

TRABAJO DE FIN DE GRADO EN INGENIERÍA INFORMÁTICA
FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

Evaluación de rendimiento de arquitecturas paralelas y de propósito específico para el aprendizaje por refuerzo en juegos

Performance evaluation of parallel and specific-purpose architectures for reinforcement learning in games

Autor:
JAVIER GUZMÁN MUÑOZ

Profesor director:
FRANCISCO IGUAL PEÑA

Profesor codirector:
LUIS M^a COSTERO VALERO



DOBLE GRADO EN INGENIERÍA INFORMÁTICA - MATEMÁTICAS
Curso 2020-2021

Resumen

Las aplicaciones de aprendizaje por refuerzo se usan en la actualidad para resolver problemas de todo tipo en campos muy diversos. Sin embargo, una de las principales desventajas que presentan es el elevado coste computacional del entrenamiento de los modelos necesarios. Con este trabajo de fin de grado se pretende mejorar este proceso mediante la parallelización de los algoritmos empleados y el uso de distintas arquitecturas hardware que variarán los tiempos requeridos. Los modelos entrenados pueden aplicarse para obtener la mejor secuencia de acciones que podemos realizar sobre un entorno y mejorar la recompensa obtenida. Este proceso, que se denomina inferencia, aunque tiene menor complejidad computacional, se realiza muchas más veces, por lo que se han desarrollado procesadores de propósito específico para llevar a cabo esta tarea. Por ello, también es conveniente evaluar su rendimiento en estos soportes y compararlos con otras unidades de procesamiento más generales. Tras definir el escenario en el que nos vamos a mover y los recursos necesarios para ello, se proponen una serie de experimentos de los procesos de entrenamiento e inferencia que nos permitirán evaluar el rendimiento en términos del tiempo empleado, de la utilización de los recursos disponibles y del consumo de energía de distintas arquitecturas hardware, viendo cuál es más conveniente usar en cada caso.

Palabras clave

Aprendizaje por refuerzo, algoritmo PPO, red neuronal de convolución, *Ray RLlib*, entornos *Gym*, TPU Google Coral, aceleradores hardware.

Abstract

Nowadays, reinforcement learning applications are used to solve all kinds of problems in a wide variety of fields. However, one of their main disadvantages is the high computational cost of training the necessary models. This Bachelor's thesis aims at improving this process by parallelizing the involved algorithms and by using different hardware architectures, which will differ in the amount of time used. We can run previously trained models to obtain the best sequence of actions to interact with the environment in order to improve the reward obtained. Although this process, called inference, has a lower computational complexity, it is usually repeated many times and requires a fast response. In order to execute inference in an efficient way, specific-purpose processors have been developed, so it is convenient to evaluate its performance on these devices and compare them with more general processing units. After defining the scenario and the resources needed, we propose a series of experiments to test the training and inference processes, evaluating the performance in terms of the time spent, the resource usage and the power consumption when using different architectures, analyzing which is the best option in each case.

Keywords

Reinforcement learning, PPO algorithm, convolutional neural network, *Ray RLlib*, *Gym* environments, Google Coral TPU, hardware accelerators.

Índice general

1. Introducción	7
2. Fundamentos	15
2.1. Aprendizaje por refuerzo	15
2.1.1. Políticas en el aprendizaje por refuerzo	16
2.2. Algoritmo de Optimización de Política Próxima (PPO)	17
2.2.1. Algoritmos de gradiente	18
2.2.2. Algoritmos de región de confianza	18
2.2.3. Fundamentos del algoritmo PPO	19
2.3. Redes Neuronales de Convolución	21
2.3.1. Redes de convolución y aprendizaje por refuerzo	24
2.4. Tensorflow	24
2.4.1. Keras	25
2.4.2. Tensorflow Lite	25
2.5. Ray y RLlib	25
2.5.1. RLlib	26
2.5.2. Algoritmo PPO en RLlib	26
2.6. Entornos Gym	28
2.7. TPU y Google Coral	30
2.8. Cuantización de modelos	32
3. Implementación	35
3.1. Descripción de los entornos de pruebas	35
3.2. Descripción de los modelos	36
3.2.1. Descripción del entorno del agente	36
3.2.2. Modelo de Tensorflow Keras	37
3.2.3. Modelos propuestos	39
3.2.4. Modelos seleccionados	40
3.3. Análisis del rendimiento	41
3.3.1. Implementación de los experimentos de entrenamiento	41
3.3.2. Implementación de los experimentos de inferencia	44
4. Resultados	51
4.1. Resultados del proceso de entrenamiento	51
4.1.1. Uso de los recursos disponibles	51
4.1.2. Tiempos empleados	57
4.1.3. Análisis del uso que se hace de las distintas CPUs	66
4.1.4. Conclusiones extraídas de los experimentos de entrenamiento	68

4.2. Resultados de la inferencia de modelos	68
4.2.1. Inferencia en RLlib	68
4.2.2. Inferencia sobre el acelerador Google Coral	70
5. Conclusiones	73
A. Funcionamiento de los scripts de Python	79
A.1. Script de entrenamiento	79
A.2. Script de inferencia en RLlib	81
A.3. Scripts de exportación y cuantización de modelos para la TPU	82
A.3.1. Script de exportación de modelos	82
A.3.2. Script de creación de modelos de Tensorflow Lite	83
A.3.3. Script de creación de datasets para la cuantización	83
A.3.4. Script de cuantización de modelos de Tensorflow Lite	83
A.4. Scripts de inferencia de modelos de Tensorflow Lite	84
Bibliografía	88

Capítulo 1

Introducción

Hoy en día, el **aprendizaje automático** o *machine learning* está presente en prácticamente todos los campos del conocimiento (ingeniería, finanzas, medicina, ...). Esta rama de la computación trata de emular la manera en la que el cerebro humano aprende a partir de la información que le llega de su entorno. Son múltiples los paradigmas que existen dentro de este campo. En este trabajo nos vamos a centrar en uno de ellos, el **aprendizaje por refuerzo**. Su idea fundamental es llevar a cabo el aprendizaje por medio de la interacción continua y el intercambio de información entre un **agente** y un **entorno** con el que desea aprender a interaccionar. El agente recibe **observaciones** del entorno y, basándose en ellas, entrena una **política** que determinará una **acción** que realizará en el entorno y por la que obtendrá una **recompensa**. El objetivo final será desarrollar un modelo que maximice las recompensas obtenidas, esto es, que dada una observación del entorno sepa cual es la mejor acción que nos llevará a optimizar la recompensa final acumulada.

Para obtener un modelo de aprendizaje automático hay que realizar un **proceso de entrenamiento**, en el que se configura el propio modelo a través de interacciones sucesivas con el entorno en el que va mejorando su toma de decisiones. Una vez concluya este proceso ya podemos usar nuestro modelo para realizar **inferencias**, esto es, interacciones con el entorno en las que en cada momento se escoja la mejor acción de acuerdo con la política que hemos entrenado. Existen diversos modelos (algoritmos) que se pueden usar en problemas de aprendizaje. Uno de ellos son las **redes neuronales de convolución**, que reciben imágenes y son capaces de capturar dependencias espaciales y temporales en ellas.

El escenario de aprendizaje por refuerzo puede aplicarse a todo tipo de problemas de aprendizaje. En este trabajo se adaptará el problema para el caso en el que queremos aprender a jugar a **videojuegos sencillos** a partir de las observaciones de la pantalla y de las puntuaciones obtenidas.

Para modelar el entorno del problema nos ayudaremos de la biblioteca **Gym**¹, que proporciona entornos sobre los que podemos desarrollar escenarios de aprendizaje por refuerzo. Los entornos que tomaremos nos suministrarán datos en forma de **imágenes** como observaciones del entorno, lo que propiciará también el uso de redes neuronales de convolución.

El **soporte hardware** sobre el que se pueden desarrollar estos procesos es muy variado: desde procesadores genéricos (CPUs) hasta otros de propósito específico para tareas concretas (como por ejemplo la inferencia), pasando por aceleradores gráficos como GPUs (*Graphics Processing Units*).

¹<https://gym.openai.com/>

Motivaciones

Uno de los principales problemas que presenta el desarrollo de modelos de aprendizaje por refuerzo es el **elevado coste computacional** de los algoritmos empleados para el proceso de entrenamiento. Para ello, se tratan de acelerar los cálculos implicados con el uso de GPUs. Además, la estructura de muchos de los algoritmos de entrenamiento habituales en este tipo de problemas va a permitir una **paralelización** que mejore también su rendimiento.

Sin embargo, es necesario adaptar el proceso de entrenamiento para que sea posible su correcta ejecución en este soporte y la paralelización de los algoritmos, que en muchos casos no será tarea sencilla. Y aquí es donde aparece la biblioteca ***RLLib***² de ***Ray***, que elimina estas dificultades.

Por otro lado, introducir recursos como las GPUs incrementa notablemente el consumo de potencia de nuestro sistema, por lo que es un factor también a tener en cuenta cuando configuremos el entrenamiento.

En cuanto al proceso de inferencia, aunque una inferencia individual tenga un coste mucho menor que una iteración de entrenamiento, las inferencias se realizan un número muy elevado de veces, por lo que, en conjunto, el coste de las inferencias suele ser mayor que el del entrenamiento (al fin y al cabo, el entrenamiento lo realizamos una vez pero las inferencias siempre que queramos usar nuestro modelo para resolver el problema de aprendizaje). Por ello, es importante realizar las inferencias en dispositivos en los que el coste de cada inferencia individual sea muy bajo, para que una diferencia de coste casi imperceptible no se convierta en un problema al verse multiplicada cuando realizamos gran cantidad de pasos de inferencia. Así, cabe la posibilidad de que variando el hardware sobre el que se ejecute se mejore su rendimiento en este aspecto, al igual que en el entrenamiento. Y es que existe hardware específico para realizar este proceso, como la **TPU de Google Coral**³, ideada como un procesador de matrices que resulta idóneo para ejecutar inferencias sobre modelos de aprendizaje automático. Además, otra de las ventajas que ofrece es el **ahorro en consumo de energía** respecto a las CPUs y GPUs. El uso de este dispositivo añade la dificultad de que es necesario adaptar los modelos para que podamos ejecutar inferencias en él, y este proceso podría en muchos casos no ser trivial.

Objetivos

Motivados por lo expuesto anteriormente, el **objetivo principal** del presente trabajo de fin de grado es **evaluar el rendimiento de los procesos de aprendizaje e inferencia en el escenario del aprendizaje por refuerzo en diferentes arquitecturas hardware**, analizando las ventajas e inconvenientes de cada una de ellas. Así, tendremos información útil a la hora de entrenar o de aplicar modelos de aprendizaje por refuerzo que nos permitirá optimizar el tiempo, la utilización de recursos o el consumo de energía en cada caso concreto. Listamos a continuación algunos de los objetivos específicos en los que podemos desglosar este objetivo principal:

1. Modelar correctamente el escenario de aprendizaje haciendo uso de las bibliotecas *RLLib* y *Gym*.
2. Ejecutar un mismo proceso de entrenamiento sobre diferentes arquitecturas hardware haciendo uso de las utilidades de *Ray*.
3. Evaluar el rendimiento variando los parámetros propios de los modelos que se consideran.
4. Analizar y extraer conclusiones sobre el proceso de entrenamiento, que permitan determinar las ventajas e inconvenientes en términos de tiempo de aprendizaje y utilización de los recursos de cada uno de los escenarios evaluados.

²<https://docs.ray.io/en/master/rllib.html>

³<https://coral.ai/products/>

5. Ejecutar procesos de inferencia haciendo uso de las utilidades de *Ray*, variando los recursos hardware empleados para ello.
6. Ser capaces de ejecutar inferencias en el acelerador Google Coral sobre modelos previamente entrenados con *RLlib* y posteriormente cuantizados.
7. Extraer conclusiones respecto a la inferencia de modelos y analizar las ventajas e inconvenientes que supone el uso de aceleradores de propósito específico.

Metodología

Para poder lograr estos objetivos trataremos de manera separada los procesos de entrenamiento e inferencia. Elegiremos una serie de modelos (dados como redes neuronales de convolución) que se distingan entre sí fundamentalmente por el tamaño de las entradas que reciben. Una vez elegidos los modelos, propondremos una serie de **experimentos** que permitan evaluar el rendimiento de los procesos de entrenamiento e inferencia variando los recursos hardware empleados, desarrollando el **código Python** necesario para ello. Posteriormente, transformamos algunos de los modelos obtenidos como resultado de los experimentos de entrenamiento para que puedan ser ejecutados sobre la TPU Google Coral, proceso que, como veremos, no es trivial. En todo momento, nos preocuparemos también de tener en cuenta qué aspectos vamos a medir (**métricas**) en cada uno de los procesos y cómo obtendremos esta información.

Tras plantear los experimentos se procede a su **ejecución en servidores remotos** del Departamento de Arquitectura de Computadores y Automática, almacenando los datos obtenidos como resultado de estas ejecuciones.

Seguidamente, debemos organizar los datos para proceder a su **análisis**. Se elaboran una serie de *scripts* de *Python* que agrupan los datos y generan **gráficas y ficheros tabulares** de los mismos, comparando los resultados de los distintos experimentos y facilitando así la **extracción de conclusiones**. Para ello, se hace uso de bibliotecas específicas de *Python* para el tratamiento y la información de datos, como *Pandas* y *Matplotlib*.

Estructura del trabajo

Este trabajo de fin de grado está compuesto por la presente **memoria** y por el código empleado para la realización de los distintos experimentos antes mencionados y la obtención de resultados, que se encuentra en un **repositorio de Github** del usuario *javigm98* y al que se puede acceder mediante la siguiente URL: <https://github.com/javigm98/Mejorando-el-Aprendizaje-Automático>.

La memoria se organiza en **cinco capítulos**. Esta introducción, en la que se presenta el tema y los objetivos que se pretenden conseguir, constituye el **primero** de los capítulos.

En el **segundo** de ellos se presentan los **fundamentos** de muchos de los conceptos con los que vamos a trabajar. Así, comenzamos definiendo de manera teórica el paradigma del **aprendizaje por refuerzo** y el **algoritmo PPO** que usaremos durante el entrenamiento. También, se explica en qué consisten las **redes neuronales de convolución** y cómo las usaremos para procesar observaciones en nuestro escenario de aprendizaje por refuerzo. Además, se introducen los *frameworks* y bibliotecas de *Python* que se utilizarán para la implementación del trabajo: *Tensorflow*, *Ray*, *RLlib* y *Gym*. Por último, se describe en qué consiste el proceso de **cuantización** de modelos, que será necesario para poder ejecutar inferencias sobre el acelerador **TPU de Google Coral**, cuyas características y las ventajas que ofrece se detallan en una última sección.

En el **tercer** capítulo exponemos los **detalles de implementación** tenidos en cuenta para la consecución de los objetivos del trabajo. Tras detallar los sistemas de los que disponemos para la

realización de este trabajo de fin de grado, se proponen y seleccionan una serie de modelos con los que evaluaremos el rendimiento de los procesos propios del aprendizaje por refuerzo en diferentes arquitecturas hardware. Además, se especifican los experimentos que llevaremos a cabo y cómo los implementaremos con los recursos de los que disponemos, tanto a nivel de software y programación como del propio hardware.

El **cuarto** capítulo recoge los **resultados de los experimentos** que se proponían en el capítulo anterior. Se recogen datos tanto de los procesos de **entrenamiento** como de **inferencia** y se analizan detalladamente, incluyendo múltiples gráficas y tablas para facilitar el análisis. De estos resultados extraeremos una serie de **conclusiones** que nos permitan establecer diferencias entre cada una de las configuraciones probadas.

Por último, el **quinto** capítulo recoge las **conclusiones globales** obtenidas tras las realización del trabajo y se valora la **consecución de los objetivos** anteriormente propuestos.

Adicionalmente y a modo de apéndice, se añade una **guía detallada de los principales *scripts*** que constituyen la parte de código de este trabajo. Se explica el propósito y modo de uso de los mismos.

La mayoría de las imágenes y diagramas que aparecen en esta memoria son de elaboración propia, generadas con la herramienta *Mathcha*⁴ o con la biblioteca de *Python Matplotlib*⁵. Para aquellas que se toman de otras fuentes se indica la procedencia de las mismas en su descripción.

⁴<https://www.mathcha.io/>

⁵<https://matplotlib.org/>

Introduction

Nowadays, **machine learning** applications can be found in almost all fields of knowledge (engineering, finance, medicine...). This part of the computational sciences tries to emulate the way in which human beings learn from the information that they get from their surrounding environment. There are many areas inside this general field. In this thesis, we are going to focus on one of them, the **reinforcement learning**. Its main idea is to accomplish the learning task through continuous interaction and information exchange between an **agent**, that wants to learn how to interact with an environment, and this **environment**. An agent receives **observations** from the environment and uses that information to train a **policy** capable to decide which **action** has to be taken in the environment, receiving back a **reward** for taking that action. The main target is to build a model that maximizes cumulative reward, in other words, when we provide an observation to it, the model will know which is the best action that will lead us to optimize the final reward.

In order to build a machine learning model, a **training process** is required, where the model itself is customized through successive iterations with the environment that improve its decision making. Once this process is finished, we can use the model to run **inferences**, that is to say, interactions with the environment taking the best action at each time in accordance with the previously trained policy. There are different models (algorithms) that can be used in learning problems. **Convolutional neural networks** are one of them. This algorithm receives images as input and is able to capture spatial and temporal dependencies present on them.

A reinforcement learning scenario can be applied to solve all kinds of problems. In this thesis, the scenario will be adapted to a learning task whose goal is to learn how to play **simple video games** properly, using screen images and the scores obtained.

Gym⁶ library offers us many environments that can be used in the modeling of our learning scenario. The environments that we will use provide **images** as observations, so it makes sense to use convolutional neural networks to process them.

There are many **hardware components** where these processes can be executed: from general-purpose processors (CPUs) to specific-purpose ones (for example inference specific processing units), including graphic accelerators such as GPUs.

Motivations

One of the main problems that come out when developing reinforcement learning models is the **high computational cost** of the algorithms involved in the training process. One approach to solve this problem is to use GPUs to speed up the required calculations. In addition to that, the structure of many of the most used training algorithms allows a parallelization that improves its performance.

However, it is necessary to adapt the training process so that it can be properly run on this platform and to make possible to parallelize the algorithms, something that won't be easy in many cases. And

⁶<https://gym.openai.com/>

now is the moment when ***RLLib***⁷ library from ***Ray*** appears, removing these difficulties.

By the way, adding a GPU to our system increases considerably the power consumption, so this is also a fact that we should have in mind when setting up the training processes.

Furthermore, each individual inference has a low cost if we compare it with that for a training iteration, but inferences are run a high number of times, so, the whole process of inference usually has a higher cost than the training process (after all we only train our model once, but we run inferences each time we want to use our model to solve the learning problem). That's why it is important to run inferences on devices with a low cost for each single inference, in order to avoid that a irrelevant difference of costs turns into a real problem when it is multiplied due to the large number of inference steps executed. So, it is still possible to improve the performance varying the hardware used, as we can do for the training. And the key here is that there is specific hardware to run this process, such as the **Google Coral TPU**⁸, designed as a matrix processor that is quite suitable platform to run machine learning inferences on it. What's more, one of the advantages that these processors offer is **power consumption saving** compared to CPUs and GPUs. Using these devices adds an extra difficulty due to the transformations we need to carry out on the models, being this process non trivial.

Objectives

Motivated by the above, the **main objective** of this Bachelor's thesis is to **evaluate training and inference processes in a reinforcement learning scenario on different hardware architectures**, analyzing the advantages and disadvantages of each of them. So, we will have useful information when training or inferring reinforcement learning models and we will be able to optimize the time involved, the resources utilization and the power consumption in each specific case. We now list some of the specific objectives derived from the main one:

1. Correctly modeling the reinforcement learning scenario using *RLLib* and *Gym* libraries.
2. Running the same training process on different hardware architectures using *Ray* utilities.
3. Evaluating performance by varying the parameters of the models.
4. Analyzing and drawing conclusions from the training process, so that we can establish the advantages and disadvantages in terms of the learning time and the resources utilization in each of the evaluated scenarios.
5. Running inference processes using *Ray* utilities to vary available hardware resources.
6. Being able to run inferences on the Google Coral accelerator using models previously trained and quantized with *RLLib*.
7. Drawing conclusions about model inference and analyzing the advantages and disadvantages of using specific-purpose accelerators.

Methodology

In order to achieve the previous objectives we will consider separately training and inference processes. We will choose a series of models (given as convolutional neural networks) that mainly differ in the size of the given inputs. Once we have chosen the models, we will propose a series of **experiments**

⁷<https://docs.ray.io/en/master/rllib.html>

⁸<https://coral.ai/products/>

that allow us to evaluate the training and inference processes performance varying the hardware resources involved, and developing the required ***Python*** code to accomplish this task. After that, we will transform some of models obtained as a result of the training process so that they can be run on the Google Coral TPU, a process that as we will see is not easy to carry out. At all times, we will also take care of the aspects that we want to measure (**metrics**) in each of the processes and how we will collect this information.

After setting up the experiments we **execute them on some remote servers** from the Computer Architecture and Automation Department, saving the data obtained as a result of these executions.

Then, we must organize the data so that we can **analyze** it. We create a series of *Python* scripts that gather different data and generate **graphs and tabular files**, comparing the results from the different experiments and making it easy to **draw conclusions**. For this purpose, some specific *Python* libraries for data processing are used, such as *Pandas* and *Matplotlib*.

Document structure

This Bachelor's thesis is composed of this **report** and the **code** used for running the different experiments and collecting their results, that can be found in the **GitHub repository** that can be accessed from this URL: <https://github.com/javigm98/Mejorando-el-Aprendizaje-Automatico>.

The report is made up of **five chapters**. This introduction, where the main ideas and objectives of the thesis are showed, is the **first** of them.

The **second** chapter introduces the **fundamentals** of many of the concepts that we are going to work with. We start defining in a theoretical way the **reinforcement learning** paradigm and **PPO algorithm** that we will use for the training. We also explain what **convolutional neural networks** are and how can we use them to process images in our reinforcement learning scenario. Also, we briefly introduce the frameworks and libraries used to implement the code: **Tensorflow**, **Ray**, **RLLib** and **Gym**. To end, we describe the **quantization** process required to run inference on the **Google Coral TPU**, an accelerator whose features and advantages are detailed in the last section.

In the **third** chapter we explain the **implementation details** taken into account for achieving the objectives of this thesis. After detailing the available servers that we can access to, we propose and select a series of models that we will use to evaluate the performance of reinforcement learning processes in different hardware architectures. Furthermore, we specify the experiments to be executed and how we will implement them with the available resources that we have, form the point of view of the software and programming as well as the hardware.

The **fourth** chapter collects the **results of the experiments** proposed in the previous chapter. We collect data from the **training** process as well as from the **inference** one and we carefully analyze the information, including many graphs and tables to make the analysis clearer. We will draw a series of conclusions from these results that will allow us to set a series of differences among all tested configurations.

Lastly, the **fifth** chapter contains the **conclusions** drawn from the whole thesis and we discuss the achievement of the initial goals.

To complement all the previous information, we add an appendix containing a **detailed guide of the main scripts** that make up the coding part of the thesis. We explain the purpose and the way of use of each of them.

Most of the images and graphs that this report contains are self-prepared using *Mathcha*⁹ tool and *Matplotlib*¹⁰ *Python* library. For those borrowed from other sources we cite where they come from in their descriptions.

⁹<https://www.mathcha.io/>

¹⁰<https://matplotlib.org/>

Capítulo 2

Fundamentos

2.1. Aprendizaje por refuerzo

Describiremos en esta sección el paradigma de aprendizaje automático que usaremos en todo el trabajo: el **aprendizaje por refuerzo**. Para ello sintetizamos las ideas que se exponen sobre el tema en [6] y [16].

Existen varios escenarios posibles cuando hablamos de aprendizaje automático o *machine learning*, que difieren en los tipos de datos que reciben los agentes de aprendizaje y la manera en la que se “aprende”. Tradicionalmente tendemos a pensar en **aprendizaje supervisado** cuando hablamos de aprendizaje automático. Este consiste en entrenar un modelo a partir de una serie de datos etiquetados y aprender de esas etiquetas detectando patrones en las mismas. Una vez entrenado, el modelo predice etiquetas para una serie de datos de entradas sin etiquetar y se toman decisiones basadas en esas etiquetas. Primero entrenamos el modelo con datos etiquetados y luego evaluamos el aprendizaje conseguido con datos sin etiquetar.

Nosotros trataremos aquí el aprendizaje por refuerzo, que es el área del aprendizaje automático que se centra en entrenar un modelo que aprende de las propias acciones que toma. Cuando hablamos de aprendizaje por refuerzo las fases de **entrenamiento** y **evaluación** del modelo se entremezclan. Nuestro modelo aprendiz interacciona con el entorno de aprendizaje, llegando incluso a modificarlo, y recibe información sobre el estado del mismo y una recompensa fruto de su acción. Así, el objetivo del modelo aprendiz es **maximizar la recompensa** obtenida tras una serie de acciones. Podemos representar gráficamente la idea de aprendizaje por refuerzo de la siguiente manera:

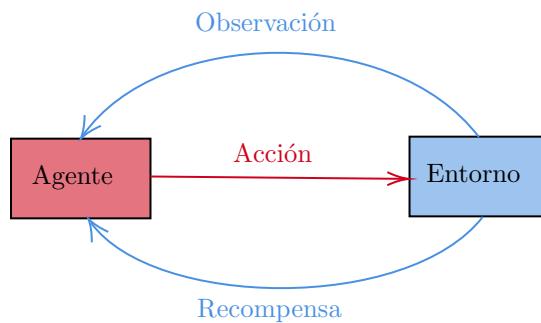


Figura 2.1: Esquema de un escenario de aprendizaje por refuerzo

Así, contamos con un **agente** que recibe una **observación** del **entorno** (esto es, el estado del

entorno en ese momento) y realiza una **acción** en la que interacciona con dicho entorno. Como consecuencia de esta acción, el agente recibe del entorno una **recompensa** específica para la acción realizada y una nueva observación con el estado del entorno tras realizarse la interacción del agente. El agente aprende tras sucesivos intentos, desde una observación inicial hasta un estado final del entorno, que bien puede ser de éxito o de fracaso en la tarea a realizar. El objetivo del agente es tratar de maximizar las recompensas obtenidas y ser capaz de determinar qué acciones tomar en cada momento para alcanzar ese objetivo.

Desde el punto de vista teórico, podemos modelar el problema de aprendizaje por refuerzo como un **proceso de decisión de Markov** (MDP, del inglés *Markov Decision Process*) con los siguientes elementos:

- Un conjunto de **estados** S , que puede ser infinito.
- Un conjunto de **acciones** A que puede realizar el agente, que puede ser también infinito.
- Probabilidad de **transición** (en tiempo t) del estado s al estado s' tomando la acción a , que denotaremos por $P_a(s, s') = \mathbb{P}[s_{t+1} = s' | s_t = s, a_t = a]$.
- **Recompensa** obtenida al pasar del estado s a s' tras tomar la acción a . Lo denotamos por $R_a(s, s')$.

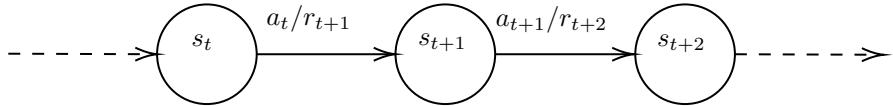


Figura 2.2: Representación de las transiciones entre estados $s \in S$ en un proceso de decisión de Markov, donde en cada estado tomamos una acción $a_i \in A$ y recibimos una recompensa $r_i \in \mathbb{R}$.

2.1.1. Políticas en el aprendizaje por refuerzo

El principal problema que debe resolver un agente de aprendizaje por refuerzo es determinar qué acción tomar en cada estado. Para ello, introducimos el concepto de *política* que nos servirá para decidir la acción que se toma en cada momento.

Definición 2.1 (Política). Dado un problema de aprendizaje por refuerzo, una *política* es una función $\pi : S \rightarrow \Delta(A)$, donde $\Delta(A)$ es un conjunto de distribuciones de probabilidad sobre el conjunto A , esto es, funciones $A \rightarrow [0, 1]$. Para un estado s y una acción a , $\pi(s)(a) = \mathbb{P}[a|s]$ denota la probabilidad de tomar la acción a si nos encontramos en el estado s .

El agente, a través de sucesivas interacciones con el entorno, entrenará una **política** que será capaz de determinar las acciones que tendrán que tomar los agentes para maximizar la recompensa acumulada.

Aunque los agentes sólo reciben por parte del entorno la recompensa inmediata para una acción tomada, la política debe determinar la secuencia de acciones a tomar tras cada observación para que la recompensa final al completar un episodio sea maximizada.

El agente se enfrentará al dilema de elegir entre **exploración** y **explotación**, esto es, explorar estados desconocidos para el agente para tratar de ganar más información sobre el entorno y las recompensas o centrarse en explotar la información de la que ya dispone de pasos anteriores para maximizar las recompensas. Normalmente los algoritmos empleados propiciarán que los agentes vayan alternando ambas opciones.

Para los problemas de aprendizaje por refuerzo que trataremos será necesario definir una **política no estacionaria**, dada por una sucesión de funciones política $\pi_t : S \rightarrow \Delta(A)$, haciendo referencia a la política vigente en cada instante t del problema. Esta política se irá modificando durante el proceso de entrenamiento, actualizándose con los resultados que obtenemos tras interaccionar con el entorno siguiendo la política válida en ese momento.

El objetivo del agente de un problema de aprendizaje por refuerzo será encontrar una política que **maximice la recompensa esperada**, esto es, no sólo la recompensa de la próxima acción sino la suma de estas a largo plazo, pues al final éste es el objetivo del aprendizaje por refuerzo. Así, en cada estado s podemos definirnos el **valor de la política** π en ese estado, $V_\pi(s)$ como el valor esperado para la recompensa total obtenida desde el estado s . Formalmente:

$$V_\pi(s) = \mathbb{E}_{a_t \sim \pi(s_t)} \left[\sum_{t=0}^{\infty} \gamma^t r_{a_t}(s_t, s_{t+1}) | s_0 = s \right]$$

En vista de esta expresión, observamos que el valor de la política viene dado por el valor de la esperanza de la recompensa total tras una serie de pasos comenzando desde el estado s , esto es la recompensa que esperamos obtener seleccionando las acciones de acuerdo con la distribución dada por la política para cada estado. Además, multiplicamos la recompensa en cada paso por γ^t , donde $\gamma \in [0, 1)$ se denomina **factor de descuento** y es un valor constante que se emplea para dar más peso a las recompensas más inmediatas.

Los agentes en cada estado s buscarán una política π con el mayor valor posible de $V_\pi(s)$. Así, diremos que una política π^* es **óptima** si su valor es maximal para todo estado $s \in S$, esto es, para cualquier otra política π y cualquier estado $s \in S$ se tiene que

$$V_{\pi^*}(s) \geq V_\pi(s).$$

De hecho, se puede probar que, si los conjuntos de estados y acciones del problema de decisión de Markov con el que modelamos nuestro problema de aprendizaje por refuerzo son finitos, existe una política que para cualquier estado inicial s es óptima. Además, se prueba también la existencia en esas condiciones de una **política óptima determinista**, esto es, que para cualquier estado s existe una acción a tal que $\pi(s)(a) = 1$. Esta política nos indicaría la secuencia exacta de acciones a tomar que maximizarían la recompensa total del problema.

La siguiente cuestión en la que nos centraremos será elaborar un algoritmo que vaya construyendo esta política óptima durante el proceso de aprendizaje. Los agentes irán interaccionando con el entorno siguiendo una política determinada y obteniendo los valores para las recompensas tras cada acción, usando esta información para mejorar esta política. Diseñaremos un algoritmo iterativo que parte de una política inicial que determine las acciones a realizar sobre el entorno y que “aprenda” con la información recolectada para ir mejorando su valor.

2.2. Algoritmo de Optimización de Política Próxima (PPO)

Presentamos a continuación uno de los algoritmos más eficientes para optimizar la política en el aprendizaje por refuerzo: el **algoritmo de Optimización de Política Próxima**, PPO (del inglés *Proximal Policy Optimization*). Veremos algunos fundamentos teóricos del mismo y también la manera en la que se implementa en *RLlib*. El algoritmo PPO aparece por primera vez en 2017 en el artículo [12]. De esta fuente y otros artículos ([3], [15]) se extrae la información que presentamos a continuación.

El algoritmo PPO se encuadra dentro de lo que denominamos como **métodos de gradiente**. En lo que sigue, denotaremos la política que usa nuestro agente en cada momento por π_θ , donde θ hace referencia a los parámetros que definen la propia función de la política y que se irán actualizando para optimizarla.

Los algoritmos con los que obtendremos la política óptima para nuestro problema de aprendizaje son en realidad **algoritmos de optimización**. Podemos clasificarlos en dos subclases: los algoritmos de **gradiente** y los algoritmos de **región de confianza**.

