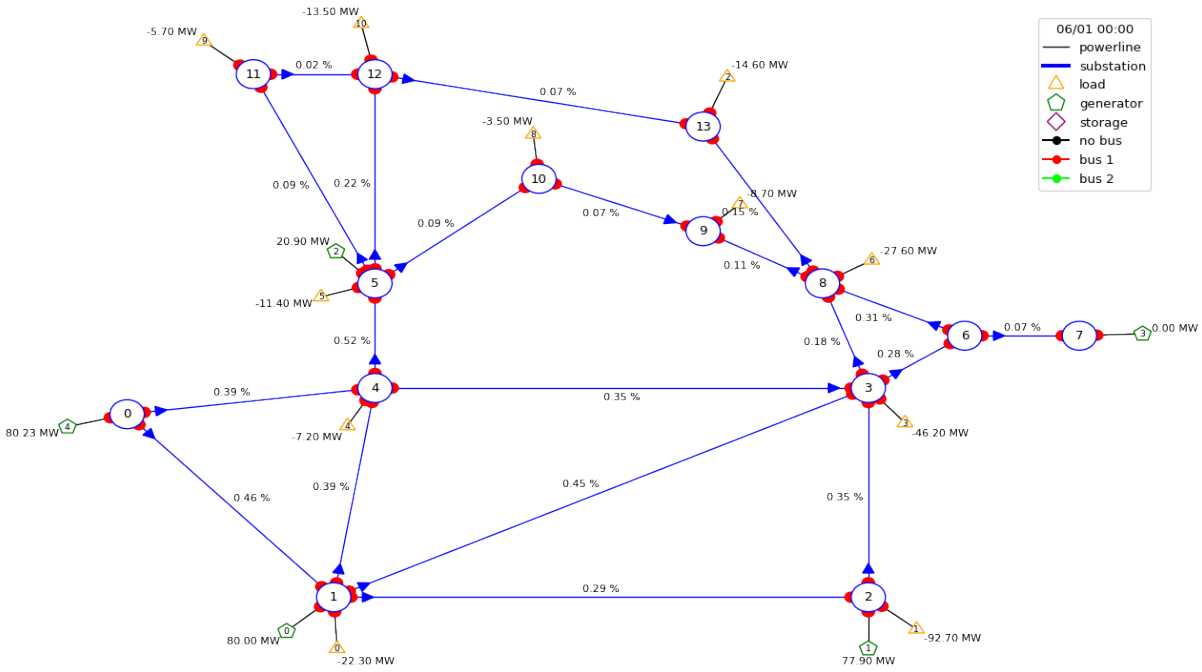


Figura 5.1: rte case14 redisp.

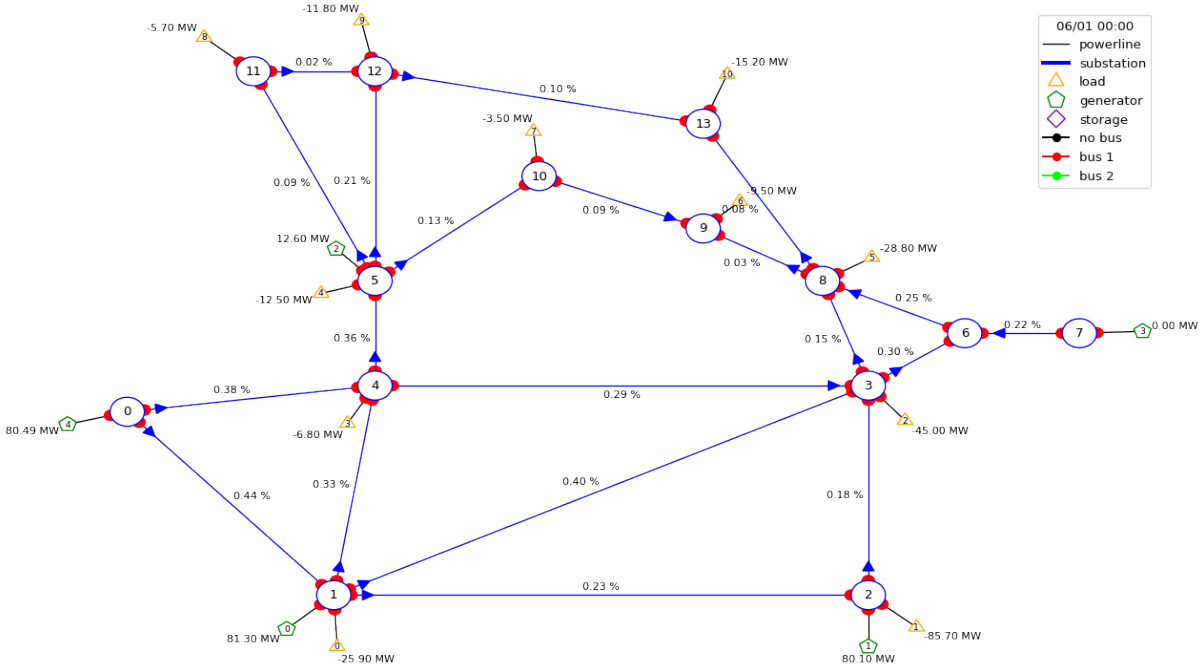


Figura 5.2: rte case14 realistic.

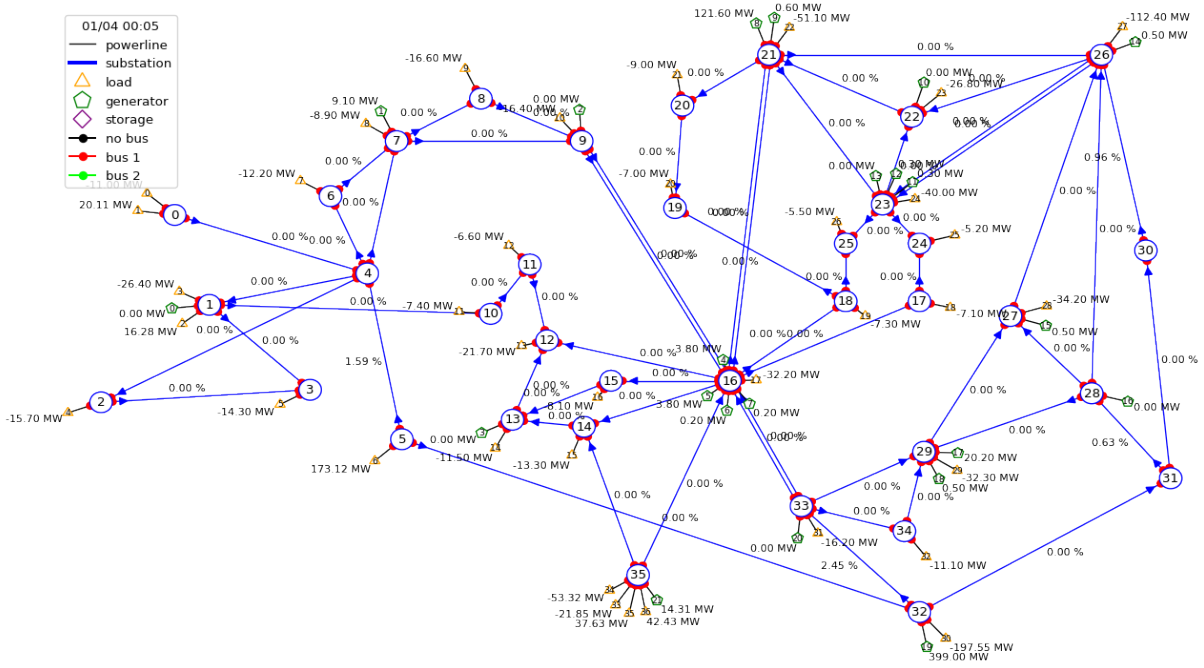


Figura 5.3: wcci test.

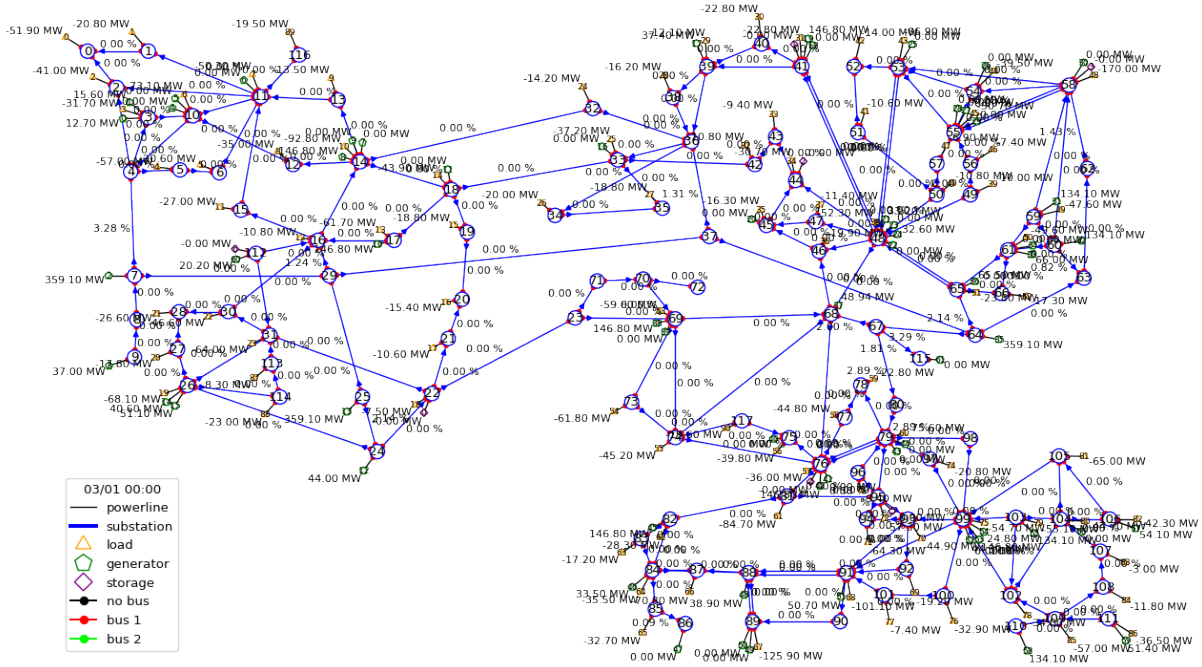


Figura 5.4: wcci 2022.

