Facultad
de
Ciencias

Gestión de recursos en Datacenters basado en Deep Reinforcement Learning

(Management resources in Datacenters based on Deep Reinforcement Learning)

Trabajo de Fin de Grado
para acceder al

GRADO EN INGENIERÍA INFORMÁTICA

Autor: Mario Ibáñez Bolado

Director: José Luis Bosque Orero

Junio-2020

Índice general

Índice de Figuras **V**

Índice de Tablas **VI**

Agradecimientos **VII**

Resumen **VIII**

Abstract **IX**

1. Introducción **1**

1.1. Planificación de tareas en datacenters 1

1.2. Machine Learning 3

1.3. Objetivos 3

1.4. Metodología y Plan de trabajo 4

1.5. Estructura del Documento 5

2. Background **6**

2.1. Conceptos Básicos 6

2.1.1. Deep Reinforcement Learning 6

2.1.2. Red Neuronal 7
2.2. HDeepRM 8
2.3. Trabajos Relacionados 11

3. Aportaciones al Diseño e Implementación de HDeepRM 13

3.1. Escalabilidad del sistema de planificación 13
3.2. Diseño de Objetivos de Planificación 16
3.2.1. Consumo energético 17
3.2.2. Eficiencia Energética 21
3.3. Escenario Dinámico 22
3.3.1. Cambio de Objetivos 22
3.3.1.1. Red adaptable al objetivo 23
3.3.1.2. Reinicio del agente 24
3.3.2. Cambio de Workloads 25

4. Evaluación 27

4.1. Metodología 27
4.2. Experimentos de escalabilidad 30
4.2.1. Experimentos básicos 30
4.2.2. Escalado de Plataformas y Workloads 32
4.3. Experimentos de Objetivos de Energía y EDP 35
4.4. Experimentos realistas 38
4.4.1. Experimento Básico 38
4.4.2. Entornos dinámicos: cambio de objetivos 40

4.4.3. Entornos dinámicos: cambio de workload 42

5. Conclusiones **46**

5.1. Objetivos Conseguidos 46

5.2. Trabajos Futuros 47

Índice de figuras

2.1. Bucle del proceso de Deep Reinforcement Learning 7

2.2. Red Neuronal 7

2.3. Representación del datacenter en HDeepRM [1] 9

3.1. Método forward con sus respectivos procesos de *forwarding* 15

3.2. Preferencia de acciones del ejemplo error de la normalización del *reward* - 700 episodios 15

3.3. Método del reward makespan 17

3.4. Ejemplo de explicación del tiempo del reward 19

3.5. Código para cambiar el tiempo del *reward energy_consumption* 20

3.6. Ejemplo de fichero de configuración en formato JSON 23

3.7. Preferencia de acciones del ejemplo para el modelo de red adaptable al objetivo . 24

4.1. Preferencia de acciones del experimento básico con 4 cores - 200 episodios 31

4.2. Makespan de las 5 políticas en la plataforma con dos procesadores 32

4.3. Pérdidas de la red del experimento básico con 4 cores - 200 episodios 32

4.4. Experimento básico de escalabilidad con 2046 cores 33

4.5. Experimento heterogéneo con 88 cores 34

4.6. Makespan para las políticas en la plataforma de 88 cores 34

4.7. Experimento heterogéneo con 5632 cores 35

4.8. Preferencia de acciones y gasto energéticos de las políticas para el objetivo de *energy_consumption* 37

4.9. Experimento de comprobación del objetivo *energy_consumption* con 5632 cores . . 38

4.10. Comparación del Makespan de las Políticas Clásicas en el datacenter Gaia 39

4.11. Experimento en el cluster de Gaia 40

4.12. Preferencias de acciones en los cambios de objetivos con 5632 cores - 1400 episodios 42

4.13. Experimento de cambio de objetivos con 5632 cores 42

4.14. Políticas Clásicas para los dos tipos de workloads 43

4.15. Preferencia de acciones del experimento de cambio de workload para 8 cores - 1500 episodios 44

4.16. Pérdidas de la red del experimento de cambio de workload para 8 cores - 1500 episodios 44

4.17. Políticas Clásicas para los dos tipos de workloads en el experimento con 2816 cores 45

4.18. Preferencia de acciones del experimento de cambio de workload para 2816 cores - 1000 episodios 45

Índice de cuadros

4.1. Configuración del UniLu Gaia cluster. 39

Agradecimientos

A mi familia, que me ha dado la oportunidad con su trabajo de llegar hasta donde estoy en mis estudios y de apoyarme en mis decisiones sobre mi futuro.

A mis amigos, con los que he pasado tan buenos momentos y en especial a mis amigos Dani y Robert. Dani por siempre estar ahí para desconectar y pasar tan buenos ratos en nuestras aficiones y a Robert por acompañarme en toda la carrera con su humor.

A mi hermana Elena y a mi pareja Mónica, que son las mujeres que más me han apoyado y soportado estos cuatro años de carrera.

A mis profesores, que me han enseñado todo el conocimiento que tengo en el mundo de la informática y me han guiado en este camino.

Y sobretodo agradecer a José Luis, que me presentó el proyecto y me ha guiado cada semana para avanzar, y conseguir este trabajo final. Además de despertar en mí una mente mucho más abierta a las ideas nuevas.

Resumen

Actualmente, los importantes avances en hardware y software ocurridos en las últimas décadas han propiciado el desarrollo de entornos para la computación de alto rendimiento. Estos entornos son desarrollados en los conocidos *centros de datos* o *datacenters*. Los datacenters mediante el uso de recursos computacionales *heterogéneos* ejecutan las tareas enviadas por los usuarios de estos entornos.

Las tareas recibidas deben ser asignadas a los recursos disponibles del datacenter. Realizar una asignación entre los recursos disponibles y los trabajos pendientes de ejecutar es una tarea compleja, para la cuál no se conoce un algoritmo determinista que resuelva este problema. Por tanto, se utilizan *algoritmos basados en políticas* que permitan determinar qué tipo de recursos y trabajos se priorizan en las asignaciones. La selección de estas políticas es realizada por un planificador que se encarga de aplicar estos algoritmos clásicos con la información del entorno.

Los avances previamente descritos han permito a su vez el desarrollo de técnicas de *inteligencia artificial*. Este tipo de técnicas desarrolla métodos que permiten solucionar de manera aproximada, problemas para los cuales no existe un algoritmo eficiente o determinista. Estos métodos han permitido avanzar en el desarrollo de campos tan importantes actualmente como son la minería de datos o el procesamiento del lenguaje natural.

Con todo lo expuesto anteriormente, el objetivo de este trabajo es realizar un estudio de cómo ciertas técnicas de *inteligencia artificial* pueden ser usadas para la selección de políticas de planificación. Para esta tarea se utiliza un simulador de datacenters heterogéneos, llamado HDeepRM que permite la creación de un planificador basado en métodos de *inteligencia artificial*.

Para validar estos métodos inteligentes se realizan una serie de experimentos reproducibles en diferentes entornos y configuraciones. Además, se implementan mejoras en el simulador que permiten aumentar tanto la calidad de las simulaciones como la complejidad de las planificaciones.

Palabras claves: Aprendizaje Reforzado Profundo, Gestor de cargas de trabajo, HDeepRM, Simulador de datacenters heterogéneos, Inteligencia artificial, Aprendizaje automático.

Abstract

Nowadays, thanks to software and hardware advances that have occurred in recent decades, have provided the development of *High Performance Computing* (HPC) environments. These environments are developed in the well-known *data centers*. Data centers through the use of heterogeneous computational resources execute jobs sent by the users of these environments.

Jobs should be scheduling to the available resources of the data center. Mapping between available resources and pending jobs is a complex task, for which no deterministic algorithm is known to solve this problem. Therefore, *policy-based algorithms* are used to determine what kind of resources and jobs are prioritized in scheduling. The selection of these policies is done by a *Workload Manager* that apply these classic algorithms with the information from the environment.

The advances previously described have in turn afforded the development of *artificial intelligence* techniques. This kind of techniques develops methods that allow approximate solutions to problems for which there is no efficient or deterministic algorithm. These methods have made it possible to advance in the development of fields as important today as data mining or natural language processing.

Given the above, the goal of this work is to carry out a study of how certain *artificial intelligence* techniques can be used for policy selection. For this task, a heterogeneous data center simulator, called HDeepRM, is used to create a scheduler, that is based on artificial intelligence methods.

To verify the intelligent methods, a series of reproducible experiments are performed to verify the correct operation of these methods. In addition, new improvements are implemented in the simulator that allow increasing the quality of the simulations and the complexity of the schedules.

Keywords: Deep Reinforcement Learning, Workload Manager, HDeepRM, heterogeneous data center simulator, Artificial Intelligent, Machine Learning.

Capítulo 1

Introducción

En este capítulo se expondrán los conceptos generales, en los cuales se basa la parte principal de este trabajo. Se analizará el problema de la planificación de tareas en datacenters utilizados para la computación de alto rendimiento, y cómo técnicas de inteligencia artificial pueden ayudar a resolverlo. Las nuevas propuestas de modelos de planificación se estudian con simuladores de datacenters. Además se expondrá el objetivo fundamental de este trabajo, junto con la metodología seguida para su realización y la estructura utilizada en el documento.

1.1. Planificación de tareas en datacenters

La computación de alto rendimiento permite mediante la agregación de distintos componentes, tanto hardware como software, obtener rendimientos muchos más elevados en comparación con computadores de propósito general. Con este rendimiento se consigue realizar tareas muy costosas de manera más rápida y eficaz. Este tipo de computación viene ligada con el uso de centros de datos o también llamados datacenters, en donde encontraremos todos los componentes necesarios para obtener rendimientos tan altos [2].

Los datacenters están formados por cientos o miles de recursos computacionales. En general, presentan una arquitectura de carácter heterogéneo [3], es decir, los recursos disponibles tienen características muy distintas. Estos recursos se utilizan en conjunto para realizar las tareas enviadas por los usuarios del datacenter. Al recibir las tareas de los usuarios, se guardan en una cola de trabajos, para luego ser planificados en los recursos disponibles del datacenter, ofreciendo un servicio rápido y eficaz.

Esta planificación de las tareas es muy compleja por tener cientos o miles de trabajos que tienen que ser ejecutados con los recursos disponibles en el datacenter, ofreciendo garantías de eficiencia a los usuarios. Además la complejidad aumenta si tenemos en cuenta la hetero-

geneidad de los datacenters, y que en este tipo de entornos es muy común el uso de técnicas de paralelización y distribución de trabajos y de datos. Esta distribución ocasiona que varios recursos puedan estar realizando la misma tarea al mismo tiempo y que se tengan que comunicar para intercambiarse datos. Este intercambio provoca que a la hora de planificar, hay que tener en cuenta aspectos como la red de interconexión de los recursos o las latencias de envío de datos, complicando muchísimo más el problema de la planificación.

Además de la heterogeneidad, otro factor fundamental que afecta a la complejidad son los objetivos de planificación. Estos objetivos de planificación priorizan distintas características de los datacenter como pueden ser el rendimiento o el consumo de energía. En concreto, el consumo de energía [4] se ha convertido en un serio problema, debido a los costes que supone, y por tanto hay que tenerlo en cuenta a la hora de planificar.

Aunque la planificación de tareas de un datacenter es un problema de decisión NP-completo [5], por lo que no se puede dar una solución óptima, actualmente se utilizan algoritmos heurísticos muy sencillos en la selección de políticas [6]. En concreto se utilizan dos tipos de políticas. El primer tipo selecciona la forma en que los trabajos saldrán de la cola de planificación, un ejemplo de este tipo de política es la conocida *shortest job first* que selecciona en primer lugar los trabajos que requieren menos tiempo. El segundo tipo de políticas prioriza los recursos donde se ejecutarán los trabajos, un ejemplo común es la política *highest computing capability* que selecciona los recursos con mayor capacidad computacional.

Los algoritmos basados en políticas intentan analizar las características de los trabajos existentes en la cola y el estado actual de los recursos. Todo este trabajo de recolección de datos y aplicación de los algoritmos se lleva a cabo por el *Workload Manager* [7]. El *Workload Manager* selecciona una política que priorice los trabajos y los recursos para ejecutar los trabajos de manera óptima. Este análisis también es complejo, y obtener las características principales de los miles de trabajos y recursos con los rendimientos requeridos para estos sistemas es una tarea inabordable. Debido a esta complejidad, los algoritmos usados se basan en aproximaciones heurísticas para estimar las políticas a seleccionar que no garantizan obtener una respuesta óptima.

Por todo esto múltiples grupos de investigación están trabajando en encontrar técnicas para mejorar los algoritmos clásicos, de forma que se consiga un mayor rendimiento de las planificaciones y un uso más eficiente de los recursos de los datacenters. Una de las técnicas más prometedoras y con mayor proyección es la utilización de métodos de *machine learning* para la planificación de las tareas. Estos métodos están siendo estudiados en gran medida gracias a la utilización de simuladores, donde se pueden simular ambientes reales y controlados para obtener resultados que sirvan para la mejora de estos métodos.

En este ámbito es donde entra HDeepRM, el cual es un simulador de datacenters heterogéneos que permite la implementación de planificadores basados en métodos de *machine learning*.

1.2. Machine Learning

El machine learning [8] es una rama de la inteligencia artificial centrada en la creación de sistemas computacionales capaces de aprender automáticamente en función de la experiencia. Este área de la inteligencia artificial tuvo su comienzo con el descubrimiento del *perceptron* [9] a finales de los años 50. Pero no sería hasta el comienzo de los años 80 donde nuevos algoritmos mejorarían las prestaciones de los algoritmos clásicos y el machine learning empezaría a tener una repercusión enorme en el mundo de la inteligencia artificial. Esta repercusión aumentaría junto con el incremento en la capacidad de cómputo producida en los años 90 y 2000, la cual permitiría a modelos como *Support Vector Machine* o los *Algoritmos Genéticos* desarrollarse y aplicarse con éxito en distintos ámbitos, a parte de la informática.

Existen múltiples maneras de aplicar el machine learning, dependiendo del problema que se pretenda realizar y aprender. Para el problema que nos atañe de planificación de tareas en datacenters, se han usado dos técnicas concretas que han dado resultados muy positivos en distintos tipos de experimentos. Estas técnicas son el *Deep Learning* [10] y *Reinforcement Learning* [11]. Ambas técnicas tienen características muy diferentes, como son:

- *Reinforcement Learning*: es uno de los tres métodos más generales en el ámbito del machine learning. A diferencia de los otros dos métodos, el *Supervised Learning* y el *Unsupervised Learning*, éste no se centra en cómo se presentan los datos para el aprendizaje, sino en la propia estructura del aprendizaje. El *Reinforcement Learning* define una estructura de aprendizaje en donde existen dos conceptos, el agente y el entorno que interactúan entre sí para realizar el aprendizaje del agente, reforzado con las observaciones del entorno.
- *Deep Learning*: esta técnica se basa en el uso de conjuntos de *perceptrones* dando lugar a estructuras en capas interconectadas. Estas estructuras darían lugar a las redes neuronales actuales que resolverían problemas en ámbitos como la clasificación de imágenes o en la creación de modelos de lenguajes en el procesado del lenguaje natural.

Estas dos técnicas fueron implementadas en el simulador HDeepRM para la creación de un planificador de tareas inteligente que pueda mejorar la eficiencia de los algoritmos clásicos de planificación.

1.3. Objetivos

El objetivo principal de este trabajo es mejorar las capacidades de HDeepRM, una herramienta de simulación desarrollada en el seno del grupo de investigación de Arquitectura y Tecnología de Computadores de la Universidad de Cantabria mediante un TFM de Adrián Herrera [1]. Debido a que este objetivo abarca una gran cantidad de aspectos, se subdivide en tres metas fundamentales:

- *Mejorar la escalabilidad*: la planificación del agente en HDeepRM está muy limitada al número de trabajos y de recursos en la simulación. Se pretende aumentar la capacidad de planificación del sistema de manera correcta, independientemente del número de recursos y de trabajos que tenga el sistema.

- *Aumentar el número de los objetivos de planificación*: los objetivos de planificación indican qué aspectos del datacenter hay que priorizar en la planificación, como pueden ser la reducción del consumo de energía o disminución del tiempo de ejecución de los trabajos. En HDeepRM hay implementados cinco objetivos, pero solamente uno de ellos es funcional con el planificador inteligente, el resto funcionan únicamente para el planificador clásico. Por tanto, se decide implementar dos nuevos objetivos funcionales para cualquier tipo de planificador, tanto clásico como inteligente.

- *Creación de entornos dinámicos de planificación*: en HDeepRM el planificador siempre trabaja con el mismo workload y el mismo objetivo, lo cual es una aproximación poco ambiciosa en comparación a las situaciones reales que nos podemos encontrar. Por ello se implementan modificaciones en el planificador para que sea capaz de adaptarse a cambios en el workload y en el objetivo.

1.4. Metodología y Plan de trabajo

Para la realización de los objetivos previos se han seguido los siguientes pasos:

- *Estudio de la planificación en datacenters*: en primer lugar se realiza un estudio de cómo son los procesos de planificación en los datacenter. Esto permite conocer en detalle el problema de planificación, y con ello poder encontrar una solución factible.

- *Conocer técnicas de inteligencia artificial*: es necesario conocer las principales técnicas como el *Reinforcement Learning* o el *Deep Learning* para poder implementarlas y conocer las capacidades que pueden llegar a tener.

- *Análisis de HDeepRM*: tener claro el diseño y la implementación de HDeepRM permite saber cómo funciona el simulador y con ello aplicar los cambios necesarios para mejorar sus prestaciones.

- *Diseño e implementación de objetivos*: para cada objetivo planteado en este trabajo se realiza un estudio de las posibles soluciones. Para las soluciones más probables se realiza la implementación correspondiente.

- *Comprobación de las soluciones*: una vez realizadas las implementaciones de las soluciones es necesario realizar experimentos en el simulador. Estos experimentos permiten validar la efectividad de las soluciones implementadas, gracias a los datos que nos proporciona HDeepRM.

- *Depuración y optimización*: una vez tenemos los resultados obtenidos de los experimentos, se realiza una modificación de la implementación con la finalidad de optimizar su rendimiento, precisión, etc.