2.2.1. Algoritmos de gradiente

Estos algoritmos se basan en elegir siempre la dirección en la que se produzca un mayor **descenso en el valor del gradiente** de la función objetivo a optimizar, en este caso la política, y se avanza en esa dirección. Esto hace, en muchos casos, que lleguemos de manera rápida a la solución óptima, pero en ocasiones podemos acabar en estados peores que de los que partíamos. Por ejemplo, imaginemos que estamos escalando una montaña y en cada etapa tenemos que avanzar un número fijo de pasos. Si elegimos ascender en la dirección con mayor pendiente siempre, la lógica nos invita a pensar que llegaremos rápidamente a la cima. Pero, sin embargo, si elegimos una dirección por ser la de mayor pendiente y avanzamos un número de pasos excesivo en esa dirección, podremos caer y aparecer en un nivel incluso más bajo del que partíamos. Desde el punto de vista teórico, estos métodos se basan en calcular un **estimador del gradiente de la política** y usarlo en un algoritmo de descenso de gradiente [1]. El estimador que más se usa en la práctica es el siguiente:

$$\hat{g} = \hat{\mathbb{E}}_t \left[\nabla_{\theta} \log \pi_{\theta}(a_t | s_t) \hat{A}_t \right],$$

donde π_{θ} es la política, \hat{A}_t es un estimador del valor de la función de ventaja en el paso t y $\hat{\mathbb{E}}_t[\dots]$ denota el valor de la media empírica sobre un conjunto de valores en distintos pasos t . La **función de ventaja** mide cómo de buena o mala es la decisión de tomar una acción en un determinado estado, esto es, qué ventaja obtenemos al tomar esta acción. Puede expresarse como

$$A(s, a) = \mathbb{E} \left[r_a(s_0, s_1) + \sum_{t=1}^{\infty} \gamma^t r(s_t, a_t) | s_0 = s, a_0 = a \right] - \mathbb{E} \left[\sum_{t=0}^{\infty} \gamma^t r_{at}(s_t, s_{t+1}) \right],$$

esto es, la ventaja para un estado s y una acción a mide la diferencia entre la recompensa total esperada empezando en el estado s si la primera acción que tomamos es a y la recompensa total esperada partiendo del estado s sin indicar cual es esa primera acción.

En este contexto, se define la **función objetivo** $L^{PG}(\theta) = \hat{\mathbb{E}}_t \left[\log \pi_{\theta}(a_t | s_t) \hat{A}_t \right]$, cuyo gradiente coincide con el valor de \hat{g} que presentábamos antes, y se optimiza su valor mediante uno de estos algoritmos.

2.2.2. Algoritmos de región de confianza

Estos algoritmos, en lugar de buscar optimizar el valor de la función objetivo de manera lineal, definen una **región de confianza** a la que restringimos las soluciones, y se busca optimizar la solución en ese subconjunto. Se itera definiendo nuevas regiones de confianza (de distintos tamaños) hasta converger a una solución óptima. La principal diferencia frente a los algoritmos de gradiente es, que en este caso, el avance no es lineal y que la “distancia” que se avanza en cada iteración no está fijada de antemano, sino que depende de la región de confianza considerada en cada momento. Así, el tamaño de la región de confianza a considerar se reajusta dinámicamente en cada iteración, haciéndose más pequeño si vemos que se producen variaciones considerables en la política. En este tipo de algoritmos debemos limitar de alguna manera cuánto puede variar la política en cada iteración respecto a la anterior. Para ello, introducimos el concepto de **divergencia-KL**, que mide la diferencia existente entre dos distribuciones de probabilidad P y Q y que se define como sigue:

$$KL(P, Q) = \mathbb{E}_x \left[\log \frac{P(x)}{Q(x)} \right].$$

Los algoritmos de región de confianza aplicados para encontrar la política óptima en problemas de aprendizaje por refuerzo impondrán restricciones en el valor de la divergencia-KL para evitar variaciones excesivas de la política en cada iteración.

El **algoritmo TRPO** (*Trust Region Policy Optimization*) [13] se basa en este método y trata de resolver el siguiente problema de optimización, compuesto por una función objetivo y una restricción que imponemos al valor de la divergencia-KL:

$$\begin{aligned} \max_{\theta} \quad & \hat{\mathbb{E}}_t \left[\frac{\pi_{\theta}(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \hat{A}_t \right] \\ \text{s. a.:} \quad & \hat{\mathbb{E}}_t [KL(\pi_{\theta_{old}}(\cdot|s_t), \pi_{\theta}(\cdot|s_t))] \leq \delta, \end{aligned}$$

donde θ_{old} representa el vector de parámetros que definían la política antes de actualizarla en cada iteración. Se prueba que podemos obtener una solución aproximada eficiente para este problema usando un algoritmo de gradiente conjugado después de aproximar linealmente la función objetivo y mediante una función cuadrática la restricción. Además, cuando se presenta este algoritmo, se sugiere modelizar el problema añadiendo una penalización a la función objetivo en lugar de la restricción que teníamos, dando lugar al problema de optimización sin restricciones siguiente:

$$\max_{\theta} \quad \hat{\mathbb{E}}_t \left[\frac{\pi_{\theta}(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \hat{A}_t - \beta KL(\pi_{\theta_{old}}(\cdot|s_t), \pi_{\theta}(\cdot|s_t)) \right], \quad (2.1)$$

para algún coeficiente β . Sin embargo, TRPO usa la versión con la restricción en lugar de esta última con la penalización por la dificultad que supone elegir un valor de β que funcione bien para todos los problemas concretos.

2.2.3. Fundamentos del algoritmo PPO

Para realizar las aproximaciones que hemos mencionado antes y que resolvían el problema de optimización del algoritmo TRPO dando una solución aproximada del problema usamos el **desarrollo de Taylor** de orden dos tanto de la función objetivo como de la restricción. Además, dado que el término de orden dos de la función objetivo va a ser mucho más pequeño que el de la divergencia-KL, podemos ignorarlo. Sin embargo, resolver este problema implicaría calcular la derivada de segundo orden de la función divergencia-KL e invertir la matriz resultante, lo cual es un cálculo bastante costoso desde el punto de vista computacional. Este problema se trata de abordar de dos maneras:

- Aproximar los cálculos que impliquen a la segunda derivada y su inversa para reducir la complejidad.
- Hacer que la solución aproximada implique sólo el cálculo de derivadas de primer orden (como el descenso de gradiente) añadiendo restricciones al problema.

En el algoritmo TRPO se toma la primera solución, mientras que la novedad de PPO es que su aproximación se parece más la segunda idea expuesta. Así, aplicaremos métodos que sólo impliquen la primera derivada como el descenso de gradiente, pero añadiendo restricciones que fuercen a que la optimización siga realizándose dentro de una región de confianza determinada.

PPO con penalización KL adaptada

En esta aproximación resolveremos el problema del algoritmo TRPO pero sustituyendo la restricción del problema de optimización por una **penalización** en la función objetivo, tal y como teníamos en la expresión (2.1). El valor de β controla la penalización que introducimos al valor de la función objetivo

y que iremos ajustando dinámicamente. Así, si el valor de la divergencia-KL entre la nueva política π_θ y la política antigua $\pi_{\theta_{old}}$ aumenta en exceso (fijamos un valor δ que marque el valor máximo de la divergencia-KL que toleramos) disminuimos el valor de β . Simplificando mucho las cosas, un esquema de cada iteración de actualización de la política en un algoritmo PPO con penalización KL sería el siguiente:

1. Computar varias etapas del algoritmo *minibatch SGD* optimizar la función objetivo con penalización KL:

$$L^{KLPEW}(\theta) = \hat{\mathbb{E}}_t \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \hat{A}_t - \beta KL(\pi_{\theta_{old}}(\cdot|s_t), \pi_\theta(\cdot|s_t)) \right] \quad (2.2)$$

2. Calcular $d = \hat{\mathbb{E}}_t [KL(\pi_{\theta_{old}}(\cdot|s_t), \pi_\theta(\cdot|s_t))]$. Ahora ajustamos el valor de β dependiendo del valor de d :

- Si $d < \frac{\delta}{1.5}$, $\beta \leftarrow \beta/2$
- Si $d > 1.5 \times \delta$, $\beta \leftarrow 2 \times \beta$.

El valor de β actualizado se usa para la siguiente iteración. Los valores 1.5 y 2 se eligen heurísticamente y aunque el valor de β inicial es un hiperparámetro del algoritmo este no se ve afectado por una mala elección del mismo, pues rápidamente se ajusta su valor.

PPO con objetivo sustituto recortado (*clipped surrogate objective*)

Para un instante de tiempo t en el que nos encontramos en un estado s_t y para una determinada acción a_t denotamos por $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$. Observamos que $r_t(\theta_{old}) = 1$. Con esta notación, la función objetivo del problema de optimización del algoritmo TRPO se puede expresar como:

$$L(\theta) = \hat{\mathbb{E}}_t [r_t(\theta) \hat{A}_t] \quad (2.3)$$

Sin imponer ninguna restricción, la maximización de esta función $L(\theta)$ podría dar lugar a incrementos muy grandes de la política entre iteraciones. Para evitar esto, en esta aproximación lo que se va a hacer es penalizar los cambios en la política que hagan que el valor de $r_t(\theta)$ se aleje de 1. Así, se propone la función objetivo

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t \left[\min\{r_t(\theta) \hat{A}_t, \text{clip}(r_t(\theta), 1 - \varepsilon, 1 + \varepsilon) \hat{A}_t\} \right], \quad (2.4)$$

donde ε es un hiperparámetro del algoritmo (habitualmente $\varepsilon = 0.2$). La función $\text{clip}(r_t(\theta), 1 - \varepsilon, 1 + \varepsilon)$ hace que el valor de $r_t(\theta)$ se mantenga siempre en el intervalo $[1 - \varepsilon, 1 + \varepsilon]$, esto es,

$$\text{clip}(r_t(\theta), 1 - \varepsilon, 1 + \varepsilon) = \begin{cases} 1 - \varepsilon & \text{si } r_t(\theta) \leq 1 - \varepsilon \\ 1 + \varepsilon & \text{si } r_t(\theta) \geq 1 + \varepsilon \\ r_t(\theta) & \text{en otro caso} \end{cases}$$

Lo que hacemos, es seleccionar el mínimo entre el resultado de aplicar la función *clip* al radio de probabilidades entre la política nueva y la antigua y el valor de ese radio. Esto provoca un comportamiento en el valor de la función objetivo como el mostrado en la figura 2.3. Así, el algoritmo se basa en maximizar el valor de esta función objetivo, tarea que se puede llevar a cabo con algoritmos de descenso de gradiente de manera sencilla.

Frente a otros algoritmos que se pueden aplicar para la optimización de la política en el aprendizaje por refuerzo, PPO ofrece simplicidad y eficiencia, en un algoritmo con el que, en la mayoría de los casos, se obtienen muy buenos resultados. Cuando más adelante describamos las utilidades de la biblioteca de aprendizaje por refuerzo *RLlib* detallaremos algunos aspectos más técnicos de su implementación y cómo se puede optimizar su rendimiento paralelizando partes de su ejecución.

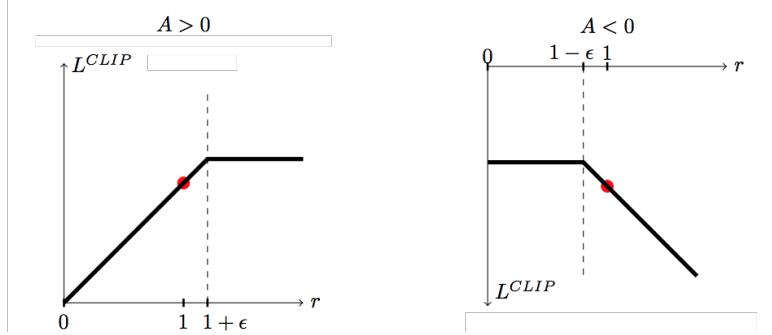


Figura 2.3: Valores de la función objetivo para distintos valores del radio de probabilidades en un instante t concreto. Obsérvese el efecto de la función *clip* en los casos en los que el estimador de ventaja es positivo o negativo. Esta imagen se obtiene del *paper* en el que se presenta el algoritmo PPO [12].

2.3. Redes Neuronales de Convolución

Las **redes neuronales de convolución** (CNN, del inglés *Convolutional Neural Network*) son un algoritmo de aprendizaje profundo que toma **imágenes** como datos de entrada, extrae información de ellas y es capaz de diferenciar unas de otras para realizar predicciones. Describiremos de manera breve cómo y por qué funcionan, siguiendo en parte lo expuesto en [8], y también como las integraremos en nuestro escenario de aprendizaje por refuerzo.

Una red neuronal está formada por una serie de **capas**, cada una con un número determinado de neuronas, que reciben datos, los multiplican por una serie de pesos y pasan la información a las neuronas de siguientes capas a las que están interconectadas. Una imagen no es más que una **matriz de píxeles**, esto es, una matriz cuyos valores representan los píxeles de la imagen. Podríamos reordenar los elementos de esta matriz dando lugar a un vector unidimensional que sirviese como entrada de una red neuronal al uso. Sin embargo, haciendo esto estaríamos perdiendo mucha información sobre dependencias espaciales de muchos de los elementos de las imágenes, por lo que debemos ser capaces de procesar la imagen en su formato habitual como matriz n-dimensional de píxeles (cada capa de color es una matriz bidimensional y podemos tener varias capas).

Los elementos de entrada de una red neuronal de convolución vendrán dados por tensores (vectores) de **cuatro dimensiones** (número de entradas, ancho de la imagen, alto de la imagen y número de canales de color de la misma). Normalmente, el número de entradas será 1.

El elemento fundamental de las redes neuronales de convolución son las **capas de convolución**, que dan nombre al algoritmo. En estas capas se extraen las características fundamentales de los datos de entrada y se crea con ellas lo que denominaremos **mapa de características**. Para ello, las capas de convolución cuentan con un **núcleo** o **filtro** (en inglés *kernel*), que viene dado por una matriz bidimensional de menor tamaño que la imagen y cuyos valores son parámetros entrenables. Tendremos un filtro para cada canal de color. La función de este filtro es extraer las características relevantes de las distintas partes de la imagen. Para ello, es necesario definir las dimensiones del filtro y lo que queremos que se desplace el filtro cada vez que lo movamos (*stride*). Supongamos que tenemos una entrada de tamaño $(1 \times \text{ancho} \times \text{alto} \times \text{canales})$, un *kernel* de tamaño $(\text{dim} \times \text{dim})$ (con $\text{dim} < \min(\text{alto}, \text{ancho})$) y un desplazamiento con valor s . Comenzamos colocando el filtro sobre los píxeles correspondientes a la esquina superior izquierda del primer canal de color de la imagen y multiplicamos cada valor en el *kernel* por el correspondiente en esa posición en la imagen, sumando todos esos valores y almacenando el resultado. Repetimos la misma acción para el resto de canales y sumamos los valores obtenidos, siendo el valor resultante el primer elemento del mapa de características que obtendremos como salida

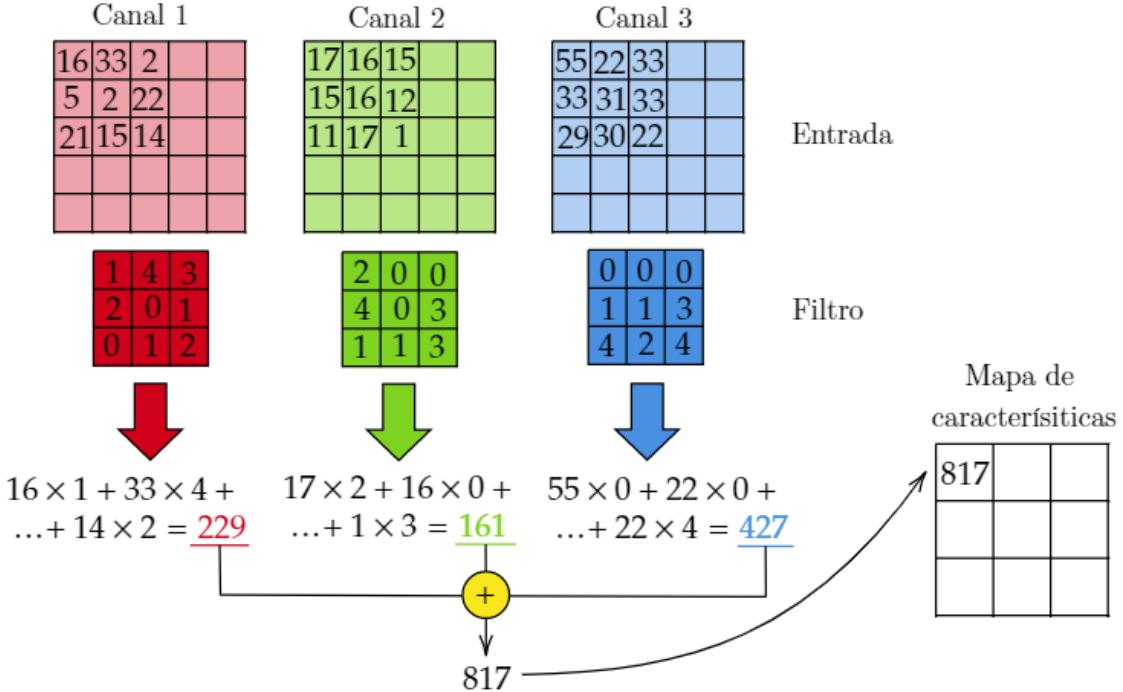


Figura 2.4: Obtención de la primera característica del mapa en una imagen con 3 canales de color y un filtro de dimensiones 3×3 .

para esta capa. Volviendo al primer canal de color, ahora desplazamos el filtro s píxeles a la derecha y repetimos la operación. Una vez no tengamos más píxeles por la derecha hacia los que desplazar el filtro (ya habremos completado la primera fila del mapa de características) volvemos a la primera posición en la que lo colocamos y lo desplazamos s píxeles hacia abajo, comenzando nuevamente el proceso anterior hasta completar la segunda fila del mapa de características. Esto es, vamos recorriendo cada capa de la imagen de izquierda a derecha y de arriba a abajo obteniendo la “característica” de cada posición que ocupa el filtro (ver figura 2.4).

Además, hay una parámetro más que debemos indicar a las capas de convolución y que determinará el tamaño de la salida: el **relleno** o *padding*. Este parámetro puede tomar dos valores: **same** y **valid**. Con el primero de ellos, forzamos a que el mapa de características tenga las mismas dimensiones que la imagen (en ancho y alto) si el *stride* fuese 1, ampliando por sus cuatro lados con filas y columnas de ceros las capas de la imagen, para que el filtro pueda desplazarse hasta obtener tantas características como píxeles tenía la imagen original. Con *valid padding*, la salida tendrá por dimensión las veces que se haya podido mover el filtro con el *stride* s en esa dirección sin salirse de la imagen y en general el tamaño del mapa de características será menor que el de la imagen de entrada. Un parámetro adicional que reciben las capas de convolución es el número de filtros a aplicar, esto es, indicamos cuantos filtros con las mismas dimensiones vamos a aplicar a la imagen, obteniendo para cada uno de ellos un mapa de características. Con este parámetro indicamos también la cuarta dimensión de la salida de la capa. Así, en general, si tenemos unos datos de entrada de tamaño $(n \times w \times h \times c)$ para una capa de convolución con un *kernel* de tamaño $(k_1 \times k_2)$, un *stride* s y aplicamos m filtros a la imagen, la salida de la capa tendrá un tamaño, dependiendo del tipo de *padding* de:

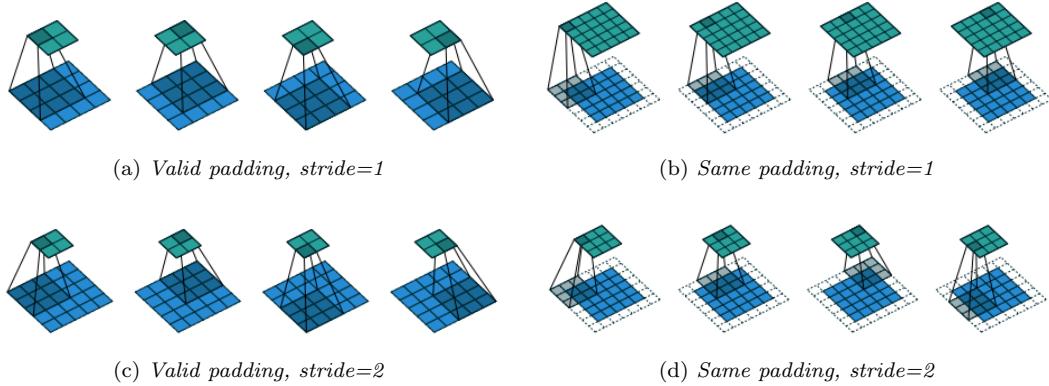


Figura 2.5: Diferentes configuraciones de la capa de convolución, con *valid* y *same padding* y *strides* de 1 y 2. Las imágenes se toman de [2].

- **Same padding:** $\left(n \times \left\lceil \frac{w}{s} \right\rceil \times \left\lceil \frac{h}{s} \right\rceil \times m \right)$.
- **Valid padding:** $\left(n \times \left\lceil \frac{w - k_1 + 1}{s} \right\rceil \times \left\lceil \frac{h - k_2 + 1}{s} \right\rceil \times m \right)$

En cuanto al número de **parámetros entrenables** de la capa (pesos), este vendrá dado por el número total de filtros que tengamos multiplicado por el tamaño de cada filtro. En el caso general, tenemos $c \times m \times k_1 \times k_2$ parámetros entrenables. Además, a la salida de cada filtro se le suma también un vector de sesgo, que tiene tamaño m y cuyos valores son parámetros entrenables también, por lo que en realidad esta cantidad viene dada por $c \times m \times k_1 \times k_2 + m$.

Además de las capas de convolución las redes neuronales cuentan con otro tipo de capas:

- **Capas de pooling:** Estas capas se encargan de reducir las dimensiones de las salidas de las capas de convolución, reduciendo la información que nos ofrece esta salida. Así, se toma la salida de la capa de convolución y convertimos la porción de imagen cubierta por el *kernel* en un único valor, que puede ser la media (*average pooling*) o el máximo (*max pooling*) de los valores presentes en esa porción de la salida. Las capas de *pooling* suelen aparecer entre dos capas de convolución.
- **Capas completamente conectadas (*fully connected*):** Estas capas, habituales en redes neuronales, conectan todas las neuronas de entrada con todas las de salida y en las redes neuronales de convolución suelen aparecer como las últimas capas de la red, para una vez hayamos extraído las características de la imagen podemos establecer relaciones entre ellas que faciliten y precisen la predicción final sobre los datos.

Las redes neuronales de convolución funcionan extraordinariamente bien como modelos de aprendizaje automático con imágenes de entrada, y en general sirven para todos aquellos datos de entrada en los que queramos capturar **dependencias espaciales** entre los elementos, que convirtiendo la imagen en un *array* de datos sería imposible capturar. Más adelante, veremos que las redes neuronales de convolución que emplearemos no reciben exactamente imágenes con 3 canales de color, sino que lo que reciben es una pila de 4 imágenes con una capa de color (escala de grises), pero que a efectos prácticos se corresponde con una imagen con cuatro capas. Al fin y al cabo aquí los colores no son tan importantes y nos centramos más en las relaciones existentes entre las imágenes que nos devuelve el entorno como observaciones en iteraciones sucesivas.

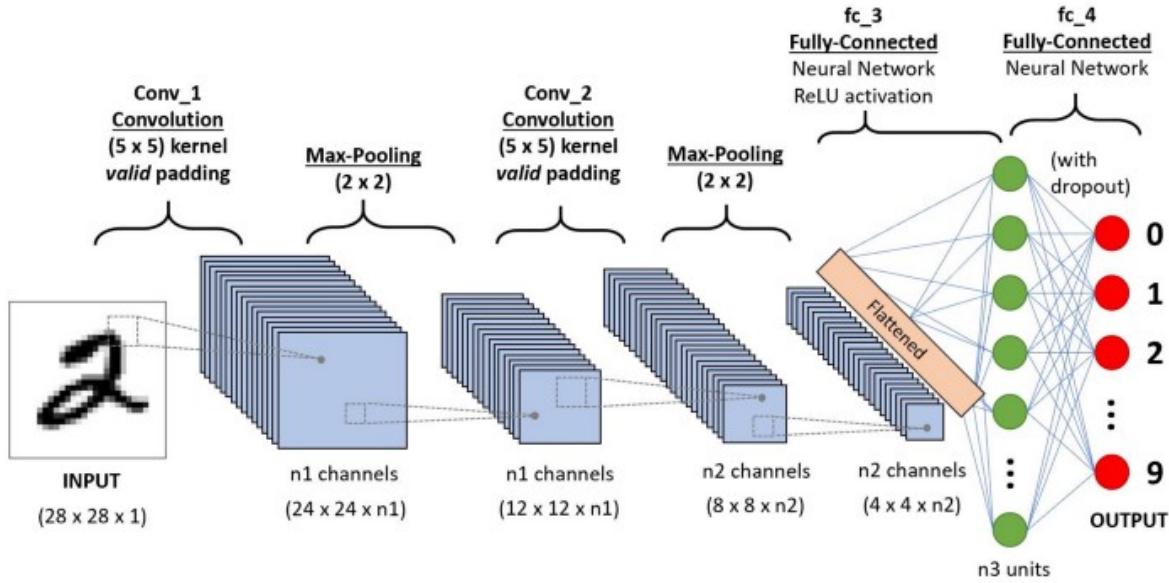


Figura 2.6: Esquema general de una red neuronal de convolución y sus capas. Imagen obtenida de [8].

2.3.1. Redes de convolución y aprendizaje por refuerzo

Dentro del marco de aprendizaje por refuerzo, las redes neuronales modelizarán la **política** y su **función de valor**. Así, mantendremos en nuestro modelo dos redes neuronales que reciben la misma entrada: una observación del entorno. Una de ellas tiene tantos valores de salida como acciones posibles pueda realizar el agente en el entorno, indicando el valor de cada salida la probabilidad de realizar esa acción en el estado representado por la observación ($\pi(a|s)$), es lo que llamamos **red de política** (*policy network*). La segunda red tiene una única salida, que indica el valor de la política actual que a la larga queremos maximizar y recibe el nombre de **red de valor** (*value network*). Lo que en realidad hacemos es entrenar una red neuronal con una serie de parámetros θ (los que decíamos que determinaban la política en el algoritmo PPO) que defina la política en ese momento y su valor, con el objetivo de encontrar una política óptima que maximice la recompensa esperada. Los modelos que diseñemos aplicarán los mismos filtros de convolución a las entradas para cada una de las dos redes, diferenciándose sólo en la última capa (*fully connected*) que tendrá distinto número de salidas en cada una de ellas.

2.4. Tensorflow

Tensorflow es una biblioteca de código abierto para **computación numérica** que permite desarrollar aplicaciones de *machine learning* de manera rápida y sencilla. Expondremos brevemente algunas detalles de su funcionamiento y en la siguiente sección veremos como se integra dentro de *RLlib*. La exposición siguiente se extrae de [18] y de la propia documentación¹. Desarrollada por Google, usa *Python* para proporcionar una API sencilla para el desarrollo de aplicaciones y C++ para la ejecución de las mismas. La idea fundamental de *Tensorflow* son los **grafos de datos**. Cada **nodo** en el grafo representa una operación matemática y cada arista entre nodos es un *array multidimensional* llamado

¹<https://www.tensorflow.org/>

tensor. Estos nodos y tensores son objetos de *Python* pero las operaciones se ejecutan en C++, que proporciona un mayor rendimiento de cálculo. Con *Tensorflow* podemos modelar, entrenar y ejecutar inferencias sobre modelos de aprendizaje profundo como redes neuronales, usando para ello la API de *keras*.

2.4.1. Keras

Keras² es la API de alto nivel de *Tensorflow* para crear y entrenar modelos de aprendizaje profundo (redes neuronales). Algunas de sus ventajas son la simplicidad de su interfaz, que es bastante accesible al usuario, y la facilidad para configurar la creación de modelos y para extender modelos previamente creados. Podemos crear modelos de aprendizaje automático, como redes neuronales, de manera sencilla con la idea del grafo de *Tensorflow*, y así es como lo hace *keras*. Los nodos se corresponden con cada una de las capas de la red y las aristas con el flujo de datos entre capas. Cada nodo representa las operaciones que hay que hacer sobre los datos de entrada, modeladas por una serie de pesos entrenables, cuyos valores se van ajustando durante las iteraciones de entrenamiento.

2.4.2. Tensorflow Lite

Tensorflow Lite es un *framework* de aprendizaje profundo que permite ejecutar modelos previamente entrenados en *Tensorflow* en dispositivos en los que se pueden **optimizar costes** de inferencia (dispositivos móviles que usen *Android* o *iOS*, sistemas como *Raspberry Pi* o aceleradores como la TPU Google Coral). Los modelos de *Tensorflow* se pueden convertir en modelos de *Tensorflow Lite*, que tienen un formato especial que permite introducir optimizaciones en los modelos. Una de las ventajas que nos ofrecerá *Tensorflow Lite* será la **cuantización** de modelos, que en la sección 2.8 explicaremos.

2.5. Ray y RLlib

Ray es un *framework* de código abierto que pretende crear una API universal para **aplicaciones distribuidas**. Este *framework* está constituido por varias bibliotecas que ofrecen funcionalidades muy diversas. Nosotros nos centraremos en la biblioteca de *Python* *RLlib* (*Reinforcement Learning Library*) que da soporte a aplicaciones relacionadas con el aprendizaje por refuerzo.

Ray cuenta con su “núcleo” (*Ray Core*) que gestiona la planificación de tareas de manera distribuida y los recursos disponibles y sobre él se desarrollan bibliotecas muy variadas, entre las que se encuentra la ya mencionada *RLlib*. *Ray* proporciona primitivas simples para construir estas aplicaciones distribuidas y para poder parallelizar de manera sencilla código escrito para una sola máquina. La API de *Ray* está disponible en *Python* y *Java* y experimentalmente en C++. Nosotros usaremos la API de *Python* a lo largo de todo este trabajo.

Frente a otros *frameworks* y bibliotecas existentes para la computación distribuida, *Ray* cuenta con la ventaja de la facilidad que ofrece para parallelizar código que originalmente no se escribió con esta intención. *Ray* transforma un código compuesto por clases y funciones en una serie de actores que se realizan tareas, permitiendo así la parallelización. Esta manera de crear los actores y las tareas basándose en la estructura de clases y funciones del código simple aporta a *Ray* esta ventaja que mencionábamos antes.

Aunque *Ray* proporciona soporte para escalar la ejecución a estructuras distribuidas con varios nodos, nosotros explotaremos su funcionalidad en **una sola máquina**, llevando a cabo esa parallelización a nivel de recursos de la propia máquina.

²<https://www.tensorflow.org/guide/keras>

2.5.1. RLlib

Como ya hemos anticipado antes, la biblioteca *RLlib*, construida sobre el núcleo de *Ray* proporciona al usuario infinidad de recursos para el desarrollo de aplicaciones que requieran algoritmos, técnicas o modelos propios del **aprendizaje por refuerzo**, con la ventaja frente a otros *frameworks* o bibliotecas de permitir el desarrollo de estas aplicaciones (que suelen ser costosas desde el punto de vista computacional) de manera distribuida. Internamente, *RLlib* hace uso de *Tensorflow*, *Tensorflow Eager* o de *Pytorch* (esto es elección del usuario) para modelar la estructura completa del problema de aprendizaje e integrar modelos propios de estos *frameworks*. En este trabajo se usará *RLlib* conjuntamente con *Tensorflow*, creándose para ello grafos que contendrán como subgrafos el modelo de *keras* y una serie de nodos para realizar las operaciones específicas de los algoritmos. *RLlib* estructura su funcionalidad en torno a cuatro conceptos: agentes, políticas, modelos y entornos.