En estas imágenes, las subestaciones se representan con los vértices del grafo numerados, las aristas son las líneas eléctricas. De cada subestación pueden salir: generadores, almacenes y consumidores de electricidad. Tanto los 3 elementos anteriores como las líneas eléctricas están asignadas a unos de los dos buses que tiene cada subestación. La conexión de los buses está explicada más en detalle en el punto 3.2.

A continuación, se puede observar la traza de las recompensas en validación a lo largo de esas 1000 iteraciones. En el eje horizontal se muestra el índice de la iteración y en el eje vertical la recompensa asociada. Por cada gráfico se indica también el agente y el entorno sobre el que se ejecuta.

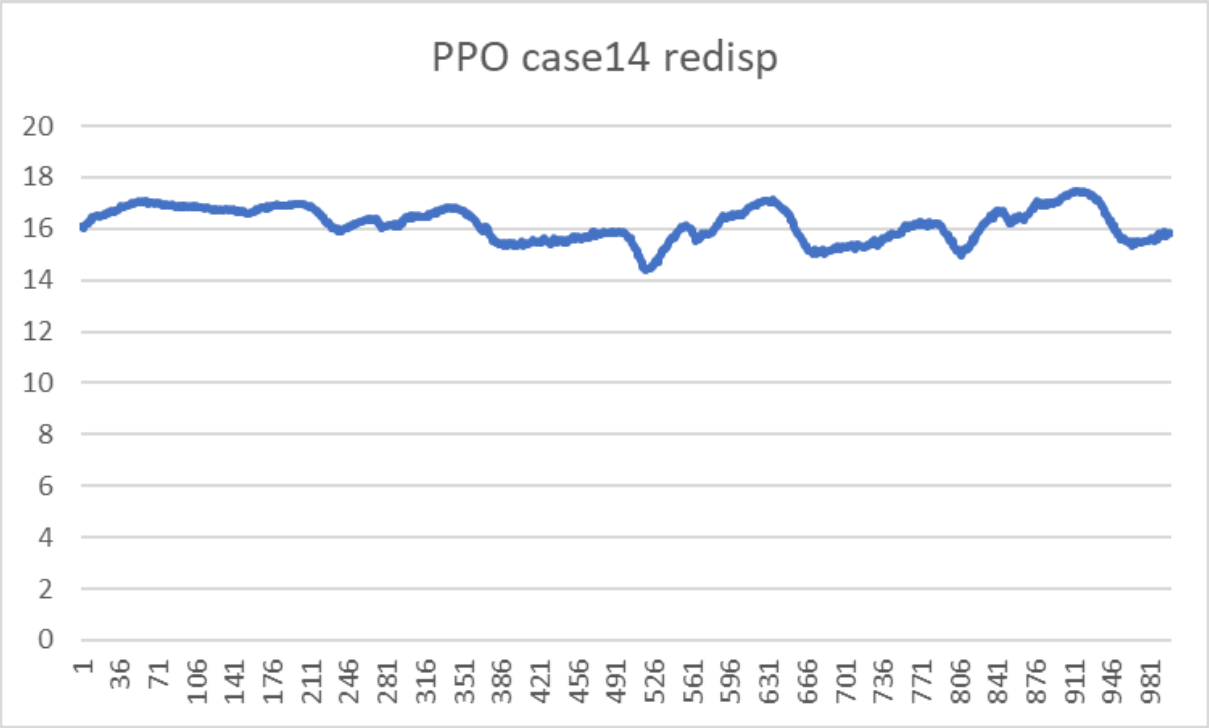


Figura 5.5: Validación de PPO en rte case14 redisp.

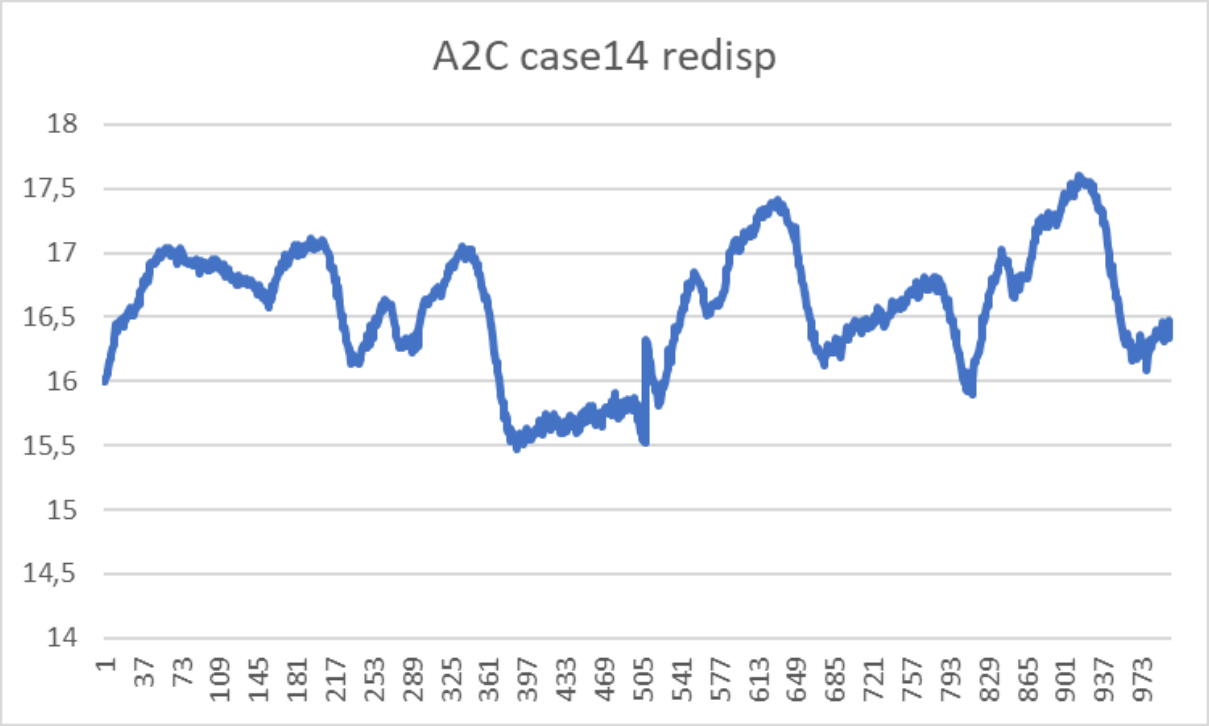


Figura 5.6: Validación de A2C rte case14 redisp.

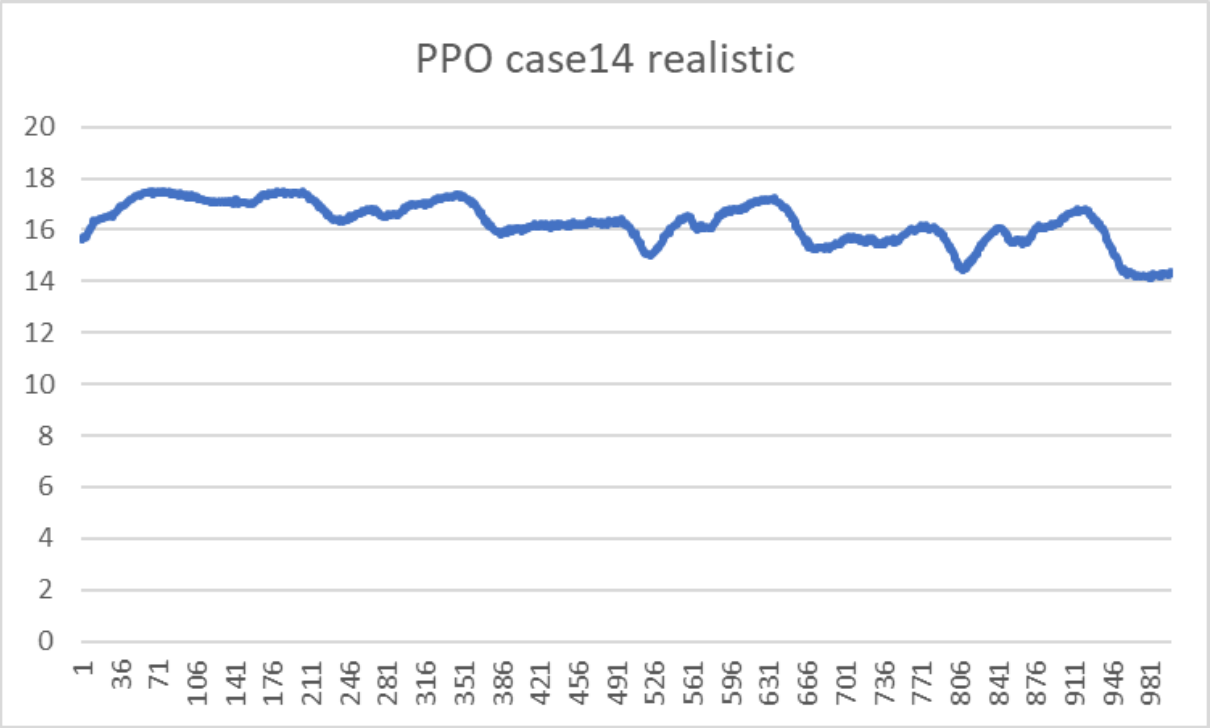


Figura 5.7: Validación de PPO en rte case14 realistic.

