Trabajo de Fin de Grado

Universidad Complutense de Madrid

Aprendizaje por refuerzo para gestión de consumo ajustando la frecuencia de ejecución en multicores

–

Setting DVFS using reinforcement learning for power management in multicores

Facultad de Informática

–

Autor: Pablo Hernández Aguado

–

Directora: Katzalin Olcoz Herrero

Codirector: Luis Mª Costero Valero

Doble Grado en Ingeniería Informática – Matemáticas

Curso 2020 – 2021

Resumen / Abstract

El trabajo plantea el uso de técnicas de aprendizaje reforzado para controlar el consumo energético del procesador mediante la modificación de las frecuencias de ejecución. Partiendo de métodos de aprendizaje cuya efectividad ya ha sido demostrada, estos se han aplicado al problema consiguiendo resultados satisfactorios. Se han entrenado agentes en entornos con distintas propiedades que han aprendido a aplicar las subidas o bajadas de frecuencia apropiadas que hacen converger al sistema al límite energético elegido. Además, se ha probado el mantenimiento de la funcionalidad de los agentes aplicados a distintas cargas de trabajo.

This work proposes the use of Reinforcement Language techniques to take control of the energetic consumption of a processor through changes in its execution frequencies. On the basis of learning methods whose effectivity has already been proved, these have been used on the problem achieving successful results. Agents have been trained over environments with different configurations and they have learned to apply the suitable frequency raises or drops to make the system converge towards the chosen energetic limit. Besides, the functionality of these agents has been shown to be conserved, even when applied to different workloads.

Palabras clave / Keywords

Agente, entorno, aprendizaje reforzado, PPO, frecuencia, potencia, RAPL.

Agent, environment, Reinforcement Learning, PPO, frequency, power, RAPL.

Repositorios

El código utilizado para la realización de esta práctica se encuentra en el repositorio de GitHub:

https://github.com/pherna06/server-consumption

Los resultados obtenidos que se presentan en la memoria se encuentran a su vez en el repositorio:

https://github.com/pherna06/server-consumption-results

Índice general

Capítulo 1

Introducción

El objetivo de este trabajo consiste en desarrollar estrategias automatizadas para imponer limitaciones energéticas específicas al procesador de un sistema, práctica conocida en inglés como *power capping*, ajustando la frecuencia de ejecución de sus núcleos. Para ello, se emplearán herramientas relativamente recientes de la disciplina de la Inteligencia Artificial. En particular, estas estarán basadas en el aprendizaje por refuerzo y el paradigma agente-entorno.

Limitación de potencia o *power capping*

El *power capping* [1] es una práctica empleada principalmente en centros de procesamiento de datos que permite limitar la potencia consumida por los servidores, lo que facilita una planificación más eficiente de estos espacios. La garantía de que, bajo una carga de trabajo intensa, la demanda de potencia no excederá nunca a la potencia disponible hace posible la instalación de un número mayor de servidores en el mismo espacio físico disponible.

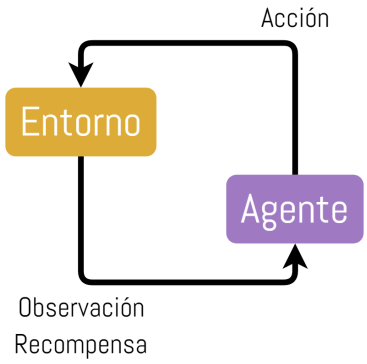
Tal es la importancia de este mecanismo que los propios fabricantes han desarrollado herramientas que implementan esta funcionalidad a nivel de *hardware*, como pueden ser el *Dynamic Power Capping* de HP y las tecnologías *Node Manager* y *RAPL* de Intel.

Aprendizaje por refuerzo y paradigma agente-entorno

El aprendizaje por refuerzo [2] es uno de los 3 modelos básicos de la Inteligencia Artificial, junto con el aprendizaje supervisado y el no supervisado. Su forma de proceder está enfocada a que un sistema determine automáticamente cuál debería ser su comportamiento ideal bajo un contexto específico, con la finalidad de maximizar su rendimiento.

Para lograr este objetivo, se emplea el paradigma agente-entorno [3], cuyo esquema se muestra en la Figura 1.1, en el cual el agente decide las acciones a realizar en el entorno en base a su estado u observaciones. Además del estado, el agente recibe del entorno un valor númerico (una *recompensa*) en función del estado del entorno tras aplicar una acción, lo que le permite *aprender* qué acciones debe tomar para maximizar la recompensa acumulada a largo plazo.

Figura 1.1
Paradigma agente-entorno.



Motivación para el empleo del aprendizaje por refuerzo

Los métodos del aprendizaje por refuerzo son apropiados para problemas complejos, en los que no parece haber una solución obvia. En este caso particular, se podría plantear una solución directa con la siguiente estrategia:

- Permitir subidas de frecuencia mientras se esté por debajo del límite energético.
- Forzar bajadas de frecuencia cuando se supere el límite energético hasta que el consumo quede por debajo de él.

No obstante, el consumo energético de un procesador es una magnitud muy variable que depende de numerosos factores: la temperatura, la frecuencia de ejecución, el uso de las unidades funcionales del procesador, etc. Dado este contexto, parece apropiado el empleo del aprendizaje por refuerzo, de forma que los agentes entrenados puedan desarrollar un entendimiento más profundo de las fluctuaciones energéticas del procesador.

Estructura de la memoria

En el Capítulo 2, se introducen y ejemplifican las distintas herramientas que se emplearán a lo largo del trabajo, siendo todas ellas módulos de `Python`. Por un lado, aquellas enfocadas al aprendizaje por refuerzo y la implementación del paradigma agente-entorno:

- El módulo *GYM*, aplicado a la creación de entornos.
- El módulo *RLlib*, que implementa la mecánica del agente y numerosos algoritmos de aprendizaje reforzado.

Por otra parte, las herramientas destinadas a interpretar y modificar el estado del procesador:

- El módulo *cpufreq*, utilizado para leer y variar la frecuencia del procesador a nivel de núcleo.
- El módulo *pyRAPL*, cuya función es recopilar medidas del consumo energético del procesador.

En el Capítulo 3, se expone la implementación de un entorno *GYM* orientado al problema de *power capping*. Este proporcionará como observación una medida de la potencia del procesador y recibirá como acciones subidas y bajadas de frecuencia.

En el Capítulo 4, se presenta el análisis energético de una carga de trabajo asociada al producto de matrices. Esta misma se utilizará durante para el entrenamiento del entorno construido en el Capítulo 3, cuyos resultados se analizarán utilizando, en parte, la huella energética de esta carga de trabajo.

En el Capítulo 5, se exponen posibles modificaciones del entorno dirigidas a mejorar su rendimiento, lo que se estudiará comparando los resultados de los nuevos entornos modificados con los del capítulo anterior.

En el Capítulo 6, se considerarán otras cargas de trabajo con distinta huella energética y se pondrá a prueba el rendimiento de los agentes cuyo aprendizaje se produjo bajo la carga de trabajo asociada al producto de matrices.

En el Capítulo 7, se manifiestan las conclusiones obtenidas, así como las posibles caminos relevantes a explorar en un futuro.

Capítulo 2

Introduction

The objective of this work is to develop automatized strategies to impose specific energetic limitations to a system's CPU, a practice known as power capping, by adjusting the execution frequency of its cores. In order to achieve this, we will employ relatively recent tools from the discipline of Artificial Intelligence. In particular, these tools will be derived from Reinforcement Learning and the agent-environment paradigm.

Power capping

Power capping is a method employed mainly in data centers that allows to limit the power being consumed by the servers. This makes it easier to achieve an efficient planification of these spaces. Besides, the assurance that power demand will not exceed the available power under an intensive workload allows for the instalation of a higher number of servers in the same physical space.

Such is the importance of this method that CPU manufacturers themselves have developed tools for implementing this functionality at hardware level. Some examples of these are *Dynamic Power Capping* from HP and the *Node Manager* and RAPL technologies from Intel.

Reinforcement Learning and the agent-environment paradigm

Reinforcement Learning is one of the 3 basic models of Artificial Intelligence, together with Supervised and Unsupervised learning. Its procedure is oriented towards training a system to be able to automatically determine which should be its ideal behaviour under a specific context, so that its performance is maximized.

To achieve this objective, the agent-environment paradigm is used, whose scheme is shown in Figure 1.1: the agent decides the actions that the environment must perform, based on its state or observation. The agent itself receives a numeric value, a *reward*, calculated as a function of the environtment's state after performing an action. This way, the agent can *learn* which actions must be taken by the environment to maximize the cumulative reward.

Motivation for using Reinforcement Learning

The methods of Reinforcement Learning are appropiate for complex problems in which there seems to be no obvius solution. However, in the particular case of power capping, an direct solution could be deployed with the following strategy:

- Allowing frequency raises while the system is below the energetic limit.
- Force frequency drops when this energetic limit is surpassed until consumption drops below it.

Nevertheless, the energetic consumption of a processor is quite a variable magnitude, depending on numerous factores, such as temperature, execution frequency, CPU's functional units use, etc. In this context, it seems appropiate to employ reinforcement learning techniques, so that trained agents can develop a proper understanding of the energetic fluctuations of the processor.

Structure of the report

In Chapter 2, the different tools used along the work are introduced with examples, all of them being Python modules. On one hand, we have those oriented towards reinforcement learning and the agent-environment pàradigm:

- The *GYM* module, used to create environments.
- The *RLlib* module, that implements the behaviour of the agent and comes with numerous algorithms from reinforcement learning.

On the other hand, we have the tools employed to understand and modify the state of the processor:

- The *cpufreq* module, used to read and vary the frequency of the processor at core level.
- The *pyRAPL* module, whose function is to gather measurements of the energetic consumption of the processor.

In Chapter 3, we present the implementation of a *GYM* environment focused towards the power capping problem. This environment will display measurements of the processor's power as observations, while receiving frequency raises and drops as actions.

In Chapter 4, the energetic analysis of a workload based on the matrix product is presented. This workload will be used during the training of the environment built in Chapter 3, their results being compared to the energetic footprint of the workload.

In Chapter 5, possible improvements of the environment are studied, with the aim of improving its performance. The results will be contrasted with the ones from the previous chapter.

In Chapter 6, new workloads with different energetic footprint will be introduced. They will be tested with the trained agents from Chapters 4 and 5, so as to confirm whether the agents keep their properties or not.

In Chapter 7, we manifest the findings throughout our work and suggest possible ways to follow in the future.

Capítulo 3

Herramientas de trabajo y ejemplo de uso

En este capítulo se introducen, en primer lugar, los módulos *GYM* y *RLlib*, que se utilizarán para implementar de forma específica el paradigma agente-entorno para problemas de aprendizaje reforzado. En particular, se estudiará el funcionamiento del entorno y el entrenamiento con un ejemplo sencillo.

A continuación, se expondrán los módulos *cpufreq* y *pyRAPL*, que permiten manipular la frecuencia del procesador, así como obtener medidas energéticas del mismo. Se examinarán, además, algunas particularidades especiales a tener en cuenta a la hora de trabajar con ellos.

3.1. Paradigma agente-entorno: *GYM* y *RLlib*

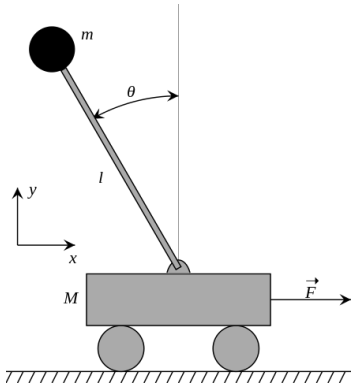
3.1.1. El módulo *GYM*

Descripción y finalidad

GYM [4] se presenta como un conjunto de herramientas para desarrollar y comparar algoritmos de aprendizaje reforzado. En particular, está formado por una colección de problemas tipo, que se definen en *entornos*, y sobre los que pueden aplicarse algoritmos de entrenamiento propios. Estos entornos mantienen una interfaz común que permite aplicar los algoritmos de forma general; es decir, los entornos no asumen nada sobre la estructura del *agente* que aplica el algoritmo.

El primer objetivo de *GYM* es confeccionar una colección extensa y diversa de entornos sobre los que aplicar técnicas de aprendizaje reforzado. En este sentido, busca proporcionar el equivalente a los conjuntos de datos, ya existentes, que se emplean para probar algoritmos de aprendizaje supervisado y no supervisado. Es por ello que el módulo presenta una amplia variedad de entornos que reproducen distintos tipos de problemas. Así, encontramos desde problemas de control clásico como el del péndulo invertido (Figura 3.1) a entornos que implementan videojuegos antiguos como el *Space Invaders* (Figura 3.2).

Figura 3.1
Esquema del problema del péndulo invertido.



Nota: imagen extraída de Wikipedia

Figura 3.2
El videojuego Space Invaders.

Nota: imagen extraída de la web de GYM

El segundo objetivo de *GYM* es el de crear un estándar para definir entornos en los algoritmos de aprendizaje reforzado. Así, todos sus entornos predefinidos comparten una misma estructura en la que nos podremos basar para diseñar entornos propios. *GYM* permite *registrar* estos entornos personalizados y manejarlos como si fueran uno más de su colección.

Definición de entornos

A continuación, se detallará la especificación básica de un entorno de *GYM*, comenzando por sus atributos imprescindibles:

- **Espacio de observación.** Determina el conjunto de estados que puede presentar el entorno. Si el número de estados es finito, tendríamos lo que *GYM* denomina un espacio discreto, `Discrete(N)`, donde N es el número fijo de estados que puede tomar el entorno. Para espacios de observación continuos y/o multidimensionales se utiliza `Box(N, M, ...)`, donde las variables N, M, ... determinan el orden de cada dimensión.
- **Espacio de acciones.** Determina el conjunto de acciones que se pueden aplicar sobre el entorno. De forma análoga al espacio de observación, el espacio de acciones puede ser finito o continuo y/o multidimensional.
- **Recompensas.** Cantidades numéricas que determinan la utilidad de las acciones aplicadas. Típicamente, se calculan en función del nuevo estado resultante de aplicar una acción.

Seguidamente, el funcionamiento principal del entorno se da mediante la función de paso `step()`. Este método recibe una acción, siendo su cometido es el de modificar el estado del entorno de acuerdo a la acción recibida y calcular la recompensa que genera el cambio de estado. En concreto, la función devuelve las siguientes variables:

- `observation`. Un objeto que representa el estado del entorno tras aplicar la acción recibida.
- `reward`. Un valor númerico con la recompensa generada por la aplicación de la acción recibida.
- `done`. Un booleano que determina si se reinicializa el entorno mediante la función `reset()`. En general, un valor `True` indica que el entorno ha llegado a un estado terminal o que ha alcanzado un límite máximo de pasos.
- `info`. Un diccionario opcional que contiene información útil para analizar el entorno en profundidad tras la acción. Esta información no se utiliza por el agente en el entrenamiento.

En rasgos generales, la función de paso `step()` es un implementación de la labor del entorno en el paradigma agente-entorno: recibe una acción del agente, la interpreta y devuelve el nuevo estado y la recompensa.

3.1.2. El módulo *RLlib*

Integración dentro del módulo *Ray*

El módulo *RLlib* [5, 6] es una biblioteca orientada al aprendizaje reforzado, pero integrada dentro de un módulo más genérico, *Ray*. *Ray* se presenta como una interfaz para construir aplicaciones distribuidas. Bajo esta premisa, el módulo *RLlib* implementa sus algoritmos de aprendizaje reforzado haciendo uso de la capacidad que proporciona el paralelismo de *Ray*.