Para modelar un escenario de aprendizaje por refuerzo crearemos un **agente**, que llevará asociado un algoritmo para el aprendizaje y la actualización de las políticas. *RLlib* implementa varios de los algoritmos más usados en el contexto del aprendizaje por refuerzo³. Este agente contará con dos elementos fundamentales: un **entorno** y una **política**. *RLlib* ofrece soporte a gran variedad de entornos, integrando perfectamente entornos provenientes de otras bibliotecas (como *Gym*) o proporcionando al usuario la API necesaria para la creación de sus propios entornos. Las políticas, por su parte, son las clases que definen como el agente interacciona con el entorno. Esta política es la que define el algoritmo de aprendizaje. Aunque podemos implementarlas haciendo uso de elementos de cualquier *framework*, *RLlib* proporciona los medios para definir de manera organizada políticas en *Tensorflow* y *Pytorch*. Las políticas cuentan con un **modelo** (generalmente una red neuronal) que depende del entorno con el que se produzcan las interacciones y que será el que se emplee durante el proceso de aprendizaje para llevar a cabo la actualización de la política y la selección de las acciones para maximizar el valor de la recompensa final. La estructura completa de un agente se puede ver en la figura 2.7. El entorno (u opcionalmente una fuente externa) nos proporciona observaciones que son preprocesadas y filtradas (en algunos casos estas dos fases no tienen efecto alguno sobre la observación) y con las que se alimenta un modelo (red neuronal). Las salidas de este modelo (*logits*) se corresponden con la probabilidad de tomar cada una de las acciones (por tanto, hay una salida por cada acción) y con esos valores se crea una distribución de probabilidad que determina la siguiente acción a tomar y que actualiza el valor de una función de pérdida que se usa para mejorar la política. Esta siguiente acción a tomar (que dependerá de varios parámetros, entre ellos si estamos evaluando o entrenando el modelo) se ejecuta sobre el entorno, iniciando nuevamente el proceso con la obtención de una nueva observación. Obsérvese además cómo la mayoría de estos elementos se pueden personalizar por el usuario, permitiéndole dar sus propias definiciones de los mismos.

RLlib cuenta además con funcionalidad para realizar de manera sencilla tanto el **entrenamiento** de estos agentes (que como ya hemos dicho abarcan todo el escenario de aprendizaje) como la **inferencia** sobre los mismos para evaluar su rendimiento. De hecho, permite combinar ambas fases y es posible ejecutar entremezcladas durante el entrenamiento iteraciones de evaluación del estado del modelo únicamente. *Ray* pone a nuestra disposición el script `rollout.py`⁴ para las inferencias y la función `train()`⁵ que podemos ejecutar sobre los objetos de la clase agente para realizar una iteración de entrenamiento sobre ese agente.

2.5.2. Algoritmo PPO en RLlib

La implementación del algoritmo de Optimización de Política Próxima aprovecha las ventajas que ofrece *Ray* como *framework* de programación distribuida para hacer que la ejecución del algoritmo

³<https://docs.ray.io/en/master/rllib-algorithms.html>

⁴<https://github.com/ray-project/ray/blob/master/rllib/rollout.py>

⁵<https://github.com/ray-project/ray/blob/master/rllib/agents/trainer.py#L588>

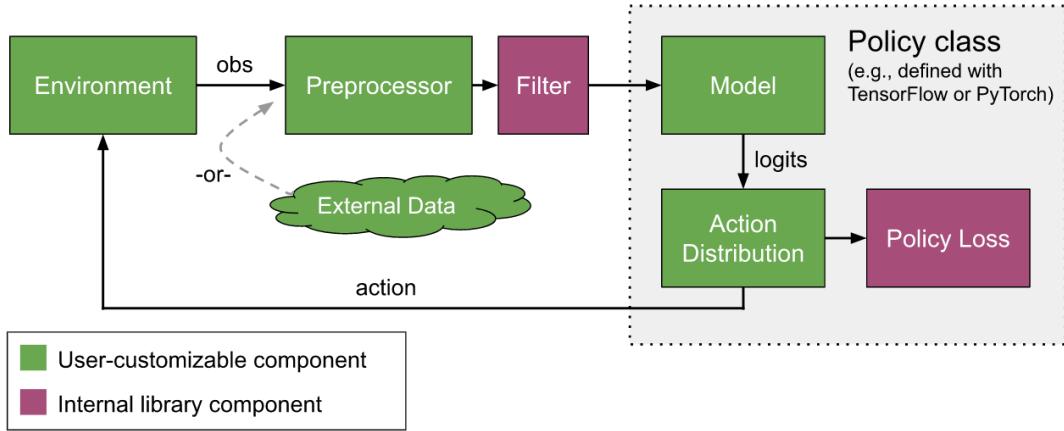


Figura 2.7: Esquema general de la organización de un agente de *RLlib* que modela un escenario completo de aprendizaje por refuerzo. La imagen se toma de la propia documentación de *RLlib* (<https://docs.ray.io/en/master/rllib.html#customization>).

durante el proceso de entrenamiento de nuestros modelos sea lo más eficiente posible. Como vimos, en cada iteración del algoritmo se realizaban una serie de acciones dadas por la política actual. Con esos datos se obtenía el valor del estimador de la ventaja obtenida con esa política, y una vez teníamos esta estimación se ejecutaban una serie de iteraciones del algoritmo de *minibatch SGD* para obtener los nuevos parámetros θ que actualizan la política, dependiendo de la versión de la función objetivo utilizada (penalización KL o *clip*). Así, la implementación de este algoritmo en *RLlib* ofrece la posibilidad de **escalar** ambas fases de cada iteración sobre los recursos disponibles. Tendremos así dos elementos diferenciados: los **rollout workers** y el **driver**.

Los *rollout workers* se encargan de **interaccionar con el entorno** y recoger los valores necesarios para la fase de actualización de la política. Esto puede llevarse acabo de manera paralela por varios *workers*, cada uno de los cuales interacciona con su propio entorno siguiendo la política válida en ese momento y recoge los resultados de esa interacción (recompensas tras episodios). El total de los *workers* tiene que realizar **train_batch_size** pasos de interacción con el entorno, que se dividen en conjuntos de **rollout_fragment_length** pasos que son recogidos por cada *worker*. Esto es, el algoritmo necesita información de **train_batch_size** pasos para ejecutar la fase de actualización de la política, pero esta tarea se divide entre los *workers*, cada uno de los cuales va ejecutando grupos de **rollout_fragment_length** iteraciones hasta que en total se llega a la cantidad de **train_batch_size**. *RLlib* permite distribuir estos *workers* usando diferentes recursos para que puedan realizar acciones en paralelo, usando por defecto una CPU para cada uno.

Por su parte, el *driver* recibe de manera síncrona esta información de los *workers* y la concatena en un único conjunto de tamaño **train_batch_size**. Una vez tiene los datos listos, puede ejecutar una serie de iteraciones de **descenso de gradiente estocástico** (SGD) para obtener una política actualizada. Para realizar esta tarea también *RLlib* ofrece la posibilidad de escalar para acelerar su ejecución. Una de las principales ventajas que ofrece es la posibilidad de introducir una **GPU** para realizar estos cálculos intensivos desde el punto de vista computacional, de hecho, cuenta con el soporte necesario para distribuir esta tarea entre varias GPUs si contamos con ellas. Incluso, en caso de disponer sólo de una GPU tanto los *workers* como el *driver* pueden compartirla, aunque en realidad nunca hacen uso simultáneo de ella (sí que lo harán los distintos *workers*). Una vez actualizada la política y su valor para la próxima iteración en el *driver* se informa de estos nuevos valores a los *workers* (en realidad los *workers* reciben los nuevos pesos de la red neuronal con los que actualizar el modelo).

RLLib permite configurar varios de los parámetros propios del algoritmo, como el valor límite del coeficiente KL, el valor de ϵ para la función *clip* o el número de iteraciones de SGD a ejecutar por cada etapa de entrenamiento. Se puede consultar la lista de parámetros y sus valores por defecto en el propio *script* en el que se implementa el agente PPO dentro del código fuente de *RLLib*⁶.

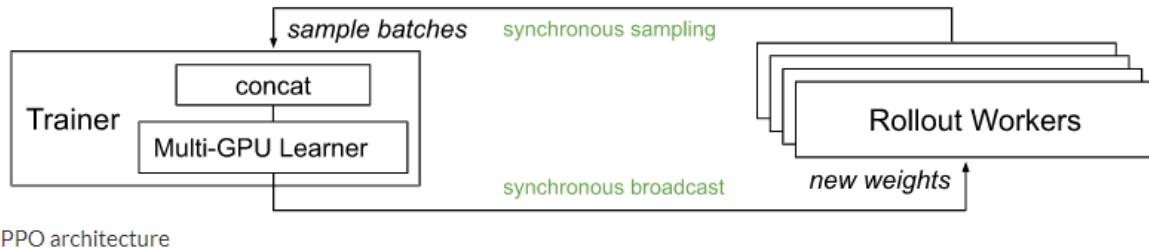


Figura 2.8: Esquema de la arquitectura del algoritmo PPO en *RLLib*. Los *rollout workers* recogen experiencias en paralelo que el *driver* usa para actualizar el valor de la política, que se actualiza también en los *workers* para volver a iniciar todo el proceso. La imagen se toma de la propia documentación de *RLLib* (<https://docs.ray.io/en/master/rllib-algorithms.html#proximal-policy-optimization-ppo>).

Podemos crear un agente que durante su fase de aprendizaje ejecute el algoritmo PPO de la siguiente manera:

```

1 import ray
2 import ray.rllib.agents.ppo as ppo
3
4 ray.init()
5 config = ppo.DEFAULT_CONFIG.copy()
6 agent=ppo.PPOTrainer(env='Pong-v0', config=config)
  
```

2.6. Entornos Gym

*Gym*⁷ es una biblioteca de código abierto de *Python* desarrollada por *OpenAI* y diseñada para desarrollar y comparar algoritmos de aprendizaje por refuerzo. *Gym* incluye una colección de **entornos** muy variada que podemos usar para la interacción con agentes que aplican un algoritmo de aprendizaje concreto. La interacción con estos entornos sigue la idea general del aprendizaje por refuerzo (figura 2.1): el agente puede realizar una serie de acciones para cada entorno (las acciones disponibles dependerán de cada entorno concreto) y como resultado el entorno nos devuelve cuatro valores, entre los que se encuentran la recompensa y una nueva observación para continuar con las interacciones. Los distintos entornos que nos ofrece *Gym* se pueden clasificar en cuatro grupos:

- **Problemas clásicos de control y entornos basados en texto:** entornos simples que resuelven tareas sencillas. Encontramos aquí ejemplos de entornos muy comunes en la bibliografía relativa al aprendizaje por refuerzo.
- **Problemas algorítmicos:** estos entornos modelan problemas que se resuelven con algún algoritmo conocido, como invertir secuencias o sumar números de varias cifras. La idea detrás de

⁶<https://github.com/ray-project/ray/blob/master/rllib/agents/ppo/ppo.py>

⁷<https://gym.openai.com/>

estos entornos es ver si nuestro agente es capaz de aprender el algoritmo sólo con la información de los ejemplos.

- **Atari**: entornos que simulan videojuegos clásicos de la marca *Atari*.
- **Robots en 2D y 3D**: el agente controla un robot en una simulación basada en el motor físico *MuJoCo*⁸.

Podemos crear un entorno concreto haciendo uso de la función `gym.make()` e indicando el identificador de este entorno. Una vez creado, obtenemos la observación de su estado inicial usando la función `reset()`. Para que el agente realice acciones sobre el entorno haremos uso de la función `step()`, a la que indicamos el identificador de la acción a realizar (un entero dentro de un rango de valores determinado) y que nos devuelve información sobre el efecto de esa acción en el entorno:

- **observation**: objeto representando una nueva observación del entorno tras realizar una acción sobre él. Su tipo dependerá del entorno concreto.
- **reward**: recompensa obtenida al realizar una acción determinada sobre el entorno y en un estado concreto. Viene dada siempre por un valor en punto flotante y el rango de valores que puede tomar depende también de cada entorno.
- **done**: valor booleano que indica si debemos resetear el entorno porque hemos completado un episodio (por ejemplo hemos acabado una partida de un juego de *Atari* porque nos hemos quedado sin vidas o porque hemos alcanzado una puntuación máxima).
- **info**: diccionario con información adicional para debuguear o para entender cómo el agente está aprendiendo, pero que en ningún caso debemos utilizar para el propio proceso de aprendizaje del agente.

Además, cada entorno tiene asociados un **espacio de acciones** (`action_space`) y un **espacio de observaciones** (`observation_space`) que definen los valores que podemos tomar como acciones (y que determinarán las salidas de la red neuronal con la que modelemos el agente) y la forma de las observaciones que usará el agente para aprender.

De todos los entornos disponibles vamos a seleccionar uno de ellos para nuestro proceso de aprendizaje y, dado que usaremos redes neuronales de convolución para nuestro agente, las observaciones del entorno serán imágenes que representen el estado del mismo. Los entornos de la categoría *Atari* satisfacen este requisito, ya que sus observaciones vienen dadas como imágenes RGB (en forma de *arrays* de *numpy*⁹) que representan el estado de la pantalla del juego en cada momento y en las que el agente que entrenamos simula el comportamiento de un jugador que interacciona con este videojuego, realizando las acciones necesarias para obtener la máxima puntuación. De entre todos ellos elegimos el denominado **Pong-v0**¹⁰, que simula una partida de tenis de mesa en la que el agente controla a uno de los dos jugadores y la “máquina” juega contra él, y el objetivo es conseguir más puntos que el otro jugador. Más adelante, veremos que no interaccionamos directamente con este entorno, sino que partiendo de él definiremos una serie de envoltorios que actuarán como preprocesador de las imágenes para simplificarlas y que sea más fácil aprender de ellas, pero la manera de realizar acciones o de obtener información del entorno es la misma, sólo cambia el formato de las imágenes que obtenemos como observaciones. Mostramos a continuación un ejemplo de cómo sería una interacción muy simple con un entorno *Gym* hasta completar un episodio y suponiendo que nuestro agente (que es quien determina qué acción tomar) elige siempre una acción aleatoria dentro del conjunto de acciones posibles:

⁸<http://www.mujoco.org/>

⁹<https://numpy.org/doc/stable/reference/generated/numpy.array.html>

¹⁰<https://gym.openai.com/envs/Pong-v0/>

```

1 import gym
2 env=gym.make('Pong-v0')
3 observation = env.reset()
4 done = False
5 total_reward= 0.0
6 while not done:
7     env.render() # Show in screen the environment state
8     action = env.action_space.sample() # Take a random action
9     observation, reward, done, info = env.step(action)
10    reward_total+=reward

```

RLLib integra en sus algoritmos los entornos de *Gym* y la interacción con ellos, añadiendo funcionalidad específica para el manejo de estos entornos, por lo que sólo tendremos que preocuparnos de elegir el entorno adecuado.

2.7. TPU y Google Coral

Uno de los objetivos fundamentales de este trabajo será evaluar el rendimiento de la inferencia sobre redes neuronales profundas en **aceleradores hardware de propósito específico**. Para ello, aceleraremos la inferencia de modelos haciendo uso de una **TPU**.

Unidades de Procesamiento Tensorial

Una **Unidad de Procesamiento Tensorial** o TPU (del inglés *Tensor Processing Unit*) es un circuito integrado de aplicación específica o ASIC (del inglés *Application-Specific Integrated Circuit*) para acelerar aplicaciones de inteligencia artificial. *Google* ofrece estas TPUs tanto dentro de su infraestructura *Cloud* (*Google Cloud*) como integradas en dispositivos hardware (*Google Coral*). Estas unidades usan el software de *Tensorflow* (desarrollado también por *Google*), por lo que tendremos que desarrollar nuestros modelos dentro de este **framework**, lo cual acabamos de ver que no es ningún problema, pues *Ray* integra perfectamente modelos de *Tensorflow* dentro de su funcionalidad. En concreto, para poder hacer uso de la versión *edge* de la TPU (en un sistema local, fuera de *Google Cloud*) tendremos que trabajar con *Tensorflow Lite*. La TPU además trabaja sólo con datos representados en forma de enteros de 8 bits (*int8*), por lo que será necesario que los modelos que vayamos a ejecutar sobre la TPU tengan esta característica. La ventaja fundamental de estos aceleradores es que pueden realizar un gran volumen de operaciones en poco tiempo con un **consumo de energía bastante reducido**. En concreto, la TPU que emplearemos es capaz de realizar 4 trillones de operaciones (*tera-operaciones*) por segundo (TOPS), usando 0.5 W de potencia para cada TOPS¹¹.

Ventajas del uso de una TPU

Siguiendo lo expuesto en [11], veremos cuáles son las ventajas de una TPU frente a una CPU y una GPU para realizar inferencia de redes neuronales profundas.

A diferencia de una CPU o una GPU, una TPU está diseñada específicamente para ejecutar modelos de *machine learning* sobre ella, especialmente **redes neuronales**.

Cuando ejecutamos inferencias sobre redes neuronales realizamos un volumen elevado de multiplicaciones y sumas entre los datos de entrada y los parámetros de la red, comprimidas como operaciones de producto de matrices.

Una CPU no deja de ser un procesador de propósito general, que si bien es capaz de llevar a cabo esta tarea no está diseñada específicamente para ella. Uno de los principales problemas que supone el uso de CPUs para estos cálculos intensivos es el incremento de la latencia de los accesos a memoria,

¹¹<https://coral.ai/docs/m2/datasheet/>

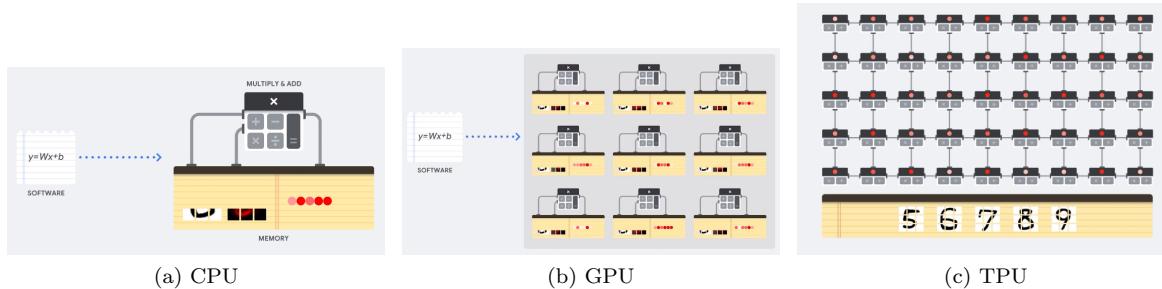


Figura 2.9: Estructura de las ALUs en una CPU, una GPU y una TPU. La principal diferencia que observamos es que, mientras la GPU replica la estructura de la CPU para realizar más operaciones en paralelo, la TPU conecta las salidas y entradas de las distintas ALUs. El acceso a memoria se representa con la hoja de cuaderno debajo de las unidades de cálculo. Mientras que en la GPU para cada operación se debe acceder antes y después para tomar y guardar los datos de las operaciones individuales (bien sea a memoria o a registros), en la TPU este acceso sólo se realiza al principio del cálculo general. Las imágenes se corresponden con capturas de pantalla tomadas de la animación <https://storage.googleapis.com/nexttpu/index.html>, donde se muestra de manera bastante sencilla la diferencia entre estas tres unidades para la inferencia sobre redes neuronales.

teniendo que almacenar los resultados tras cada cálculo (ya sea en registros o en caches), aun sabiendo que esos resultados serán necesarios para cálculos posteriores. La transferencia de datos continuada entre memoria y CPU da lugar a lo que se conoce como el **cuello de botella de von-Neumann**, dado por la latencia que supone esta transferencia de datos. Si realizamos estos cálculos en una GPU dispondremos de un número muchísimo mayor de ALUs que en una CPU que nos permitirán realizar gran cantidad de operaciones simultáneamente. Esto supone una ventaja a la hora de ejecutar estas operaciones de matrices, pues permite aumentar el paralelismo en el cálculo de estos valores, pero sigue estando presente el problema del cuello de botella de von-Neumann en lo relativo al acceso a memoria, teniendo que acceder y almacenar los datos para todos los cálculos intermedios. Además, el hecho de usar tantas ALUs simultáneamente supone un sobrecoste en el consumo de energía. A diferencia de la CPU y la GPU, la TPU está específicamente diseñada como un procesador de matrices para redes neuronales que trata de mitigar el efecto del cuello de botella de von-Neumann. Al ser una unidad de propósito específico se diseña teniendo en mente la estructura de las operaciones que tendrá que ejecutar, en este caso la multiplicación de matrices, colocando miles de multiplicadores y sumadores con sus salidas conectadas directamente, evitando así estos accesos innecesarios a memoria y dando lugar a lo que se conoce como una **arquitectura de array sistólico**¹². Para ejecutar las operaciones de inferencia, la TPU carga una única vez los datos y los valores de los parámetros desde la memoria. Las sucesivas operaciones van mandando sus resultados una vez están calculados, eliminando los accesos a memoria durante todo el proceso de cálculo. Conseguimos así mejorar el rendimiento en los cálculos y reducir el consumo de energía por los accesos a memoria (ver figura 2.9).

Google Coral

Muchos productos de **Google Coral** incluyen una TPU integrada dentro de su hardware. Nosotros usaremos un **acelerador M.2** (ver figura 2.10), que añade la TPU como coprocesador al sistema sobre el que se instalan, añadiéndole las capacidades de cómputo que esta unidad ofrece. Las especificaciones

¹²<https://www.sciencedirect.com/topics/engineering/systolic-arrays>

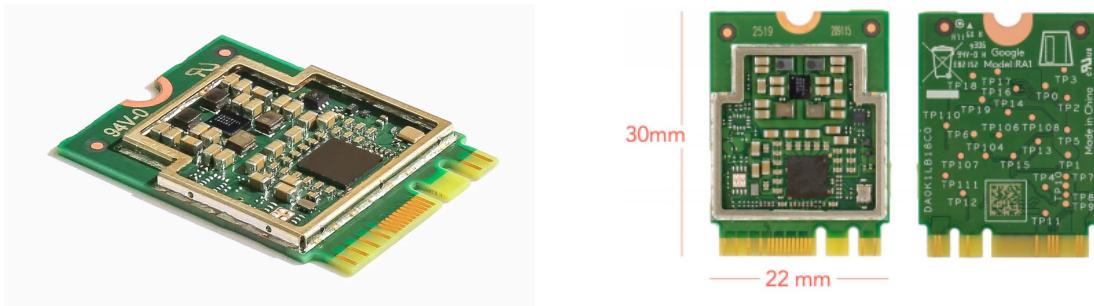


Figura 2.10: Google Coral M.2 Accelerator A+E Key, usado para mejorar la inferencia sobre modelos de *Tensorflow Lite* cuantizados.

concretas de este producto pueden consultarse en la propia página de Google Coral¹³.

El uso de la TPU como tarjeta M.2 en lugar de la versión USB del acelerador tiene ventajas adicionales, como el hecho de eliminar el sobrecoste de la interconexión USB.

A todas las ventajas ya mencionadas de estos aceleradores hemos de añadir su **bajo precio**, que podemos adquirir por un precio de 24.99\$ en la propia web de Google Coral¹⁴. Una vez tengamos el producto hay que seguir una serie de pasos para instalarlo y poder usarlo en nuestro sistema¹⁵, que básicamente consisten en instalar el *driver* PCIe, el *runtime* de Edge TPU y *PyCoral*, una biblioteca de *Python* desarrollada sobre *Tensorflow Lite* que simplifica las interacciones con la TPU.

2.8. Cuantización de modelos

Como hemos visto en la sección anterior, la TPU que usaremos para acelerar la inferencia sólo trabaja con **valores enteros de 8 bits**, pero los datos de los modelos que entrenemos (pesos de la red neuronal, entradas, salidas, ...) van a venir representados por números en **punto flotante de 32 bits**. Será necesario hacer una transformación de estos valores `float32` a `int8` para poder llevar a cabo esta inferencia sobre el acelerador de Google Coral. Este proceso va a ser lo que denominamos como **cuantización** y a continuación exponemos con más detalle en qué consiste y por qué funciona, basándonos en lo mostrado en [9].

Cuando representamos datos numéricos en un ordenador siempre lo hacemos de **manera discreta**, pues el número de valores que podemos representar es finito. Por tanto, no existe una representación para cada número real, y de hecho varios números son representados de la misma manera. El número de bits del que dispongamos para representar nuestro conjunto de números determinará una mayor o menor precisión en la representación, esto es, si podemos o no distinguir dos números próximos entre sí. El proceso de cuantización consiste en **reducir la precisión** a la hora de representar estos valores, disminuyendo el número de valores disponibles para representar los números reales y haciendo que el conjunto de números distintos con el que trabajamos sea menor. Esta reducción de la precisión trae consigo un decremento evidente en el volumen de memoria necesario para almacenar cada número, pues estamos disminuyendo el número de bits necesarios para representar cada valor, y la posibilidad de almacenar más datos en las caches o registros, reduciendo los accesos a memoria.

Nosotros aplicaremos la cuantización a los valores de una red neuronal, que en general suelen ser

¹³<https://coral.ai/docs/m2/datasheet/>

¹⁴<https://coral.ai/products/m2-accelerator-ae>

¹⁵<https://coral.ai/docs/m2/get-started#1-connect-the-module>

bastante robustas frente a pequeñas perturbaciones de esos valores. Y es que la cuantización, si se realiza correctamente, sólo trae consigo una pequeña pérdida de precisión en la representación de los valores que no afecta al comportamiento general del modelo.

Los modelos que entrenemos con *RLlib* tendrán una serie de parámetros dados por valores en punto flotante de 32 bits. Estos bits se dividen en tres conjuntos: signo, exponente y mantisa (ver figura 2.11), y el valor que representan viene dado por la siguiente expresión:

$$(-1)^{b_{31}} \times 2^{(b_{30}b_{29}\dots b_{23})_2 - 127} \times (1.b_{22}b_{21}\dots b_0)_2$$

S	Exponente	Mantisa
31 30	23 22	0

Figura 2.11: Representación de un valor en punto flotante de 32 bits.

Como podemos observar, el exponente permite representar un amplio rango de números mientras que la mantisa fija la precisión de los mismos.

Para poder usar el modelo en la TPU necesitamos que todos estos valores vengan dados por **enteros de 8 bits con signo**.

Aquí es donde entra en juego la cuantización, que va a consistir en mapear todos los valores que toman los parámetros del modelo en valores enteros en el intervalo [-127,128] (que son los que podemos representar con enteros de 8 bits). A la hora de definir esta función hemos de tener en cuenta dos aspectos:

1. Debe ser **lineal** para poder realizar la transformación inversa de manera directa.
2. **El 0 en punto flotante debe estar representado de manera correcta**, esto es, debe corresponderse con una de los valores cuantizados. Al cuantizar y descuantizar valores sólo $256 = 2^8$ de ellos volverán a tomar el mismo valor que tenían antes de la cuantización. Si aseguramos que el 0 en la representación en punto flotante sea uno de ellos obtendremos mejores resultados al cuantizar, ya que el 0 tiene un significado diferente al resto de valores en muchos de los parámetros de estas redes.

Asignaremos a los valores extremos de los datos sin cuantizar los valores extremos que pueden tomar los datos cuantizados, definiendo así un factor de escala del que serán múltiplo todos los números reales en el proceso de descuantización. Esto es, los extremos máximo y mínimo de ambos conjuntos de datos se mapean a los del otro y el 0 se mapea a uno de los valores cuantizados, asignando valores en punto flotante al resto de valores cuantizados. Así, los valores reales que no tengan un valor entero asignado se redondean al valor más próximo que sí que lo tenga, y se les asigna ese valor, perdiéndose precisión ya que dos valores distintos pero próximos en el modelo sin cuantizar pasan a representar el mismo valor en el modelo cuantizado. Podemos relacionar los valores cuantizados q y descuantizados r por medio de la siguiente expresión:

$$r = s \times (q - z),$$

donde z se denomina *zero-point* y se corresponde con el valor entero que asignamos al valor 0 en punto flotante y s es el factor de *escala*, que viene dado por el cociente entre el rango de valores reales y los que podemos representar de manera cuantizada, esto es,

$$s = \frac{r_{max} - r_{min}}{128 - (-127)} = \frac{r_{max} - r_{min}}{255},$$

con r_{max} y r_{min} los valores máximo y mínimo del conjunto de valores a cuantizar.

Los modelos que emplearemos en este trabajo estarán formados por varias capas que se implementan con valores en punto flotante:

- Tensores con los datos de la capa de convolución, que son constantes.
- Tensores con los datos de entrada.
- Tensores con el resultado de la capa para los datos de entrada.

Según lo expuesto antes, es tarea fácil convertir los parámetros propios de la red (pesos) en valores cuantizados una vez entrenada, pues sabemos de antemano el rango de valores que van a tomar. Los valores que no son constantes (como los de las entradas y salidas de cada capa) no se pueden conocer con exactitud y no se puede proceder del mismo modo que con los pesos. Sin embargo, sí que se puede estimar el rango en el que se van a mover observando los valores que toman en distintas ejecuciones. Esto es, tras una serie de ejecuciones con valores en punto flotante, podemos estimar el rango de valores que han tomado y considerar que este va a ser el rango aproximado en cualquier ejecución y realizar la cuantización con estos valores. Así, esta información para la cuantización puede obtenerse de dos formas: durante y después del entrenamiento. Dado que la fase de entrenamiento de nuestros modelos la vamos a realizar con *RLlib* optamos por la segunda opción, al no tener esa facilidad para configurar este entrenamiento cuantizado. Para ello, una vez entrenados los modelos, durante el proceso de cuantización, se ejecutarán unas cuantas inferencias con un conjunto de datos de entrada para el problema que resuelve el modelo, y los valores que tomen las distintas entradas y salidas en estas ejecuciones se utilizarán para llevar a cabo una correcta cuantización. Más adelante se detallará la implementación concreta de este proceso en nuestros modelos.

Capítulo 3

Implementación

El objetivo del trabajo será evaluar el rendimiento de los procesos de **entrenamiento** y de **inferencia** sobre modelos de aprendizaje por refuerzo. Realizaremos estas evaluaciones sobre diferentes arquitecturas hardware, que nos permitirán variar los recursos empleados. Con todo ello, obtendremos una serie de conclusiones en las que veremos qué es mejor en cada caso y los beneficios e inconvenientes de cada una de las opciones que consideremos.

3.1. Descripción de los entornos de pruebas

Tras haber realizado unas primeras pruebas para familiarizarse con el entorno de *Ray* y la biblioteca *RLlib*, el grueso del trabajo se realiza de manera remota en **tres servidores** del Departamento de Arquitectura de Computadores y Automática de la Facultad de Informática de la Universidad Complutense de Madrid. Describiremos a continuación las características de cada uno de estos sistemas:

- Servidor *esfinge*: El servidor cuenta con 16 CPUs Intel(R) Xeon(R) CPU E5-2670 0¹ de 2.60GHz y una GPU GeForce GTX 1080².
- Servidor *volta1*: Este servidor es el que más recursos nos ofrece: 40 CPUs Intel(R) Xeon(R) Gold 6138 CPU³ de 2 GHz y dos GPUs de última generación: GeForce RTX 3090⁴ y Tesla V100-PCIE-32GB⁵.
- Servidor *artecslab001*: En este servidor es donde se encuentra instalado el acelerador Google Coral sobre el que realizaremos el análisis de la inferencia. Además, cuenta con 16 CPUs Intel(R) Core(TM) i9-9900K CPU de 3.60GHz⁶.

¹<https://ark.intel.com/content/www/es/es/ark/products/64595/intel-xeon-processor-e5-2670-20m-cache-2-60-ghz-8-00-gt-s-intel-qlp.html>

²<https://www.nvidia.com/es-la/geforce/products/10series/geforce-gtx-1080/>

³<https://ark.intel.com/content/www/es/es/ark/products/120476/intel-xeon-gold-6138-processor-27-5m-cache-2-00-ghz.html>

⁴<https://www.nvidia.com/es-es/geforce/graphics-cards/30-series/rtx-3090/>

⁵<https://www.nvidia.com/es-es/data-center/tesla-v100/>

⁶<https://ark.intel.com/content/www/es/es/ark/products/186605/intel-core-i9-9900k-processor-16m-cache-up-to-5-00-ghz.html>

3.2. Descripción de los modelos

Evaluaremos el rendimiento de los procesos de entrenamiento e inferencia de aprendizaje por refuerzo usando para ello la biblioteca *RLLib* de *Ray*, de la que hemos mostrado algunas de las características y utilidades que ofrece en el capítulo anterior. Emplearemos el algoritmo PPO para el proceso de entrenamiento de nuestros modelos y el agente interaccionará con un entorno propio de *RLLib* que describimos a continuación.

3.2.1. Descripción del entorno del agente

RLLib permite la integración de entornos de otras bibliotecas como *Gym* dentro de su funcionalidad. Para la realización de los experimentos, el entorno *Gym* del que partimos es el denominado **Pong-v0**⁷, inspirado en el videojuego de Atari Inc. que simulaba una partida de tenis de mesa entre dos jugadores por medio de imágenes bidimensionales y que se lanzó originalmente en 1972 [17]. En este juego, un jugador marca un punto cuando la pelota sobrepasa la línea vertical en la que se mueve la pala del otro jugador y cada episodio concluye cuando uno de los dos jugadores alcanza los 21 puntos.

Gym se basa en la idea de este videojuego para modelar su entorno, en el que las observaciones las constituyen imágenes RGB de 210×160 píxeles (lo que da lugar a observaciones de tamaño $(210, 160, 3)$) y que están formadas por una representación esquemática de la partida, en la que se aprecian dos rectángulos simulando las palas de los jugadores (el jugador que controla el agente que entrenamos aparece a la izquierda de las imágenes) y la puntuación de cada uno de ellos en la parte superior.

