Trabajo de Fin de Master

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

Marcial Lalanda González-Bueno

2021

Contenido

[**Autoescalado horizontal en Kubernetes con Aprendizaje por Refuerzo** 1](#_Toc77350977)

[Introducción 3](#_Toc77350978)

[Entorno de trabajo 4](#_Toc77350979)

[¿Qué es Kubernetes? 4](#_Toc77350980)

[Pod 5](#_Toc77350981)

[Deployment 5](#_Toc77350982)

[Replica Set 5](#_Toc77350983)

[Service 6](#_Toc77350984)

[Namespace 6](#_Toc77350985)

[Horizontal Pod Autoscaling 6](#_Toc77350986)

[Monitorización 7](#_Toc77350987)

[Estado 8](#_Toc77350988)

[Acción 8](#_Toc77350989)

[¿Qué es el Aprendizaje por Refuerzo? 9](#_Toc77350990)

[Q-Learning 10](#_Toc77350991)

[Deep Q Network (DQN) 12](#_Toc77350992)

[Policy Gradient Methods 13](#_Toc77350993)

[Reinforce 14](#_Toc77350994)

[Implementación 16](#_Toc77350995)

[Resultados 17](#_Toc77350996)

[Conclusiones 18](#_Toc77350997)

[Bibliografía 19](#_Toc77350998)

[Otros Recursos 20](#_Toc77350999)

[Repositorio de Código 20](#_Toc77351000)

[Cliente Python de Kubernetes 20](#_Toc77351001)

[Indice de Figuras 20](#_Toc77351002)

# 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 algunos 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 diferentes algoritmos 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.

## 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 disponibulidad.

### 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 especifíca 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).

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.

Crearemos el deployment de Kubernetes mediante el fichero yaml en el que declaramos todos los recursos que vamos a utilizar. 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:



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.