En este caso, el módulo proporciona distintas clases de agentes que, asociados a un entorno específico, podrán entrenarse según el algoritmo de aprendizaje reforzado reflejado en su configuración. El comportamiento de estos agentes está concretado por lo que *RLlib* denota políticas (*policies*). Estas se encuentran implementadas en clases de Python y determinan el método de aprendizaje de

los agentes. Aunque es posible crear políticas personalizadas, se trabajará con las ya predefinidas en *RLlib*, como el algoritmo *Proximal Policy Optimization* (PPO).

Métodos *Policy Gradient* y el algoritmo PPO

Policy gradient [7] es un método particular del aprendizaje reforzado. Esta técnica mantiene cierta similitud con el aprendizaje supervisado, en el que se parte de un conjunto de datos *etiquetados*. Con respecto al paradigma agente-entorno, este conjunto estaría compuesto de una serie de observaciones de las que se conoce la acción correcta a tomar. A partir de este conjunto de datos, se calcularía una política general a aplicar para todo el rango de observaciones.

En ausencia de esta información, los métodos de *Policy Gradient* emplean políticas provisionales, que asocian una acción a cada observación y cuyos resultados se miden en función de la recompensa. Posteriormente, se llevan a cabo varios episodios con esta política, cada uno con una recompensa acumulada. En base a esta, se modifica la política de forma que se fomenten las acciones de aquellos episodios con mejor rendimiento que el resto.

Sin embargo, la obtención de buenos resultados utilizando métodos de *Policy Gradient* no es sencilla, debido principalmente a la sensibilidad del algoritmo al paso de modificación de la política. Si la variación es poco significativa, el algoritmo progresará demasiado lento; por otro lado, si el cambio es muy grande, se verá demasiado afectado por el *ruido*.

En este contexto surge el algoritmo PPO [8] que, a grandes rasgos, mejora la estabilidad del método, limitando adecuadamente la actualización de la política en cada caso, y ofrece una menor complejidad que otros algoritmos de este tipo. Además, PPO tiene un rendimiento comparable e incluso mejor que otros algoritmos de aprendizaje reforzado, convirtiéndose en la opción por defecto en algunos módulos de Inteligencia Artificial, como *OpenAI*. Por esta razón, lo utilizaremos en los entrenamientos a lo largo del trabajo.

3.1.3. Ejemplo de construcción de entorno con *GYM*

Se presenta, a continuación, la implementación de un entorno personalizado [9] como ejemplo, con el que mostrar de forma sencilla la creación de entornos en *GYM*. El entorno del ejemplo reproduce un *array* de tamaño 10 por el que desplazarse, habiendo una posición objetivo situada en el índice 5. Se analizan sus atributos de forma pormenorizada:

- **Espacio de observación.** Los estados del espacio se corresponden con las distintas posiciones del *array*. Así, tendríamos un espacio $Discrete(10)$, compuesto por los estados $\{0, 1, \ldots, 9\}$.
- **Espacio de acciones.** Para todo estado se dan 2 acciones posibles: desplazarse una posición a la izquierda o desplazarse una posición a la derecha. Ambas acciones pueden ocurrir también en los extremos del *array*, pero se evita el desbordamiento manteniendo la posición. Por tanto, el espacio de acciones sería $Discrete(2)$.
- **Terminación.** Se establece un número máximo de pasos, 10, tras los cuales el entorno se reinicializa, independiente de si ha alcanzado el estado objetivo o no.
- **Recompensas.** Se definen 3 recompensas tras la aplicación de una acción. Si la acción aleja al entorno del estado objetivo, la recompensa es `-2`. Si, por el contrario, lo acerca, la recompensa

es `-1`. Finalmente, si la acción lo desplaza justamente al estado objetivo, la recompensa es `+10`.

Para implementar el entorno, este se ha de definir como una clase que hereda de `gym.Env`, como se muestra en la Figura 3.3, y donde se han definido también los valores númericos asociados a las acciones y las recompensas, así como el número máximo de pasos y el tamaño del *array*. La inicialización de los espacios, no obstante, se lleva a cabo en el constructor del entorno, que aparece en la Figura 3.4.

Figura 3.3
Creación de una clase para el entorno.

```
class Example_v0 (gym.Env):
    # Acciones posibles
    MOVE_LEFT  = 0
    MOVE_RIGHT = 1

    # Tamaño del array y límite de pasos
    SIZE = 10
    MAX_STEPS = 10

    # Recompensas posibles.
    REWARD_AWAY = -2
    REWARD_STEP = -1
    REWARD_GOAL = 10
```

Figura 3.4
Constructor del entorno

```
def __init__(self):
    # El espacio de acciones, donde:
    #  '0' desplazamiento a la izquierda
    #  '1' desplazamiento a la derecha
    self.action_space = gym.spaces.Discrete(2)

    # El espacio de observación, determinado por el tamaño del array.
    self.observation_space = gym.spaces.Discrete(self.SIZE)

    # La posición objetivo.
    self.goal = 5

    # La función de reinicialización, que toma un estado inicial.
    self.reset()
```

Como puede advertirse, se han inicializado los valores del espacio de observación, el espacio de acciones y las recompensas, los cuales serán comunes a todos los entornos de esta clase. Finalmente, quedaría la implementación de la función `step()`, mostrada en la Figura 3.5, y cuyos valores de retorno vienen dados por:

Figura 3.5
Función de paso del entorno.

```
        def step(self, action):
            if self.count == self.MAX_STEPS:
                self.done = True
            else:
                # Contador de pasos
                self.count += 1

                if action == self.MOVE_LEFT:
                    if self.state == 0:
                        self.reward = self.REWARD_AWAY
                    else:
                        self.state -= 1

                        if self.state == self.goal:
                            self.reward = self.REWARD_GOAL
                            self.done = True
                        elif self.state < self.goal:
                            self.reward = self.REWARD_AWAY
                        else:
                            self.reward = self.REWARD_STEP

                elif action == self.MOVE_RT:
                    if self.state == self.SIZE - 1:
                        self.reward = self.REWARD_AWAY
                    else:
                        self.state += 1

                        if self.state == self.goal:
                            self.reward = self.REWARD_GOAL
                            self.done = True
                        elif self.state > self.goal:
                            self.reward = self.REWARD_AWAY
                        else:
                            self.reward = self.REWARD_STEP

            self.info["dist"] = self.goal - self.position
            return [self.state, self.reward, self.done, self.info]
```

- `observation`. Un entero que representa la posición del *array* en que se encuentra el entorno tras la acción.
- `reward`. El valor de la recompensa calculada tras aplicar la acción.
- `done`. Un booleano que toma valor `True` únicamente si el entorno se halla en el estado objetivo tras la acción o si se ha llegado al número máximo de pasos.
- `info`. Un diccionario cuyo único información es la distancia entre el estado objetivo y la posición del entorno tras la acción.

De esta forma, se habría terminado de configurar el entorno, tras lo cual únicamente faltaría registrarlo en la colección con el resto de entornos predefinidos de *GYM*. Para ello, empleamos el método `register()`, con el que se registra el entorno asociado a un identificador a elegir, como se ve en la Figura 3.6.

Figura 3.6
Registro del entorno.

```
from gym.envs.registration import register

register(
    id = "example-v0",
    entry_point = "gym_example.envs:Example-v0",
)
```

Figura 3.7
Entrenamiento del agente PPO.

```
# Inicialización de Ray
ray.init()

# Registro del entorno
select_env = "example-v0"
register_env(select_env, lambda config: Example_v0())

# Creación del agente
config = ppo.DEFAULT_CONFIG.copy()
agent = ppo.PPOTrainer(config, env=select_env)

# Entrenamiento durante 5 épocas.
for _ in range(5):
    agent.train()
    agent.save(save_dir)
```

3.1.4. Ejemplo de entrenamiento mediante PPO con *RLlib*

Una vez propuesto el entorno, se genera un agente PPO de *RLlib* basado en él. Este agente consiste de una red neuronal adaptada al espacio de observación (entrada) y al espacio de acciones (salida) del entorno.

En primer lugar, se inicializa *Ray* y se añade el entorno al registro de *Ray*. A continuación, se obtiene el agente PPO y se entrena mediante la función `train()`. Cada ejecución de esta función supone una época del entrenamiento de la red neuronal, tras la cual podemos guardar el estado del agente en un *checkpoint*. Este proceso se muestra implementado en la Figura 3.7.

Finalmente, para comprobar si el entrenamiento ha funcionado correctamente, se carga el último *checkpoint* del agente, se inicializa el entorno y se deja que el agente decida la acción a aplicar a partir del estado del entorno, como ocurre en la Figura 3.8.

Figura 3.8
Prueba del agente entrenado.

```
# Recuperación del agente
agent.restore(chkpt_file)

# Inicialización del entorno
env = gym.make(select_env)
state = env.reset()

# Ejecución durante 20 pasos
for _ in range(20):
    action = agent.compute_action(state)
    state, reward, done, _ = env.step(action)

    if done:
        state = env.reset()
```

3.2. Interacción con el sistema: *cpufreq* y *pyRAPL*

3.2.1. El módulo *cpufreq*

Descripción e inicialización

cpufreq [10] está destinado tanto a obtener como a modificar el estado de los núcleos de la CPU y sus frecuencias de trabajo. Es un módulo solo aplicable a sistemas Linux, estando su funcionamiento basado en el acceso al directorio `/sys/devices/system/cpu`, que contiene los ficheros que permiten consultar y modificar el estado de la CPU.

Para trabajar con este módulo, se ha de construir una instancia de la clase `cpuFreq()` y trabajar directamente con sus métodos. Esta instancia analiza el directorio mencionado anteriormetne, del cual obtiene los siguientes atributos:

- `available_frequencies`. Una lista de valores enteros con los escalones de frecuencia (en KHz) a los que pueden trabajar los núcleos de la CPU. Aparece también una frecuencia máxima que pone el núcleo a máxima potencia dentro de sus posibilidades, por lo que el valor no se corresponde con la frecuencia real, que puede oscilar bastante.
- `available_governors`. Una lista con los distintos esquemas de control o modos (*governors*) que pueden adoptar los núcleos del procesador. Estos determinan de qué forma variarán la frecuencia los distintos núcleos durante el funcionamiento del sistema.

Manipulación de los esquemas de control

Con respecto a los esquemas *governor* [11], estos pueden consultarse por núcleo mediante el método `get_governors()`, y modificar su estado con `set_governors()`, pasando como argumento uno de los esquemas de `available_governors`. Entre los esquemas disponibles, los más habituales son:

- `performance`. Hace trabajar al núcleo a la máxima frecuencia posible.
- `powersave`. Hace trabajar al núcleo a la mínima frecuencia posible.
- `ondemand`. Modifica la frecuencia del núcleo dinámicamente en función de la carga de trabajo. Suele ser el esquema por defecto de los núcleos de la CPU.
- `consevative`. Funciona de forma análoga a `ondemand`, pero escala la frecuencia de forma más gradual.
- `userspace`. Hace trabajar al núcleo a las frecuencias especificadas por el usuario. Por ello, a la hora de modificar frecuencias, este ha de ser el esquema establecido.

Modificación de frecuencias

Para visualizar las frecuencias de la CPU se utiliza la función `get_frequencies()`, que recopila la frecuencia de los núcleos. Estos, además, tienen como parámetros una frecuencia mínima y otra máxima, entre las que debe mantenerse la frecuencia real. Estos parámetros pueden modificarse con los métodos `set_min_frequencies()` y `set_max_frequencies()`.

Por otro lado, la forma canónica de modificar la frecuencia de un núcleo es con el método `set_frequencies()`, que sobreescribe un determinado fichero asociado a cada núcleo con uno de los valores disponibles en `available_frequencies`. No obstante, se han de tener en cuenta las siguientes consideraciones:

- En algunos casos, el empleo único de esta función no es suficiente para que el cambio se vea reflejado en la frecuencia real. Afortunadamente, modificando también los parámetros de frecuencia mínima y máxima a la frecuencia deseada el problema se soluciona.
- Incluso modificando la frecuencia con el método anterior, el cambio no será perceptible mientras no haya una carga significante de trabajo en el sistema (o, más bien, sobre los núcleos). En aquellos núcleos que no tengan tal carga, la frecuencia se mantendrá en cotas bajas para evitar un consumo innecesario.

3.2.2. El módulo *pyRAPL*

Descripción e inicialización

pyRAPL [12] es una herramienta que permite medir la huella energética de una máquina durante la ejecución de un fragmento de código en `Python`. Hace uso de la tecnología de Intel *Running Average Power Limit* (RAPL), que estima el consumo de energía en los procesadores de la marca. En concreto, el módulo puede leer el consumo de la CPU a nivel de *socket*, esto es, los distintos conjuntos en que se agrupan los núcleos del procesador. En el sistema utilizado para este trabajo, se dan 2 *sockets* de 8 núcleos cada uno (numerados como `0-7` y `8-15`, respectivamente). También existe la opción de medir el consumo de la DRAM.

Para utilizar el módulo, se comienza inicializando el proceso de medición con la función `setup()`, que acepta dos variables opcionales:

- `devices`. Una lista con los dispositivos que se medirán, a saber, la CPU, dada por `Device.PKG`, y/o la DRAM, dada por `Device.DRAM`. En caso de no proporcionarse, el entorno tomará medidas de los 2 dispositivos.
- `socket_ids`. Una lista con los identificadores numéricos de los *sockets* que se medirán en los dispositivos.

Medición de consumo energético

Para leer el consumo, se ha de tomar un objeto de la clase `Measurement`, que funcionará como un medidor. En efecto, basta colocar el bloque de código cuya huella energética se desea medir entre las llamadas a sus métodos `meter.begin()` y `meter.end()`. Tras la medida, se obtiene un diccionario `meter._results.pkg`, con la energía consumida en microjulios en cada *socket*; así como el tiempo, en microsegundos, que ha tardado en ejecutarse el bloque de código en`meter._results.duration`. Podemos utilizar ambos datos para calcular la potencia media, en vatios, a la que ha trabajado el procesador durante la ejecución del código.

El proceso mencionado está reflejado en la Figura 3.9. Como se puede observar, el bloque de medición está contenido en un bucle `while`, del que no se saldrá mientras se cumpla su condición. Esto se debe a que, internamente, el módulo *pyRAPL* trabaja leyendo un fichero determinado asociado a cada *socket* del directorio `/sys/class/powercap/intel-rapl`. En la ejecución de los métodos `begin()` y `end()`, se leen 2 valores númericos consecutivos de este fichero, de forma que la energía consumida se obtiene de la diferencia entre ambos.

No obstante, el valor numérico de estos ficheros, que crece con el tiempo, puede desbordarse. Cuando esto ocurre, *pyRAPL* no avisa de ningún error, sino que simplemente no inicializa el diccionario `meter._results.pkg`. Por ello, cuando esto ocurra, se ha de repetir la medición.