Existen seis acciones posibles que puede tomar el agente en cada momento para la interacción con el entorno, sin embargo a efectos prácticos sólo hay tres acciones distintas. Podemos consultar las acciones disponibles para este entorno de la siguiente manera:

```

1 import gym
2
3 env = gym.make('Pong-v0')
4 print(env.unwrapped.get_action_meanings())
5
6 >> ['NOOP', 'FIRE', 'RIGHT', 'LEFT', 'RIGHTFIRE', 'LEFTFIRE']

```

Las acciones NOOP y FIRE mantienen en la misma posición la pala del agente, LEFT y LEFTFIRE la mueven a la izquierda (hacia arriba en la imagen) y RIGHT y RIGHTFIRE hacen lo propio hacia la derecha (hacia abajo en la imagen).

Cada acción vendrá representada por un entero entre 0 y 5, y podemos indicar al agente que realice una de ellas con el método **step**. Esto hará que se realice sobre el entorno la acción indicada k veces consecutivas, siendo k un número seleccionado al azar del conjunto $\{2, 3, 4\}$.

Las recompensas obtenidas por cada acción individual pueden tomar tres valores distintos: 0.0 si ninguno de los jugadores suma un punto tras esa acción, 1.0 si el jugador controlado por el agente suma punto y -1.0 si el punto lo suma el jugador controlado por la “máquina”. En cada episodio la recompensa total obtenida es la suma de las recompensas de cada una de las acciones que lo constituyen y por tanto es un valor entre -21.0 y 21.0 que representa la diferencia de puntos con la que acaba la partida, siendo este valor positivo si el jugador que gana es el que controla el agente y negativo en caso contrario.

Partiendo de este entorno como base, el que realmente usa *RLLib* para modelar el problema de aprendizaje cuando indicamos Pong-v0 como entorno en la inicialización de los agentes es una modificación del mismo. En este entorno modificado las observaciones tienen la forma $(dim, dim, 4)$, donde dim es un parámetro de configuración que podemos especificar al agente en su creación y que por defecto toma el valor 84. Este entorno es el resultado de aplicar una serie de **envolturas** (*wrappers*) al

⁷<https://gym.openai.com/envs/Pong-v0/>

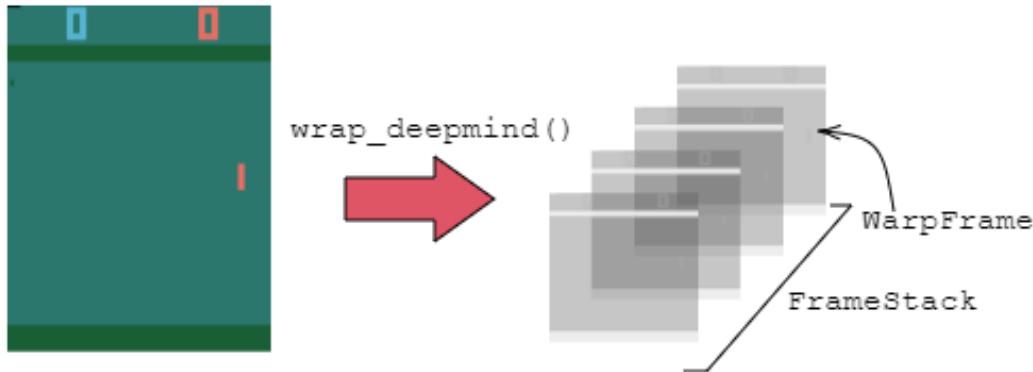


Figura 3.1: Comparación entre las imágenes del entorno *Gym* original y las del entorno de *RLLib* equivalente creado con la función `wrap_deepmind()`.

entorno original de *Gym*. Podemos crear este entorno con la función `wrap_deepmind()`⁸ indicando el entorno *Gym* sobre el que aplicar las envolturas y la dimensión de las imágenes de salida. Esta serie de envoltorios actúan realmente como un **preprocesador** de las imágenes del entorno original de *Gym*. Las imágenes en formato RGB del entorno original de *Gym* se transforman en esa misma imagen pero en **escala de grises** (haciendo uso de las utilidades de la biblioteca `cv2`⁹), pasando de tener tres a sólo un canal de color. Posteriormente, las imágenes se redimensionan para darles forma de cuadrado con la dimensión especificada como número de píxeles por lado, y se guarda una cola con las cuatro últimas imágenes procesadas en este formato (de ahí el 4 de la tercera dimensión de las imágenes), que se irá actualizando tras cada observación (ver figura 3.1). Así, las observaciones que recibirá nuestro algoritmo para aprender serán conjuntos de cuatro imágenes en escala de grises, correspondientes con las cuatro últimas observaciones obtenidas del entorno original de *Gym* correctamente redimensionadas. Usaremos una red neuronal de convolución con la que procesaremos estos datos no sólo para capturar las dependencias espaciales entre los elementos de la imagen sino también las temporales entre estados sucesivos del entorno.

3.2.2. Modelo de Tensorflow Keras

Los agentes que creemos durante la etapa de entrenamiento e inferencia tendrán asociado un modelo de *Tensorflow Keras*. Para nuestro problema, y al ser las observaciones imágenes, usaremos una **red neuronal de convolución** en nuestro modelo. Estas redes en *RLLib* se implementan como objetos de la clase `VisionNetwork`¹⁰. En la configuración del agente podemos dar valor a parámetros específicos de este modelo como la **dimensión de la entrada** (que como acabamos de ver se usaba también para crear el entorno con el que interaccionará el agente) y la **configuración de las capas de convolución** que procesarán las imágenes de entrada. De esta forma, se crea una red neuronal con dos salidas correspondientes a las salidas de la **política** (*policy network*) y la **función de valor** (*value network*). La salida de la política vendrá dada por seis valores que se corresponden con cada una de las seis acciones que podemos realizar en el entorno *Pong-v0*. Cada salida toma un valor en punto flotante,

⁸https://github.com/ray-project/ray/blob/master/rllib/env/wrappers/atari_wrappers.py#L288

⁹<https://pypi.org/project/opencv-python/>

¹⁰<https://github.com/ray-project/ray/blob/master/rllib/models/tf/visionnet.py>

al que se le aplica la función *softmax*, dada por:

$$\sigma : \mathbb{R}^6 \rightarrow [0, 1]^6$$

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{i=1}^6 e^{z_i}}$$

y que mueve los valores obtenidos al intervalo $[0, 1]$ para que realmente representen la probabilidad de que elegir cada una de las acciones sea la que maximice la recompensa final.

Podemos modificar la configuración del modelo de los agentes que vayamos a crear de manera sencilla. Cada agente que creemos en *Rllib* recibirá un diccionario `config` con los parámetros necesarios para su configuración. Una de las claves de este diccionario es `model`, que contiene como valor otro diccionario con la configuración del modelo que entrenará nuestro agente. En este diccionario `model` indicamos con los valores asociados a las claves `dim` y `conv_filters` la dimensión que queremos que tengan los datos de entrada y la configuración de las capas de convolución que aplicaremos a las imágenes, respectivamente. El primero de estos valores, la dimensión, la indicamos con un entero y la red resultante recibirá datos de entrada de tamaño ($dim, dim, 4$). La configuración de las capas de convolución la indicamos mediante una lista, en la que cada uno de sus elementos representa a una capa. A su vez, vamos a especificar los parámetros para cada capa como una lista con tres elementos:

- **`out_size`**: entero que indica el número de filtros con las mismas características que aplicaremos en esa capa (y que determina la última dimensión de la salida de esa capa).
- **`kernel`**: lista de dos elementos con la que indicamos las dimensiones del filtro de convolución que aplicaremos.
- **`stride`**: entero que indica el desplazamiento en píxeles de cada filtro de convolución.

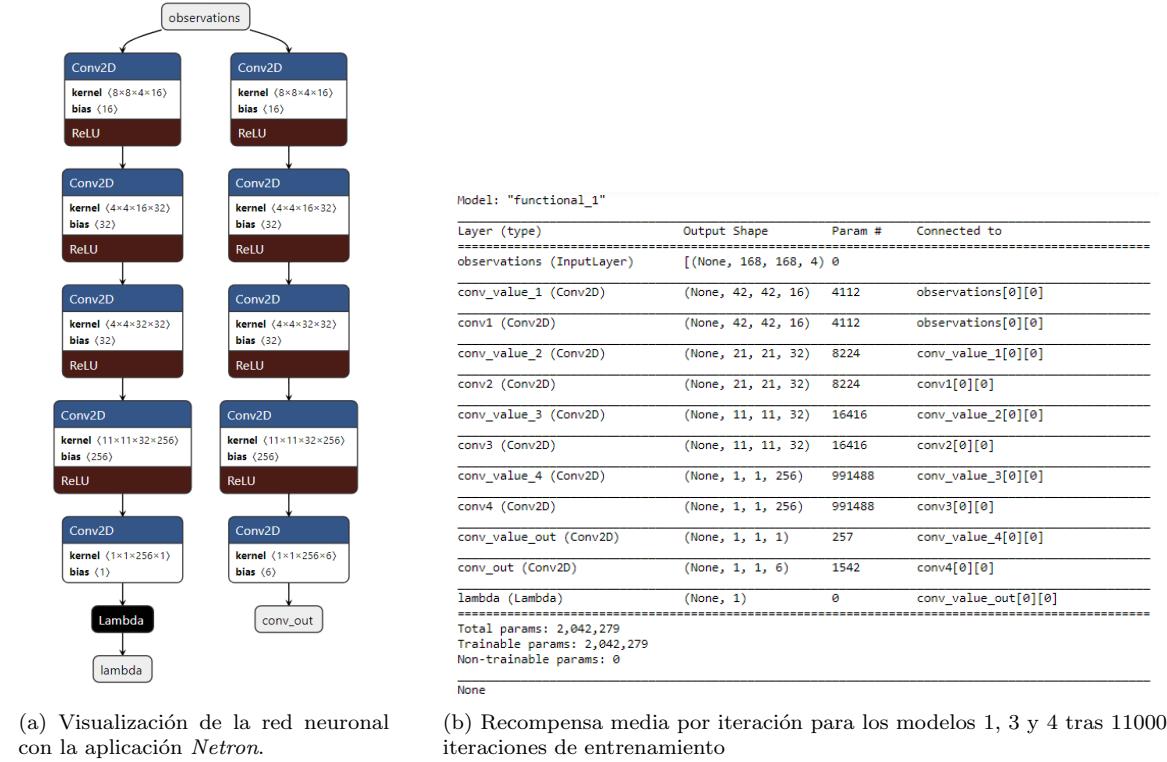
Cuando configuremos manualmente los filtros de convolución hemos de tener en cuenta que la concatenación de los mismos debe producir una salida de tamaño $(B, 1, 1, X)$, donde X es el número de filtros de convolución de la última capa. El tamaño de entrada por defecto de 84×84 de *Ray* trae ya asociada una configuración de capas de convolución. Para el resto de tamaños de entrada tendremos que especificar de manera manual la configuración de los mismos.

Rllib crea para el agente un modelo de *keras*, con una serie de capas `conv2D` correspondientes a los filtros especificados. Todas las capas, a excepción de la última, son creadas con el parámetro `padding='same'` mientras que la última de ellas se configura con `padding='valid'`. El ancho y el alto de cada capa obedecen a la fórmula $dim_salida = \text{ceil}(dim_entrada / stride)$, salvo para la última, cuyos valores vienen dados por $dim_salida = (dim_entrada - kernel) / stride + 1$. Para que la dimensión de esta última capa sea 1 tal y como requiere *Rllib*, tendremos que hacer que el `kernel` del último filtro de convolución sea de igual tamaño que el ancho y alto de las entradas que llegan a esta última capa. A continuación mostramos un ejemplo de código en el que se refleja cómo crear un agente PPO que recibe imágenes de 168×168 píxeles y para el que indicamos también el parámetro asociado a los filtros de convolución.

```

1 import ray
2 import ray.rllib.agents.ppo as ppo
3
4 ray.init()
5 config = ppo.DEFAULT_CONFIG.copy()
6
7 config['model']['dim'] = 168
8 config['model']['conv_filters'] = \
9     [16, [8, 8], 4], [32, [4, 4], 2], [32, [4, 4], 2], [256, [11, 11], 1]
10
11 agent = ppo.PPOTrainer(config, env='Pong-v0')

```

(a) Visualización de la red neuronal con la aplicación *Netron*.

(b) Recompensa media por iteración para los modelos 1, 3 y 4 tras 11000 iteraciones de entrenamiento

Figura 3.2: Representación de la red neuronal de *Keras* que se crea para procesar imágenes de tamaño 168.

Se crea así un modelo de *Keras* con 4 capas de convolución que se corresponden con la configuración especificada. La figura 3.2 muestra la red neuronal que se crea (3.2a) como modelo de *Keras* y que podemos visualizar con la herramienta *Netron*¹¹. Además, mostramos también un resumen de la estructura de capas de esta red neuronal, que podemos visualizar invocando la función `summary()` sobre el modelo de *Keras* (si tenemos un agente de *RLLib* podemos hacer esto con `agent.get_policy().model.base_model.summary()`). Además, en (3.2b) podemos ver también el número de parámetros entrenables (pesos) que hay en cada capa y el total del modelo.

3.2.3. Modelos propuestos

Para poder evaluar el rendimiento en diferentes modelos, uno de los parámetros que tendremos en cuenta será el **tamaño de las imágenes** que se toman del entorno, pues queremos ver como se gestionan los recursos y cómo varían los tiempos de inferencia y entrenamiento si los modelos procesan imágenes más o menos grandes. Proponemos así un modelo que recibe imágenes de 84×84 píxeles, ya que este es el valor por defecto en *RLib*. Además, consideraremos otros cinco modelos más en los que el tamaño de entrada es el doble o el triple del que ofrece *RLlib* por defecto (168 y 252, respectivamente).

Respecto a las capas de convolución, no tenemos manera de decidir que configuración se ajusta mejor a las imágenes de nuestro modelo, por lo que propondremos varias opciones para cada tamaño de entrada y al final seleccionaremos aquella configuración con la que obtengamos mejores resultados.

¹¹<https://netron.app/>

Para el modelo con entradas de dimensión 84 directamente tomamos la configuración de las capas de convolución que nos da *RLlib* por defecto.

La tabla 3.1 muestra las características de los **seis modelos** escogidos, que difieren entre ellos en el tamaño de los datos de entrada y en la estructura de sus capas de convolución.

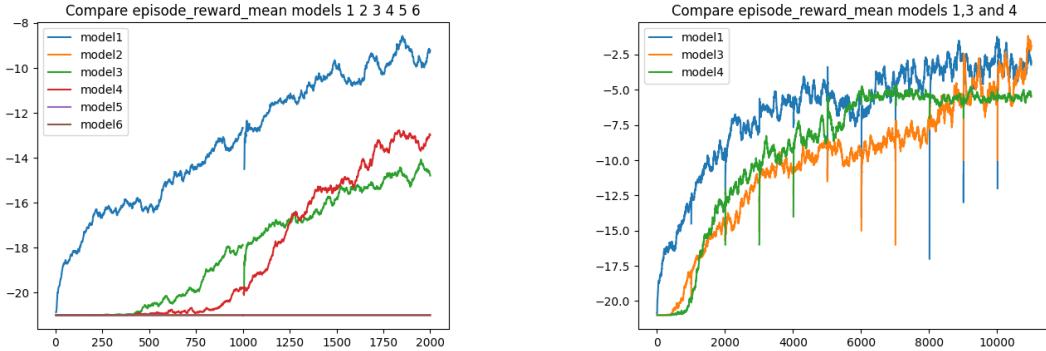
Modelo	Tamaño de la entrada	Filtros de convolución	Parámetros entrenables
1	84×84	[16, [8, 8], 4], [32, [4, 4], 2], [256, [11, 11], 1]	2.009.447
2	168×168	[16, [16, 16], 8], [32, [4, 4], 2], [256, [11, 11], 1]	2.034.023
3	252×262	[16, [8, 8], 4], [16, [8, 8], 4], [32, [4, 4], 2], [256, [8, 8], 1]	1.108.359
4	168×168	[16, [8, 8], 4], [32, [4, 4], 2], [32, [4, 4], 2], [256, [11, 11], 1]	2.042.279
5	252×252	[16, [8, 8], 4], [32, [4, 4], 2], [32, [4, 4], 2], [256, [16, 16], 1]	4.254.119
6	168×168	[16, [8, 8], 4], [32, [4, 4], 2], [256, [21, 21], 1]	7.252.327

Cuadro 3.1: Modelos propuestos, con el tamaño de las imágenes que recibe como entrada, la configuración de las capas de convolución que indicamos mediante el parámetro `conv_filters` y el número de parámetros entrenables (pesos) de la red neuronal resultante.

3.2.4. Modelos seleccionados

De los seis modelos propuestos anteriormente, vamos a seleccionar tres de ellos para continuar con el análisis en el que se va a centrar este trabajo. Para ello, vamos a seleccionar **un modelo por cada tamaño de entradas**, así podremos realizar el análisis de tiempo y rendimiento teniendo datos para modelos distintos que trabajan con imágenes de diferente tamaño. Para realizar esta selección, ejecutamos **2000 iteraciones de entrenamiento** sobre cada uno de los modelos para observar las recompensas que se obtienen al cabo de este tiempo y tener un criterio más que nos ayude a determinar qué tres modelos elegir. Al fin y al cabo, hemos elegido la configuración de las capas de convolución sin guiarnos por ningún criterio, por lo que haciendo esto intentamos determinar de manera empírica qué configuración de las propuestas comienza a mejorar sus recompensas más rápidamente. Estas iteraciones de entrenamiento se llevan a cabo en el servidor *esfinge* con 8 *rollout workers* y haciendo uso de la GPU GTX.

Como podemos observar en la gráfica 3.3a sólo hay tres de los seis modelos que mejoran la recompensa media tras 2000 iteraciones de entrenamiento. Además, cada uno de ellos recibe como datos de entrada imágenes de un tamaño distinto, por lo que elegimos los modelos 1, 3 y 4 como aquellos que emplearemos para obtener el resto de resultados y conclusiones de este trabajo. El estado de las recompensas medias por iteración tras 11000 iteraciones de entrenamiento ya sólo para estos tres modelos seleccionados puede verse en la gráfica de la imagen 3.3b. Tras 2000 iteraciones de entrenamiento vemos como el modelo 1 es el que más rápido empieza a mejorar sus recompensas y el que transcurridas estas iteraciones obtiene mayor valor para la recompensa media, mientras que los modelos 2 y 3 necesitan más iteraciones para comenzar a ofrecer valores mayores en las recompensas. Observando los resultados tras 11000 iteraciones vemos que tras una primera fase de crecimiento más rápido el valor de las recompensas se estanca y crece más lentamente. De ahora en adelante, no analizaremos más la **calidad del aprendizaje** de los distintos modelos. El hecho de variar los recursos del sistema sobre el que entrenamos los modelos no debería influir en la capacidad de aprendizaje de los mismos, pues el algoritmo que se está ejecutando no varía, únicamente lo realizamos sobre soportes distintos. Así, la calidad del aprendizaje depende únicamente de la estructura del modelo en sí (estructura de sus capas), por lo que este análisis que hemos realizado para seleccionar los tres modelos sobre los que analizar el rendimiento de los procesos cubriría esta otra parte de evaluación de la capacidad de aprendizaje y de las recompensas obtenidas.



(a) Recompensa media por iteración para los modelos 1,2,3,4,5 y 6 tras 2000 iteraciones de entrenamiento

(b) Recompensa media por iteración para los modelos 1, 3 y 4 tras 11000 iteraciones de entrenamiento

Figura 3.3: Evolución de las recompensas medias para los modelos propuestos y para los seleccionados para los experimentos. Obsérvese que, dado que este entrenamiento se realizó en varias etapas de 1000 iteraciones que continuaban desde el estado de la anterior, en los valores inmediatos a las iteraciones múltiplo de 1000 se observan oscilaciones importantes en el valor de las recompensas y que se corresponden con las iteraciones de calentamiento de cada tanda de iteraciones de entrenamiento.

3.3. Análisis del rendimiento

Detallaremos aquí qué aspectos tendremos en cuenta para analizar el rendimiento de los modelos y cómo obtendremos los datos que nos servirán para obtener diversas conclusiones, tanto para la fase de entrenamiento como la de inferencia. Propondremos una serie de **experimentos de entrenamiento e inferencia**, cada uno de los cuales contará con una configuración específica y tras ejecutarlos, compararemos y analizaremos los resultados obtenidos.

3.3.1. Implementación de los experimentos de entrenamiento

Entrenamos los modelos haciendo uso exclusivamente de las funcionalidades que nos ofrece *RLLib* para ello. Se diseñan unos *scripts* que nos van a permitir llevar a cabo un número específico de iteraciones de entrenamiento, guardando un *checkpoint* con el estado del modelo tras cada una de ellas. Esto proceso parte de la base expuesta en [7].

Para el entrenamiento nos ayudaremos del *script* `train_ppo.py`, que nos permite ajustar diversos parámetros para configurar cada uno de los experimentos de entrenamiento (modelo a entrenar, recursos a utilizar, número de iteraciones de entrenamiento, dirección de la que cargar los datos si ya habíamos entrenado antes...). Para ver más detalles consultar el apéndice A.1.

Realizaremos varios experimentos con cada uno de los tres modelos, en los que variaremos los recursos empleados para el proceso de entrenamiento. Estos experimentos se realizan en el servidor *volta1* en el que disponemos de 40 CPUs y 2 GPUs.

Los parámetros de la configuración de recursos que vamos a variar serán fundamentalmente tres:

- **GPUs a utilizar:** consideraremos cuatro opciones respecto al uso de las GPUs del sistema. Así, entrenaremos sin hacer uso de las GPUs (configuración `none`), usando sólo la GPU Nvidia Ge-Force RTX (configuración `gpu0`, usando sólo la GPU Nvidia Tesla v100-PCIE (configuración `gpu1`) y usando ambas (configuración `both`).

- **Número de *rollout workers*:** variamos el número de *rollout workers* que interaccionan con el entorno y recolectan datos para la fase de actualización de la política. Durante el entrenamiento, los diferentes experimentos crearán 2, 4, 8 y 16 *rollout workers*.
- **Uso de la GPU por el *driver*:** los *workers* y el *driver* pueden hacer uso de las mismas GPUs, pues al fin y al cabo no trabajan simultáneamente sobre ella. *Ray* nos permite especificar en la configuración de los agentes el número de GPUs que queremos que usen el *driver* y cada uno de los *workers* con números decimales. En la propia documentación de *Ray*¹² se sugiere la siguiente configuración si tenemos disponibles n GPUs para el entrenamiento, usamos un algoritmo síncrono como PPO y queremos que tanto *driver* como *workers* hagan uso de las GPUs:

```

1 # GPUs for driver
2 config['num_gpus'] = 0.0001
3
4 # GPUs for each worker
5 config['num_gpus_per_worker'] = (n-0.0001)/config['num_workers']
6

```

Con una configuración como la anterior conseguimos que tanto los *workers* como el *driver* hagan uso de las GPUs. Probaremos esta configuración y también el caso en el que sólo el *driver* haga uso de las GPUs.

Para cada uno de los modelos realizaremos **10 experimentos de entrenamiento** en los que recogeremos métricas y resultados para su posterior análisis. La tabla 3.2 muestra la configuración para cada uno de los experimentos que realizaremos para cada uno de los tres modelos seleccionados, lo que dará lugar a un total de 30 experimentos. En cada experimento llevaremos a cabo 20 iteraciones de entrenamiento, que serán suficientes para tener datos sobre el uso de recursos y los tiempos de ejecución, que es lo que nos interesa medir. En el capítulo 4 detallaremos qué datos analizaremos para cada uno de estos experimentos y cómo se obtienen.

Id	Nombre	Número de <i>workers</i>	GPU config	GPUs para el <i>driver</i>	GPUs para cada <i>worker</i>
Exp1	no_gpus_8_workers	8	none	0	0
Exp2	gpu0_8_workers	8	gpu0	0.0001	0.1249875
Exp3	gpu0_driver_8_workers	8	gpu0	1	0
Exp4	gpu1_8_workers	8	gpu1	0.0001	0.1249875
Exp5	gpu1_driver_8_workers	8	gpu1	1	0
Exp6	both_gpus_2_workers	2	both	0.001	0.9995
Exp7	both_gpus_4_workers	4	both	0.001	0.49975
Exp8	both_gpus_8_workers	8	both	0.001	0.249875
Exp9	both_gpus_driver_8_workers	8	both	2	0
Exp10	both_gpus_16_workers	16	both	0.0001	0.12499375

Cuadro 3.2: Configuraciones para los experimentos de entrenamiento de modelos. La columna *GPU config* hace referencia al parámetro con el que establecemos las GPUs visibles al proceso.

Ray funciona como un planificador y gestor de recursos que crea tareas y las lanza para que se ejecuten. Cada vez que iniciamos *Ray* podemos indicarle un número de recursos (CPUs y GPUs) que queremos que tenga en cuenta a la hora de planificar. Si no lo hacemos, considerará que tiene disponibles todos los recursos visibles en el sistema. El número de CPUs con el que trabaje *Ray*

¹²<https://docs.ray.io/en/master/rllib-training.html#specifying-resources>

influirá en el número de hilos que se crean en el sistema, si bien el número de estos que están en ejecución simultáneamente no se ve influido por este hecho. Sin embargo, inicializar *Ray* indicando que el número de CPUs es 8 no significa que necesariamente sólo vayan a ejecutar tareas en 8 de las CPUs del sistema. Es más, en la mayoría de los casos veremos cómo todas las CPUs libres del sistema ejecutan algún proceso durante las etapas de entrenamiento. Si lo que queremos es restringir las ejecuciones de *Ray* a sólo una parte de las CPUs del sistema, debemos forzar esta ejecución de manera externa, pues *Ray* no proporciona recursos para ello. Una opción sería hacer uso de la función `sched_setaffinity()`¹³ de la biblioteca `os` de *Python*. Por ejemplo, en un experimento con 2 *workers* y el *driver*, haciendo `os.sched_setaffinity(0,{0,1,2})` en el proceso que lanza *Ray* a ejecución conseguiremos que sólo se haga uso de las tres primeras CPUs del sistema. En definitiva, aunque en la configuración de los agentes de *RLlib* se indique que el número de CPUs asociadas para el *driver* y cada uno de los *workers* es uno, como realmente controlamos este uso es o bien indicando a *Ray* en su inicialización el número de CPUs que tiene que tener en cuenta o forzando el uso de un subconjunto de ellas externamente. Además, hemos de tener en cuenta que *Ray* tiene más procesos que hacen uso siempre de las CPUs (como el *dashboard* que muestra información sobre la ejecución), por lo que no es posible de ninguna manera llevar a cabo experimentos en *RLlib* dejando totalmente libres todas las CPUs del sistema.

Id	Nombre	Inicialización de Ray
Exp11	gpu0_no_cpu_limit_8_workers	ray.init()
Exp12	gpu1_no_cpu_limit_8_workers	
Exp13	both_gpus_no_cpu_limit_8_workers	
Exp14	gpu0_9_cpus_no_set_affinity_8_workers	ray.init(num_cpus=9)
Exp15	gpu1_9_cpus_no_set_affinity_8_workers	
Exp16	both_gpus_9_cpus_no_set_affinity_8_workers	
Exp17	gpu0_9_cpus_set_affinity_8_workers	ray.init(num_cpus=9) os.sched_setaffinity(0,{0,1,2,3,4,5,6,7,8})
Exp18	gpu1_9_cpus_set_affinity_8_workers	
Exp19	both_gpus_9_cpus_set_affinity_8_workers	

Cuadro 3.3: Experimentos con distintas configuraciones de uso de las CPUs del sistema.

Añadimos una serie de experimentos más para cada modelo a los expuestos anteriormente (recogidos en la tabla 3.3), en los que fijando el número de *workers* a 8 variaremos el uso de las GPUs (gpu0, gpu1 y both) y probaremos **tres configuraciones** a la hora de analizar el **uso de las CPUs** del sistema:

- **no_cpu_limit**: no indicamos a *Ray* el número de CPUs del que dispone en su inicialización ni restringimos la ejecución a un conjunto de CPUs específico, esto es, estamos en la misma situación que en los experimentos Exp2, Exp4 y Exp8, dependiendo de la configuración de GPUs elegida.
- **9_cpus_no_setaffinity**: indicamos a *Ray* en su inicialización el número de CPUs de las que dispone para planificar, que van a ser 9 (una para cada *worker* y una más para el *driver*) con `ray.init(num_cpus=9)` pero no restringimos la ejecución a un subconjunto determinado del total de las CPUs del sistema.
- **9_cpus_setaffinity**: además de indicar a *Ray* que dispone de 9 CPUs hacemos uso de la función `sched_setaffinity` para restringir la ejecución a sólo 9 de las CPUs del sistema, dejando las restantes libres para otras ejecuciones que no interfieran con *Ray*.

En la tabla 3.3 aparecen los nombres de los nueve experimentos que realizamos para cada uno de los modelos, todos ellos se realizan con 8 *workers* y las GPUs disponibles en cada caso (que serán una o dos) se comparten entre los *workers* y el *driver*.

¹³https://docs.python.org/3/library/os.html#os.sched_setaffinity

3.3.2. Implementación de los experimentos de inferencia

El proceso de inferencia consiste en evaluar la calidad del aprendizaje de los modelos una vez entrenados, esto es, llevar a cabo una serie de interacciones con el entorno en las que las observaciones que obtenemos sirven como datos de entrada para que el modelo nos indique la próxima acción que debemos realizar, sin modificar ya en ningún momento el estado del modelo. Esto es, una vez entrenado el modelo lo usamos para determinar cuál es la acción que debemos tomar en cada momento. Más que en evaluar si los modelos se han entrenado bien (esto es, si son capaces de obtener recompensas altas), analizaremos nuevamente datos relativos al rendimiento de este proceso (tiempos empleados y uso de recursos). Analizaremos esta inferencia de dos maneras: por un lado, dentro del *framework* de *Ray* y haciendo uso de los recursos hardware que usamos para el entrenamiento (CPUs y GPUs) y por otro, haciendo uso de aceleradores hardware externos como la TPU Google Coral, donde será necesario transformar los modelos y salirnos de las utilidades que nos proporciona *Ray* para poder llevar a cabo esta inferencia.

Inferencia en RLLib

RLLib permite ejecutar los modelos previamente entrenados y ver su comportamiento cuando interactúan con el entorno. Para ello tenemos a nuestra disposición el *script* `rollout.py`¹⁴, que podemos ejecutar indicando un *checkpoint* desde el que cargar el estado del agente entrenado y se llevarán a cabo una serie de ejecuciones de inferencia sobre el modelo. Como nuestro objetivo es medir el tiempo empleado en este proceso de inferencia, modificaremos el *script* previamente mencionado para **incluir temporizadores** en la llamada que se hace a cada paso de inferencia (esto es, vamos a medir el tiempo que emplea *Ray* en propagar a lo largo de las capas del modelo unos datos de entrada hasta obtener la siguiente acción a realizar. Analizando este *script* de *rollout* vemos que en cada paso de inferencia se hace una llamada a la función `compute_action`¹⁵, que recibe como argumento una observación del entorno y nos devuelve el identificador (entero del 0 al 5) de la siguiente acción a tomar. Mediremos este tiempo y almacenaremos la información necesaria para su posterior análisis.

La ejecución de este *script* lleva consigo una inicialización de *Ray*, pues hay que crear un agente y restablecer el estado de su modelo desde el *checkpoint* indicado. Así, podemos variar los recursos utilizados como hacíamos durante el proceso de entrenamiento: restringir el uso de las GPUs disponibles en el sistema a una de ellas, dos o ninguna e indicar a *Ray* el número de CPUs que tendrá disponibles durante su inicialización, además de poder restringir el uso de CPUs a un subconjunto del volumen total disponible en el sistema. Modificamos el *script* de `rollout.py` que nos proporciona *Ray*, añadiéndole argumentos adicionales para configurar la gestión de los recursos disponibles y obteniendo los datos de tiempo por cada inferencia (ver apéndice A.2).

La tabla 3.4 muestra la configuración de los distintos experimentos que realizaremos para cada uno de los modelos y cuyos resultados posteriormente analizaremos. En cada uno de ellos restauraremos el estado de la red neuronal desde cualquier *checkpoint* de un estado previo de entrenamiento, pues al final sólo nos interesa cargar la estructura del modelo y no los valores concretos de sus parámetros, que sólo harán que el resultado sea diferente, pero no el tiempo empleado en obtenerlo. Los recursos empleados en el entrenamiento de estos modelos hasta la obtención del estado de la red neuronal almacenada en los correspondientes *checkpoints* no van a influir en la inferencia, pues en cada experimento de inferencia variaremos los parámetros relativos al uso de las GPUs o del número de *workers* que se crean. Por ejemplo, aunque restablezcamos el estado del modelo desde un *checkpoint* que se entrenó con la GPU RTX, podremos realizar los experimentos de inferencia sobre ese agente con cualquier configuración de GPUs.

