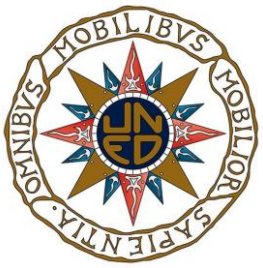

UNIVERSIDAD NACIONAL DE EDUCACIÓN A DISTANCIA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

Proyecto de Fin de Grado en Ingeniería Informática

Aplicación de técnicas de aprendizaje profundo por refuerzo para la optimización energética en plantas de tratamiento de aguas residuales mediante el control inteligente del proceso de eliminación de nitrógeno.

Oscar José Pellicer Valero

Dirigido por: Félix Hernández del Olmo

Curso: 2020-2021 Convocatoria de Junio

Aplicación de técnicas de aprendizaje profundo por refuerzo para la optimización energética en plantas de tratamiento de aguas residuales mediante el control inteligente del proceso de eliminación de nitrógeno.

MEMORIA

Proyecto de Fin de Grado en Ingeniería Informática

Modalidad Externo

Oscar José Pellicer Valero

Dirigido por: Félix Hernández del Olmo

Curso: 2020-2021 Convocatoria de Junio

Aplicación de técnicas de aprendizaje profundo por refuerzo para la optimización energética en plantas de tratamiento de aguas residuales mediante el control inteligente del proceso de eliminación de nitrógeno.

RESUMEN

En la actual situación de creciente escasez de recursos hídricos, energéticos y el aumento de la contaminación, las estaciones de depuración de aguas residuales (EDAR) se sitúan en el punto de mira de la política global. En concreto, el proceso de aireación de fangos activos es fundamental para reducir la concentración de nitratos en el agua efluente de una EDAR; sin embargo, esto es a costa de un enorme consumo eléctrico. Del equilibrio entre calidad del efluente y consumo energético surge un complejo problema de optimización que en este proyecto se aborda mediante el uso de agentes de aprendizaje por refuerzo (*Reinforcement Learning,* RL). Para ello se emplea un modelo de una EDAR de referencia, el *Benchmark Simulation Model No. 1* (BSM1) implementado en Modelica, el cual se encapsula en un entorno de la librería de Python OpenAI Gym, facilitando así enormemente el diseño y entrenamiento de agentes de RL, y desacoplando el modelo en sí de todo el resto de elementos exógenos, los cuales se gestionan desde Python. Finalmente se desarrollan y configuran tres tipos de agentes: Q-Learning, Deep Q-Learning y EV-SARSA, obteniendo ahorros similares y sostenidos en todos ellos de más de 2000 € al año, incluso frente a patrones meteorológicos diversos. Por su alta tasa de exploración, se argumenta que estos agentes deberían ser capaces de adaptarse aun cuando las condiciones de funcionamiento de la planta cambiaran radicalmente.

Todo el código se encuentra disponible en: https://github.com/OscarPellicer/BSM1_gym

Palabras clave: ahorro energético, aprendizaje por refuerzo, estación de depuración de aguas residuales, aprendizaje profundo, OpenAI Gym, Pyhton

Application of deep reinforcement learning techniques to the energy consumption optimization of wastewater treatment plants via intelligent control of the process of nitrogen elimination.

ABSTRACT

Amidst the current environment of ongoing hydric and energy shortages, as well as the sustained increase in pollution, wastewater treatment plants (WWTP) are now, more than ever, in the spotlight of global politics. More precisely, the active sludge process is instrumental in reducing the concentration of nitrates in WWTP effluent waters; however, this is at the expense of huge energy consumptions. In the balance between effluent quality and energy consumption, a complex optimization problem rises, which this project intends to tackle through the use of reinforcement learning (RL) agents. To this end, Benchmark Simulation Model No. 1 (BSM1), a WWTP model implemented in Modelica, is encapsulated within a Python OpenAI Gym environment, facilitating the design and training of RL agents, and decoupling the model in itself from any exogenous elements, which is in turn managed in Python. In the end, three different RL agents are developed and configured: Q-Learning, Deep Q-Learning, and EV-SARSA, attaining all of them similar and sustained savings of over 2000€ per year, even in the presence of varying weather patterns. Due to their high exploration rates, it is argued that these agents should be able to adapt even when the WWTP working conditions changed radically.

The code for this project can be found at: https://github.com/OscarPellicer/BSM1_gym

Key words: energy saving, reinforcement learning, wastewater treatment plant, deep learning, OpenAI Gym, Python

Índice de documentos

Índice de la memoria

Lista de figuras

Lista de tablas

Lista de fragmentos de código

Lista de algoritmos

1. INTRODUCCIÓN

El tratamiento de las aguas residuales se encuentra ahora mismo en el punto de mira de la política global debido a factores como la creciente escasez de recursos hídricos potables y su impacto ambiental (Al-Dosary et al., 2015; United Nations, 2015). Para abordar estos problemas se han propuesto medidas de diverso carácter, orientadas tanto a individuos como a empresas, con el objetivo común de optimizar el consumo de agua y reducir los contaminantes vertidos a esta y -una vez generada el agua residual- para mejorar la calidad del agua depurada y reducir el consumo energético de las estaciones de depuración de aguas residuales (EDAR).

Dentro de una EDAR, el proceso de fangos activos (Safoniuk, 2004) es el encargado la eliminación del nitrógeno, uno de los principales contaminantes presentes en el agua residual, y cuya concentración en el agua efluente de la planta va asociada a una fuerte penalización económica. Este proceso se controla mediante la aireación de los fangos activos, lo cual aumenta el oxígeno disuelto (OD) y favorece los procesos biológicos de degradación de materia orgánica y, en consecuencia, la eliminación del nitrógeno. Ahora bien, este fundamental proceso de fangos activos es simultáneamente el que mayor coste energético conlleva dentro de una EDAR, alcanzando un 50% del consumo energético total.

En la práctica es habitual establecer sencillamente una consigna de OD relativamente alta asegurando así una calidad mínima del agua de salida; sin embargo, algunos autores (Hernández-del-Olmo et al., 2012) han argumentado a favor del potencial de realizar un control automático e inteligente del OD, equilibrando de esta forma el consumo energético de la EDAR con la calidad de su efluente. En la literatura se encuentran multitud de formas de abordar este problema, desde el uso de controladores PID (Meneses et al., 2016), hasta el uso de control predictivo basado en modelo (MPC) (Holenda et al., 2008), o estrategias de control no lineal (Cristea et al., 2011). Sin embargo, la efectividad de estas formas de control se ve comprometida cuando la calidad del caudal influente fluctúa, cuando hay perturbaciones externas, cuando el estado de la planta no es constante o, en general, cuando las condiciones para las que había sido diseñado el control cambian.

Recientemente, una aproximación que ha contado con bastante éxito consiste en implantar un agente de aprendizaje por refuerzo que, recibiendo unas pocas señales de sensores de la planta, sea capaz de establecer automáticamente una consigna del OD que optimice consumo energético y calidad de efluente. En (Hernández-del-Olmo et al., 2012, 2016), los autores emplean un modelo de planta estándar conocido como *Benchmark Simulation Model No. 1* o BSM1 (Alex et al., 2008) implementado en el lenguaje Modelica y desarrollan sobre este un agente de aprendizaje por refuerzo (*Reinforcement Learning,* RL) Q-Learning, que interactuando autónomamente con la EDAR, es capaz de optimizar el coste de operación (el cual subsume tanto el coste monetario de la aireación como el de las multas dependientes de la calidad del efluente). Similarmente, en (Hernández-del-Olmo et al., 2018) se propone un agente de RL implementando el algoritmo de iteración de la política, que durante un periodo inicial es capaz de aprender del operario de la EDAR, y a continuación opera autónomamente, haciéndolo así de forma más eficiente que sin el entrenamiento supervisado previo. Finalmente, en (Hernández-del-Olmo et al., 2019) se propone el uso de un *soft sensor* que utiliza aprendizaje máquina para la predicción del tiempo meteorológico.

Los autores muestran que el uso de este sensor mejora el rendimiento del agente, ya que le permite desarrollar políticas específicas según su valor.

En este trabajo se propone una aproximación muy similar a la desarrollada en (Hernández-del-Olmo et al., 2012, 2016). Al igual que en estos artículos, se utilizará una implementación en Modelica del modelo de EDAR BSM1, y se tratará de desarrollar un agente de RL para controlar el OD minimizando el coste de operación. Sin embargo, este trabajo presenta las siguientes novedades: el modelo BSM1 se encapsulará en un entorno de la librería de Python OpenAI Gym, que se considera el estándar en el desarrollo de agentes de RL. Esto permitirá separar el modelo en sí de todos los demás aspectos como son la gestión de datos de entrada y salida, la implementación de la función de coste, o el propio diseño y entrenamiento de los agentes. Además de la gran comodidad que supone esta forma de trabajar, esto permitirá el desarrollo de agentes más complejos, como los basados en aprendizaje profundo y redes neuronales (sin la limitación de ser implementables en Modelica), y permitirá también el empleo de técnicas como la repetición de la experiencia o el entrenamiento por lotes del agente, gracias a las cuales, como se comprobará, se logra obtener un mayor ahorro económico.

El trabajo se estructurará de la siguiente forma: el capítulo Materiales y métodos sentará las bases teóricas del trabajo, centrándose en el Aprendizaje por refuerzo, el Aprendizaje supervisado, y la Descripción de BSM1 y su coste de operación. El siguiente capítulo, Trabajo realizado describirá, apoyándose en la teoría anterior, todos los pasos que se han realizado para llegar a entrenar agentes de RL (incluyendo agentes de aprendizaje profundo) sobre el BSM1 encapsulado en un entorno de OpenAI Gym desde Python, incluyendo los detalles más relevantes de la implementación de cada uno de los componentes. Finalmente, en Resultados y discusión se presentarán los resultados finales obtenidos, y se discutirán a la luz de artículos similares. El trabajo se cerrará con las Conclusiones.

Todo el código se encuentra disponible en: https://github.com/OscarPellicer/BSM1_gym

2. MATERIALES Y MÉTODOS

En este capítulo se presentarán los principales algoritmos, métodos y técnicas que se han empleado para el desarrollo del trabajo desde un punto de vista teórico. El enfoque práctico y aplicado, en cambio, se ofrecerá en el capítulo Trabajo realizado, donde se utilizarán los algoritmos aquí presentados para resolver el problema propuesto. En concreto, se introducirán primero dos temas principales: aprendizaje por refuerzo y aprendizaje profundo en el contexto del aprendizaje supervisado. Aprendizaje por refuerzo, supervisado y no supervisado son los tres principales pilares sobre los que se asienta el aprendizaje máquina o automático, y que se puede definir de la siguiente forma:

> *"El aprendizaje automático es el campo de estudio que dota a los ordenadores de la capacidad de aprender sin ser explícitamente programados para ello".*
> *Arthur Samuel, 1959*

A muy grandes rasgos, el aprendizaje por refuerzo estudia agentes que interactúan con su entorno y escogen acciones para maximizar una recompensa; por ejemplo, un agente de bolsa puede tomar la acción de comprar o vender un número de acciones, obteniendo por ello una recompensa (que puede ser positiva si hay ganancias, o negativa si hay pérdidas). El aprendizaje supervisado, en cambio, se centra en aprender funciones aproximadas (modelos) capaces de transformar una entrada en una salida de la forma más precisa posible, a base de entrenar sobre muchos ejemplos de pares entrada-salida; por ejemplo, un modelo de detección de spam entrenado en miles de millones de correos de ejemplo podría tomar el texto y metadatos de un correo electrónico como entrada, y producir una clase como salida: spam / no spam. Por último, el aprendizaje no supervisado trata de encontrar y aprender patrones de datos que no están etiquetados; por ejemplo, los algoritmos de *clustering* toman un conjunto de datos no etiquetados y tratan de buscar subconjuntos (o clústeres) de esos datos que son similares entre sí pero diferentes del resto de subconjuntos. Este último tipo de aprendizaje no se tratará en este trabajo.

Finalmente, en una tercera sección se presentarán las características del modelo BSM1 que se empleará en el proyecto a desarrollar.

2.1. Aprendizaje por refuerzo

El aprendizaje por refuerzo, o RL es uno de los tres pilares sobre los que se construye el aprendizaje máquina, y tiene por objetivo el desarrollo de **agentes** inteligentes que, tras observar el **estado** de un **entorno,** toman **acciones** con el objetivo maximizar una **recompensa** acumulativa. La Figura 1 muestra el esquema típico de RL que se acaba de describir.

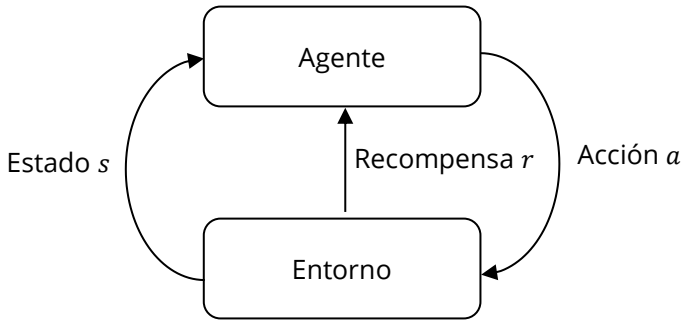

Figura 1: Marco para el aprendizaje por refuerzo

2.1.1. Conceptos básicos

Un entorno clásico, y también uno de los más sencillos es un proceso de decisión de Markov (MDP), cuya función de probabilidad de transición de estados se define en la Ecuación (1). Intuitivamente, esta ecuación nos indica que, estando el entorno en un estado s_t (en el instante de tiempo actual t), la probabilidad de que el estado evolucione hasta s_{t+1} (en el instante de tiempo siguiente $t+1$) solo depende del estado actual s_t y de la acción tomada por el agente a_t. Esta transición tendrá asociada consigo una recompensa r_t para el agente. Además, el agente actuará siguiendo una **política** $\Pi(a_t|s_t)$ o, dicho de otro modo, dado un estado s_t el agente tomará la acción a_t con probabilidad $\Pi(a_t|s_t)$.

$$p_{a_t}(s_t \rightarrow s_{t+1}) = p(s_{t+1}|s_t, a_t) \tag{1}$$

La Figura 2 muestra un ejemplo de MDP extremadamente sencillo. Otro ejemplo: si el agente fuera un jugador de ajedrez, y el entorno fuera un juego de ajedrez contra un oponente, el estado s_t sería la posición de las piezas sobre el tablero en el instante actual, a_t sería la acción tomada por el agente (p. ej., mover un caballo), y s_{t+1} sería el estado del tablero después del movimiento del agente y del oponente. Además, esta transición devengaría una recompensa inmediata r_t para el agente (p. ej., +5 si se ha comido una torre, +10000 si se ha comido al rey, o -5 si ha perdido una torre).

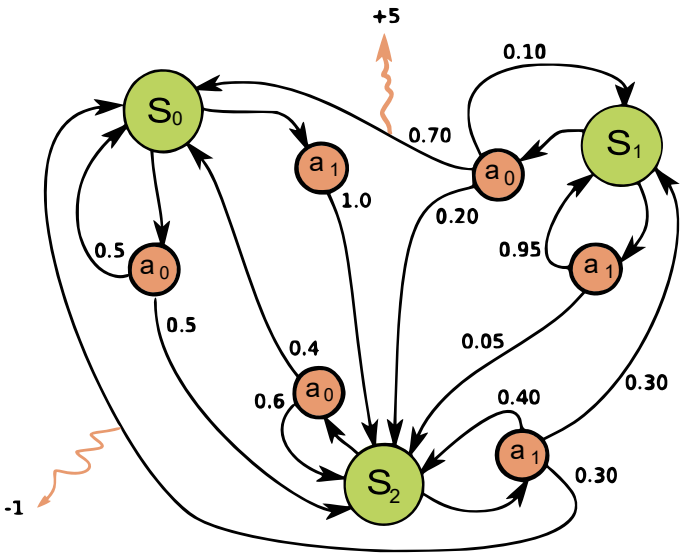

Figura 2: Ejemplo de MDP sencillo, con tres estados S_0, S_1, S_2, dos posibles acciones a_0, a_1, y dos recompensas (el resto se asume que son 0). Fuente: https://commons.wikimedia.org/wiki/File:Markov_Decision_Process.svg

Los entornos que evolucionan de acuerdo la Ecuación (1) se dice que cumplen la propiedad de Markov, esto es, que la probabilidad de evolucionar de $s_t \rightarrow s_{t+1}$ solo depende de s_t y de a_t, y no de nada de lo sucedido con anterioridad (Ecuación (2)). Aunque en la práctica rara vez se cumple esta propiedad, los MDPs son muy útiles en el campo del RL, ya que es habitual modelar entornos como si fueran MDPs, aunque realmente no lo sean.

$$\text{El entorno es un MDP} \leftrightarrow p_{a_t}(s_t \rightarrow s_{t+1}) = p(s_{t+1}|s_t, a_t, s_{t-1}, a_{t-1}, \dots) = p(s_{t+1}|s_t, a_t) \tag{2}$$

En los problemas reales el entorno suele además ser de tiempo continuo, pero el agente solo observa el estado y toma acciones cada un determinado periodo o paso temporal. En el ámbito de RL, conocemos como **episodio** o sesión una interacción prolongada del agente con el entorno durante un tiempo, hasta que se cumple una condición determinada. Por ejemplo, en el caso de un agente que juega al ajedrez, un episodio sería razonable que durase exactamente una partida, mientras que en el caso de un agente que optimiza el consumo eléctrico en una EDAR, un episodio podría durar para siempre, o se podría elegir un intervalo de tiempo arbitrario, como un año.

En este contexto, R. Sutton plantea su hipótesis de la recompensa:

> *"Todo a lo que nos referimos por objetivos y propósitos puede ser interpretado como la maximización del valor esperado de la suma cumulativa de una señal escalar recibida (llamada recompensa)."* (Sutton & Barto, 2018)

Esta hipótesis se describe matemáticamente en las Ecuaciones (3) y (4). En (3) se define el retorno G_t en un instante de tiempo t como la suma de todas las recompensas futuras que espera recibir el agente siguiendo su política Π, mientras que en (4) se verifica que el agente tomará la acción a_t^* que maximice su retorno esperado G_t. Por simplicidad se han omitido de momento las variables aleatorias sobre las que se calcula esta esperanza.

$$G_t \triangleq r_t + r_{t+1} + \cdots + r_T \tag{3}$$

$$a_t^* = \underset{a_t}{\operatorname{argmax}}\, \mathbb{E} G_t \tag{4}$$

En entornos que pueden durar indefinidamente, la definición anterior de G_t puede dar lugar al problema del horizonte infinito, es decir, que para todas las acciones posibles en un estado dado, la G_t sea infinita. Por ejemplo, un agente de conducción autónoma que recibe +1 de recompensa a cada paso de tiempo que no se ha chocado, podría adolecer de este problema. Una solución habitual es utilizar descuento de recompensas, tal y como se define en la Ecuación (5). En esencia, se multiplica la recompensa futura por un factor γ^k que es cada vez más pequeño, de tal forma que cuanto más lejos en el futuro esté la recompensa, menos peso tiene sobre G_t. Además, para $\gamma \in (0,1)$ se verifica $\sum_{k=0}^{\infty} \gamma^k = \frac{1}{1-\gamma}$, luego G_t estará acotado siempre y cuando las recompensas lo estén (y siempre lo estarán, ya que las diseñamos nosotros para que lo estén).

$$G_t \triangleq r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k} \tag{5}$$

Además de este problema, en la práctica a menudo nos encontramos con el problema de la observabilidad parcial del estado del entorno. Es decir, el esquema de la Figura 1, ahora se transforma en el de la Figura 3, donde el agente ya no es capaz de observar directamente el valor del estado de entorno, sino que solo ve una parte, potencialmente transformada, de este.

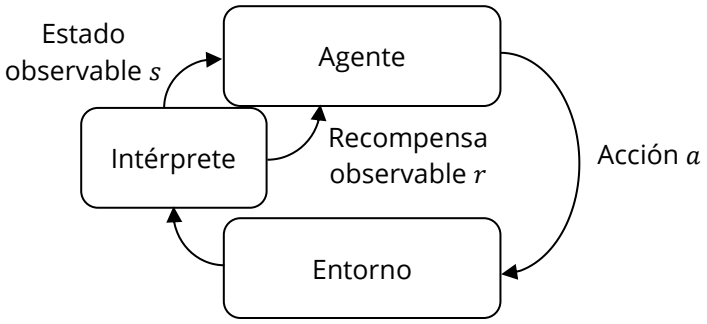

Figura 3: Marco para el aprendizaje por refuerzo con observabilidad parcial

La consecuencia de esta observabilidad parcial es que el estado observado ya no cumple la propiedad de Markov, y por tanto $p_{a_t}(s_t \to s_{t+1})$ puede depender de un número arbitrario de estados pasados. Esto es muy habitual en la práctica, donde independientemente de si el entorno realmente es o no un MDP, esto es en última instancia irrelevante porque no podemos observar todos sus estados. Una de las posibles soluciones consiste en extender el estado para que incluya los N estados anteriores, es decir, utilizar un estado extendido $\tilde{s} = [s_t, s_{t-1}, \dots, s_{t-N}]$. En determinados entornos, esto puede devolver la observabilidad total al entorno, o al menos reducir el grado de discrepancia del entorno con el de un MDP puro. Por ejemplo, si queremos controlar un péndulo invertido, pero solo podemos medir la posición del péndulo, observando los últimos $N = 2$ estados podemos ya reconstruir la velocidad y aceleración del sistema, recuperando por tanto la observabilidad total del mismo.

Para cerrar esta introducción, y a modo de curiosidad, en un artículo del 2021 varios autores muy relevantes en el campo llevan la hipótesis de la recompensa presentada anteriormente aún más lejos, y plantean una nueva hipótesis: la recompensa es suficiente:

"[...] La inteligencia, y sus habilidades asociadas, se puede entender que se supeditan a la maximización de una recompensa. Por tanto, la recompensa es suficiente para impulsar el desarrollo de comportamiento que exhiba las habilidades estudiadas en la inteligencia tanto natural como artificial, incluyendo conocimiento, aprendizaje, percepción, inteligencia social, lenguaje, generalización e imitación." (Silver et al., 2021)

2.1.2. Ecuaciones de Bellman

La función valor-estado de Bellman $V_\Pi(s)$ se define en la Ecuación (6) como el retorno G_t esperado que un agente obtendría empezando a actuar en el estado s y siguiendo la política Π en adelante.

$$V_\Pi(s) \triangleq \mathbb{E}_\Pi[G_t|s_t = s] \tag{6}$$

Desarrollando (6), llegamos a las expresiones de la Ecuación (7), donde se ha representado s' como el estado del entorno en el paso de tiempo posterior al estado s, (es decir $s' = s_{t+1}$) y del mismo modo para r y para a. Primero, hemos usado la igualdad $G_t \triangleq r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots = r_t + \gamma \cdot G_{t+1}$, y la hemos sustituido en (6). A continuación, se ha aprovechado el hecho que la esperanza es un operador lineal y se ha expandido la política Π sobre la que se evalúa por separado para r_t y G_{t+1}. Finalmente, se reemplaza la esperanza por su definición (se asumen que a, r, s' son variables discretas por simplicidad, pero bastaría reemplazar los sumatorios por integrales si no se asumiera esta hipótesis), y se ha utilizado la propiedad de la marginalización de probabilidades en la igualdad: $\sum_{a,r,s'} \Pi(a, r, s'|s) = \sum_a \Pi(a|s) \sum_{r,s'} p(r, s'|s, a)$.

$$\begin{aligned} V_\Pi(s) \triangleq \mathbb{E}_\Pi[G_t|s_t = s] &= \mathbb{E}_\Pi[r_t + \gamma \cdot G_{t+1}|s_t = s] \\ &= \mathbb{E}_{\Pi(\mathrm{a,r,s'|s=s_t})} r_t + \gamma \cdot \mathbb{E}_{\Pi(\mathrm{a,r,s',a',r',s'',a'',r'',s''',...|s=s_t})} G_{t+1} \\ &= \mathbb{E}_{\Pi(\mathrm{a,r,s'|s=s_t})} \left\{ r_t + \gamma \cdot \mathbb{E}_{\Pi(\mathrm{a',r',s'',a'',r'',s''',...|s'=s_{t+1}})} G_{t+1} \right\} \\ &= \sum_a \Pi(a|s) \sum_{r,s'} p(r, s'|s, a) \{ r_t + \gamma \cdot \mathbb{E}_\Pi[G_{t+1}|s_{t+1} = s'] \} \end{aligned} \tag{7}$$

Por tanto en la Ecuación (7): $\sum_a \Pi(a|s)$ representa la estocasticidad del agente, o el sumatorio de $\Pi(a|s)$ sobre todas las posibles acciones que el agente puede tomar en un estado s; $\sum_{r,s'} p(r, s'|s, a)$ representa la estocasticidad del entorno, o el sumatorio de $p(r, s'|s, a)$ sobre todos las posibles recompensas r y todos los posibles siguientes estados s' hacia los que el entorno puede evolucionar partiendo de s y habiendo el agente tomado la acción a; r_t es la recompensa en el instante actual; y finalmente $\mathbb{E}_\Pi[G_{t+1}|s_{t+1} = s'] = V_\Pi(s')$, es decir, la función valor-estado evaluada en s'. La Ecuación (8) muestra el resultado de los cálculos en (7) de forma algo más compacta.

$$V_\Pi(s) = \sum_a \Pi(a|s) \sum_{r,s'} p(r, s'|s, a) \{ r_t + \gamma \cdot V_\Pi(s_{t+1}) \} = \mathbb{E}_\Pi[r_t + \gamma \cdot V_\Pi(s_{t+1})|s_t = s] \tag{8}$$

La función valor-acción de Bellman $q_\Pi(s, a)$, se define en la Ecuación (9). En esencia, es idéntica a la función valor-estado, pero elimina la estocasticidad en la primera acción (se puede elegir la acción a que se quiera, sin que esta deba seguir necesariamente Π, aunque el resto de las acciones a partir de este punto sí deberán seguir Π). Expandiendo (9) obtenemos la Ecuación (10). Sustituyendo (10) en (8) se obtiene la Ecuación (11). Del mismo modo, sustituyendo (11) en (10) se obtiene la Ecuación (12).

$$q_{\Pi}(s,a) = \mathbb{E}_{\Pi}[G_t | s_t = s, a_t = a] \tag{9}$$

$$q_{\Pi}(s,a) = \sum_{r,s'} p(r,s'|s,a)\,[r + \gamma \cdot V_{\Pi}(s_{t+1})] \tag{10}$$

$$V_{\Pi}(s) = \sum_{a} \Pi(a|s)\, q_{\Pi}(s,a) \tag{11}$$

$$q_{\Pi}(s,a) = \sum_{r,s'} p(r,s'|s,a)\,[r + \gamma \cdot \sum_{a'} \Pi(a'|s')\, q_{\Pi}(s',a')] \tag{12}$$

Las Ecuaciones (8) y (12), que han sido definidas de forma recursiva, nos permitirán encontrar políticas óptimas utilizando programación dinámica, como se verá en la sección Aprendizaje sin modelo. Se dirá que una política es óptima (Π^*) si es aquella que da lugar al mayor $V_{\Pi}(s)$ para todos los estados s, y tendrá la función valor-estado óptima (V_*) definida en la Ecuación (13). Equivalentemente, la función valor-acción óptima se muestra en la Ecuación (14).

$$V_*(s) = V_{\Pi^*}(s) = \max_{\Pi} V_{\Pi}(s) \;\; \forall s \tag{13}$$

$$q_*(s,a) = q_{\Pi^*}(s,a) = \max_{\Pi} q_{\Pi}(s,a) \;\; \forall s \tag{14}$$

Expandiendo estas expresiones y utilizando (8) y (12), obtenemos las ecuaciones de optimalidad de Bellman para $V(s)$ (Ecuación (15)) y para $\mathrm{q(s,a)}$ (Ecuación (16)), respectivamente.

$$V_*(s) = \max_{a} \sum_{r,s'} p(r,s'|s,a)\,[r_t + \gamma \cdot V_*(s')] \tag{15}$$

$$q_*(s,a) = \sum_{r,s'} p(r,s'|s,a)\,[r + \gamma \cdot \max_{a'} q_{\Pi}(s',a')] \tag{16}$$

2.1.3. Aprendizaje sin modelo

En el campo del RL existe una enorme variedad de algoritmos. Una clasificación inicial podría separarlos entre aquellos que requieren que el modelo del entorno $p(r,s'|s,a)$ sea conocido, y aquellos que no. Dentro de los primeros, los más utilizados son el algoritmo de iteración de política, y el de iteración de valor. Por ejemplo, y a muy grandes rasgos, el algoritmo de iteración de valor utiliza la ecuación de optimalidad de Bellman para $V(s)$ (Ecuación (15)) para actualizar de forma iterativa una aproximación a $V_*(s)$. Ahora bien, dado que la Ecuación (15) requiere calcular el máximo de una esperanza sobre $p(r,s'|s,a)$, la dinámica del entorno debe ser necesariamente conocida para poder llevar a cabo esta actualización. No obstante, en la mayoría de las ocasiones esta no lo es, y no es por tanto posible aplicar esta clase de algoritmos.