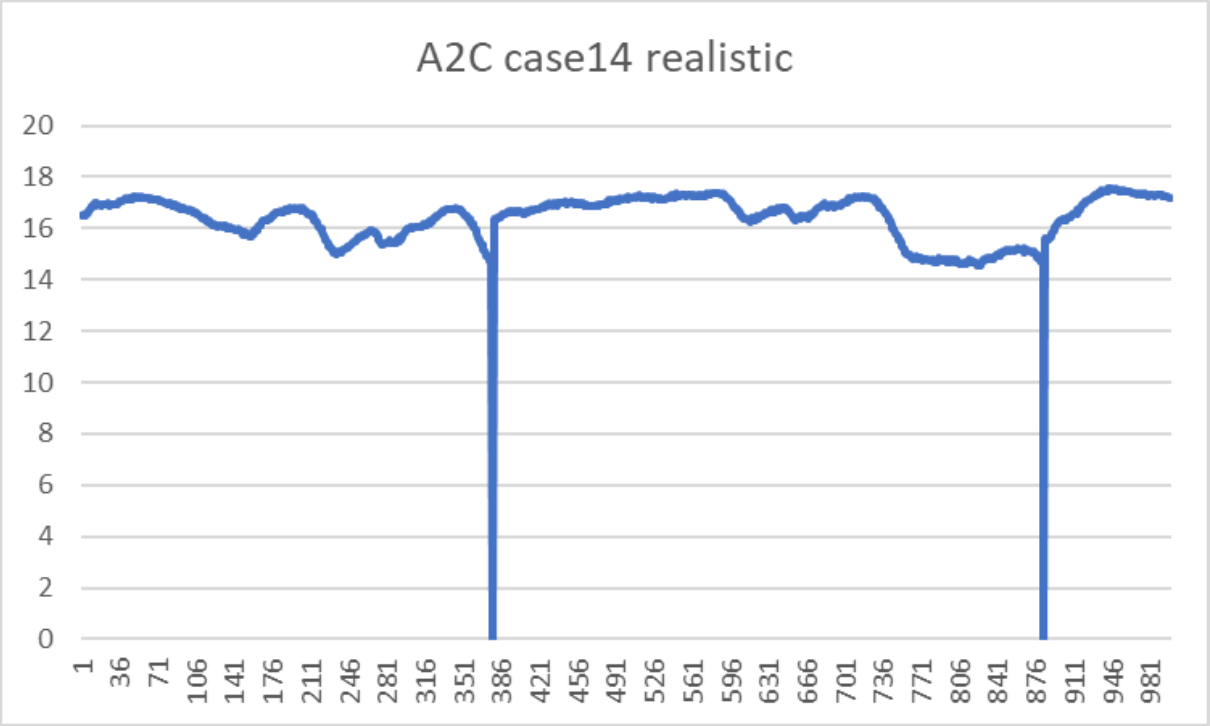


Figura 5.8: Validación de A2C en rte case14 realistic.

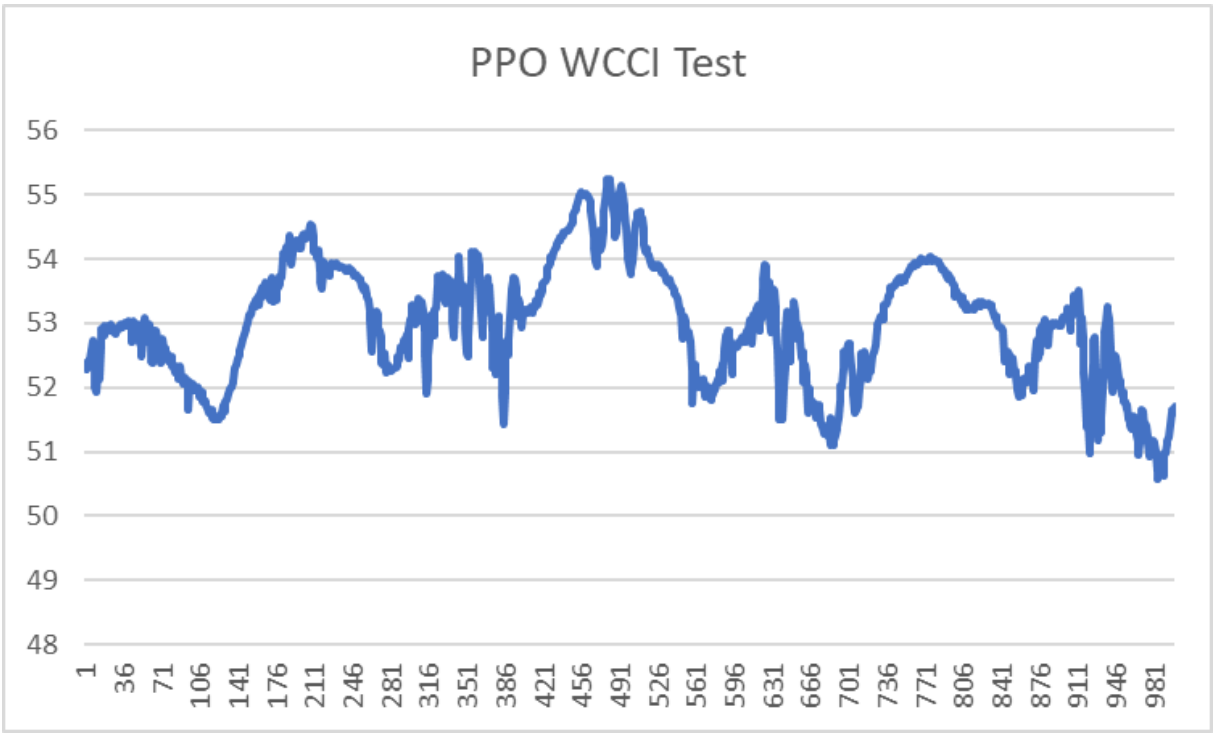


Figura 5.9: Validación de PPO enwcci 2022.

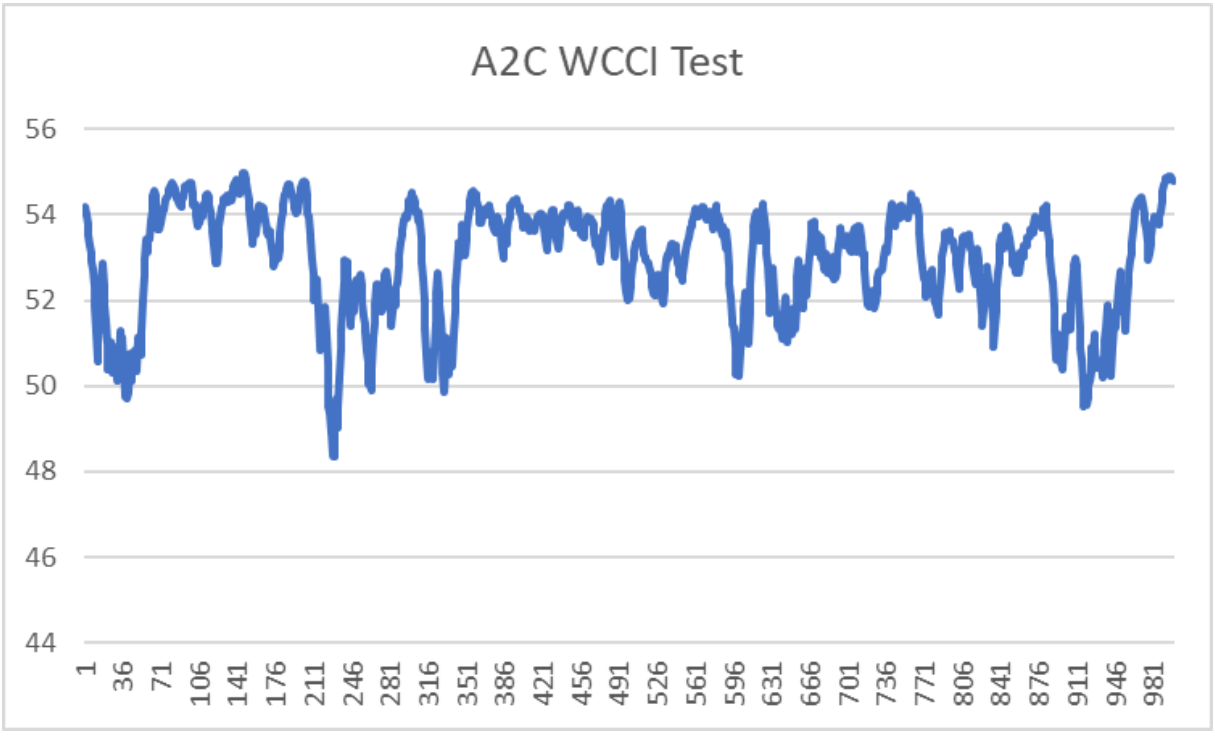


Figura 5.10: Validación de A2C en wcci 2022.