1.5. Estructura del Documento

El documento presentado consta de cinco capítulos incluyendo el presente Capítulo 1 de *Introducción*.

- *Capítulo 2*: se presentan los conceptos básicos de las técnicas de inteligencia artificial y las características fundamentales de HDeepRM.
- *Capítulo 3*: se describen los cambios que se ha realizado en el simulador, junto con una descripción detallada del diseño y la implementación de cómo estos cambios permiten mejorar la planificación de tareas con las técnicas de inteligencia artificial.
- *Capítulo 4*: los experimentos que permiten comprobar los cambios realizados en el Capítulo 3 son detallados en este capítulo. Estos experimentos son detallados de manera exacta, para que cualquier persona pueda realizarlos y obtenga los resultados descritos.
- *Capítulo 5*: para finalizar el trabajo se presentan las conclusiones finales y los posibles objetivos a futuro que se podrían implementar en HDeepRM para mejorar la calidad de las simulaciones.

Capítulo 2

Background

En este capítulo se describen una serie de conceptos importantes para comprender el resto del documento. También se hace una somera descripción de la herramienta HDeepRM que es la plataforma que se ha utilizado como punto de partida para el desarrollo del proyecto. Finalmente se presentan algunos trabajos previos que tienen relación con el desarrollo de este proyecto.

2.1. Conceptos Básicos

2.1.1. Deep Reinforcement Learning

Actualmente la investigación y desarrollo en distintos campos del ámbito tecnológico tiende a buscar soluciones mediante métodos de inteligencia artificial [12]. En este tipo de métodos se pretende obtener un agente inteligente que sea capaz de percibir un entorno que le rodea y que pueda reaccionar ante los cambios de una manera rápida y de la forma más óptima posible. En este ámbito entra en juego el *Deep Reinforcement Learning*, una técnica de inteligencia artificial, la cual gracias a la unión de las técnicas de *Deep Learning* y *Reinforcement Learning* es capaz de realizar, mediante un agente y un entorno activo un aprendizaje reforzado. Con este aprendizaje se consigue un sistema inteligente que reaccione a cambios en el entorno tomando acciones que optimizan la resolución del problema.

Deep Reinforcement Learning sigue la estructura básica de aprendizaje del *Reinforcement Learning*, que se muestra en la Figura 2.1. El agente recibe como entrada una observación, que modela el estado del entorno. Entonces ejecuta acciones que cambian el estado del entorno, las cuales provocan una notificación al agente para saber si la acción tomada en la etapa previa ha sido beneficiosa para el objetivo que se quiere alcanzar. Esta notificación se conoce como *reward*. Este proceso se repite para cada decisión que toma el agente provocando que

haya una retroalimentación continua entre el entorno y el agente.

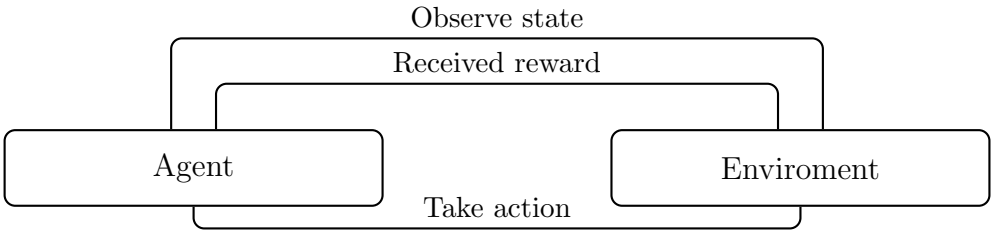


Figura 2.1: Bucle del proceso de Deep Reinforcement Learning

El entorno en este tipo de métodos puede llegar a ser muy distinto dependiendo de la tarea que se pretenda realizar, pero en cuanto al agente en multitud de ocasiones se utiliza una red neuronal, que lleva a cabo el proceso de aprendizaje.

2.1.2. Red Neuronal

Las redes neuronales [13] han sido utilizadas desde los años cuarenta, pero no fueron desarrolladas completamente debido a la poca capacidad computacional que se tenía. El incremento actual de la capacidad de cómputo, gracias a la aparición de los multiprocesadores y aceleradores específicos, han provocado una actualización de las redes, pudiéndose aplicar a problemas reales. Las redes neuronales son actualmente una de las técnicas de *Deep Learning* y aprendizaje automático más utilizadas y que tienen más éxito en multitud de ámbitos. El esquema de una red neuronal convencional podría ser como el mostrado en la Figura 2.2:

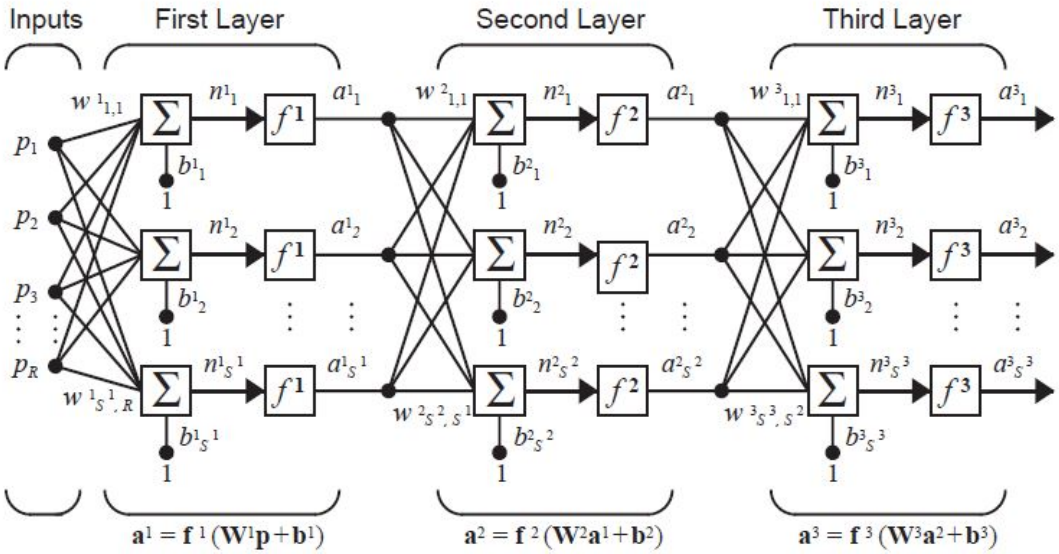


Figura 2.2: Red Neuronal[1]

[1]Imagen de Howard B Demuth, Mark H Beale, Orlando De Jess, and Martin T Hagan. *Neural network design*. Martin Hagan, 2014

Las redes neuronales están formadas por una serie de funciones matemáticas, llamadas *neuronas*, las cuales reciben unos parámetros de entrada y devuelven un conjunto de valores de salida. La red neuronal estructura estas neuronas en capas, configuradas de forma que la salida de una capa sea la entrada de la siguiente.

Una vez definida la estructura de la red es necesario entrenarla para que aprenda, lo cual se lleva a cabo mediante dos procesos indispensables:

- *Forwarding* [14]: la red neuronal recibe una serie de parámetros de entrada, los cuales son usados por la primera capa de la red para calcular los primeros valores de salida. Estos primeros valores serán recogidos por la siguiente capa de la red y se realizará el mismo procedimiento. Cuando finalice el proceso descrito, la red neuronal dará un resultado que será la salida de su última capa.
- *Backpropagation* [15]: una vez realizado el paso previo de *forwarding*, queremos que la red neuronal sea capaz de aprender una tarea y para ello se utiliza este proceso. En un primer lugar se define una *función de pérdida*, la cual calculará el error que tenemos en el proceso de *forwarding*. Una vez calculado este error, lo que se intenta es minimizar su valor, este proceso se realiza mediante el *algoritmo de backpropagation* que se basa en el *descenso del gradiente*, que pretende minimizar el valor de una función dada, cambiando los parámetros que la definen. Por tanto, el algoritmo consistiría en calcular la *función de pérdida* de la red, y aplicar el *descenso del gradiente* a cada una de las capas para ajustar los parámetros de cada neurona, y así en cada iteración reducir la pérdida de la red.

2.2. HDeepRM

HDeepRM fue creado para la investigación de técnicas de inteligencia artificial para su uso en el problema de planificación de tareas en un datacenter. Este problema como se menciona en el Sección 1.1, se fundamenta en el hecho de que asignar tareas a los recursos de un datacenter es un problema NP-completo para el cual, solo existen algoritmos heurísticos muy simples. HDeepRM, basándose en un simulador de datacenters llamado Batsim [16], añade módulos software que permiten a los investigadores implementar técnicas de inteligencia artificial para la resolución del problema. Además añade funcionalidades a la simulación, como son la capacidad de implementar un datacenter heterogéneo con miles de cores trabajando, y su capacidad para simular la contención de recursos compartidos cuando se saturan debido a que varias tareas compiten por estos recursos.

El programa es capaz de realizar simulaciones completas de datacenters con una alta heterogeneidad, tanto en cuanto a las arquitecturas como a las tareas a ejecutar. Este simulador ofrece una estructura perfectamente clara y jerarquizada de cómo se debe realizar la definición de los distintos componentes que pueden formar parte del datacenter, dando facilidad a los usuarios de la aplicación para añadir distintos tipos de recursos con estructuras fáciles

de escribir. La estructura empleada por HDeepRM para simular los dataceters se muestra en la Figura 2.3:

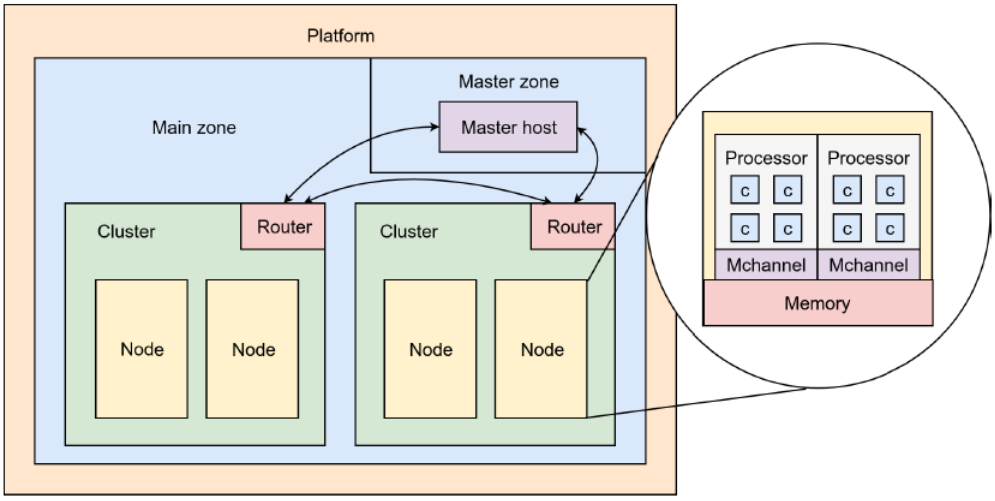


Figura 2.3: Representación del datacenter en HDeepRM [1]

En la imagen se pueden distinguir las distintas zonas:

- *Platform*: elemento principal del sistema.
- *Main zone*: zona en el cual se encuentran todos los recursos del datacenter.
- *Master zone*: zona que contiene al master host.
- *Master host*: host encargado de la ejecución del algoritmo de gestión de tareas (workload manager).
- *Cluster*: conjunto de nodos que comparten un espacio en el datacenter.
- *Router*: elemento de decisión de comunicación entre distintos clusters y el master host.
- *Node*: unidad de cómputo del cluster, donde se encuentran los procesadores.
- *Processor*: unidad computacional del nodo. Puede estar compuestos de varios cores, los cuales acceden a una memoria común mediante un canal de memoria compartido.
- *Core*: unidad mínima de cómputo del sistema. Entre las características que definen a un Core cabe destacar el P-state, que representa la capacidad de computo y el consumo energético porcentual del core.
- *Memory y channels*: Cada nodo tiene una cantidad de memoria y cada procesador tiene un ancho de banda de acceso a memoria. Los cores comparten la memoria del nodo y por tanto se reparten el ancho de banda del procesador.

Además de la estructura hardware cabe destacar la capacidad de esta aplicación para poder recibir tareas para la simulación de las planificaciones. Las tareas pueden ser definidas mediante dos tipos de formatos, un archivo JSON o un archivo Standar Workload Format (SWF) [17], permitiendo identificar los siguientes campos:

- *Subtime*: tiempo de llegada de un trabajo.
- *Res*: número de cores que necesita el trabajo.
- *Cpu*: operaciones de coma flotante por core.
- *Req_time*: tiempo de ejecución para el usuario.
- *Req_ops*: operaciones de coma flotante necesarias por core.
- *Cpu*: cantidad real de operaciones de coma flotante ajustado según los req_ops y el subtime.
- *Mem*: memoria necesaria por core. (MB)
- *Mem_bw*: ancho de banda de memoria por core. (GB/s)

El simulador de HDeepRM está profundamente ligado a la incorporación de la inteligencia artificial para la planificación de las tareas. Se pueden definir distintos tipos de agentes que realizan la tarea de aprendizaje, utilizando simplemente una interfaz básica de programación para el correcto funcionamiento del programa.

HDeepRM está escrito en uno de los lenguajes más utilizado actualmente para *Deep Learning* y el *Machine Learning*, el cual es Python [18]. Python ofrece una gran cantidad de librerías de software libre especializadas en inteligencia artificial, y por tanto en HDeepRM se decidió utilizar las librerías de Gym [19] y Pytorch [20], las cuales permiten definir un entorno y un agente, adecuado para las condiciones que el programa requiere.

Mediante las librerías previamente mencionadas en HDeepRM, se utiliza la técnica de *Deep Reinforcement Learning* para la búsqueda de una red neuronal que permita realizar la planificación de tareas. Para esto, las primeras pruebas se han realizado con dos tipos de redes neuronales, *reinforce* y *actor-critic* [11]. Ambos tipos son redes neuronales *full-connected* [13] con una estructura y neuronas prácticamente iguales. Pero la diferencia fundamental entre ambos modelos es que con *actor-critic* tenemos dos capas finales distintas, en las cuales una es igual a la utilizada en el modelo de *reinforce*, mientras que la otra calcula un valor estimado del estado del *reward* ofrecido por el entorno. Este último cálculo, permite a la red una convergencia más rápida, sin la utilización de una red complementaria.

Aunque HDeepRM es capaz de simular una gran cantidad de características de los datacentes, aún le quedan ciertos aspectos a mejorar como son:

- *Planificador inteligente*: la capacidad de usar los planificadores inteligentes, se ve limitado a datacenters de entornos con 8 y 4 cores, este número es muy limitado y no refleja lo que se podría esperar de un datacenter real.
- *Objetivos de planificación*: estos priorizan aspectos de los datacenters como son el rendimiento o la eficiencia energética. En el simulador estos objetivos no están bien definidos y conllevan errores en la selección de las políticas.
- *Comportamiento dinámico*: el planificador inteligente solo es capaz de simular correctamente un entorno sin cambios de trabajos y sin cambios de objetivos de planificación. Estos aspectos son muy importantes para poder simular entornos realistas y que se deben de tener en cuenta para las mejoras del simulador.

2.3. Trabajos Relacionados

HDeepRM no es el único experimento que intenta aplicar técnicas de inteligencia artificial en la planificación de trabajos en datacenters. Hay varios grupos de investigación por todo el mundo que están intentando aplicar estas técnicas tanto con simuladores como con entornos reales.

Uno de los proyectos más interesantes y que utiliza *Deep Learning* es *Decima* [21]. Este proyecto se basa en el desarrollo de un planificador de propósito general utilizando la técnica de *Deep Learning*. Se centran especialmente en los trabajos que presentan dependencias, debido a que muchos sistemas interpretan este tipo de trabajos como un *grafo acíclico dirigido* (DAG's) [22] y su planificación mediante algoritmos clásicos o heurísticos suele ser muy compleja [23]. En este caso además de implementar una red neuronal de planificación general, realizan un algoritmo que les permite reducir la dimensión de los DAG's pudiendo planificar este tipo de grafos con un resultado más eficiente que los actuales algoritmos heurísticos.

Un ejemplo en el cual el *Reinforcement Learning* es utilizado con resultados muy positivos es *Dynamic Colocation Policies with Reinforcement Learning* [24]. Utilizan la técnica de *Q-Learning* [25] para planificar trabajos iterativos, los cuales se caracterizan por bloquear el uso de componentes compartidos y de no cumplir con los rendimientos esperados. Debido a estas características la utilización del datacenter desciende provocando pérdidas en el rendimiento del sistema. Cabe destacar que el uso del planificador está ligado al uso de técnicas descriptivas de procesos, que permiten identificar los recursos que los procesos necesitan, esta descripción la realiza también un agente inteligente que analiza dinámicamente el workload y cambia el modelo de planificación.

Cabe destacar dos trabajos en los cuales HDeepRM se basó para su creación y que plantean ideas muy interesantes. Estos trabajos son, *Resource Management with Deep Reinforcement Learning* [26] y *RLScheduler* [27].

En el artículo [26], se plantea el problema de planificación de manera mucho más completa y

amplia en comparación a los trabajos previos, permitiendo observar la complejidad que tiene el problema. Es por esto que se plantea la alternativa de utilizar inteligencia artificial en el uso de la planificación de los trabajos en datacenters. En este caso, se plantean entornos mucho más completos, en los cuales no se planifican solo un tipo de trabajo, sino que el agente debe de ser capaz de adaptarse a los cambios en el workload. Cabe destacar el uso de objetivos de planificación, los cuales permiten definir, como en el caso de HDeepRM, características a priorizar en el datacenter.

En el caso de *RLScheduler* [27], se centra en la computación de alto rendimiento e implementan un tipo de planificador basado en *Reinforcement Learning*. Este agente pretende demostrar cómo es posible planificar las tareas que recibe un datacenter de manera correcta y permitiendo adaptarse a los distintos objetivos, plataformas y workloads con los que se puede encontrar. Es de destacar que su uso solo fue probado en entornos homogéneos, pero el éxito de sus pruebas permite tener una breve aproximación a lo que es capaz de realizar el *Reinforcement Learning* en este problema.