Figura 3.9
Ejemplo de medición de consumo con pyRAPL.

```python
# Inicialización de pyRAPL
pyRAPL.setup(
    devices = [pyRAPL.Device.PKG],
    socket_ids = [0]
)

# Método de medición.
def measure_power(mtime):
    meter = pyRAPL.Measurement()

    while meter._results is None or meter._results.pks is None:
        meter.begin()
        time.sleep(mtime)
        meter.end()

    # Lectura de datos
    m_energy = meter._results.pkg[0]
    m_time   = meter._results.duration
    power    = m_energy / m_time

    return power
```

Capítulo 4

Construcción de un entorno GYM adaptado al problema de limitación de potencia (*power capping*)

En este capítulo se presenta la implementación de un entorno GYM adaptado al problema de *power capping*. Se busca dotar al entorno de acciones que le permitan variar la frecuencia de la CPU para lograr (no superar) una potencia de consumo dada. Por tanto, el espacio de observación del entorno tendrá que trabajar con medidas de potencia y, de la misma manera, las recompensas que se definan deberán premiar aquellos cambios de frecuencia que acerquen al entorno a la potencia objetivo, penalizando a los que lo alejen de la misma.

Ciertamente, hay numerosas posibilidades a la hora de definir este entorno. No obstante, se partirá de la configuración inicial que se exponga en este capítulo, dejando para capítulos posteriores las posibles modificaciones que puedan mejorar los resultados del entrenamiento.

Atributos iniciales del entorno

En primer lugar, se exponen una serie de parámetros del entorno, configurables en el momento de su inicialización. Únicamente se expondrá una breve definición, pues su utilidad se explicará de forma particular en los siguientes apartados.

- `POWER`. El valor de la potencia a la que debe aspirar el entorno.
- `SOCKET`. El número de *socket* del que se obtendrán las mediciones de energía/potencia. En lo que sigue, se utilizará el *socket* número 1, dejando el número 0 para el entrenamiento del agente.
- `CORES`. Una lista con los núcleos asignados al `SOCKET`.
- `MAXSTEPS`. El número máximo de pasos que puede ejecutar el entorno. En general, se empleará un valor de 200 pasos.

- `SEED`. La semilla del generador de números aleatorios, utilizado para elegir un estado inicial al reiniciar el entorno.
- `MINPOWER`. El límite inferior del rango de potencia. Será de $15\,W$, la potencia en reposo del procesador empleado.
- `MAXPOWER`. El límite superior del rango potencia. Será de $115\,W$ por el TDP del procesador empleado.
- `POWERSTEP`. El tamaño fijo de los intervalos en los que se dividirá el rango de potencia.
- `MEASURE_TIME`. El tiempo empleado entre acciones para medir la potencia a través de *pyRAPL*.
- `SLEEP_TIME`. El tiempo posterior a tomar una acción y previo a medir la potencia del procesador con la nueva frecuencia dada por la acción.
- `DECISION_TIME`. El tiempo que transcurre entre la realización de una acción y la elección de la siguiente. Equivale a la suma de los 2 valores anteriores.

4.1. Espacio de acciones

Se emplearán únicamente 2 acciones:

- `LOWER_FREQ`. Disminuye un escalón de frecuencia a los núcleos del procesador.
- `RAISE_FREQ`. Aumenta un escalón de frecuencia a los núcleos del procesador.

Consideraciones sobre la manipulación de frecuencias

En primer lugar, puesto que el consumo energético del procesador solo puede obtenerse a nivel de *socket*, la variación de frecuencia que generan las acciones se realizará exclusivamente sobre los núcleos asociados a los *sockets* que se vayan a medir. En este caso, únicamente se tomarán datos de uno, dado por `SOCKET`, y sus núcleos asociados vienen dados por `CORES`.

Por otro lado, se excluirá el escalón de frecuencia de *turbo* al modificar las frecuencias, pues su volatilidad se trasladaría a las mediciones de potencia bajo ese escalón de frecuencia, afectando a la integridad de las observaciones del entorno.

4.2. Espacio de observación

Con la intención de evitar complejidad, se construirá un espacio de observación discreto, a pesar de que el rango de potencia sea continuo. Así, este se particionará en un determinado número de intervalos que representarán los estados del entorno. De esta forma, tras tomar una medida de potencia, el estado del entorno será aquel intervalo en que esté contenido este valor.

Los intervalos estarán determinados por un tamaño fijo, que vendrá dado por el parámetro `POWERSTEP`. Sin embargo, para que el número de intervalos sea finito, se necesitan unos límites que acoten el rango de potencia; estos límites están representados por los parámetros `MINPOWER` y `MAXPOWER`.

De esta forma, comenzando por el límite interior, basta con sumar el tamaño fijado para conseguir los extremos de los intervalos, lo que se hace en el método de la Figura 4.1.

Figura 4.1
Partición equiespaciada del rango de potencia.

```
def get_powerpoints(self, pstep):
    powers = []
    ppoint = self.MINPOWER
    powers.append(ppoint)
    while ppoint < self.MAXPOWER:
        ppoint += pstep
        powers.append(ppoint)

    return powers
```

Figura 4.2
Construcción de los intervalos a partir de la partición.

```
def get_intervals(self, powerpoints):
    intervals = []

    # Primer intervalo abierto
    ppoint = powerpoints[0]
    intervals.append( [None, ppoint] )

    # Intervalos intermedios
    for i in range(1, len(powerpoints)):
        intervals.append( [ppoint, powerpoints[i]] )
        ppoint = powerpoints[i]

    # Último intervalo abierto
    intervals.append( [ppoint, None] )

    return intervals
```

Finalmente, también se incluirán otros 2 intervalos abiertos de tamaño *infinito*: uno para los valores inferiores a `MINPOWER` y otro para los superiores a `MAXPOWER`. La motivación de estos 2 intervalos es la de evitar errores en el entorno si en algún momento se mide una potencia que esté fuera de los límites establecidos. Se entiende que estos casos sólo ocurren en situaciones excepcionales provocadas por situaciones anómalas del propio RAPL. Así, los intervalos se determinan a partir de la partición en el método de la Figura 4.2.

Así, el espacio de observación queda determinado por el bloque de código mostrado en la Figura 4.3. Nótese que el tamaño del espacio de observación es una unidad mayor que el número de

intervalos. Esto se debe a que *RLlib* no acepta el valor 0 para una observación discreta.

Figura 4.3
Determinación del espacio de observación.

```
self.POWERSTEP    = config.get('powstep', self.DEF_POWERSTEP)
self.POWERPOINTS = self.get_powerpoints(self.POWERSTEP)
self.INTERVALS    = self.get_intervals(self.POWERPOINTS)

self.observation_space = gym.spaces.Discrete( len(self.INTERVALS) + 1 )
```

Medida de la potencia tras modificar la frecuencia

El estado del entorno se calcula a partir de una medida de potencia, en la mayoría de casos tras una acción que modifica la frecuencia del procesador. Para otorgar un margen de tiempo para que el procesador se estabilice en el nuevo escalón de frecuencia, se introduce un tiempo de espera previo a la medición subsiguiente, que viene dado por el parámetro `SLEEP_TIME`. Tras este periodo de tiempo, se procede a medir la potencia del procesador, lo cual también requiere de un fragmento de tiempo durante el que tiene lugar este proceso, cuyo valor lo establece `MEASURE_TIME`.

La suma de ambos parámetros está reflejada en `DECISION_TIME`, que representa el tiempo que transcurre entre la aplicación de una acción y el cálculo del nuevo estado del entorno. Todo este proceso está implementado en la Figura 4.4.

Figura 4.4
Proceso de cambio de frecuencia y medición de la potencia.

```
def set_wait_measure(self, freq):
    # Modificación de la frecuencia
    self.set_frequency(freq)

    # Tiempo de espera
    time.sleep(self.SLEEP_TIME)

    # Medición de la potencia
    power = self.measure_power(self.MEASURE_TIME)

    return power
```

En capítulos posteriores, se fijará un `DECISION_TIME` de 0,25 segundos, de forma que se ejecutarían 4 pasos por segundo. Además, el tiempo de medición sería de 0,15 segundos, dejando 100 milisegundos para que se realice completamente el cambio de frecuencia.

4.3. Recompensas

Las recompensas estarán determinadas por el cambio en la potencia (el estado) que haya provocado la acción respecto del estado objetivo.

- `REWARD_CLOSER`. Recompensa otorgada cuando el nuevo intervalo de potencia está más cercano al objetivo que el anterior. Se aplica el valor $+1$.
- `REWARD_FARTHER`. Recompensa otorgada cuando el nuevo intervalo de potencia está más lejos del objetivo que el anterior. Se aplica el valor -1.
- `REWARD_GOAL`. Recompensa otorgada cuando el nuevo intervalo de potencia es el estado objetivo. Se aplica el valor $+2$

Este mecanismo de recompensa está implementado como un método en la Figura 4.5. En efecto, se penaliza que la acción aleje al entorno del objetivo, mientras que se recompensa positivamente que se acerque al mismo, dando un bonus adicional si le lleva justamente a él.

Figura 4.5
Proceso de cambio de frecuencia y medición de la potencia.

```
self.REWARD_CLOSER  = +1
self.REWARD_FARTHER = -1
self.REWARD_GOAL    = +2

def get_reward(self, state, prev_state):
    ### Positive while on goal.
    if state == self._goal:
        return  self.REWARD_GOAL

    if state < self._goal:
        if state - prev_state > 0:
            return  self.REWARD_CLOSER
        else:
            return  self.REWARD_FARTHER
    if state > self._goal:
        if state - prev_state < 0:
            return  self.REWARD_CLOSER
        else:
            return  self.REWARD_FARTHER
```

4.4. Función de reinicialización

El método `reset()` se utiliza para dar al entorno un estado inicial. En este caso, se ha de elegir una frecuencia inicial que aplicar al procesador para posteriormente medir el consumo energético y obtener un estado inicial en función de la potencia. La frecuencia inicial se puede asignar de dos maneras, como se observa en la Figura 4.6.

- A partir del generador de números aleatorios del entorno, que se crea en función del valor del parámetro `SEED`. Si no se aporta este valor, la elección de la frecuencia será impredecible, mientras que si se da, se asegurará que la sucesión de frecuencias obtenidas será la misma para todo generador con una misma semilla.
- Pasando un valor de frecuencia inicial como argumento a la función de `reset()`. Es decir, el entorno generará su estado inicial en función de esa frecuencia.

Figura 4.6
Función de reinicialización.

```
def reset(self, reset_freqpos = None):
    # Reinicio de variables
    self._reward     = 0
    self._acc_reward = 0
    self._done   = False
    self._info   = {}
    self._count  = 0

    # Elección de frecuencia inicial
    if reset_freqpos is None:
        self._freqpos = self.RNG.choice( np.arange( len(self._frequencies) ) )
    else:
        self._freqpos = reset_freqpos
    freq = self._frequencies[ self._freqpos ]

    # Obtención del estado inicial
    self._power = self.set_wait_measure(freq, 'Reset')
    self._state = self.get_state( self._power )

    # Actualización de las variables del entorno
    self.update_info()

    return self._state
```

4.5. Función de paso

Para la función de paso `step()` se implementa el siguiente procedimiento, reflejado en la Figura 4.7. Dependiendo de la acción, se decide si subir o bajar un escalón la frecuencia de los núcleos. A continuación, se modifica la frecuencia y se mide la potencia del *socket* con el método `set_wait_measure()`. Seguidamente, se calcula la recompensa derivada de la acción mediante la función `get_reward()`. Finalmente, se actualizan las variables del entorno con el nuevo estado y la función `update_info()`, implementada en la Figura 4.8

Figura 4.7
Función de paso.

```
def step(self, action):
    # Comprobación del límite de pasos
    if self._count == self.MAXSTEPS:
        self._done = True
        return self._state, self._reward, self._done, self._info

    # Aplicación de la acción
    if action == self.RAISE_FREQ:
        if self._freqpos == len(self._frequencies) - 1:
            pass
        else:
            self._freqpos += 1
    elif action == self.LOWER_FREQ:
        if self._freqpos == 0:
            pass
        else:
            self._freqpos -= 1

    # Modificación de frecuencia y medición de potencia
    freq = self._frequencies[ self._freqpos ]
    next_power = self.set_wait_measure(freq)
    next_state = self.get_state( next_power )

    # Cálculo de recompensa
    self._reward      = self.get_reward(next_state, self._state)
    self._acc_reward += self._reward

    # Actualización de variables del entorno
    self._power = next_power
    self._state = next_state
    self._count += 1
    self.update_info()

    return [self._state, self._reward, self._done, self._info]
```

Figura 4.8
Función de actualización de variables del entorno

```
        def update_info(self):
            self._info['step'] = self._count

            self._info['state']    = self._state
            self._info['interval'] = self.INTERVALS[self._state - 1]

            self._info['reward']     = self._reward
            self._info['acc_reward'] = self._acc_reward

            self._info['freqpos']   = self._freqpos
            self._info['frequency'] = self._frequencies[ self._freqpos ]

            self._info['power'] = self._power
```

Capítulo 5

Entrenamiento del entorno y resultados

En este capítulo, se presenta, en primer lugar, un análisis del consumo energético del procesador bajo una carga de trabajo basada en el producto de matrices. Como ya se comentó en el Capítulo 2 al introducir la herramienta *cpufreq*, los cambios en la frecuencia (en términos de consumo energético) no serán perceptibles a no ser que el procesador esté sujeto a una carga de trabajo significativa. Por tanto, el análisis de esta operación permitirá comparar los resultados de rendimiento de los agentes, cuyo proceso de entrenamiento se realizará con esta carga de trabajo ejecutándose en segundo plano.

A continuación, se elegirá, por una parte, un tamaño de intervalo en función de los resultados del producto de matrices, y, por otra, un valor de potencia para el problema de *power capping*. Con esta configuración, se entrenará el entorno del capítulo anterior durante varias épocas, con el objetivo de analizar el sobreaprendizaje del agente y, con ello, elegir la época más eficiente para entrenamientos futuros.

Posteriormente, se realizarán otro par de entrenamientos sobre entornos con tamaños de intervalo distintos, con la finalidad de observar cuál es el efecto de este parámetro en el aprendizaje.

Finalmente, se llevarán a cabo otro par de entrenamientos, esta vez empleando los valores de potencia extremos (inferior y superior) que alcanza la carga de trabajo. Se comprobará también cómo afecta esto al rendimiento de los agentes.

5.1. Entorno de trabajo

Antes de presentar cualquier tipo de resultado, expondremos las características del entorno de trabajo en que se llevarán a cabo los distintos problemas. Se trata de un servidor cuyo modelo de CPU es *Intel(R) Xeon(R) CPU E5-2670 0 @ 2.60GHz*, el cual contiene 16 núcleos divididos en 2 *sockets* de 8 núcleos cada uno. Se tienen 15 escalones de frecuencia entre 1200 y 2600 MHz con un paso de 100 MHz. Hay también un escalón adicional de turbo que, como ya comentamos previamente, prescindiremos de utilizar.

Con respecto al lenguaje de programación, la versión de `Python` utilizada es la 3.8.5, mientras que para el resto de módulos utilizados se da:

- *cpufreq* 0.3.3.
- *pyRAPL* 0.2.3.1.
- *GYM* 0.17.3.
- *Ray* 1.0.1.
- *NumPy* 1.18.5.

5.2. Análisis de una carga de trabajo basada en el producto de matrices

Para implementar esta carga de trabajo, que denotaremos mediante el identificador `producto`, se utilizará el módulo *NumPy*, que se emplea principalmente para realizar operaciones y cálculos sobre *arrays* multidimensionales. En este caso, se inicializan aleatoriamente 2 matrices reales A, B y se almacena su producto AB en una matriz destino C. Esta operación implica multiplicaciones y sumas con tipos `float` y cargas/almacenamientos en memoria, lo que supondrá un uso intensivo de todas las unidades funcionales del procesador (el producto de matrices es realmente una operación intensiva en cómputo, ya que se realizan $\mathcal{O}(n^3)$ operaciones en punto flotante para $\mathcal{O}(n^2)$ accesos a memoria, siendo n la dimensión de las matrices –consideradas aquí cuadradas–).