¹⁴<https://github.com/ray-project/ray/blob/master/rllib/rollout.py>

¹⁵<https://github.com/ray-project/ray/blob/ba45e41b4d3f8f0e84b7e8c6ae16d996dcb48f07/rllib/agents/trainer.py#L907>

Id	Nombre	Workers	GPU config	GPUs para el driver	GPUs para los workers
R1	8_workers_gpu0	8	gpu0	0.0001	0.1249875
R2	8_workers_gpu1	8	gpu1	0.0001	0.1249875
R3	8_workers_both_gpus	8	both	0.001	0.249875
R4	16_workers_both_gpus	16	both	0.0001	0.12499375
R5	0_workers_gpu0	0	gpu0	1	0
R6	0_workers_gpu1	0	gpu1	1	0
R7	0_workers_both_gpus	0	both	2	0
R8	0_workers_no_gpus	0	none	0	0

Cuadro 3.4: Resumen de los experimentos realizados para analizar la inferencia en *RLlib*.

Mantenemos en nuestro *script* la idea que desarrolla *Ray* para obtener la configuración del agente. En primer lugar, se comprobará si en el directorio en el que se encuentra el *checkpoint* desde el que vamos a restablecer el estado del modelo o en su directorio padre se encuentra un fichero *params.pkl* que contiene la información de configuración del agente. *Ray* genera automáticamente este fichero tras cada entrenamiento y no contiene información sobre el entrenamiento en sí, sino sobre el agente que creamos. El *script* de entrenamiento lleva a cabo la tarea de mover este fichero hasta el directorio en el que se encuentra el *checkpoint* de manera automática, por lo que tras realizar un entrenamiento podremos hacer un *rollout* desde el último *checkpoint* guardado sin preocuparnos por este fichero. Adicionalmente, si no existe el archivo *params.pkl* podemos especificar la configuración del agente con el parámetro de configuración *--config* cuando ejecutemos el *script*, indicando mediante un diccionario los parámetros necesarios. Nosotros usaremos esta opción para **sobrescribir la configuración** cargada del archivo *params.pkl*, pues si ya hemos cargado la configuración de esta manera podemos indicar en *config* las claves para las que queremos considerar un valor distinto, que en nuestro caso vendrán a ser aquellas relacionadas con el uso de las GPUs y el número de *workers* que se crean.

```

1 # R1
2 python rollout_with_time.py \
3     checkpoints/ppo/model1_gpu/checkpoint_11999/checkpoint-11999 \
4     --run=PPO --env=Pong-v0 --no-render --episodes=10 --gpu=gpu0 \
5     --config='{"num_gpus":0.0001, "num_gpus_per_worker":0.1249875, "num_workers":8}' \
6     --time-output=rollout_results/volta1/model1/model1_8_workers_gpu0
7
8 # R4
9 python rollout_with_time.py \
10    checkpoints/ppo/model1_gpu/checkpoint_11999/checkpoint-11999 \
11    --run=PPO --env=Pong-v0 --no-render --episodes=10 --gpu=both \
12    --config='{"num_gpus":0.0001, "num_gpus_per_worker":0.12499375, "num_workers":16}' \
13    --time-output=rollout_results/volta1/model1/model1_16_workers_both_gpus
14
15 # R6
16 python rollout_with_time.py \
17     checkpoints/ppo/model1_gpu/checkpoint_11999/checkpoint-11999 \
18     --run=PPO --env=Pong-v0 --no-render --episodes=10 --gpu=gpu1 \
19     --config='{"num_gpus":1, "num_gpus_per_worker":0, "num_workers":0}' \
20     --time-output=rollout_results/volta1/model1/model1_0_workers_gpu1
21
22 # R8
23 python rollout_with_time.py \
24     checkpoints/ppo/model1_gpu/checkpoint_11999/checkpoint-11999 \
25     --run=PPO --env=Pong-v0 --no-render --episodes=10 --gpu=none \
26     --config='{"num_gpus":0, "num_gpus_per_worker":0, "num_workers":0}' \
27     --time-output=rollout_results/volta1/model1/model1_0_workers_no_gpus

```

El fragmento de código anterior muestra las instrucciones *shell* con las que ejecutamos algunos de los experimentos de inferencia en *RLlib* previamente descritos.

Inferencia sobre el acelerador Google Coral

Para evaluar el rendimiento de la inferencia sobre el acelerador Google Coral será necesario abandonar el *framework* de *Ray*. Tomaremos los modelos de *Tensorflow* obtenidos como resultado del proceso de entrenamiento que hemos llevado a cabo (será necesario exportarlos fuera de *RLlib*) y les aplicaremos una **serie de transformaciones** para poder inferir sobre ellos en la TPU¹⁶. La propia documentación de Coral nos indica las características que deben reunir los modelos para poder ser ejecutados sobre este acelerador.

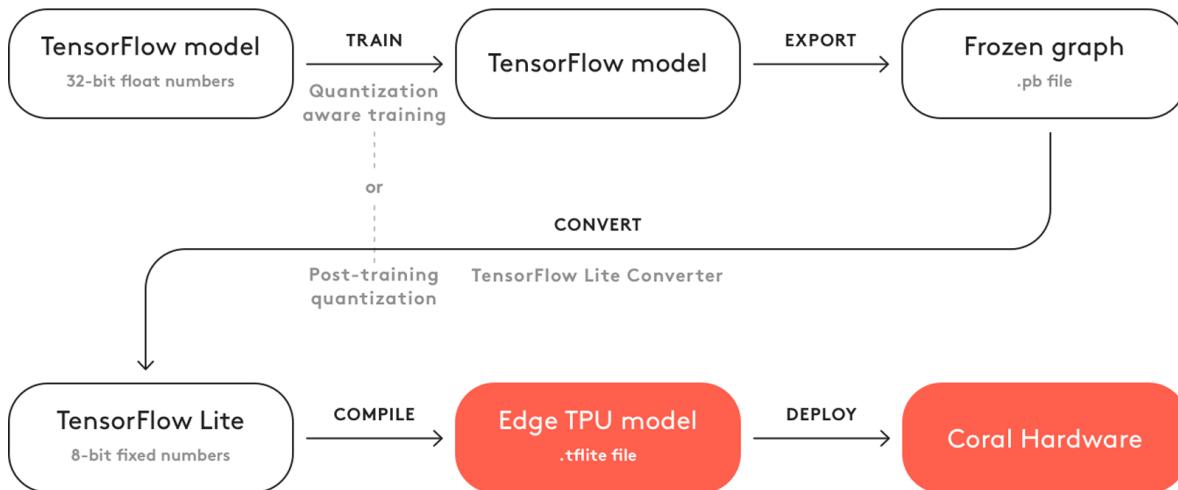


Figura 3.4: Imagen obtenida de la documentación de Coral en la que se muestran los pasos a seguir para obtener un modelo ejecutable en la TPU.

La figura 3.4 muestra el proceso de transformación de modelos hasta conseguir uno con el que podamos ejecutar inferencias sobre la TPU. Básicamente necesitamos conseguir un **modelo de *Tensorflow Lite* cuantizado**, esto es, un modelo de *Tensorflow Lite* en el que los valores de los pesos en la red neuronal vengan dados por enteros de ocho bits (`int8`) y no por números en punto flotante de 32 bits (`float32`) como suele ser habitual en los modelos de *Tensorflow*. Cuando entrenamos agentes con *RLlib* internamente se crea un modelo de *Tensorflow* que representa toda la estructura de la política que estamos entrenando. Para convertir el modelo de *Tensorflow* a uno de *Tensorflow Lite* debemos en primer lugar exportar el modelo, tal y como se muestra en el diagrama de la imagen 3.4. Podríamos obtener directamente un modelo de *Tensorflow* en formato `.pb` para un agente de *Ray* previamente creado (y restableciendo su estado desde un *checkpoint*) ejecutando el siguiente código, donde `checkpoint_dir` es la ruta a un *checkpoint* de *RLlib*:

```

1 agent=ppo.PPOTrainer(config, env='Pong-v0')
2 agent.restore(checkpoint_dir)
3 agent.export_policy_model(export_name)
  
```

La ejecución de estas instrucciones creará un directorio en la ruta indicada por `export_name`, en el que uno de los archivos que contendrá será el modelo en formato *exported model* de *Tensorflow* (fichero

¹⁶<https://coral.ai/docs/edgetpu/models-intro/#compatibility-overview>

`export_name/saved_model.pb`). Sin embargo, proceder de esta manera traerá consigo problemas a la hora de obtener el modelo de *Tensorflow Lite*. En realidad, haciendo esto lo que estamos guardando es el grafo completo de *Tensorflow* que se crea para el proceso de entrenamiento, con multitud de nodos que van a ser irrelevantes para nuestro propósito y que contienen operaciones propias para *RLlib* que no será trivial convertir a *Tensorflow Lite*. Para solventar esta situación, prescindiremos de todo aquello que no nos interese para conseguir nuestro modelo de *Tensorflow Lite*, centrándonos únicamente en la red neuronal de convolución que procesa las imágenes obtenidas desde el entorno. Esta red se corresponde con un **modelo de keras**, al que podemos acceder a través del atributo `base_model` de la política asociada a nuestro agente en *RLlib*, y va a ser este modelo el que vamos a exportar para su posterior conversión a *Tensorflow Lite*. Guardaremos esta información en un fichero con extensión `.h5`, que contiene sólo la estructura de la red neuronal y los valores de sus pesos. A continuación, podemos ver el código necesario para llevar a cabo esta acción, que podemos encontrar en el *script model_saver.py*¹⁷:

```

1 agent = ppo.PPOTrainer(config, env='Pong-v0')
2 agent.restore(checkpoint_dir)
3 print(agent.get_policy().model.base_model.summary())
4
5 with agent.get_policy().get_session().graph.as_default():
6     export_model = agent.get_policy().model.base_model.save(export_name + '.h5')
```

Una vez tenemos nuestro fichero `.h5` el siguiente paso a realizar es obtener el modelo de *Tensorflow Lite* correspondiente. Para ello, seguimos los pasos indicados en la documentación de *Tensorflow*¹⁸. El código con el que realizamos la conversión, perteneciente al *script tflite_converter.py*¹⁹ es el siguiente:

```

1 model = tf.keras.models.load_model(h5_dir, custom_objects={'tf':tf})
2 converter = tf.lite.TFLiteConverter.from_keras_model(model)
3 tflite_model = converter.convert()
4 open(tflite_dir, "wb").write(tflite_model)
```

Ahora que ya tenemos el modelo de *Tensorflow Lite* debemos cuantizarlo, esto es, cambiar los valores de sus parámetros de punto flotante de 32 bits a enteros de 8 bits. Mirando el diagrama de la figura 3.4 vemos que el proceso de cuantización puede llevarse a cabo en dos puntos distintos: durante el propio entrenamiento del modelo o una vez tenemos el modelo entrenado. Dado que el entrenamiento lo hemos realizado haciendo uso de *RLlib*, optamos por la segunda opción y realizamos lo que se denomina como **cuantización post-entrenamiento**. Además, por las características del acelerador es necesario realizar una cuantización a enteros de todos los valores de la red (*full integer quantization*²⁰), que observando el tamaño del fichero resultante, reduce su tamaño en cuatro respecto al fichero sin cuantizar, por lo que el ahorro en memoria que anticipábamos en la sección 2.8 se hace efectivo. Para llevar a cabo la cuantización es necesario estimar el rango de valores que pueden tomar los tensores del modelo. Aunque para los pesos, por ejemplo, estos valores sean fijos, no ocurre lo mismo con los tensores de entrada y de salida (pues son variables), y es necesario que se ejecuten unas primeras inferencias para calibrar estos valores. Para ello será necesario un *dataset* (de entre 100 y 500 imágenes) que contenga datos de entrada del modelo y con los que se obtendrá una aproximación de los valores de entrada y salida del modelo. Para generar estos conjuntos de datos (crearemos uno por modelo) simplemente creamos un entorno con el que interactuará el modelo (con la función `wrap_deepmind()` ya mencionada anteriormente) y vamos tomando de él observaciones fruto de interacciones aleatorias (al fin y al cabo sólo queremos un conjunto de imágenes del entorno). Esto lo realizamos con la secuencia de instrucciones que se muestra en el siguiente fragmento de código, extraído del *script*

¹⁷https://github.com/javigm98/Mejorando-el-Aprendizaje-Automático/blob/main/model_saver.py

¹⁸<https://www.tensorflow.org/lite/convert>

¹⁹https://github.com/javigm98/Mejorando-el-Aprendizaje-Automático/blob/main/tflite_converter.py

²⁰https://www.tensorflow.org/lite/performance/post_training_quantization

`dataset_creator.py`²¹. Generaremos de esta manera un *dataset* con imágenes de tamaño `dim` para uno de los modelos, que se almacenará en el fichero `dataset_name.npy`.

```

1 env = wrappers.wrap_deepmind(gym.make('Pong-v0'), dim=dim)
2 obs = env.reset()
3 with open(dataset_name + '.npy', 'wb') as f:
4     for _ in range(500):
5         np.save(f, obs)
6         action = env.action_space.sample()
7         obs, _, _, _ = env.step(action)

```

Una vez somos capaces de generar el *dataset* creamos el modelo cuantizado. Para ello, comenzamos leyendo los datos del *dataset* anteriormente generado:

```

1 images = []
2 with open(dataset_dir, 'rb') as f:
3     for _ in range(500):
4         images.append(np.load(f))

```

A continuación, definimos la función `representative_data_gen()`, que devuelve un generador con una muestra de 100 de los datos del conjunto:

```

1 def representative_data_gen():
2     for data in tf.data.Dataset.from_tensor_slices((images)).batch(1).take(100):
3         yield[tf.dtypes.cast(data, tf.float32)]

```

Ahora cargamos el modelo de `keras` que previamente habíamos guardado en un fichero con extensión `.h5` y lo convertimos a uno de *Tensorflow Lite* pero cuantizándolo. Para ello:

```

1 model = tf.keras.models.load_model(h5_dir, custom_objects={'tf':tf})
2 converter = tf.lite.TFLiteConverter.from_keras_model(model)
3 converter.optimizations = [tf.lite.Optimize.DEFAULT]
4 converter.representative_dataset = representative_data_gen
5 converter.target_spec.supported_ops = [tf.lite.OpsSet.TFLITE_BUILTINS_INT8]
6 converter.inference_input_type = tf.uint8
7 converter.inference_output_type = tf.uint8
8 tflite_model_quant= converter.convert()
9 open(tflite_dir, "wb").write(tflite_model_quant)

```

Todo este proceso lo llevamos acabo con el *script* `quantizer.py`²². A continuación, y para poder ejecutar el modelo en la TPU debemos compilarlo haciendo uso de la herramienta Edge TPU Compiler²³, que creará a partir del modelo `.tflite` cuantizado un modelo compatible con la TPU Google Coral. Podemos realizar esta tarea con la interfaz de línea de comandos `edgetpu_compiler`, por ejemplo con `edgetpu_compiler model_quant.tflite`, que generará un archivo `model_quant_edgetpu.tflite` que ya sí que reunirá todos los requisitos para poder ser ejecutado en la TPU. Como consecuencia de todo este proceso generamos varias versiones de cada modelo:

- `modelX.h5`: modelo de *Tensorflow keras*
- `modelX.tflite`: modelo de *Tensorflow Lite* equivalente, con tensores con valores `float32`.
- `modelX_quant.tflite`: modelo de *Tensorflow Lite* cuantizado, con tensores con valores `int8`.
- `modelX_quant_edgetpu.tflite`: modelo de *Tensorflow Lite* cuantizado y preparado para poder ejecutar inferencias sobre él en la TPU.

La figura 3.5 muestra todos los *scripts* que serán necesarios para poder llevar a cabo los experimentos de inferencia sobre la TPU. Los ficheros que contienen los modelos se encuentran en el directorio `exported_models`²⁴.

²¹https://github.com/javigm98/Mejorando-el-Aprendizaje-Automático/blob/main/dataset_creator.py

²²<https://github.com/javigm98/Mejorando-el-Aprendizaje-Automático/blob/main/quantizer.py>

²³<https://coral.ai/docs/edgetpu/compiler/>

²⁴https://github.com/javigm98/Mejorando-el-Aprendizaje-Automático/tree/main/exported_models

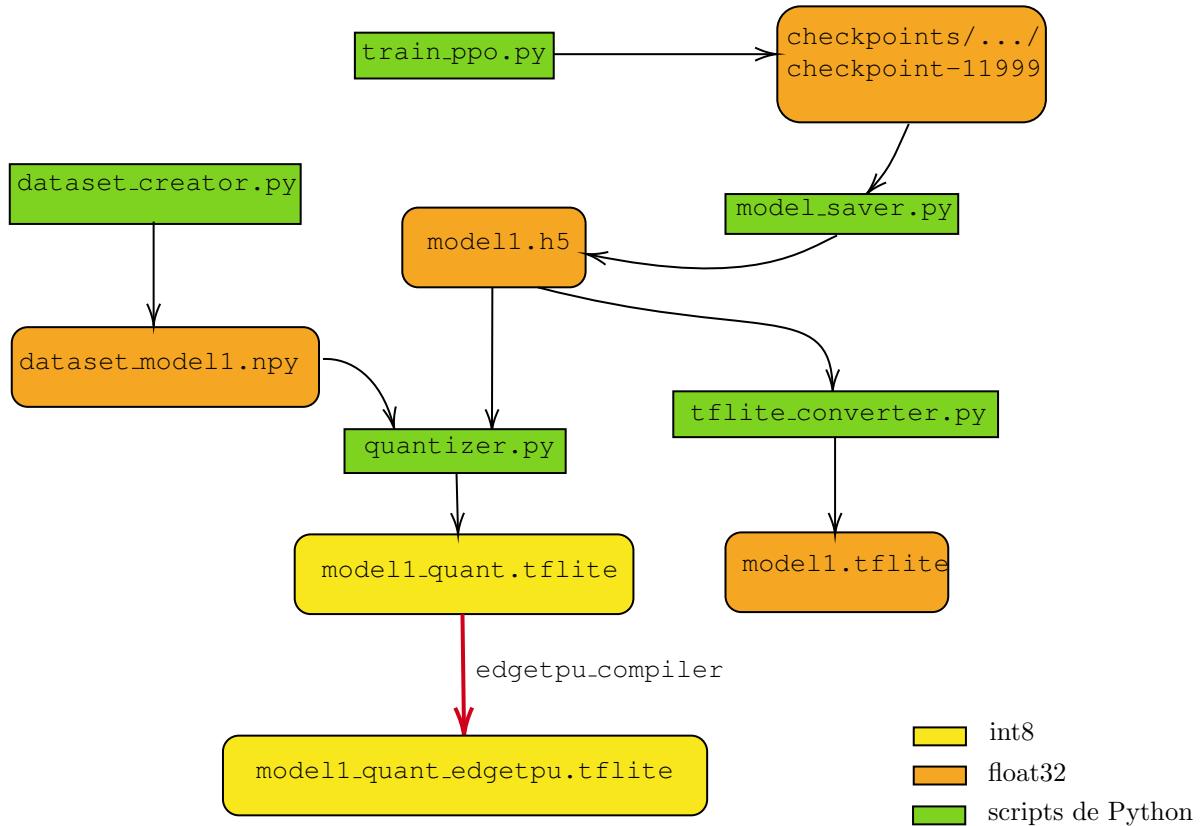


Figura 3.5: Relación entre los distintos archivos que contienen modelos guardados y los *scripts* de Python que realizan las conversiones y guardan el modelo resultante.

Una vez tenemos los tres modelos de *TFLite* podemos realizar inferencias sobre ellos y recolectar métricas sobre el tiempo empleado para su posterior análisis. Siguiendo el ejemplo de clasificación de imágenes²⁵ accesible en el repositorio de Google Coral, creamos el *script* *rollout_coral.py*²⁶, que cargará un modelo de *TFLite* cuantizado y compilado para poder ser ejecutado sobre la TPU y realizará tantos pasos de inferencia como indiquemos, reportando el tiempo empleado en realizarlo. Será necesario crear un intérprete de *TFLite*²⁷ en el que habilitamos la ejecución sobre la TPU por medio de un *delegado* de *TensorFlowLite*²⁸. Creamos este intérprete con la función que muestra el siguiente fragmento de código, dónde previamente hemos indicado las bibliotecas necesarias para la inferencia sobre la TPU en función del sistema operativo sobre el que estemos ejecutando:

```

1 EDGEPU_SHARED_LIB = {
2     'Linux': 'libedgetpu.so.1',
3     'Darwin': 'libedgetpu.1.dylib',
4     'Windows': 'edgetpu.dll'
5 }[platform.system()]
6
7
8

```

²⁵https://github.com/google-coral/pycoral/blob/master/examples/classify_image.py

²⁶https://github.com/javigm98/Mejorando-el-Aprendizaje-Automático/blob/main/rollout_coral.py

²⁷https://www.tensorflow.org/api_docs/python/tf/lite/Interpreter

²⁸<https://www.tensorflow.org/lite/performance/delegates>

```

9 def make_interpreter(model_file):
10    model_file, *device = model_file.split('@')
11    return tflite.Interpreter(
12        model_path=model_file,
13        experimental_delegates=[
14            tflite.load_delegate(EDGETPU_SHARED_LIB,
15                {'device': device[0]} if device else {})
16    ])

```

La estructura general del *script* para realizar las inferencias seguirá la idea del que nos proporciona *RLLib* para la misma tarea (*rollout.py*). Partiremos de un modelo y especificaremos o bien el número de pasos de inferencia o bien el número de episodios completos (partidas de tenis de mesa finalizadas) que queremos ejecutar sobre ese modelo. En cada paso de inferencia tendremos una imagen obtenida del entorno con el que estamos interaccionando, la colocaremos como tensor de entrada al modelo y lo invocaremos. De las dos salidas que produce el modelo nos quedaremos con la primera de ellas, que se corresponde con la salida de la red de la política y que cuenta con seis valores, cada uno de los cuales (tras aplicar la función *softmax*) representa la probabilidad de que tomando esa acción en ese estado mejoremos la recompensa global. Por ello, y tal y como se hace en *RLLib*, seleccionaremos como siguiente acción a tomar el máximo de estos valores. Una vez obtenida la acción, la ejecutamos sobre el entorno, obteniendo la siguiente observación a procesar (que representa el estado del entorno tras realizarse esa acción), la recompensa asociada a esa acción y si hemos concluido o no el episodio. En todo momento debemos asegurar que los valores de las imágenes deben ser del tipo aceptado por el modelo para sus entradas, por lo que antes de colocar el tensor como entrada del modelo hacemos la conversión de tipos si es necesario. Cada paso de inferencia ejecuta la siguiente secuencia de instrucciones:

```

1 start = time.perf_counter()
2 interpreter.invoke()
3 inference_time = time.perf_counter() - start
4 episode_times.append(inference_time)
5
6 output_data = interpreter.get_tensor(output_details[0]['index'])
7
8 action = np.argmax(output_data)
9
10 # Step environment and get reward and done information
11 image, reward, done, _ = env.step(action)
12
13 # Place new image as the new model's input
14 image = image[np.newaxis, ...]
15 if input_details[0]['dtype'] == np.float32:
16    image=np.float32(image)
17 if input_details[0]['dtype'] == np.uint8:
18    image=np.uint8(image)
19
20 interpreter.set_tensor(input_details[0]['index'], image)

```

Mediremos los tiempos de inferencia sobre cada uno de los 3 modelos cuantizados sobre la TPU. Además, y para ver la ganancia al usar este acelerador, elaboraremos un *script* similar para ejecutar inferencias sobre los modelos de *TFLite* en la CPU. Este *script*, *rollout_tflite.py*²⁹ sólo se diferencia de *rollout_coral.py* en que no habilita la ejecución sobre la TPU por medio de un *delegado* al crear el intérprete de *TFLite*.

²⁹https://github.com/javigm98/Mejorando-el-Aprendizaje-Automático/blob/main/rollout_tflite.py

Capítulo 4

Resultados

El objetivo del trabajo era analizar el rendimiento en diferentes arquitecturas de los procesos de entrenamiento e inferencia de modelos de aprendizaje por refuerzo. En este capítulo vamos a presentar los resultados obtenidos tras realizar varios experimentos, implementados siguiendo las pautas descritas en el capítulo anterior, y las conclusiones que de los resultados se derivan. Así, trataremos por separado los resultados obtenidos en cada uno de los dos procesos (entrenamiento e inferencia) considerados.

4.1. Resultados del proceso de entrenamiento

Analizaremos aquí el valor de varias **métricas** que *Ray* genera para cada iteración de entrenamiento. Estas métricas se referirán al **uso de recursos** y a los **tiempos empleados**, pero no a la capacidad de aprendizaje de los modelos, pues eso es algo que ya se tuvo en cuenta a la hora de seleccionar los modelos representativos. Y es que, aunque variemos las configuraciones de recursos en los distintos experimentos, la capacidad de aprendizaje de cada modelo será la misma, pues el algoritmo no varía (la red neuronal que entrenamos es la misma), y las únicas diferencias son el mayor o menor grado de paralelización de algunas de sus partes y el soporte hardware sobre el que se desarrolla este proceso. Es por ello que nos vamos a centrar en analizar aquellos factores del proceso de entrenamiento que sí se ven afectados cuando variamos la configuración de recursos del entrenamiento, esto es, el tiempo de ejecución de sus distintas fases y la utilización de los recursos de cómputo y almacenamiento durante el proceso. Presentamos los resultados gráficamente para que resulte más cómoda su interpretación y la comparación entre ellos, obtenidos a partir de los que podemos encontrar en formato tabular (archivos *csv*) en la carpeta *ray_results*¹ del repositorio de Github de este proyecto. Para cada experimento encontramos la salida que generará *Ray*, entre ellos los ficheros *progress.csv*, que contienen los valores de las métricas por cada iteración. Los valores que se presentan y analizan a continuación son la media de los obtenidos para cada una de las 20 iteraciones de entrenamiento, desecharlo las 3 o 4 primeras en el caso de los experimentos que usan alguna de las GPUs, que no reportan valores numéricos (el valor aparece como *NaN*) y se denominan **iteraciones de calentamiento**.

4.1.1. Uso de los recursos disponibles

Para evaluar el uso de los recursos empleados durante el proceso de entrenamiento, estudiamos los valores de las siguientes métricas relativas al **uso de las CPUs, las GPUs y la memoria RAM** del sistema, obteniendo con todas ellas porcentajes de utilización:

¹https://github.com/javigm98/Mejorando-el-Aprendizaje-Automatico/tree/main/ray_results

- `cpu_util_percent`: porcentaje medio de utilización del conjunto de las CPUs del sistema durante cada iteración de entrenamiento, por lo que los valores que obtendremos se encontrarán en el intervalo 0-100. Estos valores se obtienen apoyándose en la función `cpu_percent`² de la biblioteca de Python `psutil`.
- `ram_util_percent`: porcentaje medio de utilización de la memoria RAM total del sistema durante cada iteración de entrenamiento, así que los valores nuevamente se encontrarán entre 0 y 100. *Ray* usa la función `virtual_memory`³ de la biblioteca `psutil` para obtener los valores asociados a esta métrica.
- `gpu_util_percentX`: porcentaje medio de utilización de la GPU con identificador X durante cada iteración de entrenamiento. Viene dada por un valor en el intervalo 0-1.
- `vram_util_percentX`: porcentaje medio de utilización de la memoria de la GPU con identificador X durante cada iteración de entrenamiento (se mide el porcentaje del tiempo en el que se están realizando operaciones de lectura o escritura sobre la memoria). Viene dada por un valor en el rango 0-1.

Las dos últimas métricas hacen uso de la biblioteca `GPUUtil`⁴ (que será necesario que tengamos instalada en nuestro sistema), obteniendo información de las GPUs disponibles con `GPUUtil.getGPUs()` y para la lista de GPUs devueltas obtiene el porcentaje de uso (con el atributo `load`) y de utilización de la memoria de la GPU (con el atributo `memoryUtil`) de cada una de ellas. Es por eso, que al realizar las pruebas en el servidor *volta1* obtendremos resultados separados para cada una de las dos GPUs disponibles en el sistema.

Uso de la CPU

En la figura 4.1 podemos ver el **porcentaje de utilización** de la CPU para los tres modelos y para las distintas configuraciones probadas. Algunas conclusiones que obtenemos del análisis de los diagramas son las siguientes:

1. En primer lugar, observamos que, en general y para todas las configuraciones probadas, el porcentaje de uso de la CPU es bajo (el valor máximo se alcanza para el modelo 3 entrenando sin GPUs y está próximo al 16%). Hemos de tener en cuenta que el sistema sobre el que estamos realizando los experimentos dispone de una gran capacidad de cómputo y que puede dar soporte a las tareas necesarias para el entrenamiento de modelos con *RLlib* sin mayor problema.
2. El primer patrón que destacamos y que parece repetirse en los tres modelos es que aumentar el número de *workers* aumenta también el porcentaje de uso de las CPUs, como queda reflejado en los resultados obtenidos para el entrenamiento con ambas GPUs y 2, 4, 8 y 16 *workers*. Esto tiene sentido, pues *Ray* asigna una CPU a cada *worker* cuando planifica el proceso de entrenamiento.
3. Observamos también que el uso de la GPU sólo para el *driver* (y que los *workers* interactúen con el entorno haciendo uso sólo de la CPU) conlleva un sobrecoste en el porcentaje de uso de la CPU, que por ligero que sea, no deja de repetirse en los tres modelos y para las tres configuraciones de GPUs probadas.
4. Respecto a los entrenamientos realizados haciendo uso de la CPU, los resultados son independientes para cada modelo. Mientras que en el modelo 1 este porcentaje es menor que en muchos

²https://psutil.readthedocs.io/en/latest/#psutil.cpu_percent

³https://psutil.readthedocs.io/en/latest/#psutil.virtual_memory

⁴<https://github.com/anderskm/gputil>

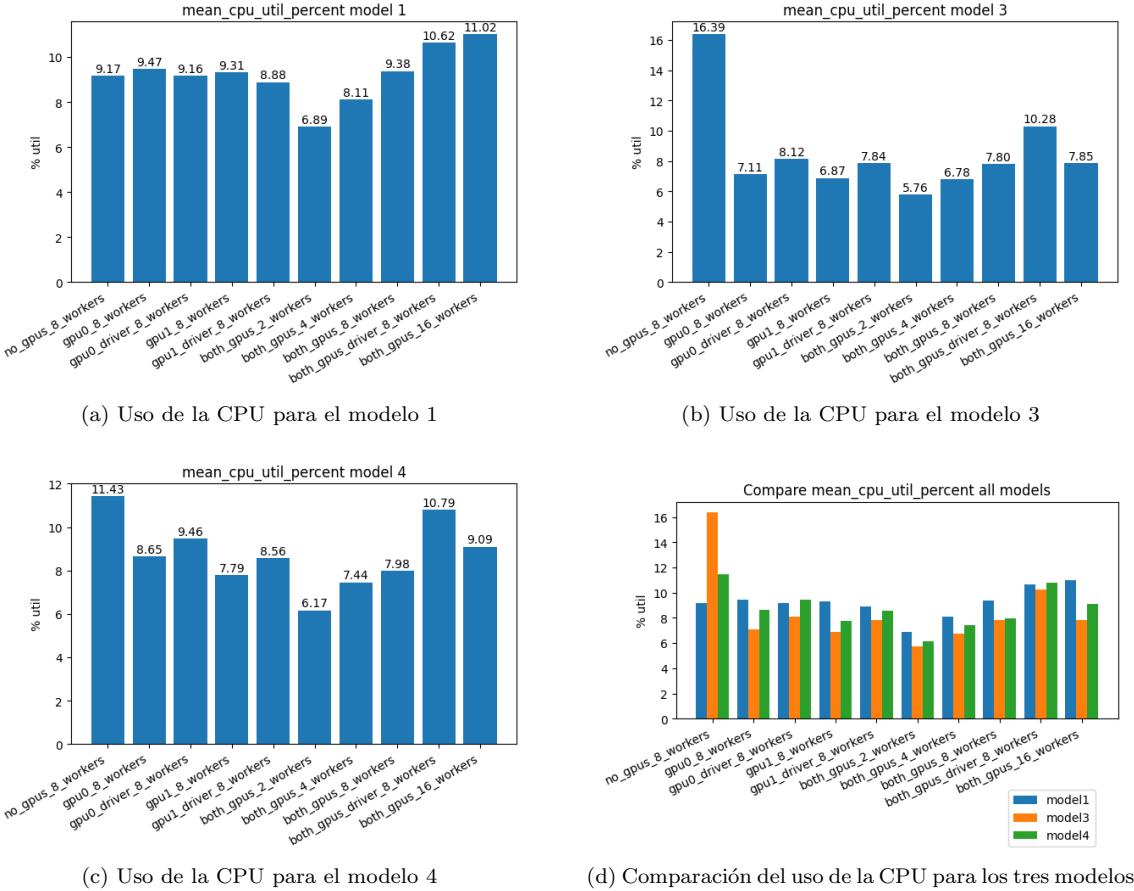


Figura 4.1: Porcentaje medio por iteración de entrenamiento de uso de la CPU del servidor *volta1* para diferentes configuraciones de los modelos 1, 3 y 4 tras 20 iteraciones de entrenamiento.

de los experimentos con GPUs y el mismo número de *workers* (8), en los modelos 3 y 4 sí que este valor es el mayor de los que se reportan. El modelo 3 es en el que la diferencia es más significativa, y este hecho puede deberse a que el tamaño de las imágenes que se procesan es mayor, y este procesamiento podría verse acelerado en mayor medida con el uso de las GPUs.