Capítulo 3

Aportaciones al Diseño e Implementación de HDeepRM

En este capítulo se explican las contribuciones de este trabajo en HDeepRM. Para esto se describen los procesos de diseño e implementación necesarios para crear un agente inteligente, capaz de realizar una planificación de tareas con cualquier tipo de workload o de plataforma, independientemente del tamaño de ambas. Además se añade al simulador dos objetivos de planificación nuevos, y junto con ellos la funcionalidad de que el agente, detecte en plena simulación un cambio de objetivos o en las características del workload, y responda ante estas situaciones.

3.1. Escalabilidad del sistema de planificación

En la versión inicial de HDeepRM se había conseguido simular de manera efectiva datacenters reales utilizando un agente no inteligente, pero en el desarrollo de las técnicas de *Deep Reinforcement Learning* los resultados no fueron tan positivos. Esto fue debido a que el agente inteligente no era escalable, simplemente era capaz de realizar un ejemplo con un número muy reducido de cores y muy pocos trabajos en la cola de planificación. Por tanto, se decide implementar una primera aportación importante, que permita al agente inteligente mediante *Deep Reinforcement Learning* planificar de manera correcta, independientemente del número de recursos y de trabajos de la simulación.

Para permitir la escalabilidad del sistema se comenzó por un análisis completo del código de la aplicación. Esto permite conocer el entorno del programa y los defectos que podía tener en la implementación del agente inteligente. Además, de poder adaptar el estilo de programación al utilizado previamente en la aplicación y tener un código coherente de principio a fin.

En este análisis de código se decide que tanto el diseño como su implementación podía ser

modificado. Aunque, hay que destacar que en cuanto al diseño no fue necesario realizar ninguna modificación y los cambios fueron realizados en la implementación concreta del programa.

Una vez conocida la estructura y el diseño del código, se continúa con la ejecución del mismo, para comprobar su comportamiento. Primero se realizan las ejecuciones con el agente de tipo clásico, que ejecuta una única política para toda fase de planificación. Esta ejecución provocó problemas a la hora de seleccionar la política descrita en el fichero de configuración. Esto era debido a un error en el código del agente clásico que siempre decidía una política de planificación de manera aleatoria.

La solución del error fue leer la política escrita en el fichero de configuración del programa y guardando su valor en el agente clásico para que se escoja siempre esta política. Los valores de las políticas se calculan teniendo en cuenta la posición de dicha política en un diccionario. Por tanto, se comprueba que la política decidida se encuentra dentro del diccionario y se calcula su correspondiente índice y se guarda en el agente clásico.

La solución previamente descrita arregla perfectamente el agente clásico permitiendo realizar una simulación completa con una única política. Con los resultados de la simulación, se pueden comparar los comportamientos de los trabajos y recursos dependiendo de la política seleccionada. Y con esto saber cuál política es la óptima para una simulación en concreto, ayudando a la depuración del agente inteligente.

El siguiente paso es ejecutar el simulador con el agente inteligente. En la Sección 2.2 se habla de dos diseños de agentes distintos, *reinforcement* y *actor-critic*, ambos agentes se prueban para comprobar que no hay errores en el código de ninguno de los dos. En el primer caso con el agente de *reinforcement* no se aprecia ningún fallo importante que comprometa ni a la eficiencia, ni a la efectividad del planificador.

En cuanto al agente *actor-critic* hay varios errores destacables que ponen en compromiso su funcionamiento. Los errores más destacables encontrados fueron:

- *Método process*: el agente inteligente después de realizar el algoritmo de *forwarding* devuelve una lista con las probabilidades de selección de cada política y el valor del *reward* esperado. En la primera ejecución de la aplicación el programa retorna un error debido a que el contenido de la lista de las probabilidades es vacío. Este error era debido al uso del método *process*, que tenía que realizar el proceso de *forwarding* pero no lo realizaba debido a su falta de implementación. Por tanto no retornaba las probabilidades de las políticas y provocaba el error previamente descrito.

  Este error se soluciona creando un método *forward* que se encarga de realizar el proceso de *forwarding*. Este método se divide en dos procesos de *forwarding*, uno para calcular las probabilidades de las políticas y otro para calcular el valor esperado. La implementación de cada uno de los procesos se realiza utilizando las funciones proporcionadas por Pytorch. En la Figura 3.1 se muestra la implementación del método y los dos procesos de *forwarding*.

Con este cambio se consigue que la aplicación funcione sin que se impida la ejecución final del programa. Por tanto, ya se puede comprobar el funcionamiento del modelo *actor-critic*, el cual, como se puede ver en el Capítulo 4 será el modelo más utilizado gracias a su gran capacidad de selección de políticas.

```
def forward(self, observation: np.ndarray) -> tuple:
    observation = torch.from_numpy(observation).float().unsqueeze(0)
    probs = self.forward_policy(observation)
    value = self.forward_value(observation)
    return probs, value

def forward_policy(self, observation: np.ndarray) -> torch.Tensor:
    out_0 = F.leaky_relu(self.input(observation))
    out_1 = F.leaky_relu(self.actor_hidden_0(out_0))
    out_2 = F.leaky_relu(self.actor_hidden_1(out_1))
    out_3 = F.leaky_relu(self.actor_hidden_2(out_2))
    return F.softmax(self.actor_output(out_3), dim=1)


def forward_value(self, observation: np.ndarray) -> torch.Tensor:
    out_0 = F.leaky_relu(self.input(observation))
    out_1 = F.leaky_relu(self.critic_hidden_0(out_0))
    out_2 = F.leaky_relu(self.critic_hidden_1(out_1))
    out_3 = F.leaky_relu(self.critic_hidden_2(out_2))
    return self.critic_output(out_3)
```

Figura 3.1: Método forward con sus respectivos procesos de *forwarding*

- *Normalización de los rewards*: cuando se calculan las pérdidas de la red, los *rewards* obtenidos en toda la simulación recibían un tratamiento de normalización. Este tratamiento provocaba errores en la planificación de los agentes. Como se puede observar en la Figura 3.2, el agente no selecciona ninguna política en concreto y tampoco las prioriza con una diferencia importante en la probabilidad.

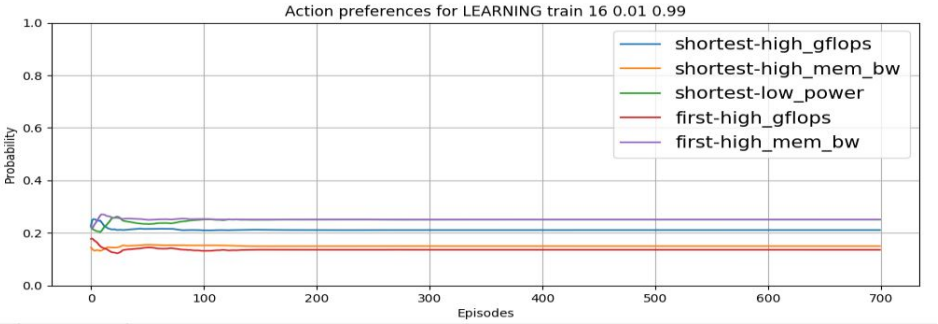


Figura 3.2: Preferencia de acciones del ejemplo error de la normalización del *reward* - 700 episodios

La causa del problema era la normalización, debido a que al normalizar los valores del *reward* podían ser muy pequeños, siendo en torno a las decenas o incluso en ciertos experimentos de unidades. Valores tan reducidos provocaban que el planificador no fuera capaz de detectar las pérdidas de la red debido a tener un *learning rate* demasiado grande para el valor de los *rewards*. Una solución sencilla hubiera sido bajar el *learning rate*, pero esto ocasiona que el agente tarda mucho más en converger. Por tanto se decide modificar la normalización de los *rewards*.

De esta forma, la solución tomada ha sido aumentar el valor mínimo que pueden tomar los *rewards*. El valor mínimo que se impone a los *rewards* se establece en torno a los millares. Según la experiencia obtenida con la aplicación este rango permite entrenar eficazmente al agente sin ralentizar en exceso su aprendizaje. Con este cambio se consigue que el agente sea capaz de aprender políticas sin quedarse parado en ningún punto del aprendizaje. Por tanto, se pueden realizar simulaciones con cualquier plataforma y workload, y el agente tenderá a escoger las políticas que benefician el objetivo marcado.

Cabe destacar que existen otros errores menores que se debían a incorrecciones en la sintaxis de Python o errores en el uso de la librería de Pytorch. Estos errores son bastante poco reseñables y no afectaban al desarrollo de las planificaciones ni del aprendizaje de los agentes.

Arreglados todos los tipos de errores se realizan una serie de pruebas funcionales para verificar la escalabilidad del framework, las cuales se pueden observar en la Sección 4.2. Las conclusiones que se pueden extraer de estos experimentos son:

- El sistema es ahora capaz de planificar mediante *Deep Reinforcement Learning*, plataformas y workloads con tamaños de hasta miles de cores y miles de trabajos de manera efectiva y eficaz.
- El agente inteligente puede planificar de manera correcta en entornos heterogéneos independientemente del tamaño de la plataforma o el workload.
- Se pueden comparar resultados del uso de políticas fijas en las simulaciones, y con esto comprobar los resultados obtenidos con el agente inteligente.

3.2. Diseño de Objetivos de Planificación

Una de las funcionalidades que definen a HDeepRM es poder escoger objetivos de planificación. Estos objetivos permiten poder centrarnos en un aspecto del datacenter que queremos optimizar, como pueden ser tener un menor consumo de energía o que los tiempos de finalización de los trabajos disminuyan lo máximo posible. Debido a esta funcionalidad el planificador debe ser adaptable a este cambio de objetivos y modificar su comportamiento en función del objetivo escogido en cada momento.

En el Trabajo Fin de Máster de HDeepRM [1], se definen cinco de estos objetivos, los cuales pueden ser utilizados en la plataforma simplemente indicándolo en los ficheros de configuración del programa. En la aplicación un objetivo es definido como el *reward* que es utilizado por el agente para reforzar su aprendizaje. Estos *rewards* actúan como una etiqueta que indica el valor, al cual el agente debe de acercarse lo más posible.

Entrando en temas de implementación el *reward*, es definido como un método, como se puede ver en la Figura 3.3 para el objetivo *makespan*. El método debe de retornar el valor del *reward*

que utilizará el agente para el proceso de aprendizaje reforzado. Además para poder utilizar el método se debe introducir en un diccionario, el nombre del objetivo junto con el puntero a la función. Con esto se consigue, que leyendo el fichero de configuración y seleccionando el nombre del objetivo, se pueda seleccionar el método de *reward* de manera rápida y muy sencilla.

```
    def makespan_reward(self) -> float:
        """Reward when the objective is to minimize makespan.

Makespan is the total time from the arrival of the first job to the completion of the last one.
Reward is the total number of current GFLOPs in the data centre. Higher throughputs will lead to
lower makespans.

Returns:
    Current total GFLOPs provided by the data centre service.
        """

        return (sum([core.state['current_gflops'] for core
                     in self.workload_manager.resource_manager.core_pool]))
```

Figura 3.3: Método del reward makespan

En la aplicación, cuatro de los cinco objetivos definidos en la implementación original no permitían su uso en el agente inteligente, o no tenían definida una buena métrica para mejorar el objetivo buscado. El único objetivo realmente funcional era el *makespan*. Esta métrica mide el tiempo desde que se planifica el primer trabajo, hasta que se acaban de ejecutar todos los trabajos de la simulación, fundamentalmente es el tiempo que tardan en ejecutar todos los trabajos del workload.

En la implementación del *makespan* como *reward*, este es definido como $\sum_r^R c_r$ siendo R el conjunto de todos los recursos del sistema y c_r la capacidad computacional total usada por el recurso r en toda la simulación. Esta definición representa que a mayor capacidad computacional utilizada menor será el tiempo en acabar los trabajos en el workload, por tanto el agente tenderá a maximizar la capacidad computacional total.

El *makespan* es una métrica que mide el rendimiento que se obtiene en la plataforma. Por tanto, al tener solo este objetivo funcional, se deciden implementar dos nuevos objetivos no basados en el rendimiento. El primer objetivo a implementar se basaría en la reducción del consumo energético en la plataforma, y el segundo se centra en aumentar la eficiencia energética en la simulación.

Ambos objetivos permiten comparar dos métricas muy importantes actualmente en el mundo de los datacenters. Cualquier disminución en el consumo de energía de un datacenter con cientos o miles de cores puede resultar un ahorro en costes fundamental. Y es por esto que la capacidad de usar métricas de energía y de eficiencia energética hacen al simulador aumentar su calidad.

3.2.1. Consumo energético

El objetivo basado en el consumo energético será llamado *energy_consumption*, y pretende reducir el gasto energético que se produce en la plataforma. Este objetivo se define como

$\sum_r^R e_r$ siendo R el conjunto de todos los recursos del sistema y e_r el consumo total energético del core r en toda la simulación. El valor de este sumatorio será expresado en *Julios* y debido a la definición misma del objetivo, el agente intentará minimizar este valor para reducir el consumo energético.

En la implementación de este objetivo es necesario realizar varios cambios en la aplicación, para poder tener todos los datos fundamentales para el cálculo del *reward*. Varios de estos cambios son:

- *Estructura de almacenamiento de la potencia eléctrica consumida y el tiempo*: como bien es sabido, el consumo energético es una unidad de energía, y por tanto la energía consumida en un core se calcula sabiendo su potencia y el tiempo que ha estado trabajando.

  La potencia máxima alcanzable por el core es definida por la plataforma, pero varía en función del *P-state*. Este *P-state* determina la potencia del core en un momento dado de la simulación y varía en función de los trabajos que se ejecutan en el core o incluso en el procesador que se encuentra el core.

  Debido a la variación de potencia que sufren los cores por los cambios en los P-state, es fundamental llevar un registro de cuándo se producen estos cambios para poder calcular el gasto energético total y con ello el *reward*.

  Por tanto, se decide aportar una nueva estructura en el diseño de HDeepRM que permita llevar un registro de los cambios de potencia de un core. Esta estructura se llama *ChangesCores* y en ella se almacena el índice del core que representa, junto con una lista de tuplas que contienen los cambios de potencia del core, y el tiempo en el cual se produjo el cambio.

- *Recopilación de los cambios de estado*: con esta nueva estructura se pueden guardar los datos necesarios para el cálculo del *reward*, pero aún así faltaría saber de qué forma se pueden capturar estos datos dentro del programa para poder guardarlos.

  Al ser HDeepRM el sistema de planificación de Batsim, éste tiene que mandar la orden de cambiar los estados de los procesadores, que dará lugar al cambio de potencia. Para esto, HDeepRM utiliza un método llamado *change_resource_state*, en el cual se reciben todos los cambios de *P-state* de los cores.

  Por tanto, se decide implementar un diccionario que contenga como claves, los identificadores del core y como valores un objeto del tipo *ChangesCores* que irá guardando sucesivamente los cambios de potencia. Estos cambios serán recogidos del método *change_resource_state* añadiendo al diccionario cada cambio para utilizarlo en el cálculo del *reward*.

- *Tiempo de recepción del reward*: con el modelo original de planificación, cuando el agente toma una decisión es recompensado con el *reward* en el mismo instante de tiempo de la simulación.

  Este modelo para el *makespan* era válido debido a que el cálculo de la capacidad computacional total no depende del tiempo en el que se recibe el *reward*. Pero en el

caso del consumo de energía, este depende del tiempo, y es fundamental saber cuánto tiempo un core ha estado ejecutando a una potencia dada.

En la Figura 3.4 se puede observar un ejemplo de la planificación de un procesador con dos cores, al cual le llegan dos trabajos en instantes de tiempo distintos y sus ejecuciones se solapan. También se muestra el momento del cálculo de los *rewards* que recibirá el agente para el objetivo *energy_consumption*.

En el instante de tiempo inicial el trabajo T0 es planificado en el primer core del procesador y empieza a ejecutarse y consume un cincuenta por ciento de la potencia máxima del procesador. El cambio de potencia del core será guardado en su estructura *ChangesCores* particular. Cuando llega el trabajo T1 se produce una nueva fase de planificación, pero previamente se calcula el *reward* correspondiente con la planificación en el instante inicial, utilizando los datos guardados. Este cálculo se realizará multiplicando la potencia de los core por el tiempo entre la planificación inicial y la planificación del trabajo T1. Cuando el trabajo T1 es planificado en el segundo core, este aumenta su potencia para poder abastecer la demanda computacional, alcanzando el máximo de potencia para el procesador. Este cambio será guardado en la estructura *ChangesCores* del segundo core.

Una vez planificado el trabajo T1, ya no quedan nuevos trabajos en el workload, y por lo tanto no se recibirá un nuevo *reward* hasta que finalicen todos los trabajos en ejecución. Como se puede observar el trabajo T0 finaliza antes que el trabajo T1 y por tanto la potencia del procesador disminuye al cincuenta por cierto. Esta disminución de potencia será guardada como previamente se realizó en el paso inicial.

Cuando finaliza el trabajo T1 se calcula el *reward* correspondiente a la planificación del trabajo T1. Se sumará la energía correspondiente a los dos intervalos distintos de potencia, entre la planificación de T1 y la finalización de T0 y entre la finalización del T0 y la finalización del T1.

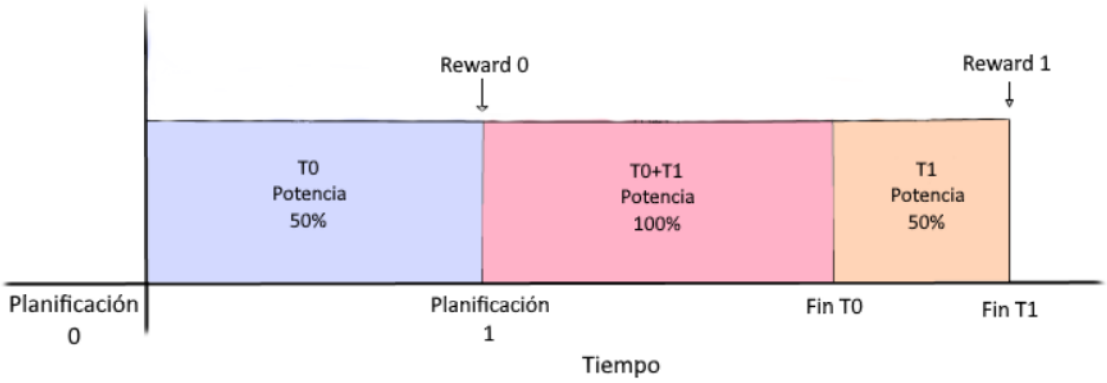


Figura 3.4: Ejemplo de explicación del tiempo del reward

Este ejemplo sencillo ilustra cómo se debe calcular la energía consumida entre los intervalos de planificación y cómo se guardan los cambios de potencia en la simulación. El ejemplo es escalable a cientos de procesadores y se realiza de la misma manera independiente del número de cores por procesador.