En esta sección se centrará la atención en el segundo tipo de algoritmos: aquellos que no requieren un modelo del entorno. En concreto, se estudiarán dos tipos de algoritmos de aprendizaje sin modelo que se emplearán en este trabajo: Q-Learning y SARSA/EV-SARSA. En ambos algoritmos, el objetivo será aprender una aproximación a $q(s,a)$, que llamaremos $Q(s,a)$. Una vez $Q(s,a)$ es conocida, entonces la política óptima es sencillamente la que se muestra en la Ecuación (17). Como se aprecia, esta política no requiere calcular el máximo de una esperanza sobre $p(r,s'|s,a)$, y es por tanto innecesario conocer el modelo del entorno para aplicar estos algoritmos.

$$\Pi_*(s) = argmax_a\, Q(s,a) \tag{17}$$

No obstante, en las ecuaciones que definen $q(s,a)$, sí que aparece el término del entorno, el cual por tanto cabrá aproximar también de algún modo. La solución más habitual es utilizar la Ecuación (10) para actualizar $Q(s,a)$ en un esquema en diferencias temporales, aproximando la esperanza sobre el entorno mediante muestreo de Monte-Carlo. Esto se refleja en la Ecuación (18), donde además hemos sustituido $V_\Pi(s_{i+1})$ por su definición en (11). En el último paso de la igualdad se ha asumido que $N = 1$, es decir, que la aproximación por Monte-Carlo solo utiliza una muestra.

$$\begin{aligned} q_\Pi(s,a) = \sum_{r,s'} p(r,s'|s,a)\,[r + \gamma \cdot V_\Pi(s')] &\cong \frac{1}{N}\sum_i^N r_i + \gamma \cdot \mathbb{E}_{a_{i+1}\sim\Pi(a_{i+1}|s_{i+1})} q_\Pi(s_{i+1}, a_{i+1}) \\ &\cong r + \gamma \cdot \mathbb{E}_{a'\sim\Pi(a'|s')} q_\Pi(s',a') \end{aligned} \tag{18}$$

2.1.3.1. Q-Learning

Q-Learning se caracterizará por utilizar directamente la política óptima definida en (17) durante el aprendizaje para realizar la actualización de Q. Por tanto, sustituyendo (17) en (18), la regla de actualización para la función Q queda como se muestra en la Ecuación (19).

$$Q(s,a) = r + \gamma \cdot \mathbb{E}_{a'\sim argmax_{a'}\, Q(s',a')} Q(s',a') = r + \gamma \cdot max_{a'}\, Q(s',a') \tag{19}$$

Con (19), ya podemos plantear el Algoritmo 1: Q-learning (tabular). Como se observa, el algoritmo no es más que la aplicación reiterada de la Ecuación (19) sobre diferentes tuplas $< s,a,r,s' >$ que se han muestreado a partir de la interacción del agente con el entorno, logrando así tener aproximaciones sucesivamente mejores de $q(s,a)$ (usando $Q(s,a)$ para aproximarla), e implementando $Q(s,a)$ como una tabla. En la última línea se realiza la actualización de $Q(s,a)$, la cual como se observa no es directa (es decir: $Q(s,a) \neq \hat{Q}(s,a)$), sino que se hace a través de una media móvil exponencial entre el valor actual $\hat{Q}(s,a)$, y todos los valores anteriores de $Q(s,a)$, aliviando así en cierto modo el enorme ruido estocástico asociado a realizar una aproximación por Monte-Carlo usando solo una muestra. La ratio de actualización α controlará cuánto peso tiene cada nuevo $\hat{Q}$ con respecto al valor Q anterior a la actualización.

Algoritmo 1: Q-learning (tabular)

Entradas: α: ratio de actualización de la media móvil exponencial, una política $\Pi(s)$

Salidas: $Q(s,a)$ aproximación a $q(s,a)$

- Inicializa una tabla $Q(s,a)$ (con ceros, valores aleatorios, etc.)
- Itera hasta convergencia:
  - Muestrea una tupla $< s,a,r,s' >$, donde a se ha obtenido siguiendo una política $\Pi(s)$
  - $\hat{Q}(s,a) = r + \gamma \cdot max_{a'}\, Q(s',a')$
  - $Q(s,a) = \alpha \cdot Q(s,a) + (1-\alpha) \cdot \hat{Q}(s,a)$

Respecto a la política que utiliza el agente, esta se debe elegir en base al **equilibrio exploración/explotación**. Si, por ejemplo, el agente siempre utilizara la política óptima Π_* (aproximada a partir de $Q(s,a)$ mediante (17), pues obviamente la política óptima real se desconoce) durante el entrenamiento, entonces dejaría de explorar muchas acciones que

potencialmente le podrían conducir a una mejor política, y probablemente la tabla $Q(s,a)$ contendría muchos elementos que nunca se han actualizado. Por tanto, es habitual utilizar políticas con un componente estocástico que permita que en ocasiones se elijan acciones que parecen subóptimas, con el objetivo de explorar mejor todo el espacio de acciones. Entre ellas, una de las políticas más sencillas y también de las más utilizadas es la conocida como **ε-voraz**, y que consiste en seguir Π_* (17) con probabilidad $1-\epsilon$, y tomar una acción aleatoria (de entre las posibles dado s) con probabilidad ϵ.

2.1.3.2. SARSA y EV-SARSA

SARSA es un algoritmo muy similar a Q-learning, que difiere únicamente en la política empleada para realizar la actualización de Q, que en este caso será exactamente la misma política que se emplea durante el entrenamiento. En efecto, en Q-learning se explora usando una política ε-voraz, pero se entrena utilizando la política óptima Π_*; por tanto, a Q-learning se le conoce como un algoritmo *off-policy*. SARSA, en cambio, es *on-policy*, lo cual nos permite aproximar la esperanza de la Ecuación (19) del mismo modo que lo hemos hecho con la esperanza sobre la estocasticidad del entorno: por Monte-Carlo, mediante muestreo de una única muestra, la muestra que se corresponde con la acción a' que toma el agente en s' siguiendo su política Π (Ecuación (20)).

$$Q(s,a) = r + \gamma \cdot \mathbb{E}_{a' \sim \Pi(a'|s')} Q(s',a') = r + \gamma \cdot Q(s',a') \tag{20}$$

Si reemplazamos la función de actualización del Algoritmo 1 por la de la Ecuación (20), otenemos el Algoritmo 2: SARSA. Notar que ahora es necesario recoger también la siguiente acción a', ya que esta se requiere en la función de actualización. Precisamante, esta tupla $< s,a,r,s',a' >$ de los valores requeridos para la actualización da nombre al método.

Algoritmo 2: SARSA

Entradas: α: ratio de actualización de la media móvil exponencial, una política $\Pi(s)$

Salidas: $Q(s,a)$ aproximación a $q(s,a)$

- Inicializa una tabla $Q(s,a)$ (con ceros, valores aleatorios, etc.)
- Itera hasta convergencia:
  - Muestrea una tupla $< s,a,r,s',a' >$, donde a y a' se han obtenido siguiendo una política $\Pi(s)$
  - $\hat{Q}(s,a) = r + \gamma \cdot Q(s',a')$
  - $Q(s,a) = \alpha \cdot Q(s,a) + (1-\alpha) \cdot \hat{Q}(s,a)$

Por último, *Expected Value*-SARSA, o EV-SARSA, va un paso más allá de SARSA y calcula la esperanza sobre la estocasticidad del agente de forma exacta, como se puede ver en la Ecuación (21). A diferencia de SARSA, no basta con las obtener una tupla $< s,a,r,s',a' >$ sino que habrá que obtener todas las a' posibles dado un estado s', así como la probabilidad de que el agente elija cada una de esas acciones: $\Pi(a',s')$.

$$Q(s,a) = r + \gamma \cdot \mathbb{E}_{a' \sim \Pi(a'|s')} q_\Pi(s',a') = r + \gamma \cdot \sum_{a'} \Pi(a',s') \cdot Q(s',a') \tag{21}$$

La descripción de este método se puede encontrar en el Algoritmo 3: EV-SARSA.

Algoritmo 3: EV-SARSA

Entradas: α: ratio de actualización de la media móvil exponencial, una política $\Pi(s)$

Salidas: $Q(s,a)$ aproximación a $q(s,a)$

- Inicializa una tabla $Q(s,a)$ (con ceros, valores aleatorios, etc.)
- Itera hasta convergencia:
  - Muestrea una tupla $< s,a,r,s' >$, así como todos los posibles valores de $a' \in A'$ donde a y a' se han obtenido siguiendo la política $\Pi(s)$
  - $\hat{Q}(s,a) = r + \gamma \cdot \sum_{a'} \Pi(a',s') \cdot Q(s',a')$
  - $Q(s,a) = \alpha \cdot Q(s,a) + (1-\alpha) \cdot \hat{Q}(s,a)$

Una de las políticas más habituales en EV-SARSA es la conocida como muestreo de Boltzman, que se describe en la Ecuación (22), donde la función $\mathrm{softmax}$ (Ecuación (23)) transforma un vector de valores reales en una distribución multinomial. Siguiendo esta política, el agente tomará una acción a con una probabilidad que es aproximadamente proporcional al Q-valor que espera obtener por dicha acción $Q(s,a)$. Además, el factor τ controla la estocasticidad de la política, de tal forma que para valores de τ muy altos, todas las acciones tendrán una probabilidad similar de ser elegidas, mientras que para valores de τ bajos, se tenderá a elegir únicamente la acción cuyo Q-valor sea el más alto.

$$\Pi(a|s) = softmax\left(\frac{Q(s,a)}{\tau}\right) \tag{22}$$

$$softmax(\vec{z}) = \frac{e^{\vec{z}}}{\sum_j e^{z_j}} \tag{23}$$

2.1.4. Aproximando Q con aproximadores de funciones generales

Esta última sección sirve de enlace con la siguiente, Aprendizaje profundo (la cual se recomienda leer primero) ya que sienta las bases para la utilización de aproximadores de la función Q generales, que no necesariamente estén basados en una tabla. Esto puede ser deseable, por ejemplo, cuando el espacio de estados es continuo, o cuando el espacio de estados tiene una dimensionalidad muy alta. En ambos casos el tamaño de la tabla Q debe ser muy grande, o incluso infinito (en problemas con estados continuos). En la práctica, Q-learning tabular se puede aplicar aun cuando los estados son continuos, haciendo primero una discretización de estos, si bien esta aproximación no suele ser ideal.

En contraste, si utilizamos un aproximador de funciones general, como puede ser una red neuronal, para aproximar $Q(s,a)$, podemos evitar tener que discretizar estados, y podemos ser capaces de manejar espacios de estados potencialmente mucho más grandes, debido a la capacidad de compactación de la información de estos algoritmos. En la práctica se han obtenido muy buenos resultados en entornos altamente complejos, siendo quizás el algoritmo Deep Q-Network el principal referente que dio inicio a este campo, logrando en 2013 un rendimiento sobrehumano en muchos juegos de Atari (Mnih et al., 2013). Más recientemente, en 2016, el algoritmo Alpha Go (Silver et al., 2016), basado también en Q-Learning profundo logró ganar al campeón del mundo en Go, un juego de mesa asiático similar, e incluso más complejo que ajedrez. Este acontecimiento, relativamente ignorado en occidente, supuso una llamada de atención a China, que comenzó a invertir grandes

cantidades de recursos en el campo de la inteligencia artificial, posicionando al país a día de hoy como uno de los pesos pesados en el campo. Más recientemente, en 2019, OpenAI Five (OpenAI et al., 2019) se convirtió en el primer sistema de inteligencia artificial en ganar a los campeones mundiales en el juego altamente competitivo Dota 2, utilizando técnicas similares.

Comenzaremos definiendo la función Q a aproximar como $Q: s \rightarrow \overrightarrow{q_\Pi(s,a)}$, la cual tomará un estado de entrada, y generará un vector de salida con el Q-valor de todas las acciones posibles que se pueden tomar en dicho estado. Esta función estará determinada por unos parámetros θ, respecto de los cuales se deberá minimizar una función de coste que ayude a lograr el objetivo que perseguimos: aproximar la q real (para más información, por favor consultar la sección Aprendizaje profundo). Una posible función de coste será la del error cuadrado medio (mean square error, MAE), que aplicada al problema de aproximar q en el algoritmo Q-learning da lugar a la Ecuación (24).

$$\mathcal{L}(\theta) = \frac{1}{2N}\sum_{s,a}(r + \gamma \cdot max_{a'}\, Q_\theta(s', a') - \bar{Q}_\theta(s, a))^2 \tag{24}$$

$\bar{Q}$ se debe entender como una referencia a la que Q quiere parecerse, que idealmente sería $\bar{Q} = q$, si acaso q fuera conocida. En la práctica, $\bar{Q}$ suele ser una versión estática de la propia función Q, a la que se le impide que propague el gradiente a la hora de realizar la optimización. Al minimizar esta función de coste $\mathcal{L}$, se logrará que $r + \gamma \cdot max_{a'}\, Q_\theta(s', a')$ y $\bar{Q}_\theta(s, a)$ tengan valores lo más similares posibles, logrando así que Q_θ converja a la aproximación de q en virtud del método de diferencias temporales.

En la literatura se han hecho propuestas muy variadas sobre cómo aproximar $\bar{Q}$ con el objetivo de aumentar la estabilidad del método. Por ejemplo, una propuesta alternativa es utilizar una versión "congelada" o "más estática" de Q, donde los parámetros θ de $\bar{Q}$ se copian desde Q cada cierto tiempo, o se calculan como una media móvil exponencial de los valores que toman los pesos de Q a largo del entrenamiento (van Hasselt et al., 2015). Otra posibilidad es utilizar dos aproximadores de funciones Q_1 y Q_2 con pesos totalmente diferentes, y que se van intercambiando los papeles de Q y $\bar{Q}$ según algún criterio (por ejemplo, con una probabilidad del 50%) (Wang et al., 2015).

Pese a todo, resulta complicado aplicar estos algoritmos en la práctica, ya que presentan problemas de convergencia y otras dificultades de todo tipo. Otra técnica que se introdujo junto con la Deep Q-Network es la repetición de experiencia, que consiste en mantener un buffer de las últimas K tuplas $< s, a, r, s' >$ muestreadas. Por tanto, a la hora de actualizar la función Q, en lugar de muestrear solo la última tupla, se muestrea un lote de tuplas del buffer. Esto permite aumentar la eficiencia de las muestras (ya que se utilizan en más de una ocasión) y permite reducir la varianza de las actualizaciones (menos ruido en la aproximación por Monte Carlo de la esperanza sobre el entorno). Sin embargo, dado que se utilizan tuplas correspondientes a políticas anteriores, que por lo general son más débiles que la actual, estas actualizaciones serán menos valiosas, y además solo se podrán utilizar en algoritmos *off-policy*, como Q-learning, ya que la actualización incluye muestras obtenidas bajo una política diferente a la actual (al menos en teoría, ya que se podría utilizar p. ej. un tamaño de buffer muy pequeño). Para mitigar estas dificultades, también puede utilizarse repetición de experiencia con prioridad, de tal forma que la probabilidad de muestreo de cada tupla, en lugar de ser uniforme, sea inversamente proporcional a su antigüedad.

2.2. Aprendizaje supervisado

El aprendizaje supervisado se centra en el desarrollo de modelos capaces de aproximar cualquier relación entrada-salida en base a ejemplos. La Figura 4 muestra un esquema general del paradigma de aprendizaje supervisado. Como se puede ver, el modelo no es más que una función matemática paramétrica $f_\theta(x)$ con parámetros θ, que toma una x como entrada y produce una salida $\hat{y}$. Durante el proceso de entrenamiento, al modelo se le presentarán muchos pares $< x, y >$ de entrenamiento, y su objetivo será modificar sus parámetros θ de tal forma que la salida $\hat{y}$ se parezca lo máximo posible a la muestra real y para una x dada, y para todos los pares de entrenamiento.

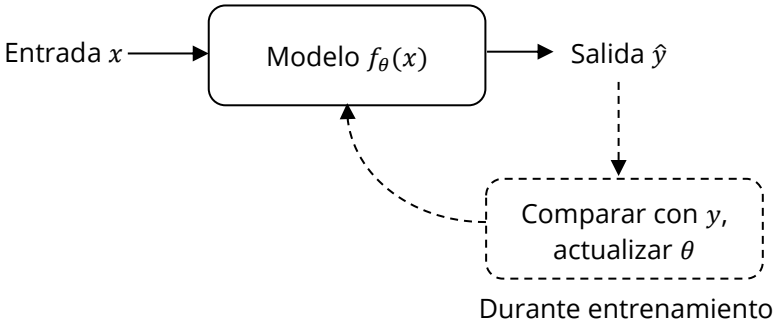

Figura 4: Marco para el aprendizaje supervisado

2.2.1. Conceptos básicos

Las intuiciones anteriores se pueden formalizar con el concepto de función de coste, que será la función que valore numéricamente cuánto discrepan las predicciones del modelo $\hat{y}$ con respecto a los valores reales y. Una de las funciones de coste más usuales es el error cuadrado medio (*mean square error*, MAE), que está definido en la Ecuación (25), donde N es el número de muestras en el conjunto de entrenamiento y K es la dimensionalidad de la salida.

$$MSE(\hat{y}, y) = \frac{1}{NK} \cdot \sum_{n}^{N} \sum_{k}^{K} \left(\hat{y}_{n,k} - y_{n,k}\right)^2 \tag{25}$$

Por tanto, el objetivo del entrenamiento se puede definir como la minimización de una función de coste J (como podría ser el MSE), con respecto a los parámetros θ, tal y como se ve en la Ecuación (26). Para realizar esta optimización, uno de los optimizadores más habituales es el algoritmo de descenso del gradiente estocástico (*stochastic gradient descent,* SGD), un proceso iterativo basado en la regla de actualización de los parámetros de la Ecuación (27), donde μ es un hiperparámetro conocido como ratio de aprendizaje que controla la velocidad con la que se actualiza el valor de los parámetros en cada iteración de SGD, y $\nabla_\theta J(\theta)$ es el gradiente de la función de coste J con respecto a los parámetros θ.

$$\theta_* = argmin_\theta J(\theta) \tag{26}$$

$$\theta = \theta - \mu \cdot \nabla_\theta J(\theta) \tag{27}$$

Intuitivamente, SGD calcula $\nabla_\theta J(\theta)$, que representará la dirección y magnitud del máximo crecimiento de J en función de los parámetros, y a continuación actualiza θ en la dirección opuesta, con una magnitud (una fuerza de descenso) que dependerá de μ. A modo de ejemplo, la Figura 5 muestra la aplicación de este algoritmo a la optimización de una función

cuadrática sencilla. Los parámetros θ_1, θ_2 de esta función se han inicializado a un valor arbitrario: $\theta_1 = 10, \theta_2 = 10$, y se ha aplicado el optimizador SGD para encontrar su mínimo. Cada paso del optimizador se ha marcado en rojo sobre la superficie de la función.

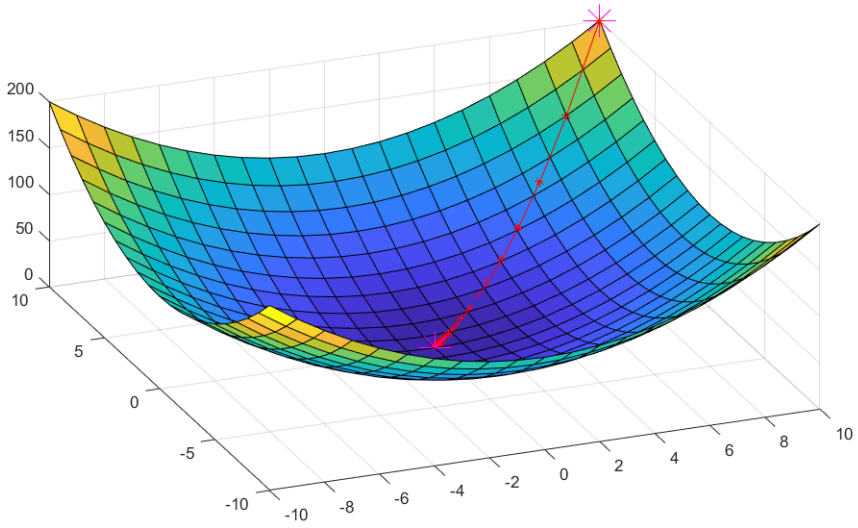

Figura 5: Visualización del algoritmo de descenso del gradiente sobre la función $f(\theta_1, \theta_2) = \theta_1^2 + \theta_2^2$

Dentro del campo del aprendizaje supervisado existen una gran variedad de modelos capaces de cumplir la función de aproximador general paramétrico $f_\theta(x)$. Uno de los paradigmas que más interés ha despertado y que más desarrollo ha tenido en la última década es el conexionista, en el que se incluyen las redes neuronales, especialmente interesantes por su gran escalabilidad con el tamaño de los datos, y su versatilidad.

2.2.2. Redes neuronales prealimentadas

Las redes neuronales prealimentadas, (*feed-forward neural networks*, FFNN) están constituidas esencialmente por una pila de proyecciones lineales seguidas cada una de ellas por una transformación no lineal, que transforman paulatinamente una entrada x hasta convertirla en una salida $\hat{y}$. Se llaman *feed-forward* porque este tipo de arquitectura carece de bucles internos, y solo permite que la propagación de la información sea de atrás hacia delante. Esto es en contraste con otras arquitecturas como las redes recurrentes, o los Transformers, ampliamente utilizados en procesado del lenguaje natural y otras aplicaciones donde la dimensión temporal de la entrada es de gran relevancia, lo cual exige la presencia de estos bucles.

Por tanto, cada capa de una FFNN consta de una matriz de proyección lineal W que gira y escala el vector (z_{-1}) de entrada a dicha capa (proveniente de la capa anterior), un vector b que desplaza este resultado, y una función de activación f que transforma el resultado final de forma -por lo general- no lineal (Ecuación (28)). Obviamente, en la primera capa $x = z_{-1}$, y en la última capa $\hat{y} = z$. En este contexto, una regresión lineal puede entenderse como la FFNN más sencilla posible, donde tan solo existe una capa en la que la función de activación es la identidad.

$$z = f(W \cdot z_{-1} + b) \tag{28}$$

Resulta obvio que apilando capas como la definida en la Ecuación (28), cada vez la complejidad de la red será mayor, y esta será potencialmente más capaz de aprender mapeos $x \to y$ cada más complejos. A parte del número de capas de la red, otro factor importante a considerar es la dimensionalidad de las representaciones intermedias z, o dicho de otro modo, el número de neuronas de cada capa. Es un resultado probado y ampliamente conocido que una red con una única capa oculta (esto es, una única capa entre la primera y la última) es un aproximador universal capaz de aproximar cualquier función bajo un umbral de error determinado (Chong, 2020). Otro factor por considerar es la elección de la función de activación; las funciones de activación más comunes se han representado en la Figura 6 (de izquierda a derecha): tangente hiperbólica (Ecuación (29)), función sigmoide (Ecuación (30)) y ReLU (Ecuación (31)). No obstante, estas no son las únicas: es habitual utilizar la función identidad en la última capa en problemas de regresión, o la función $softmax$ (Ecuación (23)) en problemas clasificación, donde la salida debe ser una distribución multinomial.

$$tanh(u) = \frac{e^{u} - e^{-u}}{e^{u} + e^{-u}} \quad (29) \qquad sigm(u) = \frac{1}{1 + e^{-u}} \quad (30) \qquad ReLU(u) = u \cdot (u > 0) \quad (31)$$

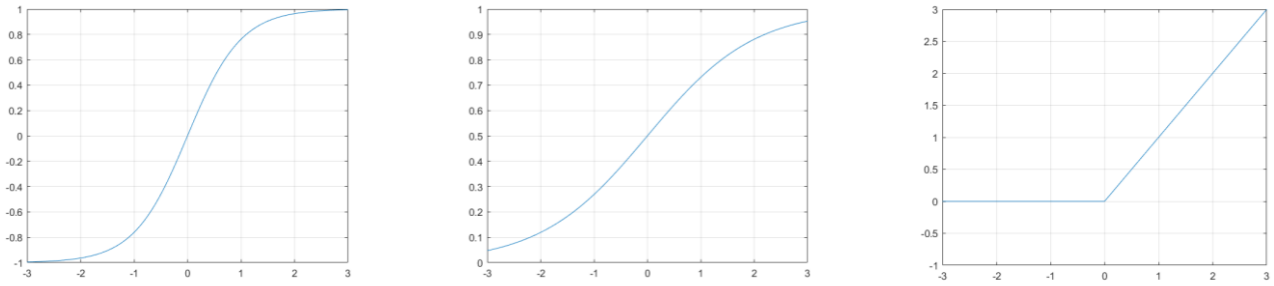

Figura 6: Funciones de activación más comunes (de izquierda a derecha): tangente hiperbólica, función sigmoide, y ReLU

Para entrenar una red neuronal se emplea SGD (Ecuación (27)), por lo que es necesario calcular el gradiente de la función de coste $\nabla_\theta J(\theta)$ respecto a los parámetros de la red $\theta = (W_0, b_0, W_1, b_1, \dots)$. Para ello basta con aplicar la ley de la cadena, o lo que en el contexto de redes neuronales se denomina propagación hacia atrás. La Ecuación (32) muestra una idea de cómo comenzaría este cálculo, si bien en este trabajo se omitirá. A día de hoy, este cómputo resulta a menudo innecesario gracias a técnicas como la Diferenciación automática. Una vez obtenido el gradiente $\nabla_\theta J(\theta)$ para todos los parámetros θ, aplicar SGD es inmediato.

$$\nabla_\theta J(\theta) = \frac{d}{d\theta}\left(\frac{1}{NK} \cdot \sum_{n}^{N}\sum_{k}^{K}\left(f(\theta, x_{n,k}) - y_{n,k}\right)^2\right) \quad (32)$$

2.2.3. Diferenciación automática

Una de las razones por las que el campo de la IA, y en especial el de las redes neuronales, ha avanzado tan enormemente, es gracias al desarrollo de librerías de diferenciación automática, que hacen que rara vez sea necesario calcular a mano las expresiones de propagación hacia atrás. Estas librerías basan su funcionamiento en la construcción de un

grafo con cada una de las operaciones elementales que se van a aplicar, sobre el que a continuación se aplica de forma sistemática la regla de la cadena, logrando así calcular la derivada de cualquier nodo del grafo con respecto a cualquier otro.

Para ilustrar el funcionamiento de este procedimiento, se empleará una función J muy sencilla definida en la Ecuación (33). El primer paso es construir el grafo computacional de J y realizar la propagación hacia delante. Esto se muestra en la Figura 7. Aunque en esta figura se han representado las relaciones z_i de forma simbólica, en la práctica todos los valores de entrada y parámetros tendrían un valor real asociado, de tal forma que tras aplicar la propagación hacia delante todas las variables z_1, z_2, z_3, J tendrían también su valor real asociado.

$$J = (w_1 \cdot x + w_2 \cdot x + y)^2 \tag{33}$$

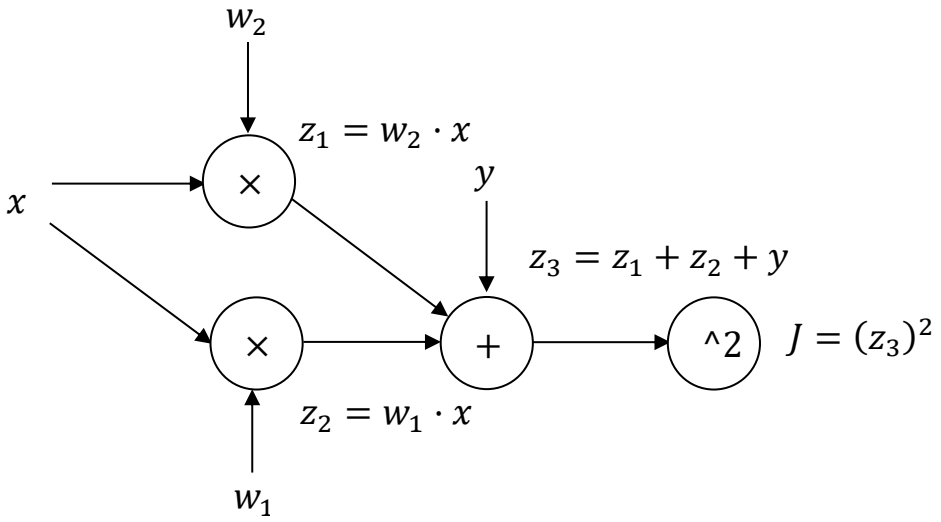

Figura 7: Propagación hacia delante: primer paso del algoritmo de diferenciación automática

Si no se hiciera nada más, se habría obtenido J, y no hubiera sido realmente necesario construir el grafo. Sin embargo, ahora se aplica el segundo paso, conocido como propagación hacia atrás y que permitirá obtener la derivada de cualquier nodo con respecto a cualquier otro.

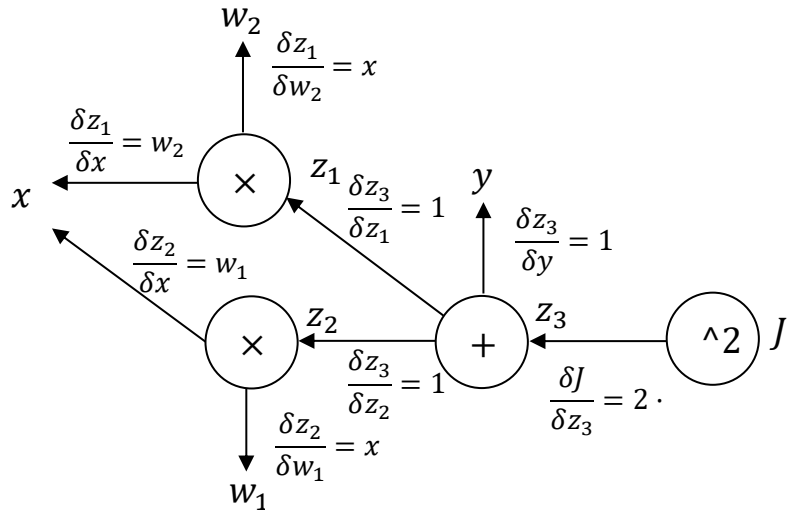

Figura 8: Propagación hacia atrás: segundo paso del algoritmo de diferenciación automática.