- Analizando la gráfica 4.1d, vemos que se sigue un patrón claro en los valores que obtenemos para cada experimento para cada uno de los modelos. Matizamos dos aspectos:

- Si observamos el experimento en el que no se usa ninguna de las GPUs, vemos que si ordenamos los modelos de mayor a menor porcentaje de CPU utilizado, tenemos en primer lugar al modelo 3, seguido del 4 y por último el 1, orden que se corresponde también con el del tamaño de las imágenes que toma cada modelo como entrada.
- En el resto de experimentos (en los que usamos una o ambas GPUs) vemos que el patrón es distinto. Así, observamos que los datos para los modelos 1 y 4 van casi a la par en muchos casos (hay más experimentos en los que el uso para el modelo 1 es ligeramente mayor) pero el modelo 3 es siempre el que reporta un porcentaje de utilización más bajo. Esta clasificación se corresponde con la que podríamos hacer si miramos el número de parámetros entrenables

de cada modelo (tabla 3.1), pues los modelos 1 y 4 tienen un número en torno a los 2 millones de pesos entrenables y este dato para el modelo 1 ronda los 1.1 millones.

Estas observaciones refuerzan la hecha en el punto anterior, y es que el hecho de introducir la GPU optimiza el procesamiento de estos datos de entrada de mayor tamaño, pasando el modelo 1 de ser el que peores resultados obtenía al que reporta unos porcentajes más bajos de uso. Además, una vez integramos el uso de las GPUs, la tendencia es que el porcentaje de utilización de la CPU sea mayor si el número de pesos que debemos ajustar en el modelo lo es, y este hecho no se ve condicionado por el tamaño de los datos de entrada.

En definitiva, podemos concluir que el uso de la CPU en un servidor como en el que hemos realizado los entrenamientos no supone un problema, pues **los valores obtenidos son relativamente bajos**. Además, para aquellos modelos que procesen imágenes de mayor tamaño, el uso de la CPU se ve reducido si añadimos también el uso de GPUs durante el proceso.

Uso de la memoria RAM

La figura 4.2 muestra el **porcentaje medio de utilización de la memoria RAM** del sistema durante el proceso de entrenamiento para los distintos modelos y configuraciones que venimos considerando. A continuación detallamos algunas observaciones que se desprenden de estos datos:

1. En este caso, las gráficas de los 3 modelos siguen un patrón idéntico, por lo que el hecho de comparar un experimento con otro no va a depender del modelo concreto.
2. Vemos que los valores más bajos en todos los casos se obtienen cuando sólo usamos las CPUs para el entrenamiento.
3. Además, para cada configuración de GPUs, el hecho de usarlas sólo para el *driver* reduce en más de la mitad este porcentaje frente a cuando las usamos también para los *workers*.
4. Como ocurría con el uso de la CPU, vemos una tendencia creciente, más marcada en este caso, pasando a multiplicar casi por cuatro el valor para 16 *workers* si lo comparamos con el de 2 *workers*. Para los 3 modelos el experimento con 16 *workers* utiliza, de media, más de el 50% de la memoria RAM disponible.
5. Se observan también ligeras diferencias en los resultados dependiendo de la configuración de GPU empleada en el experimento, observándose unos valores menores cuando usamos la GPU 1 (*Tesla-v100*).
6. Comparando los resultados obtenidos para los tres modelos, que aparecen reflejados en la gráfica 4.2d, observamos que las diferencias son poco significativas entre los distintos modelos, si bien en todos los experimentos el orden de los modelos de mayor a menor porcentaje de utilización de la RAM es el mismo: modelo 3, seguido del 4 y el modelo 1 por último. Este orden es el mismo que si ordenamos de mayor a menor tamaño de las entradas de la red neuronal.

Agrupando todas las ideas anteriores, podemos afirmar que **usar GPUs para el entrenamiento trae consigo un sobrecoste en la capacidad de RAM ocupada**, que se acentúa más cuando los *workers* hacen también uso de las GPUs y que se va incrementando según añadimos más *workers* que hacen uso de ella. Además, el uso es ligeramente mayor para los modelos que trabajan con observaciones de mayor tamaño.

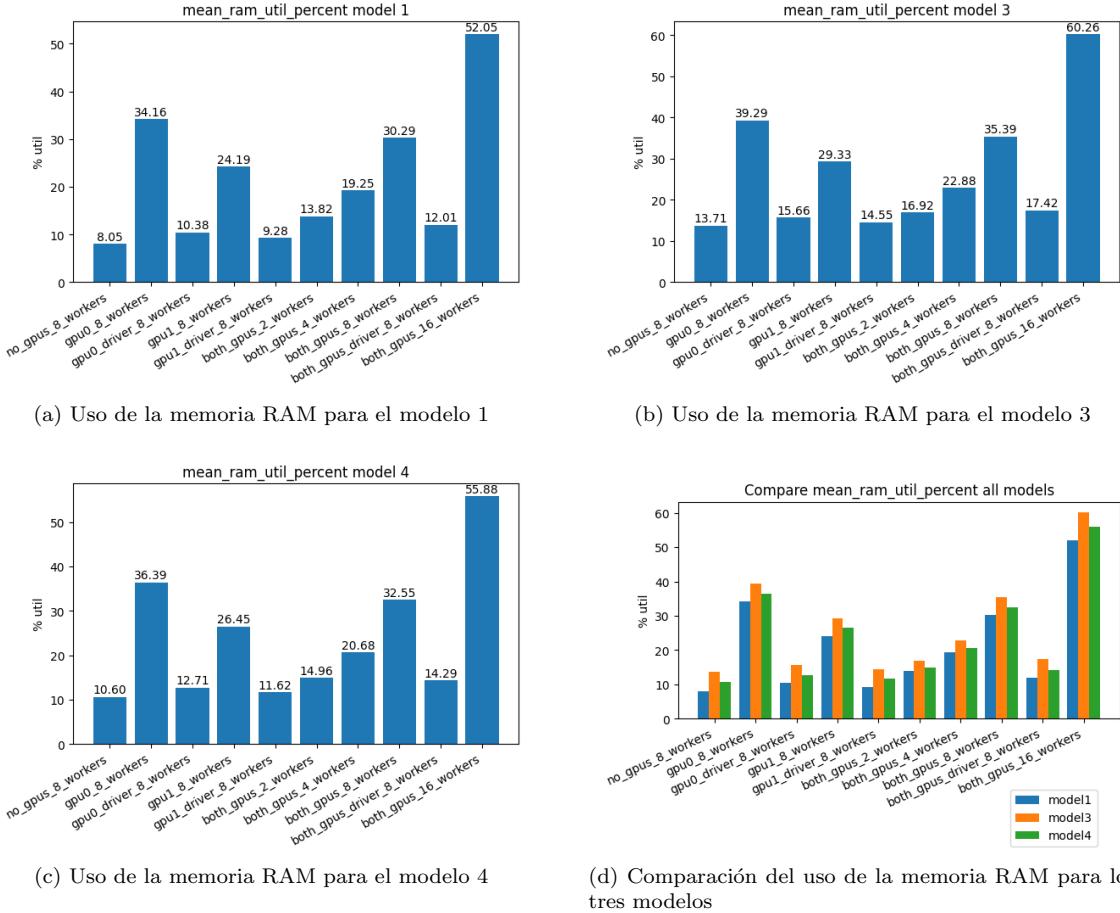


Figura 4.2: Porcentaje medio de uso de la memoria RAM del servidor *volta1* para diferentes experimentos realizados con los modelos 1, 3 y 4 y diferentes configuraciones durante 20 iteraciones de entrenamiento.

Uso de las GPUs y su memoria

La figura 4.3 muestra los resultados obtenidos relativos a la **utilización de las GPUs** disponibles en el servidor *volta1* durante las iteraciones de entrenamiento y también sobre el **porcentaje de tiempo que se accede a la memoria** de estas GPUs. Las métricas que representamos aquí son *gpu_util_percent0* (utilización de la GPU RTX), *gpu_util_percent1* (utilización de la GPU Tesla-v100), *vram_util_percent0* (accesos a memoria en la GPU RTX) y *vram_util_percent1* (accesos a memoria en la GPU Tesla-v100), que, aunque *Ray* las reporte con valores en el intervalo [0, 1], se representan tomando valores en [0, 100] para facilitar su interpretación. Analizando los datos podemos extraer las siguientes conclusiones:

1. El porcentaje de uso de las GPUs es bastante elevado, sobre todo cuando se usa sólo una de ellas compartida entre el *driver* y los *workers* (entre 50 % y 60 %). Además, cuando su uso se reserva sólo al *driver*, como es lógico, su porcentaje de utilización baja, pues durante la fase de *sampling* no se usa la GPU.

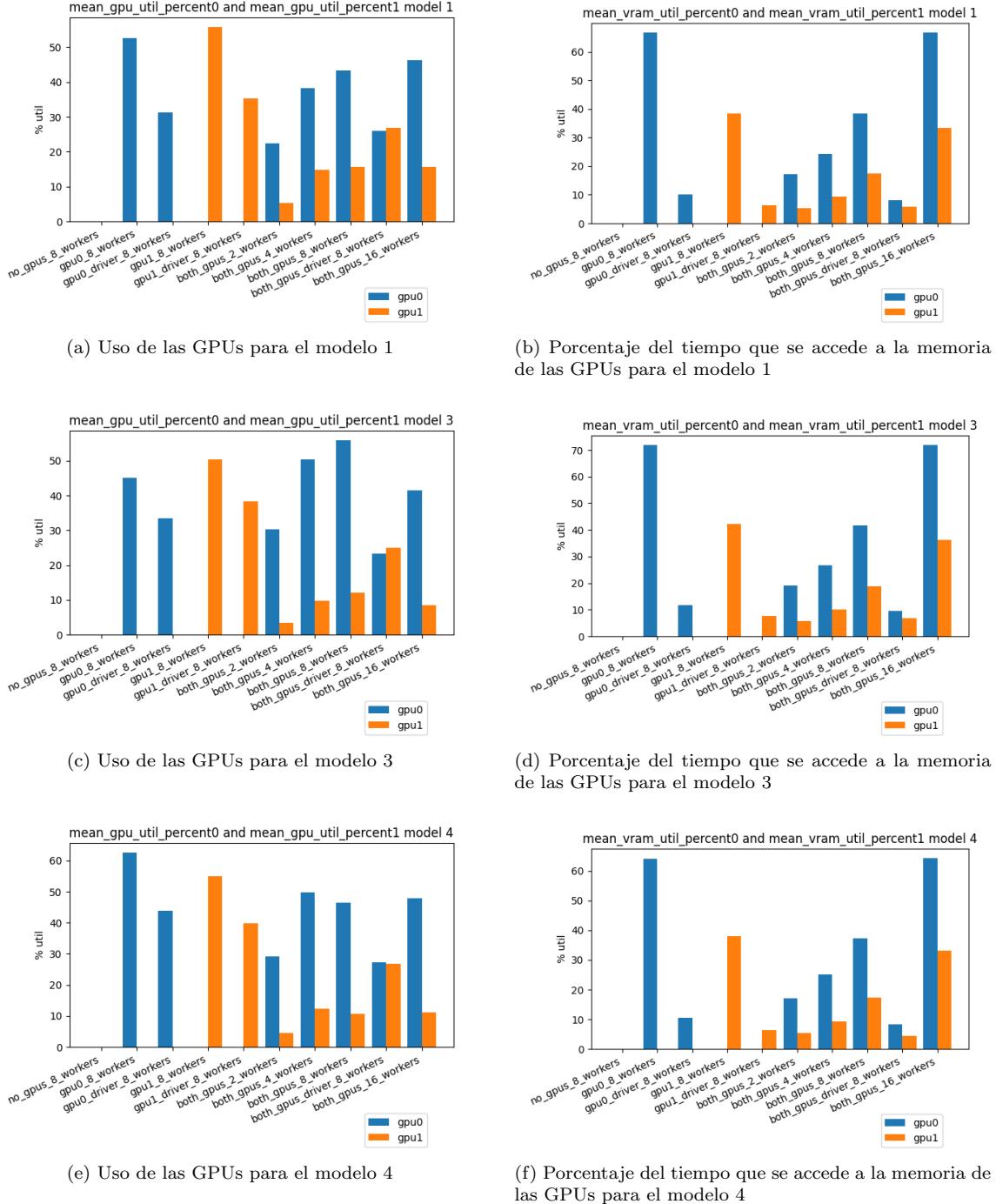


Figura 4.3: Porcentaje medio de uso (izquierda) y porcentaje medio del tiempo que se accede a la memoria (derecha) de ambas GPUs del servidor *volta1* para diferentes experimentos realizados con los modelos 1, 3 y 4 y diferentes configuraciones durante 20 iteraciones de entrenamiento.

2. En los experimentos en los que se usan ambas GPUs tanto para el *driver* como para los *workers* el porcentaje de uso de la GPU RTX es bastante mayor que el de la GPU Tesla-v100. Cuando excluimos a los *rollout workers* del uso de las GPUs, el porcentaje de uso de ambas sí que se equilibra más.
3. En cuanto al porcentaje de tiempo que se accede a la memoria de las GPUs este disminuye notablemente en todos los casos cuando la GPU se usa sólo para el *driver*, lo que nos indica que cuando usamos la GPU para los *rollout workers* los accesos a memoria de estos son bastante significativos.
4. En general, el porcentaje de acceso a memoria cuando se usa únicamente la GPU RTX es mayor que cuando sólo se usa la GPU Tesla-v100.
5. Al igual que ocurría con el porcentaje de utilización de las GPUs, el porcentaje de accesos a memoria es mayor en la GPU RTX que en la Tesla-v100 cuando ambas se usan simultáneamente.
6. Por último observamos que si bien el número de *workers* que se crean no tiene un efecto claro en el uso de la GPUs (en los modelos 1 y 4 sí que se observa cierto incremento si aumentamos el número de *workers*), en el porcentaje de accesos a memoria sí que observamos una correlación clara entre aumentar el número de *workers* y que aumenten los accesos a memoria.

Concluyendo, podemos afirmar que **el porcentaje de uso de las GPUs es alto** en la mayoría de los casos, que **los *workers* hacen un uso considerable** de la misma cuando se les permite usarla, incrementando los accesos a memoria según aumentamos los *workers* que se crean y que el hecho de usar ambas GPUs simultáneamente crea una mayor carga de trabajo en la GPU RTX que en la GPU Tesla-v100, teniendo en cuenta el porcentaje total de la capacidad de cada una que está en uso durante este proceso.

4.1.2. Tiempos empleados

Analizaremos en este apartado los valores obtenidos en los diferentes experimentos realizados para los **temporizadores de las distintas fases** de cada iteración del algoritmo de entrenamiento PPO. Así, consideraremos los valores de estas cuatro métricas que reporta *Ray* y que se corresponden con cada una de las cuatro fases que tiene la implementación que se hace en *RLlib* del algoritmo PPO (ver imagen 2.8):

- **sample_time_ms**: tiempo (en milisegundos) que emplean los *rollout workers* en obtener del entorno los datos necesarios para la actualización de la política. Cuando analizamos la implementación que hacia *RLlib* del algoritmo veíamos que en total los *workers* debían realizar **train_batch_size** interacciones con el entorno, tomando cada *worker* series de **rollout_fragment_length** pasos. Esta fase se ejecuta de manera paralela entre los distintos *workers*, en caso de haberlos.
- **load_time_ms**: tiempo (en milisegundos) que se emplea en cargar y concatenar las experiencias recolectadas por los *workers* en el *driver* antes de comenzar la fase de actualización de la política.
- **learn_time_ms**: tiempo (en milisegundos) que emplea el *driver* en completar una etapa de actualización del modelo que estamos entrenando (computar una serie de iteraciones (por defecto 30, vienen dadas por el parámetro de configuración **num_sgd_iter**) del algoritmo de descenso de gradiente) que nos dará unos nuevos valores para los pesos de la red neuronal de convolución del modelo.
- **update_time_ms**: tiempo (en milisegundos) que se emplea en actualizar el modelo en los *workers* (actualizar los pesos en la red neuronal) tras finalizar la etapa de aprendizaje en el *driver*.

Además de analizar los valores para estos cuatro temporizadores, recogeremos también el valor del **tiempo total por cada iteración** de entrenamiento (que es la suma de estos cuatro valores) y que *Ray* reporta en la métrica `time_this_iter_s`. Además, veremos el desglose de este tiempo en las cuatro etapas, viendo cuáles ocupan mayor o menor fracción del tiempo total.

Tiempo de interacción con el entorno

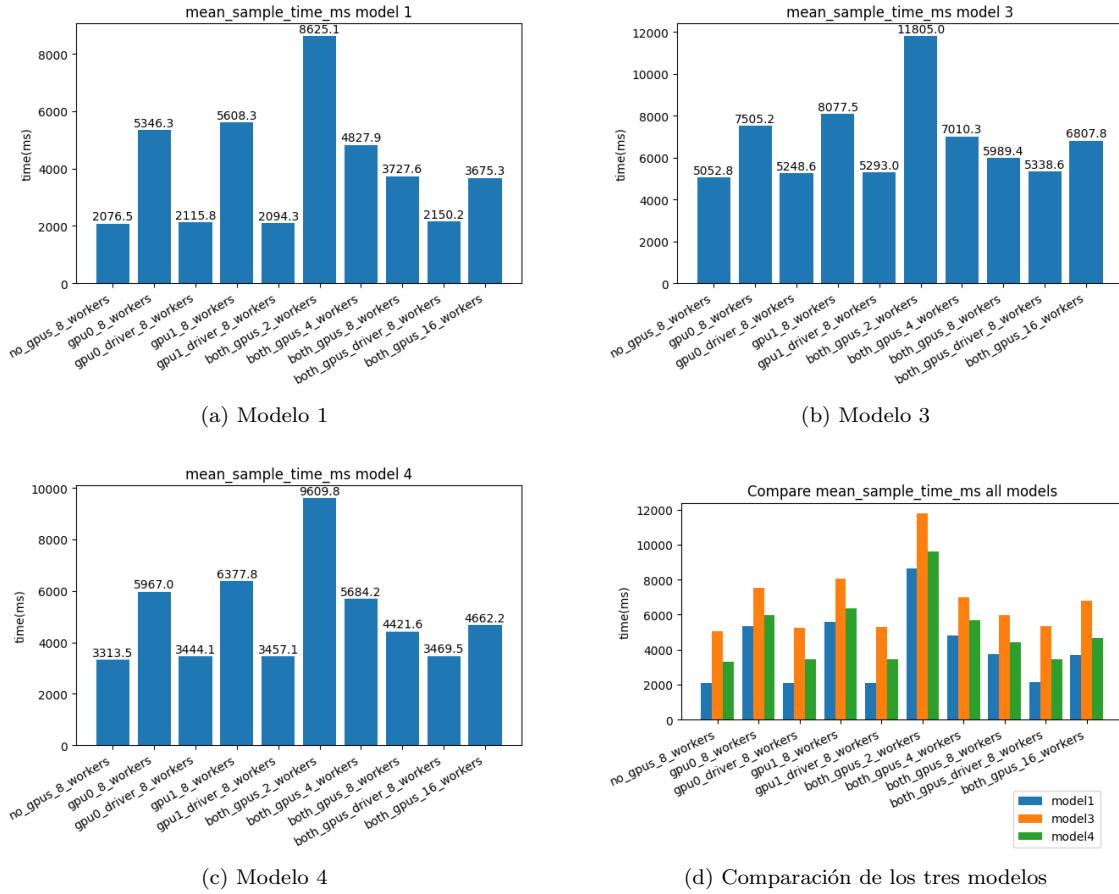


Figura 4.4: Tiempos promedios por iteración (en milisegundos) que se emplea en cada una de las configuraciones probadas y para cada uno de los modelos en recoger experiencias del entorno por los *rollout workers*, tras realizarse 20 iteraciones de entrenamiento en el servidor *volta1*.

La figura 4.4 muestra los valores obtenidos en cada experimento para la fase de interacción y obtención de datos con el entorno por parte de los *rollout workers*. Podemos observar lo siguiente mirando a las gráficas que se presentan:

1. Los tres modelos siguen el mismo patrón para los distintos experimentos, por lo que el hecho de comparar los resultados en un experimento con los de otro, en general, no dependerá del modelo.
2. Observamos que el hecho de no usar la GPU para los *workers* mejora significativamente este tiempo, pues, tanto en los casos en los que no se usa GPU como en los que se usa pero sólo para

el *driver*, estos valores son muchos más bajos. En los experimentos en los que se usa GPU para los *workers* se observa que cuando se usan ambas este tiempo es menor que cuando se usa una sola.

3. El número de *workers* que se usan en esta fase también influye en el tiempo de *sampling*. Así, los valores máximos se obtienen para los experimentos con 2 *workers*, disminuyendo este valor de manera progresiva para 4 y 8 *workers*. Esta observación tiene sentido, pues el volumen total de información que se debe recolectar es el mismo, por lo que si añadimos más *workers* esta tarea estará más paralelizada y por tanto es menor. Sin embargo, con 16 *workers* el dato no mejora si lo comparamos con el de 8 *workers*, quizás debido a que con 8 *workers* ya es suficiente para una buena paralelización en la recogida de experiencias.
4. Centrándonos en la gráfica 4.4d, vemos que para todos los experimentos, es el modelo 3 el que requiere más tiempo para esta fase, seguido del 4 y del 1. Este orden es, nuevamente, el que obtenemos si ordenamos los modelos por el tamaño de las entradas que reciben, por lo que podemos pensar que el tiempo que tardamos en aplicar el modelo depende directamente del tamaño de las observaciones que obtenemos del entorno.

Los datos de esta fase son un buen indicador de los que obtendremos cuando analicemos la **inferencia** de los modelos, pues aquí lo que se llevan a cabo son varias inferencias (además no son inferencias directas, pues es aquí donde se exploran nuevas posibilidades y no siempre se toma la acción con mayor valor para la política en ese estado). En resumen, el tiempo de recogida de experiencias es mejor cuando este proceso se realiza sólo en las CPUs, va mejorando con el número de *workers* (estabilizándose en 8 *workers*) y aumenta si las imágenes que obtenemos del entorno son de mayor tamaño.

Tiempos de carga de las experiencias en el *driver*

En la figura 4.5 podemos ver la representación gráfica de los datos obtenidos para la medición del tiempo de la fase de carga en el *driver* de las experiencias recolectadas en la fase anterior para usarlas para actualizar la red neuronal del modelo. Aquí llegan fragmentos de datos de todos los *workers* y se concatenan en uno sólo fragmento de longitud `train_batch_size` que será el que se emplee para el aprendizaje. Algunas conclusiones que obtenemos son:

1. Las gráficas no muestran diferencias muy marcadas entre los tiempos obtenidos en los distintos experimentos para cada modelo, aunque sí que hay pequeños matices que se repiten para todos los modelos.
2. En todos los casos, donde más tiempo se emplea es en el caso en el que tenemos 16 *workers*.
3. El uso de la GPU (ya sea para los *workers* o para el *driver*) no supone una mejora directa en los tiempos que se obtienen. Aun así, con los modelos 3 y 4 se obtienen valores relativamente más bajos cuando no se usan GPUs.
4. Se aprecia también una mejora ligera en el experimento con 4 *workers* y ambas GPUs respecto a los experimentos que usan otro número de *workers*.
5. Comparando los tres modelos (figura 4.5d) se ven diferencias muy significativas entre los tiempos para todos los experimentos. El modelo 3 emplea casi 10 veces más tiempo en esta fase que el modelo 1, y casi el doble que el modelo 4. Establecemos así una correlación directa entre el tamaño de las imágenes que toma la red neuronal como entrada y el tiempo empleado en cargar los datos obtenidos del entorno en el *driver*. Este hecho tiene sentido, pues si los datos que hay que cargar son en su mayoría imágenes de un tamaño mayor, el tiempo en realizar esta carga será también más elevado.

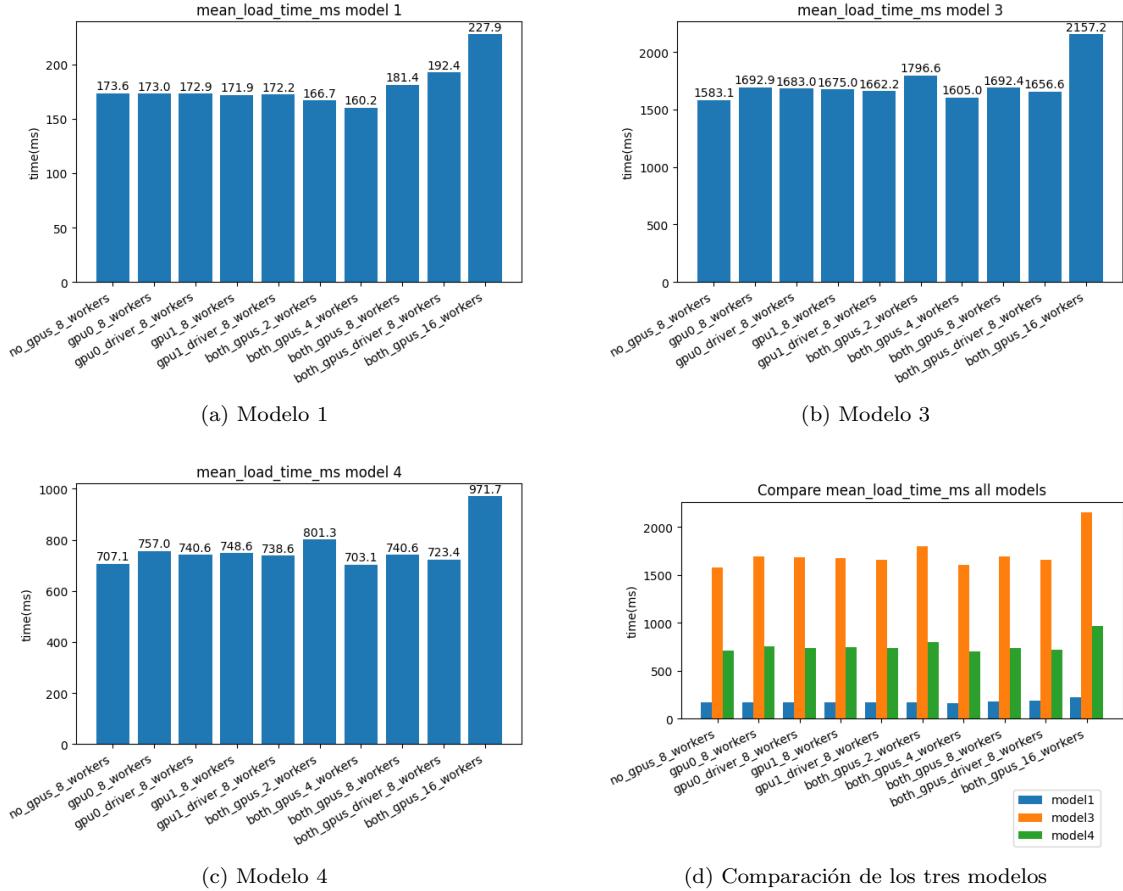


Figura 4.5: Tiempos promedios por iteración (en milisegundos) que se emplea en cada una de las configuraciones probadas y para cada uno de los modelos en cargar las experiencias recolectadas por los *rollout workers* en el *driver*, tras realizarse 20 iteraciones de entrenamiento en el servidor *volta1*.

Así, como conclusión, podemos destacar el hecho de que en este caso la configuración de las GPUs o el número de *workers* que se usan en el algoritmo no es un factor realmente determinante en el tiempo empleado en cargar los modelos, algo que sí que es el tamaño de las imágenes con las que trabaja cada modelo, siendo mayor este tiempo si las dimensiones de las imágenes aumentan.

Tiempos de aprendizaje en el *driver*

Se presentan en la figura 4.6 los tiempos de aprendizaje en cada iteración del algoritmo, esto es, de la fase de actualización de la red neuronal que modela la política y su valor que tiene lugar en el *driver*. Aunque *Ray* reporte los valores para esta métrica en milisegundos, debido a la magnitud de los datos, por comodidad y para que sea más clara su presentación en las gráficas aparecen los valores en segundos. Además en este caso será muy importante tener en cuenta los valores numéricos asociados a cada barra, pues el hecho (que ahora analizaremos) de que haya tanta diferencia entre el experimento sin GPU y el resto hace que la diferencia de tamaño de las barras en los experimentos sea prácticamente imperceptible. Es por ello que incluimos también la figura 4.7, en la que se incluyen

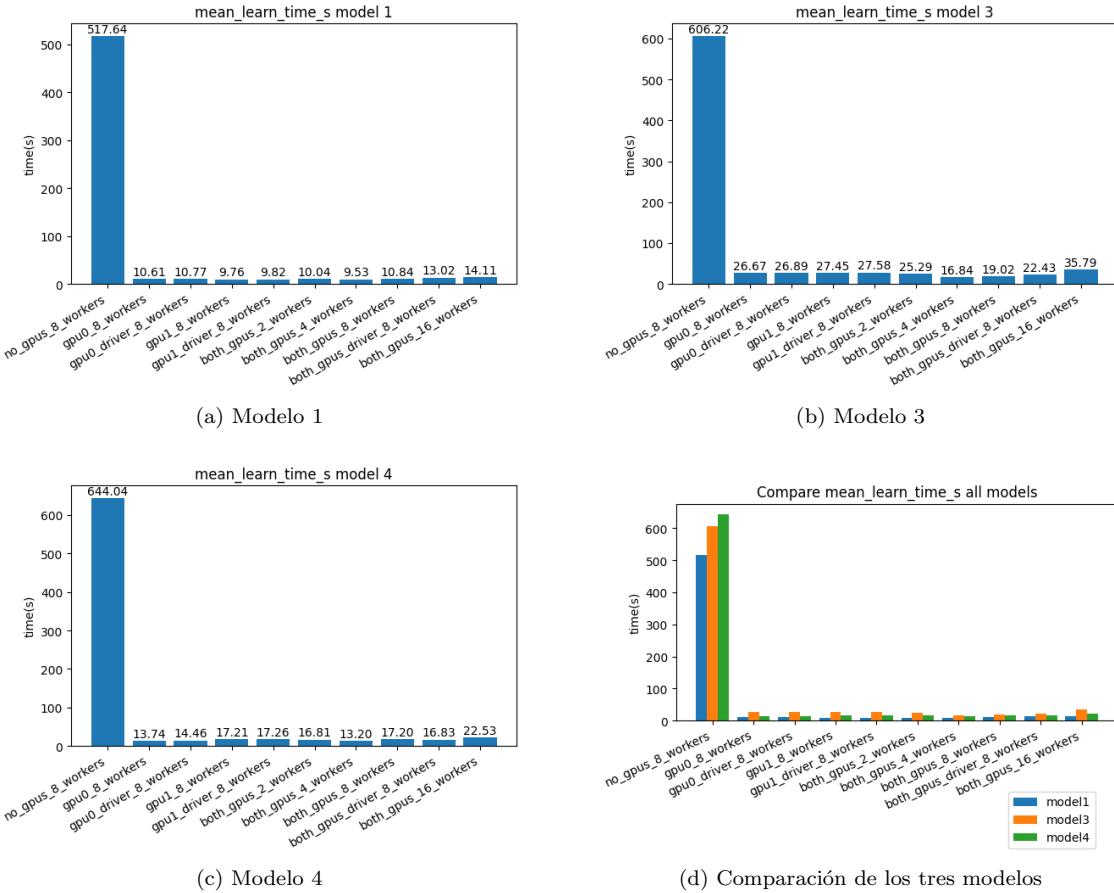


Figura 4.6: Tiempos promedios por iteración (en segundos) que se emplea en cada una de las configuraciones probadas y para cada uno de los modelos en computar una etapa de descenso de gradiente con las experiencias recolectadas, tras 20 iteraciones de entrenamiento en el servidor *volta1*.

los datos de tiempo de aprendizaje para todos los experimentos en los que se hace uso de la GPU (no aparece el experimento *no_gpus_8_workers*). Así, remarcamos:

1. En primer lugar, y como ya se ha anticipado, la característica más destacable de las gráficas es la diferencia abismal que hay en el tiempo de aprendizaje entre los experimentos que usan las GPUs y el que realiza todo el proceso de entrenamiento sólo con CPUs. Los *speedups* que se consiguen respecto al experimento con peores resultados usando GPU (*both_gpus_16_workers*) son bastante elevados (36.39 para el modelo 1, 16.93 para el modelo 3 y 28.58 para el modelo 4).
2. Como hemos anticipado en el punto anterior, en los tres casos es el experimento con 16 *workers* aquel que emplea más tiempo en esta fase.
3. Los valores más bajos se obtienen para el experimento con 4 *workers* y ambas GPUs para los tres modelos.
4. Respecto al uso de cada una de las GPUs, mientras que para el modelo 1 se obtienen mejores resultados cuando se usa la GPU Tesla-v100 que la GPU RTX, para el modelo 3 los resultados

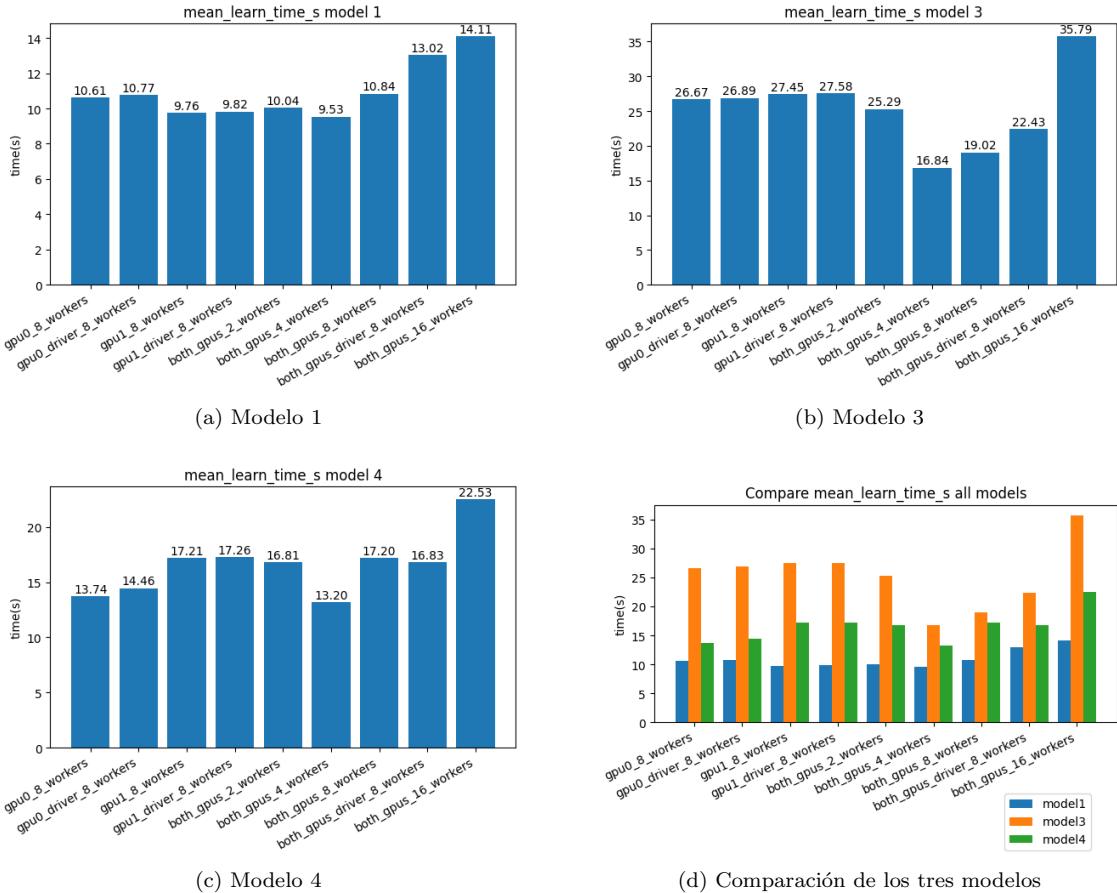


Figura 4.7: Tiempos promedios por iteración (en segundos) que se emplea en cada una de las configuraciones probadas y para cada uno de los modelos en computar una etapa de descenso de gradiente con las experiencias recolectadas, tras 20 iteraciones de entrenamiento en el servidor *volta1*, sin incluir el experimento en el que no se usan GPUs.

son muy similares (ligeramente mejores para la RTX) y en el modelo 3 los valores obtenidos son más bajos cuando se usa la RTX.