A la hora de medir el consumo energético de estas tareas se ha de tener en cuenta, en primer lugar, que la medida es a nivel de *socket*; por tanto, es indispensable asociar la carga de trabajo a todos los núcleos del *socket* que sobre le que se van a realizar las mediciones. Además, para evitar las variaciones que podría generar cualquier tipo de paralelismo, se ejecutará la misma operación aislada en cada núcleo del *socket*.

Finalmente, puesto que la medida se hará durante un periodo de tiempo determinado, las operaciones se repetirán una y otra vez en cada núcleo durante ese tiempo. Para ello, al ejecutar estas tareas, se empleará el método *fork*, asociando cada proceso a su núcleo correspondiente y, finalmente, se acabará con las tareas externamente desde el proceso de medición. De la misma forma, el proceso de medición estará asociado a los núcleos de otro *socket*, para no interferir en con el *socket* donde se están realizando las mediciones.

Resultados del consumo energético

Para llevar a cabo el estudio energético, se emplean matrices cuadradas de dimensión 1000×1000, el tiempo de medición se fija en 30 segundos y se ejecuta la carga de trabajo sobre los núcleos del *socket* 1 del servidor. Con ello, se obtienen los siguientes datos de potencia para cada escalón de frecuencia, reflejados en la Tabla 5.1, y representados gráficamente en la Figura 5.1.

Figura 5.1
Gráfica del análisis energético del producto de matrices.

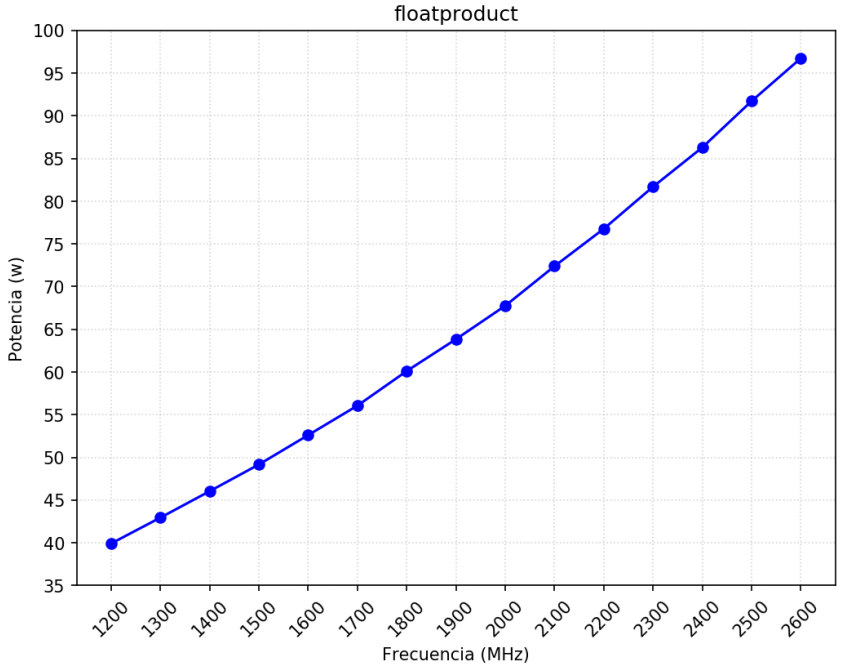


Tabla 5.1
Tabla del análisis energético del producto de matrices.

Frecuencia (MHz)	1200	1300	1400	1500	1600	1700	1800	1900
Potencia (W)	39,915	42,947	46,042	49,169	52,612	56,063	60,112	63,842

Frecuencia (MHz)	2000	2100	2200	2300	2400	2500	2600
Potencia (W)	67,758	72,393	76,772	81,695	86,299	91,724	96,744

5.3. Análisis del sobreaprendizaje

Configuración del entorno

Para realizar el primer entrenamiento, debemos elegir previamente la configuración inicial de este primer entorno, que denotaremos por el identificador `Env1`. Por una parte, se ha de tomar un tamaño fijo de intervalo. Puesto que los valores del análisis de la carga de trabajo presentan saltos

entre escalones consecutivos que van de los 3 a 5 vatios, se elegirá el valor más pequeño, 3, como tamaño para evitar que varios escalones recaigan en un mismo intervalo y, por tanto estén ambos representados por un único estado del entorno.

Por otra parte, se ha de fijar un valor de *power capping*, y se hará de forma que este se aproxime a algún valor intermedio de potencia obtenido en el análisis energético del producto de matrices. Parece adecuado elegir la potencia de 52,5, que además tiene una posición bastante centrada en su intervalo asociado [51,0, 54,0].

Duración del entrenamiento

Una vez establecida la configuración del entorno, se procede a realizar el entrenamiento del agente, empleando el algoritmo PPO. El entrenamiento tendrá 8 épocas de duración. Según la política predefinida por *RLlib* del algoritmo, en cada época se analizan 4000 pasos del entorno. Con esto en cuenta, añadido a la decisión comentada en el Capítulo 3 de que el tiempo de decisión, `DECISION_TIME`, sea de 250 milisegundos, se da un tiempo mínimo por época de:

$$4000 * 0{,}25\,s = 1000\,s = 16\,minutos\,40\,segundos$$

El tiempo por cada paso es lo suficientemente grande como para que el tiempo dedicado por el algoritmo a la reparametrización de la red neuronal es, en comparación, despreciable. Por tanto, el tiempo de entrenamiento no se verá afectado por la complejidad del algoritmo, como ocurre generalmente en los problemas de aprendizaje reforzado, sino que en este caso, estará determinado principalmente por el entorno.

Método de testeo

Una vez terminado el entrenamiento, siendo los agentes de cada época accesibles en sus respectivos *checkpoints*, se ha de determinar una forma lo más equilibrada posible de testear cada agente, de forma que los resultados obtenidos sean comparables y reproducibles.

Una opción disponible, sería la de obtener los resultados de un número N de episodios para obtener una posible media de los mismos. No obstante, los entornos se inicilizarían a frecuencias iniciales aleatorias, con lo que habría que realizar un número importante de iteraciones para asegurar que se cubre bien todo el espectro de escalones de frecuencia iniciales.

Sin embargo, puesto que los episodios se ejecutan durante 200 pasos, esto da una duración de 50 segundos, lo que puede generar largos tiempos de testeo para un número grande de iteraciones.

Por lo anterior, se ha optado por realizar un testeo de 15 iteraciones, una por cada escalón de frecuencia, de forma que cada iteración dé al entorno una frecuencia inicial distinta. Esto produce un tiempo de testeo más aceptable de 12 minutos y 30 segundos.

Resultados del entrenamiento

A continuación, se presentan en las Figuras 5.2 y 5.3 los resultados testear los agentes de cada época, mostrando la potencia y el número de intervalo (estado) a cada paso, respectivamente. Se marca también en cada gráfica la potencia y el estado objetivos con una línea horizontal.

Como puede observarse en las figuras, los agentes generan un comportamiento bastante caótico en las 2 primeras épocas, que evoluciona a comportamientos convergentes tanto a la potencia como al estado objetivo en las épocas consecutivas. Esta convergencia, además, se estabiliza a partir de la época 5.

Aunque estas gráficas son útiles para apreciar cómo el agente consigue hacer converger los resultados hacia el objetivo, partiendo de cualquier estado inicial, resulta más útil para realizar comparaciones utilizar el error medio de cada testeo. Esto es, sea $\tilde{x}$ el objetivo y x_i^j la medición correspondiente al paso i de la iteración j, se define su error como

$$\varepsilon_i^j = \left| x_i^j - \tilde{x} \right|$$

De esta forma, para cada época se obtiene el error medio de las iteraciones como la función:

$$\varepsilon^j(t) = \frac{1}{15} \sum_{j=1}^{15} \varepsilon_t^j$$

Como se aprecia en las Figuras 5.4 y 5.5, se percibe con mayor claridad en qué épocas se dan convergencias más estables y rápidas, disponiendo de una función comparable entre épocas paso a paso. No obstante, el objetivo final sería el de otorgar un valor a cada testeo, es decir, cada época que nos dé una idea de su rendimiento y que sea comparable con el resto.

Puesto que la finalidad del problema de *power capping* es la de mantener la potencia del procesador estable por debajo de la potencia especificada, se puede desechar la rapidez de la convergencia como un atributo significativo; pues, como se ha contemplado, la convergencia se produce con mayor o menor rapidez a partir de la tercera época. Se erige entonces la estabilidad, es decir, la capacidad del entorno de mantenerse en la potencia objetivo a lo largo del tiempo sin oscilar demasiado, como el parámetro a tener en cuenta para determinar el rendimiento.

Figura 5.2
Gráfica de la potencia de las diferentes iteraciones.

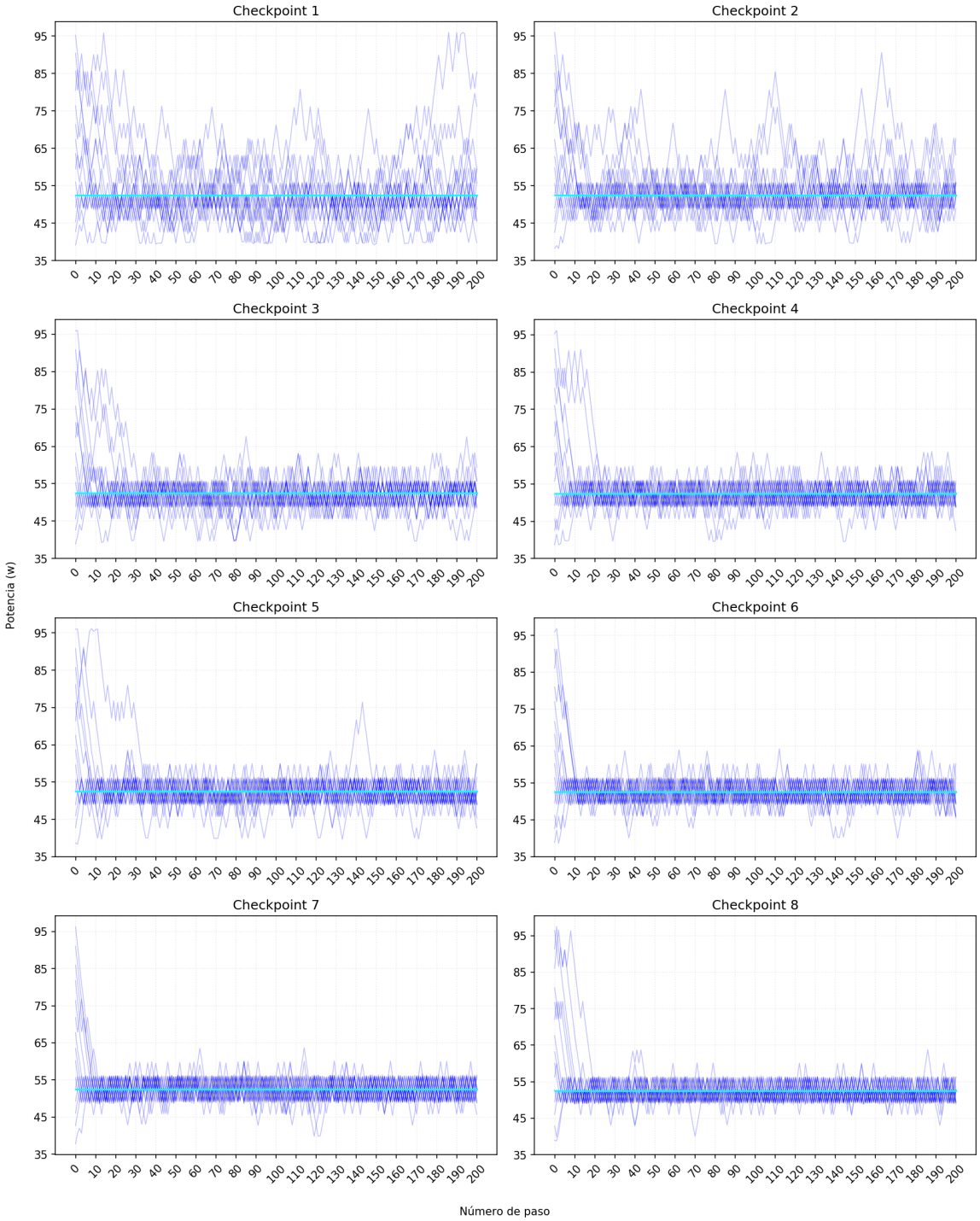


Figura 5.3
Gráfica del estado de las diferentes iteraciones.

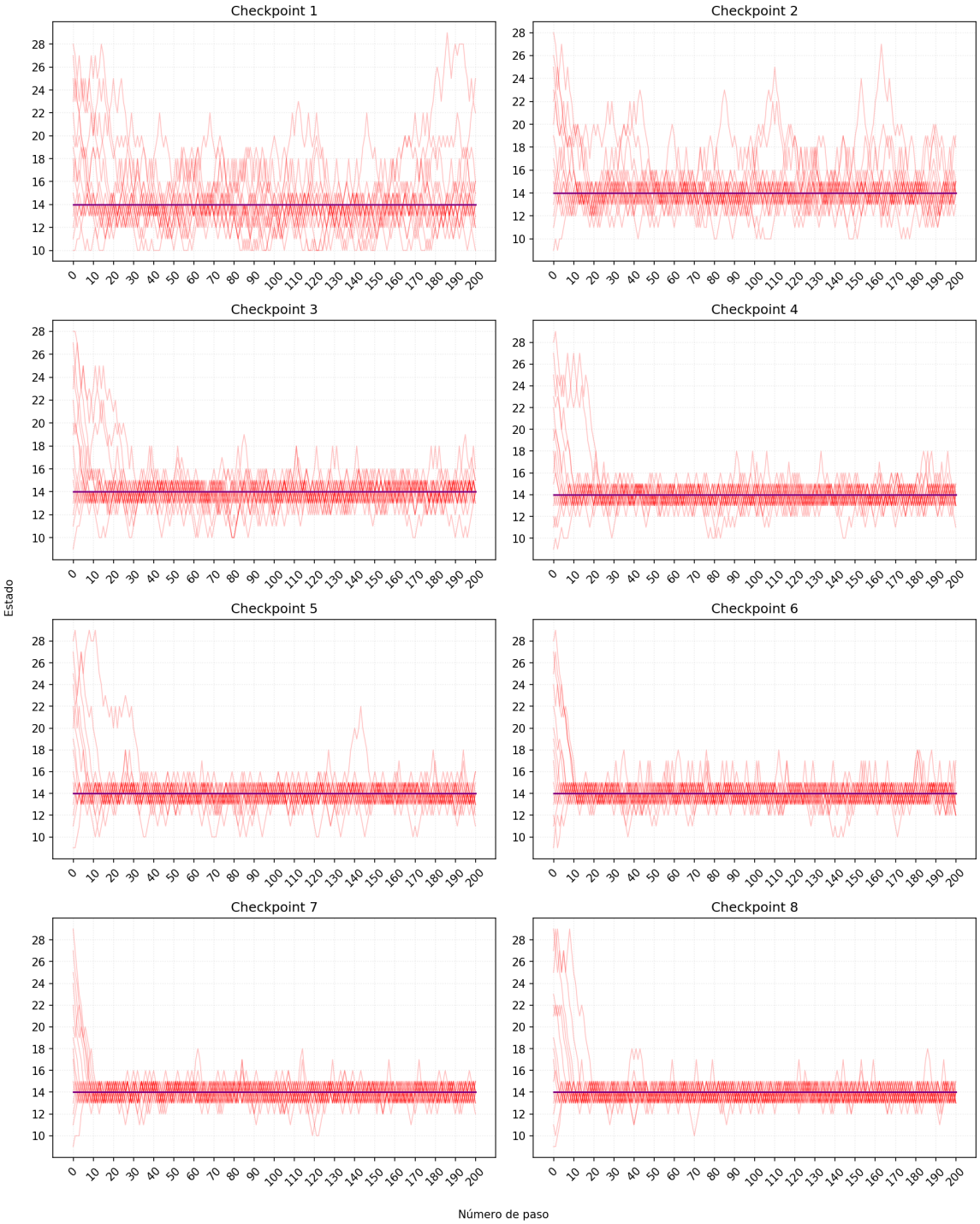


Figura 5.4
Gráfica del error medio de la potencia.