Por tanto, dependiendo del *reward* utilizado se debe recibir la recompensa en un tiempo distinto. Esto se implementa de manera que en cualquier evento que provoque una planificación del agente y el *reward* sea el *energy_consumption*, éste sea recompensado con el consumo energético. Mientras que si el *reward* es el *makespan*, simplemente al finalizar la planificación y no quedar más eventos a realizar se calculará el *makespan* y se recompensará al agente. En la Figura 3.5, se puede observar el código añadido para calcular el consumo energético cuando se produce un evento que provoca una planificación.

```
if self.flow_flags['action_taken'] and self.reward != "makespan":
    # The Agent is rewarded
    self.agent.rewarded(self.env)
    self.time_last_step = self.bs.time()
    logging.info(self.time_last_step)
    self.flow_flags['action_taken'] = False
```

Figura 3.5: Código para cambiar el tiempo del *reward* *energy_consumption*

Una vez tenemos el algoritmo necesario para el cálculo del *reward*, se debe implementar en HDeepRM el método correspondiente. El algoritmo se basa en los principios explicados con el ejemplo de la Figura 3.4. Entrando en detalle de la implementación cuando se decide calcular el *reward* se debe recorrer todos los cores del sistema, para poder observar si se han producido cambios en su potencia. Si no se ha producido ningún cambio desde el inicio de la simulación, se calcula su energía en función de la potencia de reposo del core. Si el core ha sufrido cambios en su potencia, en primer lugar se observa en la estructura de *ChangesCores* del core si se guardan cambios que son muy antiguos en el tiempo y por tanto innecesarios de guardar. Esto último se realiza para ahorrar memoria de la aplicación. Una vez eliminados los cambios innecesarios, se calcula la energía consumida con los intervalos de tiempo y potencia guardados, y se suma al total de la energía consumida.

Para finalizar cuando tenemos la energía consumida total es fundamental retornar esta energía como negativa. Esto es debido a que el agente inteligente intenta buscar la política que maximiza el *reward* ofrecido por el entorno. Por tanto, como en este caso queremos minimizar la energía consumida, debemos de poner en negativo este valor para que el consumo de energía más bajo se corresponda con el *reward* más alto.

Con todo esto tendríamos perfectamente en funcionamiento el objetivo del *energy_consumption* cuya depuración de errores y experimentación se pueden observar en la Sección 4.3. Este objetivo prioriza el consumo de energía, pero no aporta información acerca de cómo afecta la reducción del consumo en el rendimiento, y para esto se implementa el siguiente objetivo.

3.2.2. Eficiencia Energética

El siguiente objetivo a implementar es una medida de eficiencia energética conocida como *energy-delay product*. Esta medida se define como $\sum_r^R e_r t_s$ siendo R todos los recursos del sistema y e_r el consumo total energético del core r en toda la simulación y t_s el tiempo total de la simulación. Esta definición de eficiencia permite ligar el consumo energético junto con el rendimiento que es representado con el tiempo. Cuanto menor sea el valor del sumatorio mayor eficiencia energética se obtendrá en la simulación.

Un ejemplo simple del funcionamiento de la métrica de *energy-delay product*, es en la comparación de dos cores con el mismo trabajo. Si tenemos dos cores que realizan el mismo trabajo, y el primero de ellos tiene una potencia de 80 watios y acaba el trabajo en 50 segundos, y el otro core necesita una potencia 120 watios y acaba el trabajo en 40 segundos. Calculando de forma sencilla la energía como el producto de la potencia por el tiempo tenemos que para el primer core la energía consumida es 4000 julios y el segundo 4800 julio. Se observa que el core que tarda menos tiempo gasta más energía para realizar el mismo trabajo y si se utiliza una métrica de consumo energético sería penalizado el core que tarda menos tiempo.

Por tanto, para realizar la métrica de *energy-delay product* multiplicamos la energía calculada previamente por el tiempo en ejecutar la aplicación. Con este cálculo obtenemos que el primer core tiene una eficiencia de 200.000 $julios * segundos$ y en el caso del segundo obtenemos 192.000 $julios * segundos$. Como se observa ahora el valor de eficiencia energética beneficia al segundo core debido a que gracias a esta operación sencilla de multiplicación aportamos más peso al rendimiento en el cálculo. Este segundo core mejora el rendimiento, y por tanto aunque gasta más energía en total es más eficiente en el uso de la energía debido a que acaba su trabajo antes.

Con este ejemplo se puede apreciar la utilidad de esta métrica y el gran aporte que supone al simulador tener una medida que prioriza la forma más eficiente de gastar la energía de la plataforma. Esto aumenta en gran medida la calidad del simulador debido a que con este objetivo se reduce el consumo energético sin tener que perjudicar de manera brusca el rendimiento.

Entrando en la implementación del objetivo, se realiza de manera similar al objetivo de *energy_consumption* debido a que el cálculo de la energía consumida es igual al previamente realizado. En este caso la única diferencia que se aporta es al cálculo del *reward*, en el cual se añade una multiplicación por el tiempo que teníamos guardado en la estructura *ChangesCores*. Por tanto, todos los cambios que se realizaron para el objetivo del *energy_consumption* son aplicables al nuevo objetivo.

Cabe destacar que este *reward* también se calcula antes de la etapa de planificación para poder apreciar los cambios que ha provocado la selección de políticas. También se retorna el valor de este *reward* de manera negativa, debido a que el agente siempre intenta buscar la política que maximice el valor del *reward*, pero en este caso se quiere minimizar el valor. Entonces es necesario poner el resultado de manera negativa para que la eficiencia energética

mínima que se puede obtener, al aplicarle el cambio se convierta en la máxima y el agente seleccione de manera correcta la política que reduce el valor de la eficiencia energética.

Como en el caso del objetivo *energy_consumption* se realizan unos experimentos de comprobación en la sección 4.3 para comprobar los errores que se pueden encontrar en el código y asegurar que el nuevo *reward* funciona perfectamente para la planificación con un agente inteligente.

3.3. Escenario Dinámico

Los datacenters están funcionando constantemente recibiendo tareas que tienen que planificar. Sin embargo, es posible que necesiten cambiar los objetivos de planificación, o bien que se produzcan cambios en el tipo de tareas que llegan. A continuación, se describirán los cambios realizados en el simulador para que el agente sea capaz de responder a estas situaciones de forma dinámica y autónoma.

3.3.1. Cambio de Objetivos

Una de las aportaciones principales al simulador es añadir la capacidad de que el agente detecte el cambio de objetivo y responda ante este cambio. Esta reacción del agente se realizará de manera automática una vez se detecte el cambio de objetivo.

Estos cambios de objetivos son fundamentales, debido a que los administradores de los sistemas dependiendo de la situación del datacenter, deciden priorizar distintos objetivos de planificación, a medida de las necesidades. Por tanto, es fundamental que el agente sea capaz de detectar estos cambios y planifique los nuevos trabajos priorizando el nuevo objetivo.

Estos cambios de objetivos en HDeepRM se realizan de manera sencilla en el fichero de configuración del programa. Un ejemplo de este tipo de fichero es el que se muestra en la Figura 3.6. En él se pueden observar que el fichero es definido con formato JSON en el que tenemos distintos campos para definir. Entre todos los campos observables en la imagen, cabe destacar el campo *objective*, donde se define el objetivo que se quiere priorizar en la simulación. Por tanto con la aportación a realizar, el agente debe detectar el cambio de este campo en el fichero de configuración y actuar de la manera correspondiente.

Una vez tenemos claro el problema, es fundamental decidir cómo el agente va a actuar ante este cambio de objetivo. Esta decisión es fundamental debido a que un mal diseño del comportamiento del agente puede provocar que su eficiencia y efectividad se vea comprometida. Además no conseguir un modelo adecuado para realizar esta función, implicaría que el agente basado en *Deep Reinforment Learning* no podría competir con otros modelos que actualmente se aplican en los datacenters.

```
{
  "seed": 1995,
  "nb_resources": 2816,
  "nb_jobs": 128,
  "workload_file_path": "/workspace/workloads/my_workload.swf",
  "platform_file_path": "/home/mario/PruebasDelSoftware/PruebasComplejas/
2816_cores_16_20_24_28_25%/platform.json",
  "pybatsim": {
    "log_level": "DEBUG",
    "env": {
      "objective": "energy_consumption",
      "actions": {
        "selection": [
          {"shortest": ["high_gflops", "high_mem_bw", "low_power"]},
          {"first": ["high_gflops", "high_mem_bw"]}
        ],
        "void": false
      },
      "queue_sensitivity": 0.005
    },
    "agent": {
      "type": "LEARNING",
      "run": "train",
      "hidden": 16,
      "lr": 0.005,
      "gamma": 0.99
    }
  }
}
```


Figura 3.6: Ejemplo de fichero de configuración en formato JSON

En consecuencia, es necesario definir varios modelos de comportamiento para poder compararlos y poder observa cuál de los modelos funciona, y además de la manera más eficiente. En el desarrollo de estas comparaciones se comprueban dos modelos para la red neuronal muy distintos que se presentan en las siguientes secciones.

3.3.1.1. Red adaptable al objetivo

Con este modelo se pretende que la red detecte el cambio de objetivo y sea capaz de cambiar a la nueva política que optimiza el objetivo de manera gradual. Es decir, cuando hay un cambio en el objetivo las pérdidas de la red deberían subir debido a que los valores de *reward* recibidos son muy distintos a los valores predichos por la red.

Debido a las pérdidas obtenidas, el agente inteligente cambiará de política gradualmente intentando reducir esta pérdida. Este proceso puede llevar varios episodios de simulación debido a que el cambio de política se realiza de manera gradual, gracias al parámetro de *learning rate* que indica el ritmo al cuál la red neuronal debe aprender.

Gracias a los cambios descritos en la Sección 3.1, la propia estructura del agente permite detectar la pérdida de la red, y con ello el agente cambia de políticas de manera automática. Y por tanto, no es necesario realizar ningún tipo de cambio en el agente inteligente.

Por tanto, con el modelo de red diseñado, se realizan los experimentos correspondientes para comprobar su funcionamiento. En este caso, se repiten experimentos descritos en la Sección 4.2, pero una vez escogida la política se cambia el objetivo para obligar al agente a cambiar de política. Este cambio se realiza una vez que el agente ha seleccionado la primera

política, para comprobar que la selección de la nueva política, no se ve sesgado por la política previamente escogida y planifique de manera incorrecta.

En la ejecución de estos experimentos se pueden observar que en ciertas ocasiones el agente no consigue realizar el cambio de política. Un ejemplo de experimento fallido se puede observar en la Figura 3.7. En este experimento en el episodio 550 se cambia de objetivo en la simulación. En la imagen se puede observar cómo en el episodio 580 la red detecta el cambio de objetivo, pero es incapaz de cambiar de política, debido a que la política óptima es distinta a la previa.

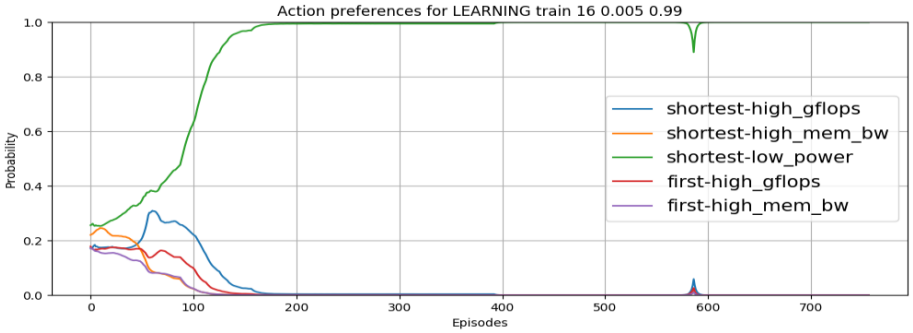


Figura 3.7: Preferencia de acciones del ejemplo para el modelo de red adaptable al objetivo

Lo sucedido es provocado porque la red es capaz de predecir el nuevo valor del *reward* de manera correcta, pero lo realiza excesivamente rápido en comparación con el cálculo de la política. Debido a esto, la política previamente escogida se ve premiada al no detectar pérdidas en el *reward* y no se decide escoger otro tipo de políticas que mejorarían el rendimiento del datacenter para el objetivo dado.

Para este problema se planteó la solución de cambiar el *learning rate* para que el aprendizaje de la red cambie de velocidad para ajustar el aprendizaje de las políticas. Este planteamiento puede parecer buena idea, pero en la realidad el modificar el *learning rate* no es una solución viable. El cambio del *learning rate* puede provocar que la red se descontrole si utilizamos un valor muy grande, o que se quede atascada si aplicamos un valor muy pequeño. Ajustar el *learning rate* para tener un buen rendimiento es muy complicado, ya que no se puede predecir con exactitud cómo va a afectar este valor al aprendizaje de manera exacta.

Por tanto este modelo, aunque efectivo para algunas simulaciones, no es de carácter general y tiene solución muy compleja para el problema previamente expuesto.

3.3.1.2. Reinicio del agente

La segunda opción del modelo consiste en detectar el cambio de objetivo y volver a entrenar a la red desde el principio. Se vuelve a entrenar a la red reiniciando el modelo, para que

vuelva a un estado similar al que se puede encontrar al principio de las simulaciones, donde la red no ha recibido ningún tipo de aprendizaje.

La aplicación original de HDeepRM no contempla el cambio de objetivo en la simulación, por consiguiente es necesario que la aplicación sea capaz de detectar el cambio para que el agente reaccione de la manera correcta. Para esta detección se decidió crear un nuevo fichero de salida para guardar qué objetivo se creó en el episodio anterior. Con esto conseguimos tener de manera sencilla un historial de los objetivos que se han utilizado. Cuando se cambia el objetivo y empieza un nuevo episodio, se detecta que el último objetivo del fichero ha cambiado con respecto al original, y es cuando se toman las medidas correspondientes para que el agente reaccione a estos cambios.

La medida tomada en este caso, es crear un nuevo agente sin entrenar. El agente al no estar entrenado no tiene ninguna prioridad en la selección de políticas. Por tanto, el agente aprende la política óptima de manera independiente al objetivo utilizado en episodios anteriores. Con esto se consigue que el agente encuentre la nueva política con la misma probabilidad como si se realizase en un experimento distinto.

Este modelo proporciona varias ventajas respecto al modelo previamente explicado. En primer lugar se tiene la certeza de que es independiente del orden de los objetivos. Teniendo definido un experimento con distintos objetivos a probar, si realizamos cada objetivo por separado se obtendrá el mismo resultado que si se aplican los objetivos uno detrás de otro. Además otra ventaja es el tiempo de convergencia, el tiempo que tarda el simulador en encontrar la nueva política es similar a entrenar la red en un experimento aparte para el nuevo objetivo.

Por tanto con este modelo, tenemos un agente que es aplicable a cualquier experimento, y además su tiempo de aprendizaje es similar al de experimentos probados con objetivos individuales. Con el modelo realizado e implementado se realizan los experimentos de comprobación que se pueden observar en la Sección 4.4.2.

Con un modelo funcional y eficiente, ahora el simulador es capaz de adaptarse a los cambios de objetivos que puedan suceder durante las simulaciones. Esto mejora en gran medida la calidad del simulador al ser capaz de representar lo que sucede en la realidad en un datacenter, ya que en cualquier momento el administrador del sistema puede cambiar el objetivo a priorizar.

3.3.2. Cambio de Workloads

Una nueva aportación que se ha empezado a probar en el simulador, es el cambio de workload. En las anteriores secciones siempre se realizaban experimentos con un workload fijo. En esta sección se trata de realizar un cambio en el workload durante la simulación y que el agente sea capaz de cambiar de política óptima si es necesario.

Este problema es muy complejo debido a que la cantidad de workloads que se pueden generar para una única plataforma es prácticamente infinita. Pero aún así, es positivo considerar y testear distintos tipos de workloads para comprobar si el agente es capaz de realizar esta operación.

En esta ocasión no se puede plantear un modelo parecido al utilizado para el cambio de objetivos de la Sección 3.3.1. Es por esto que se decide no detectar directamente que la definición del workload ha cambiado, sino que se pretende que el agente detecte automáticamente que los trabajos son distintos y necesitan otra política óptima a aplicar.

La detección del cambio de workload se consigue gracias a que un nuevo workload implica que los valores de *rewards* son distintos, y esto provoca un fallo en la predicción de la red, que conlleva pérdidas en la red. Las pérdidas de la red provocan que tanto la predicción de la política como la del valor del *reward* tengan que cambiar. Ambos valores cambiarán para acercarse a los valores de *reward* retornados por el entorno y así reducir la pérdida. Esto garantiza que si el workload cambia, la red responda con un cambio de políticas para garantizar minimizar la pérdida de la red.

En este caso no será necesario realizar ningún tipo de cambio en la aplicación, debido a que el agente con los cambios realizados en la Sección 3.1, detecta los cambios que se realizan en el workload. Por tanto, con el agente implementado se realizan una serie de experimentos que permiten medir cuál de eficiente es el agente para este tipo de entornos. Estos experimentos se pueden observar en la Sección 4.4.3.

Capítulo 4

Evaluación

En este capítulo se describen los distintos experimentos que se han realizado con HDeepRM para probar su funcionamiento y sus límites a la hora de la planificación de tareas mediante técnicas de *Deep Reinforcement Learning*. Estos experimentos han sido realizados y diseñados para testear las aportaciones realizadas en el Capítulo 3.

