# 

Trabajo de Fin de Master

# **Autoescalado horizontal en Kubernetes con Aprendizaje por Refuerzo**

Marcial Lalanda González-Bueno

2021

Tutor: José Ignacio Olmeda

Contenido

[**Autoescalado horizontal en Kubernetes con Aprendizaje por Refuerzo** 1](#_Toc84151828)

[Introducción 4](#_Toc84151829)

[Planteamiento y Entorno de trabajo 5](#_Toc84151830)

[¿Qué es Kubernetes? 5](#_Toc84151831)

[Pod 6](#_Toc84151832)

[Deployment 6](#_Toc84151833)

[Replica Set 6](#_Toc84151834)

[Service 7](#_Toc84151835)

[Namespace 7](#_Toc84151836)

[Horizontal Pod Autoscaling 7](#_Toc84151837)

[Monitorización 9](#_Toc84151838)

[Estado 9](#_Toc84151839)

[Acción 10](#_Toc84151840)

[¿Qué es el Aprendizaje por Refuerzo? 10](#_Toc84151841)

[Q-Learning 12](#_Toc84151842)

[Deep Q Network (DQN) 15](#_Toc84151843)

[Diseño e Implementación 18](#_Toc84151844)

[Entorno 18](#_Toc84151845)

[Agente 23](#_Toc84151846)

[Entrenamiento 29](#_Toc84151847)

[Ejecución 31](#_Toc84151848)

[Resultados 34](#_Toc84151849)

[Primer ciclo. Prueba HPA 35](#_Toc84151850)

[Primer ciclo. Prueba Agente con aprendizaje reforzado 36](#_Toc84151851)

[Segundo ciclo. Prueba HPA 37](#_Toc84151852)

[Segundo ciclo. Prueba Agente con aprendizaje reforzado 38](#_Toc84151853)

[Conclusiones 39](#_Toc84151854)

[Posibles nuevas líneas de trabajo 40](#_Toc84151855)

[Bibliografía 42](#_Toc84151856)

[Otros Recursos 43](#_Toc84151857)

[Repositorio de Código 43](#_Toc84151858)

[Cliente Python de Kubernetes 43](#_Toc84151859)

#### Índice de Figuras

[Ilustración 1: Despliegue en contenedores 4](#_Toc83492768)

[Ilustración 2: Arquitectura de Kubernetes 5](#_Toc83492769)

[Ilustración 3: Kubernetes Dashboard 8](#_Toc83492770)

[Ilustración 4: Elementos de un sistema de aprendizaje por refuerzo 10](#_Toc83492771)

[Ilustración 5: Arquitectura del Sistema propuesto 11](#_Toc83492772)

[Ilustración 6: Algoritmo Q-Learning 12](#_Toc83492773)

[Ilustración 7: Función Q implementada como tabla 12](#_Toc83492774)

[Ilustración 8: Deep Q Network (DQN) 15](#_Toc83492775)

[Ilustración 9: Agente DQN con Replay Buffer 15](#_Toc83492776)

[Ilustración 10: Vector estado con valores continuos 17](#_Toc83492777)

[Ilustración 11: Vector estado con valores discretizados 18](#_Toc83492778)

[Ilustración 12: Conversión estado a clave para tabla de recompensas 21](#_Toc83492779)

[Ilustración 13: Metáfora de la inestabilidad que se introduce si la red objetivo es la misma que la principal 26](#_Toc83492780)

[Ilustración 14: Target Network y Main Network 27](#_Toc83492781)

[Ilustración 15: Patrón de carga para la prueba 33](#_Toc83492782)

## Introducción

Con el advenimiento de los servicios en Cloud proporcionados por las grandes compañías tecnológicas (AWS, Azure, GCP) cada vez son más las empresas o particulares que deciden utilizarlos para desplegar sus aplicaciones. A medida que se comienzan a utilizar de forma más generalizada e intensiva estas infraestructuras surge inevitablemente la preocupación por hacer el uso más racional de las mismas dado los costes que suponen. Estos se basan normalmente en la cantidad de recursos (número de máquinas y tipo de las mismas) instanciadas. La gestión manual del escalado horizontal en estas plataformas es siempre susceptible de ser mejorada por sistemas automáticos que optimicen el rendimiento de las aplicaciones y el coste. De hecho las plataformas disponen normalmente de servicios que permiten a los clientes configurar el autoescalado de su infraestructura mediante sistemas de reglas basadas en umbrales para distintos parámetros. Sin embargo estos sistemas pueden ser complicados de configurar correctamente y presentar limitaciones a la hora de poder optimizar los mencionados umbrales. En el presente trabajo exploramos la posibilidad de aplicar alguno de los algoritmos existentes de Aprendizaje por Refuerzo al problema del auto-escalado horizontal para comprobar cómo se comportan.

En nuestro caso no lo haremos directamente sobre las plataformas Cloud mencionadas anteriormente, debido a los costes en los que podríamos incurrir, sino sobre un pequeño laboratorio desplegado en un equipo portátil consistente en un cluster de Kubernetes en el que estableceremos una comparación de rendimiento entre el sistema nativo de autescalado, denominado HPA (Horizontal Pod Autoscaler) y una solución que aplique un algoritmo de aprendizaje por refuerzo.

El objetivo es, por tanto, evaluar si alguno de estos algoritmos puede aportar alguna ventaja significativa, y en qué condiciones sobre el mencionado sistema propio de autoescalado horizontal de Kubernetes.

## Planteamiento y Entorno de trabajo

La pieza fundamental de nuestro laboratorio es Kubernetes. Afortunadamente podemos instalarlo de forma sencilla en nuestro ordenador personal gracias a herramientas cómo Minikube que, si bien no permite explotar todas las capacidades disponibles en un cluster de Kubernetes, sí permite trabajar con la mayoría de ellas y, desde luego, nos proporciona un entorno adecuado a nuestras necesidades [[1]](#footnote-1).

### ¿Qué es Kubernetes?

Tal y como se define en su propio sitio web “Kubernetes es una plataforma portable y extensible de código abierto para administrar cargas de trabajo y servicios” [[2]](#footnote-2). Para nuestros propósitos podemos pensar en ella como una plataforma de contenedores. En cada uno de estos contenedores se desplegará una aplicación y podremos tener varios contenedores con la misma aplicación desplegada, tal y cómo se muestra en la imagen siguiente:

Interfaz de usuario gráfica

Descripción generada automáticamente

Ilustración 1: Despliegue en contenedores [[3]](#footnote-3)

Un esquema que muestra con algo más de detalle la arquitectura de kubernetes sería el siguiente:

Diagrama

Descripción generada automáticamente

Ilustración 2: Arquitectura de Kubernetes [[4]](#footnote-4)

### Pod

Un Pod representa en Kubernetes una unidad de ejecución de un proceso. Puede contener uno o más contenedores en los que, a su vez, se ejecuta, por regla general, una única aplicación.

Los pods tienen una vida efímera y, en el caso de que presenten fallos, el sistema puede eliminarlos y crear otros nuevos que los sustituyan. De esa forma se mantiene el servicio en alta disponibilidad.

### Deployment

El Deployment es un objeto de Kubernetes que permite declarar el estado y las características deseadas para el despliegue de una determinada aplicación. En él se especifica información cómo pueda ser, por ejemplo, la imagen del contenedor con la aplicación a desplegar, el número de Pods que queremos se levanten, los selectores o etiquetas con los que lo identificaremos, puertos etc.

### Replica Set

Dentro del Deployment el “Replica Set” en concreto es el elemento encargado de velar por la alta disponibilidad del sistema. Su labor es, por tanto, mantener constante el número de pods activos que se haya indicado en el Deployment.

### Service

Debido la mencionada naturaleza efímera de los Pods, Kubernetes proporciona una fachada que no adolece de dicho problema y a la que se pueden dirigir las peticiones a los diferentes recursos de la aplicación. El servicio redirigirá a su vez dichas peticiones a cualquiera de los Pods que estén disponibles en ese momento.

### Namespace

Los namespaces permiten establecer clusters virtuales dentro de un cluster físico para aislar unos recursos de otros, por ejemplo, para separar entornos de Desarrollo, Test y Producción, o aplicaciones de distintos equipos de trabajo. En nuestro caso crearemos un Namespace para desplegar en él los contenedores con los que vamos a establecer nuestra comparativa.