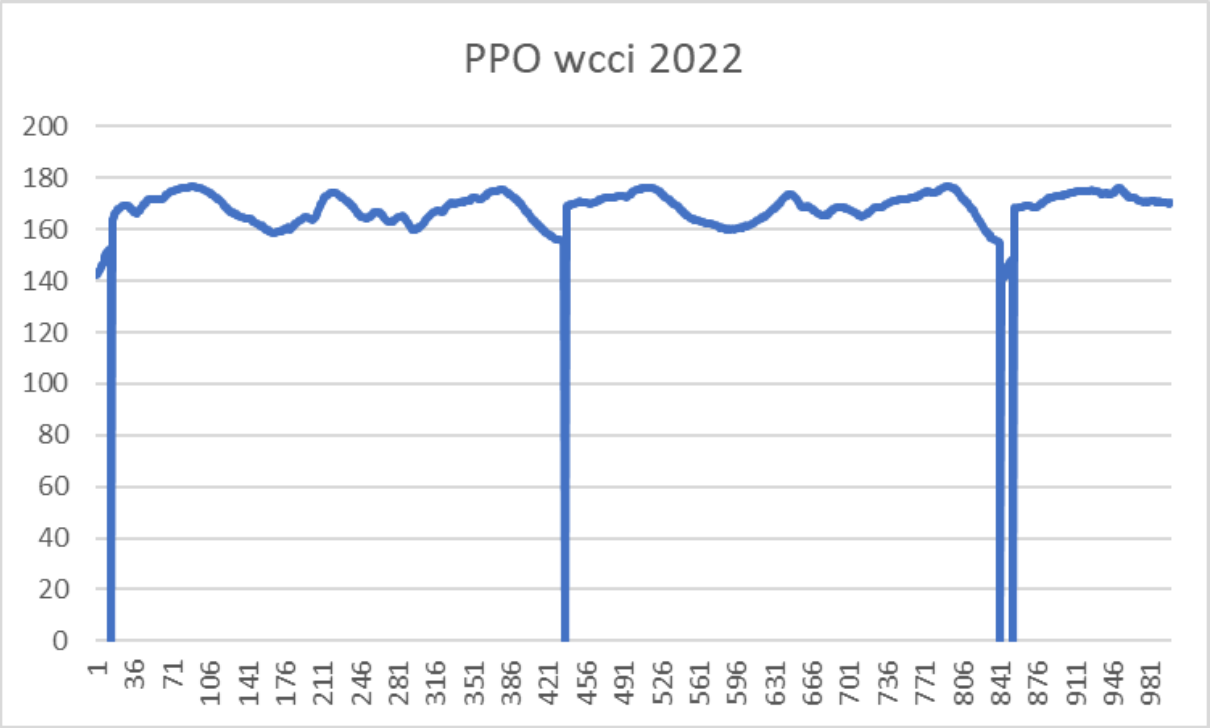


Figura 5.11: Validación de PPO en wcci 2022.

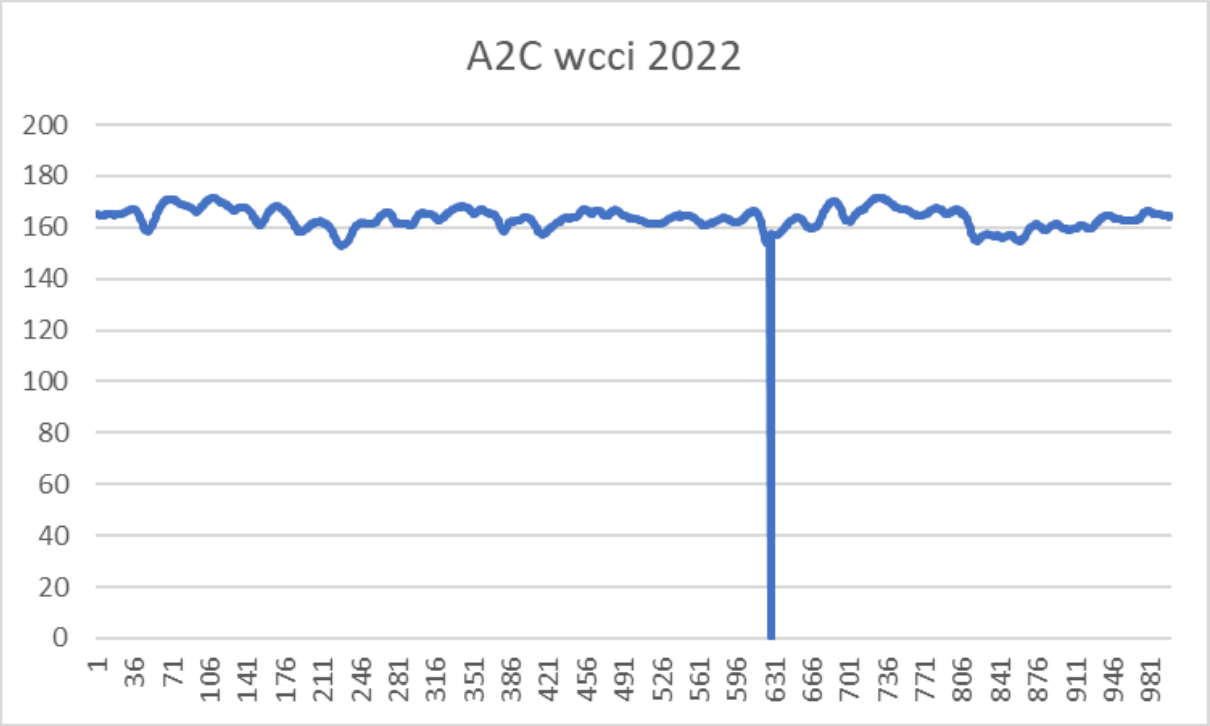


Figura 5.12: Validación de A2C en wcci 2022.

Al finalizar una primera iteración de experimentos obtuvimos los resultado anteriores. En algunos casos como en la ejecución de A2C en rte case 14 realistic se pueden ver dos caídas de la red. En los varios experimentos que se realizaron con ese agente y ese entorno estas caídas fueron recurrentes. Esto nos llamó la atención y pasamos a realizar experimentos centrándonos en observar que acciones se estaban llevando a cabo para identificar las que estaban provocando estas caídas.

Para nuestra sorpresa, en muchos casos los agentes optaban por mantener la red sin cambios a lo largo de toda la ejecución. En los únicos casos en los que el agente realizaba cambios y ajustaba la red fueron con el agente A2C y los entornos más sencillos. A continuación se puede ver una pequeña tabla resumen de los resultados que se obtuvieron:

Entorno	PPO	A2C
rte case14 redisp	Se repiten	Cambia
rte case14 realistic	Se repiten	Cambia
wcci test	Se repiten	Se repiten
wcci 2022	Se repiten	Se repiten

Cuadro 5.1: Resultados con los agentes PPO y A2C.

Buscando el motivo de este comportamiento se llegó a la conclusión de que la recompensa no estaba demasiado bien ajustada para el entrenamiento de estos agentes. La recompensa que se obtiene no haciendo nada es bastante elevada. En el caso de ajustar la red y mejorar el estado, la recompensa aumenta levemente. Por el contrario, si se ajusta de manera errónea, la recompensa cae drásticamente. Este es un error conocido dentro del mundo de los algoritmos de DRL y se llama specification gaming. Esto se provoca cuando el agente optimiza una función de recompensa mal definida o con efectos imprevistos.

5.1.2. Resultados con primera implementación de DQN

Los agentes de DQN se han entrenado de la misma manera que se ha comentado al comienzo de la metodología. Con 2000 iteraciones para el entrenamiento y 1000 para la validación. En un principio los experimentos se iban a realizar con los mismos entornos que con las Stables Baselines, pero fue imposible en los dos entornos más grandes. Tanto para el "wcci testçomo para el "wcci 2022"se hace una reserva de memoria demasiado elevada al comenzar el entrenamiento del DQN. El entorno de trabajo de Google Colab directamente paró la ejecución y en el entorno de ejecución local también se bloqueó por el mismo motivo.