Por ejemplo, si se quisiera calcular $\frac{\delta J}{\delta w_2}$ (Ecuación (34)), bastaría con recorrer el grafo en sentido contrario, empezando desde J y acabando en w_2. Las derivadas de las funciones elementales que componen el grafo son triviales y están preprogramadas, de tal forma que es posible ir calculando los valores intermedios $\frac{\delta J}{\delta z_3}, \frac{\delta z_3}{\delta z_1}, \frac{\delta z_1}{\delta w_2}$ a medida que se retrocede. Una vez alcanzado w_2, la derivada buscada ya está calculada.

$$\frac{\delta J}{\delta w_2} = \frac{\delta J}{\delta z_3} \cdot \frac{\delta z_3}{\delta z_1} \cdot \frac{\delta z_1}{\delta w_2} = 2 \cdot 1 \cdot x \tag{34}$$

Un segundo ejemplo ligeramente más complejo sería el cálculo de $\frac{\delta J}{\delta x}$. En este caso hay que tener en cuenta las derivadas de todos los caminos que conducen a x (en el caso de w_2 solo había uno) y sumarlas para obtener el resultado buscado. Para una introducción más completa, se puede consultar (Güne¸ et al., 2018).

$$\frac{\delta J}{\delta x} = \frac{\delta J}{\delta z_3}\left(\frac{\delta z_3}{\delta z_1} \cdot \frac{\delta z_1}{\delta x} + \frac{\delta z_3}{\delta z_2} \cdot \frac{\delta z_2}{\delta x}\right) = 2 \cdot (1 \cdot w_1 + 1 \cdot w_2) \tag{35}$$

El algoritmo de diferenciación automática es eficiente (tiene un coste temporal similar al de evaluar la función original, si bien el coste en espacio es mayor), exacto (hasta el límite de representación numérica escogido, ya que se utilizan las expresiones exactas de las derivadas) y, sobre todo, extremadamente cómodo para el ingeniero, que solo tiene que preocuparse por definir las arquitecturas neuronales hacia delante, y puede dejar que el algoritmo se ocupe de los cálculos del gradiente, independientemente de su complejidad. Además, no debe confundirse este algoritmo con los de diferenciación numérica (que aproximan las derivadas por diferencias finitas, con la penalización en precisión y eficiencia que esto acarrea), o con la diferenciación simbólica (que puede dar lugar a código ineficiente, y puede fallar a la hora de manejar funciones muy complejas).

En la práctica, existen a día de hoy multitud de librerías como Tensorflow (Abadi et al., 2016) o Pytorch (Paszke et al., 2019) que implementan esta funcionalidad y la hacen muy fácilmente accesible al desarrollador. En este trabajo se utilizará esta última. El Código 1 muestra cómo podríamos obtener $\frac{\delta J}{\delta w_2}$ y $\frac{\delta J}{\delta x}$ utilizando Pytorch:

```
import torch
from torch.autograd import Variable

#x e y son valores conocidos
x= Variable(torch.tensor([5.]), requires_grad=True)
y= Variable(torch.tensor([10.]), requires_grad=True)

#w1 y w2 son los parámetros del modelo
w1 = Variable(torch.randn((1,)), requires_grad=True)
w2 = Variable(torch.randn((1,)), requires_grad=True)

#J es la función de coste
J= (w1*x + w2*x + y)**2

#Calculamos los gradientes de todos los elementos anteriores respecto a J
J.backward()

#Imprimimos dJ/dw2 y dJ/dx
print(w2.grad, x.grad)
```

```
#Esto muestra por pantalla:
#tensor([26.9381]) tensor([-7.8726])
```

Código 1: Ejemplo de diferenciación automática con Pytorch

Además, con unas pocas líneas más de código resulta trivial realizar una optimización de J con respecto a w_1 y w_2 (Código 2). En este caso la función es extremadamente fácil de optimizar ya que bastará que $(w_1 + w_2) \cdot x = -y$. Si $x = 5, y = 10$, entonces: $w_1 + w_2 = -2$, lo cual se verifica con los parámetros encontrados por SGD: $w_1 = -1.9420, w_2 = -0.0580$.

```
for i in range(3):
    #Propagación hacia delante
    J= (w1*x + w2*x + y)**2
    print('%d: %.2f'%(i, J))

    #Propagación hacia atrás
    J.backward()

    #Un paso de descenso del gradiente
    with torch.no_grad():
        w1 -= 0.005 * w1.grad
        w2 -= 0.005 * w2.grad
        w1.grad, w2.grad= None, None

print(w1, w2)

#Esto muestra por pantalla:
#0: 33.97
#1: 0.00
#2: 0.00
#tensor([-1.9420], requires_grad=True) tensor([-0.0580], requires_grad=True)
```

Código 2: Ejemplo de optimización por descenso del gradiente en Pytorch

2.2.4. Otros aspectos prácticos

Aunque no es estrictamente necesario, en la práctica suele resultar muy útil estandarizar las entradas a las redes neuronales. Como se observa en la Ecuación (36), dada una variable x, la variable estandarizada $\tilde{x}$ resulta de centrar la distribución de x en cero restándole la media μ_x, y escalarla en torno a la unidad dividiéndola por su desviación estándar σ_x. De esta forma, se normaliza la importancia relativa de las diferentes variables, eliminando posibles sesgos a priori. Además, las funciones de activación típicamente empleadas (Figura 6) tienen la mayor riqueza en torno a valores cercanos al cero, de tal forma que es deseable mantener los valores de las activaciones de la red en esta zona.

$$\tilde{x} = \frac{x - \mu_x}{\sigma_x} \tag{36}$$

Existe una gran cantidad variaciones dentro del campo de los algoritmos por descenso del gradiente. En una primera instancia, podríamos distinguir entre: descenso estocástico, por lotes (o completo), o por mini-lotes. El descenso estocástico es quizás el más sencillo, y se caracteriza por emplear una única tupla $< x_i, y_i >$ para calcular la actualización de la Ecuación (27); SGD completo, en cambio, utiliza todo el conjunto de datos, es decir, todas las tuplas $< x_1, y_1 >, \ldots, < x_N, y_N >$ de entrenamiento cada vez que realiza un paso de actualización; por último, SGD por mini-lotes emplea solo un subconjunto de $M < N$ tuplas

del conjunto de entrenamiento. La Figura 9 muestra una comparativa en la evolución de estas tres versiones de SGD en un problema de optimización sencillo.

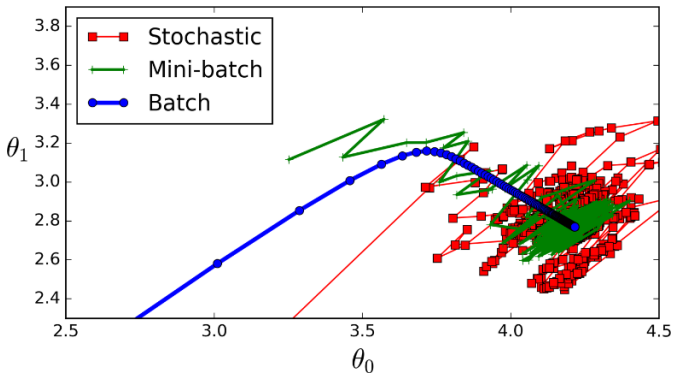

Figura 9: Diferentes variantes de SGD aplicadas a una optimización sencilla.
Fuente: Suryansh S., https://hackernoon.com/gradient-descent-aynk-7cbe95a778da

Como se puede ver, cuando se utiliza el conjunto de entrenamiento completo (*batch* SGD), la actualización presenta mucho menos ruido que cuando se utiliza una sola muestra (*stochastic* SGD), o cuando se utiliza un mini-lote (*mini-batch* SGD). No obstante, en la práctica no siempre es posible utilizar el conjunto completo (por problemas de memoria, por ejemplo), ni tampoco es siempre deseable, pues se observa empíricamente que, en superficies de coste no convexas (en redes neuronales casi nunca son convexas), el ruido aportado por el SGD por mini-lotes ayuda a superar los mínimos locales, mejorando así la eficacia del algoritmo.

Una adición habitual a SGD es un término de inercia, que reduce el ruido y acelera la convergencia, pues permite mantener la inercia del descenso a pesar de que un mini-lote especialmente ruidoso trate de apartar al optimizador del camino hacia el mínimo. Las Ecuaciones (37) y (38) muestran las reglas de actualización para el SGD con inercia, donde γ (normalmente $\gamma = 0.9$) es un parámetro que controla la intensidad de la inercia.

$$\delta\theta_i = \gamma \cdot \delta\theta_{i-1} + \mu \cdot \nabla_\theta J(\theta) \tag{37}$$

$$\theta_i = \theta_{i-1} - \delta\theta_i \tag{38}$$

Otra modificación habitual consiste en variar la ratio de aprendizaje μ conforme avanza el entrenamiento, bien de forma manual siguiendo algún tipo de programa predeterminado (típicamente, reduciéndolo con el tiempo para realizar cada vez actualizaciones más pequeñas que favorezcan la convergencia), o bien siguiendo algún criterio automático (como reducirlo cuando el aprendizaje parece estancarse). También hay optimizadores que incluyen internamente algún tipo de criterio, como *RMSProp*, que calcula una ratio de aprendizaje que es inversamente proporcional a la raíz cuadrada de la media (calculada en decaimiento exponencial) del cuadrado de los gradientes. De esta forma, parámetros que hayan experimentado cambios bruscos (grandes gradientes) recientemente, tenderán a tener un μ más bajo, y el foco de la actualización se centrará en cambio en aquellos parámetros que se hayan actualizado menos recientemente. Las reglas de actualización de *RMSProp* se encuentran en las Ecuaciones (39) y (40), donde ν (habitualmente $\nu = 0.9$) es un parámetro que controla el tiempo del decaimiento exponencial.

$$c_i = \nu \cdot c_{i-1} + (1 - \nu) \cdot \left(\nabla_\theta J(\theta)\right)^2 \tag{39}$$

$$\theta_i = \theta_{i-1} - \frac{\mu}{\sqrt{c_t + \epsilon}} \cdot \nabla_\theta J(\theta) \tag{40}$$

Con respecto a esto último, cabe hacer un pequeño inciso sobre el filtro de decaimiento exponencial en el contexto del campo del aprendizaje automático, ya que este se utiliza con mucha frecuencia para obtener la media móvil de una variable. La Ecuación (41) define, para una variable x, su versión filtrada $\bar{x}$, como una combinación convexa del valor anterior de esta, $\bar{x}_{-1}$, y el valor de una nueva muestra x. De esta forma, valores anteriores de x tienen sucesivamente menos ponderación sobre el valor actual de $\bar{x}$. En concreto, tras t actualizaciones, la influencia de x_{-t} será α^t; por tanto, $\alpha = 0.01^{1/t}$ nos proporciona una forma sencilla de calcular el valor de α para que la influencia de x_{-t} sea del 1% dado un número de pasos de tiempo t, lo cual puede resultar útil para el diseño de este tipo de filtros.

$$\bar{x} = \alpha \cdot \bar{x}_{-1} + (1 - \alpha) \cdot x \tag{41}$$

Una de las variantes de SGD más empleada, y la que se usa en este trabajo, es Adam (*Adaptive Moment Estimation*) (Kingma & Ba, 2015). Este optimizador básicamente combina la inercia con el criterio de actualización del gradiente de *RMSProp*, obteniendo así las Ecuaciones (42), (43) y (45). Además, dado que m_0 y c_0 se inicializan a 0, es necesario realizar una corrección del sesgo de m_i y c_i durante los primeros pasos del Adam, lo cual se logra con la Ecuación (44), que da mayor peso a los nuevos valores durante las primeras iteraciones, y luego, a medida que i crece, $\gamma^i \to 0$ y $\hat{m}_i \to m_i$ (y, por tanto, $\hat{c}_i \to c_i$).

$$m_i = \gamma \cdot m_{i-1} + (1 - \gamma) \cdot \nabla_\theta J(\theta) \tag{42}$$

$$c_i = \nu \cdot c_{i-1} + (1 - \nu) \cdot \left(\nabla_\theta J(\theta)\right)^2 \tag{43}$$

$$\hat{m}_i = \frac{m_i}{1 - \gamma^i}; \quad \hat{c}_i = \frac{c_i}{1 - \nu^i} \tag{44}$$

$$\theta_i = \theta_{i-1} - \frac{\mu}{\sqrt{\hat{c}_i + \epsilon}} \cdot \hat{m}_i \tag{45}$$

Por último, queda por definir el concepto de la optimización de hiperparámetros. Los hiperparámetros son cualquier decisión de diseño que puede afectar al rendimiento del modelo, tal como puede ser el número de capas, el número de neuronas en cada capa, el tamaño del mini-lote de SGD, o la ratio de aprendizaje μ. En aprendizaje supervisado es habitual reservar una parte de los datos para entrenamiento, otra para validación y otra para test. En este contexto, los parámetros de la red se optimizan sobre el conjunto de entrenamiento usando SGD, los hiperparámetros se optimizan con algún optimizador de caja negra o búsqueda exhaustiva sobre el conjunto de validación, y el conjunto de test se mantiene a parte durante todo este proceso, ya que servirá para comprobar el rendimiento final del algoritmo sobre muestras que nunca antes había "visto". Para la aplicación que aquí se desarrolla no tiene sentido esta separación en subconjuntos, puesto que la semántica de los datos de entrenamiento es diferente a la de un problema supervisado típico, ya que la propia exploración del entorno es entrenamiento, validación y test. No obstante, los agentes sí ser testearán exponiéndolos a entornos con unos patrones meteorológicos cambiantes, que sean diferentes a aquellos con los que han sido validados.

2.3. Descripción de BSM1 y su coste de operación

2.3.1. BSM1

El BSM1 (Alex et al., 2008) es un modelo de una EDAR de referencia incluyendo las ecuaciones que definen todos los subsistemas que la componen, la distribución de estos, datos con las características de los afluentes a la planta, procedimientos de test y evaluación, así como implementaciones de referencia en Matlab-Simulink y FORTRAN. Para este trabajo se utilizará una implementación realizada en Modelica por los autores de varios artículos similares a este trabajo (Hernández-del-Olmo et al., 2012, 2016, 2018, 2019). Sin entrar en detalle (pues se reserva para el capítulo Trabajo realizado), mencionar que Modelica es un lenguaje no propietario, orientado a objetos y basado en ecuaciones que permite modelar sistemas físicos complejos de todo tipo.

La distribución de la planta se muestra en la Figura 10. El objetivo de esta es la eliminación de nitrógeno y carbono del agua residual a través de cinco reactores de fangos activos seguidos de un tanque de sedimentación. Los dos primeros reactores son anóxicos (1000 m^3 cada uno), mientras que los tres siguientes están aireados (1333 m^3 cada uno). Los procesos biológicos que tienen lugar en los reactores se modelan de acuerdo al Modelo de Fangos Activos No. 1 (*Activated Sludge Model No.1*, ASM1) (Henze et al., 2015), con los parámetros biológicos ajustados a una temperatura de 15ºC. Por otra parte, para el sedimentador se ha tomado una unidad lineal no reactiva (donde no tienen lugar reacciones biológicas) de 10 capas de grosor constante y un modelo de velocidad de sedimentación exponencial doble (Takács et al., 1991); este tiene un área de 1500 m^2 y una profundidad de 4 m. Además, la planta cuenta con dos bucles de reciclaje que toman parte de los residuos generados en el último reactor y en el sedimentador y los recirculan hasta la entrada con el objetivo de minimizar el volumen de residuos generados, a costa de un sobredimensionado de la instalación.

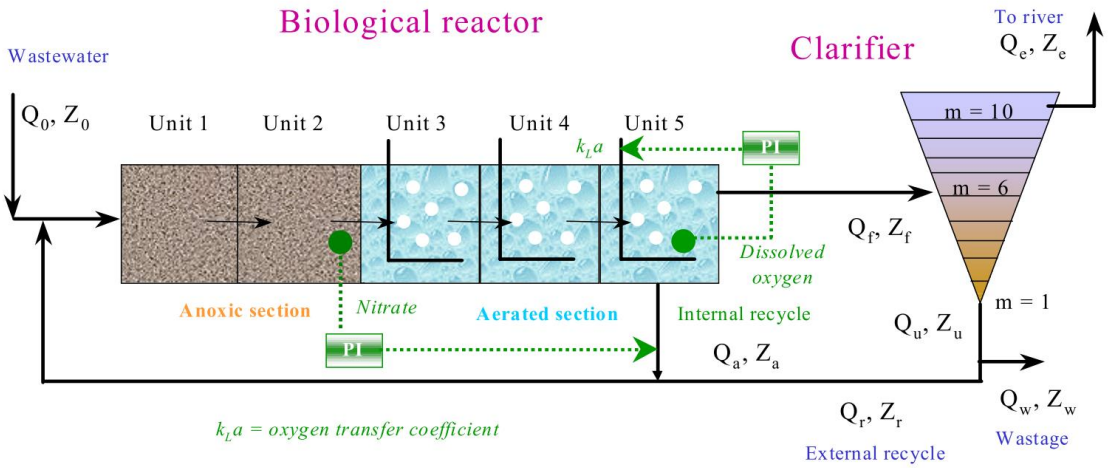

Figura 10: Vista general de la planta BSM1, tomada de (Alex et al., 2008)

Por último, la planta cuenta con dos controladores PI. El primero controla la concentración de $NO_3 - N$ en el segundo reactor mediante la manipulación del caudal de reciclaje Q_a, mientras que el segundo se encarga de controlar el oxígeno disuelto (DO) en el último reactor manipulando el coeficiente de transferencia de oxígeno ($K_L a_5$) a través de la bomba

de aireación presente en este tanque. En este contexto, el objetivo del proyecto será el diseño de un agente que establezca de forma automática el punto de funcionamiento de este último PI, de tal forma que el coste de operación sea mínimo. De forma intuitiva, aumentar la cantidad de DO mejorará la eliminación de las partículas en suspensión del agua residual, lo cual dará lugar a un agua de salida más pura y una reducción de las penalizaciones; no obstante, esto será a costa de un mayor consumo eléctrico, con su coste asociado. Por tanto, el objetivo del agente será encontrar el equilibrio entre estos factores en cada estado de funcionamiento de la planta, permitiendo así obtener un ahorro económico. Aunque en el modelo se podría medir cualquier variable de estado, se considerará que existen sensores (a los cuales el agente tiene acceso) para medir el NH_4 y el O_2 del tanque 5. Adicionalmente, se asume también el conocimiento de la concentración de sólidos en suspensión del caudal de desecho, ya que esta variable es necesaria para poder calcular los costes (ver Coste de operación).

Como datos de entrada a la planta (caudal, composición, sólidos en suspensión, etc.) se utilizarán los datos de (Vanhooren et al., 1996), que se encuentran disponibles abiertamente para su descarga en http://www.benchmarkWWTP.org en forma de tres archivos de texto separados por comas (`Inf_dry_2006.txt`, `Inf_str_2006.txt`, `Inf_rain_2006.txt`), conteniendo tablas de datos de entrada muestreadas cada 15 min durante 14 días para tiempo seco, tormentoso y lluvioso, respectivamente.

2.3.2. Coste de operación

El coste de operación (OC) de la planta será la métrica empleada para valorar el coste del funcionamiento de la planta, y poder así comparar diferentes regímenes de funcionamiento. Además, el coste de operación diario instantáneo multiplicado por -1 será la recompensa que reciba el agente de RL durante su entrenamiento, de tal forma que pueda aprender las estrategias de control que maximicen este índice (es decir que minimicen OC). El OC se define en la Ecuación (46), (Hernández-del-Olmo et al., 2016), donde AE es la energía de aireación, ME es la energía de mezclado, PE es la energía de bombeo (todas ellas en KWh), SP es la producción de fangos a eliminar (Kg), EF (€) son las multas por superar determinados niveles de partículas en el agua efluente, $\gamma_1 = 0.1€/Kwh$ y $\gamma_2 = 0.5\ €/Kg$.

$$OC(t) = \gamma_1\big(AE(t) + ME(t) + PE(t)\big) + \gamma_2 SP(t) + EF(t) \tag{46}$$

Siguiendo (Bagheri et al., 2015), el resto de componentes de OC están definidos en las Ecuaciones (47)-(50). Para el cálculo de la energía de aireación, AE, S_0^{sat} es la concentración de oxígeno, que será proporcional a la suma de los volúmenes de cada reactor aireado (V_3, V_4, V_5) por el coeficiente volumétrico de transferencia de oxígeno $K_L a(t)$. La energía de mezclado, ME, es proporcional al volumen de los reactores de mezclado (V_1, V_2). La energía de bombeo, PE, es la suma de los caudales de reciclaje interno Q_{in}, externo Q_{ext}, y el caudal de desecho Q_w, multiplicados por un factor de consumo. Y, por último, la producción de fangos (SP), será del factor de sólidos en suspensión del agua de salida TSS_w por el caudal de salida Q_w.

$$AE(t) = \frac{S_0^{sat}}{1.8 \cdot 1000} \sum_{i=3,4,5} V_i \cdot K_L a(t) \tag{47}$$

(48)

$$ME(t) = 24 \cdot 0.005 \sum_{i=1,2} V_i$$

$$PE(t) = 0.004Q_{in}(t) + 0.008Q_{ext}(t) + 0.05Q_w(t) \tag{49}$$

$$SP(t) = TSS_w \cdot Q_w(t) \tag{50}$$

Finalmente, las multas por efluente (EF) se calculan teniendo en cuenta la cantidad de amoníaco ($S_{NH,eff}$) y nitrógeno ($S_{TN,eff}$) del agua depurada que abandona la EDAR según la Ecuación (51), donde $S_{NH,lim} = 4mg/L$ y $S_{TN,lim}mg/L$ son los límites de emisión de cada una de estas sustancias. Si estos límites son superados, el coste de emisión de la cantidad que lo supera es algo mayor, y además se aplica una multa fija.

$$EF = Q_{eff} \cdot \left(\begin{cases} 4 \cdot S_{NH,eff} & si\ S_{NH,eff} \leq S_{NH,lim} \\ 4 \cdot S_{NH,lim} + \frac{2.7}{1000}\frac{€}{m^3} + 12 \cdot \left(S_{NH,eff} - S_{NH,lim}\right) & si\ S_{NH,eff} > S_{NH,lim} \end{cases} + \right. \tag{51}$$
$$\left. \begin{cases} 2.7 \cdot S_{TN,eff} & si\ S_{TN,eff} \leq S_{TN,lim} \\ 2.7 \cdot S_{TN,lim} + \frac{1.4}{1000}\frac{€}{m^3} + 8.1 \cdot \left(S_{TN,eff} - S_{TN,lim}\right) & si\ S_{TN,eff} > S_{TN,lim} \end{cases} \right)$$

Con esto, queda pues definida la planta BSM1, así como el coste de operación de esta, que será la métrica que el agente trate de optimizar.

3. TRABAJO REALIZADO

En este capítulo se describirán paso a paso los aspectos más destacables del trabajo llevado a cabo, comenzando por la Preparación de BSM1 en Open Modelica y la Exportación de BSM1 al formato FMU usando OMPython, lo cual permite su portabilidad y elimina la dependencia del modelo con el propio lenguaje Modelica en el que fue definido. A continuación, se procederá a la Simulación de FMU desde Python con pyFMI, y la Creación de un entorno de Open AI Gym de BSM1, siendo Open AI Gym el estándar actual para realizar experimentos de RL. Por último, se describirá el Diseño de los agentes de aprendizaje por refuerzo que interactuarán con el entorno y el Entrenamiento de los agentes.

Para ello, se hará uso de los algoritmos y consideraciones teóricas introducidas en el capítulo Materiales y métodos, pero adaptando estos a las características concretas del trabajo que se ha realizado. Finalmente, en el capítulo de Resultados y discusión, se presentarán los resultados obtenidos.

3.1. Preparación de BSM1 en Open Modelica

Como se ha introducido ya en el apartado de Descripción de BSM1 y su coste de operación, la planta BSM1 está implementada en Modelica, que es un lenguaje de modelado de sistemas físicos. Más concretamente, esta se implementó utilizando Open Modelica (OM), un entorno de código abierto para el modelado, compilación y simulación de código Modelica para su uso en investigación, docencia e industria (Fritzson et al., 2020). Una de las herramientas clave de OM es su editor, OMEdit, que combina la edición y diseño del modelo mediante la definición e interconexión de bloques, con un editor de texto que permite programar en Modelica el funcionamiento de cada uno de los bloques.

El primer paso, por tanto, será tomar el BSM1 implementado en Modelica y editarlo con el objetivo de poder realizar su simulación desde Python. Una de las mayores ventajas de esta aproximación es que permite desacoplar el modelo en sí (sus bloques constitutivos, conexiones, ecuaciones diferenciales, etc.) de la parte del manejo de los datos entrada, salida y control, así como de la implementación, configuración y entrenamiento de los agentes de RL. En cambio, en la implementación de BSM1 utilizada por ejemplo en (Hernández-del-Olmo et al., 2016), y en la que se basa este trabajo, todo se realiza directamente en Modelica, incluyendo la lectura de los datos de entrada según patrón meteorológico, el cálculo de la función de coste, y la definición y entrenamiento del agente; pero en realidad Modelica no es la herramienta ideal para este tipo de tareas, y está muy limitado a la hora de implementar agentes y estrategias de control complejas. Por ejemplo, hubiera resultado casi imposible implementar los agentes basados en redes neuronales que se utilizan en este trabajo (y gestionar su entrenamiento) con esta aproximación.

Por tanto, lo primero que se hizo es eliminar de BSM1 todos los componentes que no pertenecen propiamente al modelo en sí de la EDAR, y dejar todas las entradas de datos definidas como tal con la idea de aportar estos datos desde Python externamente durante la simulación. Esto dio lugar a la planta cuyo esquema se muestra en la Figura 11. En este esquema hay dos entradas que se han dejado sin conectar por un extremo y que representan las entradas del modelo. En el extremo superior izquierdo, la primera entrada se conecta a un *WWSource* (en la Figura 11 solo se puede leer la etiqueta "WW…") que es una fuente de aguas residuales, y que a cada instante de tiempo de simulación debe recibir las características del agua residual de entrada a la planta, las cuales provienen de los archivos `Inf_dry_2006.txt`, `Inf_str_2006.txt`, `Inf_rain_2006.txt`, discutidos en Descripción de BSM1 y su coste de operación. En el apartado Simulación de FMU desde Python con pyFMI se discutirá cómo se utilizan estos datos para generar un año de datos de entrada para la simulación. La segunda entrada se encuentra hacia la derecha de la Figura 11, y está conectada al punto de consigna del controlador PI que controla la aireación del tanque 5. Esta entrada será precisamente sobre la que actúe el agente de aprendizaje por refuerzo.

Aunque no es obvio observando la Figura 11, también se eliminó del modelo toda la lógica asociada al cálculo del coste de operación, OC, ya que se consideró que estos aspectos no debían formar parte semántica del modelo BSM1. Además, implementarlo en Python facilita mucho su manejo y abre la puerta a realizar modificaciones de este de forma sencilla (por ejemplo, si se considerara un aumento del coste de la energía eléctrica, o si se quiere implementar una tarifa con discriminación horaria).

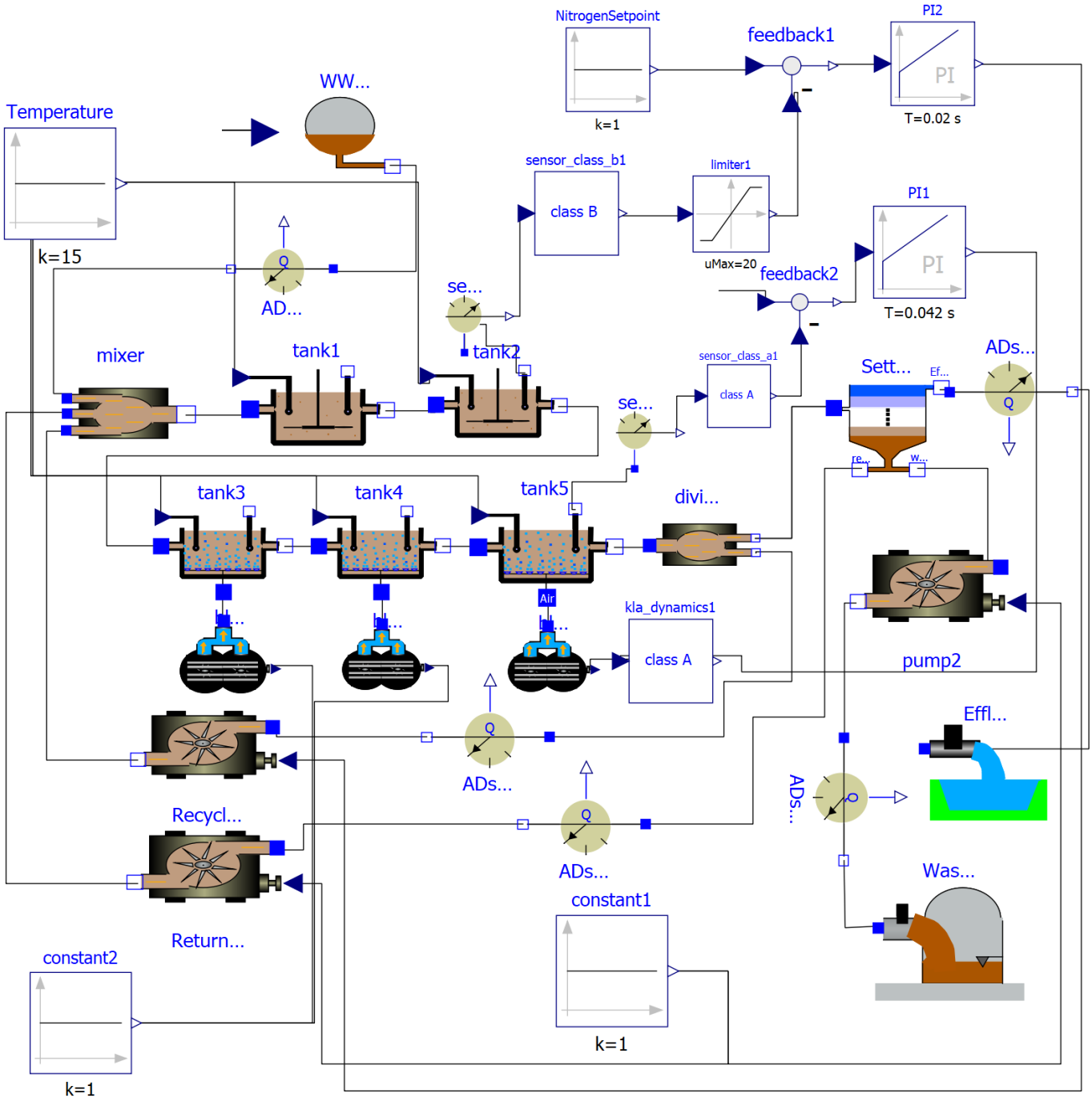

Figura 11: Captura del esquema de BSM1en OMEdit

3.2. Exportación de BSM1 al formato FMI usando OMPython

Una vez definido el modelo en Modelica, el siguiente paso será exportarlo de tal forma que pueda ser simulado desde programas externos a OMEdit, e incluso desde lenguajes diferentes a Modelica. A este fin, existe un estándar llamado *Functional Mockup Interface*, (FMI), que define un contenedor y una interfaz para el intercambio de modelos dinámicos, usando una combinación de archivos XML, binarios y código C comprimidos en un único archivo FMU (Functional Mockup Interface, 2018).