### Horizontal Pod Autoscaling

Kubernetes tiene su propio sistema para aumentar o disminuir el número de pods de un deployment de acuerdo al número de peticiones que recibe o a la carga de trabajo a la que está sometido. Lo hace mediante un tipo de recurso denominado HPA (Horizontal Pod Autoscaler) que consulta el API (Application Program Interface) del servidor de Métricas (que es necesario activar en el cluster) para decidir, en función de las reglas que se le proporcionen, si es necesario incrementar o disminuir el número de pods.

Para nuestra comparación utilizaremos cómo referencia el ejemplo descrito en la web de documentación de Kubernetes [[5]](#footnote-5) (<https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale-walkthrough/>). Se trata de una página web desplegada en un servidor Apache que realiza un elevado número de operaciones con el objetivo de aumentar significativamente el consumo de CPU. Tal y cómo se indica en la web, el fichero que define el contenedor (Dockerfile) define una página php:



Que simplemente realiza gran número de cálculos para hacer un uso intensivo de la CPU



Crearemos los pods de Kubernetes mediante el fichero yaml ([php-apache.yml](https://github.com/mlalandag/HPAwDRL/blob/main/configuration/php-apache.yml)) en el que declaramos todos los recursos que vamos a utilizar, el Namespace, el Deployment propiamente dicho (en el que se especifica el tipo de contenedor que desplegaremos en los pods), el Servicio y el HorizontalPodAutoscaler. Previamente tendremos que haber habilitado el plugin con el servidor de métricas, para que el componente HPA pueda funcionar. El Deployment se denominará “php-apache” y los nombres de los pods comenzarán también con la misma cadena de caracteres. También será el nombre de nuestro Namespace.

El HPA podemos crearlo también ejecutando el siguiente comando en un terminal:



En este caso el número máximo de pods será de 5 y el mínimo de 1, siendo el umbral máximo de uso de CPU que deseamos de un 50%. Si se supera en los pods, el HPA creará otros nuevos para que descienda.

Cuando queramos aumentar el porcentaje de CPU utilizado, lo haremos ejecutando el siguiente comando:



Con el mismo crearemos un nuevo Pod en el que se ejecutará continuamente un bucle de llamadas al servidor Apache.

### Monitorización

Para poder consultar el estado del cluster de forma visual activaremos el Dashboard que nos proporciona Minikube. En la pantalla podremos ver el nivel de consumo de CPU y Memoria además de los distintos Pods que están funcionando en cada momento.

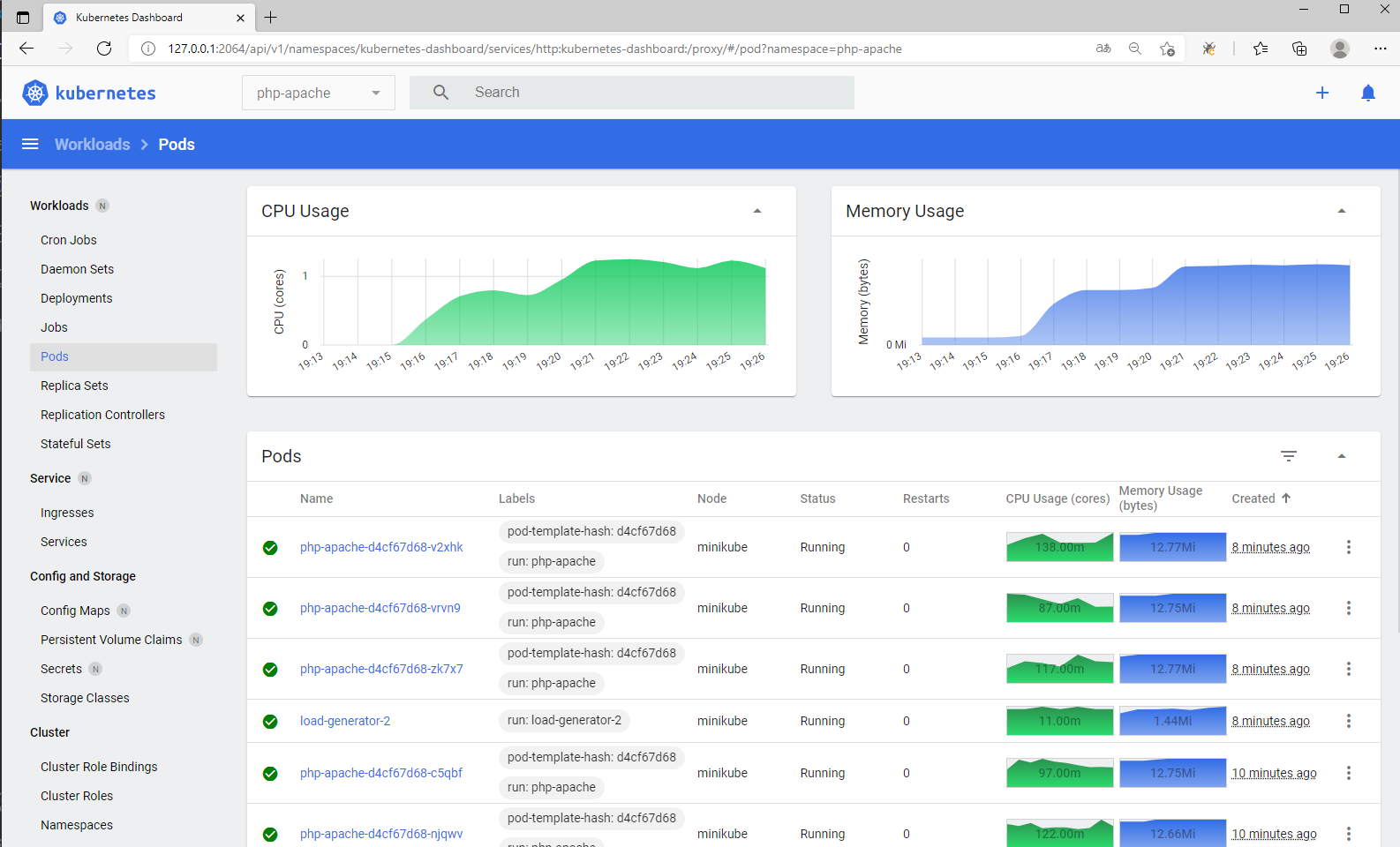


Ilustración 3: Kubernetes Dashboard

También, durante los procesos de ejecución del entrenamiento del agente, y del modelo generaremos gráficas con información similar que nos permita ver la evolución del número de pods frente al uso de CPU de forma más precisa.

### Estado

Además de poder consultar el estado de forma visual con el Dashboard necesitaremos para poder alimentar nuestro algoritmo de aprendizaje por refuerzo una forma programática de obtenerlo. Utilizaremos para ello el Cliente Python de Kubernetes [[6]](#footnote-6).

Con apenas cuatro líneas de código podemos obtener gracias a esta herramienta el consumo de CPU y Memoria de cada uno de los Pods desplegados, en este caso, en el namespace “php-apache”



### Acción

Para poder aplicar también de forma programática sobre el cluster de Kubernetes el comando para aumentar o disminuir el número de Pods haremos uso de un script Python que utilizará el paquete “os” y su método “system”.



El script recibirá por parámetro el número de Pods que decida el algoritmo de aprendizaje por refuerzo y se lo comunicará a kubernetes para que éste actúe en consecuencia.

### ¿Qué es el Aprendizaje por Refuerzo?

Uno de los rasgos diferenciales de los algoritmos de aprendizaje por refuerzo es que son capaces, hasta cierto punto, de aprender por sí mismos de la información que reciben sin que sea necesario “enseñarles” directamente cuales son las respuestas correctas a determinado problema y cuáles no. Esta característica los diferencia de los algoritmos de Aprendizaje Supervisado si bien no los significa demasiado frente a otros algoritmos del campo denominado Aprendizaje No Supervisado, cuyo ejemplo más representativo es el “Clustering”. La diferencia con respecto a este último, cuyo propósito principal es encontrar patrones en un conjunto de datos no etiquetados, es que el aprendizaje por refuerzo tiene una cierta componente de orientación hacia la consecución de un objetivo concreto, formulado mediante una función que trata de representar lo que se considera el éxito o fracaso a la hora de resolver la tarea o problema.

El Aprendizaje Supervisado implica una suerte de relación Profesor-Alumno en el que éste último, el algoritmo, puede llegar a imitar perfectamente a su profesor, el conjunto de datos etiquetados [[7]](#footnote-7). Éstos, sin embargo, pueden estar limitados en cuanto a la cantidad de la que podemos disponer además de que, en ocasiones, son muy complicados de obtener e incluso el esfuerzo para ello puede no compensar el resultado posterior.