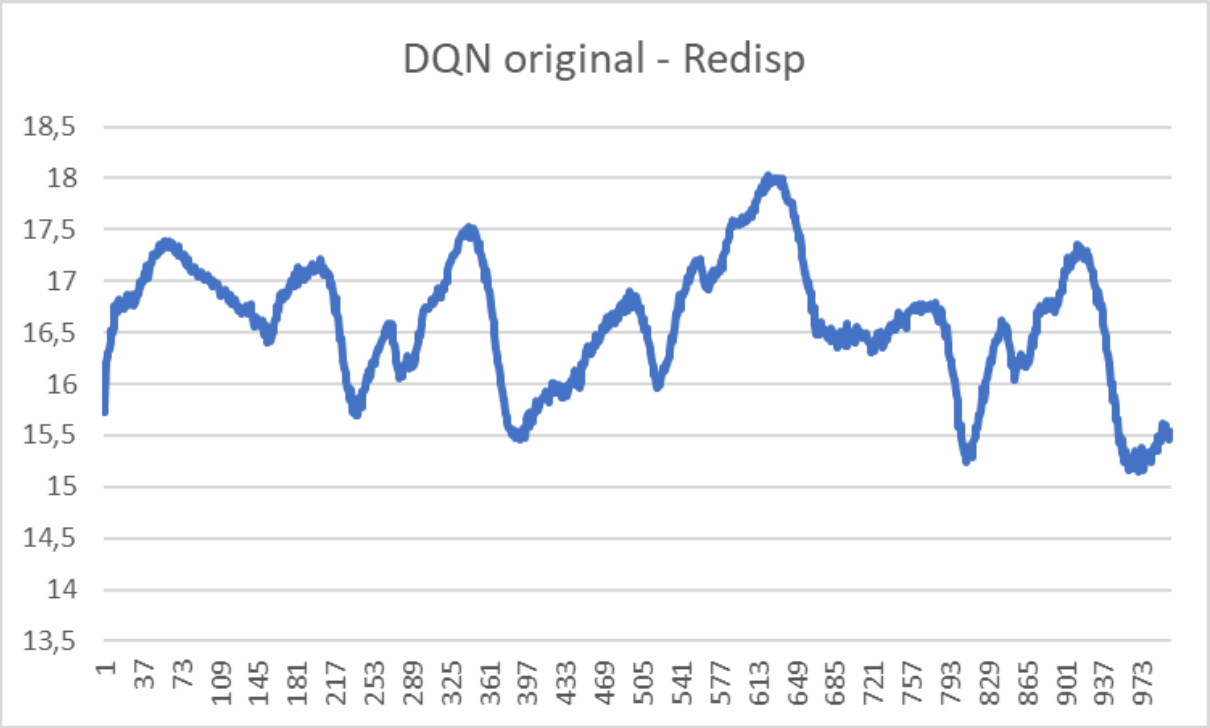


Figura 5.13: Validación de DQN original en rte case14 redisp.

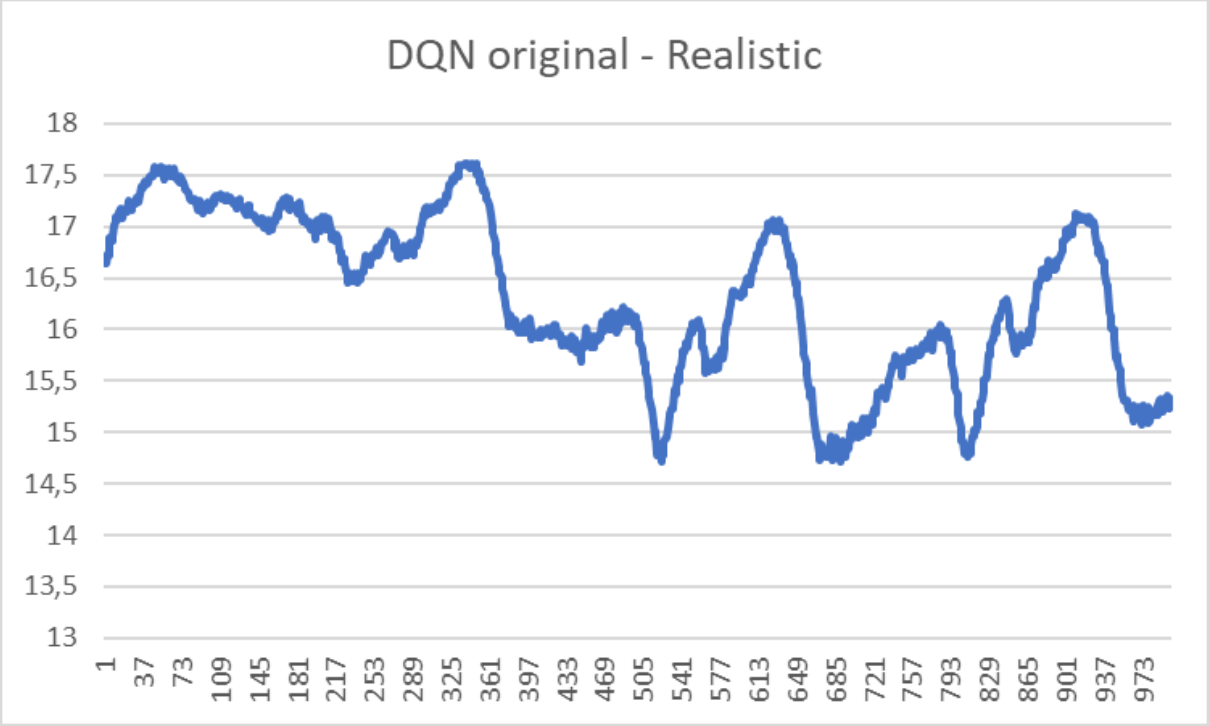


Figura 5.14: Validación de DQN original en rte case14 realistic.

Pudimos ver que el agente seguía teniendo el problema del sobreentrenamiento. Mantenía siempre el entorno sin cambios. A raíz de lo anterior el resto de modificaciones se basaron en intentar corregir este comportamiento.

5.1.3. Resultados con adaptación de DQN

Viendo los resultados anteriores y conociendo que la recompensa podía llevar a errores decidimos modificarla para ver el comportamiento que tenía sobre el entrenamiento del agente.

Para hacer esto se normalizó la recompensa. Después de realizar una acción, normalizamos la recompensa obtenida. Es esa nueva recompensa la que recibe el agente. A continuación se muestra la validación de los agentes con la recompensa normalizada. La recompensa que se muestra es la que se obtiene del entorno, no la normalizada.

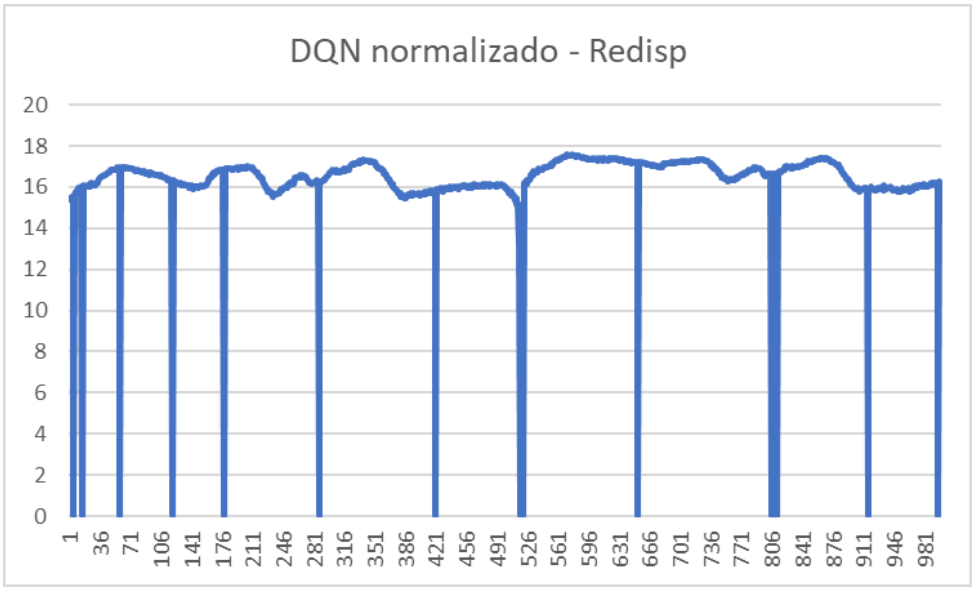


Figura 5.15: Validación de DQN con recompensa normalizada en rte case14 redisp.

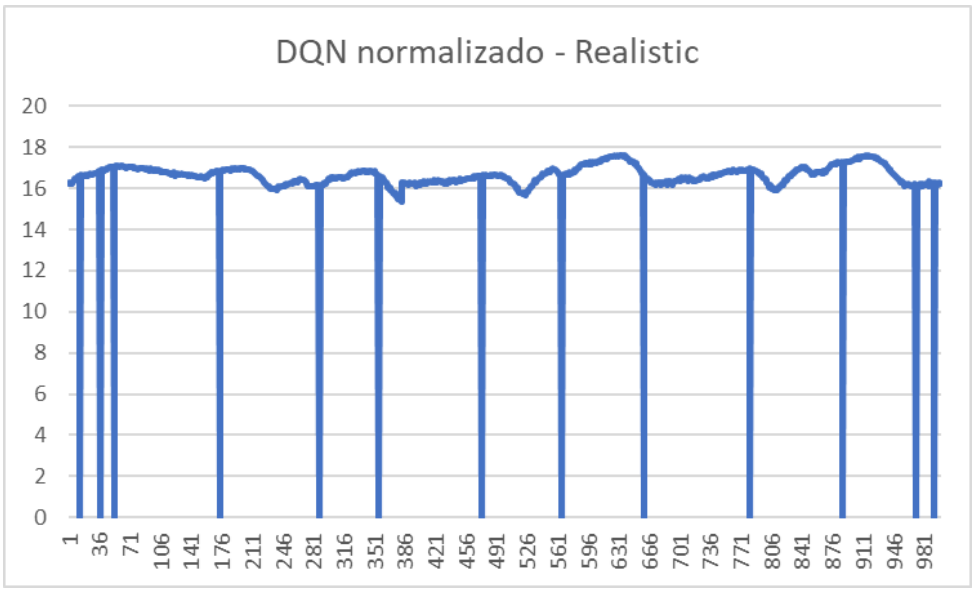


Figura 5.16: Validación de DQN con recompensa normalizada en rte case14 realistic.