4.1. Metodología

La definición y ejecución de experimentos es fundamental para comprobar si las aportaciones realizadas en el capítulo previo contribuyen a mejorar el software para su uso en entornos más realistas. Para cada una de las aportaciones descritas, se crean distintos diseños de experimentos para poner al límite las capacidades añadidas al simulador. Los experimentos comparten aspectos de diseño en común, que serán descritos en esta sección.

El diseño de las distintas pruebas se basan en dos consignas claras, escalabilidad y heterogeneidad. En cuanto a la escalabilidad es muy importante destacar que el número de recursos y el número de trabajos a planificar afectan tanto al tiempo como a la complejidad de la planificación. Por tanto, debido a estos factores se decide realizar distintos tipos de experimentos, los cuales van aumentando tanto el número de recursos como de trabajos. Esto sirve para testear cómo el planificador inteligente se puede ver afectado por estos cambios tanto en su eficiencia, como en la efectividad en la toma de decisiones.

Respecto a la heterogeneidad de los experimentos, se pretende que HDeepRM pueda simular cualquier tipo de plataforma de la manera más realista posible. Por tanto también es de vital importancia que el planificador pueda ejecutar su tarea de manera correcta y esperada, independientemente del entorno en el cual se utiliza este planificador. Por todo ello, los experimentos que se han creado para evaluar al planificador van evolucionando, probando distintos tipos de plataforma.

Con estas consignas, se diseñan los experimentos para validar las aportaciones realizadas al simulador. Para diseñar estos experimentos hay que tener en cuenta todos los aspectos necesarios a definir. Los siguientes pasos, describen el proceso de definición de los experimentos:

- *Definir una necesidad de experimentación*: en este apartado se expresa la necesidad de probar el comportamiento de los métodos de *Deep Reinfocement Learning*. Es necesario definir qué aspecto de la planificación se desea probar, que pueden ir desde probar la eficacia de la planificación hasta probar la capacidad de adaptación de la red a cambios en el objetivo. Estos aspectos son las aportaciones descritas en el Capítulo 3, las cuales necesitan ser testeadas en tiempo de ejecución.

- *Diseño de plataforma y workload*: una vez definido el por qué realizar un experimento, necesitamos definir tanto en qué tipo de plataforma se van a realizar las pruebas, como qué tipos de tareas se van a planificar. Los elementos para definir tanto los trabajos del workload como la plataforma están definidos en la Sección 2.2.

  En cuanto a la plataforma cabe destacar un elemento básico como son los procesadores, en los cuales hay que definir ciertas características. Estas características son el número de cores que contiene, la capacidad computacional de los cores medida en GFLOPS, el ancho de banda total disponible para los cores en GB/s y la potencia consumida por core en vatios. Y para los tipos de trabajos se especifican en concreto el tiempo de llegada de los trabajos, el tiempo de ejecución total en segundos y el ancho de banda total requerido GB/s.

- *Describir el comportamiento del planificador*: una vez tenemos todo el entorno necesario para la simulación, es fundamental poder predecir cuál es el comportamiento que deberá tomar el planificador. Esta descripción se puede obtener desde dos puntos de vista diferentes pero a la vez complementarios. En un primer lugar mediante un análisis teórico de la plataforma y el workload, permitiendo realizar una predicción muy precisa del comportamiento del planificador. En segundo lugar, se realiza una experimentación separada de todas las políticas clásicas que puede escoger el planificador. Esto permite identificar cuál es la mejor política para el experimento y con ello verificar a posteriori, si el agente inteligente ha tomado las decisiones más adecuadas.

  Es importante mencionar las políticas que se seleccionan para la realización de los experimentos. En los experimentos expuestos se utilizan cinco políticas clásicas, las cuales priorizan distintos tipos de trabajos y recursos a la hora de planificar. Se utilizan solo estas cinco políticas debido a que aumentar el número de políticas no influye de manera determinante en el tiempo de convergencia de la red ni en su efectividad. Además de que aumenta el número de políticas complica en gran medida el análisis de los resultados. Las políticas utilizadas en los experimentos son: *shortest-high-mem_bw*, *first-high-mem_bw*, *shortest-low_power*, *shorst-high_gflops* y *first-high_gflops*.

- *Realización del experimento*: en los experimentos cabe destacar el uso de distintos hiperparámetros los cuales permiten modificar cómo se comporta el agente inteligente en el aprendizaje. Es necesario seleccionar mediante experimentación el valor más adecuado para que el planificador aprenda cuales son las mejores políticas de la forma más

eficiente. Además en todos los experimentos se utiliza la red neuronal del planificador basada en el modelo *actor-critic*, debido a su mejor rendimiento como se explicó en la Sección 2.2.

- *Comprobación de los resultados*: cuando se acaba de simular todos los trabajos es necesario comprobar los resultados obtenidos del aprendizaje y verificar si el agente inteligente ha seleccionado la política óptima definida previamente. Para comprobar el correcto funcionamiento se utilizan tres métricas fundamentales:

  - *Preferencia de acciones del agente*: el agente calcula la probabilidad de seleccionar cada una de las políticas disponibles. Esta probabilidad varía en función del aprendizaje que se va realizando, por tanto con el avance de los episodios de entrenamiento el agente tenderá a aumentar la probabilidad de las políticas que optimizan el objetivo. Si al agente aumenta la probabilidad de políticas no óptimas el experimento se puede considerar fallido, al no conseguir planificar de manera óptima. Por tanto, sabiendo cuál es la preferencia de acciones del agente se puede conocer el resultado del experimento.
  - *Pérdida de la red*: otra medida fundamental para comprobar el aprendizaje son las pérdidas de la red. Las pérdidas de la red indican el proceso del aprendizaje del agente. Unas pérdidas altas indican que el agente está aún en proceso de aprendizaje y no se decanta por ninguna política en concreto. En cambio, un valor de pérdida bajo, es decir cercano a cero, está relacionado con la selección del agente por una política.
  - *Eficiencia de las Políticas Clásicas*: como se mencionó en el punto anterior, se realiza una ejecución por separado de cada política para el mismo experimento. Esta ejecución permite conocer cuál es la política óptima para un objetivo en concreto, comparando los valores finales de las métricas que determinan la función objetivo de cada política.

  Si alguna de las dos primeras métricas no alcanzan los valores esperados para la simulación, se considera que el agente no ha alcanzado los objetivos planeados. Si no se consigue el objetivo, se realizan distintos tipos de acciones dependiendo de los resultados obtenidos. Una de estas acciones sería repetir el experimento varias veces más para comprobar que el resultado obtenido en la primera prueba no ha sido producido fruto de la probabilidad de error inherente de los sistemas inteligentes.

  Otra opción es modificar los hiper-parámetros utilizados por si su valor ha sido un factor clave en el aprendizaje de agente. Una vez efectuadas estas dos soluciones, se pasaría a revisar cada uno de los puntos referidos previamente, observando si hay errores en la plataforma, el workload o también en el comportamiento planeado teóricamente por si hay errores en la interpretación de los datos utilizados.

Una vez realizado un experimento, se puede decir que ha sido un éxito si se consigue llegar a una conclusión del por qué, las decisiones que el planificador ha aprendido son las óptimas para el objetivo dado. Con un experimento exitoso no es suficiente, se repite el experimento

entre 20 y 30 veces dependiendo de la complejidad y de los resultados obtenidos para poder verificar que el resultado no ha sido azaroso y se puede dar por válido el experimento. Por lo tanto, los valores presentados en las siguientes secciones son resultado de ese número de ejecuciones por experimento.

Además de todo lo expuesto anteriormente, para comprobar una hipótesis se deciden ejecutar experimentos con una plataforma y un workload reducido, el cual permite depurar el experimento con un control total sobre el estado de la simulación. Una vez que los experimentos reducidos se han realizado con éxito, se van realizando progresivos aumentos en ambos aspectos para observar la reacción de la planificación en temas de escalabilidad y apreciar el impacto en el aprendizaje del modelo.

El desarrollo de las pruebas se ha realizado en un portátil comercial, en concreto un HP Pavilion 15-ak003ng que contiene cuatro núcleos físicos i7-6700HQ a 2,6GHZ con hyperthreading y dieciseis gigabytes de memoria RAM. Cabe destacar que el desarrollo de las pruebas se han realizado mediante una máquina virtual que tenía capacidad para utilizar dos de los núcleos físicos y la mitad de la memoria RAM del sistema.

4.2. Experimentos de escalabilidad

4.2.1. Experimentos básicos

En esta sección se describen los experimentos llevados a cabo para la comprobación de las aportaciones realizadas de la sección 3.1. Con estas aportaciones se pretende mejorar la escalabilidad del planificador inteligente, y mediante la realización de los siguientes experimentos se comprueba el correcto funcionamiento de las aportaciones. Los primeros experimentos realizados en el simulador se diseñan en un entorno simple, para poder escalar los trabajos y recursos de manera rápida y sencilla.

En el entorno de la primera simulación tenemos una plataforma que consta de un cluster, con un único nodo, y dos procesadores dual-core. Estos dos procesadores tienen respectivamente una capacidad de computo de 4 GFLOPS y 4,4 GFLOPS por core aportando un total de cómputo de 8 y 8,8 GFLOPS por procesador. Los cores con menor capacidad de cómputo tienen una potencia de 80 vatios máximo, mientras que los otros cores tienen una potencia máxima 100 vatios. Y para terminar con la plataforma, ambos procesadores tiene cada uno, un ancho de banda de 32 GB/s. En cuanto al workload tenemos dos trabajos que llegan a la vez en el instante inicial de la simulación y ambos requieren de un ancho de banda de 24 GB/s durante 5 segundo. Además cabe destacar el uso de un *learning rate* de 0.005 y un objetivo de minimizar el *makespan*.

Con estos datos se puede analizar de forma teórica que el comportamiento del planificador debe de ser el escoger políticas que no saturen el ancho de banda de los procesadores para

que el tiempo en ejecutar el trabajo no aumente considerablemente. Si no se toma este tipo de políticas es posible que los dos trabajos, se planifiquen en el mismo procesador. Dado que la suma de los anchos de bandas es mayor al ancho de banda disponible del procesador, los trabajos tardarán más en ejecutarse debido a la congestión provocada y empeorarán el *makespan*. Por tanto para poder observar este comportamiento necesitamos políticas que priorizan el comportamiento enunciado, y en este caso se utilizan las políticas de *shortest-high-mem_bw* y *first-high-mem_bw*.

En cuanto a los resultados, en la Figura 4.1 se observa en el eje X el número de episodios realizados, y en el eje Y la probabilidad de selección de cada política. Cada política en esta figura, está representada mediante una línea, que evoluciona en función de la probabilidad que tiene el agente de seleccionarla, en cada episodio. Cada episodio representa una ejecución completa del workload en la plataforma. En esta figura el planificador, sobre el episodio 50, descartó las políticas que empeoraban el *makespan* y decidió priorizar *shortest-high-mem_bw* y *first-high-mem_bw*. Como se explicó en el párrafo anterior estas dos políticas son las que minimizan el *makespan* y por tanto, el agente selecciona correctamente una de las políticas óptimas.

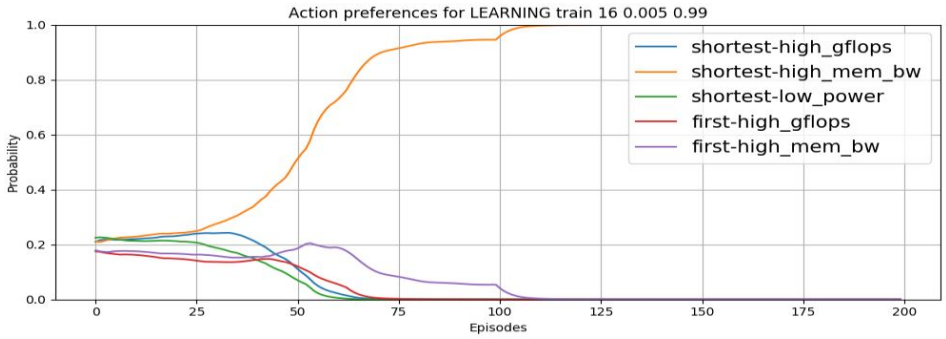


Figura 4.1: Preferencia de acciones del experimento básico con 4 cores - 200 episodios

En la Figura 4.2 se observa el valor del *makespan* obtenido al realizar el mismo experimento solamente con una política fija. En este caso sucede exactamente lo previsto en el análisis teórico, las políticas *shortest-high-mem_bw* y *first-high-mem_bw* son las que minimizan el *makespan* y por tanto, las óptimas para el entorno simulado.

Por otro lado, las pérdidas de la red representan como el agente realiza aprendizaje a lo largo de la simulación. En la figura 4.3 se muestra en el eje X los episodios ejecutados, y en el eje Y el valor de pérdida del agente. Durante los primeros 50 episodios la red varía fuertemente su pérdida debido a que no sabe cuál es la política óptima. A partir del episodio 50, cuando ya no se seleccionan con tanta frecuencia políticas no óptimas, las pérdidas de la red tienden al valor cero, lo que representar que la red ha decidido una política óptima para el experimento. Sobre el episodio 95 se ve un pico en la pérdida, esto fue debido a la selección puntual de una política no óptima que por probabilidad pueden ser seleccionadas.

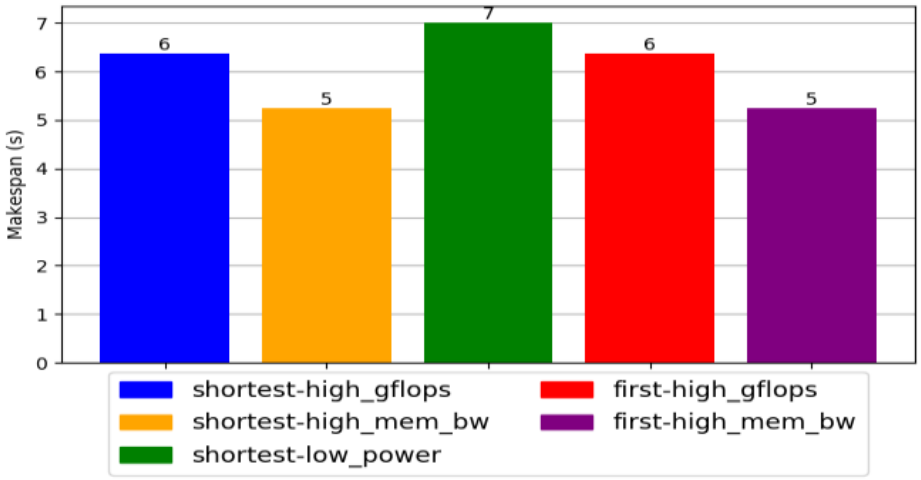


Figura 4.2: Makespan de las 5 políticas en la plataforma con dos procesadores

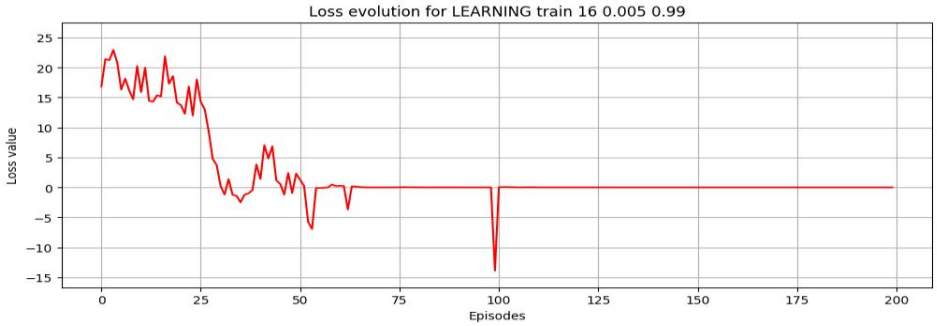


Figura 4.3: Pérdidas de la red del experimento básico con 4 cores - 200 episodios

Con este experimento comprobamos que las aportaciones realizadas no afectan a las capacidades de planificar ejemplos muy sencillos. A continuación se realizar un escalado de este experimento para comprobar la escalabilidad del planificador y sus límites de planificación.

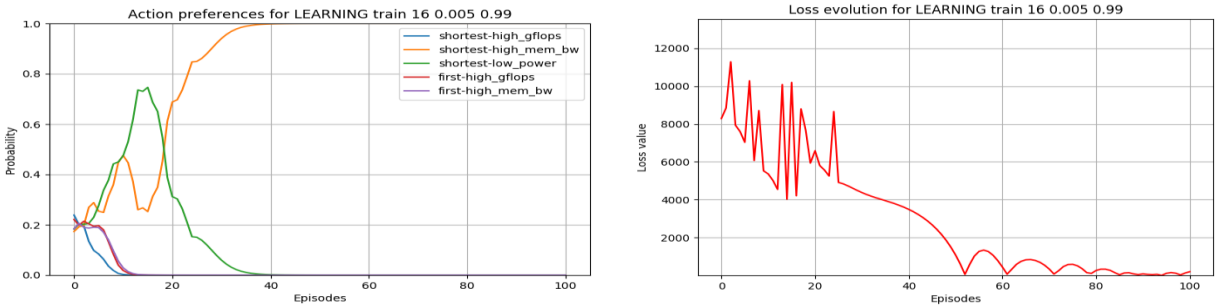
4.2.2. Escalado de Plataformas y Workloads

En los siguientes experimentos se aumenta el número de procesadores duplicando su cantidad en cada iteración. Para que los experimentos sigan teniendo el mismo paradigma de comportamiento se duplica el número de trabajos utilizando trabajos con el mismo perfil, a los previamente descritos. Los nuevos experimentos siguen con el mismo problema del ancho de banda del procesador y por tanto las políticas óptimas serán *shortest-high-mem_bw* y *first-high-mem_bw*, que priorizan los recursos con mayor ancho de banda disponible.

Se simularon hasta 2048 cores y 1024 trabajos, haciendo experimentos cada vez más grandes. Este tamaño máximo de la plataforma y el workload, se podría aumentar utilizando un

computador más potente. Las siguientes gráficas y explicaciones se basan en el experimento de 2048 cores, pero aún así las explicaciones son equivalentes a los experimentos con menor número de cores. El experimento se ha realizado con el mismo *learning rate* y el mismo objetivo que en el caso anterior.