En el caso de tareas de Control o decisión frente a tareas de predicción o de clasificación (supervisadas) en muchas ocasiones el entorno no está controlado y produce sus propios datos además de poder tener una componente probabilística. Es en estos escenarios donde el Aprendizaje por Refuerzo sobresale por sus características.

El siguiente diagrama muestra los elementos esenciales que intervienen en un sistema de Aprendizaje por Refuerzo.



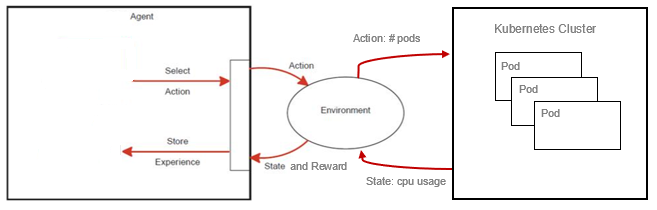
Ilustración 4: Elementos de un sistema de aprendizaje por refuerzo [[8]](#footnote-8)

El agente realiza acciones (del conjunto de acciones posibles) sobre el entorno en el que se desenvuelve. En respuesta a cada una de ellas, el entorno realiza una transición a un nuevo estado (o continúa en el anterior) y se lo comunica al agente, junto con un valor que representa una “recompensa”. Ésta podrá ser positiva o negativa en función del objetivo que se le haya definido al agente. La misión de éste es, por tanto, maximizar esta recompensa en la medida de los posible.

De lo anterior se deduce que es fundamental definir adecuadamente la función que determina las recompensas para representar correctamente el objetivo que se persigue.

Por otro lado, el agente implementa el algoritmo que, recibiendo los estados y las recompensas y guardando esta información bien sea en forma de algún tipo de colección o tabla, o mediante alguna función de aproximación, trata de escoger las mejores acciones en cada momento, entendiendo por “mejores” las que le permiten acumular una mayor recompensa en el largo plazo.

A la hora de aplicarlo a nuestro sistema el esquema quedaría de la siguiente manera:



Reinforcemement Learning Algorithm

Ilustración 5: Arquitectura del Sistema propuesto

En este caso el entorno constituye, por un lado, un proxy o fachada del Cluster de Kubernetes, actuando cómo intermediando y comunicándole las acciones deseadas y recuperando su estado y, por otro, se encarga de elaborar y calcular la recompensa que llevará asociada la acción que el agente decidió en función del estado.

### Q-Learning

La siguiente imagen muestra una versión del algoritmo que utilizaremos en nuestra comparativa, Q-learning. Partiremos de la misma para ilustrar los elementos básicos que intervienen en el mismo y que encontraremos en la mayoría de algoritmos de Aprendizaje por Refuerzo en mayor o menor medida y con distintas variantes.

****

Ilustración 6: Algoritmo Q-Learning **[[9]](#footnote-9)**

#### Function Q

La función Q, también llamada la función valor estado-acción, devuelve la recompensa que recibiría un agente que partiendo de un estado ***s*** realizara la acción ***a*** siguiendo la póliza ***π.*** Suvalor se denota por *Q(s,a)* y se le denomina valor Q. Su expresión es [[10]](#footnote-10):

En un entorno cuyos tamaños de espacios de estados (número de posibles estados) y acciones no sea excesivamente elevado la función Q puede tomar la forma de una tabla en la que se almacena el valor para cada pareja Estado-Acción.

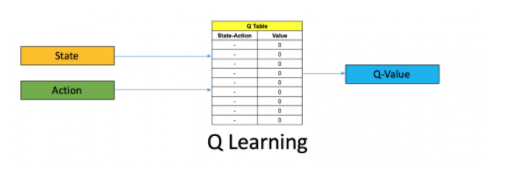


Ilustración 7: Función Q implementada como tabla[[11]](#footnote-11)

#### Póliza

La póliza, representada por el símbolo ***π,*** es la función que define el comportamiento del agente en el entorno y le indica que qué acción debe ejecutar en cada estado. Cuando el agente interactúa por primera vez con el entorno la póliza se inicializa de forma aleatoria y también lo serán, por lo tanto, las acciones del agente. A medida que se producen más interacciones el agente empezará a poder discernir entre acciones “buenas” y acciones “malas” en función de las recompensas que va recibiendo. Ello le permitirá ir aprendiendo y mejorar la póliza. La mejor póliza para un determinado entorno se denomina la póliza óptima y será la que proporcione la mayor recompensa acumulada para el agente.

#### Episodios

Cómo se puede observar en la imagen que representa el algoritmo de Q-Learning, éste contiene dos bucles anidados, uno por cada episodio y otro por cada paso (step) dentro de cada episodio.

Cada episodio representa las interacciones del agente con el entorno desde un estado inicial hasta el estado final. Cuando se llega hasta este último finaliza el episodio. Cada paso dentro del episodio se realiza al efectuar el agente una acción y producirse una transición a otro estado. Otro término que se utiliza de forma intercambiable con el término episodio es “trayectoria” y se denota con el símbolo ***ꞇ***.

Al iniciarse el algoritmo se inicializa la función Q con valores aleatorios asignando el valor 0 al estado final. Antes de comenzar cada episodio se coloca el entorno en el estado inicial y, a continuación, se comienza el proceso eligiendo una acción partiendo de dicho estado en función del valor de Q. En este momento se utiliza la póliza, que en este caso es del tipo *ε-greedy, y* que consiste en elegir la acción que proporciona el mayor valor de Q la mayoría de las veces excepto en algunas ocasiones que se elige la acción al azar, esta proporción se determina dándole un valor pequeño a ε, que determinará con qué probabilidad se efectuarán esas acciones aleatorias. *ε-greedy* permitirá al algoritmo realizar tareas de “Exploración” de estados que de otra forma no serían visitados.

Tras efectuar la acción se observan la recompensa y nuevo estado del entorno y se procede a actualizar el valor de la función Q para el estado previo y la acción ejecutada tal y cómo se indica en la siguiente expresión [[12]](#footnote-12):

Dicha expresión es la regla de actualización y, cómo vemos añade al propio *Q(s,a)* un término en el que intervienen varios factores. Por un lado, ***α,*** que es la tasa de aprendizaje o “learning rate” y que permite aumentar o disminuir el efecto del término entre paréntesis en el aprendizaje. Una tasa pequeña hará que el aprendizaje sea más lento pero una tasa demasiado grande puede hacer que el aprendizaje sea inestable y que, incluso, diverja del objetivo esperado.

***r*** es la recompensa que se ha obtenido del entorno al realizar la acción a en el estado s.

***γ*** (gamma) es el factor de descuento y adopta valores entre 0 y 1. Determina cuanta importancia le damos a valores de la función Q futuros, normalmente es un valor menor (pero próximo) a 1 que reduce el peso que se le da a valores aún por venir dentro del episodio.

Y, por último, tenemos el término:

Que indica que debemos tomar de entre todos los valores de la función Q para el estado s’, independientemente de la acción a’, el máximo valor.

De esta forma el agente irá episodio tras episodio actualizando los valores de Q en función de las recompensas obtenidas e irá mejorando paulatinamente su comportamiento a la hora de elegir las mejores acciones.

### Deep Q Network (DQN)

En el algoritmo visto en el punto anterior debemos ir almacenando los distintos valores de Q en una tabla en la que, por cada fila, tendremos el estado, la acción y el valor que hemos otorgado a dicha combinación. En el caso de que nuestro entorno pueda generar un gran número de estas combinaciones o si el espacio de estados es continuo (podría tomar prácticamente infinitos valores) tener que almacenar toda esa información se convierte en un problema o simplemente es impracticable.

Para solucionar este problema, en lugar de guardar los valores de esta forma lo que podemos hacer es utilizar una función de aproximación que nos los proporcione. Esta función puede implementarse con una Red Neuronal Artificial, de forma que al alimentar la misma con el estado del entorno ésta nos retornará los valores Q de todas las acciones posibles para dicho estado.