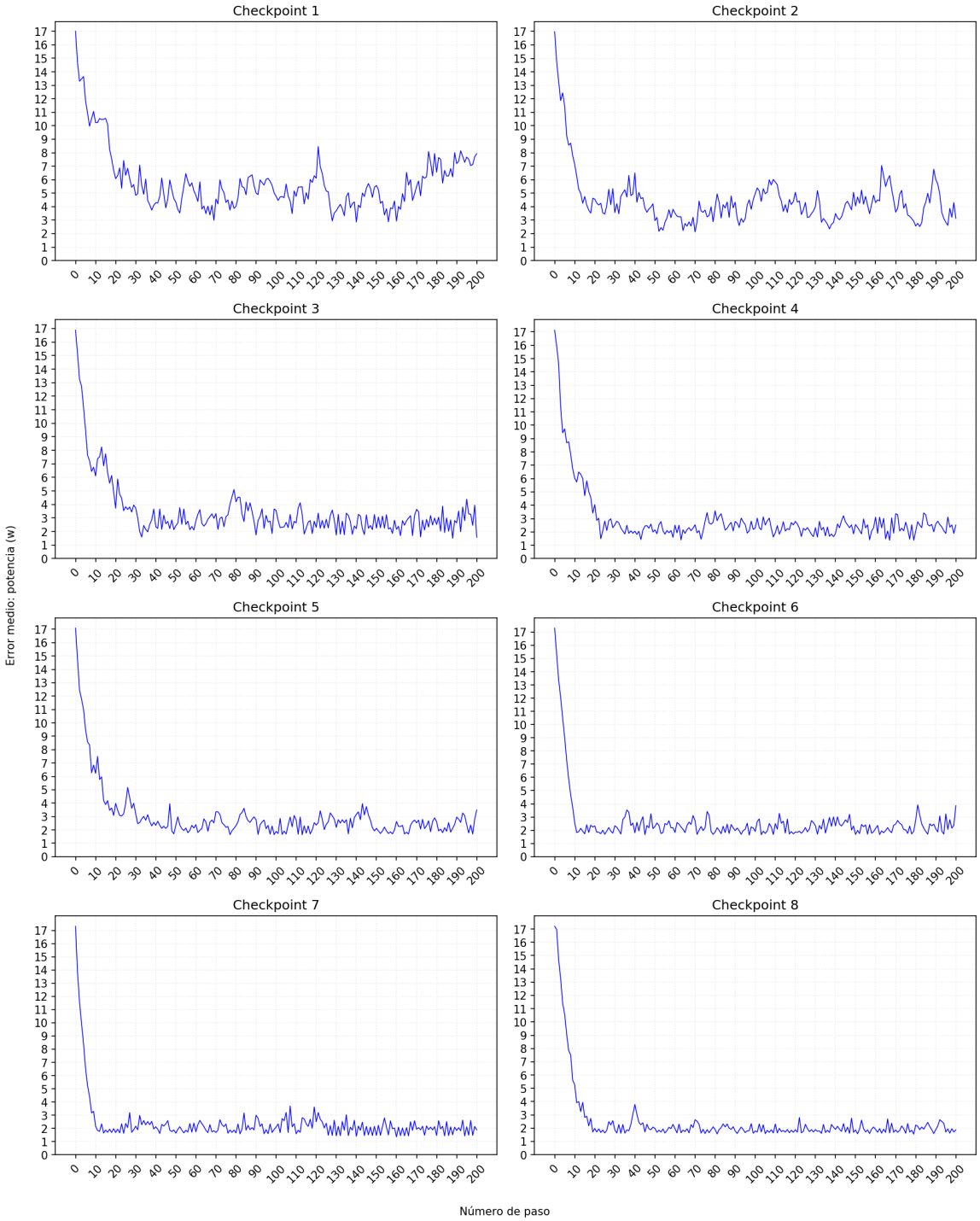
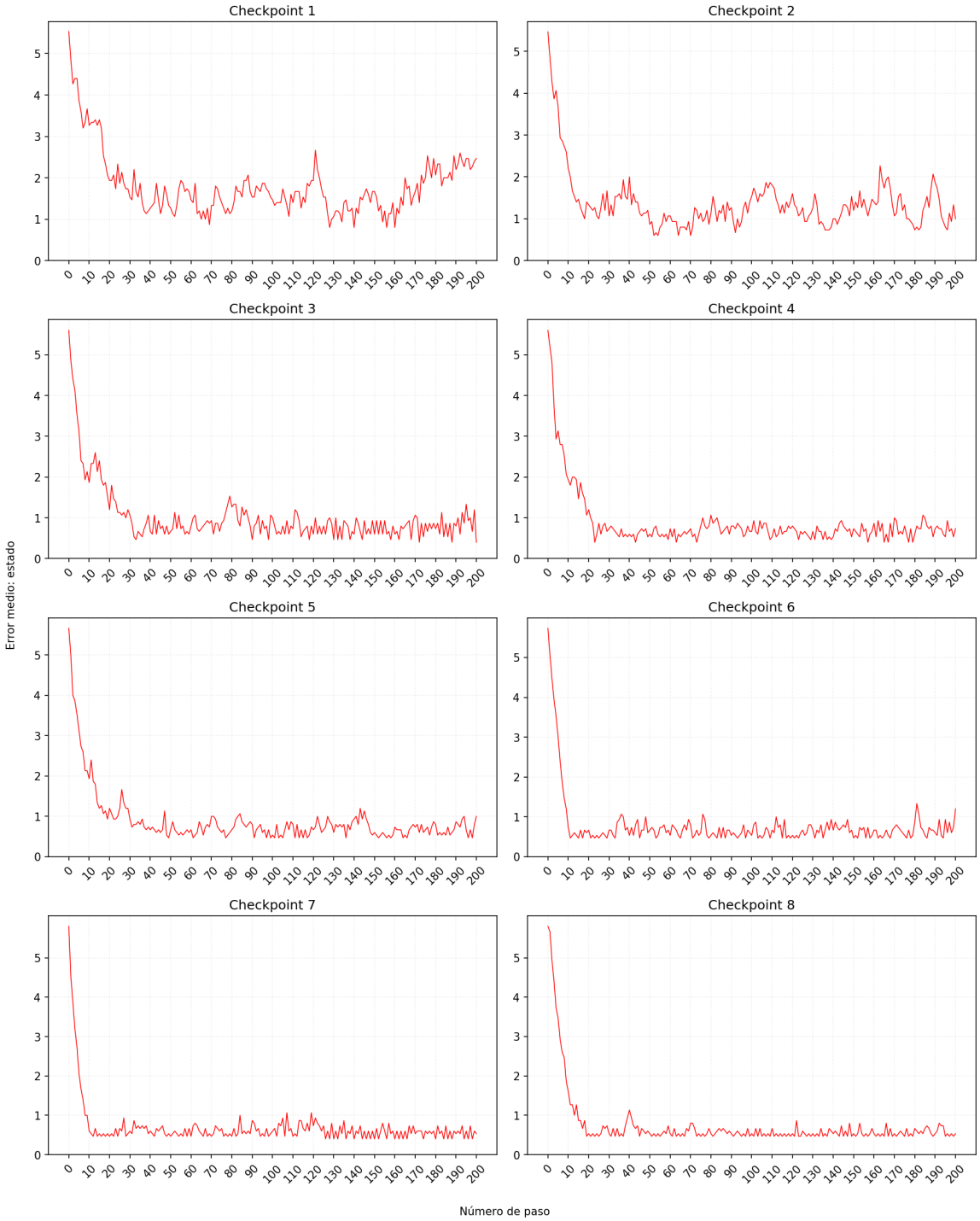


Figura 5.5
Gráfica del error medio del estado.



Así, para calcular un valor que represente la estabilidad, se obtendrá la media de la función de error medio a partir de un determinado paso $t = n$, en el que ya se haya producido la convergencia y la función oscile levemente en valores cercanos al error 0. Por las figuras, se podría tomar $t = 75$, y el valor para cada época j sería:

$$E^j = \frac{1}{201 - 75} \sum_{t=75}^{200} \varepsilon^j(t)$$

Se muestran estos valores calculados en las Figuras 5.6 y 5.7. Por tanto, una buena época para realizar los siguientes entrenamientos, sería la 4 o la 5; se elegirá la última para evitar que se den malos entrenamientos con configuraciones iniciales que puedieran ser peores.

Figura 5.6
Gráfica del error medio acumulado de la potencia.

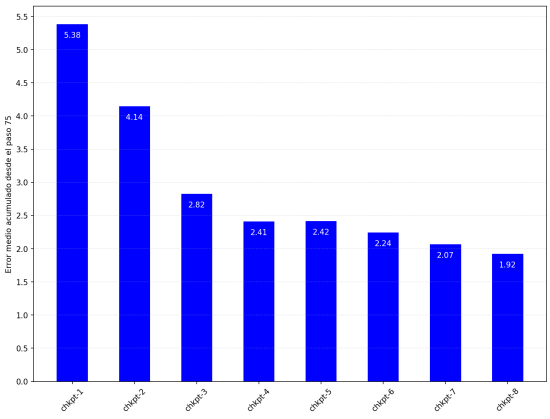
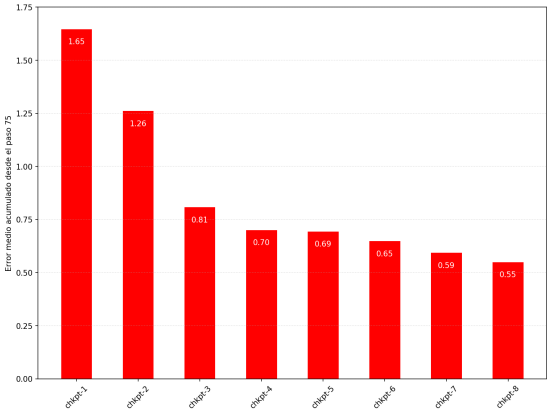


Figura 5.7
Gráfica del error medio acumulado del estado.



5.4. Efecto del tamaño del intervalo en el entrenamiento

Una vez establecido hasta qué época llevar a cabo los entrenamientos, junto con la definición de la métrica que aporta información sobre la estabilidad del agente en base a los resultados del testeo, cabe preguntarse cómo afecta al aprendizaje la configuración inicial del entorno.

Por ello, en este apartado, se llevará a cabo el entrenamiento de otros dos entornos de tipo `Env1`, pero tomando como tamaños de intervalo los valores 2 y 4. En el caso del valor más bajo, se podría dar la existencia de estados (intervalos) que prácticamente no se alcanzarán en tanto que su rango de potencia no se corresponde con ningún escalón del producto de matrices. Para el tamaño 4, el problema generado sería el contrario: la existencia de intervalos (estados), que alberguen dos escalones consecutivos de potencia de la carga de trabajo.

Así, se entrenará un agente para cada uno de estos 2 entornos y para la potencia objetivo de 52,5 vatios, de forma que podamos comparar los resultados con el agente entrenado en el apartado anterior para el entorno con tamaño de intervalo 3.

Resultados del entrenamiento

En primer lugar, se ha de tener en cuenta que la modificación del tamaño del intervalo hace que los entornos tengan espacios de observación distintos: para tamaño 2, se generarán más intervalos, mientras que para tamaño 4 se reducirá su número. Por tanto, los resultados en función del estado no son útiles en este caso.

Así, se procede directamente a analizar, para cada entorno, el error medio y su valor medio a partir del paso $t = 75$, que se muestran en las Figuras 5.8 y 5.9.

Figura 5.8
Gráfica del error medio de la potencia para cada tamaño de intervalo.

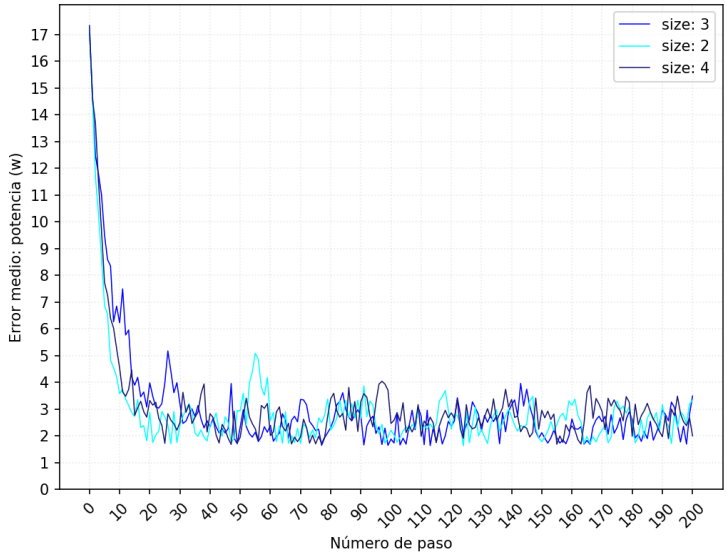
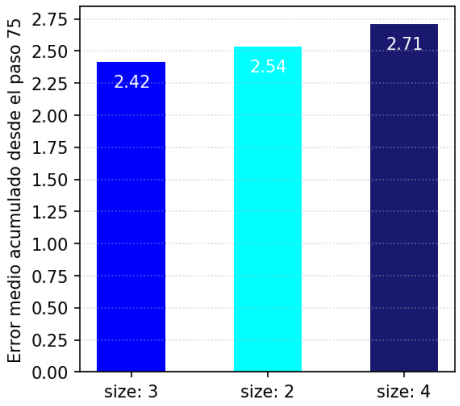


Figura 5.9
Gráfica del error medio acumulado de la potencia para cada tamaño de intervalo.



Como puede observarse, aunque el entorno de tamaño 3 tiene mejor rendimiento que los otros 2, la diferencia no es significativa. Seguramente, ello se debe a que la potencia elegida, 52,5 w, está asociada a intervalos en los que no se encuentra muy próxima a los extremos. Así es, para tamaño 2, se encuentra en el intervalo [51,0, 53,0], y para tamaño 4, tiene asociado el intervalo [51,0, 55,0], que además, no comparte con ningún otro escalón, como se puede comprobar en la Tabla 5.1 de la carga de trabajo.