En este experimento se comprueba como el resultado es similar al caso previo. En las Figuras 4.4(a) y 4.4(b), se puede ver cómo tanto la acción tomada, como el desarrollo de las pérdidas de la red son consecuentes con el entorno de simulación con el que estamos trabajando.



(a) Preferencia de acciones del experimento básico con 2046 cores - 100 episodios

(b) Pérdida de la red del experimento básico con 2046 cores - 100 episodios

Figura 4.4: Experimento básico de escalabilidad con 2046 cores

En este caso se observa que el agente duda entre dos políticas distintas *shortest-high-mem_bw* y *shortest-low_power*. Una vez se decanta por la correcta se puede observar una reducción considerable en la función de pérdida de la red. Este resultado es fundamental, porque permite saber que ante dos situaciones iguales, la escalabilidad del problema a tratar no influye en el resultado final. Aún así la plataforma utilizada era muy poco heterogénea y la diferencia entre los procesadores era muy pequeña. Además de contar con procesadores con un número de cores no realistas, en comparación a los encontrados en el mercado actualmente.

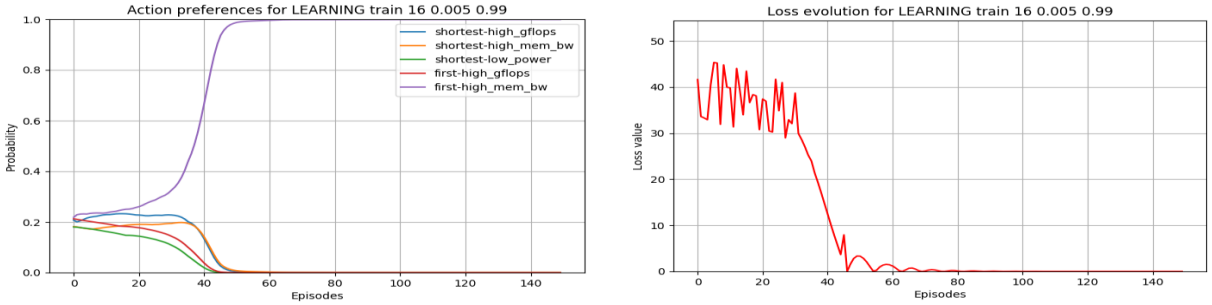
Por tanto, se decide implementar nuevos experimentos que cubran las desventajas de la plataforma anterior. Con los nuevos experimentos se pone a prueba la capacidad que tiene el simulador de poder planificar en un entorno más heterogéneo y realista sin perder la capacidad que de escoger las políticas más óptimas para el experimento.

En la nueva plataforma tenemos una estructura con dos nodos distintos, uno de los cuales tiene dos procesadores de 16 y 20 cores con capacidad de ejecutar 4 GFLOPS por core, con una potencia máxima de 80 vatios. Y el segundo nodo con dos procesadores de 24 y 28 cores, de capacidad de cómputo de 4,4 GFLOPS por core, y 100 vatios de potencia máxima. Todos los procesadores tienen un ancho de banda de 32 GB/s y en total se usan en la plataforma 88 cores. Se utiliza un workload con cuatro trabajos que requieren cada uno un ancho de banda de 24 GB/s durante 5 segundo.

El problema al que se enfrenta el planificador en este experimento es el mismo que en los experimentos previos. Se pretende seleccionar políticas que no limiten el ancho de banda de los procesadores y limiten el *makespan*. Por tanto el planificador deberá escoger las políticas

shortest-high-mem_bw o *first-high-mem_bw*, las cuales impiden que el ancho de banda del procesador se desborde y el *makespan* aumente. En el experimento se utiliza un *learning rate* de 0.005 y un objetivo de minimizar el *makespan*

En las figuras 4.5(a) y 4.5(b), se puede apreciar cómo la convergencia de la red a una de las dos políticas óptimas ha funcionado con éxito. Las pérdidas consiguen alcanzar valores cercanos a cero y además el agente consigue planificar con una de las acciones óptimas para el experimento en concreto.



(a) Preferencia de acciones del experimento heterogéneo con 88 cores - 150 episodios

(b) Pérdida de la red del experimento heterogéneo con 88 cores - 150 episodios

Figura 4.5: Experimento heterogéneo con 88 cores

Cabe destacar que en este caso, la política seleccionada mostrada en la gráfica 4.5(a) es distinta a la presentada en los anteriores experimentos. Aún así esta política optimiza el objetivo de la simulación, como se puede ver en la Figura 4.6.

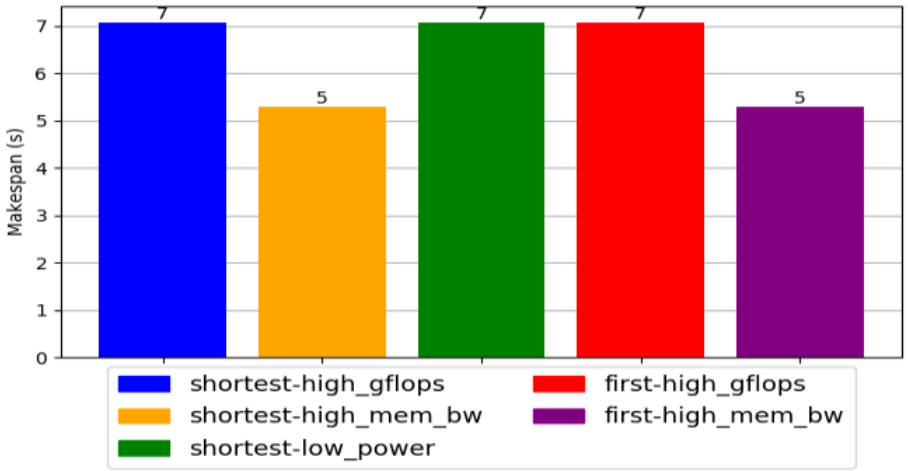
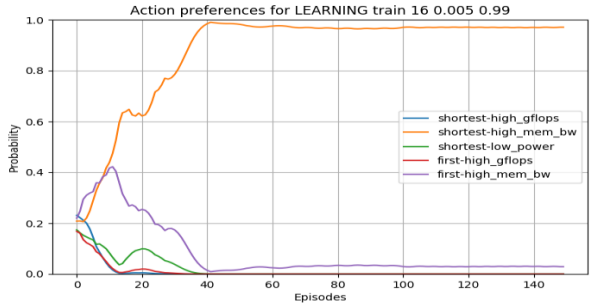


Figura 4.6: Makespan para las políticas en la plataforma de 88 cores

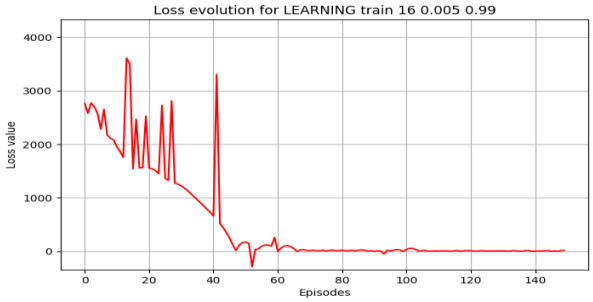
Estos experimentos son una buena aproximación, pero para acercarnos más a la realidad deberíamos realizar estos experimentos con muchos más cores. En las Figura 4.7 se muestran los resultados del mismo experimento previamente descrito, pero multiplicando por 64 el número de nodos, dando un total de 5632 cores. También se realizaron experimentos más pequeños con un total de 176, 352, 704, 1408 y 2816 cores que tuvieron un resultado favorable,

pero el de 5632 cores es el más representativo al tener la mayor cantidad de cores y de trabajos. Además se amplia el workload a un total de 256 trabajos, iguales a los previamente descritos. Con esta acción se pretende escalar el experimento anterior intentando llegar a las mismas conclusiones, debido a que el problema principal de la limitación del ancho de banda se sigue manteniendo.

El problema de planificación para este experimento con 5632 cores es el mismo que en los experimentos previos provocando que las políticas óptimas vuelvan a ser *shortest-high-mem_-bw* y *first-high-mem_bw*. Los resultados de las Figuras 4.7(a) y 4.7(b) muestran cómo el agente vuelve a predecir perfectamente la política óptima reduciendo las pérdidas hasta cero. Cabe destacar que la probabilidad de la política escogida por el agente, en este caso la *shortest-high-mem_bw*, no llega a ser de 1.0, sino que es de entorno al 0.97. Esto es debido a que el agente concede una probabilidad de 0.03 a la otra política óptima *first-high-mem_bw*. Esto indica que el agente cuando tiene dos o más políticas óptimas puede tardar unos episodios más en decantarse completamente por una de ellas debido a ser equivalentes.



(a) Preferencia de acciones del experimento heterogéneo con 5632 cores - 150 episodios



(b) Pérdida de la red del experimento heterogéneo con 5632 cores - 150 episodios

Figura 4.7: Experimento heterogéneo con 5632 cores

Estos resultados son un gran avance para el planificador, ya que se consigue simular un ambiente bastante heterogéneo y con una cantidad de recursos a planificar bastante grande, en comparación con el planificador original. Con estos resultados se pueden comprobar que la aportación a la escalabilidad de la aplicación ha sido un éxito. Se ha conseguido escalar experimentos básicos con muy pocos recursos y trabajos, consiguiendo simular con planificaciones correctas experimentos con miles de recursos y trabajos.

4.3. Experimentos de Objetivos de Energía y EDP

En esta sección se describen los experimentos de comprobación para las aportaciones realizadas en la sección 3.2, donde se crean dos nuevos objetivos de planificación *energy_consumption* y *energy-delay product*.

Una vez implementados los objetivos previamente mencionados es fundamental realizar experimentos básicos que abarquen ciertas situaciones donde la implementación pueda fallar y

se vea comprometido su efectividad. Los experimentos son la manera más fiable y completa de comprobar el correcto funcionamiento de los objetivos. Debido a que los objetivos sirven al planificador para priorizar ciertas políticas, comparar tanto los cálculos de los objetivos como las acciones que toma el agente, son una medida perfecta para comprobar el funcionamiento de los objetivos.

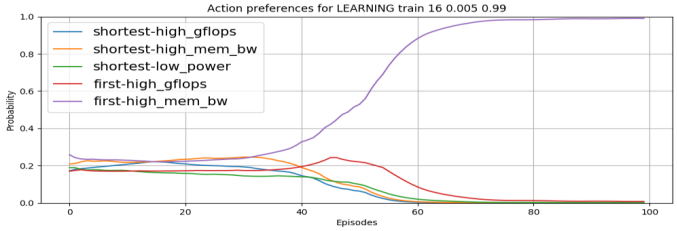
Por tanto se plantean tres experimentos distintos, en los cuales con la misma plataforma, se cambia el workload para comprobar cómo ante el orden de los trabajos el *reward* cambia y lo hace de manera correcta. La plataforma utilizada es la misma que la aplicada en el primer experimento mostrado en la sección 4.2. Los tres workloads tienen dos trabajos y se diferencian en el momento en el cual el segundo trabajo llega a la cola, y por tanto se plantean tres posibilidades:

- *Trabajos sincronizados*: este es el ejemplo más sencillo. Ambos trabajos se inician al mismo tiempo, y por tanto el cálculo del *reward* se realiza al final de la ejecución del trabajo que más tarda.
- *Inicio desplazados*: mientras se ejecuta el primer trabajo, el segundo trabajo llega y esto provoca un cálculo del *reward*. Luego se finaliza la ejecución de ambos trabajos, y se vuelve a calcular un nuevo *reward*.
- *Inicio secuencial*: en este caso se hacen tres cálculos del *reward*. Un primer cálculo al finalizar el primer trabajo, un segundo cuando llega el siguiente trabajo, y un tercero cuando se finaliza el último trabajo.

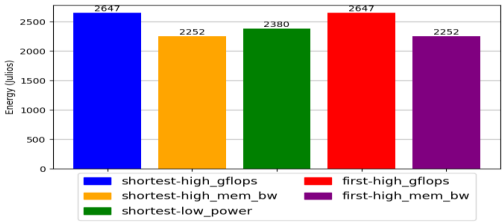
Con estos tres tipos de workloads definidos, se pueden ejecutar tres experimentos distintos por cada objetivo, para comprobar el funcionamiento tanto con un agente clásico, como con el agente inteligente. En este caso se simula con los agentes clásicos para comprobar que los cálculos de los *rewards* para cada política individual son correctos, y además comprobar qué política optimiza el objetivo.

Los resultados de estos experimentos son muy positivos. En la Figura 4.8, se pueden ver los resultados con los tres tipos de workloads previamente descritos, con el objetivo *energy_consumption*. En las gráficas de la figura se puede observar cómo el agente decide siempre una política que reduce el gasto energético, que se pueden observar en las Figuras 4.8(b), 4.8(d), 4.8(f), donde tenemos para cada política el gasto energético que supone aplicarla para la simulación. En las figuras solo se observan las gráficas para el objetivo *energy_consumption*, pero estos resultados fueron conseguidos de igual manera con el objetivo de *energy-delay product*. Además debido a que tanto la plataforma como el workload son bastante simples, se han realizado los cálculos de manera manual de los valores de los objetivos y se han comparado con los cálculos obtenidos en la simulación. De las dos formas se han conseguido los mismos resultados lo cuál asegura que la implementación del *reward* se ha realizado de manera correcta.

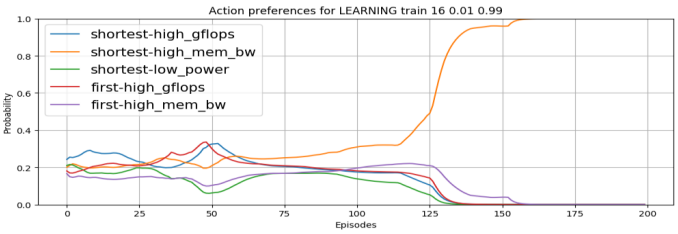
Una vez comprobado el funcionamiento básico de estos objetivos se pretende que como mínimo, el agente sean capaz de realizar las mismas tareas que realiza el objetivo *makespan*



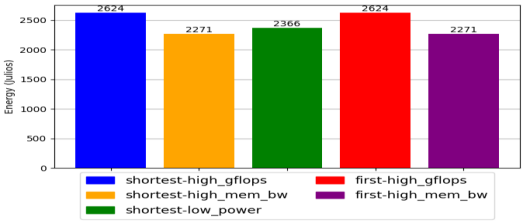
(a) Preferencia de acciones del experimento con trabajos sincronizados



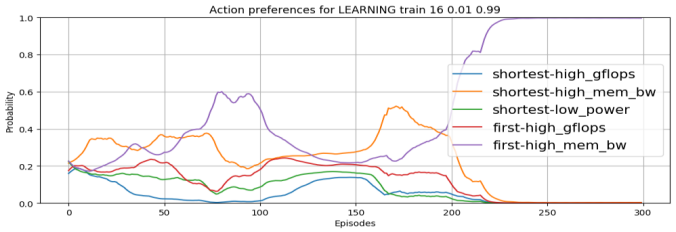
(b) Gasto de energía de cada política del experimento con trabajos sincronizados



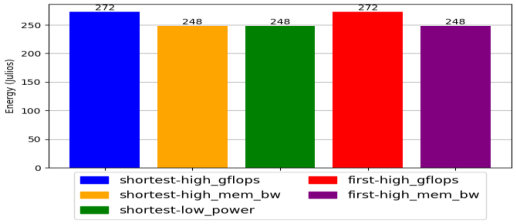
(c) Preferencia de acciones del experimento con inicio desplazados



(d) Gasto de energía de cada política del experimento con inicio desplazados



(e) Preferencia de acciones del experimento con inicio secuencial



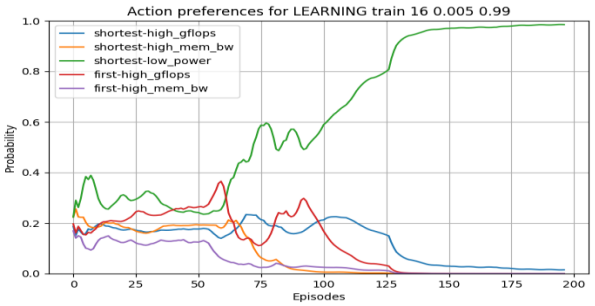
(f) Gasto de energía de cada política del experimento con inicio secuencial

Figura 4.8: Preferencia de acciones y gasto energéticos de las políticas para el objetivo de *energy_consumption*

con una eficiencia similar. Por ello se repiten los experimentos de la Sección 4.2 con las plataformas de 704, 2816 y 5632 cores y sus workloads correspondientes, debido a que fueron las plataformas más heterogéneas y con más recursos, pero en este caso se usan los objetivos de *energy_consumption* y *energy-delay product*. Los resultados mostrados a continuación son realizados para el objetivo *energy_consumption*, pero las conclusiones de los resultados son equivalentes a los experimentos con el objetivo de *energy-delay product*.

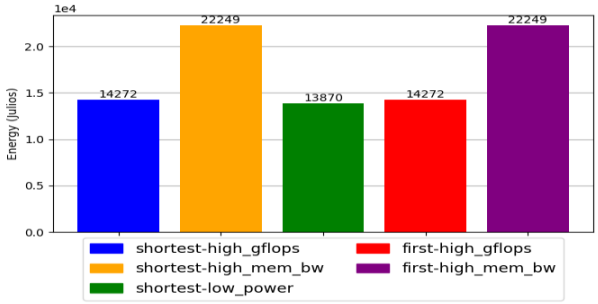
Los resultados con la nueva plataforma se pueden observar en las siguientes figuras. En la Figura 4.9(a), se pueden observar como a partir del episodio 100 el agente toma la decisión de decantarse por la política *shortest-low_power*. Esta decisión como se puede apreciar en la Figura 4.9(c), es la correcta debido a que la política de *shortest-low_power* es la que tiene un menor consumo de energía.