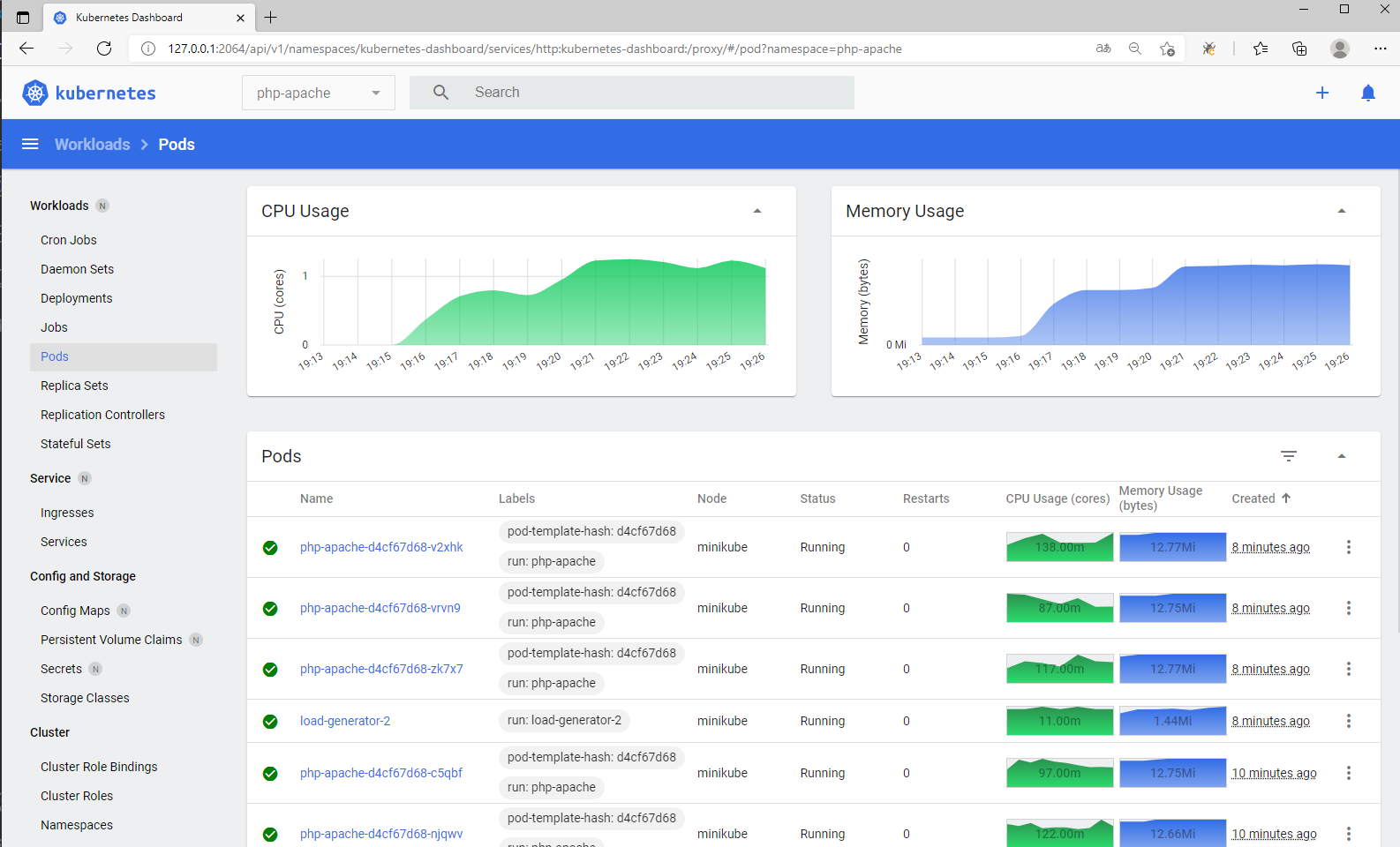


Ilustración 3: Kubernetes Dashboard

### 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 lo 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 cuales 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 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.

### Q-Learning

La siguiente imagen muestra una versión de uno de los algoritmos 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 5: 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):

#### Poliza

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 éste 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.

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 [[11]](#footnote-11):

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.

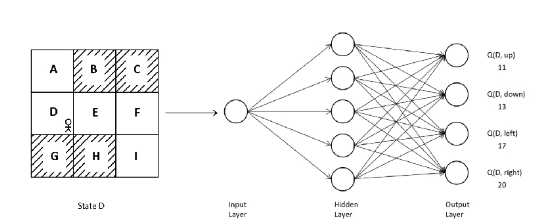


Ilustración 6: Deep Q Network (DQN) [[12]](#footnote-12)

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

DQN será el primer algoritmo que utilicemos para realizar nuestra comparativa.

### Policy Gradient Methods

Otra familia de algoritmos de aprendizaje reforzado son los denominados métodos de gradiente de póliza, en los que, a diferencia de lo visto anteriormente, el cálculo de la póliza óptima se puede hacer sin tener que calcular la función Q óptima. Esto evita alguna que otra desventaja de DQN, cómo el hecho de que sólo puedan aplicarse a entornos con un espacio de acciones discreto. Aunque en nuestro caso el espacio de acciones también es discreto trataremos de ver que resultado obtenemos al aplicar alguno de estos algoritmos.

A grandes rasgos los métodos de gradiente de póliza se basan en la utilización de una póliza estocástica, es decir, que la acción a aplicar se selecciona en función de una distribución de probabilidad sobre el espacio de acciones del entorno, y en aproximar dicha póliza con una red neuronal. Podemos caracterizar nuestra póliza mediante la siguiente expresión:

Que nos da la probabilidad de tomar la acción a dado el estado s y que está parametrizada por el conjunto de parámetros *θ.*

Si alimentamos la red neuronal que aproxima la póliza con un estado s obtendremos la probabilidad para todas las acciones posibles en este estado y, al tratarse de una póliza estocástica, se seleccionará la siguiente acción basándose en dicha distribución de probabilidad.