5. El hecho de usar ambas GPUs sólo mejora a los dos casos en los que se usa una de ellas para el modelo 3.
6. Comparando los 3 modelos (gráficas 4.6d y 4.7d), realizamos dos observaciones:
 - Cuando no se usa GPU es el modelo 4 el que consume más tiempo en la fase de aprendizaje de cada iteración, seguido del modelo 3 y del 1. Vemos aquí que no existe una correlación directa entre el número de parámetros entrenables o el tamaño de las imágenes y este dato, sino que ambos influyen. Así, el modelo 4 suma que el tamaño de las imágenes es el doble que el del modelo 1 y además tiene casi el doble de parámetros que ajustar que el modelo 3 (2 millones frente a 1.1).

- En los experimentos que hacen uso de la GPU sí que se observa la relación directa que veíamos en otros casos entre tamaño de las observaciones y tiempo empleado.

Sobre el tiempo de aprendizaje por iteración de los modelos, podemos concluir que el uso de GPUs reduce su valor en gran medida, además, en este caso el tamaño de las entradas de la red neuronal determinará este tiempo.

Tiempo de actualización de los modelos

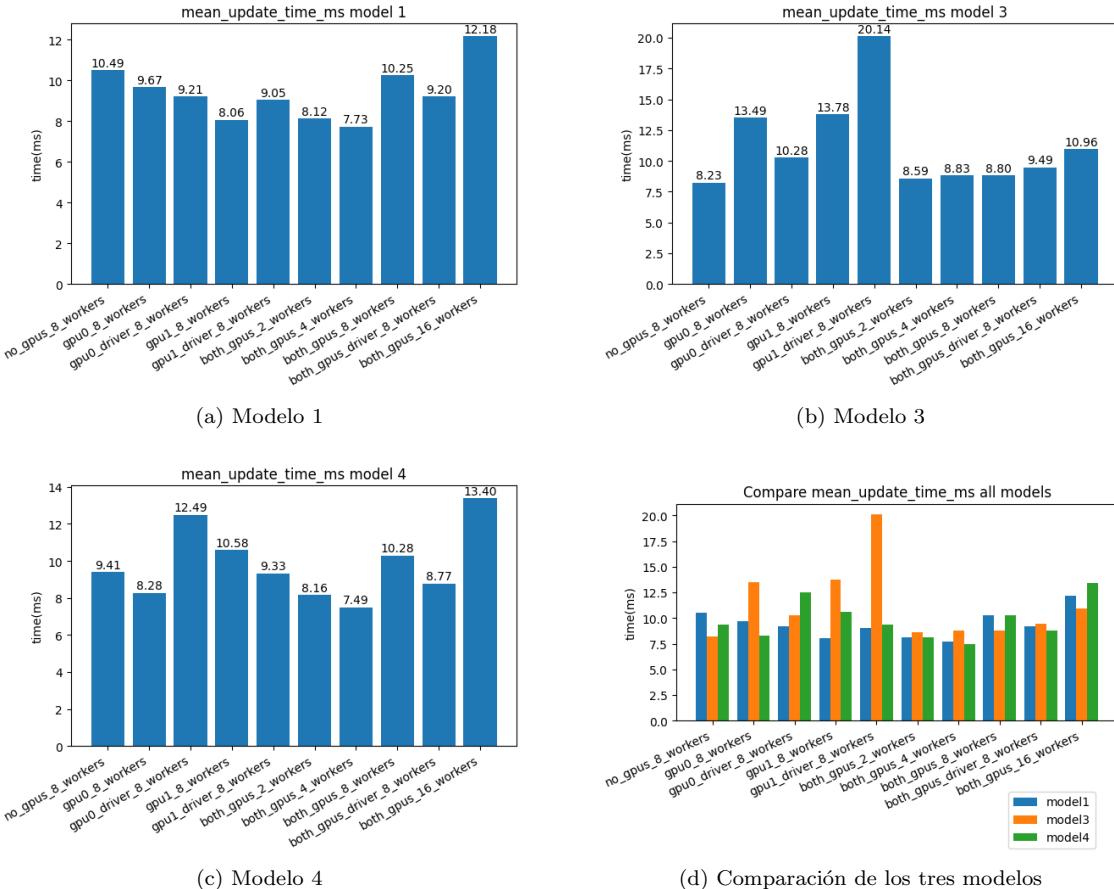


Figura 4.8: Tiempos promedios por iteración (en milisegundos) que se emplea en cada una de las configuraciones probadas y para cada uno de los modelos en actualizar la política del agente (los pesos de la red neuronal) en los *workers*, para la siguiente iteración de entrenamiento, tras ejecutarse 20 iteraciones de entrenamiento en el servidor *volta1*.

Nos centraremos ahora en analizar los resultados obtenidos para los tiempos que se emplean en los distintos experimentos en actualizar los parámetros de la red neuronal una vez calculados sus nuevos valores, partiendo de los datos que representamos en la figura 4.8:

- La primera observación que realizamos está en los valores que toma esta métrica, entre los 8 y los 20 milisegundos, que son bastante bajos.

2. Cada modelo parece seguir su propio patrón en lo que respecta a estos valores.
3. Sí que se aprecia que en general la tendencia parece ser ascendente si aumentamos el número de *workers*, aunque aun en los modelos 1 y 4 el resultado con 4 *workers* es mejor que con 2.
4. Respecto al uso de la GPU exclusivamente para el *driver* o la compartición de la misma entre *workers* y *driver* los tiempos varían dependiendo del modelo y de la GPU empleada.
5. Comparando los tres modelos tampoco se observa una correlación clara entre el número de valores a actualizar (pesos) y el tiempo que se tarda en llevar a cabo esta actualización, siendo en muchos experimentos mucho mayor el tiempo para el modelo 3, que, aunque sea el que trabaja durante todo el proceso con datos de entrada mayores, es el que tiene pesos que actualizar en su red neuronal.

Así, concluimos que los resultados obtenidos para el tiempo de actualización de los modelos arrojan que esta fracción del tiempo es muy pequeña comparada con el resto y que los valores no dependen de manera clara de la configuración de recursos o del modelo en cada experimento.

Tiempo promedio por iteración

La figura 4.9 recoge datos acerca de los tiempos empleados en cada iteración para los distintos experimentos. Las gráficas de la izquierda muestran los datos absolutos y su desglose en las cuatro etapas que hemos analizado previamente, donde se ha omitido el experimento en el que no se usan GPUs para que la diferencia entre los datos representados sea apreciable a simple vista. A la dercha, se representa la distribución de los cuatro valores analizados para cada experimento, mostrándose como porcentajes sobre el total del tiempo empleado, incluyéndose ya aquí el experimento `no_gpus_8_workers` puesto que lo que representamos son porcentajes y no valores absolutos. Analizando detenidamente los datos, podemos remarcar:

1. La fase de aprendizaje es la que ocupa la mayor parte del tiempo de cada iteración de entrenamiento, seguida de la fase de recogida de experiencias del entorno (*sampling*), que también ocupa una fracción considerable. Más residuales son los tiempos empleados para la carga de los datos de la fase *sampling* en el *driver* y la de actualización de los modelos, ocupando esta última una fracción despreciable del tiempo total de las iteraciones.
2. Vemos que, en general, usar la GPU sólo para el *driver* mejora los tiempos obtenidos. Este hecho era algo que ya venía produciéndose para cada uno de los tiempos considerados de manera independiente, por lo que, como es normal, se repite cuando consideramos el tiempo total.
3. Nuevamente y como ya habíamos indicado en algunos casos, cuando variamos el número de *workers*, los mejores resultados se obtienen cuando este número es 4.
4. El uso de una u otra GPU parece no tener influencia en los resultados obtenidos. Sin embargo, para los modelos 1 y 3 vemos que usar ambas simultáneamente mejora el dato de tiempo que se obtiene con cada una de ellas por separado.
5. Mirando ahora las gráficas de porcentajes de distribución del tiempo vemos como cuando no usamos GPU prácticamente la totalidad del tiempo se emplea en la fase de aprendizaje.
6. Comparando los valores numéricos que toman los datos de tiempo para cada modelo, observamos que el que más tiempo consume por iteración es el modelo 3, a continuación el 4 y por último el 1. Esto es, los modelos que procesan imágenes más grandes son aquellos que más tiempo tardan en entrenarse.

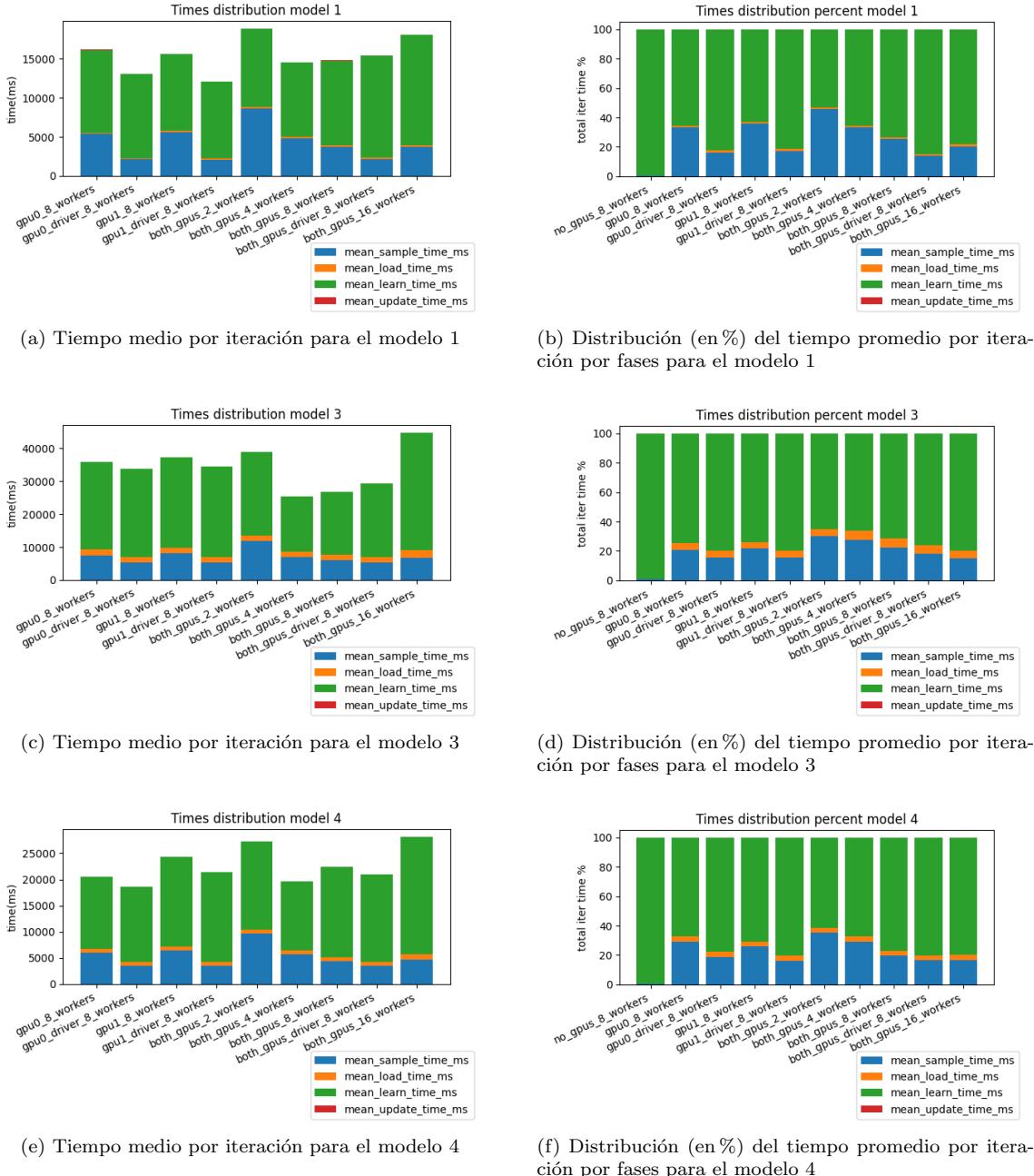


Figura 4.9: Tiempo total promedio por iteración de entrenamiento (izquierda) y distribución de ese tiempo (en porcentaje) en cada una de las etapas de las iteraciones de entrenamiento (derecha) para distintas configuraciones probadas sobre los modelos 1, 3 y 4, tras 20 iteraciones de entrenamiento en el servidor *volta1*.

En definitiva, vemos aquí corroboradas todas las conclusiones que veníamos extrayendo en los análisis anteriores. Destacamos así la importancia del uso de las GPUs para reducir abruptamente el tiempo de entrenamiento (pues se reduce el tiempo de la fase de aprendizaje que ocupa la mayor parte del tiempo en todos los casos), el uso de la GPU sólo para el *driver* para reducir también estos tiempos (se reduce notablemente el tiempo de *sampling*) y que cuando usamos 4 workers los tiempos son en general mejores.

4.1.3. Análisis del uso que se hace de las distintas CPUs

Analizamos en este apartado los resultados obtenidos en los distintos experimentos que presentábamos en la tabla 3.3 y que tenían por objetivo evaluar si tenía algún efecto sobre el coste en tiempo y la utilización de recursos las restricciones que podíamos imponer a *Ray* en la **utilización de las CPUs** del sistema, recogiendo las métricas que venímos tratando para experimentos en los que no imponemos restricciones en este sentido (como los que ya hemos analizado), otros en los que indicamos a *Ray* el número de CPUs con los que debe planificar las tareas que crea para la ejecución del algoritmo y por último probamos a restringir las CPUs del sistema que se pueden usar a sólo unas específicas de manera externa a *Ray*.

En primer lugar podemos observar en la figura 4.10 como afectan estas configuraciones al porcentaje total de **uso de la CPU** del sistema que se usa y a la **ocupación de la memoria RAM** durante el proceso de entrenamiento.

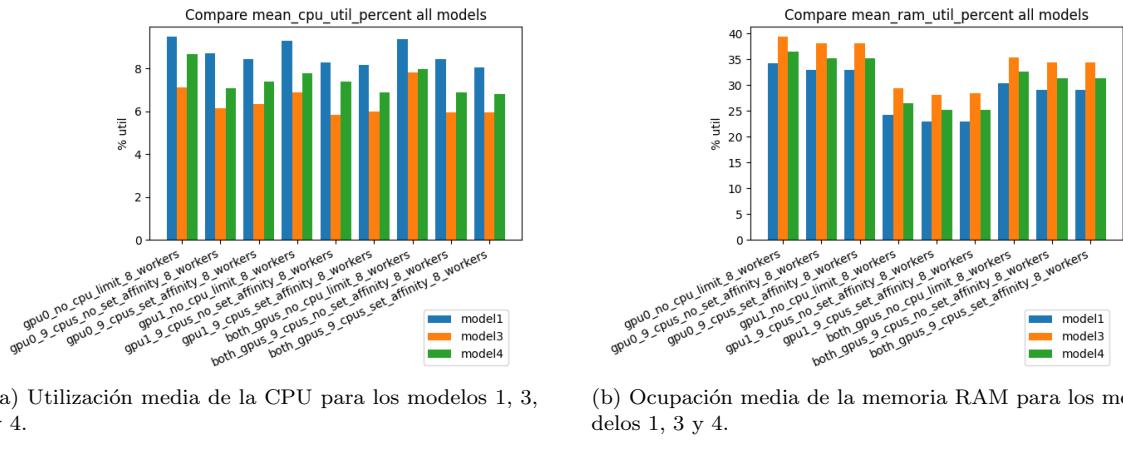


Figura 4.10: Utilización de los recursos del servidor *volta1* para las distintas configuraciones de uso de las CPUs probadas sobre los modelos 1, 3 y 4, tras 20 iteraciones de entrenamiento.

Analizando los resultados, vemos a grandes rasgos que las distintas configuraciones, en lo que a la gestión del uso de las CPUs respecta, no afectan en el porcentaje de utilización total de las CPUs y de la memoria RAM de manera significativa. Sí que es cierto que fijándonos en la gráfica 4.10a observamos que el porcentaje de uso de la CPU total del sistema es ligeramente menor en aquellos casos en los que indicamos a *Ray* que el número de CPUs de las que dispone es de 9 (una para el *driver* y el resto para los *workers*), pero la variación es mínima (en torno al 1-2 %). En cuanto a la ocupación de memoria RAM las variaciones son prácticamente imperceptibles.

En las gráficas de la figura 4.11 vemos el efecto que tiene esta manera de configurar el uso de las CPUs sobre el **tiempo promedio por iteración**, desglosado en fases. Como podemos observar, los resultados dependen de cada modelo y configuración de GPUs. Así, podemos destacar:

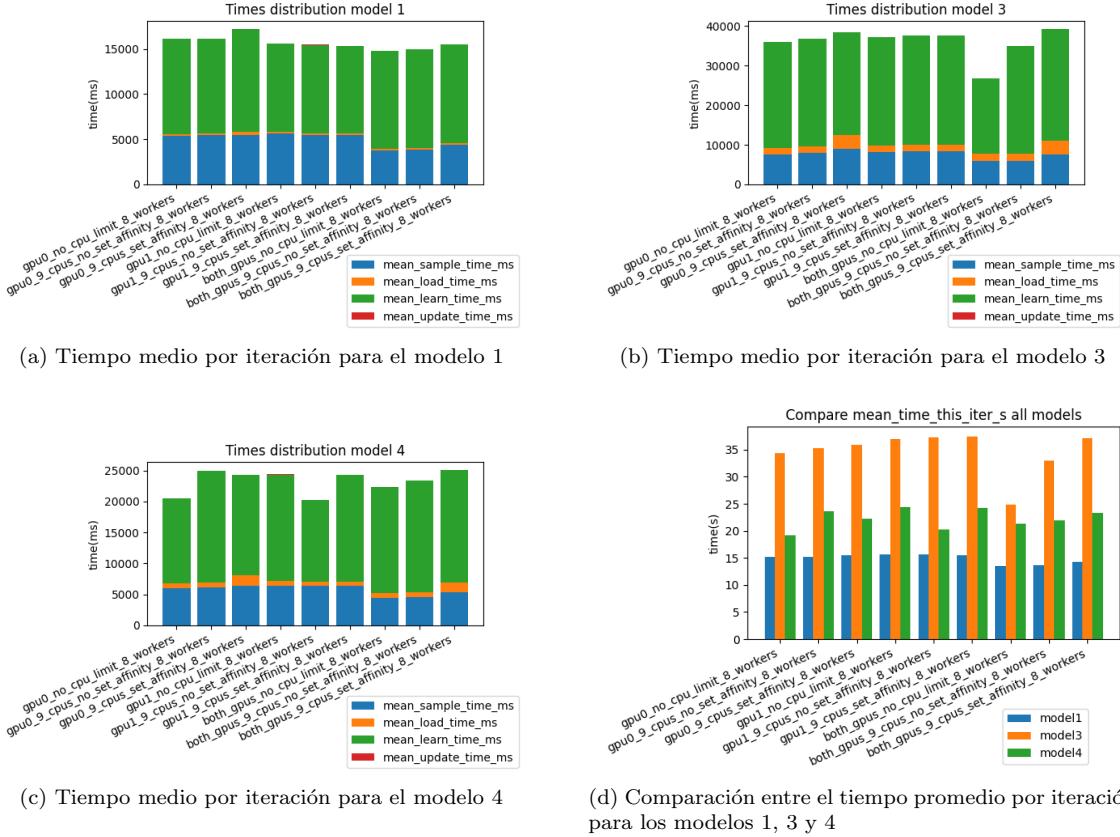


Figura 4.11: Tiempo total promedio por iteración de entrenamiento para distintas configuraciones de uso de las CPUs probadas sobre los modelos 1, 3 y 4, tras 20 iteraciones de entrenamiento en el servidor *volta1*.

1. Para el modelo 1 observamos que las variaciones en los tiempos son prácticamente nulas, si bien en algunos casos se ve un ligero sobrecoste si restringimos el uso de las CPUs a nueve de ellas en concreto.
2. Respecto a los resultados obtenidos para el modelo 4 comprobamos que tanto cuando usamos la GPU RTX como cuando usamos ambas, los resultados son ligeramente mejores cuando no establecemos restricción en el uso de las CPUs. Sin embargo, para la GPU Tesla-v100 esta mejora se observa cuando indicamos a *Ray* que tiene que trabajar sólo con 9 CPUs pero no forzamos a que sean ninguna en concreto.
3. Analizando los resultados para el modelo 3, observamos que en los casos en los que se usa una de las dos GPUs las diferencias son mínimas, si bien hay un suave incremento del tiempo en los experimentos más restrictivos. Si ponemos la mirada en los experimentos con ambas GPUs vemos que hay una tendencia a la alza en el tiempo total empleado según aumentamos las restricciones al uso de las CPUs (más de 10s de diferencia por iteración entre los experimentos `both_gpus_9_cpus_no_cpu_limit_8_workers` y `both_gpus_9_cpus_set_affinity_8_workers`). Esta tendencia se puede observar de manera muy ligera en las tres últimas columnas de la gráfica 4.11a, se acentúa un poco más en 4.11c y la diferencia se hace bastante notable en 4.11b, lo que

nos hace pensar que el tamaño de los datos con los que trabaja el algoritmo quizás sea un factor que influya en esta apreciación, siendo necesaria mayor flexibilidad en el uso de las CPUs si las observaciones que recibimos del entorno son mayores.

En definitiva, podemos concluir que salvo en excepciones, el hecho de restringir el uso de las CPUs a unas en concreto reduce muy ligeramente el porcentaje de uso total de las CPUs del sistema y no supone un sobrecoste elevado en el tiempo de ejecución. Aun así, los mejores resultados de tiempo se obtienen siempre cuando no imponemos restricciones en el uso de las CPUs, pero si por algún motivo necesitamos restringir la ejecución de *Ray* a unas cuantas CPUs el sobrecoste que obtendríamos en la mayoría de los casos sería asumible y no supondría mayor problema.

4.1.4. Conclusiones extraídas de los experimentos de entrenamiento

Poniendo en común las observaciones y conclusiones que hemos ido extrayendo sobre todo el estudio del entrenamiento, podemos afirmar que **no hay una configuración que optimice tanto el uso de recursos como el tiempo empleado por iteración** respecto a las demás.

- La principal consideración a tener en cuenta es realizar el entrenamiento haciendo uso de una o varias GPUs.
- Usar las GPUs sólo para el *driver* y que los *workers* no hagan uso de ellas produce mejores resultados que cuando estas son compartidas tanto por el *driver* y los *workers*. Esto nos incita a pensar que la interacción con el entorno es más rápida cuando no usamos GPU para ella, resultado que más adelante veremos corroborado cuando analicemos la inferencia.
- Respecto al número de *workers* a usar, fijaremos el número en 4, pues es donde se optimiza el tiempo medio por iteración.
- Respecto a la localización de las CPUs, si es posible, no restringimos su uso, pero en caso de tener que forzar la ejecución, tendremos un ligero sobrecoste que en la mayoría de los casos será asumible.

Así, proponemos realizar los entrenamientos con un agente con 4 *workers*, con tantas GPUs como tenga el sistema para el *driver* y ninguna GPU para los *workers*.

4.2. Resultados de la inferencia de modelos

Analizamos aquí los resultados obtenidos en los distintos experimentos realizados para probar la inferencia de modelos previamente entrenados. Dividimos este análisis en dos: por un lado la **inferencia en *RLLib*** haciendo uso de las funcionalidades que nos ofrece su API, y por otro, la inferencia sobre el acelerador **Google Coral**.

4.2.1. Inferencia en *RLLib*

La figura 4.12 muestra las mediciones de tiempo realizadas en los experimentos de inferencia de modelos. Para cada modelo, se ejecutan **10 episodios completos** de inferencia haciendo uso del *script rollout.py* ya descrito en el capítulo anterior. Los resultados que aparecen en la gráfica se obtienen como la media de los tiempos todos los pasos de inferencia ejecutados durante los 10 episodios. Aunque el *script* usado mide el tiempo en segundos, por claridad decidimos representarlo en milisegundos. Durante la ejecución de los experimentos observamos que la creación de *workers* no tiene mucho sentido en este caso, pues el *script* de rollout no paralleliza las ejecuciones y las inferencias se van ejecutando de manera secuencial. A la vista de las gráficas podemos extraer una serie de conclusiones:

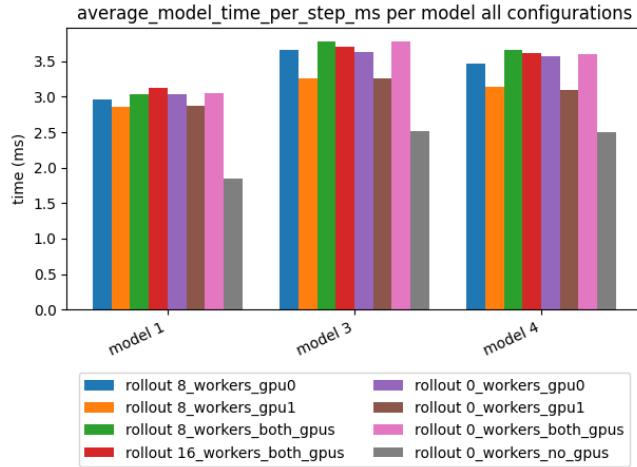


Figura 4.12: Tiempos en milisegundos que se tarda en ejecutar un paso de inferencia para las distintas configuraciones probadas y los modelos 1, 3 y 4.

1. Como ya hemos anticipado, el hecho de crear *workers* no modifica el tiempo de ejecutar las inferencias, pues estas se ejecutan de manera secuencial sobre un único *worker* que se crea en el *driver*. Así, cuando ejecutemos estas inferencias indicaremos en la configuración que el número de *workers* a crear es 0 para que no se creen hilos y procesos innecesarios.
2. Vemos en todos los casos que el uso de la GPU1 reduce el tiempo respecto al uso de ambas GPUs y el uso de únicamente la GPU0. Esto se debe a que durante la ejecución de una serie de episodios de inferencia, siempre que se usa la GPU0, el primer episodio es más lento que el resto e introduce esta ligera penalización. Esta iteración de “calentamiento” no aparece cuando usamos la GPU1. El hecho de realizar varios episodios de inferencia en cada experimento nos ha permitido observar esta peculiaridad, ya que si sólo hubiésemos ejecutado un episodio podríamos haber pensado que este era el tiempo real que se tardaba en ejecutar las inferencias con la GPU0 y la diferencia de tiempos sería mucho mayor. Aún así la pequeña diferencia que se observa en las gráficas nos advierte de este hecho y de la necesidad de descartar los tiempos de la primera ejecución cuando trabajemos con la GPU0.
3. El experimento para el que obtenemos mejores resultados para los tres modelos es aquel en el que la inferencia se realiza sin el uso de GPUs. Esto no es una novedad, pues es algo que ya se había observado analizando los tiempos de interacciones con el entorno (figura 4.4), de donde extraímos que los tiempos eran menores cuando los *workers* (que eran los que realizaban esta interacción) no hacían uso de las GPUs. Y es que en ambos casos lo que estamos haciendo es ejecutar inferencias sobre el modelo, por lo que es lógico que hagamos la misma apreciación. Este hecho podría deberse a que dadas las características y el tamaño de los modelos no sea conveniente desplazar la realización de los cálculos de las inferencias a la GPU.
4. Comparando resultados obtenidos para los distintos modelos, comprobamos que hay una ligera diferencia condicionada por el tamaño de los datos con los que trabaja cada uno de ellos. Así, el tiempo será suavemente mayor si las imágenes que tomamos del entorno lo son.

En conclusión, la mejor configuración para ejecutar inferencias sobre modelos previamente entrenados con *RLlib* implica **no crear *workers*** (no desempeñan ninguna función) y ejecutarlas **sólo sobre las CPUs** del sistema.

4.2.2. Inferencia sobre el acelerador Google Coral

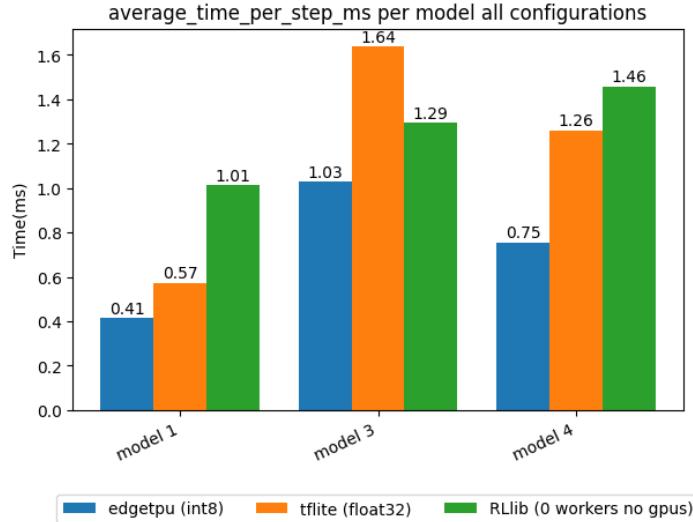


Figura 4.13: Tiempo por cada ejecución de inferencia para distintas configuraciones en el servidor *artecslab001* y los modelos 1, 3 y 4.