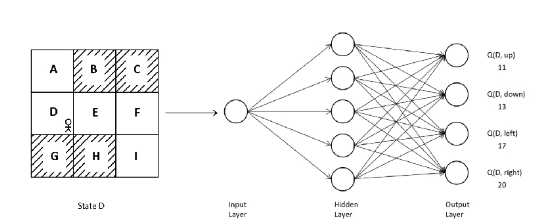


Ilustración 8: Deep Q Network (DQN) [[13]](#footnote-13)

La red neuronal utilizada se denomina “Q network” y, si estamos utilizando una red neuronal profunda tendremos una **Deep Q Network (DQN)**

Si utilizamos el símbolo *θ* para denotar los parámetros de la red neuronal, expresaremos la función Q que estamos aproximando cómo

En nuestro agente haremos uso de este algoritmo por lo que, visto lo anterior, podemos representarlo mediante la siguiente figura:

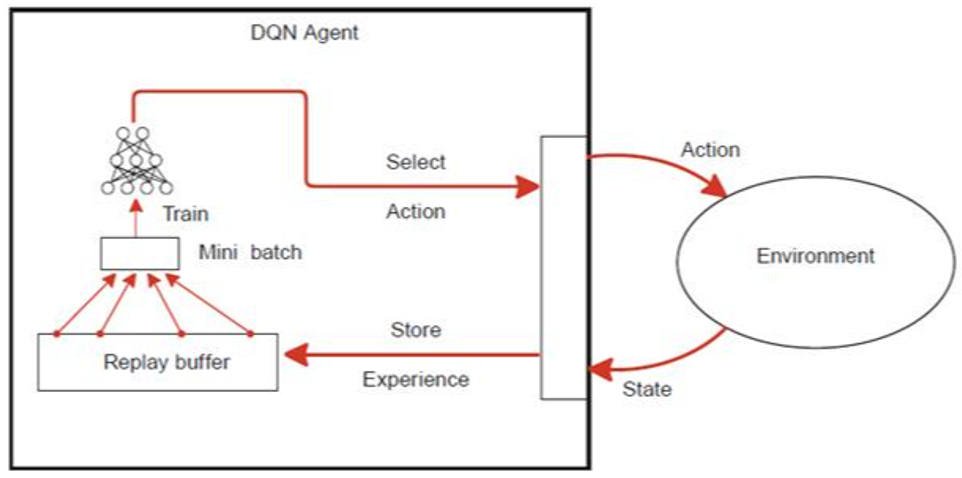


Ilustración 9: Agente DQN con Replay Buffer[[14]](#footnote-14)

En la que quedan de manifiesto la utilización de la red neuronal para implementar la función Q(s, a), la existencia del proceso de entrenamiento de la misma y otro elemento (del que hablaremos en una sección posterior) que consiste en una forma de almacenamiento de las experiencias que va teniendo el agente al interactuar con el entorno y que nos ayudará precisamente a mejorar ese entrenamiento.

## Diseño e Implementación

En esta sección describiremos la implementación del algoritmo DQN que vamos a utilizar en nuestra comparativa. El código del proyecto se puede encontrar en el siguiente repositorio de GitHub:

<https://github.com/mlalandag/HPAwDRL>

Como veíamos en la sección dedicada a exponer los rasgos fundamentales de los algoritmos de aprendizaje por refuerzo, los dos elementos fundamentales son el Entorno y el Agente. El primero refleja el estado del sistema y, según los casos, las posibles transiciones de un estado a otro. También es el encargado de proporcionar al Agente las recompensas o penalizaciones consecuencia de las acciones realizadas por este sobre el sistema.

### Entorno

#### Estados

Lo primero que vamos a caracterizar es el conjunto de valores que determinarán el estado del sistema. Dado que lo que nos preocupa en esta comparativa es el rendimiento que nos ofrece Kubernetes en función de la carga en forma de peticiones que web que está soportando, partiremos de un vector que nos refleje el número de Pods que están funcionando en cada momento seguido del consumo de CPU en cada uno de ellos. En nuestra comparativa permitiremos un máximo de 5 pods levantados simultáneamente para reducir complejidad y coste de entrenamiento pero es un parámetro que podrá ser modificado por configuración. Por tanto, podemos ilustrar la forma que tendrá nuestro vector estado con el siguiente ejemplo:

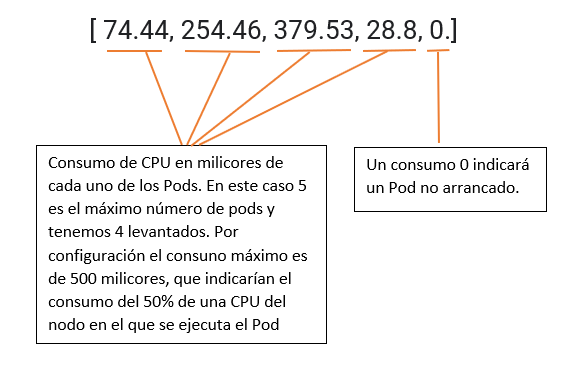


Ilustración 10: Vector estado con valores continuos

Para permitir un aprendizaje más acelerado (si bien a costa de una menor precisión) podemos discretizar el estado, de forma que asignaremos un valor fijo a cada pod en función de intervalos de consumo de CPU. Podemos hacerlo mediante la aplicación del siguiente código:



Que asigna el valor 3 a un pod con alto consumo de CPU, 2 al de consumo medio y 1 al de bajo (Los umbrales han sido elegidos teniendo en cuenta que en el “deployment” de Kubernetes hemos indicado un uso de CPU máximo por pod de 500 milicores). Por tanto, el anterior ejemplo de vector estado quedaría:



Ilustración 11: Vector estado con valores discretizados

En el proyecto hay una clase Python dedicada a implementar el Entorno, con el nombre “K8Senvironment” y que contiene los métodos necesarios para realizar las distintas operaciones que se necesitan. Una de ellas es precisamente obtener el estado del Cluster de Kubernetes y el método que la implementa contiene el siguiente código:



El objeto “resource” se obtiene haciendo uso del “Cliente Python” mencionado en una sección anterior. Para posibilitar el uso del mismo lo importamos del módulo apropiado y lo instanciamos en el método “\_init\_” de la clase que representa al Entorno:



Tal y cómo se puede observar en el código hemos solicitado al API la información de los Pods levantados en el “namespace” denominado “php-apache”. A continuación, para cada Pod leemos los parámetros de uso de CPU y, finalmente, construimos el vector estado con los datos del número total de Pods y el uso de CPU de cada uno de ellos en milicores (dividimos por 1000000 porque la información obtenida del API viene en nanocores).

#### Acciones

Las acciones que ejecutará el Agente sobre el entorno consistirán en indicarle a éste el número de Pods que se requieren en cada momento en función del estado del mismo. Por ejemplo, el Agente le indicará al Entorno que hay que tener funcionando 3 Pods en el Cluster y el Entorno se encargará de transmitir a Kubernetes la orden para que así sea. Además, una vez ejecutada la misma y esperado un tiempo prudencial para que a Kubernetes le dé tiempo a alcanzar el número de Réplicas/Pods deseado se hará una nueva lectura del estado y el cálculo de la recompensa. El método encargado de aplicar las acciones indicadas por el Agente es el siguiente:



El método recibe la acción, el número de pods deseado, y lo primero que se hace es una nueva lectura del estado actual para, a continuación, mediante otro método, “set\_replicas” transmitir a Kubernetes dicho comando.



Como vemos lo hace haciendo uso del módulo “os” de Python.

A continuación, se esperan unos segundos para permitir a Kubernetes establecer el nuevo estado y se vuelve a realizar una lectura para recuperar el mismo mediante un nuevo método que se describirá en el apartado siguiente. Por último, se llama a la función para calcular la recompensa que se ha obtenido al aplicar al estado anterior la acción determinada por el Agente y se devuelve este dato junto con la última lectura del estado.

#### Recompensas

La otra función clave del Entorno es, como hemos indicado, la que calcula la recompensa en función del estado y la acción efectuada. Para implementarla haremos uso de una tabla en la que cómo clave utilizaremos una lista con el número de pods que consideramos hacen un uso bajo de CPU, el número de los que hacen un uso medio, el número de los que hacen un uso intensivo y, finalmente, la acción decidida por el agente para ese estado. Por ejemplo, para el vector estado visto con anterioridad:

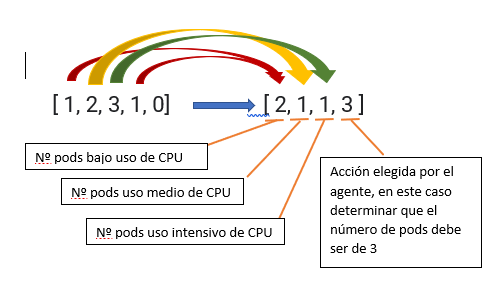


Ilustración 12: Conversión estado a clave para tabla de recompensas

Dado que hay dos pods con bajo consumo de CPU (los que tienen el valor 1), 1 con uso medio (valor 2) y otro con uso intensivo (el de valor 3) y que, a la vista del estado, el agente ha decidido levantar tres pods nuestra clave quedaría tal y cómo vemos en la figura anterior

Con esta clave acudiremos a la tabla de recompensas para recuperar el valor de la recompensa asociada. Dicha tabla tendrá la forma:



Como vemos, en nuestro caso la recompensa será de 5 puntos. Para otras claves vemos que podemos tener también penalizaciones en la forma de puntuaciones negativas si interpretamos que la decisión tomada por el agente ha sido mala.