Aunque OpenModelica soporta la generación de FMUs, el entorno debe ser adecuadamente configurado según las características del modelo, la versión de Modelica empleada en su definición, las librerías que son necesarias para su funcionamiento, etc. Por tanto, se consideró interesante automatizar este proceso también desde Python, haciendo uso de la librería OMPython, que es una interfaz entre este lenguaje de programación y OpenModelica. A este fin, se desarrolló una sencilla clase, `Runner`, dentro del módulo `utils.py`, que abre una sesión de comunicación con OpenModelica utilizando OMPython, y permite fácilmente enviar comandos y recibir respuesta. Utilizando esta clase, la exportación del modelo a FMU se puede hacer directamente de Python con el Código 3.

```
base_path= 'WasteWaterResearch'
model_name= 'BSM1'

from utils import Runner
R= Runner()
R.run('loadModel(Modelica,{"3.2.3"},true,"",false)')
R.run('loadFile("%s/WasteWater/package.mo","UTF-8",true,true,false)'%base_path)
R.run('loadFile("%s/BSM1/package.mo","UTF-8",true,true,false)'%base_path)
R.run('disableNewInstantiation()')
R.run('translateModelFMU(%s, version="2.0", fmuType="me")'%model_name)
```

Código 3: Código para la exportación del FMU de BSM1

3.3. Simulación de FMU desde Python con PyFMI

El siguiente paso consistirá en lograr simular el FMU anterior desde Python, utilizando para ello la librería de Python PyFMI, diseñada a este fin. El Código 4 muestra un ejemplo de cómo podría llevarse a cabo esta simulación. En primer lugar, se carga el modelo pasándole a `load_fmu` el nombre del archivo FMU generado en el paso anterior. A continuación, se obtiene el diccionario de opciones de simulación, y se modifican algunos parámetros; a modo de ejemplo, en este código se ha indicado al simulador que cuando use CVode como *solver* (CVode se usa por defecto en PyFMI), que emplee un *solver* de ecuaciones lineales denso, en lugar de uno disperso, ya que de caso contrario la versión actual de la librería produce un fallo y no es posible realizar la simulación.

```
from pyfmi import load_fmu
import numpy as np
model = load_fmu('%s.fmu'%model_name)
opts = model.simulate_options()
opts["CVode_options"]["linear_solver"] = "DENSE"
input= (['agent_action'], np.array([[0, 1.2]]))
res = model.simulate(start_time=0., final_time=1., options=opts, input=input)
print(res.keys(), res['time'])
```

Código 4: Ejemplo de simulación del FMU de BSM1 desde Python usando PyFMI

El siguiente paso es definir las entradas, que en este caso será una única acción constante de valor 1.2 aplicada desde el instante 0 en la entrada del modelo con nombre `agent_action`. Esta entrada será precisamente la acción del agente, que se corresponde con la consigna del controlador PI del tanque 5 de BSM1. Llamar al método `simulate` del modelo inicia la simulación, que debido a los argumentos especificados se produce entre el día 0 y el día 1, generando de salida un objeto de resultados `res`, el cual contiene los valores que han

experimentado todas las variables del modelo (de la EDAR) durante el tiempo de simulación, pudiendo acceder a ellas a través de una interfaz similar a la de un diccionario Python.

De esta forma, se puede ya vislumbrar cómo se podrá adaptar este entorno a la interfaz de OpenAI Gym para realizar experimentos de aprendizaje por refuerzo: la idea será realizar la simulación de la planta aplicando una acción constante del agente durante un paso de tiempo (15 min), obtener el estado de la planta y el coste de operación instantáneo tras esos 15 min leyendo la estructura de resultados que devuelve `model.simulate()`, y utilizar toda esta información para entrenar al agente, calcular la siguiente acción, y repetir el proceso para el siguiente paso de simulación. Esta simulación paso a paso será posible gracias a que el FMU, a menos que se reinicie explícitamente, almacena por defecto el estado completo del modelo entre simulaciones.

Se debe notar que el código anterior es solo de ejemplo y no funcionaría correctamente con el modelo de la Figura 11, ya que habría que definir los `inputs` para todas las entradas del modelo: no solo la acción del agente, sino también las características y caudal del agua residual de entrada a la EDAR. No obstante, este detalle se ha omitido aquí por claridad.

Por último, cabe hablar de la clase `TrajectoryInterpolation`, que se encuentra en el módulo `utils.py`, y que hereda de la clase `pyfmi.common.core.Trajectory`, la cual es utilizada internamente por PyFMI para interpolar los `inputs` del modelo de tal forma que puedan ser evaluados en cualquier instante de tiempo arbitrario t durante la simulación. Pues bien, `TrajectoryInterpolation` es una implementación de esta clase que ha sido optimizada para lograr mayor velocidad en el contexto de este problema. Llamando a la función `patch_pyfmi_interpolator` de `utils.py`, se configura la librería PyFMI para utilizar la clase `TrajectoryInterpolation` en lugar de la que emplea por defecto, y se logra así una mejora en la velocidad de simulación de BSM1 de casi un 30% sin sacrificar precisión y sin ningún otro perjuicio. De forma muy resumida, se observó que una gran parte del tiempo de CPU de PyFMI se empleaba en interpolar las entradas, así que se analizó esta región de código crítica, y se propuso una forma de interpolación alternativa utilizando ajuste por *splines* lineales del vector de las entradas, en lugar de la técnica que empleaba por defecto en la librería, y que implicaba calcular los parámetros de la interpolación para cada variable por separado cada vez que se evaluaba la entrada en un instante t.

3.4. Creación de un entorno de OpenAI Gym de BSM1

OpenAI Gym se autodefine como un conjunto de herramientas para la investigación en aprendizaje por refuerzo, incluyendo una creciente colección de problemas de referencia con una interfaz común (entornos) que permiten el desarrollo y la comparativa de algoritmos de aprendizaje por refuerzo (Brockman et al., 2016). En la práctica, es la librería por excelencia en lo que RL se refiere, y es por tanto extremadamente interesante portar a su interfaz cualquier modelo susceptible de ser utilizado junto con un agente de RL.

Para ello, basta con conocer las interfaces que la librería proporciona e implementarlas con el modelo del entorno que se desea simular, lo cual se ha llevado a cabo en el módulo `BSM1Envs.bsm1_env.py`, cuyo diagrama de clases se muestra en la Figura 12. Como se puede ver, hay tres clases principales, de las cuales `gym.core.Env` es la interfaz proporcionada por OpenAI Gym (aunque no está definida como tal, ya que en Python no existe explícitamente el concepto de interfaz), y `bsm1_env.ModelicaEnv` es una clase abstracta desarrollada como

parte de este trabajo, y que implementa `gym.core.Env`, aglomerando toda la lógica requerida para simular FMUs de Modelica desde Python. De esta forma, dado cualquier modelo del tipo que sea (eléctrico, mecánico, químico, etc.) implementado en Modelica y exportado a FMU, el programador tan solo tendrá que implementar los métodos abstractos de esta clase en su clase particular (y potencialmente reescribir algún otro si la lógica por defecto no es la adecuada), obteniendo así en muy poco tiempo en entorno (un *Environment*) de su modelo. Esto es precisamente lo que se hace la clase `bsm1_env.BSM1Env`, la cual además es relativamente corta.

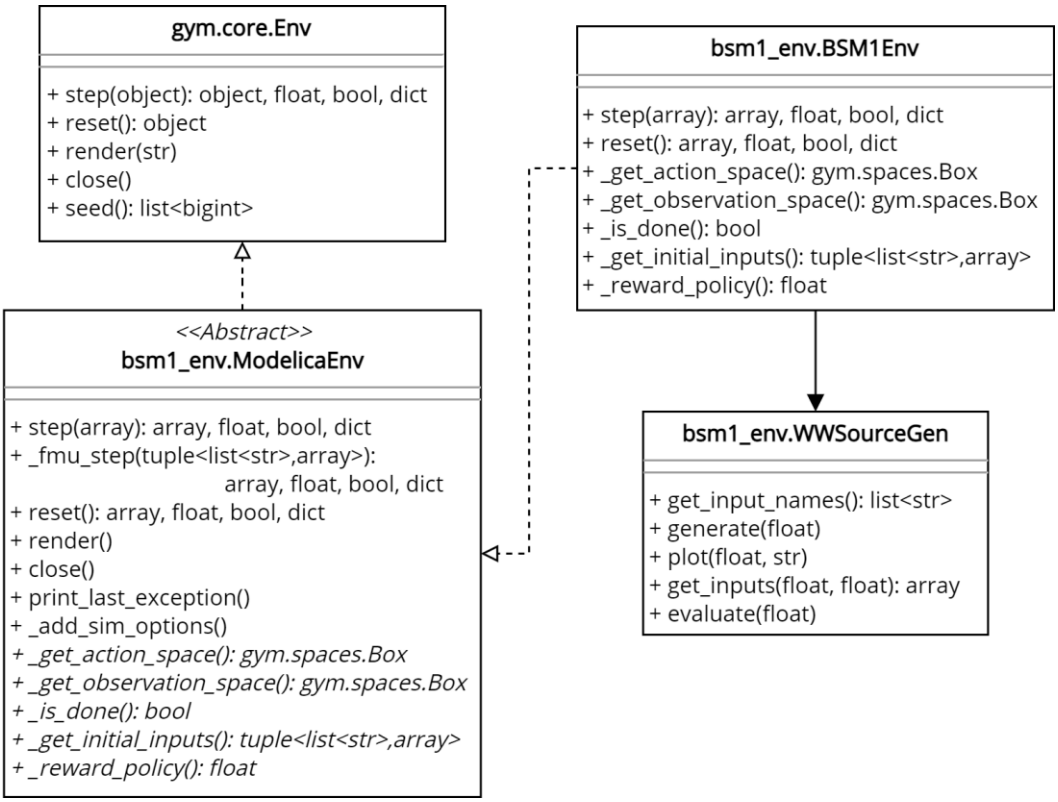

Figura 12: Diagrama de clases del módulo BSM1Envs.bsm1_env. Los atributos se han omitido.

De forma muy general, el método `reset()` se encargará de dejar el entorno completamente preparado para comenzar a interaccionar con un agente. Este agente llamará al método `step()`, que acepta como parámetro una acción, y avanzará la simulación un paso de tiempo (15 min en este caso). Cuando se alcance el tiempo final establecido para la simulación (aproximadamente un año, sin contar el tiempo de inicialización), entonces el entorno devolverá un indicador booleano de que la simulación ha terminado, y a partir de ese momento será necesario llamar al método `reset()` si se quiere seguir interactuando con él. Internamente, estos dos métodos emplean una lógica similar a la utilizada en el Código 4 para realizar una simulación parcial de la planta e interaccionar con ella. El resto de los métodos no son especialmente relevantes, sobre todo si se tiene en cuenta que los métodos con una barra baja "_" en Python por convención son métodos para uso interno de la clase. Es importante notar que esos no son métodos privados, ya que Python no hace distinción fuerte entre diferentes formas de visibilidad. Simplemente sirven para indicar al usuario final de la clase que no debería tener que acceder a estos.

Por último, la clase `bsm1_env.WWSourceGen` implementa la generación de las características del agua residual que entra a la planta en cada instante de tiempo. Para ello, como se adelantaba en Descripción de BSM1 y su coste de operación, se hace uso de 4 archivos de datos (`Inf_dry_2006.txt`, `Inf_str_2006.txt`, `Inf_rain_2006.txt`, `Inf_steady_2006.txt`), donde los tres primeros contienen datos de 14 días de tiempo seco, tormentoso y lluvioso (respectivamente) muestreado cada 15min, y el último se ha diseñado con los valores medios de las entradas y contiene un único vector de valores de entrada que son constantes en todo momento. Estos valores constantes se utilizarán durante los primeros meses de la simulación de la planta BSM1, ya que en esta se produce un transitorio de arranque hasta que las reacciones químicas desarrolladas en su interior se estabilizan, y se pretende evitar entrenar al agente en este estado cambiante. Además, se observó que la simulación tendía incluso a fallar con bastante frecuencia (valores infinitos de los estados internos) si no se permitía a la plata evolucionar hacia el equilibrio antes de comenzar el entrenamiento.

Por tanto, durante las primeras 12 semanas, se alimentará a la simulación unas entradas correspondientes a un tiempo `steady`; durante las siguientes 4 semanas, las correspondientes a un tiempo `dry` (que también es relativamente estable), y a partir de ese momento la planta recibirá una mezcla aleatoria de tiempos de todo tipo (excepto `steady`), con probabilidad de `dry` del 70%, `str` del 10%, y `rain` del 20%. Este periodo de inicialización de un total de 16 semanas se realizará precisamente en el método `reset()` de `BSM1Env`, y por tanto será un tiempo durante el cual el agente no interactuará todavía con la planta. En cambio, se aplicará a la planta una acción por defecto, que se obtiene del método `_get_initial_inputs()` de `BSM1Env`.

La Figura 13 muestra el caudal (uno de los valores de entrada que se proporcionan a la planta, junto con las características químicas de este caudal) que genera la clase `WWSourceGen` para todo un episodio. En concreto, en el método `reset()` el entorno llama al método `generate()` de `WWSourceGen`, generando así ya los datos que serán necesarios para toda la duración de la simulación. Cada vez que se llame el método `step()` del entorno, se llamará a `get_inputs(t0, t1)` de `WWSourceGen`, que acepta un tiempo de inicio (coincidente con el paso de tiempo actual del `step()`), y un tiempo final (el tiempo de simulación tras el `step()`), y devuelve los `inputs` generados que se encuentran entre esos dos pasos de tiempo (en realidad, siempre se devuelve uno más por cada extremo para que internamente el simulador pueda siempre interpolar linealmente sin necesidad de extrapolar).

Finalmente, dentro de `bsm1_env.py` se define también una función: `operation_cost`, que implementa las funciones de coste discutidas en Coste de operación, y que toma de entrada un objeto de resultados de simulación y devuelve como salida el coste medio diario de operación (u, opcionalmente, los vectores con los costes instantáneos separados por concepto del coste). Esta función es utilizada tras cada `step()` para obtener la recompensa r asociada a la tupla $< s, a >$, que será sencillamente la salida de la función multiplicada por -1. También se empleará al final de la simulación de un episodio para calcular el coste medio diario en el que se ha incurrido durante este.

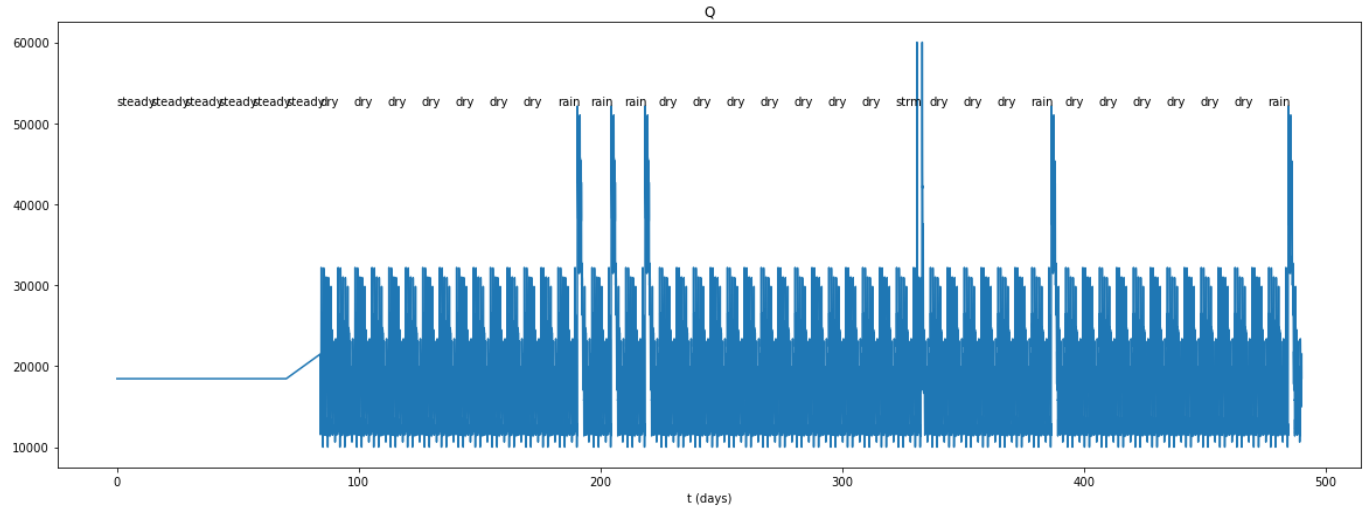

Figura 13: Caudal de entrada en función del patrón meteorológico a lo largo de todo un episodio

3.5. Diseño de los agentes de aprendizaje por refuerzo

Una vez definido el entorno, el siguiente paso es definir los agentes que deben interaccionar con él. Concretamente, se han implementado tres tipos de agentes: Q-Learning, EV-SARSA y Deep Q-Learning, siendo este último una versión del clásico Q-Learning donde el aproximador a la función Q es una red neuronal, en lugar de ser una tabla. También se ha definido un cuarto tipo de agente: Dumb, que en realidad es un agente "tonto" que siempre toma la misma acción, y que servirá como comparativa para justificar que el resto de los agentes verdaderamente obtienen un beneficio.

La Figura 14 muestra el diagrama de clases del módulo `agents.py`, donde se implementan todos estos agentes. Como se puede observar, todos ellos heredan de la clase abstracta `QAgent`, que hace las veces de interfaz, y únicamente implementa la lógica para generar el nombre (`name`) del agente, que se corresponderá con el nombre de la clase seguido de un identificador único asociado al momento de instanciación del agente; y el `save_path`, que se construye a partir de este. El agente más sencillo posible es `DumbQAgent`, que recibe durante su inicialización el valor de la acción por defecto (1.2 en este caso), y siempre retornará este valor al llamar a los métodos `get_action()` o `get_best_action()`.

Por otra parte, `QLearningAgent` implementa el algoritmo Q-Learning (Algoritmo 1). Hay dos aspectos destacables de la implementación: En primer lugar, permite que los argumentos del método `train()` sean una lista, lo que le permitirá entrenar simultáneamente con un lote de varias tuplas $< s, a, r, s' >$, lo cual facilita su uso en conjunción con la técnica de la repetición de experiencia. En segundo lugar, la tabla Q se va construyendo sobre la marcha. Así, al intentar acceder o actualizar un valor Q que aún no había sido definido, el agente automáticamente reservará espacio para dicho elemento de la tabla y, en el caso de ser un acceso (y no una actualización), establecerá el valor inicial a `initial_qvalue`. De esta forma, el usuario de la clase no tiene que preocuparse por estos detalles.

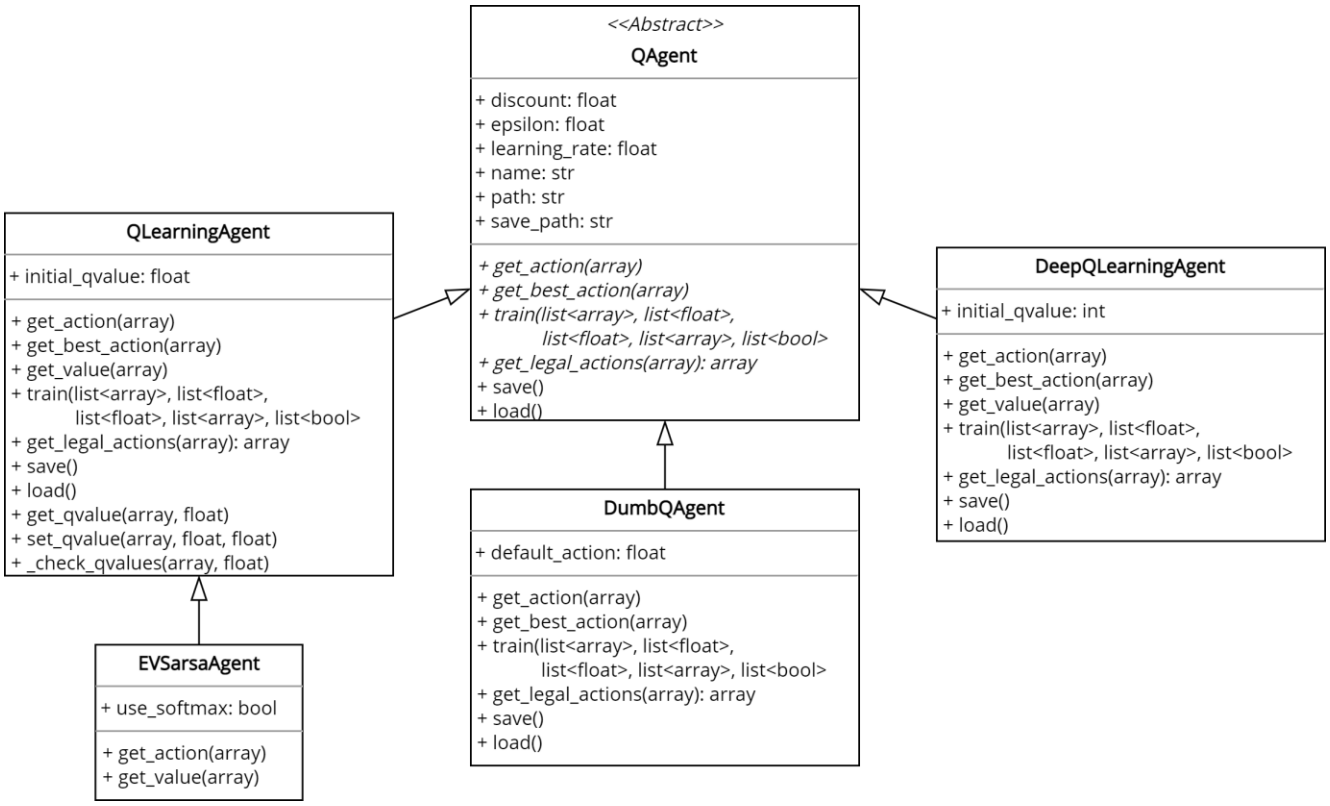

Figura 14: Diagrama de clases del módulo `agents.py`.

La clase `DeepQLearningAgent` implementa un agente Q-Learning que emplea una red neuronal para aproximar la función Q, en lugar de una tabla. El código básico que realiza el entrenamiento de la red neuronal en el método `train()` se muestra en el Código 5. De forma muy resumida, se comienza obteniendo los Q-valores estimados por la red para cada estado de `state` (recordemos que `state` es una lista de estados, o una matriz en este caso) dadas las `actions` tomadas por el agente (la red proporciona los Q-valores para todas las acciones posibles dado un estado, pero se deben indexar solo aquellas acciones que realmente se llevaron a cabo).

```
#Get q-values for chosen actions
predicted_qvalues = self.network(states)
actions_indices= np.array([np.where(np.abs(self.get_legal_actions(None)-a) < 0.001) for a in
    actions]).flatten()
predicted_qvalues_for_actions = predicted_qvalues[range(states.shape[0]), actions_indices]

#Compute V*(next_states) and "target q-values" for loss
next_state_values= self.get_value(next_states)
target_qvalues_for_actions = rewards + self.discount * next_state_values
target_qvalues_for_actions = torch.where(is_done, rewards, target_qvalues_for_actions)

#Mean squared error loss to minimize
loss = torch.mean((predicted_qvalues_for_actions - target_qvalues_for_actions.detach()) ** 2)

#Train
self.optimizer.zero_grad()
loss.backward()
self.optimizer.step()
```

Código 5: Fragmento del método `train()` de `DeepQLearningAgent`

A continuación se computa la función V_* para los próximos estados, ya que en Q-learning $V_*(s') = max_{a'} Q(s', a')$, y se calcula el nuevo valor de Q utilizando la aproximación por diferencias temporales de la Ecuación (19) ($Q(S, a) = r + \gamma \cdot V_*(s')$), teniendo en cuenta que en los últimos estados de un episodio (cuando `is_done == True`), la Ecuación (20) se simplifica a $Q(S, a) = r$, ya que $V_*(s')$ es efectivamente 0. Una vez obtenido el valor actual de Q (`predicted_qvalues_for_actions`), y el que la regla de actualización indica que debería tener (`target_qvalues_for_actions`), se calcula el error cuadrático medio entre ambos (utilizando `.detach()` en el valor objetivo para evitar que el gradiente se propague también a través de este), se calcula el pase hacia atrás (`loss.backward()`) para obtener las derivadas de todos los elementos de la red respecto a la función de coste, y se avanza el optimizador un paso. En esta implementación se ha utilizado Adam como optimizador.

Por último, la clase `EVSarsaAgent` es idéntica a `QLearningAgent` excepto en los métodos `get_action()` y `get_value()`, y en el parámetro de inicialización `use_softmax`. Si `use_softmax == True`, entonces se utilizará la política de Boltzman (Ecuaciones (22)-(23)). En caso contrario se empleará la política ε-greedy (por defecto en Q-Learning). Además, `get_value()` ahora debe calcular el valor esperado (Ecuación (21)) (y no el máximo como en Q-Learninig), y `get_action()` elegirá la acción dependiendo de la política según el valor de `use_softmax`.

Junto con el módulo `agents.py`, el módulo `wrappers.py` (Figura 15) proporciona una serie de utilidades para facilitar la integración de los agentes en un entorno. Todas estas utilidades implementan la clase `gym.core.ObservationWrapper`, la cual hereda en última instancia de `gym.core.Env`, y permite procesar y extender las observaciones (los estados) que devuelve el entorno. De forma muy resumida, la clase `Normalizer` permite estandarizar (Ecuación (36)) los estados que retorna el entorno de tal forma que puedan ser usados directamente por un agente profundo (los cuales por lo general requieren de esta estandarización); la clase `Binarizer` permite discretizar los estados para que puedan ser usados por un agente tabular; y la clase `StateHolder` incluye en la observación actual los valores de las últimas `N` observaciones, permitiendo así mejorar potencialmente la observabilidad de un entorno parcialmente observable.

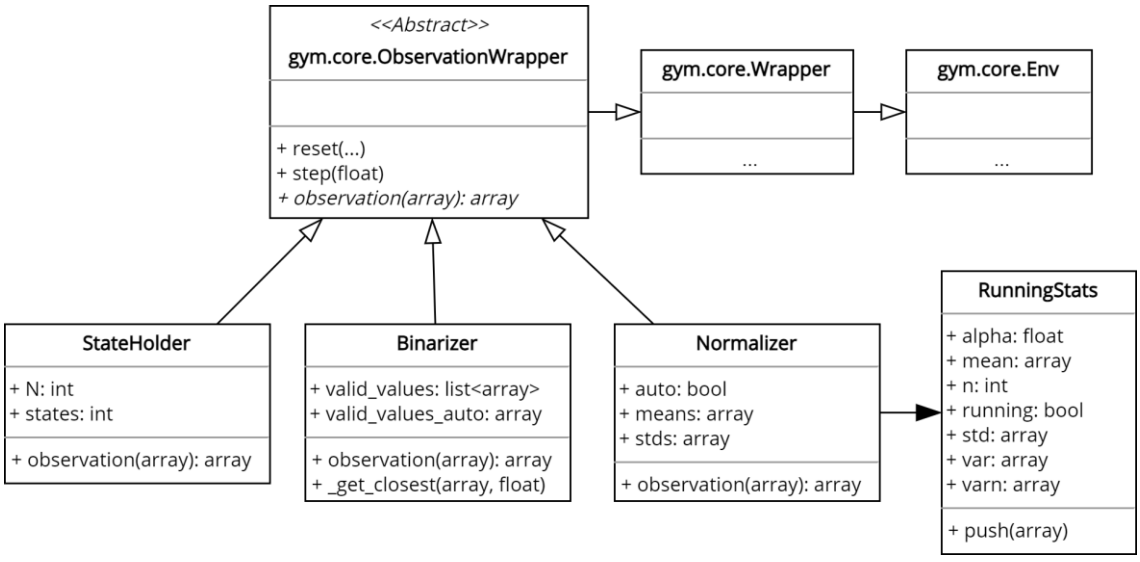

Figura 15: Diagrama de clases del módulo `wrappers.py`. Se han omitidos los métodos y atributos de las clases `Wrapper` y `Env` por claridad, y dado que pertenecen a la librería Gym, y no al módulo `wrappers.py`.

Ahora bien, lo que hace a las clases `Binarizer` y `Normalizer` especialmente interesantes es que tienen un modo de funcionamiento manual, y uno totalmente automático. Así, en el caso de `Normalizer`, el usuario de la clase puede bien proporcional los valores de la media y desviación estándar del estado (los cuales normalmente no son conocidos a priori), o puede dejar que `Normalizer` los calcule por sí mismo de forma dinámica haciendo uso de la clase `RunningStats`. Del mismo modo, en el caso de `Binarizer` se pueden definir manualmente los valores en los que se quiere discretizar cada variable del estado o, alternativamente, si se aplica primero un `Normalizer` al entorno, y luego un `Binarizer`, se sabe ya que discretizando, por ejemplo, los valores entre -2 y 2 en pasos de 0.5, el ~95.5% de los valores del estado entrarán en ese rango (dado que los estados tienen ahora una distribución normal de media 0 y desviación estándar 1 gracias a la acción de `Normalizer`), y por tanto la discretización puede ser totalmente automática (excepto quizás la decisión del paso de digitalización). Así, se logra que el usuario no tenga que preocuparse de estos detalles (que no obstante son imprescindibles para el correcto funcionamiento de los agentes), ya que estos *wrappers* lo hacen por él.