Otra modificación que se realizó fue la de controlar el entrenamiento en los casos en los que el agente no realizase cambios sobre la red. De esta manera, el agente no aprendería de esta acción en todos los casos. A continuación se muestran los resultados obtenidos en validación controlando el entrenamientos:

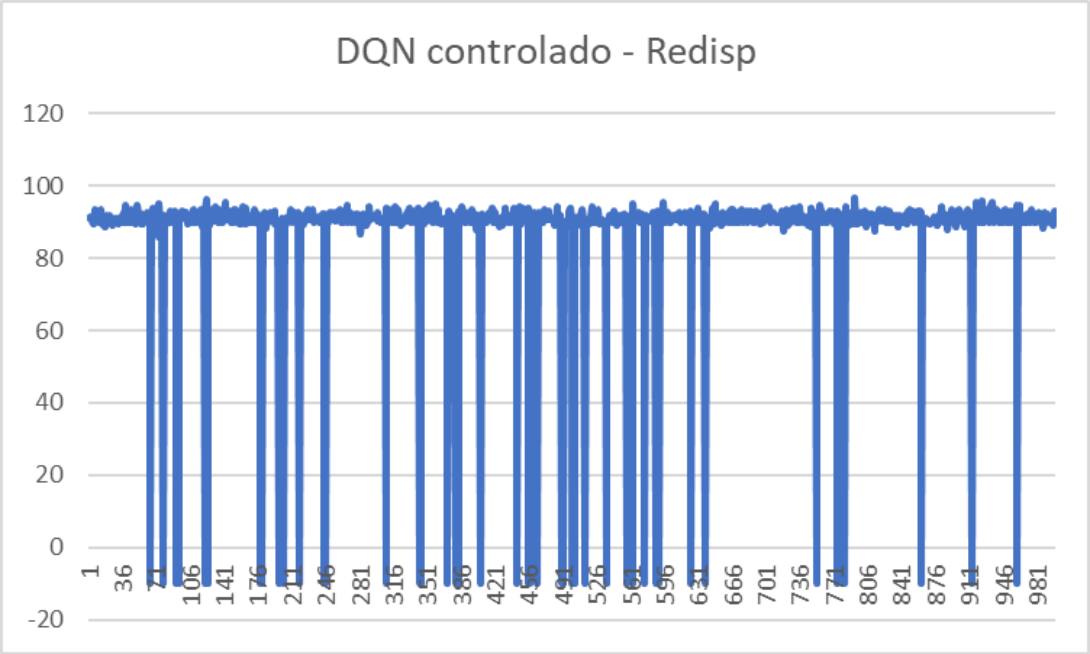


Figura 5.17: Validación de DQN con entrenamiento controlado en rte case14 redisp.

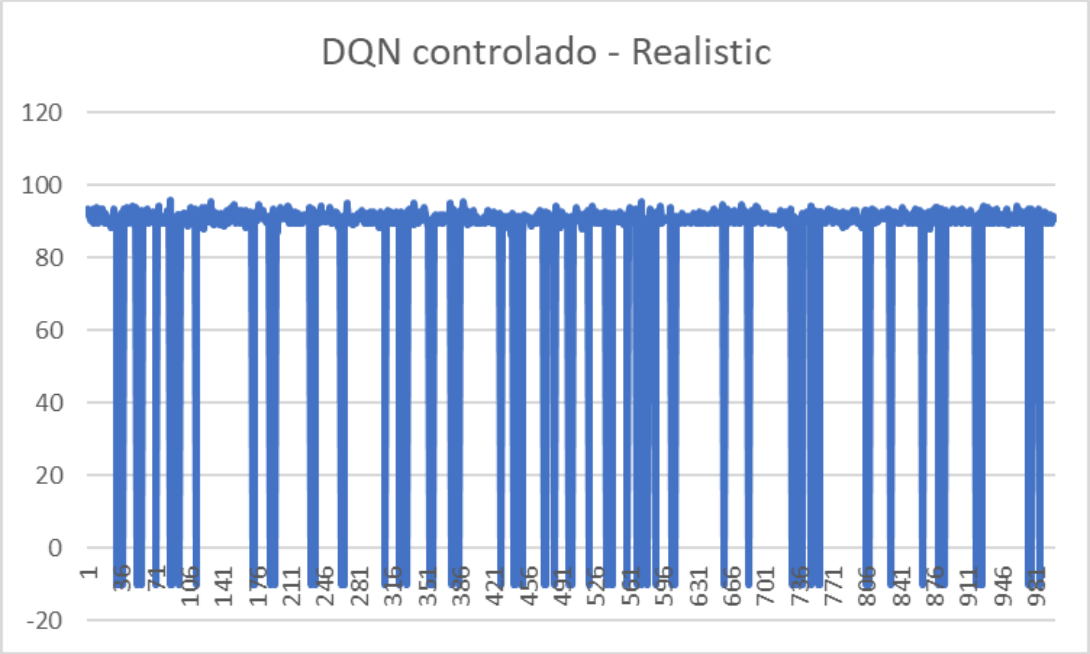


Figura 5.18: Validación de DQN con entrenamiento controlado en rte case14 redisp

Viendo que con ambas modificaciones se habían mejorado los resultados, se optó por observar los resultados de combinarlas.

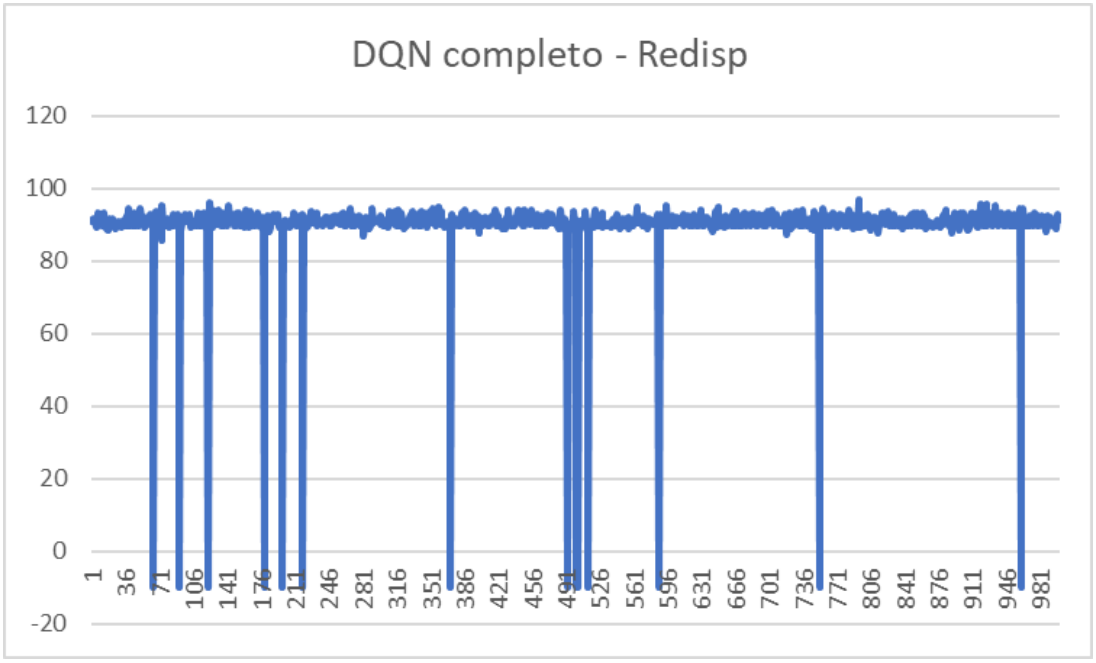


Figura 5.19: Validación de DQN con entrenamiento completo en rte case14 redisp.

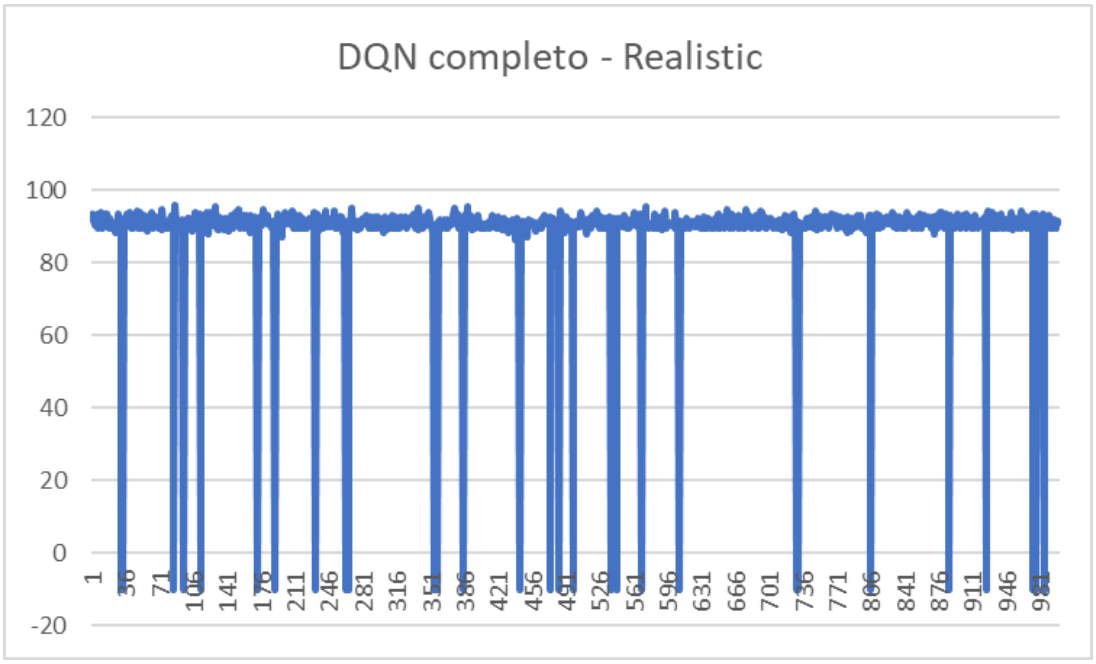


Figura 5.20: Validación de DQN con entrenamiento completo en rte case14 redisp

Finalmente mostramos una tabla de los entrenamientos y el porcentaje de veces que se ha escogido la acción de mantener la red intacta en las 1000 iteraciones de validación.