El problema con esta aproximación es que tendremos que tener una entrada en la tabla por cada situación posible y en nuestro caso, con un máximo de 5 pods, no son muchas pero, a medida que ampliemos ese límite, pueden llegar a suponer un número complicado de manejar.

La forma final de la función de obtención de la recompensa es:



### Agente

El agente es la representación del algoritmo de aprendizaje por refuerzo propiamente dicho y el responsable de elegir la mejor acción dado un determinado estado del entorno. En nuestro sistema lo implementaremos con una clase llamada “DQNAgent”. Las propiedades que necesita para poder realizar su labor las definimos en su método “\_init\_” y son las siguientes:



Tanto alpha y gamma, cómo épsilon fueron comentados previamente en la sección en la que se describía el algoritmo de Q-Learning. Representan respectivamente, la tasa de aprendizaje o “learning rate” (**α**), el factor de descuento (**γ**) y el parámetro **ε** que necesita la póliza durante el entrenamiento para poder aplicar *ε-greedy* en su toma de decisiones y permitir al agente “explorar” nuevos estados en lugar de “explotar” siempre los que más recompensa le proporcionan. En nuestro caso añadimos otros dos parámetros, el “min\_epsilon” y el “epsilon\_decay” que nos permitirán modular el valor de épsilon durante los periodos de entrenamiento, comenzando con un **ε** grande en las primeras fases del mismo de forma que haya mayor número de acciones elegidas al azar y que, paulatinamente, se vaya reduciendo, permitiendo al agente elegir las acciones que sabe le proporcionan mayores recompensas.

#### Modelo

El corazón del agente lo constituye la función que decide cada acción y que, cómo vimos en la descripción del algoritmo DQN, está implementada mediante una red neuronal. En nuestro caso hemos utilizado el módulo Keras de Tensorflow para darle forma y lo hacemos mediante el método:



Cómo se puede ver en el código, el modelo tiene varias campas “densamente conectadas” (Dense), es decir, en las que cada neurona de una capa está conectada a todas las de la capa siguiente. La primera capa recibe el estado proporcionado por el Entorno y, por ello, definimos en la misma un “input\_shape” que tiene el mismo número de elementos que el vector estado descrito anteriormente. La última capa tiene tantas salidas cómo defina el espacio de acciones del agente. En nuestro caso hemos optado por que este espacio de acciones tenga las mismas salidas que el máximo número de pods que podemos instanciar, de forma que la posición del valor máximo represente el número de pods que el agente interpretará cómo el más óptimo.

Finalmente, se le indica a la red neuronal los parámetros de la función de coste (Mean Squared Error), el tipo de optimizador (Adam) y la métrica para evaluar el rendimiento de la red.

#### Replay Buffer

Otro de los elementos fundamentales de este algoritmo es el “replay buffer” que consiste básicamente en un almacén en el que vamos depositando registros con la información que el agente va recopilando durante su proceso de entrenamiento. Esta información la constituyen el estado, la acción que elige el agente en función de este, la recompensa obtenida al efectuar la acción y el nuevo estado del sistema tras la misma. El agente tiene una función para almacenar dicho estado tras cada uno de las transiciones que se produzcan en el sistema.



El sentido del replay buffer es evitar, durante el entrenamiento del agente, que los estados que este se vaya encontrando estén demasiado correlacionados entre sí, debido a qué no cambien mucho de una transición a la siguiente. Lo que se hace entonces es, cuando el buffer tiene un número suficiente de registros, obtener una muestra con un determinado tamaño (batch\_size), de forma que no se tomen registros consecutivos, y realizar un paso de entrenamiento con cada uno de ellos.

#### Target Network

Cómo se puede observar en la función de inicio de nuestro agente, se construyen dos redes neuronales, la Target (objetivo) y la Main (principal). La explicación la encontramos en la naturaleza del proceso de entrenamiento que podemos ver a continuación:



En cada paso de entrenamiento se toma una muestra de vectores y para cada uno de ellos calculamos el nuevo valor Q según la fórmula ya vista anteriormente:

A continuación, se realiza un paso entrenamiento de la red con los nuevos Q values. Dado que, en cada uno de los pasos, estamos modificando la red principal, si utilizáramos esta misma red para obtener los valores objetivo (target) estaríamos introduciendo inestabilidad en el sistema ya que el objetivo se estará “moviendo” a la vez que intentamos alcanzarlo. Por utilizar una metáfora gráfica, sería una situación similar a la siguiente:

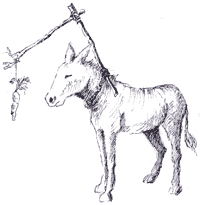


Ilustración 13: Metáfora de la inestabilidad que se introduce si la red objetivo es la misma que la principal

En la que el burro, nuestra red principal, no lograría alcanzar la zanahoria, el valor objetivo, porque se está moviendo con él. Es por ello que se utiliza una segunda red, la target network, que es idéntica a la principal, pero cuyos parámetros sólo se modifican cada determinado número de pasos copiándolos de la red principal. De esta forma la red objetivo permanece estable durante el número de pasos elegido y, cada vez que se repita ese número de pasos se actualiza de la forma siguiente:



Esquemáticamente podemos verlo resumido en la siguiente figura:

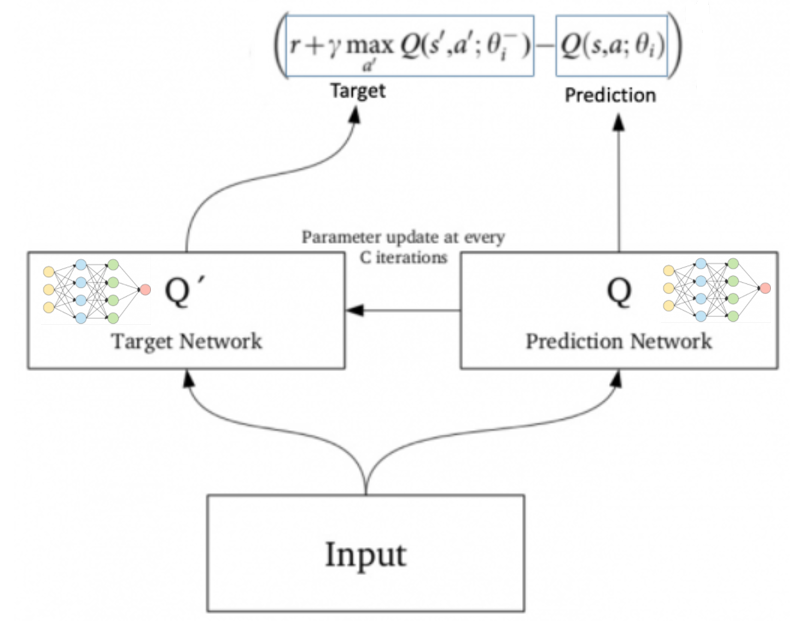


Ilustración 14: Target Network y Main Network[[15]](#footnote-15)

#### Acción

Por último, tenemos el método que utiliza el agente para elegir su acción y que vemos implementado en el siguiente fragmento de código:



En este método se comprueba en primer lugar si estamos en modo entrenamiento o no, para, en dicho caso poder realizar “exploración” eligiendo acciones al azar aplicando *ε-greedy*. El valor de ε irá decayendo cada vez que se ejecute el método hasta alcanzar un valor mínimo.

En el caso de que, por probabilidad, o porque no estemos ya en modo de entrenamiento (producción), no se produzca la elección aleatoria de la acción, esta se obtendrá a partir de nuestra red neuronal principal, y representará el número de pods que debe tener el cluster de Kubernetes para cubrir la demanda de peticiones que se pueda estar produciendo en ese momento.