También se puede comprobar en la Figura 4.9(b), como a partir del episodio 100 las pérdidas de la red, se aproximan a valores cercanos a cero. Esto indica que la red ha convergido correctamente tanto a una política óptima, como a predecir el valor del *reward* adecuado.



(a) Preferencia de acciones para el experimento de 5632 cores con el objetivo de *energy_consumption*

(b) Pérdidas de la red para el experimento de 5632 cores con el objetivo de *energy_consumption*



(c) Consumo de energía de las políticas clásicas pare el experimento de 5632 cores con el objetivo de *energy_consumption*

Figura 4.9: Experimento de comprobación del objetivo *energy_consumption* con 5632 cores

Con estos experimentos conseguimos comprobar el perfecto funcionamiento de los dos objetivos implementados, además de igualar el nivel de comprobación de los nuevos objetivos con los del *makespan*. Con esta aportación conseguimos tener un simulador que es capaz de planificar priorizando tres objetivos distintos, uno basado en el rendimiento, el segundo comprueba el consumo energético y el tercero prioriza la eficiencia energética.

4.4. Experimentos realistas

4.4.1. Experimento Básico

Un objetivo fundamental y principal del proyecto es experimentar con una plataforma y un workload realista, y en esta sección se muestran experimentos que simulan situaciones que se pueden encontrar en datacenters reales.

El objetivo de este primer experimento es comprobar si el agente es capaz de planificar con una plataforma y workload real. En este experimento se utiliza la descripción del Gaia Cluster de la Universidad de Luxemburgo [28]. Este cluster heterogéneo está compuesto de 153 nodos, con 342 procesadores y sumando un total de 2280 cores. La descripción detalla

del cluster se puede observar en el cuadro 4.1. En cuanto al workload utilizado, se usa una traza real ofrecido por el Parallel Workloads Archive (PWA) [29], el cual ofrece un workload en formato *SWF* del cluster Gaia con 51987 trabajos recogidos durante tres meses, desde mayo hasta agosto del año 2014. Debido a las limitaciones del computador utilizado, las pruebas se realizan con 932 trabajos del workload original. Para el experimento utilizamos un *learning rate* de 0.005 y además se utiliza el objetivo *makespan*.

Tipo de Nodos	Nodos	Tipo de Procesador	Procesadores	Cores	GFLOPS	Mem(GB)	MBW(GB/s)
Bullx B500 (0)	60	Xeon L5640	2	6	9.04	48	32
Bullx B505 (0)	2	Xeon L5640	2	6	9.04	96	32
Bullx B505 (1)	10	Xeon L5640	2	6	9.04	96	32
Bullx S6030	1	Xeon E7-4850	16	10	8	1024	51.2
Dell R820	1	Xeon E5-4640	4	8	19.2	1024	51.2
Dell R720	5	Xeon E5-2260	2	8	17.6	64	51.2
Bullx B500 (1)	72	Xeon X5670	2	6	11.72	48	32
Delta D88x-M8-BI	1	Xeon E7-8880 v2	8	15	20	3072	85
SGI UV-2000	1	Xeon E5-4650 v2	16	10	19.2	4096	59.7

Cuadro 4.1: Configuración del UniLu Gaia cluster.

En este caso realizar un estudio de cómo se comportará la plataforma junto con el workload es una tarea excesivamente compleja de analizar, debido al tamaño de ambos. Por tanto la mejor manera de comprobar el funcionamiento de la red, es ejecutar cada una de las políticas clásicas de manera independiente, para observar cuál de ellas es la mejor posible. En la figura 4.10 se puede ver cómo la política que minimiza el *makespan* es la de *shortest-low_power*, por tanto, el planificador debería priorizar esta política antes que el resto. Cabe destacar que una política como es *shortest-low_power*, que prioriza el ahorro de energía, sea a su vez también la que mejor optimiza el rendimiento en la plataforma. Esta apreciación hace aún más interesante el experimento, al tener el agente que decidir una política que de por sí, no sería escogida para priorizar el *makespan*.

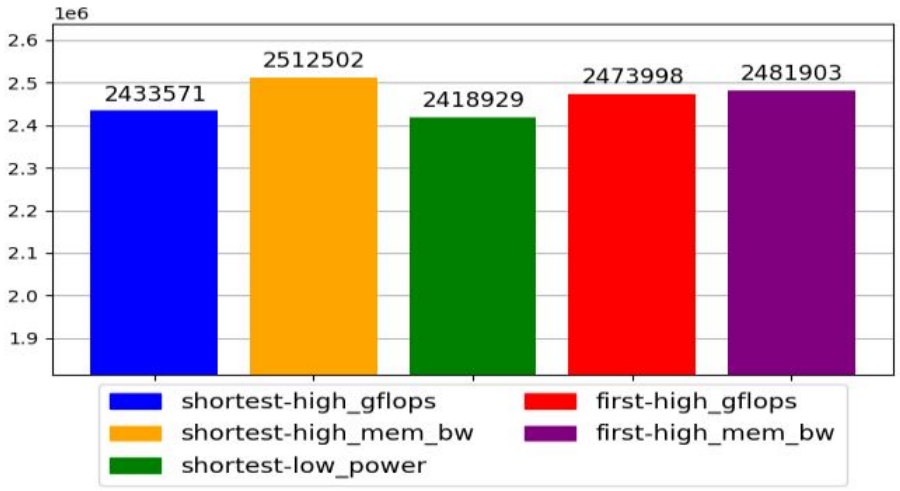
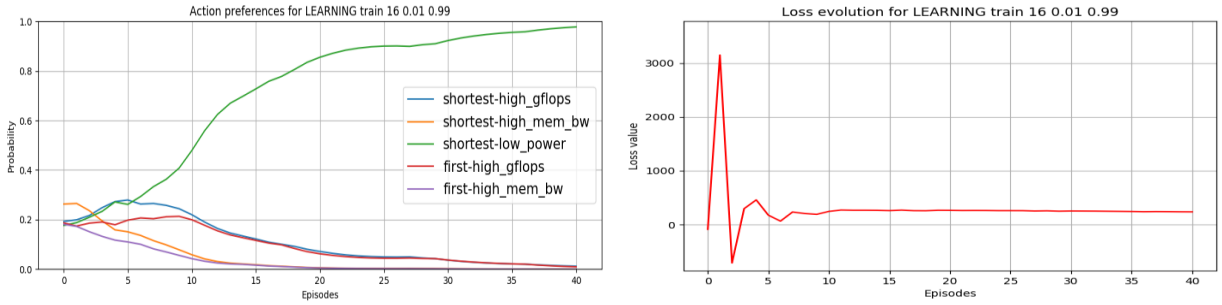


Figura 4.10: Comparación del Makespan de las Políticas Clásicas en el datacenter Gaia

Los resultados del experimento previo son muy positivos, en las figuras 4.11(a) y 4.11(b) se puede observar cómo el agente aprende en menos de 40 episodios la política óptima

previamente descrita. Respecto a las pérdidas también se observa cómo los valores de pérdida rondan el valor cero, lo cual indica la convergencia de la red.



(a) Preferencia de acciones del experimento con la plataforma de Gaia - 40 episodios

(b) Pérdidas de la red en el datacenter Gaia - 40 episodios

Figura 4.11: Experimento en el cluster de Gaia

Este resultado es muy alentador, debido a que es la primera vez que el planificador inteligente, selecciona la política óptima en una plataforma y workload reales. Esto es una primera aproximación a las capacidades que puede llegar a tener las técnicas de *Deep Reinforcement Learning* en la planificación de tareas en un datacenter real. Esto abre la puerta a HDeepRM a simular ejemplos de plataforma mucho más grandes y con workloads más complejos.

A continuación, los siguientes experimentos comprueban las aportaciones de las Secciones 3.3.1 y 3.3.2. Estas aportaciones se realizaban para poder simular situaciones que suceden en datacenters reales, como pueden ser el cambio de objetivos en las simulaciones o el cambio de los tipos de trabajos que recibe el datacenter.

4.4.2. Entornos dinámicos: cambio de objetivos

En primer lugar, se comprueba el cambio de objetivos dinámicos. En este caso, para probar el comportamiento del agente, se realiza un cambio de objetivo cuando el agente ha decidido qué política escoger para un objetivo en concreto. Esto permite saber que el agente no tiene ninguna dependencia previa a la política seleccionada, y es capaz de aprender para la misma plataforma y workload pero con distinto objetivo una nueva política óptima. Para el cambio de objetivo se deciden utilizar los tres objetivos implementados, *energy_consumption*, *makespan* y por último *energy-delay product*.

Los experimentos realizados para comprobar la aportación se han realizado con plataformas y workloads de la sección 4.2, en concreto con las plataformas 704, 2816 y 5632 cores junto con sus respectivos workloads. Los resultados mostrados y descritos se han realizado con la plataforma y el workload de 5632 cores, pero estos resultados son equivalentes para el resto de plataformas. El *learning rate* utilizado en el experimento es de 0.005.

El experimento realizado consiste en cambiar el objetivo de planificación una vez el agente ha

seleccionado la política óptima. Esta operación se realiza en repetidas ocasiones para aportar más dificultad a la planificación, y comprobar el funcionamiento con distintos objetivos. En primer lugar se utiliza el objetivo de *energy_consumption*, que se cambiaría por el *makespan*, en tercer lugar se utilizaría el *energy-delay product* y para finalizar se volvería a escoger el *makespan*. La asignación de estos objetivos no es azarosa, se selecciona el siguiente objetivo sabiendo que la nueva política óptima es distinta a la previamente seleccionada. Esto permite saber si existen dependencias en la selección de la nueva política, debido a que si se cambia de objetivo por otro que comparte la misma política óptima, es más complejo conocer si la nueva selección de política es debido a un error por la dependencia con el objetivo previo, o es un acierto en la planificación y el agente funciona correctamente.

Los resultados de la simulación se pueden observar en las Figuras 4.12 y 4.13(a). En la primera imagen se observa como una vez el agente ha convergido para cada política se decide cambiar al siguiente objetivo. Estos cambios suceden en los episodios 400, 800 y 1100. Como se puede ver en experimentos con la misma plataforma en las Secciones 4.2 y 4.3, tanto para el objetivo *energy_consumption* como para el *makespan*, el agente escoge correctamente la política correcta en cada caso. Para el objetivo *energy-delay product*, no se había mencionado cuál es la política óptima previamente, pero con los experimentos de comprobación de este objetivo mencionados en la Sección 4.3 obtenemos la Figura 4.13(b). En la figura previa se muestra que la política óptima es el *shortest-low_power* al ser la que menos valor de energía por tiempo consume, y como se puede ver en la Figura 4.12 entre los episodios 800 y 1100 donde se utiliza el objetivo de *energy-delay product* el agente selecciona la política correcta.

En la Figura 4.13(a) que representa las pérdidas de la red se puede observar cómo cuando hay un cambio en el objetivo aumentan las pérdidas debido al nuevo aprendizaje que tiene que realizar la red para el nuevo objetivo. Este comportamiento es el buscado para el agente en esta situación, y gracias a este aumento el agente aprende una nueva política óptima para el objetivo. Sobre el episodio 300 se aprecian unas pérdidas importantes, pero es debido a la selección de un política no óptima que provoca esta pérdida tan alta.

Cabe destacar que en las dos etapas en las cuales se utiliza el objetivo de *makespan*, entre los episodios 400 a 800 para la primera etapa, y entre el episodio 1100 y 1400 para la segunda, se seleccionan dos políticas distintas. Aún así el resultado es correcto, ambas políticas optimizan a la vez el valor del *makespan*, debido a que los valores para este objetivo son el mismo en ambas políticas.

Estos resultados comprueban la eficacia de la aportación realizada en la Sección 3.2, añadiendo al simulador, y sobretodo al agente, una característica fundamental y muy utilizada en los datacenters reales como son los cambios de objetivos de planificación.

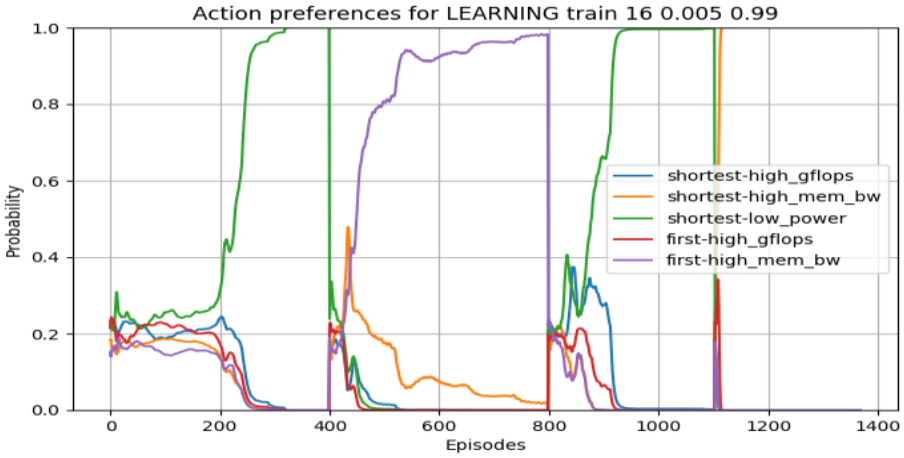
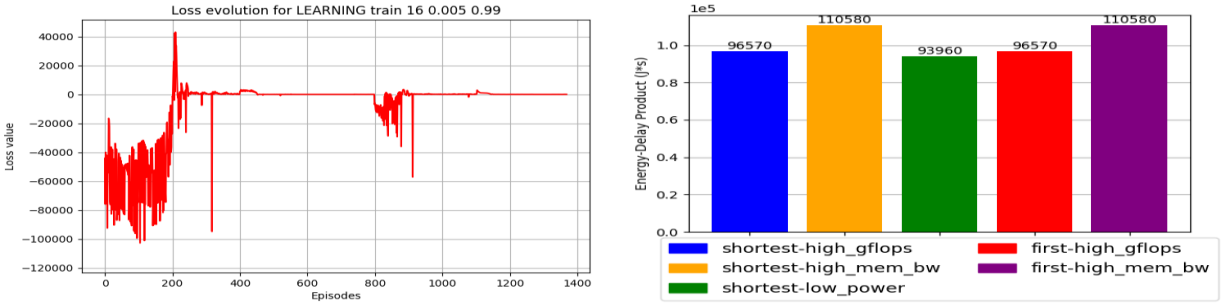


Figura 4.12: Preferencias de acciones en los cambios de objetivos con 5632 cores - 1400 episodios



(a) Pérdidas de la red en los cambios de objetivos - 1400 episodios

(b) Eficiencia de las políticas básicas para el objetivo de *energy-delay product* en el experimento del cambio de objetivos

Figura 4.13: Experimento de cambio de objetivos con 5632 cores

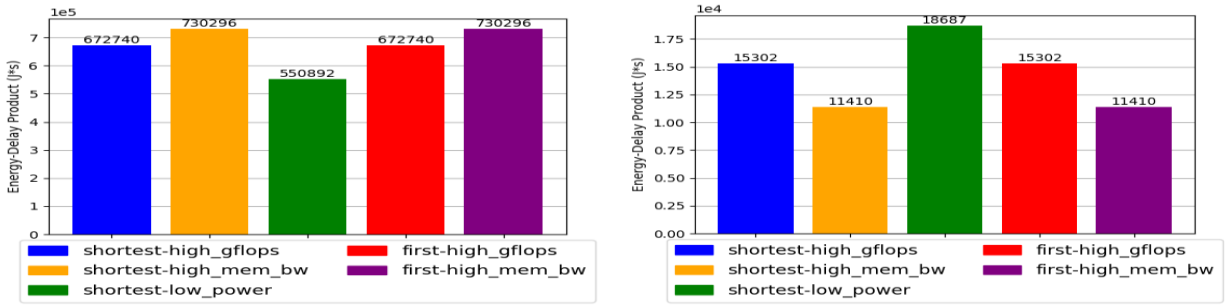
4.4.3. Entornos dinámicos: cambio de workload

Para finalizar este capítulo, a continuación se describen experimentos para comprobar la última aportación realizada en el Capítulo 3. Con esta aportación se pretende que el agente sea capaz de aprender a planificar políticas para tipos distintos de workloads. En concreto, se deciden implementar los experimentos de manera similar al caso anterior del cambio de objetivos, pero esta vez cambiando el workload. Una vez que el agente selecciona una política para un workload, se decide cambiar el workload con otro tipo de trabajos para que el agente sea capaz de seleccionar una nueva política óptima que sea capaz de optimizar el objetivo de planificación.

En este experimento se ha utilizado una plataforma utilizada en la Sección 4.2, en concreto la primera plataforma de todas con cuatro procesadores dual-core. En este caso no se utiliza el mismo workload que en los experimentos previos, sino que se crean dos workloads distintos con el mismo número de trabajos, 2 trabajos en total. Para cada workload hay un tipo de trabajo distinto, que se diferencian en dos aspectos. El primero aspecto es el tiempo requerido

para la ejecución y el segundo el ancho de banda de memoria por proceso. Para el primer workload, el tiempo requerido para la ejecución es de cinco segundos y el ancho de banda de memoria por proceso es de 24 GB/s, mientras que en el segundo workload tenemos un tiempo de cómputo de 40 segundos y un ancho de banda de memoria de 4 GB/s.

Teniendo ya dos workloads distintos es importante conocer cuál es la política óptima para cada workload. En este caso se usa el objetivo de *energy-delay product* y los resultados para cada workload se pueden observar en la Figura 4.14. Para el primer workloads tenemos la política óptima de *shortest-low_power* y para el segundo tenemos dos políticas óptimas, *shortest-high-mem_bw* y *first-high-mem_bw*. En el experimento completo se han realizando tres intercambios de workloads para comprobar de manera precisa que el agente es capaz de realizar esta operación independientemente del orden en el cual ejecutemos los workloads.