Agente	Redisp	Realistic
DQN original	100 %	100 %
DQN normalizado	83 %	81 %
DQN controlado (40 %)	67 %	65 %
DQN controlado (50 %)	56 %	54 %
DQN controlado (60 %)	48 %	47 %
DQN controlado (70 %)	41 %	39 %
DQN completo	34 %	32 %

Cuadro 5.2: Porcentaje de veces que los agentes escogen no hacer nada en validación.

Como se puede ver en la tabla, partimos de un agente que escogía siempre esa acción. Con la normalización de la recompensa logramos reducir algo este comportamiento. Controlando directamente el entrenamiento del agente se logró reducir en gran medida que los agentes no modificasen la red. Hay que tener en cuenta que puede haber situaciones en las que realmente la mejor opción será dejar intacta la red, por eso nos pareció prudente trabajar controlando en el aprendizaje el 50 % de las acciones de no modificar la red. El porcentaje que aparece en la tabla a la derecha de cada 'DQN controlado' es el porcentaje de acciones de este tipo de las que no se estaba aprendiendo.

Capítulo 6

Conclusiones

6.1. Conclusiones

Siguiendo los objetivos propuestos al comenzar este trabajo, se puede concluir lo siguiente:

El objetivo principal fue el de construir un agente de aprendizaje por refuerzo profundo para afrontar el desafío L2RPN. La implementación final de DQN cumplió con este objetivo después de realizar las adaptaciones necesarias.

El campo del aprendizaje por refuerzo y el del aprendizaje profundo por refuerzo eran nuevos para mi, apenas había oído hablar de ellos por los logros de Alpha Zero jugando al ajedrez. A raíz de esto, otro de los objetivos fue el de profundizar en estos temas y aprender la teoría que subyace.

Centrándonos en el desafío, el tercer objetivo propuesto fue aprender sobre L2RPN y sobre el framework de Grid2Op. En el apartado 3.1 detallo en que consiste el desafío L2RPN y en el apartado 5.1 muestro toda la experimentación que se ha realizado con Grid2Op.

El cuarto objetivo que se planteó fue trabajar con las librerías de DRL más comunes. En esta línea realizamos trabajos con la librería de Stable Baselines.

Finalmente se propuso realizar la computación en la nube. Como se detalló en los objetivos, se ha usado Google Colab para realizar las tareas más pesadas.

Por último, como conclusión de los resultados de la experimentación, podemos concluir que la librería de Stable Baselines no es la más adecuada para este desafío. El problema de Specification Gaming que nos encontramos es difícil de solucionar cuando no se pueden hacer adaptaciones sobre la implementación de los agentes. Por otro lado, respecto a DQN, se puede ver que se llegó a reducir en gran medida el condicionamiento hacía la acción de no modificar la red.

6.2. Líneas futuras

Nuestro trabajo se tubo que centrar en poder adaptar estos agentes para solucionar algunos problemas imprevistos que encontramos. Partiendo de este trabajo, se podría profundizar en otros métodos para reducir el Specification Gaming como, por ejemplo, modificar el valor de épsilon cuando se selecciona la acción no hacer modificaciones o utilizar dos redes/agentes. Una entrenada para la situación estacionaria y otra para cuando haya que hacer modificaciones en la red.

Otra línea de trabajo puede ser la de mejorar el rendimiento de los agentes. Para ello se podría trabajar en la búsqueda de los mejores hipérparametros para la implementación de DQN o directamente buscar otro agente que puede tener mejores resultados.

Otra propuesta podría ser afrontar el problema como un grafo y plantear el problema para solucionarlo con una red neuronal para grafos (GNN). Son técnicas relativamente nuevas y podría ser interesante ver cómo se comportan con este problema.

Bibliografía

[1] *¿Cómo funcionan las Convolutional Neural Networks? Visión por Ordenador*. URL: https://www.aprendemachinelearning.com/como-funcionan-las-convolutional-neural-networks-vision-por-ordenador/.

[2] *Ajuste de entorno para Gym*. URL: https://grid2op.readthedocs.io/en/latest/gym.html.

[3] *Alan Turing, el padre de la inteligencia artificial*. URL: https://www.cultura.gob.ar/alan-turing-el-padre-de-la-inteligencia-artificial-9162/#:~:text=En%201950%20Turing%20inici%C3%B3su,hoy%20conocemos%20como%20inteligencia%20artificial..

[4] Chema Alonso. *Proyecto Maquet: Una IA para escribir un relato del Capitán Alatriste al estilo de Arturo Pérez-Reverte*. URL: https://www.elladodelmal.com/2021/01/proyecto-maquet-una-ia-para-escribir-un.html.

[5] «AlphaZero: Shedding new light on chess, shogi, and Go». En: (). URL: https://deepmind.com/blog/article/alphazero-shedding-new-light-grand-games-chess-shogi-and-go.

[6] *Aprendizaje profundo por refuerzo*. URL: https://www.iic.uam.es/aprendizaje-profundo-por-refuerzo/.

[7] *Apuntes de aprendizaje por refuerzo*. URL: https://cayetanoguerra.github.io/ia/rl/Aprendizaje_por_refuerzo_apuntes.pdf.

[8] *DALL·E: Creating Images from Text*. URL: https://openai.com/blog/dall-e/.

[9] *DALLE2*. URL: https://openai.com/dall-e-2/.

[10] *Deep Q Learning*. URL: https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/.

[11] *Documentación oficial Grid2Op*. URL: https://grid2op.readthedocs.io/en/latest/.

[12] Hugo Jair Escalante y Katja Hofmann. «Learning to run a Power Network Challenge: a Retrospective Analysis». En: (). DOI: https://drive.google.com/file/d/1LdQ7lCiJCRCA4vwOOjDlWfUkcBECxxn9/view?pli=1.

[13] *Guía de desalladores: propagación inversa*. URL: https://developers-dot-devsite-v2-prod.appspot.com/machine-learning/crash-course/backprop-scroll.

[14] hill-a. *Repositorio GitHub de Stable Baselines*. URL: https://github.com/hill-a/stable-baselines.

[15] *Librería Grid2Op (converter)*. URL: https://grid2op.readthedocs.io/en/latest/converter.html.

[16] Nguyen Cong Luong y col. «Applications of Deep Reinforcement Learning in Communications and Networking: A Survey». En: *IEEE Communications Surveys Tutorials* 21.4 (2019), págs. 3133-3174. DOI: 10.1109/COMST.2019.2916583.

[17] OpenAI. *Repositorio GitHub OpenAI Baselines*. URL: https://github.com/openai/baselines.

[18] JAVIER PASTOR. *AlphaStar es la inteligencia artificial de DeepMind que ha logrado ganar 10-1 a los profesionales de 'StarCraft II'*. URL: https://www.xataka.com/robotica-e-ia/alphastar-inteligencia-artificial-deepmind-que-ha-logrado-ganar-10-1-a-profesionales-starcraft-ii.

[19] Dr. Juan Gómez Romero. *Aprendizaje Profundo por Refuerzo*. URL: https://github.com/jgromero/eci2019-DRL/blob/master/Tema%204%20-%20Aprendizaje%20Profundo%20por%20Refuerzo/Aprendizaje%20profundo%20por%20refuerzo.pdf.

[20] rte-france. *Repositorio GitHub del desafío*. URL: https://github.com/rte-france/Grid2Op/tree/master/getting_started.

[21] rte-france. *Repositorio GitHub rte-france Grid2Op*. URL: https://github.com/rte-france/Grid2Op.

[22] rte-france. *Repositorio GitHub rte-france gridAlive*. URL: https://github.com/rte-france/gridAlive.

[23] *Specification gaming: the flip side of AI ingenuity*. URL: https://www.deepmind.com/blog/specification-gaming-the-flip-side-of-ai-ingenuity.

[24] Jordi TORRES.AI. *10. Métodos policy-based: REINFORCE*. URL: https://medium.com/aprendizaje-por-refuerzo/10-m%C3%A9todos-policy-based-reinforce-2f13c11b290f#:~:text=Con%20el%20algoritmo%20REINFORCE%2C%20el,entre%20s%C3%AD%20en%20gran%20medida..

[25] Wikipedia contributors. *Artificial neural network — Wikipedia, The Free Encyclopedia*. [Online; accessed 30-May-2021]. 2021. URL: https://en.wikipedia.org/w/index.php?title=Artificial_neural_network&oldid=1022750251.

[26] Wikipedia contributors. *Reinforcement learning — Wikipedia, The Free Encyclopedia*. [Online; accessed 4-May-2021]. 2021. URL: https://en.wikipedia.org/w/index.php?title=Reinforcement_learning&oldid=1021388794.