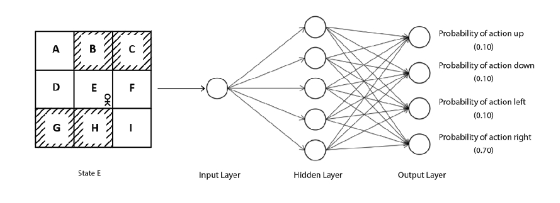


Ilustración 7: Policy Network [[13]](#footnote-13)

La pregunta es entonces: ¿Cómo vamos actualizando los parámetros de nuestra red neuronal para poder optimizar la póliza? Para ello ejecutaremos distintos episodios y nos guardaremos el estado, la acción y la recompensa hasta el final de cada episodio. Si hemos obtenido un buen resultado asignaremos una alta probabilidad a todas las acciones que se hayan aplicado en dicho episodio, en caso contrario asignaremos una baja probabilidad a dichas acciones. Ello nos permitirá ir ajustando los parámetros de la red neuronal en consecuencia. Este ajuste lo podremos realizar actualizando en cada iteración los pesos de la red neuronal de la siguiente forma:

Donde *J(θ)* depende de forma directamente proporcional de la recompensa obtenida durante el episodio o episodios. Estamos por tanto tratando de maximizar dicha función y, por tanto, en lugar de utilizar descenso de gradiente lo que aplicamos en este caso es ascenso de gradiente.

### Reinforce

El más sencillo de la familia de los algoritmos basados en gradientes de póliza es el denominado REINFORCE, del que se muestra el pseudocódigo en la siguiente figura.

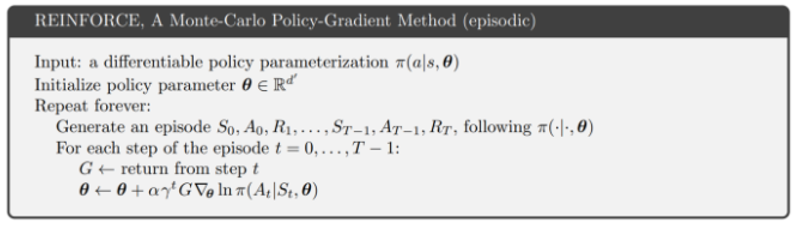


Ilustración 8: Pseudocódigo del algoritmo REINFORCE [[14]](#footnote-14)

Vemos que el primer paso es la inicialización de los parámetros de la red, representados por *θ*.

A continuación generamos episodios o trayectorias de determinado número de pasos T siguiendo la póliza *π*.

En el pseudocódigo vemos que, por cada paso de un episodio, se guarda el retorno o recompensa (denotado por G en lugar de R(t)) y se procede a la actualización de los parámetros. Esta actualización también se podría hacer tras un episodio completo o tras completar varios.

El cálculo del gradiente se realiza conforme a la fórmula mostrada de tal forma que podemos establecer la equivalencia:

=

Donde la expresión del logaritmo representa la log-probabilidad de escoger la acción A en el estado S en el paso *t*. Y cómo hemos dicho antes G es el retorno obtenido. *γ* corresponde al factor de descuento.

El principal problema que presenta este algoritmo es que adolece de una alta varianza debido a que pertenece a la familia de los algoritmos denominados “on-policy”. Esto quiere decir que estamos actualizando tras cada paso, episodio o conjunto de episodios la misma póliza con la que los estamos generando. Al poder obtener resultados muy distintos de cada episodio estamos introduciendo inestabilidad en el aprendizaje. Aún así veremos cómo se comporta en nuestro escenario.

## Implementación

**Environment**

**Estados**

**Acciones**

**Recompensas**

**Agente**

**Simulación**

## Resultados

## Conclusiones

## Bibliografía

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

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>

### Indice de Figuras

[Ilustración 1: Despliegue en contenedores 3](#_Toc77350137)

[Ilustración 2: Arquitectura de Kubernetes 4](#_Toc77350138)

[Ilustración 3: Kubernetes Dashboard 6](#_Toc77350139)

[Ilustración 4: Elementos de un sistema de aprendizaje por refuerzo 8](#_Toc77350140)

[Ilustración 5: Algoritmo Q-Learning 9](#_Toc77350141)

[Ilustración 6: Deep Q Network (DQN) 12](#_Toc77350142)

[Ilustración 7: Policy Network 13](#_Toc77350143)

[Ilustración 8: Pseudocódigo del algoritmo REINFORCE 14](#_Toc77350144)

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. Ravichandiran. [↑](#footnote-ref-11)
12. Ravichandiran. [↑](#footnote-ref-12)
13. Ravichandiran. [↑](#footnote-ref-13)
14. Sutton y Barto, *Reinforcement Learning*. [↑](#footnote-ref-14)