3.6. Entrenamiento de los agentes

Una vez implementado el entorno y los agentes, solo queda pasar al entrenamiento de estos. Para ello se hace uso de la Jupyter Notebook `Training.ipynb`, y de las funciones definidas en el módulo `training.py`. Jupyter Notbeook es una herramienta que se ejecuta directamente en un explorador de internet y, conectada a un kernel Python, permite ejecutar celdas de código Python y obtener la salida de cada una de estas, a modo de libreta, combinando así código y elementos gráficos de todo tipo en un mismo documento. `Training.ipynb` se encarga de realizar cuatro tareas principales: Instanciación y configuración del entorno BSM1Env, Instanciación y configuración del agente, Entrenamiento del agente en el entorno y, finalmente se encarga de mostrar los Resultados.

3.6.1. Instanciación y configuración del entorno BSM1Env

En este primer paso se establece la configuración del entorno. Algunos de los parámetros más interesantes se muestran en el Código 6. `start_time` define el número de días de estabilización de la planta, `max_time` define la duración del episodio, `time_step` define el paso de tiempo que transcurre cada vez que se llama a `step()` (y que en este caso se ha fijado a 15 minutos), `discount_factor_period` establece el número de días para que el factor $\gamma^{días}$ valga tan solo 1%, y define por tanto la longevidad de la recompensa (Ecuación (5)), `action_names` es la lista de las variables (según el criterio de nombres definidos en el modelo original en Modelica) sobre las que el agente actúa, y `output_names` define la lista de las variables que definen el estado de la planta (o mejor dicho, aquellas que el agente observa). Más concretamente, las variables `output_names`: `'limiter1.y','sensor_class_a1.y'`, se refieren, respectivamente a la concentración de NH_4 (mg/L), y de O_2 (mg/L) del tanque 5. Una variable que aquí no aparece, pero que también se utilizó con el agente de Deep Q-Learning, es `'ADsensor_Waste.TSS'`, que representa la concentración de sólidos en suspensión en el caudal de desecho de la EDAR (en mg/L). Se recuerda también que `'agent_action'` actúa sobre el punto de consigna de oxígeno disuelto en el tanque 5, lo que indirectamente actuará en el aireador de dicho tanque a través de un control PI.

```
#Time-related configuration
start_time= 14*8 #Wait (days) for plant stabilization. This time is simulated during env.reset()
time_step= 1./24./4. #Simulation time step when env.step() is called (days)
max_time= start_time + 365 #Total simulation time (days)
discount_factor_period= 14 #After how many days the reward is worth 1%
discount_factor= 0.01**(time_step/discount_factor_period)
action_names= ['agent_action']
output_names= ['limiter1.y', 'sensor_class_a1.y']
default_opts= {
    'solver':'Dopri5',
    'Dopri5_options': {'inith': 0.0001} #This value has a big impact in the simulation speed
    'result_handling':'memory', #Keep results in memory rather than saving them into a file
}

#Create an env instance
env= gym.make('BSM1Env-v0')
```

Código 6: Configuración de BSM1Env

También se definen `default_opts`, que será la configuración que se proporciona al simulador de PyFMI. En este caso, se vio empíricamente que el *solver* de ecuaciones diferenciales ordinarias *DormandPrince* (`'Dopri5'`) era el más apropiado en términos de estabilidad, velocidad y espacio (en RAM), así que fue el que finalmente se empleó. Además, se indica que los resultados se gestionen en memoria, para evitar así la creación innecesaria de archivos temporales en disco. Por último, se instancia el entorno con esta configuración en la variable `env` (aquí se omiten algunos pasos, que se pueden consultar en `Training.ipynb`).

3.6.2. Instanciación y configuración del agente

Respecto a la configuración del agente, el Código 7 muestra los principales parámetros: `AGENT` define el tipo del agente (`'q'` para QLearningAgent, `'deepq'` para DeepQLearningAgent, etc.), `EPSILON` define el ratio de exploración (es la épsilon en la política ε-greedy), `EPISODES` indica el número de episodios a simular, `REWARD_SCALE` es un factor por el que se multiplica la recompensa para evitar que tenga valores muy grandes (lo cual perjudica a algoritmos basados en función de coste, como Deep Q-Learning), `REPLAY_BUFFER` es una instancia de un `ReplayBuffer`, una clase que implementa un buffer de repetición de experiencia y que se encuentra en `training.py`, `BATCH_SIZE` es el tamaño de lote que se muestreará del `REPLAY_BUFFER` cada vez que se llame al método `train()` del agente, `HOLD_N` es el número de muestras que el *wrapper* `StateHolder` debe mantener, `SCHEDULER_POWER` indica el tipo de decaimiento de `EPSILON` a lo largo de un episodio (p.ej.: 0: no hay decaimiento, 1: lineal, 2: cuadrático, etc.), y `get_legal_actions` es una función que toma un estado y devuelve un array con las acciones que se pueden llevar a cabo en dicho estado (notar que en este caso se ignora el parámetro estado, porque siempre se pueden llevar a cabo todas las acciones en cualquier estado).

```
#Agent configuration
AGENT= 'deepq' #['q', 'deepq', 'sarsa', 'dumb']
EPSILON= 0.75
EPISODES= 1
REWARD_SCALE= 1e-3 / 200
REPLAY_BUFFER= ReplayBuffer(100)
BACTH_SIZE= 64
HOLD_N= 1 #If HOLD_N > 0: Use StateHolder wrapper
SCHEDULER_POWER= 0
get_legal_actions=lambda state: np.array([1.2, 1.5, 1.8])
```

```
#Define a network architecture, needed for deep algorithms
state_dim = np.array(env.observation_space.shape) * (HOLD_N + 1)
n_actions= len(get_legal_actions(None))
network = nn.Sequential(
          nn.Linear(state_dim[0], 50), nn.ReLU(),
          nn.Linear(50, 50),           nn.ReLU(),
          nn.Linear(50, n_actions),
          )

#Choose the agent that we will train with
if AGENT == 'q':
    env = StateHolder(Binarizer(Normalizer(env), hold_last_N=HOLD_N)
    agent= QLearningAgent(epsilon=EPSILON, discount=discount_factor,
                          get_legal_actions=get_legal_actions)
elif AGENT == 'deepq':
    env = StateHolder(Normalizer(env), hold_last_N=HOLD_N)
    agent= DeepQLearningAgent(network, epsilon=EPSILON, discount=discount_factor,
                              get_legal_actions=get_legal_actions, learning_rate=1e-5)
(...)
```

Código 7: Configuración del agente

A continuación, se define la red neuronal (`network`) que aproximará la función Q para un DeepQLearningAgent, y que contiene dos capas ocultas de dimensión 50, y con activación ReLU. Finalmente, en función del tipo de agente, se envuelve el entorno `env` en unos *wrappers* u otros, según lo discutido en Diseño de los agentes de aprendizaje por refuerzo, y se instancia finalmente el agente (`agent`).

3.6.3. Entrenamiento del agente en el entorno

El último paso es el entrenamiento en sí, que se realiza a través de dos bucles anidados, el primero de los cuales se ha representado simplificado en el Código 8. Este bucle ejecutará la función `play_episode()`, que está definida en `training.py`, tantas veces como `EPISODES` se haya definido (es decir, solo una vez, en este caso).

```
RESULTS_NAME= os.path.join('./data', agent.name + '_results')
total_rewards, rewards, states= [], [], []

for i in range(EPISODES):
    total_reward, reward, state, result= play_episode(env, agent,
        rb_batch=BACTH_SIZE, reward_scale=REWARD_SCALE,
        replay_buffer=REPLAY_BUFFER, epsilon_scheduler=EPSILON_SCHEDULER)
    total_rewards.append(total_reward); rewards.append(reward); states.append(state)
    agent.save()
    pickle.dump(result, open(RESULTS_NAME + '_%d.pkl'%i,'wb'))
```

Código 8: Bucle externo de entrenamiento

Dentro de `play_episode()` existe otro bucle que va avanzando paso a paso la simulación y entrenando al agente. Una versión reducida de este código se encuentra en Código 9. Dado que el código se explica mayoritariamente por si mismo, únicamente se mencionará que el parámetro `train` tiene la función de indicar si el agente se debe entrenar o no. En caso de que no, la acción seleccionada será siempre la mejor (y no siguiendo una política subóptima de exploración), y el método de `agent.train()` no será utilizado. En este caso concreto, el agente siempre se encuentra en entrenamiento (`train= False` siempre), ya que el objetivo es que este se pueda incorporar a la planta y tomar el control desde el primer momento, y

que además sea capaz de adaptarse a cambios significativos en los parámetros de funcionamiento de esta.

```
#Reset the environment
s= env.reset()

#Simulate step by step
for t in trange(t_max):
    #Get action and step
    if train:
        a = agent.get_action(s)
    else:
        a = agent.get_best_action(s)
    next_s, r, done, result = env.step(np.array([a]))
    r*= reward_scale

    #Train the agent
    if train:
        replay_buffer.add(s, a, r, next_s, done)
        agent.train(*replay_buffer.sample(rb_batch))
    s = next_s

    #Plot every 14 days
    (...)

    #Exit loop if environment is done
    if done: break
```

Código 9: Bucle interno de entrenamiento

Durante el entrenamiento se muestra cada 14 días de simulación una visualización del progreso, cuya captura se encuentra en la Figura 16 para un QLearningAgent, y en la Figura 17 para un DeepQLearningAgent. En la gráfica superior se muestra la recompensa (en azul) y también filtrada con un filtro de media móvil (en naranja), mientras que en la gráfica inferior se muestra el valor de cada uno de los estados, así como de la acción del agente. Como se aprecia, en la Figura 16 estos están discretizados debido al uso del *wrapper* `Binarizer`, mientras que en la Figura 17 no está discretizados. En ambos casos las observaciones están previamente estandarizadas con el *wrapper* `Normalizer` (esto se aprecia porque tienen un valor cercano al cero). Se usa además el *wrapper* `StateHolder` con $N = 1$, lo cual se comprueba en la leyenda de las figuras, la cual indica que hay cuatro estados, que en realidad son dos mantenidos un paso de tiempo adicional. En la gráfica no se aprecian estos estados adicionales porque coinciden casi exactamente con los estados actuales, y por tanto están sobrepuestos.

Llegados a este punto, se realizó un barrido de hiperparámetros para encontrar los agentes que mejores resultados (en términos de ahorro diario) proporcionaran. La configuración de final de estos agentes y el ahorro logrado se discute en el apartado Resultados y discusión.

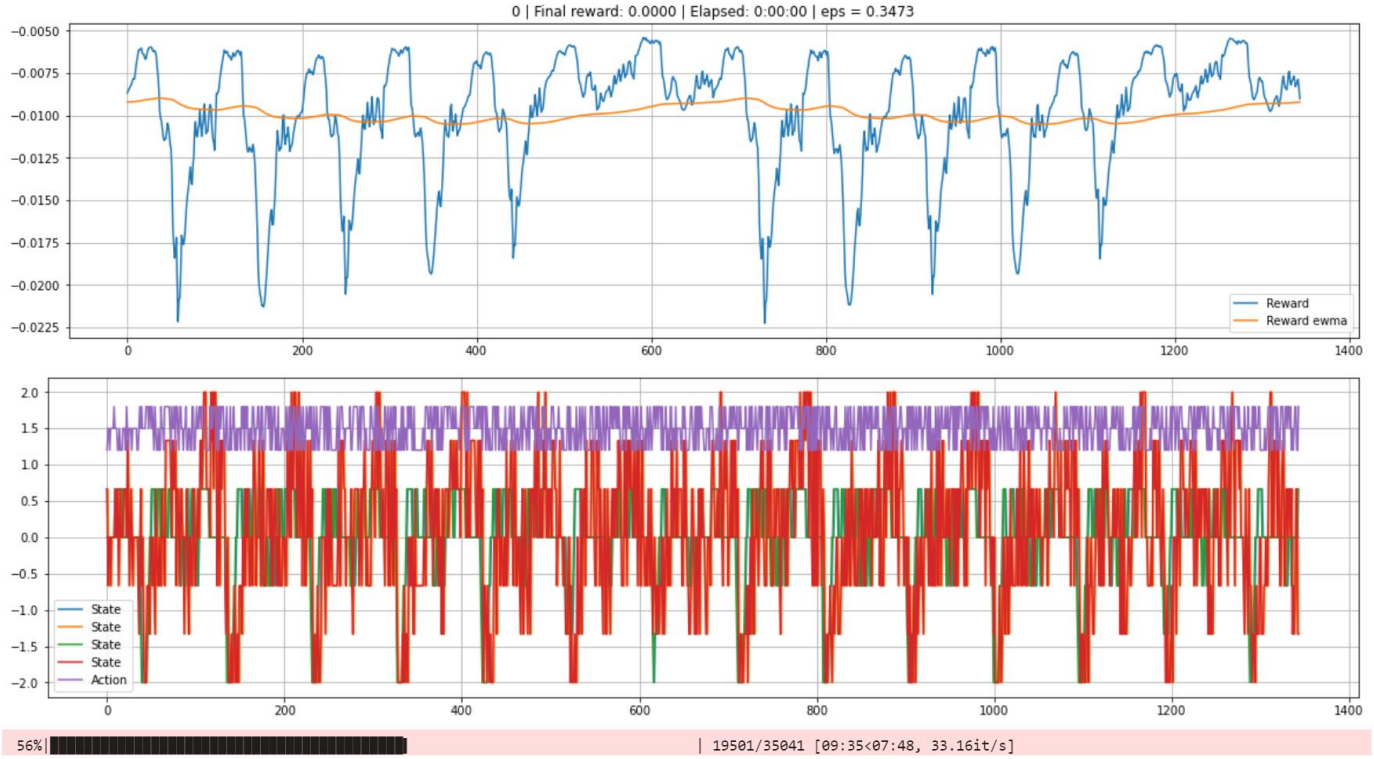

Figura 16: Visualización del progreso del entrenamiento de un QLearningAgent con wrappers StateHolder(N=1), Binarizer y Normalizer

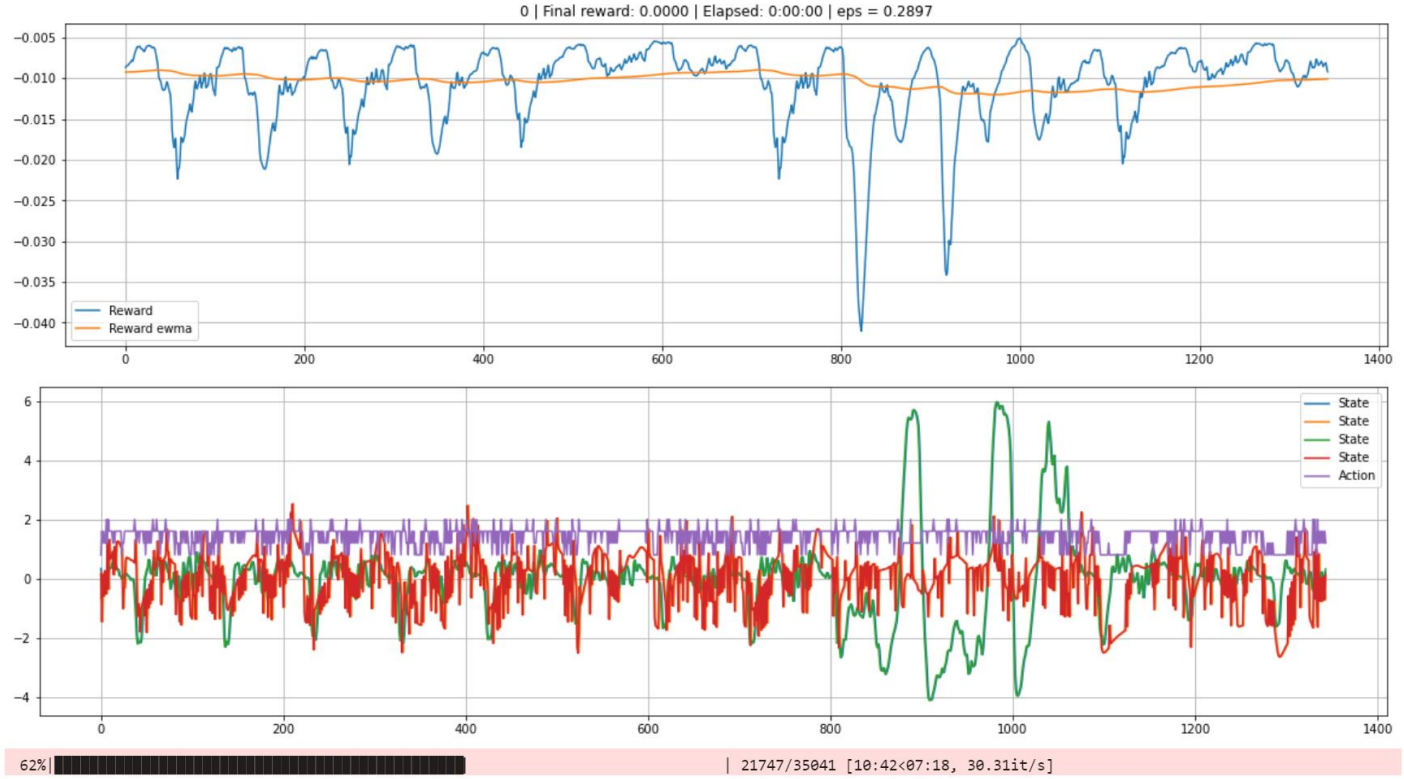

Figura 17: Visualización del progreso del entrenamiento de un DeepQLearningAgent con wrappers StateHolder(N=1), y Normalizer

4. RESULTADOS Y DISCUSIÓN

Tras múltiples pruebas con todos los agentes, y multitud de combinaciones de hiperparámetros, finalmente se vio que la configuración con mejor rendimiento usaba un tamaño de lote de 32, es decir, cada vez que se llama al método `train()` del agente, se entrena con 32 tuplas $< s, a, r, s' >$ obtenidas de un buffer de repetición de experiencia, el cual se ajustó a 100 muestras de tamaño. Además, la ratio de exploración $\epsilon = \tau = 0.5$, siendo ϵ el parámetro utilizado en Q-Learning y Deep Q-Learning (que emplean exploración ε-voraz), y τ el parámetro empleado en EV-SARSA (que realiza muestreo de Boltzman); ambos valores se mantienen constantes durante todo el episodio. La ratio de aprendizaje se mantuvo en su valor por defecto de 0.5 en los agentes tabulares, y $1 \cdot 10^{-4}$ en el agente profundo. Respecto a este último, los mejores resultados se obtuvieron utilizando una red con dos capas ocultas de dimensión 50 cada una. Con todos los agentes se usaron los *wrappers* `StateHolder` y `Normalizer`, y con los algoritmos tabulares se empleó adicionalmente `Binarizer`, todos ellos funcionando en el modo totalmente automático.

4.1. Rendimiento en tres escenarios diferentes

Con Deep Q-Learning el agente observó un estado adicional, `ADsensor_Waste.TSS`, (o la concentración de partículas en suspensión en el agua efluente, a parte del NH_4 y del O_2 del tanque 5) pues medir este mejoraba el rendimiento de este agente (si bien no el de los tabulares). La Tabla 1 muestra una comparativa del coste diario medio (€) y del ahorro anual total (€) que proporciona el uso de cada agente. Esta comparativa se ha repetido para tres episodios con diferentes patrones meteorológicos, con el fin de demostrar que el rendimiento se mantiene independientemente de este factor. Se incluye también el modelo *Dumb*, que se limita a mantener una acción constante de valor 1.2, y que sería la referencia a batir por el resto de las agentes. En efecto, los ahorros anuales se han calculado como la diferencia de coste diario entre un agente y el agente *Dumb* acumulado a lo largo de un año.

Patrón meteorológico		Agente			
		Q-Learning	*EV-SARSA*	*Deep Q-Lear.*	*Dumb*
A	Coste diario (€)	2032.36	**2031.56**	2033.25	2037.83
	Ahorro anual (€)	1995.98	**2289.31**	1670.62	0
B	Coste diario (€)	**2018.76**	2018.82	2019.29	2031.26
	Ahorro anual (€)	4565.54	**4640.89**	4381.22	0
C	Coste diario (€)	2031.98	**2031.48**	2033.24	2037.42
	Ahorro anual (€)	1986.24	**2170.85**	1524.97	0

Tabla 1: Comparativa del rendimiento de diversos agentes sobre tres episodios con patrones meteorológicos aleatorios diferentes de un año de duración.

Como se aprecia, en todos los escenarios el ahorro es significativo utilizando los agentes. Como comparativa, la Figura 18 muestra una gráfica (para el patrón A) similar a las presentadas en (Hernández-del-Olmo et al., 2016, 2018), donde se puede ver claramente que el coste diario de operación dada una consigna constante (el agente *Dumb*) es superior al del resto de los agentes, que prácticamente coinciden unos sobre los otros.

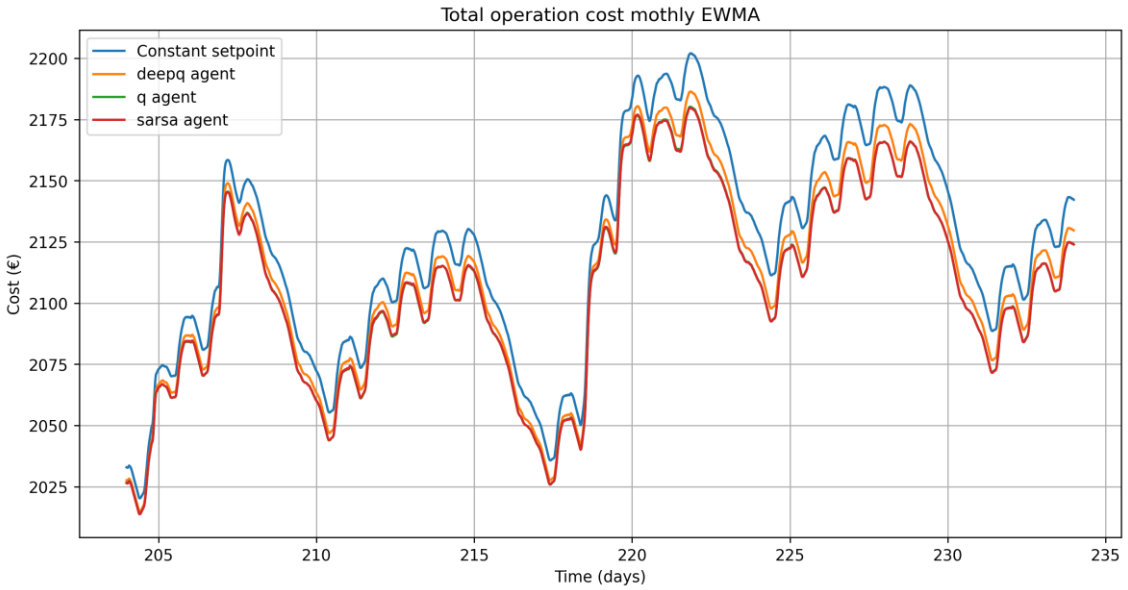

Figura 18: Patrón A: 30 días de evolución del coste diario (filtrado con media móvil mensual) de cada uno de los agentes. Se recuerda que el entrenamiento comienza en el día 112 (tras 8 semanas de inicialización).

Una representación especialmente interesante (patrón A) es la que se introduce en la Figura 19, que muestra un desglose del ahorro acumulado que va obteniendo el agente EV-SARSA (para el patrón A) durante el año que dura el episodio. Como se aprecia, el mayor ahorro (respecto al agente *Dumb* con consigna constante) proviene de las multas por efluente (línea marrón), mientras que el mayor consumo del agente proviene del coste de la energía de aireación (línea naranja). Como se había previsto, el agente logra ahorro al reducir emisiones contaminantes, pero lo hace a costa de aumentar el gasto en energía de aireación. Los ahorros totales se muestran en azul, y son ascendentes, lo cual justifica la utilidad del agente. Además, es especialmente destacable que las zonas de mayor ahorro neto (las subidas más marcadas de la línea azul) se corresponden con un tiempo lluvioso o tormentoso, dando a entender que el agente tiene poco margen de maniobra en tiempo seco, y saca la ventaja sobre todo en estos tiempos más extremos.

A modo de curiosidad, la Figura 20 muestra los estados y las acciones llevadas a cabo por el agente EV-SARSA (patrón A) durante tres días. Un hecho a destacar es el alto nivel de rizado de la acción del agente, lo cual es una consecuencia inevitable de tener un régimen de exploración tan agresivo $\tau = \epsilon = 0.5$. En simulaciones realizadas se comprueba que reducir el régimen de exploración da lugar a acciones de control menos erráticas, pero que no obstante obtienen un ahorro menor. En efecto, el hecho de que el tamaño óptimo del buffer de repetición de la experiencia sea de apenas 100 elementos evidencia la enorme importancia de la exploración continua en este entorno (ya que parece que tan solo las últimas 100 experiencias aporten utilidad).

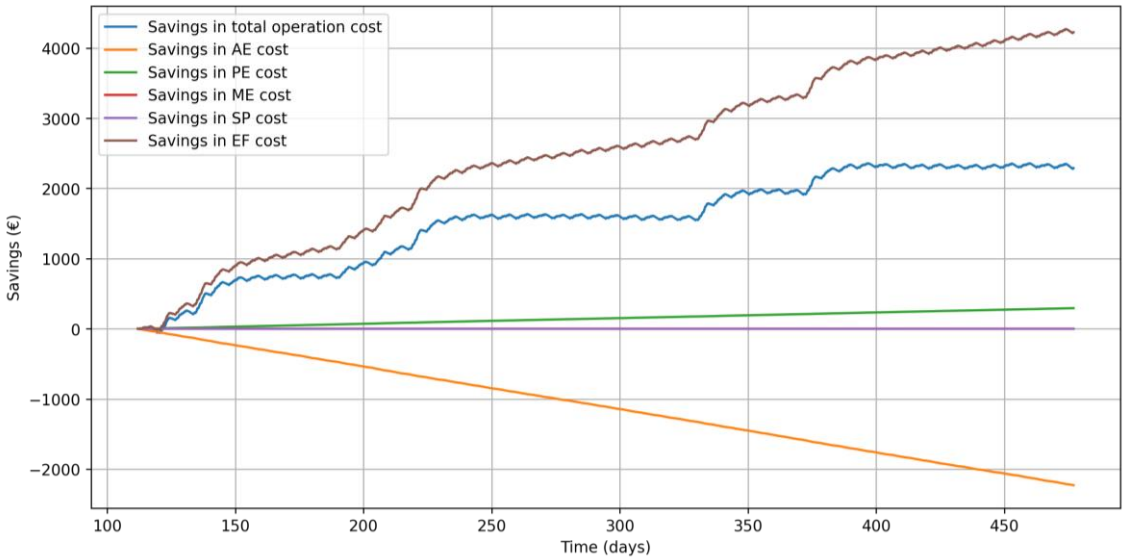

Figura 19: Patrón A: Evolución del ahorro total del agente EV-SARSA desglosado según concepto del coste, a lo largo del año de duración del episodio.

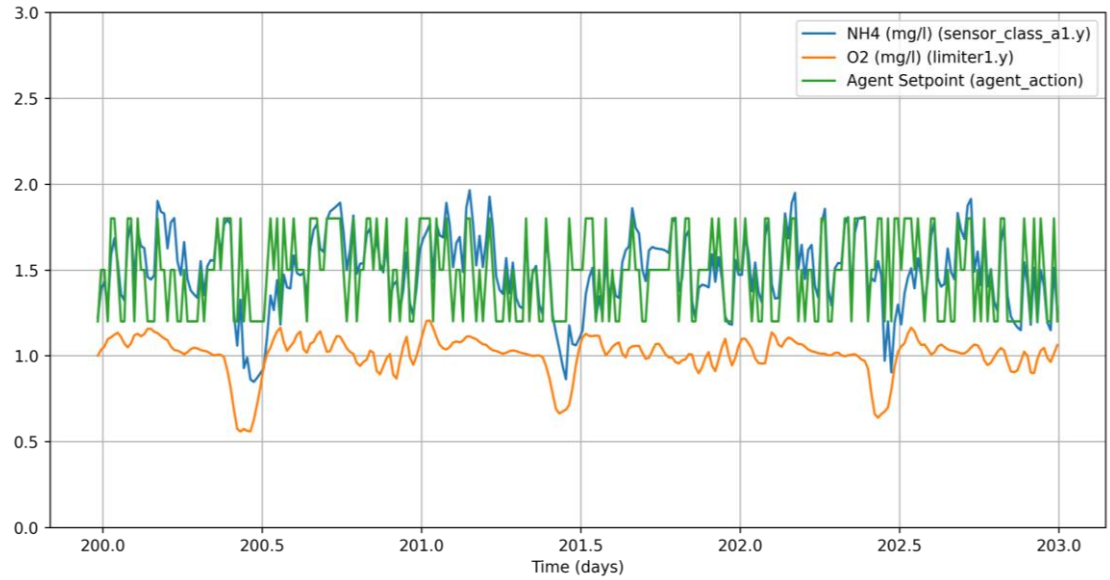

Figura 20: Patrón A: Muestra de los valores de los estados (en realidad, el valor medio de estos cada 15 minutos) y de la acción del agente EV-SARSA durante 3 días.

Un último aspecto que remarcar es que, aunque en la Tabla 1 se han representado unos resultados concretos, si se volviera a ejecutar la simulación los costes y ahorros sería algo distintos debido a la estocasticidad del agente y del propio simulador (notar que las entradas no aportan estocasticidad ya que son siempre las mismas para un patrón dado). Lo que se observa empíricamente es que EV-SARSA, además de ser el algoritmo superior en los tres escenarios, resulta ser también el más estable frente a esta estocasticidad. Se concluye por tanto que EV-SARSA (con la configuración descrita) es el agente más adecuado para este entorno.

En definitiva, comparando los resultados con (Hernández-del-Olmo et al., 2016, 2018), aunque se observan resultados similares en algunas gráficas (Figura 18), en general se logran ahorros menores acumulados anualmente, quedando pues aún trabajo pendiente.