[27] Wikipedia contributors. *Temporal difference learning — Wikipedia, The Free Encyclopedia*. [Online; accessed 6-May-2021]. 2020. URL: https://en.wikipedia.org/w/index.php?title=Temporal_difference_learning&oldid=996979181.

Apéndice A

Apéndice

A.1. Código DQN

Código del agente:

```
from grid2op.Agent import MLAgent
from grid2op.Converter import ToVect

import numpy as np
import random
from collections import namedtuple, deque

from Grid2Op.grid2op.Converter.IdToAct import IdToAct
from model import QNetwork

import torch
import torch.nn.functional as F
import torch.optim as optim

BUFFER_SIZE = int(1e5)  # replay buffer size (size D)
BATCH_SIZE = 64  # minibatch size (n_batch)
GAMMA = 0.99  # discount factor (gamma)
TAU = 1e-3  # for soft update of target parameters (tau)
LR = 5e-4  # learning rate (eta)
UPDATE_EVERY = 4  # how often to update the target network (C)

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")


class DqnGrid2op(MLAgent):

    def __init__(self, ENV, seed, observation_space_converter=ToVect,
        action_space_converter=IdToAct,
                 **kwargs_converter):
        MLAgent.__init__(self, ENV.action_space, action_space_converter, **
            kwargs_converter)
        self.max_reward = 0
        self.reward_range = ENV.reward_range
        self.action_space = ENV.action_space
        self.do_nothing_act = self.action_space()
        self.action_converter = action_space_converter(self.action_space)  # (64, 200,
            200, 3)
        self.action_converter.seed(0)
        self.action_converter.init_converter(change_bus_vect=True, redispatch=True,
            curtail=False, storage=True,
                                             change_line_status=True, set_line_status=
                                                 True)

```

```
39          self.observation_converter = observation_space_converter(self.action_space)
40          self.action_converter.seed(0)
41          self.action_converter.init_converter()
42
43          sample_obs_vect = self.observation_converter.convert_obs(ENV.reset())
44
45          self.state_size = len(sample_obs_vect)
46          self.action_size = self.action_converter.n
47          random.seed(seed)
48
49          ##BUilding the modeL:
50          # Q-Network
51          units = self.state_size+self.action_size
52
53          self.qnetwork_local = QNetwork(self.state_size, self.action_size, seed,fc1_units=
                units, fc2_units=units).to(device)
54          self.qnetwork_target = QNetwork(self.state_size, self.action_size, seed,fc1_units
                =units, fc2_units=units).to(device)
55          self.optimizer = optim.Adam(self.qnetwork_local.parameters(), lr=LR)
56
57          # Replay memory
58          self.memory = ReplayBuffer(self.action_size, BUFFER_SIZE, BATCH_SIZE, seed)
59          # Initialize time step (for updating every UPDATE_EVERY steps)
60          self.t_step = 0
61
62      def convert_obs(self, observation):
63          sample_obs_vect = self.observation_converter.convert_obs(observation)
64          return sample_obs_vect
65          # convert the observation
66          # return np.concatenate((observation.load_p, observation.rho + observation.p_or))
67
68      def convert_act(self, encoded_act):
69          return self.action_converter.convert_act(int(encoded_act))
70
71      def act(self, observation, reward, done=False):
72          return self.convert_act(self._act(obs=observation, eps=100))
73
74      def norm_reward(self,reward):
75          return (reward/self.reward_range[1])*(reward/self.reward_range[1]) * (100 if
                reward > 0 else 1)
76
77      def _act(self, obs, eps=0.):
78          """Returns actions for given state as per current policy.
79          Params
80          ======
81              state (array_like): current state
82              eps (float): epsilon, for epsilon-greedy action selection
83          """
84          state = self.convert_obs(obs)
85          state = torch.from_numpy(state).float().unsqueeze(0).to(device)
86          self.qnetwork_local.eval()
87          with torch.no_grad():
88              action_values = self.qnetwork_local(state)
89          self.qnetwork_local.train()
90
91          # Epsilon-greedy action selection
92          if random.random() > eps:
93              return np.argmax(action_values.cpu().data.numpy())
94          else:
95              return random.choice(np.arange(self.action_size))
96
97      def step(self, state, action, reward, next_state, done):
98          # ------------------- train with mini-batch sample of experiences
                ------------------- #
99          if len(self.memory) > BATCH_SIZE:
100             # If enough samples are available in memory, get random subset and learn
101             experiences = self.memory.sample()
102             self.learn(experiences, GAMMA)
103
104         # ------------------- update target network
```

```
            # ----------------------------------------- #
        self.t_step = (self.t_step + 1) % UPDATE_EVERY
        if self.t_step == 0:
            # If C (UPDATE_EVERY) steps have been reached, blend weights to the target network
            self.soft_update(self.qnetwork_local, self.qnetwork_target, TAU)

    def learn(self, experiences, gamma):
        """Update value parameters using given batch of experience tuples.
        Params
        ======
            experiences (Tuple[torch.Tensor]): tuple of (s, a, r, s', done) tuples
            gamma (float): discount factor
        """
        states, actions, rewards, next_states, dones = experiences

        # Get max predicted Q values (for next states) from target model
        # - qnetwork_target : apply forward pass for the whole mini-batch
        # - detach : do not backpropagate
        # - max : get maximizing action for each sample of the mini-batch (dim=1)
        # - [0].unsqueeze(1) : transform output into a flat array
        Q_targets_next = self.qnetwork_target(next_states).detach().max(1)[0].unsqueeze(1)

        # Compute Q targets for current states (y)
        # - dones : detect if the episode has finished
        Q_targets = rewards + (gamma * Q_targets_next * (1 - dones))

        # Get expected Q values from local model (Q(Sj, Aj, w))
        # - gather : for each sample select only the output value for action Aj
        Q_expected = self.qnetwork_local(states).gather(1, actions)

        # Optimize over (yj-Q(Sj, Aj, w))^2
        # * compute loss
        loss = F.mse_loss(Q_expected, Q_targets)
        # * minimize the loss
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()

    def soft_update(self, local_model, target_model, tau):
        """Soft update model parameters.
          _target   =    * _local   + (1 -   )* _target
        Params
        ======
            local_model (PyTorch model): weights will be copied from
            target_model (PyTorch model): weights will be copied to
            tau (float): interpolation parameter
        """
        for target_param, local_param in zip(target_model.parameters(), local_model.parameters()):
            target_param.data.copy_(tau * local_param.data + (1.0 - tau) * target_param.data)


class ReplayBuffer:
    """Fixed-size buffer to store experience tuples."""

    def __init__(self, action_size, buffer_size, batch_size, seed):
        """Initialize a ReplayBuffer object.
        Params
        ======
            action_size (int): dimension of each action
            buffer_size (int): maximum size of buffer
            batch_size (int): size of each training batch
            seed (int): random seed
        """
        self.action_size = action_size
        self.memory = deque(maxlen=buffer_size)
        self.batch_size = batch_size
```

```
170         self.experience = namedtuple("Experience", field_names=["state", "action", "
                reward", "next_state", "done"])
171         self.seed = random.seed(seed)
172 
173     def add(self, state, action, reward, next_state, done):
174         """Add a new experience to memory."""
175         e = self.experience(state, action, reward, next_state, done)
176         self.memory.append(e)
177 
178     def sample(self):
179         """Randomly sample a batch of experiences from memory."""
180         experiences = random.sample(self.memory, k=self.batch_size)
181         states = torch.from_numpy(np.vstack([e.state for e in experiences if e is not
                None])).float().to(device)
182         actions = torch.from_numpy(np.vstack([e.action for e in experiences if e is not
                None])).long().to(device)
183         rewards = torch.from_numpy(np.vstack([e.reward for e in experiences if e is not
                None])).float().to(device)
184         next_states = torch.from_numpy(np.vstack([e.next_state for e in experiences if e
                is not None])).float().to(
185             device)
186         dones = torch.from_numpy(np.vstack([e.done for e in experiences if e is not None
                ]).astype(np.uint8)).float().to(
187             device)
188 
189         return (states, actions, rewards, next_states, dones)
190 
191     def __len__(self):
192         """Return the current size of internal memory."""
193         return len(self.memory)
```

Código de la red neuronal del agente anterior:

```
1  import torch
2  import torch.nn as nn
3  import torch.nn.functional as F
4  
5  class QNetwork(nn.Module):
6      """Actor (Policy) Model."""
7  
8      def __init__(self, state_size, action_size, seed, fc1_units=64, fc2_units=64):
9          """Initialize parameters and build model.
10         Params
11         ======
12             state_size (int): Dimension of each state
13             action_size (int): Dimension of each action
14             seed (int): Random seed
15             fc1_units (int): Number of nodes in first hidden layer
16             fc2_units (int): Number of nodes in second hidden layer
17         """
18         super(QNetwork, self).__init__()
19         self.seed = torch.manual_seed(seed)
20         self.fc1 = nn.Linear(state_size, fc1_units)
21         self.fc2 = nn.Linear(fc1_units, fc1_units+fc2_units)
22         self.fc3 = nn.Linear(fc1_units+fc2_units, fc1_units+fc2_units)
23         self.fc4 = nn.Linear(fc1_units+fc2_units, fc2_units)
24         self.fc5 = nn.Linear(fc2_units, action_size)
25 
26     def forward(self, state):
27         """Build a network that maps state -> action values."""
28         x = F.relu(self.fc1(state))
29         x = F.relu(self.fc2(x))
30         x = F.relu(self.fc3(x))
31         x = F.relu(self.fc4(x))
32         return self.fc5(x)
```