5.5. Efecto de potencias límite en el entrenamiento

Continuando con el entorno de tipo `Env1` y tamaño de intervalo 3, en este apartado se comprobará cómo de bien entrena el agente cuando el valor de *power capping* es una potencia límite; esto es, los valores de potencia asociados a los escalones de frecuencia mínimo y máximo en el estudio energético de la carga de trabajo. Para el producto de matrices, estos valores serían 40,0 y 97,0 vatios.

Los entornos asociados a estas potencias son especiales, en tanto que cuando se están funcionando con la frecuencia asociada a su potencia, las acciones de bajar y subir frecuencia la mantienen en el caso de la potencia mínima y máxima respectivamente.

Además, se ha de tener en cuenta que a potencias más altas el calentamiento del procesador es mayor, hecho que genera mayores oscilaciones en el consumo energético, lo que podría afectar al aprendizaje.

Resultados del entrenamiento

En este caso, merece la pena observar el comportamiento del entorno para el testeo completo, como se muestra en las Figuras 5.10 y 5.11.

Figura 5.10
Gráfica de la potencia de las diferentes iteraciones para la potencia objetivo 40.0.

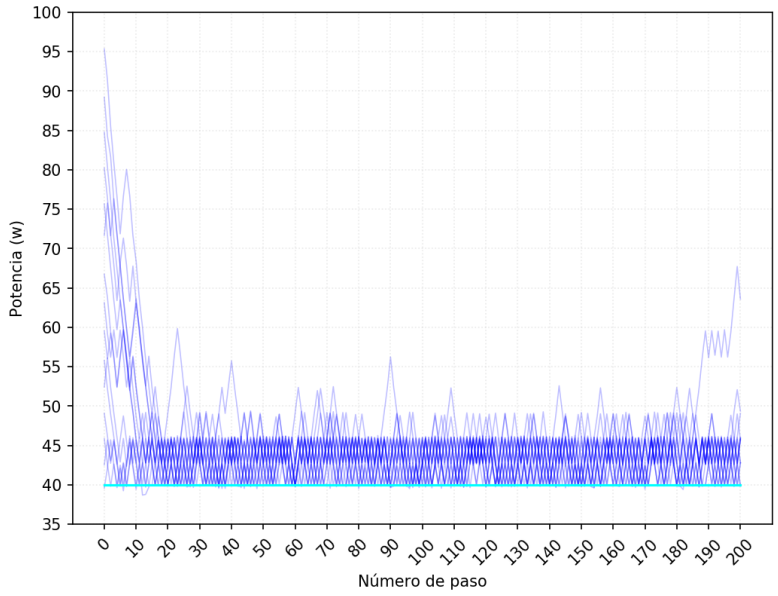
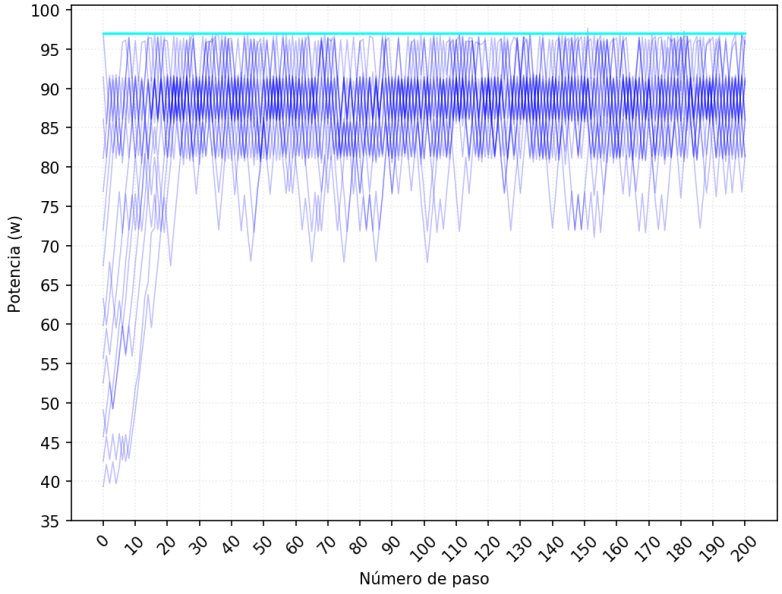


Figura 5.11
Gráfica de la potencia de las diferentes iteraciones para la potencia objetivo 97.0.



En ambos casos, la potencia no oscila en torno al objetivo, sino que lo hace por debajo y por encima, respectivamente. Además, para la potencia máxima, la variación es mucho mayor, de forma que la convergencia pierde estabilidad. Estas observaciones se perciben más claramente en la Figura 5.12, donde se contrapone el error medio de estos dos entornos al del entorno inicial del primer apartado, cuya potencia objetivo era de 52,5 vatios. Así, el valor medio de la gráfica anterior a partir de $t = 75$, mostrado en la Figura 5.13, presenta esa amplia diferencia de rendimiento entre el entorno de potencia máxima y los otros dos.

Figura 5.12
Gráfica del error medio de la potencia para cada potencia objetivo.

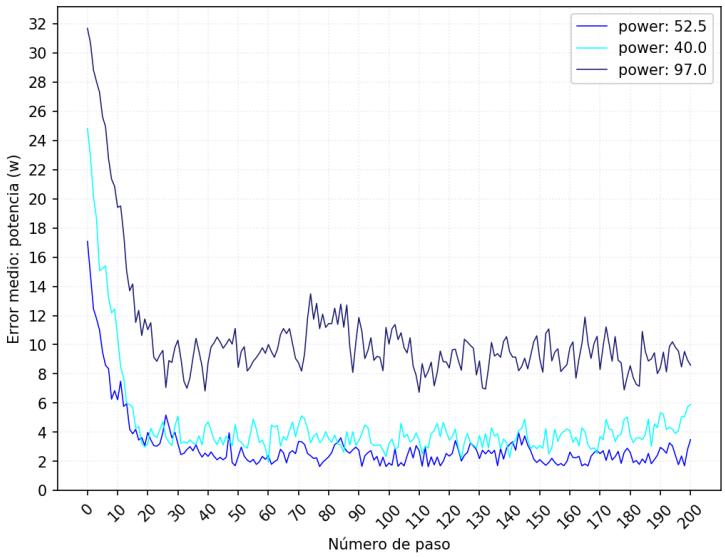
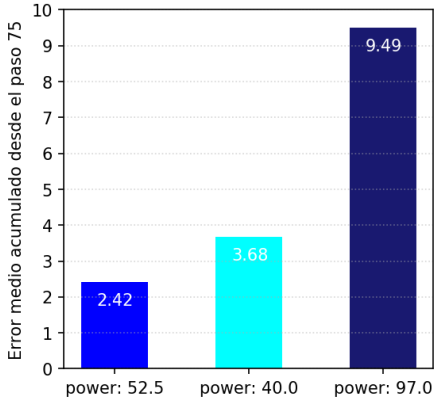


Figura 5.13
Gráfica del error medio acumulado de la potencia para cada potencia objetivo.



5.6. Conclusiones del capítulo

Partiendo del entorno experimental, hemos podido comprobar que el entrenamiento ha conseguido muy buenos resultados de convergencia a la potencia objetivo a partir de la tercera época. Por ello, se ha elegido la época número 5, como la época hasta la que se ha de llegar en futuros entrenamientos.

Posteriormente, el estudio sobre el efecto del tamaño de los intervalos nos ha enseñado que para el tamaño 3 el agente tiene un ligero mejor rendimiento que para los otros dos tamaños 2 y 4. No obstante, no se ha dado un resultado significativo que nos permita afirmar sin ninguna duda que el tamaño 3 sea la mejor elección. Por ello, al ser ligeramente mejor, continuaremos utilizándolo.

Finalmente, el análisis sobre el entrenamiento con potencias objetivo límites ha mostrado que este entorno funciona peor a la hora de entrenar teniendo estos valores extremos como objetivos.

Capítulo 6

Análisis de posibles mejoras del entorno

En este capítulo, se expondrán 2 nuevos entornos surgidos de realizar algunas modificaciones al entorno inicial `Env1`, con el objetivo de mejorar el rendimiento del agente entrenado.

6.1. Incorporación de una acción para mantener la frecuencia

El nuevo entorno, que denotaremos por el identificador `Env2`, es prácticamente equivalente al del capítulo anterior, con tamaño de intervalo 3, y una única variación: la incorporación de una tercera acción que mantiene la frecuencia del entorno.

La aplicación de esta acción sería parecido a mantener el estado del entorno, partiendo de la suposición de que al mantener la frecuencia, el consumo medido será el mismo. No obstante, aunque esto pueda ocurrir la mayoría de veces, la volatibilidad del procesador puede hacer que se produzca un cambio de intervalo, o sea, de estado. Este hecho ocurrirá en mayor medida en aquellos escalones de frecuencia cuya potencia asociada esté próxima al extremo compartido de dos intervalos consecutivos.

Sin embargo, la integración de esta nueva acción debería favorecer a aquellos entornos cuya potencia objetivo esté lo suficientemente distanciada de los extremos de su intervalo.

En concreto, se compararán los resultados de este entorno con los del apartado anterior para las potencias objetivo 52,5, 40,0, 97,0.

Resultados del entrenamiento para una potencia objetivo intermedia

Para 52,5 vatios, se dan las siguientes funciones de error medio para potencia y estado, mostradas en las Figuras 6.1 y 6.2. En este caso, y también para las otras dos potencias, la información del estado es útil, pues los entornos comparten el mismo estado objetivo.

Figura 6.1
Gráfica del error medio de la potencia para la potencia objetivo 52,5.

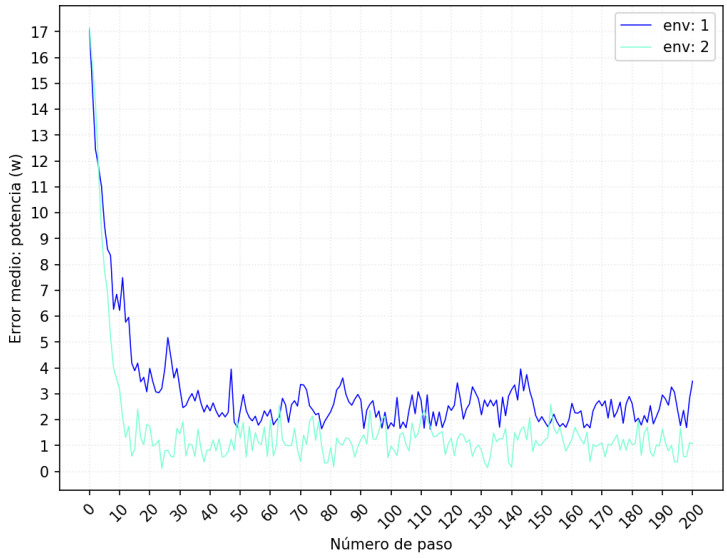
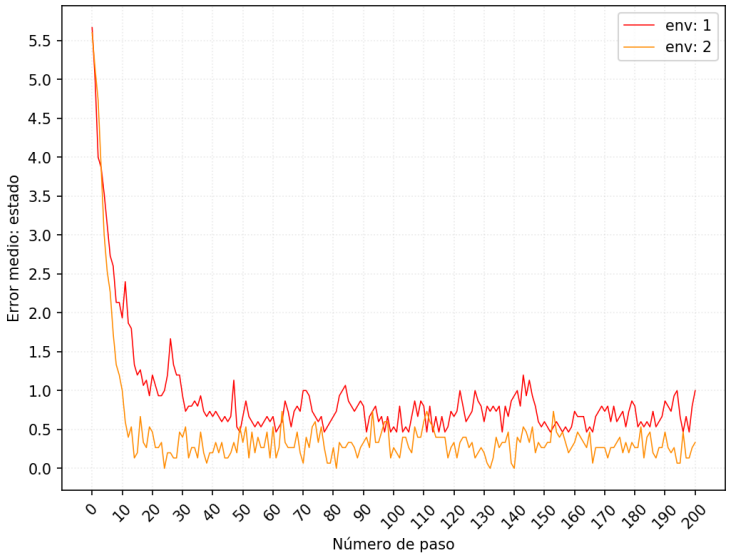


Figura 6.2
Gráfica del error medio del estado para la potencia objetivo 52,5.



Como se puede contemplar en las gráficas, el nuevo entorno consigue resultados más pequeños para el error medio, por lo que es más estable respecto a `Env1`. Esto también es visible la mejora en el valor medio de las funciones a partir del paso $t = 75$, como muestran las Figuras 6.3 y 6.4.

Figura 6.3
Gráfica del error medio acumulado de la potencia para la potencia objetivo 52,5.

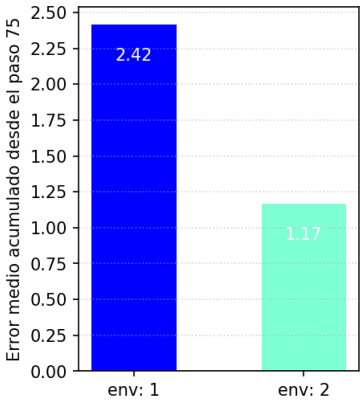
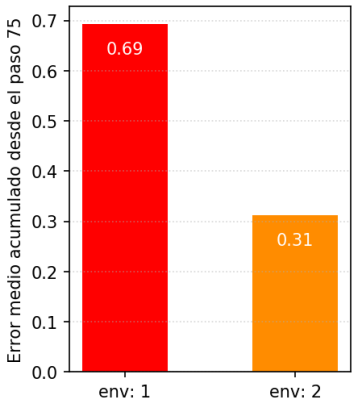


Figura 6.4
Gráfica del error medio acumulado del estado para la potencia objetivo 52,5.



En efecto, los resultados para potencia y estado son directamente proporcionales. Por tanto, para no sobrecargar con la memoria, sólo se mostrarán los resultados asociados a la potencia para los entornos siguientes.

Resultados del entrenamiento para la potencia objetivo inferior

Para 40,0 vatios, la funciones de error medio para potencia está reflejada en la figura 6.5, junto con su error medio acumulado en la Figura 6.6. En este caso, la mejora respecto al entorno `Env1` es ligeramente mejor que para la potencia objetivo intermedia.

Figura 6.5
Gráfica del error medio de la potencia para la potencia objetivo 40,0.

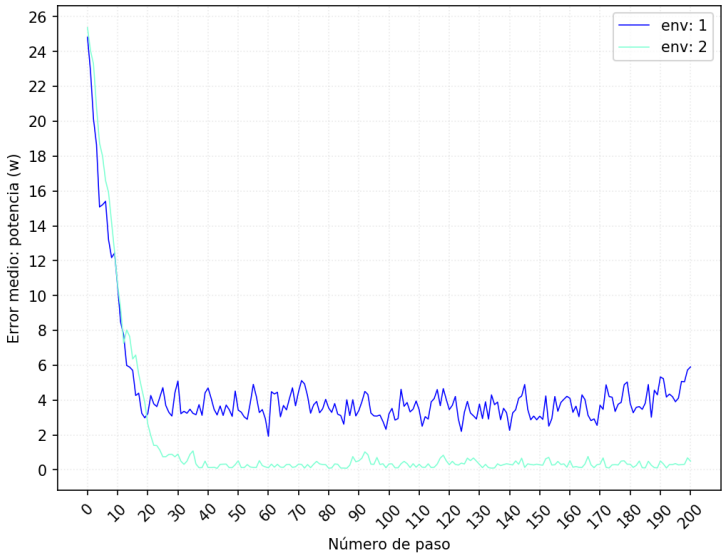
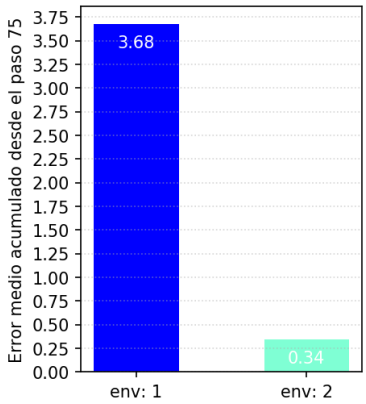


Figura 6.6
Gráfica del error medio acumulado del estado para la potencia objetivo 40,0.