### Entrenamiento

Para realizar el entrenamiento del agente configuraremos el sistema para hacerlo funcionar durante un cierto número de episodios con un determinado número de pasos cada uno. Simultáneamente sometemos al cluster de Kubernetes ha diferentes cargas de peticiones para que el agente pueda visitar el mayor número de estados posibles. Previamente tendremos que haber desactivado el HPA (HorizontalPodAutoscaler) de Kubernetes para que no interfiera con nuestro proceso eliminándolo con el comando:



Empezamos creando en primer lugar el entorno y el agente, inicializamos el tamaño del batch de muestras a recoger del replay buffer y hacemos una primera lectura del estado del cluster:



A continuación, ejecutamos un doble bucle, para cada episodio y para cada paso dentro de un episodio haciendo las siguientes operaciones:



Se actualiza la Target network con la frecuencia que se haya establecido en el fichero de configuración, el agente actúa sobre el entorno y se recogen la recompensa y el nuevo estado consecuencia de haber efectuado la acción elegida. A continuación, se guarda la combinación “estado-acción-recompensa-nuevo\_estado” en el replay buffer para su posterior utilización. Se actualiza el estado con el nuevo estado para el siguiente paso y, finalmente, y si el buffer ha recogido ya el suficiente número de muestras se efectúa el paso de entrenamiento del agente que implica el cálculo de los valores Q y la actualización de los parámetros de la red principal.

Al finalizar cada episodio salvamos los parámetros que se hayan calculado hasta el momento por seguridad.

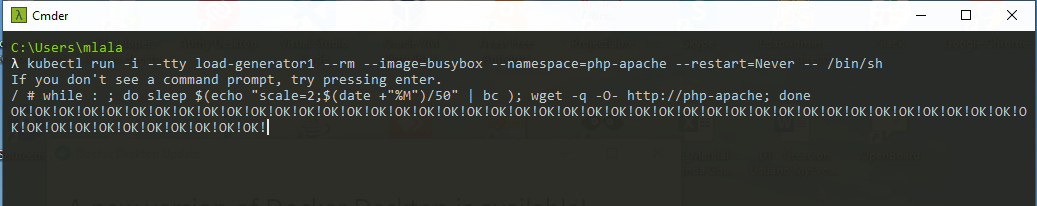
Una vez se ha lanzado este proceso y, con el fin de que el agente se vea sometido al mayor número de escenarios posibles abriremos uno o varios terminales simultáneos para hacer llamadas http a los pods modulando la carga con el tiempo. En cada uno de los terminales ejecutaremos la siguiente combinación de comandos:



Para crear un pod sencillo a partir de la imagen “busybox” que nos permite ejecutar comandos Linux desde un contenedor diminuto en tamaño y desde el que podemos efectuar las llamadas. El contenedor se llamará en este caso “load-generator1” (si creamos varios hay que ponerles nombres distintos) y se creará en nuestro namespace “php-apache”. Además, una vez levantado abriremos dentro de él una línea de comandos con “-/bin/sh” para ejecutar el siguiente:



Se trata de un bucle que hace la llamada a la url [http://php-apache](http://php-apache/) con diferentes frecuencia en función del minuto en el que se encuentre la hora del sistema. Los primeros diez minutos de cada hora no aplicará prácticamente carga sobre el sistema (una llamada por minuto), los siguientes diez provocará una carga media y a continuación otros diez de carga alta. En la siguiente media hora repetirá el patrón. Por cada llamada efectuada se verá un “OK” en la consola:



### Ejecución

Una vez completado el proceso de entrenamiento del agente este estará listo para ser puesto a funcionar. Para ello despojaremos al código que hemos utilizado en el punto anterior de los elementos inherentes al entrenamiento, cargando al inicio el modelo con los parámetros que se generaron durante el mismo:



Y, a continuación, ejecutamos un bucle infinito (o con la duración deseada) durante el cual el agente estará decidiendo continuamente la mejor acción a tomar en función del estado. Se obvian, por tanto, las llamadas al método “train” del agente, el almacenamiento de la información en el replay buffer y el guardado de los parámetros de la red. Así mismo, en la ejecución del método “act” del agente no se “explorará” haciendo uso de ε-greedy si no que siempre se “explotará”, eligiendo la acción que el agente entiende proporciona la mayor recompensa.



Tras cada acción se hará una espera (en el ejemplo de 15 segundos) hasta la siguiente lectura del estado y posterior acción para dar tiempo a Kubernetes a completar el estado indicado cómo deseado en el paso previo.

Dado que lo que queremos hacer es comparar el rendimiento de nuestro agente con el sistema HPA de Kubernetes, prepararemos también un programa para recoger los datos ejecución de este último de la misma forma que hemos preparado para el agente, al margen de monitorizar también con el Dashboard de Kubernetes. Este código simplemente recogerá el estado del Cluster con la misma frecuencia que para el agente y construirá los mismos gráficos sin interferir en el trabajo del HPA.



## Resultados

Para hacer las pruebas de los modelos entrenados y comparar el comportamiento del HPA de Kubernetes con el de nuestro agente, durante el desarrollo del proyecto se realizaron dos ciclos de pruebas.

En el primero, aplicado a los primeros modelos obtenidos, la prueba consistía en mantener distintas cargas de trabajo sobre el Cluster en intervalos de cinco minutos. Durante los cinco minutos iniciales se mantenía el sistema sin carga para, a continuación, aplicar una carga media durante los cinco minutos siguientes (una llamada cada 0.1 segundos) y finalizar con otros cinco a carga máxima (1 llamada cada 0.01 segundos), repitiendo este patrón dos veces.

En consecuencia, el perfil de consumo de CPU obtenido por el Dashboard de Kubernetes era el siguiente (por duplicado)

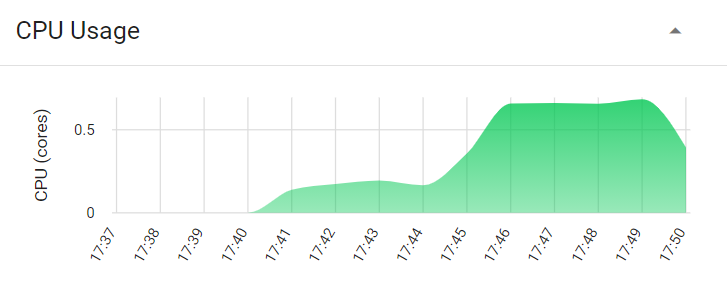


Ilustración 15: Patrón de carga para la prueba

Cada 15 segundos se recogía el nivel de uso de CPU y el número de pods levantados en cada intervalo y se almacenaba la información para plasmarlos posteriormente en una gráfica permitiendo así visualizar la evolución del segundo parámetro en función del primero.

Durante la prueba del Agente DQN se detiene el HPA para que no interfiera con el primero.

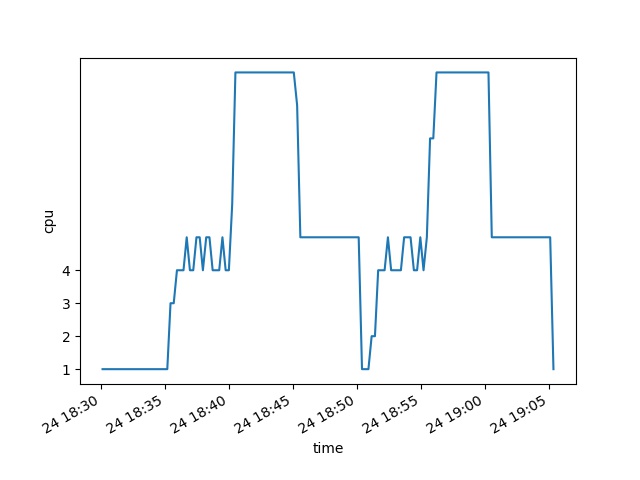
Aunque en durante este ciclo de pruebas no se obtuvo el modelo con los mejores resultados sí se pudo constatar que el agente era capaz de diferenciar el sistema cuando este estaba sometido a poca carga del momento en que se le sometía a un uso intensivo de CPU. Se incluye este resultado para ilustrar el proceso.