4.2. Rendimiento a largo plazo

Por último, se analizó el rendimiento a largo plazo del agente realizando una simulación de más de 5 años de duración (5 años de interacción del agente con el entorno, más el tiempo de inicialización). Por motivos de economía, se mostrarán solo los resultados para EV-SARSA, ya que además se ha considerado que es el agente más apropiado para el entorno.

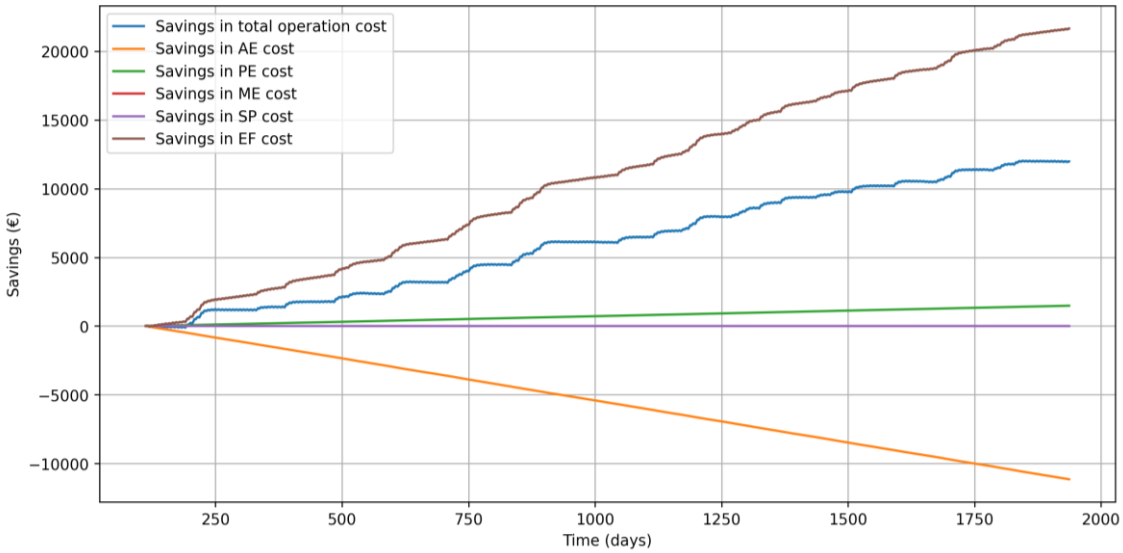

Figura 21: Evolución del ahorro total del agente EV-SARSA desglosado según concepto del coste, a lo largo de un episodio extendido de 5 años.

Los ahorros de este agente desglosados por los distintos tipos de coste se pueden ver en la Figura 21, donde se comprueba que el ahorro es mantenido a lo largo de los años, a pesar de los cambios aleatorios que pueda haber de las condiciones meteorológicas. En efecto, el ahorro al cabo de los 5 años (con respeto al agente *Dumb*) es de 11986,17 €, o lo que es lo mismo, un ahorro de 2397,23 € por año, similar a lo que se vio en la Tabla 1.

5. CONCLUSIONES

En este proyecto se han desarrollado un conjunto de herramientas que permiten trasladar cualquier modelo implementado en Modelica a un entorno de OpenAI Gym, desarrollar agentes que interactúen con este, y entrenarlos para optimizar un determinado aspecto de su funcionamiento. En concreto, este procedimiento se ha aplicado al modelo BSM1 de una EDAR, en el que se requería un agente que balanceara el consumo eléctrico con la calidad del agua efluente mediante la optimización de la consigna de la concentración del oxígeno del último tanque aeróbico de esta planta. Una de las ventajas inmediatas de trasladar el problema a este nuevo entorno es la libertad para desarrollar cualquier tipo de agente de RL (incluidos agentes profundos), empleando diferentes técnicas (como la repetición de la experiencia), y optimizando con facilidad su configuración.

Concretamente se han empleado agentes Q-Learning, Deep Q-Learning y EV-SARSA, siendo este último el que mejor rendimiento y estabilidad ha mostrado en las pruebas realizadas, y manteniendo un ahorro continuado de más de 2000€ al año en todas ellas, incluso en una de 5 años de duración. En la práctica, este agente podría implementarse en cualquier EDAR similar a BSM1 y, dado que siempre se mantiene explorando, debería ser capaz de adaptarse tanto a las características de la nueva planta, como a cambios profundos en sus condiciones de funcionamiento, en las características de los caudales influyentes, o frente a la presencia de perturbaciones externas, como demuestra su estabilidad frente a los variantes patrones meteorológicos. En general, los tres agentes propuestos funcionan similarmente bien, lo que puede significar que el problema está tan optimizado como sus características lo permiten. De esta forma, se ha resuelto satisfactoriamente (en la medida de los posible) el problema económico y medioambiental que la operación de estas EDAR supone.

Finalmente, las herramientas desarrolladas tienen una potencialidad inmensa para optimizar con facilidad otros aspectos del funcionamiento de BSM1, o de cualquier otro modelo en Modelica. En este sentido, los *wrappers* desarrollados sobre el entorno permiten automatizar aspectos como la estandarización y discretización de los datos de entrada, que son por lo general tediosos, pero necesarios para el correcto funcionamiento de este tipo de algoritmos; y la modularidad del código desarrollado permite el desarrollo de más agentes, aprovechando una interfaz común, y haciéndolos de esta forma muy fácilmente integrables en el sistema. Por tanto, con este trabajo, y gracias a la publicación de la totalidad del código en abierto, se espera dejar la puerta abierta a futuros desarrollos sobre este sistema.

6. BIBLIOGRAFÍA

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., … Zheng, X. (2016). TensorFlow: A system for large-scale machine learning. *ArXiv*. http://arxiv.org/abs/1605.08695

Al-Dosary, S., Galal, M. M., & Abdel-Halim, H. (2015). Environmental Impact Assessment of Wastewater Treatment Plants-(Zenien and 6 th of October WWTP). In *Int.J.Curr.Microbiol.App.Sci* (Vol. 4, Issue 1). http://www.ijcmas.com

Alex, J., Benedetti, L., Copp, J. B., Gernaey, K. V, Jeppsson, U., Nopens, I., Pons, M.-N., Rieger, L. P., Rosén, C., Steyer, J.-P., Vanrolleghem, P. A., & Winkler, S. (2008). Benchmark Simulation Model no.1 (BSM1). In *Lutedx: Vol. TEIE-7229* (Issue 1).

Bagheri, M., Mirbagheri, S. A., Bagheri, Z., & Kamarkhani, A. M. (2015). Modeling and optimization of activated sludge bulking for a real wastewater treatment plant using hybrid artificial neural networks-genetic algorithm approach. *Process Safety and Environmental Protection*, *95*, 12–25. https://doi.org/10.1016/j.psep.2015.02.008

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., & Zaremba, W. (2016). *OpenAI Gym*. http://arxiv.org/abs/1606.01540

Chong, K. F. E. (2020). A closer look at the approximation capabilities of neural networks. *ICLR 2020*. http://arxiv.org/abs/2002.06505

Cristea, S., de Prada, C., Sarabia, D., & Gutiérrez, G. (2011). Aeration control of a wastewater treatment plant using hybrid NMPC. *Computers and Chemical Engineering*, *35*(4), 638–650. https://doi.org/10.1016/j.compchemeng.2010.07.021

Fritzson, P., Pop, A., Abdelhak, K., Asghar, A., Bachmann, B., Braun, W., Bouskela, D., Braun, R., Buffoni, L., Casella, F., Castro, R., Franke, R., Fritzson, D., Gebremedhin, M., Heuermann, A., Lie, B., Mengist, A., Mikelsons, L., Moudgalya, K., … Ostlund, P. (2020). The OpenModelica integrated environment for modeling, simulation, and model-based development. *Modeling, Identification and Control*, *41*(4), 241–285. https://doi.org/10.4173/MIC.2020.4.1

Functional Mockup Interface. (2018). *Functional Mock-up Interface*. https://fmi-standard.org/

Güne¸, A., Baydin, G., Pearlmutter, B. A., & Siskind, J. M. (2018). Automatic Differentiation in Machine Learning: a Survey. In *Journal of Machine Learning Research* (Vol. 18).

Henze, M., Gujer, W., Mino, T., & van Loosedrecht, M. (2015). Activated Sludge Models ASM1, ASM2, ASM2d and ASM3. In *Water Intelligence Online* (Vol. 5, Issue 0). https://doi.org/10.2166/9781780402369

Hernández-del-Olmo, F., Gaudioso, E., Dormido, R., & Duro, N. (2016). Energy and Environmental Efficiency for the N-Ammonia Removal Process in Wastewater Treatment Plants by Means of Reinforcement Learning. *Energies*, *9*(9), 755. https://doi.org/10.3390/en9090755

Hernández-del-Olmo, F., Gaudioso, E., Dormido, R., & Duro, N. (2018). Tackling the start-up of a reinforcement learning agent for the control of wastewater treatment plants. *Knowledge-Based Systems*, *144*, 9–15. https://doi.org/10.1016/j.knosys.2017.12.019

Hernández-del-Olmo, F., Gaudioso, E., Duro, N., & Dormido, R. (2019). Machine learning weather soft-sensor for advanced control of wastewater treatment plants. *Sensors (Switzerland)*, *19*(14), 1–12. https://doi.org/10.3390/s19143139

Hernández-del-Olmo, F., Llanes, F. H., & Gaudioso, E. (2012). An emergent approach for the control of wastewater treatment plants by means of reinforcement learning techniques. *Expert Systems with Applications*, *39*(3), 2355–2360. https://doi.org/10.1016/j.eswa.2011.08.062

Holenda, B., Domokos, E., Rédey, Á., & Fazakas, J. (2008). Dissolved oxygen control of the activated sludge wastewater treatment process using model predictive control. *Computers and Chemical Engineering*, *32*(6), 1270–1278. https://doi.org/10.1016/j.compchemeng.2007.06.008

Kingma, D. P., & Ba, J. L. (2015, December 22). Adam: A method for stochastic optimization. *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings*.

Meneses, M., Concepción, H., & Vilanova, R. (2016). Joint environmental and economical analysis of wastewater treatment plants control strategies: A benchmark scenario analysis. *Sustainability (Switzerland)*, *8*(4), 360. https://doi.org/10.3390/su8040360

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). *Playing Atari with Deep Reinforcement Learning*. http://arxiv.org/abs/1312.5602

OpenAI, :, Berner, C., Brockman, G., Chan, B., Cheung, V., Dębiak, P., Dennison, C., Farhi, D., Fischer, Q., Hashme, S., Hesse, C., Józefowicz, R., Gray, S., Olsson, C., Pachocki, J., Petrov, M., Pinto, H. P. d. O., Raiman, J., … Zhang, S. (2019). *Dota 2 with Large Scale Deep Reinforcement Learning*. http://arxiv.org/abs/1912.06680

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., … Chintala, S. (2019). PyTorch: An Imperative Style, High-Performance Deep Learning Library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d\textquotesingle Alché-Buc, E. Fox, & R. Garnett (Eds.), *Advances in Neural Information Processing Systems 32* (pp. 8024–8035). Curran Associates, Inc. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

Safoniuk, M. (2004). Wastewater Engineering: Treatment and Reuse (Book). In *Chemical engineering* (Issue 7, pp. 10–11).

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., & Hassabis, D. (2016). Mastering the game of Go with deep neural networks and tree search. *Nature*, *529*. https://doi.org/10.1038/nature16961

Silver, D., Singh, S., Precup, D., & Sutton, R. S. (2021). Reward is enough. *Artificial Intelligence*, *299*, 103535. https://doi.org/10.1016/j.artint.2021.103535

Sutton, R. S., & Barto, A. G. (2018). *Reinforcement Learning: An Introduction. Second edition* (2nd ed.). A Bradford Book, The MIT Press. http://incompleteideas.net/book/bookdraft2018jan1.pdf

Takács, I., Patry, G. G., & Nolasco, D. (1991). A dynamic model of the clarification-thickening

process. *Water Research*, *25*(10), 1263–1271. https://doi.org/10.1016/0043-1354(91)90066-Y

United Nations. (2015). The Millennium Development Goals Report. *United Nations*, 72. https://doi.org/978-92-1-101320-7

van Hasselt, H., Guez, A., & Silver, D. (2015). Deep Reinforcement Learning with Double Q-learning. *30th AAAI Conference on Artificial Intelligence, AAAI 2016*, 2094–2100. http://arxiv.org/abs/1509.06461

Vanhooren, H., Nguyen, K., Vanrolleghem, P., & Spanjers, H. (1996). *Development of a simulation protocol for evaluation of respirometry-based control strategies*.

Wang, Z., Schaul, T., Hessel, M., van Hasselt, H., Lanctot, M., & de Freitas, N. (2015). Dueling Network Architectures for Deep Reinforcement Learning. *33rd International Conference on Machine Learning, ICML 2016*, *4*, 2939–2947. http://arxiv.org/abs/1511.06581

Aplicación de técnicas de aprendizaje profundo por refuerzo para la optimización energética en plantas de tratamiento de aguas residuales mediante el control inteligente del proceso de eliminación de nitrógeno.

PRESUPUESTO

Proyecto de Fin de Grado en Ingeniería Informática

Modalidad Externo

Oscar José Pellicer Valero

Dirigido por: Félix Hernández del Olmo

Curso: 2020-2021 Convocatoria de Junio

PRECIO DETALLADO POR CAPÍTULOS

Cód.	Ud.	Descripción	Rdto.	Precio	Importe
1		Capítulo 1: Trabajo previo y formación			
1.1		*MOOC* sobre *Machine Learning*			
		Curso masivo online *Machine Learning* realizado por Stanford y ofertado por Coursera.			
	h	Graduado en Ingeniería Informática	60	0	0
	u	Certificación del curso	1	40	40
	%	Costes directos complementarios	0.02	40	0.8
			Coste total		40,8 €
1.2		*5 MOCs* sobre *Neural Networks*			
		Cursos masivos online sobre *Neural Networks* en Python (*Deep learning, Unsupervised Deep Learning, Deep Learning in Theano and Tensorflow*) ofertados por Udemy.			
	h	Graduado en Ingeniería Informática	25	0	0
	u	Precio del curso	3	15	45
	%	Costes directos complementarios	0.02	45	0.9
			Coste total		75.9 €
1.3		*MOOC* sobre aprendizaje por refuerzo			
		Curso masivo online *Practical Reinforcement Learning* realizado por HSE University y ofertado por Coursera			
	h	Graduado en Ingeniería Informática	60	0	0
	u	Precio del curso	1	40	40
	%	Costes directos complementarios	0.02	40	0.8
			Coste total		40.8 €

Cód.	Ud.	Descripción	Rdto.	Precio	Importe
2		Capítulo 2: Materiales utilizados			
2.1		Ordenador propio			
		Adquisición de un ordenador de sobremesa con las características necesarias para llevar a cabo las simulaciones de la planta.			
	u	Ordenador sobremessa con procesador AMD Ryzen 3700 y 16Gb de RAM	1	750	750
	%	Costes directos complementarios	0.02	750	15
			Coste total		765 €
2.2		Software			
		Software empleado en el proyecto			
	u	Python, Open Modelica, y otras librerías abiertas	1	0	0
	%	Costes directos complementarios	0.02	0	0
			Coste total		0 €

Cód.	Ud.	Descripción	Rdto.	Precio	Importe
3		Capítulo 3: Desarrollo del proyecto			
3.1		Preparación de BSM1 en OpenModelica			
		Aprendizaje del lenguaje OpenModelica, comprensión del modelo BSM1 y adaptación de este a las necesidades del proyecto.			
	h	Graduado en Ingeniería Informática	60	15	900
	%	Costes directos complementarios	0.02	900	18
			Coste total		918 €
3.2		Exportación de BSM1 al formato FMI			
		Configuración del modelo del BSM1 para su exportación, aprendizaje de la librería OMPython y utilización para la exportación final del FMU usando Python.			
	h	Graduado en Ingeniería Informática	40	15	600
	%	Costes directos complementarios	0.02	600	12
			Coste total		612 €
3.3		Simulación del FMU desde Python			
		Aprendizaje de la librería PyFMI, configuración de esta para simular el FMU de BSM1 rápida y eficientemente, incluyendo el desarrollo de un interpolador más eficiente.			
	h	Graduado en Ingeniería Informática	60	15	900
	%	Costes directos complementarios	0.02	900	18
			Coste total		918 €
3.4		Creación de un entorno OpenAI Gym de BSM1			
		Aprendizaje de la API OpenAI Gym, y adaptación de esta al entorno BSM1			
	h	Graduado en Ingeniería Informática	100	15	1200
	%	Costes directos complementarios	0.02	1200	24
			Coste total		1224 €
3.5		Diseño de los agentes de aprendizaje por refuerzo			
		Diseño de los agentes QLearningAgent, DeepQLearningAgent, EVSarsaAgent y DumbQAgent y prueba de los mismos en el entorno BSM1 creado en el paso anterior.			
	h	Graduado en Ingeniería Informática	70	15	1050
	%	Costes directos complementarios	0.02	1050	21
			Coste total		1071 €
3.6		Entrenamiento de los agentes			
		Creación del código para el entrenamiento de los agentes, y optimización de sus hiperparámetros para lograr el mayor ahorro posible.			
	h	Graduado en Ingeniería Informática	100	15	1500
	%	Costes directos complementarios	0.02	1500	30
			Coste total		1530 €
3.7		Análisis de los resultados			
		Elaboración de representaciones y gráficas que permitan el análisis de todos los resultados generados.			
	h	Graduado en Ingeniería Informática	20	15	300
	%	Costes directos complementarios	0.02	300	6
			Coste total		306 €

Cód.	Ud.	Descripción	Rdto.	Precio	Importe
4		Capítulo 4: Redacción			
4.1		Redacción de esta memoria y este presupuesto			
		Tareas de redacción			
	u	Microsoft Office 365 (incluido en la matrícula de UNED)	1	0	0
	h	Graduado en Ingeniería Informática	120	15	1800
	%	Costes directos complementarios	0.02	1800	36
			Coste total		1836 €

PRESUPUESTO TOTAL DEL TFG

RESUMEN		Importe (€)
Capítulo 1:	Trabajo previo	157.50
Capítulo 2:	Materiales utilizados	765.00
Capítulo 3:	Desarrollo del proyecto	6579.00
Capítulo 4:	Redacción	1836.00
TOTAL EJECUCIÓN MATERIAL		9337.50
Gastos generales (13%)		1213.88
Beneficio industrial (6%)		560.25
TOTAL EJECUCIÓN POR CONTRATA		11111.63
IVA (21%)		2333.44
PRESUPUESTO TOTAL		13445.07

El presupuesto total asciende a:

TRECEMIL CUATROCIENTOS CUARENTA Y CINCO EUROS Y SIETE CÉNTIMOS

Aplicación de técnicas de aprendizaje profundo por refuerzo para la optimización energética en plantas de tratamiento de aguas residuales mediante el control inteligente del proceso de eliminación de nitrógeno.

ANEXO I: Listado de código

Proyecto de Fin de Grado en Ingeniería Informática

Modalidad Externo

Oscar José Pellicer Valero

Dirigido por: Félix Hernández del Olmo

Curso: 2020-2021 Convocatoria de Junio

Todo el código se encuentra disponible en: https://github.com/OscarPellicer/BSM1_gym

utils.py

```
# Copyright 2021 Oscar José Pellicer Valero
#
# Permission is hereby granted, free of charge, to any person obtaining a copy of this software and
# associated documentation files (the "Software"), to deal in the Software without restriction,
# including without limitation the rights to use, copy, modify, merge, publish, distribute,
# sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is
# furnished to do so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be included in all copies or
# substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING
# BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
# NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM,
# DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
# OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

'''
This file contains some utilities that do not match any concrete category
'''
import numpy as np
from pyfmi.common import core
def patch_pyfmi_interpolator():
    '''
        Running this function, the internal pyFMI interpolator is hot-patched to use
        TrajectoryInterpolation instead
    '''
    core.TrajectoryLinearInterpolation= TrajectoryInterpolation

from scipy import interpolate
class TrajectoryInterpolation(core.Trajectory):
    def __init__(self, abscissa, ordinate, *args, kind='slinear', extrapolate=True, **kwargs):
        '''
            General interpolator for use within the PyFMI library

            Parameters
            ---------
            abscissa: array
                1D numpy array of T time steps
            ordinate: array
                T x F (features) numpy array with the values of the F features at each
                time step provided in `abscissa`
            kind: str, default 'slinear'
                Specifies the kind of interpolation as a string or as an integer
                specifying the order of the spline interpolator to use. The string has
                to be one of ‘linear’, ‘nearest’, ‘nearest-up’, ‘zero’, ‘slinear’,
                ‘quadratic’, ‘cubic’, ‘previous’, or ‘next’. ‘zero’, ‘slinear’,
                ‘quadratic’ and ‘cubic’ refer to a spline interpolation of zeroth,
                first, second or third order; ‘previous’ and ‘next’ simply return
                the previous or next value of the point; ‘nearest-up’ and ‘nearest’
                differ when interpolating half-integers (e.g. 0.5, 1.5) in that
                ‘nearest-up’ rounds up and ‘nearest’ rounds down. Default is ‘linear’.
            extrapolate: bool, default True
                If True, perform extrapolation. If False, hold first/last value

            NOTE: To keep the original TrajectoryLinearInterpolation behaviour, set:
            kind='linear' & extrapolate=False. Different setups lead to different speeds
            and behaviours depeding on the problem and the data.
        '''
        #Check inputs
        if abscissa.shape[0] == 0:
            raise ValueError('Inputs are empty')
        elif abscissa.shape[0] == 1:
            self._f= lambda x: ordinate
        else:
            if extrapolate:
                self._f= interpolate.interp1d(abscissa, ordinate, kind=kind, copy=False,
                                              axis=0, assume_sorted=True, fill_value='extrapolate')
            else:
                fill_value= (ordinate[0], ordinate[-1])
```

```
                self._f= interpolate.interp1d(abscissa, ordinate, kind=kind, copy=False,
                                              axis=0, assume_sorted=True, bounds_error=False,
                                              fill_value=fill_value)
        super().__init__(abscissa, ordinate, *args, **kwargs)

    def eval(self, t):
        """
            Evaluate the trajectory at the specifed abscissa(s).

            Parameters
            ----------
            t: array, int
                1D numpy array, or scalar number, containing N abscissa value(s).

            Returns
            -------
            2D N x F array containing the ordinate values corresponding to t.
        """
        t_arr= np.array(t, copy=False)
        return self._f(t_arr if t_arr.ndim == 2 else t_arr[None])

class MatHandler():
    '''
        Very simple class for reading and plotting the results from a Modelica .mat file
    '''
    def __init__(self, model_name):
        '''
        Parameters
        ----------
        model_name: str
            Name of the .mat file
        '''
        from DyMat import DyMatFile
        file_name= model_name# + '_res.mat'
        self.mat= DyMatFile(file_name)

        print(' > Read:', file_name)

    def plot(self, var_names=None, plot_indep=True):
        '''
            Plots the chosen variables `var_names`

            Parameters
            ----------
            var_names: list or None, default None
                List of variable names to plot, or None to plot all
            plot_indep: bool, default True
                If True, each plot is produced in a new Figure

        '''
        import matplotlib.pyplot as plt
        if var_names is None:
            var_names= self.mat.names()
        if not plot_indep:
            plt.figure()
        for block, names_in_block in self.mat.sortByBlocks(self.mat.names()).items():
            for name in names_in_block:
                if name in var_names:
                    t= self.mat.abscissa(block, valuesOnly=True)
                    description, data= self.mat.description(name), self.mat.data(name)
                    if plot_indep:
                        plt.figure()
                    plt.plot(t, data, label=name + ': '+ description)
                    plt.ylabel(name)
                    plt.xlabel('t')
                    plt.title(description + ' (Block %d)'%block)
        if not plot_indep:
            plt.legend()

    @property
    def names(self):
        '''
            Names of all the variables
        '''
```

```
        return list(self.mat.names())

    @property
    def descriptions(self):
        '''
            Description of all the variables
        '''
        return [self.mat.description(v) for v in self.mat.names()]

class Runner():
    def __init__(self, params=''):
        '''
            Simple class to run an OMPython command

            Example:
            >>> R= Runner()
            >>> R.run('translateModelFMU("BSM1.BSM1.ClosedLoop", version="2.0", fmuType="me")')

            Parameters
            ----------
            params: str, default ''
                Parameters to pass to OMCSessionZMQ initializer
        '''
        from OMPython import OMCSessionZMQ
        self.omc = OMCSessionZMQ(params)
        print(self.omc.sendExpression("getVersion()"))

    def run(self, command, print_result=True, print_command=True, print_error=True):
        '''
            Runs a command

            Parameters
            ----------
            command: str
                Command to run
            print_result: bool, default True
                Print the response
            print_command: bool, default True
                Print the command
            print_error: bool, default True
                If no result was returned, check for errors
        '''
        if print_command:
            print('\n > ' + command)
        result= self.omc.sendExpression(command)
        if print_result:
            if isinstance(result, dict):
                for k, v in result.items():
                    print('%s: %s'%(k,v))
            else:
                print(result)
            if not result and print_error:
                print('ErrorString:', self.run('getErrorString()', print_error=False))

class EasyTimer():
    '''
        Very simple timer class
    '''
    def __init__(self):
        self.reset()

    def time(self, title='Time elapsed'):
        '''
            Prints the elapsed time since the instance was created, or since the
            method reset() was last called

            Parameters
            ----------
            title: str, default 'Time elapsed'
                Message to show alongside the ellpased time
        '''
        self.last= self.current
        self.current= datetime.now()
        print('%s: %s'%(title, self.current - self.last))
```

```
    def reset(self):
        '''
            Reset the timer
        '''
        self.current= datetime.now()
        self.last= None
```