(a) *Energy-Delay Product* de las políticas clásicas del primer workload

(b) *Energy-Delay Product* de las políticas clásicas del segundo workload

Figura 4.14: Políticas Clásicas para los dos tipos de workloads

Una vez tenemos los workloads y sabemos qué comportamiento debe de tener el agente, se ejecuta el experimento con un *learning rate* de 0.005. Los resultados de este experimento son visibles en las Figuras 4.15 y 4.16. En la Figura 4.15 tenemos la selección de las políticas que realiza el agente a lo largo de la simulación. Durante los primeros 400 episodios donde se ha utilizado el primer workload, el agente decide la política correcta para este workload sobre el episodios 250. Se puede apreciar también en la Figura 4.16 como las pérdidas de la red se van reduciendo sobre el episodio 200, hasta que se llega al episodio 250 y las pérdidas alcanzan valores cercanos a cero.

En el episodio 400 se decide cambiar de workload, y durante unos pocos episodios el agente no cambia de política mientras que las pérdidas se disparan debido a este cambio en el workload. Después de estos pocos episodios el agente cambia de política en pocos episodios más y se decanta rápidamente por la política óptima para el segundo workload, bajando las pérdidas rápidamente también. En los episodios 950 y 1150 también se decide cambiar el workload, y en ambos cambios suceden situaciones equivalente a la ocurrida en el episodio 400.

Por tanto, con este ejemplo simple se consigue observar que el agente es capaz de realizar esta adaptación a los distintos tipos de workloads que se le presentan en un entorno simple. Esto aporta un primer paso para adaptar al agente a situaciones reales en donde los trabajos

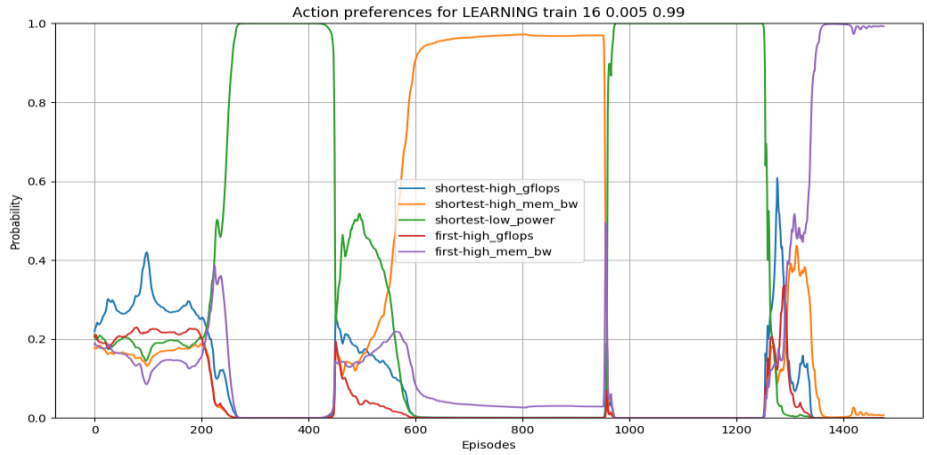


Figura 4.15: Preferencia de acciones del experimento de cambio de workload para 8 cores - 1500 episodios

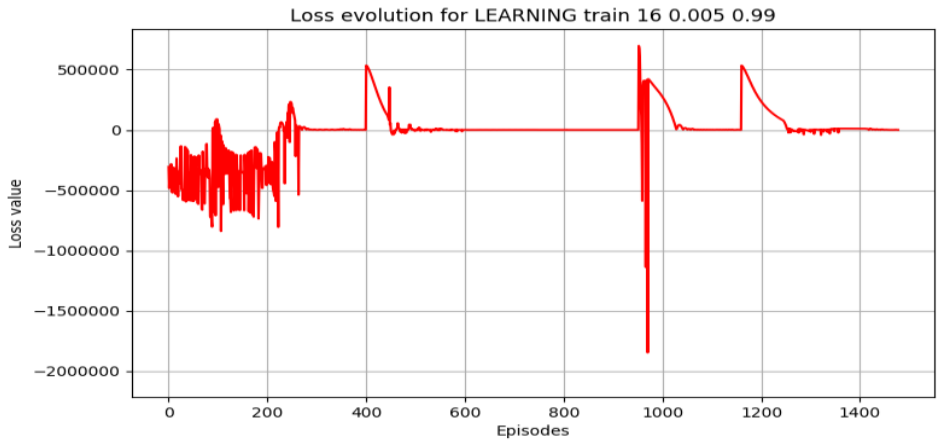
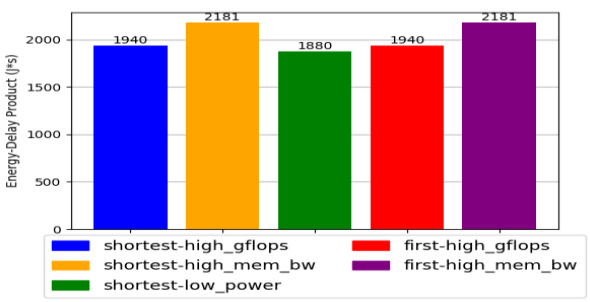


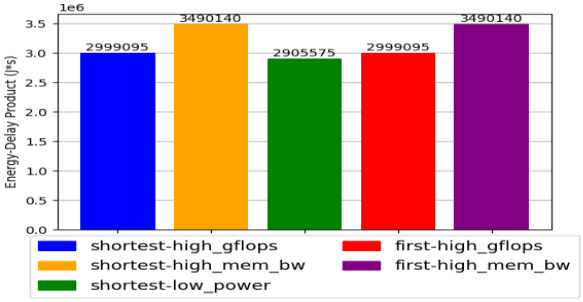
Figura 4.16: Pérdidas de la red del experimento de cambio de workload para 8 cores - 1500 episodios

llegan constantemente como sucede en los datacenters reales, el cual es el objetivo final del simulador HDeepRM.

Este ejemplo es muy sencillo por eso se ha realizado el mismo experimento con la plataforma de 2816 cores con unos workloads muy parecidos. Los únicos cambios realizados en los workloads han sido el multiplicar el número de trabajos por 64, dando un total de 128 trabajos en cada workload. Y además se ha cambiado el tiempo de ejecución de los trabajos del primer workload a un segundo. Las políticas óptimas para los workloads son las mostradas en la Figura 4.17.



(a) *Energy-Delay Product* de las políticas clásicas del primer workload



(b) *Energy-Delay Product* de las políticas clásicas del segundo workload

Figura 4.17: Políticas Clásicas para los dos tipos de workloads en el experimento con 2816 cores

En la Figura 4.17 se muestra cómo para los dos distintos workloads el agente es capaz de seleccionar la política correcta. Cabe destacar, que aunque las políticas óptimas sean las mismas, el agente tiene que volver a un estado de incertidumbre en la selección de políticas. Esto es debido a que los *rewards* de los workloads son distintos y por tanto el agente debe de volverse a entrenar para el nuevo workload.

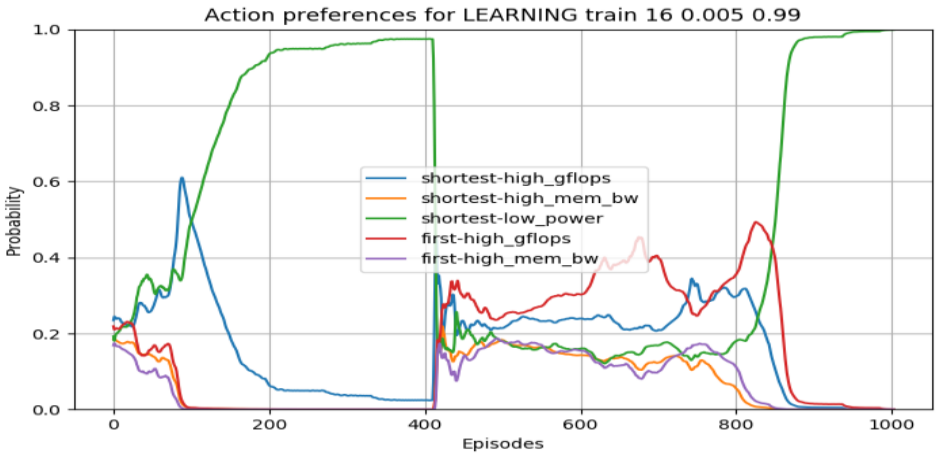


Figura 4.18: Preferencia de acciones del experimento de cambio de workload para 2816 cores - 1000 episodios

En la Figura 4.18 se puede observar como sucede el comportamiento previamente mencionado. Durante los primeros 400 episodios el el agente escoge la política correcta de *shortest-low_power*. En el momento del cambio del workload en el episodio 400 el agente detecta el cambio y vuelve a un estado de incertidumbre de las políticas. Luego cerca del episodio 900 el agente vuelve a seleccionar la política óptima para el segundo workload.

Este experimento da lugar a multitud de posibilidades para el simulador. Independientemente del escalado de la plataforma y de los workloads el agente es capaz de reaprender sus políticas en función de *reward* recibido, pudiendo así cambiar su comportamiento en función de los trabajos recibidos.

Capítulo 5

Conclusiones

En este capítulo final se destacan los objetivos conseguidos con este trabajo, y a su vez las posibles mejoras que se podrían implementar en el simulador, en particular en la mejora del planificador inteligente.

5.1. Objetivos Conseguidos

Los objetivos propuestos en este proyecto son descritos en la Sección 1.3, y cabe destacar que la mayoría de las metas propuestas han obtenido un logro positivo, como son:

- *Escalabilidad en la planificación*: actualmente el agente inteligente en HDeepRM es capaz de planificar entornos con miles de recursos y cientos/miles de trabajos. Esto es un gran logro comparándolo con el punto de partida donde solamente se podían simular un par de cores y de trabajos. Este avance fundamental dota al simulador con la capacidad de simular datacenter reales con todos sus recursos y trabajos como sucede en el experimento 4.2. Y además de simularlos se pueden aplicar las técnicas de *Deep Reinforcement Learning* en la planificación de los trabajos, convirtiéndose en uno de los primeros simuladores que tienen esta capacidad. El uso del *Deep Reinforcement Learning* en la planificación de trabajos permite el estudio del *Machine Learning* para este problema. Con este estudio se puede medir la capacidad que tienen estas técnicas de inteligencia artificial para resolver el problema de planificación de tareas en datacenters y HDeepRM, con los cambios realizados en este trabajo, se convierte en uno de los primeros simuladores que permite este estudio de manera tan compleja y con entornos heterogéneos.
- *Objetivos de planificación*: la implementación de nuevos objetivos para la planificación de trabajo descrita en la Sección 3.2 ha permitido planificar al simulador en función de distintos aspectos. Este tipo de objetivos es muy común en los datacenters reales, y

pueden cambiarse a lo largo del tiempo, muchas veces de forma periódica. Implementar distintos tipos objetivos aporta realismo al simulador y además permite estudiar nuevas situaciones de planificación con *Machine Learning*. Gracias a la implementación de dos nuevos objetivos en el simulador, se pueden priorizar las políticas en función del gasto energético o de la eficiencia energética que provocan. Estos objetivos aumentan en gran medida la calidad del simulador, debido a que el uso de la energía es un aspecto fundamental en el cual, las empresas propietarias de datacenters siempre intentan mejorar para reducir un consumo innecesario de energía que aumente sus costes y sean menos competitivas en el mercado. Además de que la eficiencia energética tiene un fuerte impacto en el medio ambiente y en problemas como el cambio climático.

- *Creación de entornos dinámicos de planificación*: en este caso se han creado experimentos que prueban la capacidad del agente en entornos donde los objetivos de planificación o el workload cambian a lo largo de la simulación. Esto permite realizar simulaciones mucho más realistas donde los trabajos no se repiten de manera infinita. Además se pone a prueba el planificador inteligente haciendo su tarea mucho más compleja, pero a la vez lo hace más adaptable a los cambios. En este aspecto el planificador inteligente es capaz de adaptarse a los cambios en el objetivo de manera correcta sin perder prácticamente nada de eficiencia. Y en cuanto al cambio de workload, se consiguen planificar experimentos tanto reducidos como con gran número de cores, lo que permite concluir que el planificador es capaz de aprender distintos tipos de workloads.

Todos estas metas consiguen el objetivo fundamental de este trabajo, mejorar las capacidades de planificación en HDeepRM con agentes inteligentes. El planificador inteligente basado en *Deep Reinforcement Learning*, actualmente es capaz de planificar en una gran cantidad de entornos distintos con resultados muy positivos y muy alentadores de las capacidades que pueden llegar a tener.

5.2. Trabajos Futuros

En esta sección se quiere destacar algunas ideas de mejora para HDeepRM en cuanto a la planificación inteligente, y qué cambios podrían implementarse para mejorar el simulador y hacerle más realista. En cuanto a la planificación inteligente se podría implementar:

- *Cambios de Workloads en la simulación*: el planificador actualmente es capaz de planificar cambios en el workload en una plataforma con un número de cores bastante grande, pero la cantidad de workloads que se pueden generar es prácticamente infinita. Por esto, una tarea a futuro sería aumentar esta capacidad del planificador realizando varios experimentos con distintos tipos de workloads interesantes para comprobar el mayor número de situaciones posibles. Este aumento de la capacidad del planificador puede se posible gracias al uso de otros modelos de redes neuronales como pueden ser las redes neuronales recurrentes o las convoluciones.

- *Simulaciones más potentes*: el límite de las plataformas mostradas en los experimentos de la Sección 4.2 era debido a la poca capacidad del computador utilizado para ejecutar la aplicación. Una buena idea sería utilizar un servidor para ejecutar la aplicación y así no tener las restricciones ni de hardware ni software que tenemos en los ordenadores personales. Con ello se conseguiría observar cuál es el límite real que podría tener la planificación inteligente. Este trabajo se empezó a realizar en el servidor del departamento, pero debido a la pandemia no se ha podido finalizar la tarea.

- *Uso de hardware dedicado*: para mejorar el rendimiento de las redes neuronales es muy normal hacer uso de GPU's. Las GPU's tienen un mayor rendimiento en comparación con las CPU's para la realización de los cálculos necesarios en las redes neuronales. Gracias a que la implementación de las redes neuronales en HDeepRM es mediante la PyTorch, esta librería permite el uso de CUDA[30] que es el lenguaje para utilizar las GPU's creadas por NVIDIA, una de las empresas más grandes en las fabricación de GPU's. En HDeepRM se podría implementar el planificador usando el lenguaje CUDA y así se aprovecharía la capacidad de cómputo de las GPU's, reduciendo el tiempo de planificación del agente.

- *Comparar eficiencia con algoritmos clásicos*: un método interesante para comprobar la eficiencia de un planificador inteligente sería compararlos con algoritmos clásicos de planificador. Los algoritmos clásicos realizan una aproximación en la selección de las políticas, por tanto no aseguran la obtención de políticas óptimas. Y en este caso es interesante comparar estos algoritmos clásicos con los inteligente para obtener una comparativa de los resultados y poder concluir si los planificadores inteligentes tienen un futuro en la solución de esta tarea.

Bibliografía

[1] Adrián Herrera Arcila et al. Hdeeprm: Deep reinforcement learning for workload management in heterogeneous clusters. 2019.

[2] Luiz André Barroso, Urs Hölzle, and Parthasarathy Ranganathan. The datacenter as a computer: Designing warehouse-scale machines. *Synthesis Lectures on Computer Architecture*, 13(3):i–189, 2018.

[3] Jason Mars, Lingjia Tang, and Robert Hundt. Heterogeneity in "homogeneous" warehouse-scale computers: A performance opportunity. *IEEE Computer Architecture Letters*, 10(2):29–32, 2011.

[4] R. Nathuji, C. Isci, and E. Gorbatov. Exploiting platform heterogeneity for power efficient data centers. In *Fourth International Conference on Autonomic Computing (ICAC'07)*, pages 5–5, 2007.

[5] Jeffrey D. Ullman. Np-complete scheduling problems. *Journal of Computer and System sciences*, 10(3):384–393, 1975.

[6] Chung Laung Liu and James W Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973.

[7] Mireille Raby and Christopher D Wickens. Strategic workload management and decision biases in aviation. *The International Journal of Aviation Psychology*, 4(3):211–240, 1994.

[8] Thomas M Mitchell et al. Machine learning, 1997.

[9] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[10] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[11] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[12] Nils J Nilsson. *Principles of artificial intelligence*. Morgan Kaufmann, 2014.

[13] Howard B Demuth, Mark H Beale, Orlando De Jess, and Martin T Hagan. *Neural network design*. Martin Hagan, 2014.

[14] Mohamad H Hassoun et al. *Fundamentals of artificial neural networks*. MIT press, 1995.

[15] Raúl Rojas. *Neural networks: a systematic introduction*. Springer Science & Business Media, 2013.

[16] Batsim Team. Batsim github repository.

[17] Steve J Chapin, Walfredo Cirne, Dror G Feitelson, James Patton Jones, Scott T Leutenegger, Uwe Schwiegelshohn, Warren Smith, and David Talby. Benchmarks and standards for the evaluation of parallel job schedulers. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 67–90. Springer, 1999.

[18] Guido Van Rossum and Fred L Drake Jr. *Python tutorial*. Centrum voor Wiskunde en Informatica Amsterdam, The Netherlands, 1995.

[19] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[20] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, pages 8024–8035. 2019.

[21] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 270–288. 2019.

[22] Henan Zhao and Rizos Sakellariou. Scheduling multiple dags onto heterogeneous systems. In *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*, pages 14–pp. IEEE, 2006.

[23] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. {GRAPHENE}: Packing and dependency-aware scheduling for data-parallel clusters. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 81–97, 2016.

[24] Yuhao Li, Dan Sun, and Benjamin C Lee. Dynamic colocation policies with reinforcement learning. *ACM Transactions on Architecture and Code Optimization (TACO)*, 17(1):1–25, 2020.

[25] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[26] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. Resource management with deep reinforcement learning. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, pages 50–56, 2016.

[27] Di Zhang, Dong Dai, Youbiao He, and Forrest Sheng Bao. Rlscheduler: Learn to schedule hpc batch jobs using deep reinforcement learning. *arXiv preprint arXiv:1910.08925*, 2019.

[28] J Emeras. Parallel workloads archive university of luxemburg gaia cluster, 2016.

[29] Dror G Feitelson. Parallel workload archive. *http://www. cs. huji. ac. il/labs/parallel/workload*, 2007.

[30] NVIDIA Corporation. *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*. NVIDIA Corporation, 2007.