El resumen de los resultados obtenidos queda reflejado en las siguientes gráficas:

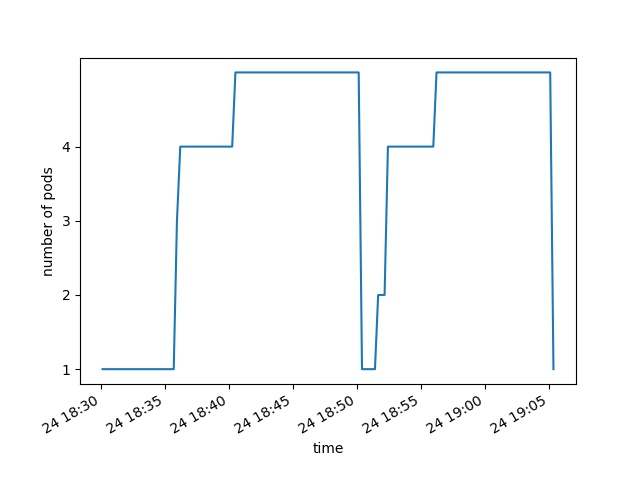
### Primer ciclo. Prueba HPA

Tras efectuar la prueba con el HPA las gráficas obtenidas son las siguientes:

Uso de CPU:

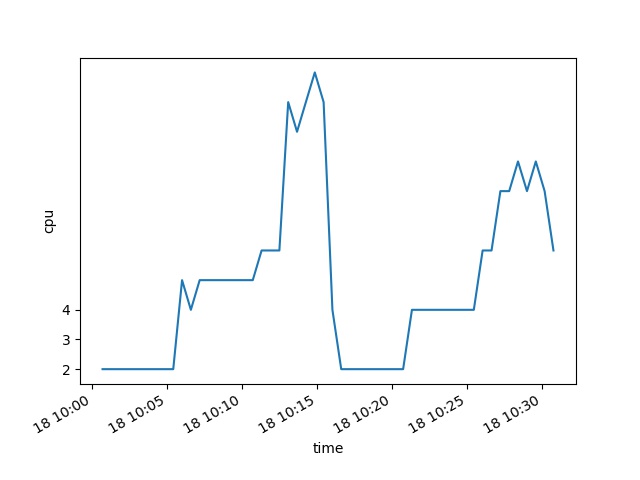


Número de Pods en funcionamiento:

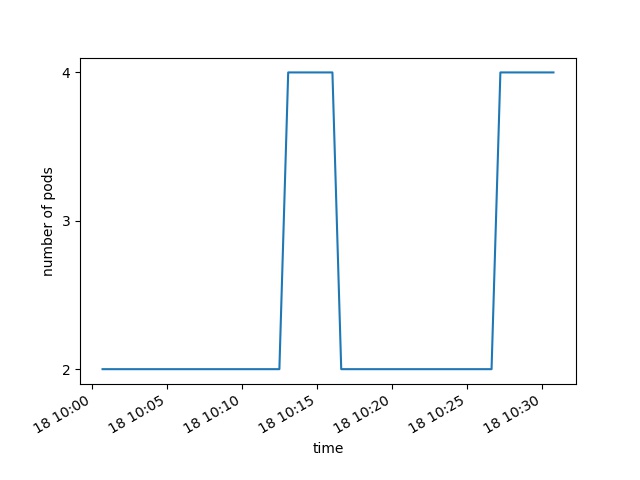


### Primer ciclo. Prueba Agente con aprendizaje reforzado

Uso de CPU:



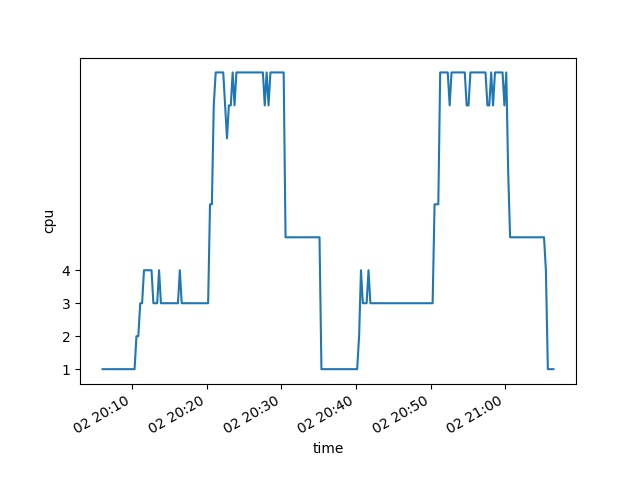
Número de Pods en funcionamiento:



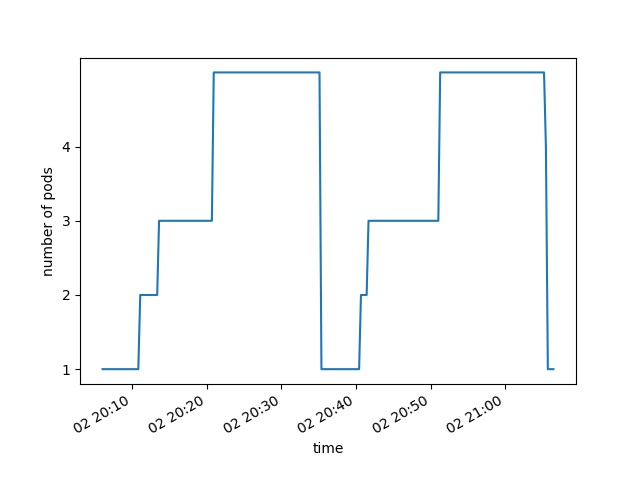
Con objeto de automatizar mejor las pruebas se diseño un segundo ciclo en el cada intervalo de carga se aumentaban a 10 minutos. Teníamos así una prueba de una hora con el mismo patrón duplicado de carga Baja-Media-Alta, que podía realizarse de forma desatendida aplicando la carga con el comando ya visto en la sección dedicada al entrenamiento del agente.

### Segundo ciclo. Prueba HPA

Uso de CPU:

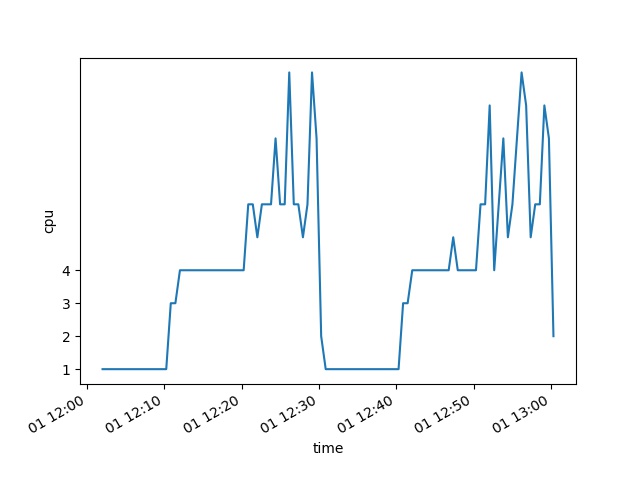


Número de Pods en funcionamiento:

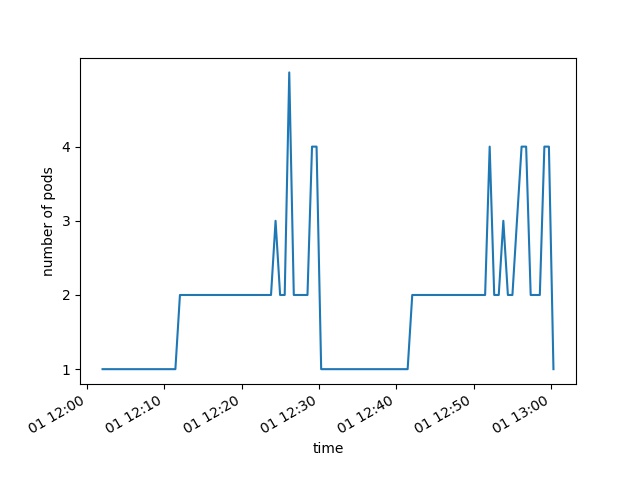


### Segundo ciclo. Prueba Agente con aprendizaje reforzado

Uso de CPU:



Número de Pods en funcionamiento:



## Conclusiones

A la vista de las gráficas correspondientes al primer ciclo de pruebas pudimos constatar que la red neuronal artificial del agente pudo capturar la diferencia entre una situación de bajo uso de la CPU frente a otra de uso intensivo de la misma. Sin embargo, no llegamos a una buena precisión, ya que sólo ha sido capaz de decidir dos acciones de entre las cinco posibles, dos pods para el escenario de poca carga y cuatro pods para el de alta.