bsm1_env.py

```
# Copyright 2021 Oscar José Pellicer Valero
#
# Permission is hereby granted, free of charge, to any person obtaining a copy of this software and
# associated documentation files (the "Software"), to deal in the Software without restriction,
# including without limitation the rights to use, copy, modify, merge, publish, distribute,
# sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is
# furnished to do so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be included in all copies or
# substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING
# BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
# NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM,
# DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
# OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

'''
This contains the base ModelicaEnv, to be used by any Modelica environment that is to be
simulated using the OpenAI gym framework, as well as a particular subclass BSM1Env, for the
Modelica BSM1 environment. It also contains addional functions needed by this environment.
'''

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from gym import spaces
import gym
from pyfmi import load_fmu

from datetime import datetime
import os, abc

class ModelicaEnv(gym.Env, metaclass=abc.ABCMeta):
    def __init__(self, path, fmi_version= 2, time_step=0.1, start_time= 0.,
                 sim_options={}, sim_ncp=100, parameters={}, output_names=[],
                 action_names=[], verbose=False, reset=True):
        '''
            Superclass for all environments to be simulated with Modelica-exported FMUs.
            To create your own environment inherit this class and implement all abstract methods.

            If you need more sophisticated logic to be implemented, you can always override
            the rest of the methods in the child-class, such as:
            reset(), step(), render(), close(), seed()

            Based on: https://github.com/ucuapps/modelicagym

            Parameters
            ----------
            path: str
                Path to the fmu file
            fmi_version: int, default 2
                Version of the FMU file
            time_step: float, default 0.1
                Size of the time step: how much the simulation is advanced every time
                step() method is called
            start_time: float, default 0.
                Start time for the simulation
            sim_options: dict, default {}
                Dictionary with simulation options to pass to pyfmi's model.simulate() methods.
                See https://jmodelica.org/pyfmi/tutorial.html
            sim_ncp: int, default 2
                Number of points time points in which the results will be discretized and returned
            parameters: dict, defeault {}
                Dictionary with model parameters and their initial values
            output_names: list, default []
                List of names of variables that will represent the state of the environment
            verbose: bool, default False
                Print additional information on-screen
            reset: bool, default True
```

```
            Whether to automatically reset the environment during its instantiation.
            If set to False, reset() method will have to be manually called at a later point.
        '''
        #Load model from fmu
        self.fmi_version= fmi_version
        self.model_name = path.split(os.path.sep)[-1]
        self.verbose= verbose
        self.path= path
        self.sim_options= sim_options
        self.sim_ncp= sim_ncp

        #Parameters required by this implementation
        self.time_step = time_step
        self.parameters = parameters
        self.start_time= start_time
        self.output_names = output_names
        self.action_names= action_names

        #Initialize the model time and state
        if reset:
            self.state = self.reset()

        #OpenAI Gym requirements
        self.action_space = self._get_action_space()
        self.observation_space = self._get_observation_space()

    def render(self, **kwargs):
        '''
            OpenAI Gym API. Determines how current environment state should be rendered.
            Environments can overwrite this method to show their env.
        '''
        pass

    def close(self):
        '''
            OpenAI Gym API. Finishes the environment, closing all open files, all
            acquired locks, etc. Environments can overwrite this method if needed.
        '''
        return True

    def reset(self):
        '''
            OpenAI Gym API. Restarts environment and sets it ready for experiments.
            It does the following:
                - Resets model
                - Sets simulation start time to 0
                - Sets initial parameters of the model
                - Initializes the model
                - Sets environment class attributes, e.g. start and stop time.

            Returns
            -------
            state: array
        '''
        import gc
        gc.collect()
        self.model = load_fmu(self.path)#,log_file_name=self.get_log_file_name())

        #Handle simulation options
        self.options= self.model.simulate_options()
        self._add_sim_options(self.sim_options)
        self._add_sim_options({'ncp': self.sim_ncp})

        self.model.reset()
        if self.fmi_version == 2:
            self.model.setup_experiment(start_time=0)

        if len(self.parameters):
            self.model.set(list(self.parameters),
                           list(self.parameters.values()))
        #self.model.initialize()

        #Simulate and observe result state
        self._add_sim_options({'initialize': True})
```

```
        self.start= self.start_time
        if self.verbose:
            print(' - Intializing model from %.1f to %.1f. This may take a couple of minutes.'%(0., self.start))
        self.results = self.model.simulate(start_time=0., final_time=self.start,
                                           options=self.options, input=self._get_initial_inputs())
        self.state= np.array([self.results.final(k) for k in self.output_names])

        #Update times
        self.stop = self.start + self.time_step
        self.done = False
        self.is_reset= True

        return self.state

    def step(self, actions):
        '''
            OpenAI Gym API. Determines how one simulation step is performed for the environment.


            Parameters
            ----------
            inputs: array
                Action to be applied to model, corresponding with FMU variables
                defined in `self.action_names`

            Returns
            -------
            state: array
                (Obsevable) state of the environment
            reward: float
                Reward given away by the environment after applying the action
            done: bool
                True if the episode has finished
            results: dict
                Arbitrary dictionary containing extra information about the step
        '''
        return self._fmu_step( (self.action_names,
                                np.concatenate([np.array([0.]), action])) )

    def _fmu_step(self, inputs):
        '''
            Advances the internal FMU simulation one step, applying some `input` values to it.

            Parameters
            ----------
            inputs: tuple of two elements: a list of variable names, and an array
                The first element of the tuple contains the list of the N FMU variables whose
                values will be updated. The second element contains an array of size T*(N+1)
                with the values of those variables at T time points. The first column of the
                array must always contain the time to which a given row corresponds.
                Other input modalities are also supported. See for more information:
                https://jmodelica.org/pyfmi/tutorial.html

            Returns
            -------
            state: array
                (Obsevable) state of the environment
            reward: float
                Reward given away by the environment after applying the action
            done: bool
                True if the episode has finished
            results: dict
                Arbitrary dictionary containing extra information about the step
        '''
        if self.done:
            print('You should not call step() once the environment has already returned done=True.',
                  'You must call reset() before.')
            return self.state, self._reward_policy(), self.done, self.results

        # Simulate and observe result state
        self._add_sim_options({'initialize': False})
        self.results = self.model.simulate(start_time=self.start, final_time=self.stop,
                                           options=self.options, input=inputs)
        self.is_reset= False
```

```
        self.state= np.array([self.results.final(k) for k in self.output_names])

        # Check if episode has finished
        self.done = self._is_done()
        if not self.done:
            self.start = self.stop
            self.stop += self.time_step

        return self.state, self._reward_policy(), self.done, self.results

    def print_last_exception(self):
        import traceback
        traceback.print_tb(self.last_exception.__traceback__)

#     def get_log_file_name(self):
#         '''
#             Generates a logfile name. Currently unused
#         '''
#         log_date = datetime.utcnow()
#         log_file_name = "{}-{}-{}_{}.txt".replace('fmu', 'txt').format(
#             self.model_name, log_date.year, log_date.month, log_date.day)
#         return log_file_name

    @abc.abstractmethod
    def _get_action_space(self):
        '''
            Returns action space according to OpenAI Gym API requirements.

            Returns
            -------
            action_space: one of gym.spaces classes that describes action space
        '''
        pass

    @abc.abstractmethod
    def _get_observation_space(self):
        '''
            Returns state space according to OpenAI Gym API requirements.

            Returns
            -------
            state_space: one of gym.spaces classes that describes state space
        '''
        pass

    @abc.abstractmethod
    def _is_done(self):
        '''
            Implements the logic that determines when a session is considered to be done.

            Returns
            -------
            done: True if the session has ended
        '''
        pass

    @abc.abstractmethod
    def _get_initial_inputs(self):
        '''
            Retuns the inputs to be used for model intialization during a reset()
            See the documentation in self._fmu_step to see how these inputs
            should be structured
        '''
        pass

    def _add_sim_options(self, options):
        '''
            Internal function to add simulation options to the current
            dictionary of options at self.options
        '''
        for k,v in options.items():
            self.options[k]= v

    @abc.abstractmethod
```

```
    def _reward_policy(self):
        '''
            Computes the reward after a step().

            Returns
            -------
            reward: float
        '''
        pass

class BSM1Env(ModelicaEnv):
    def __init__(self, *args, max_time=365, **kwargs):
        '''
            Implements the BSM1 environment using the class ModelicaEnv that implements
            the OpenAI Gym API

            Parameters
            ----------
            max_time: float, default 365
                Determines number of days that the episode will last, which in turn
                defines the stopping criterion for this environment
        '''
        self.max_time= max_time
        self.WWSG= WWSourceGen()
        super().__init__(*args, **kwargs)

    def reset(self, reset_weather=True):
        if reset_weather:
            self.WWSG= WWSourceGen()
        self.WWSG.generate(self.max_time) #New weather patterns each episode
        return super().reset()

    def _is_done(self):
        '''
            Episode will run until `max_time` is reached

            Returns
            -------
            done: True if the session has ended
        '''
        return self.stop >= self.max_time

    def _get_action_space(self):
        return spaces.Box(np.array([0.0], dtype=np.float32), np.array([6.5], dtype=np.float32))

    def _get_observation_space(self):
        return spaces.Box(np.array([-1000.]*len(self.output_names), dtype=np.float32),\
                          np.array([1000.]*len(self.output_names), dtype=np.float32))

    def _get_initial_inputs(self):
        action= 1.2
        return ( self.WWSG.get_input_names() + self.action_names,
                 #lambda t: add_constant_inputs(self.WWSG.evaluate(t), action) )
                 add_constant_inputs(self.WWSG.get_inputs(0, self.start_time), action) )

    def step(self, action):
        #Convert action to inputs and peform some checks
        action= np.array(action)
        if len(action) != len(self.action_names):
            raise RuntimeError('The number of actions does not coincide with the number of action_names')

        inputs= ( self.WWSG.get_input_names() + self.action_names,
                  #lambda t: add_constant_inputs(self.WWSG.evaluate(t), action) )
                  add_constant_inputs(self.WWSG.get_inputs(self.start, self.stop), action) )

        if isinstance(inputs[1], list) and len(inputs[1]) == 0:
            raise RuntimeError('Inputs are empty!')

        #Sometimes simmulation breaks due to random errors
        #In those situations, we retry the step, and if this does not work, we just return done=True
        try:
            return super()._fmu_step(inputs)
        except Exception as e:
            self.last_exception= e
```

```
            print('Exception found during simulation: \"%s\".' %str(e),
                  'Setting simulation as done. Please call env.reset()',
                  'To access the full traceback: env.print_last_exception()')
            self.print_last_exception()
            self.done= True
            return self.state, self._reward_policy(), self.done, self.results

    def _reward_policy(self):
        '''
            Computes the reward after a step().
            It computes the daily average operation cost of the plant
            and multiplyies it times -1 to transform it into a reward

            Returns
            -------
            reward: float
        '''
        return - operation_cost(self.results, get_daily_avg=True)

def add_constant_inputs(inputs, ct_inputs):
    '''
        Utility function to add a constant input to a set of time-dependent inputs
        To learn more about the structure of the inputs, see:
        https://jmodelica.org/pyfmi/tutorial.html

        Parameters
        ----------
        inputs: array
            Inputs array of dimensions T*(N+1), which we want to expand to dimensions
            T*(N+1+len(ct_inputs)) by filling the last len(ct_inputs) columns with the
            values from `ct_inputs`
        ct_inputs: array
            List of constant inputs to add to `inputs`

        Returns
        -------
        inputs: array
            The `inputs` array with the `ct_inputs` added to the last columns and repeated
            T times.
    '''
    return np.c_[inputs, np.tile(np.array(ct_inputs), (inputs.shape[0], 1))]

class WWSourceGen():
    def __init__(self, weather_period=14, weather_probabilties={'dry':0.7, 'strm':0.1, 'rain':0.2, 'steady':0.},
                 data_path='./inputs/', verbose=False, initial_weather=(['steady']*6) + (['dry']*2), rng_seed=22):
        '''
            Set the WasteWater input values according to the current weather, which changes randomly
            with `weather_probabilities`. Depending on the probability, a different set of
            values are read from the `data_path`, where several files with different weather conditions
            are present.

            Parameters
            ----------
            weather_period: int, default 14
                During how many days is a certain kind of weather pattern sampled
            weather_probabilities: dict, default {'dry':0.7, 'strm':0.1, 'rain':0.2, 'steady':0.}
                Probabilities for sampling a weather pattern. The keys in the dictionary
                also refference to the name of the files where the values will be taken from.
                For instanca, 'dry' weather values will be taken form Inf_dry_2006.txt
            data_path: str, default './inputs/'
                Path to the .txt files where weather values asre stored
            verbose: bool, default True
                Print some information
            initial_weather: list of str, default ['steady', 'dry']
                When a new set of weather patters is randomly generated, generate the first
                weather periods to some specific weather with probability 1. For instance, by default,
                the first 14 days will have steady weather, the next 14 days will have dry weather,
                and then the weather will be random according to `weather_probabilties`
            rng_seed: int, default 22
                Seed for the random number generator
        '''
        #Read files and preprocess them
        self.rng= np.random.default_rng(rng_seed)
        self.weather_probabilties= weather_probabilties
```

```
        self.weather_period= weather_period
        self.initial_weather= initial_weather
        self.column_names= ['t','Si','Ss','Xi','Xs','Xbh','Xba','Xp','So','Sno','Snh','Snd','Xnd','Salk','Q']
        self.data_dict= {}
        for weather_condition in weather_probabilties.keys():
            df= pd.read_csv(os.path.join(data_path, 'Inf_%s_2006.txt'%weather_condition), sep='\t')
            self.data_dict[weather_condition]= df

    def get_input_names(self):
        '''
            Get names of the inputs variables that need to be updated in the FMU

            Returns
            -------
            input_names: list of str
        '''
        return self.column_names[1:] #+ ['Temperature'] #Treq

    def generate(self, days):
        '''
            Generates `days` days of random weather

            Parameters
            ----------
            days: float
                Number of days to generate weather data for. Typically we want to generate
                data for as many days as the session is expected to last, e.g. 365 days
        '''
        gen_data= []
        self.weathers= []
        for i, day in enumerate(range(0, days, self.weather_period)):

            #Get random weather for the current weather period
            if len(self.initial_weather) > i:
                current_weather= self.initial_weather[i]
            else:
                current_weather= list(self.weather_probabilties.keys())[self.rng.choice(
                    len(self.weather_probabilties), p=list(self.weather_probabilties.values()))]
            self.weathers.append((day, current_weather))

            #Get weather data. For now, the first values are always used
            last_idx= np.argmin(np.abs(self.data_dict[current_weather]['t'].values - self.weather_period)) + 1
            data= self.data_dict[current_weather].iloc[:last_idx].values.copy()
            data[:,0]+= day #Offset time appropiately
            gen_data.append(data)

        self.data= np.concatenate(gen_data, axis=0)
        #self.data= add_constant_inputs(self.data, [15.]) #Treq

        #We also create a functional evaluator at any given time, so that both options are available
        #This, however, is much slower!
        from scipy import interpolate
        self._f= interpolate.interp1d(self.data[:,0], self.data[:,1:], kind='linear', copy=False,
                                      axis=0, assume_sorted=True, fill_value='extrapolate')

    def plot(self, days=10000, var='Q'):
        '''
            Plots the value of some weather variable `var` for `days` days

            Parameters
            ----------
            days: float or None, default None
                Number of days to plot. If None, plot all generated days
            var: str, default 'Q'
                Weather variable to plot
        '''
        plt.figure(figsize=(20,7))
        last_idx=  ( np.argmin(np.abs(self.data[:,0] - days)) + 1 )
        x= self.data[:last_idx, 0]
        y= self.data[:last_idx, self.get_input_names().index('Q') + 1]
        plt.plot(x, y); plt.title(var); plt.xlabel('t (days)')

        for day, weather in self.weathers:
            if day > days: break
```

```
            plt.text(day, (plt.ylim()[1] - plt.ylim()[0])*.95, weather)

    def get_inputs(self, t0=0, t1=1):
        '''
            Returns a matrix of generated weather input variables between t0 and t1.
            Two additional samples are always sampled at the margins just outside [t0, t1]
            to allow for full linear interpolation in the [t0, t1] range

            Parameters
            ----------
            t0: float, default 0
                Start time

            t1: float, default 0
                End time

            Returns
            -------
            data: array
                Weather input values between `t0` and `t1`
        '''
        first_idx=  np.max( [np.argmin(np.abs(self.data[:,0] - t0)) - 1, 0] )
        last_idx=  np.min( [np.argmin(np.abs(self.data[:,0] - t1)) + 1, len(self.data)] )
        #if last_idx == first_idx: last_idx+= 1#If there is a single data point
        return self.data[first_idx:last_idx]

    def evaluate(self, t):
        '''
            Linearly interpolates the weather at a time `t`. For simulation, this is slower
            than using the get_inputs() method, so it is discouraged for that application

            Parameters
            ----------
            t: float
                Time at which we want to know the weather input values

            Returns
            -------
            data: array
                Wwather input values at time `t`
        '''
        t_arr= np.array(t, copy=False)
        return self._f(t_arr if t_arr.ndim == 2 else t_arr[None])

def operation_cost(results, get_daily_avg=False):
    '''
        Computes the operation cost of the BSM1 plant using the results Object
        obtained as a result of performing a simulation step on the BSM1 FMU

        Parameters
        ----------
        results: instance of PyFMI.common.algorithm_drivers.ResultBase
            PyFMI simulation results of the BSM1 FMU
        get_daily_avg: bool, default True
            If True, returns the operation cost as a daily average.
            If False, 7 array of instantaneous daily costs accounting for different
            operation costs will be returned

        Returns
        -------
        costs: float or tuple of 7 arrays
            If get_daily_avg == True, returns the operation cost as a daily average.
            Ifget_daily_avg ==  False, returns 7 arrays of instantaneous daily costs
            accounting for different operation costs. The first array contains the
            actual instantaneous total costs, the second aray contains the exponentially-
            weighted avareage cost, and the rest are several internal costs (please
            refer to the implemententatio for more information)
    '''
    #Constants
    TARIFA_VALLE, TARIFA_LLANO, TARIFA_PUNTA = 0.073477, 0.102651, 0.122547
    IVA = 21 + 5
    electricity_cost= 0.1
    sludge_cost = 0.5 #euros/Kg
```

```
    NH_alpha, TN_alpha= 4, 2.7 #eur/kg
    NH_beta, TN_beta= 2.7 / 1000, 1.4 / 1000 #eur/1000 m3 -> eur/m3
    NH_beta_delta, TN_beta_delta= 12, 8.1 #eur/kg
    NH_lim, TN_lim= 4 / 1000, 12 / 1000 #mg/l -> kg/m3

    #Time
    time= results['time']
    day = time.astype(np.int)
    hour = 24 * (time - day)

    #Energy
    AE = results['tank3.AE'] + results['tank4.AE'] + results['tank5.AE']
    PE = 0.004 * results['ADsensor_Recycle.In.Q'] + \
         0.008 * results['ADsensor_Return.In.Q'] + \
         0.050 * results['ADsensor_Waste.In.Q']
    ME = results['tank1.ME'] + results['tank2.ME']

    #SP
    #TSS is in mg/l -> kg/m3 and Q in #m3/time
#     SP= (results['ADsensor_Waste.TSS'] * results['ADsensor_Waste.In.Q'] +
#          results['ADsensor_Effluent.TSS'] * results['ADsensor_Effluent.In.Q']) / 1000
    SP= (results['ADsensor_Waste.TSS'] + results['ADsensor_Effluent.TSS']) / 1000
    energy = AE + PE + ME

    #Effluent fines
    Q_eff= results['ADsensor_Effluent.In.Q'] #Effluent flow rate: m3/time
    NH_eff= results['ADsensor_Effluent.In.Snh'] / 1000 #Amonia concentration: mg/l -> kg/m3
    TN_eff= results['ADsensor_Effluent.Ntot'] / 1000 #Total nitrogen: mg/l -> kg/m3
    EF= Q_eff * (
            (NH_eff <= NH_lim) * ( NH_alpha * NH_eff ) +\
            (NH_eff > NH_lim)  * ( NH_alpha * NH_lim + NH_beta + NH_beta_delta * (NH_eff - NH_lim) ) +\
            (TN_eff <= TN_lim) * ( TN_alpha * TN_eff ) +\
            (TN_eff > TN_lim)  * ( TN_alpha * TN_lim + TN_beta + TN_beta_delta * (TN_eff - TN_lim) ) \
            )

    #Electricity
#     electricity_cost = (1 + IVA / 100) * (TARIFA_LLANO * ( (hour >= 8) & (hour < 18) | (hour >= 22) ) +\
#                                           TARIFA_PUNTA * ( (hour >= 18) & (hour < 22) ) +\
#                                           TARIFA_VALLE * (hour < 8) )

    #Total
    operation_cost= electricity_cost * energy + sludge_cost * SP + EF
    if get_daily_avg:
        return operation_cost.mean() #(results['time'][-1]-results['time'][0])
    else:
        operation_cost_m= pd.Series(operation_cost).ewm(span=30.5/(results['time'][1]-
results['time'][0])).mean().values
        return (operation_cost, operation_cost_m,
                electricity_cost*AE, electricity_cost*PE, electricity_cost*ME, sludge_cost*SP, EF)
```

agents.py

```
# Copyright 2021 Oscar José Pellicer Valero
#
# Permission is hereby granted, free of charge, to any person obtaining a copy of this software and
# associated documentation files (the "Software"), to deal in the Software without restriction,
# including without limitation the rights to use, copy, modify, merge, publish, distribute,
# sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is
# furnished to do so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be included in all copies or
# substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING
# BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
# NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM,
# DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
# OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

'''
This file contains the implementation of some of the most common Q-like agents, using
pytorch for the deep agents.
'''

import sys, os
import numpy as np
import abc
import torch
from datetime import datetime

from scipy.special import log_softmax
softmax= lambda x: np.exp(log_softmax(x)) #Normal softmax runs into precission issues

class QAgent(metaclass=abc.ABCMeta):
    def __init__(self, learning_rate, epsilon, discount, get_legal_actions,
                 name=None, path='./data'):
        """
            Base abstract class for Q-like RL agents to be used within OpenAI Gym

            Parameters
            ----------
            learning_rate: float
                How quick the agent learns new q-values, overwriting previous ones in the time-difference
                updates
            epsilon: float
                Exploration greediness
            discount: float
                Reward discount factor according to Sutton reward hypothesis
            get_legal_actions: function
                Function that takes an state as input, and retuns a list of valid actions in that state
            name: str or None, default None
                Name of the agent. If None, a unique agent name is assigned based on creation time
            path: str, default './data'
                Name of the path to save the agent (in case such functionality is implement by the agent)

            NOTE: Parameteres might have a slightly different interpretation depending on the kind of agent
        """
        self.get_legal_actions = get_legal_actions
        self.learning_rate = learning_rate
        self.epsilon = epsilon
        self.discount = discount
        self.path= path
        self.name= self.__class__.__name__ + \
            str(datetime.now())[:22].replace(':', '').replace('-', '').replace(' ', '').replace('.', '') \
                if name is None else name

    @abc.abstractmethod
    def train(self, s, a, r, s1, is_done):
        '''
            Trains the agent

            Parameters
```

```
            ----------
            s: list of 1D arrays or 1D array
                Environment state
            a: list of floats or float
                Action taken by the agent
            r: list of floats or float
                Environment reward
            s1: list of 1D arrays or 1D array
                Environment next state
            is_done: list of bool or bool
                True if the episode has finished
        '''
        pass

    @abc.abstractmethod
    def get_best_action(self, state):
        '''
            Returns the best action that the agent knows given a state

            Parameters
            ----------
            state: 1D array
                Environment state

            Returns
            -------
            action: float
        '''
        pass

    @abc.abstractmethod
    def get_action(self, state):
        '''
            Returns an action following the agent's policy given a state.

            Parameters
            ----------
            state: 1D array
                Environment state

            Returns
            -------
            action: float
        '''
        pass

    def save(self):
        '''
            Saves the agent weights.
        '''
        raise NotImplementedError()

    def load(self):
        '''
            Loads the agent weights
        '''
        raise NotImplementedError()

    @property
    def save_path(self):
        '''
            Path to the saved weights
        '''
        return os.path.join(self.path, self.name + '_weights.pkl')

class DumbQAgent(QAgent):
    def __init__(self, *args, default_action=1.2, **kwargs):
        '''
            Simplest possible agent, that always retuns the same default action
            for every state

            Parameters
            ----------
            default_action: float
```

```
                Action to be returned for every state
        '''
        self.default_action= default_action
        super().__init__(learning_rate=-1, epsilon=-1, discount=-1,
                         get_legal_actions=lambda a: [default_action], **kwargs)

    def get_action(self, s):
        return self.default_action

    def get_best_action(self, s):
        return self.default_action

    def train(self, s, a, r, s1, is_done):
        pass

    def save(self):
        pass

    def load(self):
        pass

class QLearningAgent(QAgent):
    def __init__(self, *args, learning_rate=0.5, initial_qvalue=-1, **kwargs):
        '''
            Classical tabular Q-learning agent.
            Q-table starts empty. If a given state is not already in the Q-table,
            it is automatically added and its value set to `initial_qvalue`

            Parameters
            ----------
            initial_qvalue: float, default -1
                Default inital q-value for all the states in the Q-table
        '''
        self._qvalues = {}
        self.intial_qvalue= initial_qvalue
        super().__init__(learning_rate, *args, **kwargs)

    def _check_qvalues(self, state, action):
        '''
            Checks whether a given state-value combination exists in the Q-table,
            and adds it to the table in case it does not. Then, it returns the `key`
            to be used for accessing the table. This key is needed since `state` might
            be an array, which is not directly hashable

            Parameters
            ----------
            state: 1D array
                Environment state
            action: float
                Action taken by the agent

            Returns
            -------
            key: hashable state representation
        '''
        key= np.array(state).tobytes()
        if key not in self._qvalues:
            self._qvalues[key]= {}
        if action not in self._qvalues[key]:
            self._qvalues[key][action]= self.intial_qvalue
        return key

    def get_qvalue(self, state, action):
        '''
            Gets a q-value from the Q-table

            Parameters
            ----------
            state: 1D array
                Environment state
            action: float
                Action taken by the agent

            Returns
```

```
        -------
        q_value: float
    '''
    key= self._check_qvalues(state, action)
    return self._qvalues[key][action]

def set_qvalue(self, state, action, value):
    '''
        Sets a q-value into the Q-table

        Parameters
        ----------
        state: 1D array
            Environment state
        action: float
            Action taken by the agent
        value: float
            Value to set the indexed Q-value to
    '''
    key= self._check_qvalues(state, action)
    self._qvalues[key][action] = value


def get_value(self, state):
    '''
        Computes the agent's estimate of V(s) using current q-values
        such that: V(s) = max_over_actions Q(state,action)

        Parameters
        ----------
        state: 1D array
            Environment state

        Returns
        -------
        value: float
    '''
    possible_actions = self.get_legal_actions(state)
    if len(possible_actions) == 0:
        raise RuntimeError('No possible actions at state:', state)
    return self.get_qvalue(state, self.get_best_action(state))

def train(self, state, action, reward, next_state, is_done):
    '''
        Updates the Q-table using the time-difference update:
        Q(s,a) := (1 - alpha) * Q(s,a) + alpha * (r + gamma * V(s'))

        Parameters
        ----------
        s: list of 1D arrays or 1D array
            Environment state
        a: list of floats or float
            Action taken by the agent
        r: list of floats or float
            Environment reward
        s1: list of 1D arrays or 1D array
            Environment next state
        is_done: list of bool or bool
            True if the episode has finished
    '''
    for s, a, r, sn, done in zip(state, action, reward, next_state, is_done):
        q= (1 - self.learning_rate) * self.get_qvalue(s, a) +\
              self.learning_rate * (r + self.discount * self.get_value(sn))
        if np.isnan(q):
            raise RuntimeError('Agent tried to update a value as NaN for state, '+\
                               'action, reward, next_state, is_done:', s, a, r, sn, done)
        self.set_qvalue(s, a, q)

def get_best_action(self, state):
    possible_actions = self.get_legal_actions(state)
    if len(possible_actions) == 0:
        raise RuntimeError('No possible actions at state:', state)

    return possible_actions[np.argmax([self.get_qvalue(state, action) for action in possible_actions])]
```

```
    def get_action(self, state):
        '''
            Compute the action to take in the current state, including exploration using
            the epsilon-greedy policy: With probability self.epsilon, take a random action,
            otherwise take the best action

            Parameters
            ----------
            state: 1D array
                Environment state

            Returns
            -------
            action: float
        '''
        possible_actions = self.get_legal_actions(state)
        if len(possible_actions) == 0:
            raise RuntimeError('No possible actions at state:', state)
        return np.random.choice(possible_actions) if np.random.random() < self.epsilon \
                                                  else self.get_best_action(state)

    def save(self):
        import pickle
        pickle.dump(self._qvalues, open(self.save_path,'wb'))

    def load(self):
        import pickle
        self._qvalues= pickle.load(open(self.save_path,'rb'))

class DeepQLearningAgent(QAgent):
    def __init__(self, network, *args, learning_rate=1e-4, **kwargs):
        '''
            Deep Q-Learning agent implemented using pytorch and updated using gradient descent

            Parameters
            ----------
            network: subclass of torch.nn.Module
                Neural network that takes a state as input, and produces the Q-values associated
                with all possible actions for that state as output
        '''
        self.network= network
        self.optimizer= torch.optim.Adam(network.parameters(), lr=learning_rate)
        super().__init__(learning_rate, *args, **kwargs)

    def get_value(self, state):
        '''
            Computes the agent's estimate of V(s) using current q-values
            such that: V(s) = max_over_actions Q(state,action)

            Parameters
            ----------
            state: N-D array
                Environment state(s)

            Returns
            -------
            values: pytroch array of float(s)
        '''
        possible_actions= self.get_legal_actions(state)
        if len(possible_actions) == 0:
            raise RuntimeError('No possible actions at state:', state)
        q_values = self.network(state)#.detach().numpy()
        #We need to get the values, because max also returns the positions
        return torch.max(q_values, dim=-1).values

    def train(self, states, actions, rewards, next_states, is_done):
        '''
            Updates the weigths of the Neural Network using the time-difference scheme
            and gradient descent. Gradient is not allowed to flow for the target Q-values
            as is common for stability

            Parameters
            ----------
```

```
            states: list of 1D arrays or 1D array
                Environment state
            actions: list of floats or float
                Action taken by the agent
            rewards: list of floats or float
                Environment reward
            next_states: list of 1D arrays or 1D array
                Environment next state
            is_done: list of bool or bool
                True if the episode has finished
        '''
        #Extract action indices
        actions_indices= np.array([np.where(np.abs(self.get_legal_actions(None)-a) < 0.001) for a in
actions]).flatten()

        #Everything to torch
        states = torch.tensor(states, dtype=torch.float32)    # shape: [batch_size, state_size]
        actions_indices = torch.tensor(actions_indices, dtype=torch.long)    # shape: [batch_size]
        rewards = torch.tensor(rewards, dtype=torch.float32)  # shape: [batch_size]
        next_states = torch.tensor(next_states, dtype=torch.float32) # shape: [batch_size, state_size]
        is_done = torch.tensor(is_done, dtype=torch.uint8)  # shape: [batch_size]

        #Get q-values for chosen actions
        predicted_qvalues = self.network(states) # [batch_size, n_actions]
        predicted_qvalues_for_actions = predicted_qvalues[range(states.shape[0]), actions_indices] # [batch_size]

        #Compute V*(next_states) and "target q-values" for loss
        next_state_values= self.get_value(next_states) #[batch_size]
        assert next_state_values.dtype == torch.float32
        target_qvalues_for_actions = rewards + self.discount * next_state_values

        #At the last state we shall use simplified formula: Q(s,a) = r(s,a) since s' doesn't exist
        target_qvalues_for_actions = torch.where(is_done, rewards, target_qvalues_for_actions)

        #Mean squared error loss to minimize
        loss = torch.mean((predicted_qvalues_for_actions -
                           target_qvalues_for_actions.detach()) ** 2)

        #Train
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()

    def get_best_action(self, state):
        possible_actions= self.get_legal_actions(state)
        if len(possible_actions) == 0:
            raise RuntimeError('No possible actions at state:', state)
        state = torch.tensor(state[None], dtype=torch.float32)
        q_values = self.network(state).detach().numpy()

        return possible_actions[np.argmax(q_values)]

    def get_action(self, state):
        '''
            Compute the action to take in the current state, including exploration using
            the epsilon-greedy policy: With probability self.epsilon, take a random action,
            otherwise take the best action

            Parameters
            ----------
            state: 1D array
                Environment state

            Returns
            -------
            action: float
        '''
        possible_actions= self.get_legal_actions(state)
        if len(possible_actions) == 0:
            raise RuntimeError('No possible actions at state:', state)
        state = torch.tensor(state[None], dtype=torch.float32)
        q_values = self.network(state).detach().numpy()

        return np.random.choice(possible_actions) if np.random.random() < self.epsilon \
```

```
            else possible_actions[np.argmax(q_values)]

    def save(self):
        torch.save(self.network.state_dict(), self.save_path)

    def load(self):
        self.network.load_state_dict(torch.load(self.save_path))
                                                #map_location=torch.device('cpu')))

class EVSarsaAgent(QLearningAgent):
    def __init__(self, *args, use_softmax=True, **kwargs):
        '''
            An agent that implements Expected Value SARSA

            Parameters
            ----------
            use_softmax: bool, default True
                Use softmax function as exploration policy, instead of epsilon-greedy
                If use_softmax == True, then epsilon is the scaling constant by which
                the q-values are divided before passing them through the softmax funtion
                to produce the probabilities of every action given a state. Hence with a small
                epsilon value, the best action will almost always be chosen, and with a large
                epsilon value, all actions will have the same probability of being chosen

        '''
        self.use_softmax= use_softmax
        super().__init__(*args, **kwargs)

    def get_value(self, state):
        '''
            Returns Vpi(s) for current state under softmax or epsilon-greedy policy.
            V_{pi}(s) = sum_{over a_i} {pi(a_i | s) * Q(s, a_i)}

            Parameters
            ----------
            state: 1D array
                Environment state

            Returns
            -------
            values: float
        '''
        possible_actions = self.get_legal_actions(state)
        if len(possible_actions) == 0:
            raise RuntimeError('No possible actions at state:', state)

        q_s_a= np.array([self.get_qvalue(state, action) for action in possible_actions])
        if self.use_softmax:
            pi_s_a= softmax(q_s_a / max([self.epsilon, 0.0001]))
        else:
            best_action= self.get_best_action(state)
            pi_s_a= [(self.epsilon / len(possible_actions)) + \
                     ((1-self.epsilon) if best_action == action else 0) \
                     for action in possible_actions]
        return np.sum([pi * q for pi, q in zip(pi_s_a, q_s_a)])

    def get_action(self, state):
        '''
            Compute the action to take in the current state, including exploration using
            the either epsilon-greedy policy (if use_softmax==False) or softmax policy

            Parameters
            ----------
            state: 1D array
                Environment state

            Returns
            -------
            action: float
        '''
        possible_actions = self.get_legal_actions(state)
        if len(possible_actions) == 0:
            raise RuntimeError('No possible actions at state:', state)
```

```
        if self.use_softmax:
            q_s_a= np.array([self.get_qvalue(state, action) for action in possible_actions])
            pi_s_a= softmax(q_s_a  / max([self.epsilon, 0.0001]))
            return np.random.choice(possible_actions, p=pi_s_a)
        else:
            return np.random.choice(possible_actions) if np.random.random() < self.epsilon \
                                                      else self.get_best_action(state)
```