Resultados del entrenamiento para la potencia objetivo superior

Para 97,0 vatios, se representa la funciones de error medio para potencia en la figura 6.7, y el error medio acumulado en la Figura 6.8. Como puede observarse, la mejora es ampliamente significativa; de esta forma, el entorno `Env2` solucionaría el problema de aprendizaje del entorno anterior con esta potencia objetivo.

Figura 6.7
Gráfica del error medio de la potencia para la potencia objetivo 97,0.

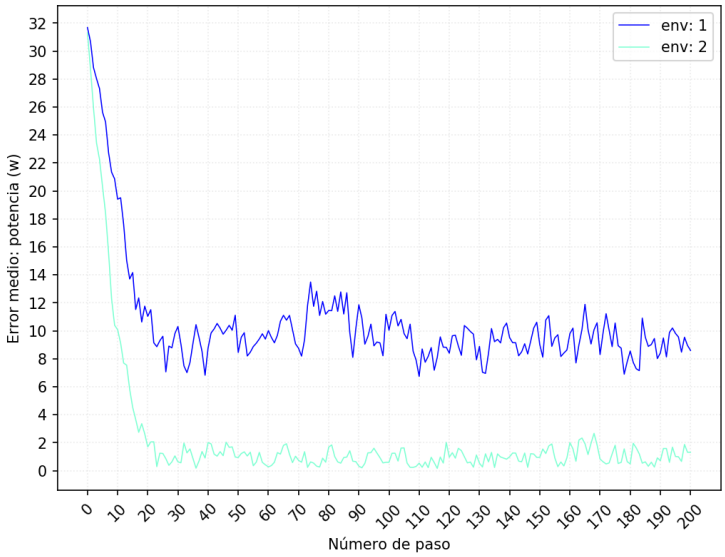
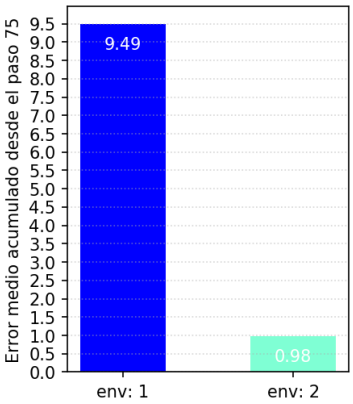


Figura 6.8
Gráfica del error medio acumulado del estado para la potencia objetivo 97,0.



6.2. Uso de intervalos personalizados asociados a la carga de trabajo

En el entorno de este apartado, que denotaremos por el identificador `Env3`, partiremos del entorno del apartado anterior, con la acción de mantener la frecuencia, modificando el espacio de observaciones. En particular, se eliminará el parámetro de tamaño fijo de los intervalos, inicializando el entorno con intervalos personalizados.

Estos intervalos estarán ajustados a las potencias que generan los escalones de frecuencia con la carga de trabajo del producto de matrices, de forma que estos valores se sitúen en el centro de los intervalos. Mediante esta modificación, se reduce considerablemente el número de estados (concretamente a 15, el número de escalones de frecuencia).

Se compararán los resultados de este entorno con los de los entornos anteriores, `Env1` y `Env2`, también para las potencias objetivo 52,5, 40,0, 97,0. Nótese, que puesto que se producirá un cambio en el tamaño del espacio de observaciones, la información relativa al estado del entorno resulta inútil para establecer comparaciones. Así, solo se enseñarán los resultados asociados a la potencia.

Resultados del entrenamiento para las potencias objetivo

Respecto al error medio, se muestran las gráficas: para la potencia 52,5 en la Figura 6.9, para la potencia 40,0 en la Figura 6.10 y para la potencia 97,0 en la Figura 6.11. Como se puede comprobar, en todas las gráficas se presenta una mejora respecto al entorno `Env1`. No obstante, el rendimiento es bastante parejo al que muestra el entorno `Env2`.

Figura 6.9
Gráfica del error medio de la potencia para la potencia objetivo 52,5.

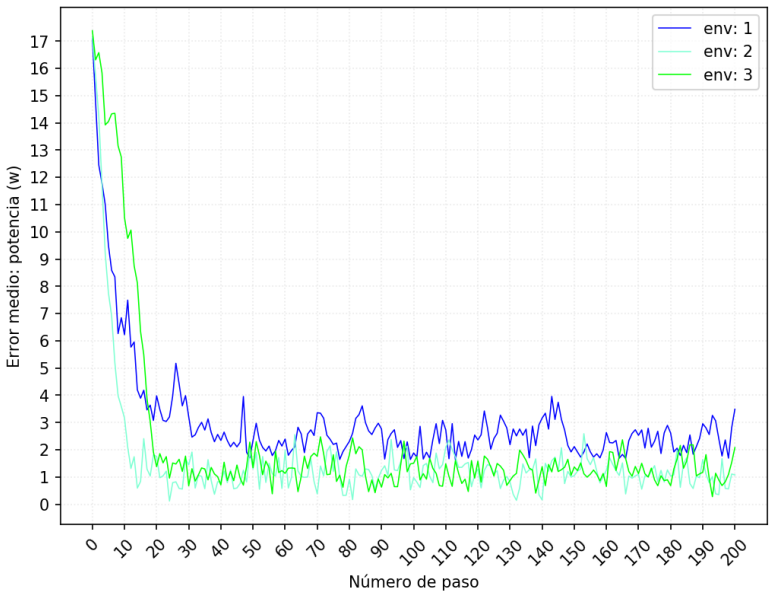


Figura 6.10
Gráfica del error medio de la potencia para la potencia objetivo 40,0.

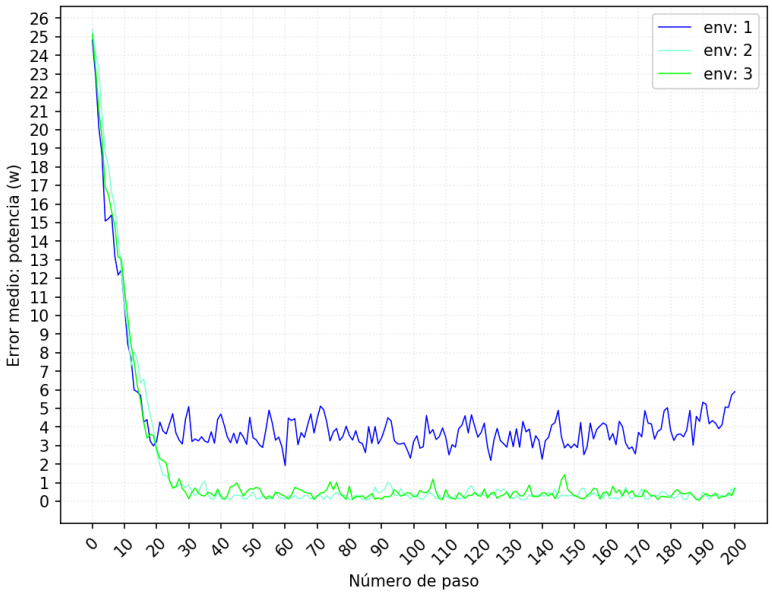
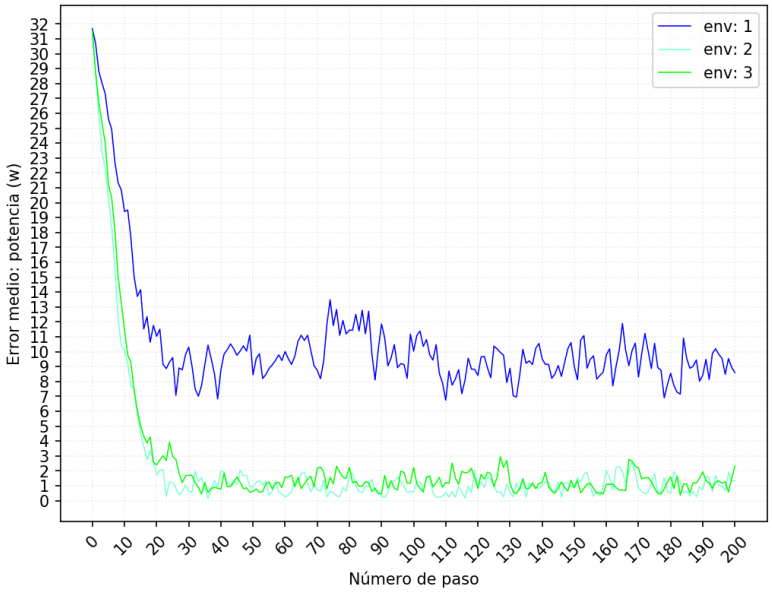


Figura 6.11
Gráfica del error medio de la potencia para la potencia objetivo 97,0.



Efectivamente, esto se ve confirmado al tomar el valor medio de las funciones a partir del paso $t = 75$, cuyo resultado aparece en las Figuras 6.12, 6.13 y 6.14, asociadas a las potencias 52,5, 40,0 y 97,0, respectivamente.

Figura 6.12
Gráfica del error medio acumulado del estado para la potencia objetivo 52,2.

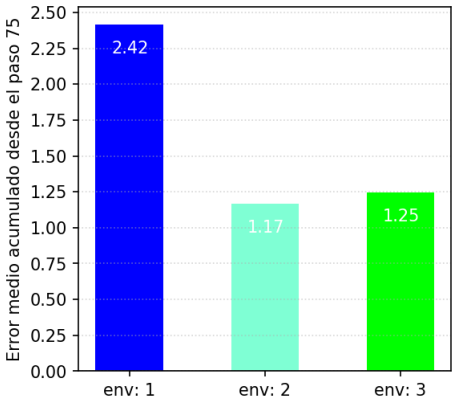


Figura 6.13
Gráfica del error medio acumulado del estado para la potencia objetivo 40,0.

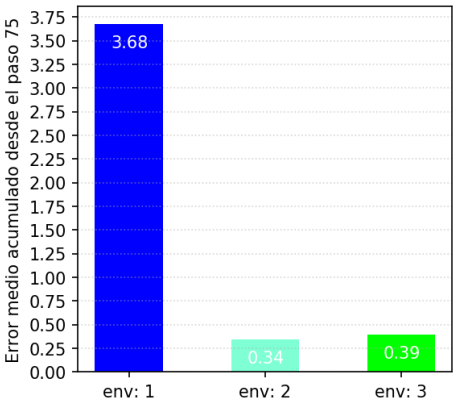
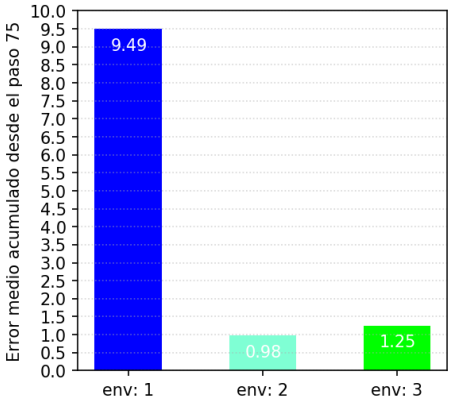


Figura 6.14
Gráfica del error medio acumulado del estado para la potencia objetivo 97,0.



6.3. Conclusiones del capítulo

La introducción de la acción para mantener la frecuencia ha tenido un impacto significativo en la mejora del rendimiento. En particular, se ha dado una mejora importante para la frecuencia intermedia 52,5, pero esta ha sido aun mayor para las potencias objetivo límite. Podemos afirmar, por tanto, que en este nuevo entorno `Env2` se ha arreglado el problema que tenía el entorno `Env1` a la hora de entrenar para valores límite.

Respecto al uso de los intervalos personalizados para la carga de trabajo `producto`, este cambio no ha tenido apenas importancia, al no conseguir siquiera resultados mejores que el entorno `Env2`. Por tanto, a pesar de que esta modificación servía para reducir el espacio de observación, esto no parece haber tenido efectos notables en el rendimiento.

Capítulo 7

Validación de los entornos bajo otras cargas de trabajo

En este capítulo, se presentarán 2 nuevas cargas de trabajo basadas en el módulo *NumPy*, caracterizadas por tener un rango energético más reducido que la carga asociada al producto de matrices del Capítulo 4.

Seguidamente, se utilizarán los agentes entrenados con el entorno `Env2`, empleando como carga el producto de matrices, para realizar testeos con las nuevas cargas de trabajo ejecutándose sobre el procesador. El análisis de estas pruebas revelará si los agentes son igual de funcionales a la hora de lidiar con cargas de trabajo menos energéticas.

7.1. Introducción y análisis de nuevas cargas de trabajo

Para implementar las nuevas cargas de trabajo, se empleará otra vez el módulo *NumPy*. Una forma de reducir el consumo energético de la carga `producto` es eliminar las operaciones con números de tipo `float` de la carga de trabajo. Así, no se utilizarán las unidades funcionales destinadas a `float` del procesador, produciendo un ahorro energético.

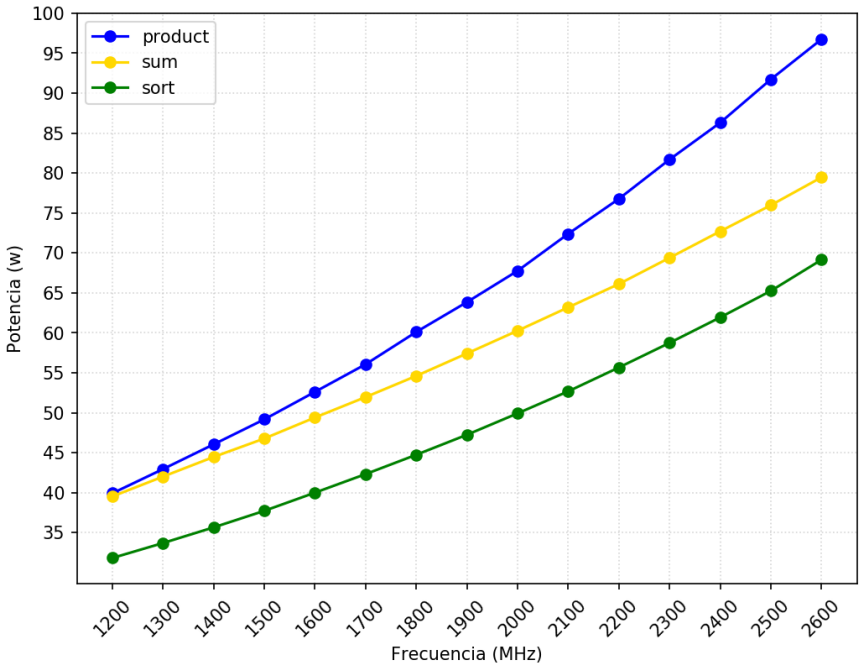
Con esto en mente, se plantea la primera nueva carga de trabajo basada en la suma de matrices realeas, que se identificará con el nombre de `suma`. En este caso, se repite la inicialización aleatoria de dos matrices reales A, B y se almacena su suma $A + B$ en una matriz destino C. De esta forma, esta operación elimina las multiplicaciones de números `float`.