Si bien no era el resultado esperado sí puso de manifiesto que el proceso era perfectamente factible y que se trataba de ajustar el sistema para intentar que diera mejores rendimientos. Se prosiguió por tanto aplicando sucesivas modificaciones tanto a los parámetros del agente como a los del entorno (recompensas y configuración) para tratar de mejorar los resultados.

Los resultados del mejor agente obtenido durante el segundo ciclo de pruebas se acercan más a los rendimientos del HPA sin llegar tampoco a suponer una mejora frente a este. En las gráficas vemos que el nuevo agente ya es capaz de mantener un único pod cuando la carga es baja y de pasar a dos cuando es media. Durante la fase de carga alta el comportamiento es más inestable alternando entre dos y cuatro pods con momentos puntuales de tres y cinco. Esto refleja que el agente ya es capaz de “visitar” todas las posibles acciones, de mantener estable la correcta en dos de las tres fases y, si bien con oscilaciones, tratar de hacer lo mismo en la tercera.

Algo que también se pone de manifiesto en las gráficas es el hecho de que el dato de CPU reflejado por las gráficas es el obtenido como sumatorio de los correspondientes a cada pod levantado, eso hace que la gráfica de la fase de uso intensivo de CPU por parte del agente tenga muchos picos (el número de pods es más inestable) mientras que la misma fase en el HPA tenga forma de meseta ya que éste último mantiene constante el número máximo de cinco pods durante toda la duración de la misma.

También podemos ver en la comparativa de las gráficas que el HPA mantiene el número de pods durante más tiempo una vez cambia la carga, manifestándose con más claridad al regresar a la fase de menor carga tras estar en la de máxima, cuando puede tardar hasta cinco minutos hasta volver a dejar un único pod en funcionamiento.

Un factor importante a tener en cuenta es que los ciclos de entrenamiento llevan mucho tiempo para un número reducido de episodios y “steps” de los mismos. Esto es debido al tiempo de espera que es necesario entre una acción en el sistema y el tiempo que necesita Kubernetes para crear o destruir los pods indicados y que estos adquieran la carga soportada por el sistema en ese momento. Este tiempo de espera ha oscilado en nuestro caso entre 15 y 20 segundos y, como consecuencia de esto, el agente no ha podido ver durante su entrenamiento un número elevado de estados. Durante los entrenamientos más largo logrado el agente pudo experimentar 5000 estados, empleando para ello entre 20 y 28 horas. Con entrenamientos más prolongados es muy posible que la inestabilidad detectada para la fase de mayor carga pudiera haber sido evitada.

### Posibles nuevas líneas de trabajo

Durante el desarrollo del proyecto uno de los puntos más delicados y que influye de forma decisiva en el aprendizaje del agente es la función de recompensa. Como ya hemos visto, en nuestro caso, y debido a las dificultades encontradas, se ha optado por una versión simple implementada con una tabla en la que para una determinada clave que recogía es estado y la acción ejercida por el agente se le asociaba un valor de recompensa. El problema fundamental de esta aproximación es que es muy rígida y que nos condiciona el número máximo de pods que podemos tener en el sistema si no hemos recogido esos casos en la tabla. Si se deseara un número alto de pods la tarea de generar la tabla puede volverse pronto intratable. Es por ello que una de las tareas de mejora sería tratar de encontrar otra fórmula para expresar la función de recompensa, quizás de forma similar a como el algoritmo DQN sustituye la tabla para la función Q por una red neuronal artificial.

Por una razón similar a lo comentado en el párrafo anterior en nuestro sistema el número de entradas de la red neuronal del agente es el máximo número de pods posibles, lo cual nos obliga a entrenar un agente diferente cada vez que queramos modificar dicho parámetro. Una solución sería definir los estados con el mismo sistema que hemos utilizado para determinar las recompensas, es decir, con un estado en el que cada entrada fuera el número de pods de cada tipo (carga baja, media y alta). Sin embargo, aunque esto eliminaría la limitación mencionada, por otro lado, nos obliga a mantener la discretización de los estados que introduce también otro elemento de rigidez en el sistema. Por tanto, una vía de mejora sería poder caracterizar el estado con los valores reales de uso de CPU (sin discretizar) pero de forma que pudiera recoger de forma dinámica la posible variación del parámetro de número máximo de pods.

Otra línea de exploración es, por supuesto, probar otros algoritmos de aprendizaje reforzado cómo los Monte Carlo o los de la familia de gradiente de póliza cómo el REINFORCE o PPO que, por limitaciones de tiempo, no ha sido posible implementar durante el desarrollo de este proyecto.

Aunque nos separemos del terreno del aprendizaje por refuerzo para adentrarnos en el terreno del aprendizaje supervisado, otra posibilidad a aplicar en nuestro problema de autoescalado es la de la utilización de una red neuronal artificial entrenada con un conjunto de estados o inputs modelados (por ejemplo) en la forma que lo hemos hecho para la tabla de recompensas. En este caso no será necesario incluir la acción del agente sino tan sólo los números de pods de cada tipo. La etiqueta para cada uno de estos estados lo constituiría el número de pods deseado en esa situación. Generando un set de datos con esa estructura y con el número suficiente de entradas la red neuronal resultante podría obtener la precisión requerida. El problema es este caso es que se dependería de los tramos utilizados para discretizar los estados, si estos variaran sería necesaria generación de un nuevo set de datos y un nuevo proceso de entrenamiento de la red.

## Bibliografía

Analytics Vidhya. «Deep Q-Learning | An Introduction To Deep Reinforcement Learning», 18 de abril de 2019. https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/.

Manning Publications. «Deep Reinforcement Learning in Action». Accedido 12 de julio de 2021. https://www.manning.com/books/deep-reinforcement-learning-in-action.

Manning Publications. «Grokking Deep Reinforcement Learning». Accedido 19 de septiembre de 2021. https://www.manning.com/books/grokking-deep-reinforcement-learning.

Kubernetes. «Horizontal Pod Autoscaler Walkthrough». Accedido 12 de julio de 2021. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale-walkthrough/.

*kubernetes-client/python*. Python. 2016. Reprint, Kubernetes Clients, 2021. https://github.com/kubernetes-client/python.

minikube. «Minikube». Accedido 12 de julio de 2021. https://minikube.sigs.k8s.io/docs/.

Kubernetes. «¿Qué es Kubernetes?» Accedido 12 de julio de 2021. https://kubernetes.io/es/docs/concepts/overview/what-is-kubernetes/.

Ravichandiran, Sudharsan. *Deep Reinforcement Learning with Python: Master classic RL, deep RL, distributional RL, inverse RL, and more with OpenAI Gym and TensorFlow, 2nd Edition*. Birmingham ; Mumbai, 2020.

Sutton, Richard S., y Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second edition. Cambridge, Mass: A Bradford Book, 1998.

Ushio, Tsuyoshi. «Kubernetes in Three Diagrams». Medium, 5 de febrero de 2018. https://tsuyoshiushio.medium.com/kubernetes-in-three-diagrams-6aba8432541c.

## Otros Recursos

### Repositorio de Código

El código del proyecto puede encontrarse en el siguiente repositorio de GitHub:

<https://github.com/mlalandag/HPAwDRL>

### Cliente Python de Kubernetes

<https://github.com/kubernetes-client/python>

1. «Minikube». [↑](#footnote-ref-1)
2. «¿Qué es Kubernetes?» [↑](#footnote-ref-2)
3. «¿Qué es Kubernetes?» [↑](#footnote-ref-3)
4. Ushio, «Kubernetes in Three Diagrams». [↑](#footnote-ref-4)
5. «Horizontal Pod Autoscaler Walkthrough». [↑](#footnote-ref-5)
6. *kubernetes-client/python*. [↑](#footnote-ref-6)
7. «Deep Reinforcement Learning in Action». [↑](#footnote-ref-7)
8. Sutton y Barto, *Reinforcement Learning*. [↑](#footnote-ref-8)
9. Sutton y Barto. [↑](#footnote-ref-9)
10. Ravichandiran, *Deep Reinforcement Learning with Python*. [↑](#footnote-ref-10)
11. «Deep Q-Learning | An Introduction To Deep Reinforcement Learning». [↑](#footnote-ref-11)
12. Ravichandiran, *Deep Reinforcement Learning with Python*. [↑](#footnote-ref-12)
13. Ravichandiran. [↑](#footnote-ref-13)
14. «Grokking Deep Reinforcement Learning». [↑](#footnote-ref-14)
15. «Deep Q-Learning | An Introduction To Deep Reinforcement Learning». [↑](#footnote-ref-15)