wrappers.py

```
# Copyright 2021 Oscar José Pellicer Valero
#
# Permission is hereby granted, free of charge, to any person obtaining a copy of this software and
# associated documentation files (the "Software"), to deal in the Software without restriction,
# including without limitation the rights to use, copy, modify, merge, publish, distribute,
# sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is
# furnished to do so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be included in all copies or
# substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING
# BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
# NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM,
# DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
# OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

'''
This contains some useful wrappers for OpenAI environments to help integrate them
with different kinds of agents.
'''

import numpy as np
from gym.core import ObservationWrapper

class StateHolder(ObservationWrapper):
    def __init__(self, env, hold_last_N=1):
        '''
            Holds the observations of the last `hold_last_N` time steps
            and makes them available as part of the state.

            Parameters
            ----------
            env: OpenAI gym environment instance
                The environment over which this wrapper acts
            hold_last_N: int
                Number of previous time steps to hold state values for
        '''
        super(StateHolder, self).__init__(env)
        self.N= hold_last_N
        self.states=[np.zeros(env.observation_space.shape, dtype=np.float32)] * hold_last_N

    def observation(self, state):
        self.states.append(state)
        new_state= np.concatenate(self.states)
        if len(self.states) > self.N:
            del self.states[0]
        return new_state

class Normalizer(ObservationWrapper):
    def __init__(self, env, means=None, stds=None, period_auto=14, running=False):
        '''
            Standardizes the observations by substracting mean and dividing by std.
            Means and stds can be provided beforehand, or be automatically calculated
            from the data (using a static approximation, or using a running approximation)

            Parameters
            ----------
            env: OpenAI gym environment instance
                The environment over which this wrapper acts
            means: array or None, default None
                State means, must have the same shape as the state
            stds: array or None, default None
                State stds, must have the same shape as the state
            period_auto: float, default 14
                Period (in time units, not in time steps), over which previous estimations
                are forgotten (only affect 1% of the current estimations). Only applicable
                when `running` == True
            running: bool, default False
                Use a running estimation of the mean and std. If True, data after `period_auto`
```

```
                time units only has a 1% influence on current estimations. Otherwise, each
                new sample at a time step t has an influence 1/t over the whole estimation.
        '''
        super(Normalizer, self).__init__(env)
        self.means, self.stds, self.auto= means, stds, means is None
        if self.auto:
            alpha= 0.01**(self.time_step/period_auto)
            stat_shape=self.env.observation_space.shape[0]
            self.rs= RunningStats(stat_shape=stat_shape, alpha=alpha, running=running)

    def observation(self, state):
        if self.auto:
            self.rs.push(state)
            return (state - self.rs.mean) / np.maximum(self.rs.std, 0.01)
        else:
            return (state - self.means) / self.stds

class Binarizer(ObservationWrapper):
    def __init__(self, env, valid_values=None, valid_values_auto=np.arange(-2., 2.1, 2/3)):
        '''
            Discretizes the state for use with tabular agents such as q-learning

            Parameters
            ----------
            env: OpenAI gym environment instance
                The environment over which this wrapper acts
            valid_values: list of arrays or None, default None
                List of valid values. `valid_values[i]` contains an array with all
                the valid values for `state[i]`
            valid_values_auto: array or None, default None
                List of valid values if in used over a Normalizer wrapper. In this case,
                all states share the same array of valid_values, since all have been
                standardized by the Normalizer
        '''
        super(Binarizer, self).__init__(env)
        self.valid_values= valid_values
        self.valid_values_auto= valid_values_auto
        if self.valid_values is None and not isinstance(self.env, Normalizer):
            raise ValueError('Either provide discrete valid values, or wrap this over a Normalizer')

    def _get_closest(self, vv, x):
        '''
            Returns the closest value to `x` for all the values in `vv`

            Parameters
            ----------
            vv: array
                Array with the valid values
            x: float
                Number to discretize to a value of `vv`

            Returns
            -------
            vvx: float
        '''
        return vv[np.argmin(np.abs(vv - x))]

    def observation(self, state):
        if self.valid_values is not None:
            return np.array([self._get_closest(valid_values_var, state_var)
                             for state_var, valid_values_var in zip(state, self.valid_values)])
        else:
            return np.array([self._get_closest(self.valid_values_auto, state_var) for state_var in state])

class RunningStats(object):
    def __init__(self, stat_shape, alpha=0.99, running=True):
        '''
            Computes the running mean and std of some 1D array using either a
            static approximation, or a running approximation
            Adapted from: https://gist.github.com/wassname/a9502f562d4d3e73729dc5b184db2501

            Parameters
            ----------
            stat_shape: list, tuple or array
```

```
                Shape of the 1D vector over which the statistics will be computed.
                Needed to initialize internal structures.
            alpha: float, default 0.99
                Memory factor, determines how far into the past the running statistics are able to
                remember. Only applicable when `running` == True
            running: bool, default False
                Use a running estimation of the mean and std. If True, data from the past loses quickly
                (depending on the value of alpha) influence on current estimations. Otherwise, each
                new sample at a time step t has an influence 1/t over the whole estimation.
        '''
        self.alpha, self.n, self.running= alpha, 1, running
        self.mean= np.zeros(stat_shape, dtype=np.float32)
        self.var= np.ones(stat_shape, dtype=np.float32)
        if not running:
            self.varn= np.ones(stat_shape, dtype=np.float32)

    def push(self, x):
        '''
            Adds a new array `x` to update current mean and std estimations

            Parameters
            ----------
            x: array
                1D array of the data for which mean and std are being computed
        '''
        prev_mean = self.mean.copy()
        if self.running:
            self.mean = self.alpha * self.mean + (1-self.alpha) * x
            self.var = self.alpha * self.var + (1-self.alpha) *  ( (x - prev_mean) * (x - self.mean) )
        else:
            self.mean += (x - self.mean) / self.n
            self.varn += (x - prev_mean) * (x - self.mean)
            self.var= self.varn / self.n
            self.n+= 1

    @property
    def std(self):
        return np.sqrt(self.var)
```

training.py

```
# Copyright 2021 Oscar José Pellicer Valero
#
# Permission is hereby granted, free of charge, to any person obtaining a copy of this software and
# associated documentation files (the "Software"), to deal in the Software without restriction,
# including without limitation the rights to use, copy, modify, merge, publish, distribute,
# sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is
# furnished to do so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be included in all copies or
# substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING
# BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
# NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM,
# DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
# OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

'''
This file contains all the code for letting an agent interact and learn from the evironment
as well as to plot the evolution of the values of the variables over time
'''

import random
import tqdm
import pandas as pd
import numpy as np
from tqdm import trange
from datetime import datetime
import matplotlib.pyplot as plt
from IPython.display import clear_output

def moving_average(x, span=7*96):
    '''
        Computes the exponentially moving average filter of `x` for a `span` of days.

        Parameters
        ----------
        x: array
            Numpy array to be filtered
        span: int
            Specify decay (α) of the exponentially moving average in terms of
            span: α=2/(span+1), for span >= 1

        Returns
        -------
        filtered_x: array
    '''
    return pd.DataFrame({'x': np.asarray(x)}).x.ewm(span=span).mean().values

def play_episode(env, agent, train=True, reward_scale=1., plot=True, title='',
                 combine_result=True, replay_buffer=None, rb_batch=10,
                 epsilon_scheduler=lambda eps, t, t_max: eps):
    '''
        Plays an episode of the evironment `env` with the `agent` interacting in it.

        Parameters
        ----------
        env: Instance of OpenAI Gym Environment subclass
            Environment to which the `agent` will be exposed
        agent: Instance of utils.QAgent subclass
            Agent that will explore the `env`
        train: bool, default True
            If True, the agent is to be trained during the episode
        reward_scale: float, default 0.1
            Reward scale factor
        plot: bool, default True
            If True, show a plot of the agent's actions and observations every some steps
        title: str, default ''
            Title of the plot. Ignored if plot == False
        combine_results: bool, default True
```

```
            Wether to apply the combine_results function to the list of results at the end, or to return
            the full list of results instead (see combine_results for further information)
        replay_buffer: Instance of ReplayBuffer or None, default None
            Experience Replay Buffer to use, or None if no Experience Replay is to be employed
        rb_batch: int, default 10
            Size of the batch to be returned by the Experience Replay Buffer. Only applicable
            if replay_buffer is not None
        epsilon_scheduler: function, default lambda eps, t, t_max: eps
            Function to update epsilon within an episode. It takes the current time step `t`,
            the maximum number of time steps `t_max`, and current epsilon `eps` as input, and returns
            the updated epsilon value

        Returns
        -------
        mean_reward: float
            Mean reward over the whole episode
        rewards: list of floats
            List of rewards for every time step
        states: list of arrays
            List of states for every time step
        results: list of pyfmi.common.io.ResultDymola subclass instances or dict
            List of results for every time step if combine_results == True, dictionary with
            the combined results otherwise

    '''
    #Initialize
    total_reward = 0.0
    s= env.reset() #if not env.is_reset else env.state
    rewards, states, results, actions= [], [], [], []

    #Try to get the t_max automatically
    try:
        t_max=int((env.max_time-env.start_time)/env.time_step) + 1
    except:
        t_max=int(1e5)

    #Iterate
    for t in trange(t_max):
        #Get action
        if train:
            a = agent.get_action(s)
        else:
            a = agent.get_best_action(s)
        next_s, r, done, result = env.step(np.array([a]))
        r*= reward_scale

        #Check for incorrect rewards
        if r > 0 or r < -50000*reward_scale:
            raise RuntimeError('Reward outside bounds:', next_s, r, done)
            #raise RuntimeError('Reward > 0: Our agent has found an exploit?')
            return total_reward, rewards, states, results

        #Train the agent
        if train:
            agent.epsilon= epsilon_scheduler(agent.epsilon, t, t_max)
            if replay_buffer is not None:
                replay_buffer.add(s, a, r, next_s, done)
                agent.train(*replay_buffer.sample(rb_batch))
            else:
                agent.train([s], [a], [r], [next_s], [done])


        rewards.append(r); states.append(s); results.append(result); actions.append(a)
        s = next_s

        #Plot every 14 days
        every_t= int(14 / env.time_step)
        if plot and (t % every_t == 0):
            rewards_ewma = moving_average(rewards)
            idx0, idx1= np.max([0, len(rewards)-every_t]), len(rewards)

            try:
                clear_output()
                plt.figure(figsize=(20,5))
```

```
                plt.plot(rewards[idx0:idx1], label='Reward')
                plt.plot(rewards_ewma[idx0:idx1], label='Reward ewma')
                plt.legend()
                plt.grid()
                plt.title(title + (' | ' if title!='' else '') + 'eps = %.4f'%agent.epsilon)

                plt.figure(figsize=(20,5))
                plt.plot(states[idx0:idx1], label='State')
                plt.plot(actions[idx0:idx1], label='Action')
                plt.legend()
                plt.grid()
                plt.show()
            except Exception as e:
                print('Exception while plotting:', e)
                print(idx0, idx1, rewards,every_t, len(states), states[0].shape, flush=True)

        if done: break

    if combine_result:
        results= combine_results(env, results)

    return np.mean(rewards), rewards, states, results

def plot_vars(results, variables, descriptions, t0=0, t1=None, title='', ylim=None):
    '''
        Plots a the values of a variable over a given interval of time

        Parameters
        ----------
        results: dict, or pyfmi.common.io.ResultDymola subclass instance
            The results dictionary to plot, with variable names as keys
        variables: list of str
            List of variables to plot
        descriptions: list of str
            List of descriptions to plot alongside the variables
        t0: float, default 0
            Starting time for the plot
        t1: float or None, default None
            Final time for the plot. If None, take t1 as the latest time available
        title: str, defeault ''
            Title of the plot
        ylim: tuple of two floats or None, default None
            Limits of the plot in the y axis
    '''
    first_idx=  np.max( [np.argmin(np.abs(results['time'] - t0)) - 1, 0] )
    last_idx=  np.min( [np.argmin(np.abs(results['time'] - t1)) + 1, len(results['time'])] )

    plt.figure(figsize=(12,6))
    plt.title(title)
    for var, description in zip(variables, descriptions) :
        #description= res.result_data.description[res.keys().index('PI1.y')] #Find better way!
        #label= var +  ((': '+ description) if description!='' else '')
        label= description + ' (%s)'%var
        plt.plot(results['time'][first_idx:last_idx], results[var][first_idx:last_idx], label=label)
        plt.ylabel('Values')
        plt.xlabel('Time (days)')
        if ylim is not None:
            plt.ylim(ylim)
    plt.legend()

def combine_results(env, result):
    '''
        Combines a list of result objects produced by pyfmi into a dictionary by taking
        the mean of all meassurements at every time step

        Parameters
        ----------
        result: list of some pyfmi.common.io.ResultDymola subclass instances
            List of the results provided by pyfmi, one result for each simulation time step

        Returns
        -------
        final_results: dict
            Python dictionary with the variable names as keys, and an array of the values associated
```

```
            with that given variable for every time step
    '''
    problematic_vars= []
    final_results= {}
    for var in tqdm.tqdm(['time'] + list(env.model.get_model_variables().keys())):
        try:
            result[0][var] #Just to see if this fails!
            final_results[var]= []
            for res in result:
                final_results[var].append(np.mean(res[var]))
                #final_results[var].append(res[var][::sampling_period])
            #final_results[var]= np.concatenate(final_results[var], axis=0)
            final_results[var]= np.array(final_results[var])
        except Exception as e:
            problematic_vars.append(var)
    if len(problematic_vars):
        print('There was a problem reading some variables. '+\
              'Redoing the simulation without filters may fix it')
    return final_results

class ReplayBuffer(object):
    def __init__(self, size):
        '''
            Create a Replay Buffer.
            Based on  https://github.com/openai/baselines/blob/master/baselines/deepq/replay_buffer.py

            Parameters
            ----------
            size: int
                Maximum number of elements to store in the buffer. When the buffer
                overflows the old memories are dropped.
        '''
        self._storage = []
        self._maxsize = size
        self._next_idx = 0

    def __len__(self):
        '''
            Returns current lenght of the buffer

            Returns
            -------
            int: lenght of the buffer
        '''
        return len(self._storage)

    def add(self, obs_t, action, reward, obs_tp1, done):
        '''
            Adds an element to the buffer

            Parameters
            ----------
            obs_t: np.array
                Observation
            action: np.array
                Action executed given obs_t
            reward: np.array
                Reward received as results of executing action
            obs_tp1: np.array
                Observation seen after executing action
            done: np.array
                True if executing action resulted in the end of an episode
        '''
        data = (obs_t, action, reward, obs_tp1, done)

        if self._next_idx >= len(self._storage):
            self._storage.append(data)
        else:
            self._storage[self._next_idx] = data
        self._next_idx = (self._next_idx + 1) % self._maxsize

    def sample(self, batch_size):
        '''
            Sample a batch of experiences.
```

```
            Parameters
            ----------
            batch_size: int
                How many transitions to sample.

            Returns
            -------
            obs_batch: np.array
                batch of observations
            act_batch: np.array
                batch of actions executed given obs_batch
            rew_batch: np.array
                rewards received as results of executing act_batch
            next_obs_batch: np.array
                next set of observations seen after executing act_batch
            done_mask: np.array
                done_mask[i] = 1 if executing act_batch[i] resulted in
                the end of an episode and 0 otherwise.
        '''
        idxes = [ random.randint(0, len(self._storage) - 1)
                  for _ in range(batch_size) ]
        obses_t, actions, rewards, obses_tp1, dones = [], [], [], [], []
        for i in idxes:
            data = self._storage[i]
            obs_t, action, reward, obs_tp1, done = data
            obses_t.append(np.array(obs_t, copy=False))
            actions.append(np.array(action, copy=False))
            rewards.append(reward)
            obses_tp1.append(np.array(obs_tp1, copy=False))
            dones.append(done)
        return (np.array(obses_t), np.array(actions), np.array(rewards),
                np.array(obses_tp1), np.array(dones) )
```

Nota: aquí solo se muestra el código de todas las celdas de `Training.ipynb`. En el repositorio también se puede encontrar una versión no interactiva en HMTL que permite una visualización más rica de esta, sin requerir la instalación de Jupyter Notebook.

Training.ipynb

```
# Autoreload modules if code changes
%load_ext autoreload
%autoreload 2

#Import from basic libraries
import sys, os
import numpy as np
import matplotlib.pyplot as plt
from datetime import datetime
import pickle
import torch.nn as nn
import gym
from gym.envs.registration import register

#Import from custom libraries
from training import play_episode, ReplayBuffer, plot_vars, combine_results
from agents import QLearningAgent, DeepQLearningAgent, DumbQAgent, EVSarsaAgent
from wrappers import Normalizer, Binarizer, StateHolder

#Import from BSM1 environment
sys.path.append('BSM1Envs')
from bsm1_env import operation_cost

#-----------------------------------------------------#

#FMU configuration
EXPORT_FMU= False
base_path= '../WasteWaterResearch'
model_name= 'BSM1' #'BSM1.BSM1_ClosedLoop_Sensor_OperatorTest4'

#THe BSM1 environment requires some careful setting up from OpenModelica to export correctly to an FMU
if EXPORT_FMU:
    from utils import Runner
    R= Runner()
    R.run('loadModel(Modelica,{"3.2.3"},true,"",false)')
    R.run('loadFile("%s/WasteWater/package.mo","UTF-8",true,true,false)'%base_path)
    R.run('loadFile("%s/BSM1/package.mo","UTF-8",true,true,false)'%base_path)
    R.run('disableNewInstantiation()')
    R.run('translateModelFMU(%s, version="2.0", fmuType="me")'%model_name)

#-----------------------------------------------------#

PATCH_INTERPOLATOR= True
if PATCH_INTERPOLATOR:
    from utils import patch_pyfmi_interpolator
    patch_pyfmi_interpolator()

#-----------------------------------------------------#

#Time-related configuration
start_time= 14*8 #During the first 30 days the plant stabilizes. This time is simulated during env.reset()
time_step= 1./24./4. #Simulation time step when env.step() is called (days)
max_time= start_time + 365 #Total simulation time (days)
verbose= True
discount_factor_period= 14 #After how many days the reward is multiplied by 0.01
discount_factor= 0.01**(time_step/discount_factor_period)
action_names= ['agent_action']

#Define model and env names
env_name = 'BSM1Env-v0'
entry_point= 'BSM1Envs:BSM1Env'
output_names= ['limiter1.y', 'sensor_class_a1.y']
#output_names+= ['ADsensor_Waste.TSS']
```

```
#Delete registered environment in case it exists to avoid errors if this cell is run multiple times
if env_name in gym.envs.registry.env_specs:
    del gym.envs.registry.env_specs[env_name]

#Customize simulation options
default_opts= {
    'solver':'Dopri5', #Dopri5, CVode | CVode seems to run slightly faster, but it leaks A TON of memory
    'CVode_options':
        {'discr': 'BDF',
        'iter': 'Newton',
        'atol': 1e-05,
        'rtol': 0.001,
        'maxh': 'Default',
        'external_event_detection': False,
        'linear_solver': 'DENSE', #Must always be set to dense, otherwise Assimulo complains and breaks
        'inith': 0.001,
        'verbosity': 50},
    'Dopri5_options':
        {'atol': 1e-05,
         'rtol': 0.001,
         'inith': 0.0001, #This value has a big impact in the simulation speed
        'verbosity': 50}, #QUIET = 50, WHISPER = 40, NORMAL = 30, LOUD = 20, SCREAM = 10
    'result_handling':'memory', #Keep results in memory rather than saving them into a file
}

#We will only save the simmulation results associated with some variables, the rest will be discarded
operation_cost_vars= ['tank3.AE', 'tank4.AE', 'tank5.AE',  #Cost variables
                      'ADsensor_Recycle.In.Q', 'ADsensor_Return.In.Q', 'ADsensor_Waste.In.Q',
                      'tank1.ME', 'tank2.ME', 'ADsensor_Waste.TSS', 'ADsensor_Waste.In.Q',
                      'ADsensor_Effluent.In.Q', 'ADsensor_Effluent.In.Snh', 'ADsensor_Effluent.Ntot',
                      'ADsensor_Influent.In.Q', 'ADsensor_Effluent.TSS']
default_opts['filter']= action_names + output_names + operation_cost_vars

#Define the parameters to pass to the initializtion of BSM1Env
parameters= {
         'action_names': action_names,
         'output_names': output_names,
         'parameters': {},

         'max_time': max_time,
         'time_step': time_step,
         'start_time': start_time,
         'verbose': verbose,

         'sim_options': default_opts,
         'sim_ncp': 30,
         'path': os.path.abspath('%s.fmu'%model_name),
         'reset': False,
}

#Register the environment
register(
    id=env_name,
    entry_point= entry_point,
    kwargs= parameters #BSM1Env class parameters as a dictionary
)

#Create an env instance
env= gym.make(env_name)

#-----------------------------------------------------#

#Agent configuration
AGENT= 'sarsa' #['q', 'deepq', 'sarsa', 'dumb']
EPSILON= 0.5 #0.3, 0.5
EPSILON_DISCOUNT= 0.8 #0.7, 0.8
EPISODES= 1 #10, 20
REWARD_SCALE= 1e-3 / 200
USE_SOFTMAX= True #For use with sarsa
REPLAY_BUFFER= ReplayBuffer(100) #None
BACTH_SIZE= 32
HOLD_N= 1 #If HOLD_N > 0: Use StateHolder wrapper
SCHEDULER_POWER= 0
```

```
#Within-epsidoe epsilon scheduling
EPSILON_SCHEDULER= lambda _, t, t_max: EPSILON*(1 - t/t_max)**SCHEDULER_POWER
ts= np.arange(0,365, 1)
plt.plot(ts, EPSILON_SCHEDULER(EPSILON, ts, 365));
plt.ylabel('Epsilon'); plt.xlabel('t'); plt.title('Episode-wise epsilon schedule')

#Define the set of discrete actions that the agent can take
#get_legal_actions=lambda state: np.arange(0.8, 2.3, 0.4)
get_legal_actions=lambda state: np.array([1.2, 1.5, 1.8])
print('Legal actions:', get_legal_actions(None))

#Define a network architecture, needed for deep algorithms
state_dim = np.array(env.observation_space.shape) * (HOLD_N + 1)
n_actions= len(get_legal_actions(None))

network = nn.Sequential(
            nn.Linear(state_dim[0], 50), nn.ReLU(),
            #nn.Linear(100, 100),           nn.ReLU(),
            nn.Linear(50, n_actions),
          )

#Choose the agent that we will train with
if AGENT == 'q':
    if HOLD_N > 0:
        env = StateHolder(Binarizer(Normalizer(env)), hold_last_N=HOLD_N)
    else:
        #env= Binarizer( env, valid_values=(np.arange(0., 11., 1.),  np.arange(0.5, 5., 0.5)) )
        env= Binarizer(Normalizer(env))
    agent= QLearningAgent(epsilon=EPSILON, discount=discount_factor,
                          get_legal_actions=get_legal_actions)
elif AGENT == 'deepq':
    env = StateHolder(Normalizer(env), hold_last_N=HOLD_N) if HOLD_N > 0 else Normalizer(env)
    agent= DeepQLearningAgent(network, epsilon=EPSILON, discount=discount_factor,
                              get_legal_actions=get_legal_actions, learning_rate=1e-5)
elif AGENT == 'sarsa':
    if HOLD_N > 0:
        env = StateHolder(Binarizer(Normalizer(env), valid_values_auto=np.arange(-2., 2.1, 2/3)))
    else:
        env= Binarizer( env, valid_values=(np.arange(0., 11., 1.),  np.arange(0.5, 5., 0.5)) )
    agent= EVSarsaAgent(epsilon=EPSILON, discount=discount_factor,
                        get_legal_actions=get_legal_actions, use_softmax=USE_SOFTMAX)
elif AGENT == 'dumb':
    agent= DumbQAgent()

print('Using agent:', agent.name)

#-----------------------------------------------------#

#Train configuration
TRAIN= True
LOAD_AGENT= False
SAVE_RESULTS= True
RESULTS_NAME= os.path.join('./data', agent.name + '_results')

total_rewards, rewards, states= [], [], []
time0, time1=  datetime.now(), datetime.now()

if LOAD_AGENT:
    agent.load()

if TRAIN:
    for i in range(EPISODES):
        time0= time1
        time1= datetime.now()
        title= '%d | Final reward: %.4f | Elapsed: %s'%(i, total_rewards[-1]/REWARD_SCALE if i!=0 else 0,
str(time1-time0))

        total_reward, reward, state, result= play_episode(env, agent, title=title, rb_batch=BACTH_SIZE,
                                                          reward_scale=REWARD_SCALE, replay_buffer=REPLAY_BUFFER,
                                                          epsilon_scheduler=EPSILON_SCHEDULER)
        total_rewards.append(total_reward); rewards.append(reward); states.append(state)
        agent.epsilon*= EPSILON_DISCOUNT

        #Save weights
```

```
        agent.save()

        #Save results
        if SAVE_RESULTS:
            pickle.dump(result, open(RESULTS_NAME + '_%d.pkl'%i,'wb'))

#-----------------------------------------------------#

print('- %s\t| Last training session: %.4f | Config: '%(
         agent.name, -total_rewards[-1]/REWARD_SCALE), BACTH_SIZE, EPSILON, HOLD_N,
         REPLAY_BUFFER._maxsize if REPLAY_BUFFER is not None else 'no-rb', SCHEDULER_POWER, str(output_names))
- EVSarsaAgent2021061613022656  | Last training session: 2018.9441 | Config:  32 0.5 1 100 0 ['limiter1.y',
'sensor_class_a1.y']

#-----------------------------------------------------#

#Plot configuration
DIFF_PLOT= True #Plot differencees wrt dumb
DUMB_RESULTS_NAME=  os.path.join('./data', 'DumbQAgent2021061521454183' + '_results_0.pkl')
RESULTS_NAME= os.path.join('./data', agent.name + '_results_0.pkl')
final_results= pickle.load(open(RESULTS_NAME,'rb'))
final_dumb_results= pickle.load(open(DUMB_RESULTS_NAME,'rb'))
plt.rcParams['figure.dpi'] = 250

#Plot states + actions
t0= 204
t1= t0 + 3
print('Daily average operation cost for agent %s: %.4f €'%(agent.name, operation_cost(final_results, True)))
variables= (['ADsensor_Waste.TSS'] if AGENT == 'deepq' else []) + ['sensor_class_a1.y', 'limiter1.y',
'agent_action'] + []
descriptions=   (['Total suspended solids (mg/L)'] if AGENT == 'deepq' else []) + ['NH4 (mg/l)', 'O2 (mg/l)',
'Agent Setpoint']
plot_vars(final_results, variables, descriptions, t0, t1, ylim=(0,3)); plt.ylabel('Values (standardized)');
plt.grid()
plot_vars(final_results, ['ADsensor_Effluent.In.Q'], ['Q'], t0, t1); plt.grid()

#Plot wheather
from bsm1_env import WWSourceGen
WWSG=WWSourceGen(); WWSG.generate(max_time)
WWSG.plot(10000)
plt.grid()

#Plot cost variables
t0, t1= start_time, max_time
plot_data= operation_cost(final_results)
plot_dumb_data= operation_cost(final_dumb_results)
plot_vars(final_results, ['ADsensor_Effluent.In.Q'], ['Q'], t0, t1); plt.grid()
plt.figure(figsize=(20,7))
for var, description in zip([0, 1, 2, 3, 4, 5, 6],
                    ['total operation cost', 'total operation cost mothly EWMA',
                     'AE cost', 'PE cost', 'ME cost', 'SP cost', 'EF cost']):
    first_idx=  np.max( [np.argmin(np.abs(final_results['time'] - t0)) - 1, 0] )
    last_idx=  np.min( [np.argmin(np.abs(final_results['time'] - t1)) + 1, len(final_results['time'])] )

    if DIFF_PLOT:
        cummulative= np.cumsum(- plot_data[var][first_idx:last_idx] +
plot_dumb_data[var][first_idx:last_idx])*time_step
        if var==0:
            print('Yearly savings:', cummulative[-1])
        if 'EWMA' in description: continue
        plt.plot(final_results['time'][first_idx:last_idx], cummulative,
                 label='Savings in ' + description)
        plt.title(description + ' (savings)')
        plt.ylabel('Savings (€)'); plt.xlabel('Time (days)')
    else:
        plt.figure(figsize=(20,7))
        plt.plot(final_results['time'][first_idx:last_idx], plot_data[var][first_idx:last_idx], label='Agent')
        plt.plot(final_results['time'][first_idx:last_idx], plot_dumb_data[var][first_idx:last_idx],
label='Constant setpoint')
        plt.title(description)
        plt.ylabel('Cost (€)'); plt.xlabel('Time (days)')
    plt.legend()
plt.grid()
```