Prosiguiendo con la reducción de operaciones con `float`, se define la segunda nueva carga de trabajo mediante la operación que ordena una *array* de números reales, identificada por el nombre `sort`. Así, se inicializa un *array* X y se aplica la función `sort()` del módulo *NumPy*. En definitiva, se obtiene una carga de trabajo sin operaciones aritméticas con `float`, siendo las comparaciones y las operaciones de memoria las principales acciones de consumo.

Para llevar a cabo el estudio energético, se emplean matrices cuadradas de dimensión 1000×1000 para `suma`, mientras que se emplear un *array* de 1000000 elementos para la carga `sort`. Con ello,

se obtienen los datos de potencia para cada escalón de frecuencia y carga de trabajo, representados gráficamente en la Figura 7.1.

Figura 7.1
Gráfica del análisis energético de las distintas cargas de trabajo.



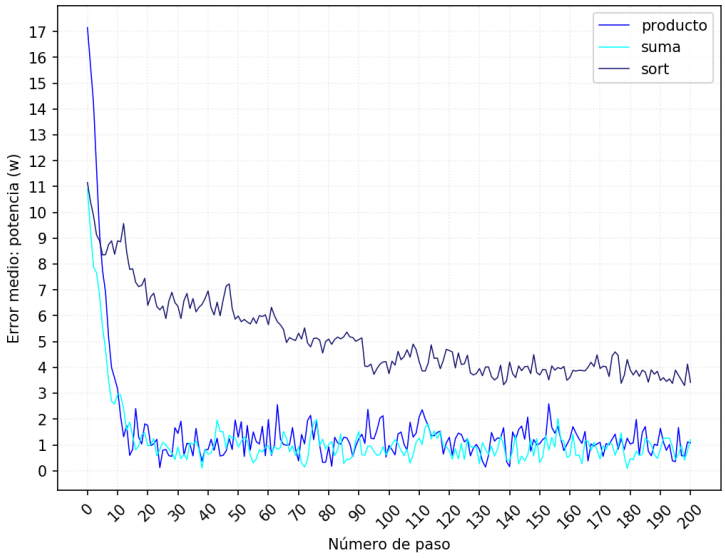
7.2. Eficiencia del agente con las nuevas cargas de trabajo

Como ya se ha comentado, se empleará un agente entrenado con un entorno de tipo `Env2` con la carga de trabajo `producto`. En concreto, se tomará el agente entrenado en el Capítulo 5 para la potencia objetivo 52,5. Así, una vez realizadas las pruebas con las cargas de trabajo `suma` y `sort`, se representan las funciones de error medio de la potencia para cada una de las 3 cargas de trabajo en la Figura 7.2.

En la figura se observa que para la carga `suma` el agente mantiene su funcionalidad holgadamente, consiguiendo resultados prácticamente equivalentes a la carga `producto` original. Sin embargo, para la carga `sort` se produce un error medio mucho mayor, dando a entender que el agente no es trasladable a esta carga.

Para justificar la veracidad de los datos, podemos emplear el valor de la frecuencia del entorno. Esto es, si se está realizando el test con una carga distinta a `producto`, entonces la frecuencia del entorno convergerá a un valor distinto que el original, pues las cargas tienen distintos escalones de frecuencia para la misma potencia objetivo.

Figura 7.2
Gráfica del error medio de la potencia para las distintas cargas de trabajo.



Así, se muestra el valor de la frecuencia para cada iteración en las Figuras 7.3 y 7.4 para las cargas `suma` y `sort`, respectivamente.

Figura 7.3
Gráfica de la frecuencia para las distintas iteraciones con la carga `suma`.

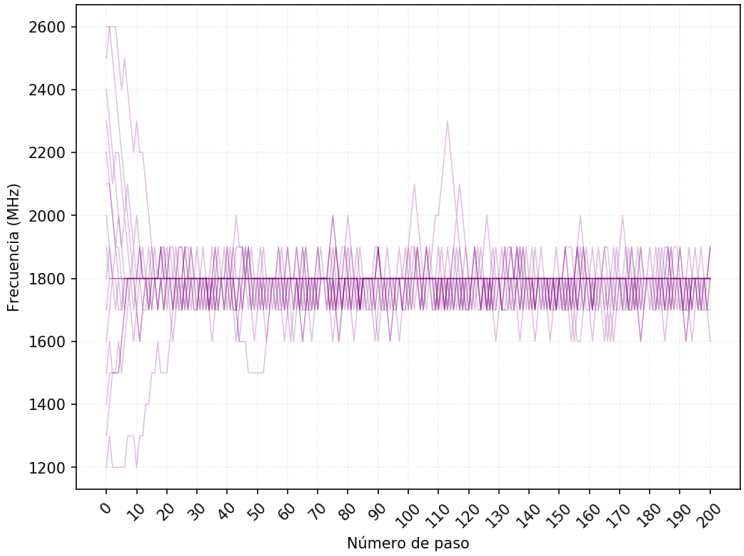
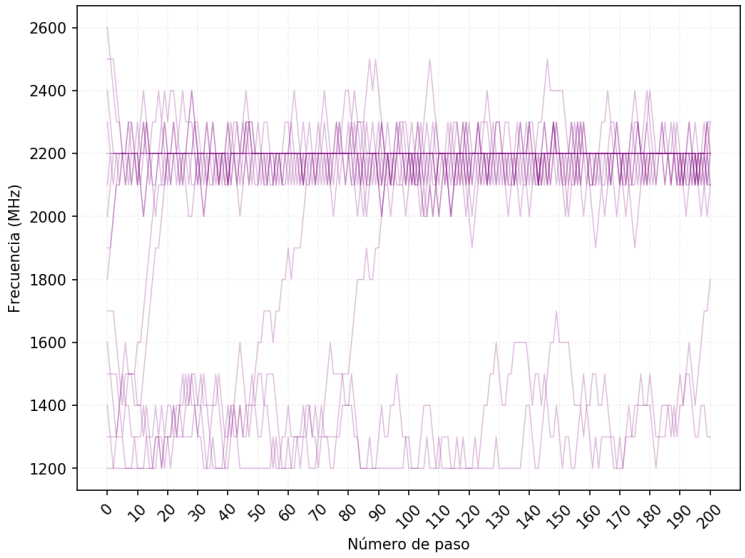


Figura 7.4
Gráfica de la frecuencia para las distintas iteraciones con la carga `sort`*.*



Por un lado, todas las iteraciones del entorno asociado a la carga `suma` convergen a un rango de frecuencias entre 1700 y 1900 MHz. Se puede comprobar en la Figura 7.1 que las potencias asociadas a estos escalones de frecuencia se hallan en torno a la potencia objetivo de 52,5 W.

Por otro lado, se da un resultado parecido para las iteraciones del entorno testeado con la carga `sort`, pero únicamente aquellas que comienzan con una frecuencia inicial superior a 1700 MHz. Las iteraciones con frecuencias iniciales por debajo de este valor no parecen ser capaces de converger al rango de frecuencias asociado a la potencia objetivo. Este hecho es fácil de comprender mirando la Figura 7.1: hay un número importante de escalones de frecuencia cuya potencia asociada para `sort` es menor que la potencia mínima que se da para la carga `producto`. Así, cuando se entrena el agente con la carga del producto de matrices, el entorno nunca podrá explorar los intervalos asociados a estas potencias tan bajas, por lo que el agente no habrá *aprendido* qué acciones aplicar cuando se le presenten algunos de estos estados como observación.

7.3. Conclusiones del capítulo

La validez de los agentes entrenados para ser utilizados con otras cargas de trabajo distintas es importante, pues implica que podemos generalizar nuestros agentes entrenados, sin necesidad de tener que repetir el entrenamiento para cada carga de trabajo distinta.

No obstante, el comportamiento no es ideal, como se ha podido comprobar con la carga de trabajo `sort`. Puesto que esta carga toma valores de potencia que están por debajo de la potencia mínima de la carga `producto`, el agente entrenado con esta última nunca ha recibido ningún tipo de *feedback* durante el entrenamiento sobre qué acciones tomar en esos estados menos energéticos.

De ahí el comportamiento impredecible del agente para `sort` cuando este empieza en frecuencias iniciales bajas

Capítulo 8

Conclusiones generales

Se ha demostrado que el problema de limitación de consumo o *power capping* puede ser abordado desde la perspectiva del aprendizaje reforzado y a nivel de *software*. Gracias a la enorme comunidad del lenguaje de programación `Python`, ha sido una tarea sencilla encontrar las herramientas para llevar a cabo esta propuesta.

Por un lado, tenemos los módulos destinados a resolver problemas de Inteligencia Artificial, como *GYM* y *RLlib*, este último proporcionando una amplia colección de complejos algoritmos de aprendizaje ya implementados y listos para usar. Por otra parte, tampoco ha resultado difícil encontrar los módulos más orientados a tratar con el *hardware*, como *cpufreq* y *pyRAPL*. Gracias a ellos, podemos modificar las frecuencias del procesador y medir su consumo energético directamente en Python, sin tener que interactuar a un nivel más bajo con el sistema operativo.

Moviéndonos ya al aprendizaje, se ha podido comprobar el potencial que ofrece la colaboración entre los módulos de *GYM* y *RLlib*, pues a partir de un primer entorno experimental adaptado al problema de *power capping*, los buenos resultados del entrenamiento han sido inmediatos. Gran parte de este mérito pertenece también al algoritmo PPO, que demuestra ser una poderosa herramienta para abordar problemas de aprendizaje reforzado.

Respecto al estudio de posibles mejoras del entorno, la incorporación de una acción que permitiese mantener la frecuencia se ha probado enormemente efectiva. No ha ocurrido así con el empleo de un espacio de observación formado por intervalos adaptados a la huella energética de la carga de trabajo, que no ha aportado ninguna mejora significativa con respecto a la idea de un tamaño de intervalo fijo.

Finalmente, ha resultado satisfactoria la comprobación de que un agente ya entrenado puede mantener su funcionalidad incluso trabajando con cargas de trabajo con una huella energética distinta a la carga empleada durante su entrenamiento. No obstante, se ha podido ver también que su rendimiento se verá perjudicado si se trabaja con cargas de trabajo cuya huella energética sobrepasa el rango de potencia de la carga de entrenamiento.

Ideas para la continuación del trabajo

Ha quedado abierta, sobre todo, la exploración de otras posibles modificiones que pudisen aplicarse al entorno para intentar mejorar su rendimiento en el entrenamiento. Respecto al espacio de observación, cabría la posibilidad de cambiar el espacio discreto por uno continuo, en el que se tuviese en cuenta todo el rango de potencia del procesador. Por otro lado, también sería posible incorporar nuevas acciones que pudiesen saltar más de un escalón de frecuencia en cada paso, aunque ello haría más grande y complejo al espacio de acciones, lo que podría afectar al desarrollo del entrenamiento.

Tampoco se ha tratado ninguna variación que afectase al modo en que se calculan las recompensas en el entorno experimental. No obstante, se podrían probar magnitudes distintas, más o menos lejanas entre sí, o incluso modificar la forma en que se calcula la recompensa (por ejemplo, que esta dependiese de la distancia a la potencia objetivo).

Finalmente, se podrían plantear distintas propuestas para solucionar el problema que aparece al utilizar a un agente con otra carga de trabajo que se sale del rango energético de la carga de trabajo empleada en el entrenamiento. Una opción sería la de confeccionar una carga de trabajo cuya huella energética abarque todo el rango de potencia del procesador. No obstante, este planteamiento podría ser problemático al usar al agente con cargas de trabajo con una rango energético pequeño, dos escalones consecutivos de frecuencia podrían tener potencias asociadas contenidas en un mismo intervalo. Una alternativa a este problema sería utilizar 2 agentes, uno entrenado con una carga de trabajo que cubra el espectro energético inferior, y el otro entrenado con otra carga más energética que cubra la parte superior del espectro.

En definitiva, el uso del aprendizaje reforzado deja un amplio abanico de opciones a explorar.

Capítulo 9

General conclusions

It has been proved that the problem of power capping can be addressed from the perspective of Reinforcement Learning, and at software level too. Thanks to the huge community of the `Python` programming language, it has been an easy task to find the necessary tools to implement this approach.

On one hand, we found the modules intended for solving problems in Artificial Intelligence, such as *GYM* and *RLlib*, the last one containing a large collection of complex algorithms already implemented and ready to use. On the other hand, finding the modules needed to deal with hardware, such us *cpufreq* and *pyRAPL*, has not been difficult neither. These allowed us to modify the processor's frequencies as well as measuring its power consumption directly through `Python`, freeing us from dealing directly with the operating system.

Changing the subject to Reinforcement Learning, it has been quite noticeable the potential of the collaboration between *GYM* and *RLlib*. Starting with an experimental environment adapted to the power capping problem, the agent training delivered immeadiate positive results. A big proportion of this merit must be given to the PPO algorithm, which has proved itself to be a powerful tool to address problems in Reinforcement Learning.

Regarding the study of finding possible improvements to the environent, the addition of an action that allowed the environment to hold its frequency has proved greatly valuable. That has not been the case with the modification of the intervals in the observation space, so that they were adapted to energetic footprint of the workload. This approach did not provide a significant improvement with respect to the same-sized intervals idea.

Finally, we got the pleasant confirmation that an already trained agent can keep its functionality even dealing with workloads whose energetic footfrint differs from the one workload used during training. However, its performance is dramatically affected if the agent deals with a workload whose consumption range goes beyond the training workload's one.

Possible courses of action

The exploration of other possible changes that could be applied to the environment to try to improve its performance is still open. Concerning the observation space, it could be possible to substitute the discrete space with a continuous one, in which the whole power spectrum of the processor can be considered. On other side, new actions could be included too such that the frequency jumps could take more than one step. However, this would enlarge the action space and make it more complex, which could affect the development of the training.

The rewards used by our environment have not been questioned yet. Changes could be made to the magnitude of the rewards value, and to the way the rewards are calculated too. For example, the reward could be reliant on the distante to the target power.

At last, some suggestions could be made to overcome the existing problem when a trained agent is used with a workload whose energetic footprint is beyond the spectrum of the workload used to train that agent. An option would be to make up a workload whose energetic consumption covers the whole power spectrum of the processor. Nevertheless, this would occasionate problems when working with workloads with small energetic range, being possible that 2 consecutive frequency steps map their associated powers to the same interval. Then, a possible solution could be using 2 agents, one in charge of workloads with a small consumption, the other covering the rest of the spectrum.

In the end, the use of Reinforcement Learning produces a wide range of options to explore.

Bibliografía

[1] Ted Samson. Power capping yields savings and floor space, 2009.

[2] Wikipedia. Reinforcement Learning.

[3] Dhairya Parikh. Learning Paradigms in Machine Learning, 2018.

[4] Documentación de *GYM*.

[5] Documentación de *Ray*.

[6] Documentación de *RLlib*.

[7] Andrej Karpathy. Deep Reinforcement Learning: Pong from Pixels, 2016.

[8] John Schulman, Oleg Klimov, Filip Wolski, Prafulla Dhariwal, and Alec Radford. Publicación de *OpenAI* sobre el algoritmo PPO.

[9] DerwenAI. Example implementation of an OpenAI Gym environment, to illustrate problem representation for RLlib use cases.

[10] Vitor Ramos and Alex Furtunato. Documentación de *cpufreq* en PyPI.

[11] ArchLinux. Scaling governors.

[12] powerapi. Documentación de *pyRAPL* en PyPI.