Modelo	RLlib	TF Lite	TF Lite Cuantizado	TPU
1	1.01225	0.57362	10.08402	0.41419
3	1.29343	1.63730	70.47366	1.02737
4	1.45621	1.25892	43.15306	0.75465

Cuadro 4.1: Tiempos (en milisegundos) para los distintos experimentos realizados sobre el servidor *artecslab001*.

Analizaremos aquí el resultado de ejecutar inferencias sobre las redes neuronales cuantizadas y compiladas para su correcta ejecución sobre la TPU Google Coral. Además, para poder medir las ventajas del uso de este acelerador, compararemos estos valores con los datos que obtendremos de ejecutar inferencias sobre el modelo de *Tensorflow Lite* sin cuantizar (valores en punto flotante de 32 bits) y de ejecutar inferencias dentro del *framework* de *RLlib* con la configuración para la que mejores resultados obteníamos en el apartado anterior (usando sólo CPUs). Además, probaremos también a ejecutar el modelo cuantizado sobre las CPUs del sistema, aunque este modelo no esté pensado para ser ejecutado sobre una arquitectura de este tipo. La figura 4.13 y la tabla 4.1 muestran los resultados obtenidos para los diferentes experimentos probados. Debido a la gran diferencia numérica que se observa entre los valores resultantes de la inferencia del modelo de *Tensorflow Lite* cuantizado sobre las CPUs y el resto de resultados, estos se omiten en la gráfica. Podemos extraer una serie de conclusiones sobre estos experimentos:

1. En primer lugar, observamos como la inferencia sobre la TPU obtiene los mejores valores de tiempo.
2. Salvo para el modelo 3, el tiempo de inferencia en *RLlib* es mayor que el del modelo de *TFLite*.

3. Los tiempos de inferencia en la TPU también se van a ver condicionados por el tamaño de las entradas que toma el modelo, obteniéndose así un tiempo mayor para el modelo 3, seguido del 4 y del 1.
4. Observamos que la CPU en flotante (modelos de *TFLite* en `float32`) es muchísimo más rápida que en enteros (modelos de *Tensorflow* cuantizados en `int8`), aunque probablemente esto se deba a que la CPU haga uso de sus unidades vectoriales.

Con esto, podemos concluir que hemos conseguido uno de los objetivos fundamentales del trabajo: evaluar el rendimiento de modelos de aprendizaje por refuerzo sobre la TPU Coral y compararlo con la ejecución de esos modelos sobre la CPU y dentro del *framework* de *RLlib*. Además, esta comprobación ha sido bastante satisfactoria, pues hemos conseguido ejecutar un modelo entrenado en *RLlib* sobre un **dispositivo de ultrabajo consumo y precio** como es la TPU Google Coral donde además **los tiempos de inferencia son bastante menores**, liberando además el resto de recursos del sistema durante este proceso.

El hecho de ejecutar el modelo cuantizado conlleva una **pequeña pérdida de precisión** en la representación de los valores obtenidos como salida, como ya anticipábamos en el primer capítulo de este trabajo. Mostramos a continuación un ejemplo de salida obtenida tras aplicar el mismo modelo y para los mismos datos de entrada en la CPU con valores en punto flotante de 32 bits y en la TPU con enteros de 8 bits (los valores de la TPU que mostramos son los resultantes de “descuantizar” los que realmente nos devuelve el modelo, haciendo uso de los valores de *zero point* y *scale* con los que se ha realizado la cuantización).

```

1 - En TPU con uint8:
2
3 ---- output[0] ----
4 INT8 DATA
5 [[[ 7.150586   -4.8753996  12.1884985  10.319595   -0.08125666 13.894889  ]]]
6 ---- output[1] ----
7 INT8 DATA
8 [[-0.70262796]]
9
10 - En CPU con float32:
11
12 ---- output[0] ----
13 FLOAT DATA
14 [[[ 7.969496   -5.3353543  13.295782   11.171885   -0.17144847  15.070765  ]]]
15 ---- output[1] ----
16 FLOAT DATA
17 [[-0.6038263]]
```

Vemos que pese a las ligeras diferencias que se observan, los valores están bastante próximos entre sí, y lo que es más importante, en ambos casos el orden los mismos en la salida 0 se mantiene, por lo que la acción a tomar, que viene determinada por el índice del mayor valor en esta salida, es la misma. Destacar finalmente que la primera de las salidas (`output[0]`) se corresponde con la salida de la red de política (*policy network*) e indica la probabilidad de que la acción en cada posición sea la que a la larga maximice la recompensa del episodio (estos valores podemos llevarlos al intervalo $[0, 1]$ para que realmente representen el valor de una probabilidad, mediante la función *softmax*, pero el orden de los mismos se sigue manteniendo). La segunda de las salidas (`output[1]`) se corresponde con el resultado de la red de valor (*value network*) e indica la recompensa esperada para la secuencia completa de acciones hasta concluir el episodio, que se usa únicamente en el entrenamiento del modelo para actualizar los parámetros de la red neuronal.

Capítulo 5

Conclusiones

Con la realización de este trabajo de fin de grado se ha completado un **estudio exhaustivo del rendimiento de las aplicaciones de aprendizaje por refuerzo en diferentes arquitecturas hardware**, analizando tanto el coste en tiempo como la utilización de recursos y el consumo de potencia. Los resultados obtenidos serán bastante útiles a la hora de diseñar los procesos de inferencia y entrenamiento para resolver problemas de aprendizaje por refuerzo, ya que basándonos en ellos podemos configurar estos procesos para acelerar su ejecución o reducir la utilización de recursos. Además, hemos integrado en el trabajo una serie de bibliotecas que ofrecen recursos específicos para la paralelización de los algoritmos o el modelado de los entornos de aprendizaje, proporcionando *scripts* que permiten llevar a cabo modelizaciones, entrenamientos e inferencias dentro del marco del aprendizaje por refuerzo y que se pueden ejecutar en diferentes arquitecturas hardware.

Analizando la **lista de objetivos** que proponíamos en la introducción de esta memoria observamos que se ha cumplido en buena medida con todos ellos:

1. Se ha conseguido **modelar el escenario de aprendizaje por refuerzo**, integrando un entorno *Gym* dentro de la funcionalidad de *RLlib*.
2. Se han propuesto varios **experimentos de entrenamiento**. Para ello se han desarrollado varios *scripts* que los ejecutaban haciendo uso de *RLlib*. Esta parte ha involucrado además un análisis profundo de la librería para tratar de explotar al máximo su funcionalidad, siendo necesario muchas veces para ello una lectura exhaustiva del código fuente de la misma y la interacción con otros usuarios a través del foro oficial de *Ray*¹ para tratar de aclarar algunas cuestiones.
3. Se ha **evaluado el entrenamiento** para modelos que interaccionaban con el entorno *Pong-v0* de *Gym*, pero que diferían entre sí en el tamaño de las imágenes que recibían de este entorno. Además, se propusieron varias opciones para cada uno de los valores del tamaño de entrada que queríamos evaluar, estableciéndose un criterio para elegir los representantes por cada tamaño. Quizás, quede como futuro trabajo evaluar también este rendimiento en diferentes entornos (en los que los valores de las recompensas y el rango de acciones sea distinto).
4. Una vez realizados los experimentos propuestos y obtenida información acerca de los mismos (la cual se encuentra también el repositorio *Github* de este trabajo), se ha conseguido organizar, estructurar y representar gráficamente esta información, **analizando los datos y extrayendo conclusiones** de los mismos.

¹<https://discuss.ray.io/categories>

5. Se han realizado **inferencias usando las utilidades de *RLLib***, siendo necesario para ello adaptar los *scripts* que nos proporcionaba la librería a nuestros objetivos específicos.
6. Se ha llevado a cabo un proceso en varias etapas que nos han permitido **ejecutar inferencias** sobre modelos entrenados en *RLLib* sobre el **acelerador de Google Coral**, pasando por varios modelos de *Tensorflow* y *Tensorflow Lite* intermedios.
7. También se ha recabado información acerca de estos experimentos, que ha permitido **compararlos con los realizados en *RLLib***, y también con otros experimentos realizados directamente sobre la CPU del sistema con los modelos intermedios obtenidos durante este proceso de transformación.

De los **resultados obtenidos** podemos destacar que se ha comprobado como el proceso de entrenamiento con el algoritmo PPO se acelera enormemente si su ejecución se ayuda de GPUs, que la inferencia sobre los modelos propuestos es más eficiente si se realiza sólo sobre CPUs y que la TPU ofrece buenos resultados para la inferencia (la pérdida de precisión por la cuantización de los datos es pequeña) y mejora el rendimiento respecto al uso de procesadores más generales, añadiéndose a esto la ventaja de su bajo consumo de potencia.

En cuanto al **trabajo futuro**, con la realización de este TFG quedan abiertas varias líneas en las que continuar con la investigación. Por un lado, se puede evaluar el rendimiento variando aun más los recursos hardware, ya sea aumentando el número de CPUs y GPUs de las que podemos disponer o **introduciendo otros aceleradores de propósito específico** para la inferencia de redes neuronales (y quizás incluso para el entrenamiento) diferentes a la TPU de Google Coral. También, podría haberse realizado un **entrenamiento de los modelos más exhaustivo**, aumentando el número de iteraciones y viendo si podemos maximizar las recompensas obtenidas. De hecho, aquí entra otra línea de trabajo que se separa un poco del análisis del hardware, y que se centraría en evaluar la **calidad del aprendizaje** variando diversos parámetros del algoritmo y de los modelos. En este trabajo nos hemos restringido a evaluar un escenario de aprendizaje por refuerzo en el que tomábamos *Pong-v0* de *Gym* como entorno y entrenábamos siguiendo el algoritmo PPO. Así, sería también interesante repetir todo el proceso aquí mostrado para **otros entornos** (incluso ya saliendo de los que simulan videojuegos y probando, por ejemplo, problemas de simulación de robots) y ver como esto influye en el uso de los recursos, pues los modelos empleados serán distintos. Además, se podría evaluar el rendimiento de **otros algoritmos** distintos del PPO y compararlos entre sí y con este último, analizando también el uso de los recursos y los tiempos empleados y la conveniencia de los distintos algoritmos para problemas diferentes.

Por último, destacar que con la realización de este trabajo se han puesto en práctica muchos de los **conocimientos y habilidades** adquiridas a lo largo del Grado en Ingeniería Informática y también del Grado en Matemáticas. Así, se han aplicado los contenidos de **asignaturas relacionadas con el hardware** para entender y poder analizar el rendimiento en las diferentes arquitecturas con las que se ha trabajado. Se han aplicado conceptos propios del **desarrollo software** en el diseño y programación de los *scripts* de *Python* necesarios para la realización del trabajo. También, se ha profundizado más en los conceptos vistos a lo largo de la carrera sobre **aprendizaje automático**, introduciendo el paradigma de aprendizaje por refuerzo que difería de los de aprendizaje supervisado y no supervisado vistos a lo largo del grado. Aquí también se aplican competencias más bien adquiridas en las asignaturas del Grado en Matemáticas para definir formalmente la idea de aprendizaje por refuerzo, de política y el algoritmo PPO, donde intervienen conceptos también de problemas generales de optimización. Por último, destacar la aplicación de técnicas para el **análisis y la visualización de datos**, que nos han permitido transformar los datos que obteníamos en información.

Fuera de los conocimientos puramente técnicos, para la realización de este trabajo han sido necesarias otra serie de habilidades, como la **organización y gestión del tiempo** disponible para el mismo. Además, ha sido una buena primera toma de contacto con el **mando de la investigación**, observando las dificultades que entraña desarrollar o probar algo novedoso sobre lo que no hay demasiada información disponible, y la satisfacción que supone poder cumplir los objetivos propuestos.

Conclusions

With this Bachelor's thesis we have fulfilled a **comprehensive study about the performance of reinforcement learning applications in different hardware architectures**, analyzing the time cost as well as the available resources utilization and the power consumption. The results that we have obtained can be quite useful when designing inference and training processes to solve reinforcement learning problems, so we can tune the processes to achieve a time speedup or to save energy using these results. Even more, we have included in this thesis a series of libraries that offer functionalities to parallelize the algorithms involved in the processes or to accurately model learning environments, providing scripts that allow us to design our reinforcement learning modeling, training and inference processes and that can be run in different hardware architectures.

If we analyze the **objectives list** introduced in the first chapter of this report, we can check that we have almost met all of them:

1. We have managed to **model reinforcement learning scenarios**, using *Gym* environments together with *RLLib* functionalities.
2. We have proposed several **training experiments**. To do that, we have developed a series of *scripts* that execute them using *RLLib*. In addition, this part of the thesis has involved a deep and comprehensive analysis of the library in order to take full advantage of its functionality, many times requiring a comprehensive reading of its source code and the discussion with other users through the official *Ray* forum² in order to clarify some doubts.
3. We have **evaluated the training process** with models that interacted with *Pong-v0* from *Gym*, but varying the size of the inputs that they received from the environment. Furthermore, we proposed several options for each of the values to consider for the input size, setting a criterion to choose the representatives of each size. Maybe, we can leave as a future job the performance evaluation in different environments (where the rewards values and the actions range are different).
4. Once we had run the experiments and we had collected the information related to them (we have this information available in the GitHub repository of this thesis), we managed to organize, structure and graphically represent this data, **analyzing and drawing conclusions** about them.
5. We have **run inferences using *RLLib* utilities**, requiring a modification of the library scripts so that they could be used for our specific purpose.
6. We have carried out a process in several steps that has allowed us to **run inferences on the TPU accelerator** using models trained with *RLLib*, generating some auxiliary *Tensorflow* and *Tensorflow Lite* models.

²<https://discuss.ray.io/categories>

7. We have also collected information about these experiments, **comparing its results with the ones that we obtained for those inferences run with *RLlib*** and also with other experiments run on the system CPUs using the auxiliary models previously mentioned.

Regarding to **the results obtained**, we can highlight that we have checked how PPO algorithms speeds up if we use GPUs for its execution, we concluded that inference process is more efficient if we run it using only CPUs or we realized how TPU reports good results for inference (precision loss due to quantization is small) and how it improves performance compared to more general processors, adding to that the advantage of the power saving.

About **future work**, we can say that with this Bachelor's thesis we have left several ways open to continue with the investigation. On the one hand, we can study the performance varying even more the hardware architectures, increasing the number of CPUs or GPUs considered or **using other specific-purpose accelerators** different from Google Coral TPU. Furthermore, we can **train more exhaustively** the models, increasing the number of iterations and seeing how can we optimize the rewards. In fact, we can propose here another area where we can focus the investigation, that leaves out the hardware analysis and that is focused on **evaluating the learning quality** of the models, customizing several parameters of the model and algorithm. In this thesis we have only used the PPO algorithm for the training process and the *Gym Pong-v0* environment, but maybe it's interesting to consider again the whole process with **other environments** (even considering not only the ones that simulate video games but also those that face robot simulation problems) and to see how this modifications influence the resource usage, as the models used are different. In addition, we can evaluate the performance of **other algorithms** different from PPO and compare all of them and also with PPO, analyzing the resources usage, the times needed and how suitable are these algorithms for each specific problem.

To end, we want to highlight that while elaborating this thesis I have put into practice many of the **skills and knowledge** that I have learned during the Computer Science Bachelor's Degree and also the Mathematics one. I have used contents from **subjects related to hardware** to understand and being able to analyze the performance in different architectures we have worked with. I have put into practice **software development** ideas when designing and programming the scripts required for the thesis. Furthermore, I have gone deeper into the concepts about **machine learning**, studying the reinforcement learning area that differs from the supervised and unsupervised paradigms taught during the degree. At this point, I have also used some skills acquired when studying subjects of the Math degree, being able to formally define the ideas of reinforcement learning, policy or the PPO algorithm, applying concepts related to general optimization problems. Lastly, I have also applied **data visualization and analysis** techniques that have allowed us to turn the data we had into structured information.

Apart from the purely technical knowledge, the elaboration of this thesis has also required some other skills, such as the **management and organization of the available time** to achieve the objectives of the thesis. Moreover, this thesis is a well first contact with the **world of investigation**, realizing the difficulties that we face when we try to develop something quite innovative and with quite little documentation about it, but also the pleasure that produces achieving your goals.

Apéndice A

Funcionamiento de los scripts de Python

Mostramos en este apéndice los detalles de implementación y la manera de usar los distintos principales *scripts* que se incluyen en el repositorio de *Github*¹ que complementa a este trabajo de fin de grado. El código ha sido probado y ejecutado con éxito en un entorno *Python 3.7.3* con las siguientes librerías instaladas:

- Ray 1.1.0
- Tensorflow 2.4.1
- Gym 0.18.0

Además, para obtener métricas relativas al uso de las GPUs ha sido necesario instalar la biblioteca *gputil* y para obtener la representación gráfica y en ficheros de los datos se emplean las bibliotecas *matplotlib* y *pandas*.

A.1. Script de entrenamiento

Proporcionamos un *script* `train_ppo.py` con el que se pueden realizar todos los experimentos de entrenamiento descritos en las tablas 3.2 y 3.3 para cualesquiera de los modelos 1 al 6 descritos en la tabla 3.1. El *script* contiene, aparte de la función `main` tres funciones auxiliares:

- `gpu_options(gpu_opt)`: establece las GPUs que se mostrarán visibles al proceso y por tanto podrán ser utilizadas para el entrenamiento. Recibe en `gpu_opt` un *string* indicando la configuración deseada: ‘`gpu0`’ para usar únicamente la GPU con identificador 0 (RTX en *volta1*), ‘`gpu1`’ para usar únicamente la GPU con identificador 1 (Tesla-v100 en *volta1*), ‘`none`’ para no usar ninguna de ellas y ‘`both`’ para usar las dos. Para ello, se establece el valor de la variable de entorno del sistema `CUDA_VISIBLE_DEVICES` con los identificadores de las GPUs que queremos que sean visibles en cada caso.
- `get_config(model)`: Devuelve un diccionario con la configuración de un agente para cada uno de los seis modelos propuestos en la tabla 3.1, que se especifican con un entero con valores entre 1 y 6 mediante el parámetro `model`.

¹<https://github.com/javigm98/Mejorando-el-Aprendizaje-Automatico>

- **full_train(checkpoint_root, agent, n_iter, save_file, n_ini = 0, header = True, restore = False, restore_dir = None)**: ejecuta una serie de iteraciones de entrenamiento sobre un agente dado y devuelve una estructura con sus resultados, además de guardar esta información en unos ficheros .csv y .json. Recibe como argumentos:
 - **checkpoint_root**: string con la ruta del directorio en el que queremos que se vayan guardando los *checkpoints* para cada paso de entrenamiento realizado.
 - **agent**: agente de *RLlib* sobre el que ejecutar las iteraciones de entrenamiento.
 - **n_iter**: entero indicando el número de iteraciones de entrenamiento del algoritmo concreto del agente (en nuestro caso PPO) a ejecutar.
 - **save_file**: ruta al archivo en el que queremos que se almacenen la información del entrenamiento. Mediante un *string* indicamos la ruta a un archivo sin extensión, así se crearán dos archivos en esa ruta con extensiones .json y .csv.
 - **n_ini**: entero indicando el número de la última iteración realizada, su valor por defecto es 0, indicando que aun no hemos comenzado a entrenar ese modelo.
 - **header**: booleano indicando si hay que añadir la línea de cabecera con los nombres de las columnas al fichero .csv con los datos del entrenamiento. Su valor por defecto es **True** indicando que si es la primera vez que estamos entrenando el modelo sí hay que añadir esta línea.
 - **restore**: booleano indicando si debemos establecer o no el estado del agente desde un *checkpoint*, cuya ruta indicamos en **restore_dir**. Su valor por defecto es **False**.
 - **restore_dir**: ruta del *checkpoint* desde el que queremos restaurar el estado del agente, si hemos indicado **restore=True**.

La función devuelve una lista con un diccionario por cada iteración de entrenamiento, en el que se incluyen el número de iteración, las recompensas mínima, media y máxima de los episodios, la longitud media de los episodios, el tiempo de la fase de aprendizaje en ms Y el tiempo total en segundos de esa iteración. Estos mismos datos se guardan en los ficheros .json y .csv antes mencionados.

Así, para ejecutar uno de los experimentos de entrenamiento ejecutamos el *script* indicando pudiendo indicarle el valor de varios argumentos:

- **-m, --model**: entero (1-6) indicando el identificador del modelo a entrenar.
- **-g, --gpu**: string con los valores **gpu0**, **gpu1**, **none**, **both** indicando la configuración de GPUs con las que realizar el entrenamiento.
- **-d, --driver-gpus**: número de GPUs que se asignarán al driver (`config[num_gpus]`), puede ser un número decimal. El resto se repartirán a partes iguales entre los *workers*.
- **-w, --workers**: número de *workers* que se crearán en el algoritmo para recoger experiencias del entorno.
- **-s, --save-name**: ruta del fichero, sin extensión, en el que se guardarán los datos de entrenamiento en formatos .json y .csv.
- **-i, --iters**: número de iteraciones de entrenamiento a ejecutar.
- **-c, --cpus**: número de CPUs que indicamos a *Ray* en su inicialización. Su valor por defecto es **None**, que indica que *Ray* usará todas las que encuentre disponibles.

- **-a, --set-affinity:** conjunto con los identificadores de las CPUs a las que queremos restringir la ejecución con `sched.setaffinity`. Su valor por defecto es el conjunto vacío (`{}`), que indica que no forzamos a que el programa se ejecute en unas CPUs concretas.
- **-r, --restore_dir:** dirección del *checkpoint* desde el que queremos restaurar el estado del agente. Su valor por defecto es `None` que indica que no queremos restaurar desde ningún *checkpoint*.

Además de realizar las iteraciones de entrenamiento indicadas, la ejecución de este *script* mueve los ficheros con las métricas que reporta Ray (y que por defecto se guardan en un directorio dentro de `~/ray_results` cuyo nombre viene dado por el *timestamp* del momento en que se inicia la ejecución) y los almacena en un directorio dentro de la carpeta `ray_results` del proyecto y con el nombre indicado por `save_name`. Además, también copia el fichero `params.pkl` de este directorio en el que se guardan los *checkpoints*, pues luego será necesario que este ahí para la ejecución de inferencias.

Por ejemplo, podemos ejecutar 1000 iteraciones de entrenamiento para el modelo 3 usando sólo la GPU 0 del sistema, con 0.001 GPUs para el *driver* y 4 *workers* que se reparten el resto de la GPU con la siguiente instrucción:

```
1 $ python training_scripts/train_ppo.py --model=3 --gpu=gpu0 --driver-gpus=0.001 \
2   --workers=4 --save-name=model3_4_workers_gpu0 --iters=1000
```

Esto generará un directorio para cada *checkpoint* en `checkpoints/ppo/model3_4_workers_gpu0` y unos ficheros `training_results/ppo/model3_4_workers_gpu0.csv` y el mismo pero con extensión `.json` con algunos datos del entrenamiento. Además, tendremos en `ray_results/model3_4_workers_gpu0` los ficheros con las métricas que genera *Ray*.

A.2. Script de inferencia en RLlib

El *script* `rollout_with_time.py`² será el que utilizaremos para realizar los experimentos de inferencia en *RLlib*. Este *script* es una modificación del que proporciona ya *RLlib* (`rollout.py`³, al que se le añade el código necesario para medir y guardar datos sobre el tiempo que se toma en cada inferencia y para la gestión de los recursos disponibles. Así, podemos especificar una serie de parámetros cuando ejecutemos este *script*, algunos de los cuales proviene del *script* original de *RLlib*:

- **checkpoint:** primer argumento, con él indicamos la ruta al *checkpoint* desde el que queremos restablecer el estado del agente para las inferencias.
- **--run:** algoritmo con el que hemos entrenado al agente. En nuestro caso siempre tomará el valor `PPO`.
- **--env:** entorno *Gym* sobre el que ejecutar las inferencias. En nuestro caso tomará el valor `Pong-v0`.
- **--time-output:** ruta a un fichero `.csv` en el que se guardarán los datos de tiempo de las inferencias.
- **--no-render:** es necesario añadir este argumento si no queremos que se muestre por pantalla las interacciones con el entorno. Nosotros siempre lo añadiremos.
- **--gpu:** configuración de GPUs con las que realizar la inferencia. Puede tomar los valores `gpu0`, `gpu1`, `none` y `both`.

²https://github.com/javigm98/Mejorando-el-Aprendizaje-Automático/blob/main/rollout_with_time.py

³<https://github.com/ray-project/ray/blob/master/rllib/rollout.py>

- **--video-dir**: directorio en el que guardaremos videos de las interacciones. No lo utilizamos en este trabajo.
- **--seteps**: número de pasos de inferencia a ejecutar. Si especificamos un número de episodios (con **--episodes**) el valor que le hayamos dado al número de pasos quedará sin efecto.
- **--episodes**: número de episodios completos a ejecutar. **--config**: diccionario con la configuración del agente, que sobreescribe a la cargada del fichero `params.pkl` del directorio del *checkpoint*.
- **--save-info**: guarda información sobre las observaciones y las acciones de cada paso de inferencia. No lo utilizaremos.
- **--use-shelve**: guarda la información sobre las observaciones y las acciones de cada paso de inferencia con formato *shelf*.
- **--set-affinity**: Conjunto (*set*) con los identificadores de las CPUs a las que queremos restringir la ejecución.
- **--num-cpus-ray**: número de CPUs que indicamos a *Ray* en su inicialización. Si su valor es 0 (lo es por defecto), le estamos indicando a *Ray* que puede usar todas las que encuentre disponibles.

La configuración de recursos específica (número de *workers*, GPUs para el *driver...*) podemos especificarla en el parámetro **--config**. Un ejemplo de ejecución de inferencia sin GPUs y sin crear *workers* sería:

```
1 $ python rollout_with_time.py checkpoints/ppo/model1_gpu/checkpoint_11000/checkpoint
   -11000 --run=PPO --env=Pong-v0 --time-output=rollout_results/volta1/
   model1_no_gpus_0_workers.csv --no-render --gpu=none --episodes=10 --config='{
   num_workers":0, "num_gpus_per_worker":0, "num_gpus":0}
```

A.3. Scripts de exportación y cuantización de modelos para la TPU

Detallaremos ahora el contenido y manera de uso de los cuatro *scripts* que llevan a cabo el proceso completo de creación de modelos de *Tensorflow Lite* cuantizados que pueden ser ejecutados en la TPU.

A.3.1. Script de exportación de modelos

El *script* `model_saver.py`⁴ parte de un modelo entrenado en *RLlib* y exporta la red neuronal con la que se modela la política y su valor en formato `.h5`. Para ello, la ejecución del *script* requiere dos parámetros en su llamada:

- Dirección a un *checkpoint* desde el que restableceremos el estado del agente a exportar.
- Ruta donde queremos guardar el modelo en formato `.h5`. Se indicará la ruta al fichero y su nombre sin extensión.

El *script* creará un agente PPO restaurando el estado del *checkpoint* pasado como primer argumento y guardará el modelo de *keras* que contiene la red neuronal de la política y su valor en un fichero con extensión `.h5` en la dirección especificada como segundo argumento. Por ejemplo, podemos obtener un fichero `.h5` del modelo 1 ejecutando:

```
1 $ python model_saver.py checkpoints/ppo/model1_gpu/checkpoint_1000/checkpoint-1000
   exported_models/model1
```

⁴https://github.com/javigm98/Mejorando-el-Aprendizaje-Automático/blob/main/model_saver.py

A.3.2. Script de creación de modelos de Tensorflow Lite

El *script* `tflite_converter.py`⁵ crea y guarda un modelo de *Tensorflow Lite* a partir de un modelo de *keras* previamente exportado en formato `.h5`. En su ejecución debemos indicarle el valor de dos argumentos:

- Dirección del archivo con extensión `.h5` donde se encuentra el modelo de *keras* exportado.
- Dirección del fichero `.tflite` con extensión donde queremos guardar el modelo resultante.

El *script* creará un objeto `TFLiteConverter` que llevará a cabo la conversión a partir del modelo de *keras* previamente cargado. Por ejemplo, para crear un modelo de *Tensorflow Lite* del modelo 1 podemos ejecutar:

```
1 $ python tflite_converter.py exported_models/model1.h5 exported_models/model1.tflite
```

A.3.3. Script de creación de datasets para la cuantización

El *script* `dataset_creator.py`⁶ crea y guarda conjuntos de imágenes del entorno con el que interaccionan los modelos y que toman como entradas y que son necesarias para que durante el proceso de cuantización se puedan estimar los rangos que toman los tensores de entrada y de salida del modelo (pues sus valores son variables) y el modelo cuantizado pierda la menor precisión posible respecto al original. Debemos especificar el valor de dos argumentos en la ejecución del *script*:

- Dimensión de las imágenes que guardaremos en el *dataset*.
- Ruta en la que se guardará el *dataset* que se cree, sin extensión.

Una vez ejecutemos el *script*, se creará un entorno como con el que interaccionan los agentes y se tomarán 500 imágenes obtenidas como observaciones tras ejecutar una serie de acciones aleatorias sobre este entorno. Estas imágenes se guardarán en un fichero con extensión `.npy` (pues son en realidad *arrays* de *Numpy*) en la ruta indicada como segundo argumento. Por ejemplo, podemos crear un *dataset* con imágenes de dimensión $(168 \times 168 \times 4)$, que podrían ser usado para la cuantización del modelo 4, ejecutando:

```
1 $ python dataset_creator.py 168 datasets/dataset_model4
```

A.3.4. Script de cuantización de modelos de Tensorflow Lite

El *script* `quantizer.py`⁷ lleva a cabo la creación de un modelo de *Tensorflow Lite* cuantizado, con todos su parámetros como enteros de 8 bits, a partir de un modelo de *keras* exportado en un fichero `.h5`. Para ello requerirá tres argumentos cuando lo ejecutemos:

- Dirección a un dataset, con extensión `.npy` que contenga al menos 500 imágenes que podrían ser entrada del modelo que queremos convertir.
- Dirección del modelo de *keras* con extensión `.h5` que queremos convertir a *Tensorflow Lite* y cuantizar.
- Dirección del fichero con extensión `.tflite` donde queremos guardar el modelo convertido a *Tensorflow Lite* y cuantizado.

⁵https://github.com/javigm98/Mejorando-el-Aprendizaje-Automático/blob/main/tflite_converter.py

⁶https://github.com/javigm98/Mejorando-el-Aprendizaje-Automático/blob/main/dataset_creator.py

⁷<https://github.com/javigm98/Mejorando-el-Aprendizaje-Automático/blob/main/quantizer.py>

El *script* contiene la función `representative_data_gen()` que toma 100 imágenes del *dataset* cargado de la ruta especificada como primer parámetro para poder estimar el rango de las entradas y las salidas del modelo y que la cuantización de estos valores sea correcta. Así, se carga el modelo de *keras* guardado en la dirección del segundo argumento y se convierte a *Tensorflow Lite* cuantizando los valores de sus parámetros, guardando el modelo resultante en el fichero especificado como tercer argumento. Por ejemplo, podemos crear una versión cuantizada del modelo 3 ejecutando:

```
1 $ python quantizer.py datasets/dataset_model3.py exported_models/model3.h5
   exported_models/model3_quant.tflite
```

A.4. Scripts de inferencia de modelos de Tensorflow Lite

Detallaremos aquí como se implementan y el modo de uso de los *scripts* `rollout_coral.py`⁸ y `rollout_tflite.py`⁹ que ejecutan inferencias sobre modelos de *Tensorflow Lite*, bien cuantizados o sin cuantizar sobre el acelerador Google Coral (`rollout_coral.py`) o sobre las CPUs del sistema (`rollout_tflite.py`). La estructura de estos dos *scripts* es la misma, salvo que el primero de ellos al crear el intérprete del modelo de *Tensorflow Lite* establece como delegado la TPU. Además, de la función principal de los *scripts*, estos cuenta con dos funciones auxiliares:

- `make_interpreter(model_file)`: Recibe como parámetro la ruta a un modelo guardado de *Tensorflow Lite* y devuelve un objeto de la clase `Interpreter` sobre el que podremos ejecutar inferencias. En el caso del *script* para la TPU, aquí se indica mediante un delegado que las ejecuciones se realizarán en este soporte.
- `keep_gping(steps, num_steps, episodes, num_episodes)`: Función que implementa la condición del bucle, indicando cuando debemos parar de ejecutar pasos de inferencia. Se toma directamente del *script* de inferencia que nos proporciona *RLlib* (`rollout.py`).

Cuando ejecutemos el *script* podemos dar valor a una serie de parámetros que configuran las inferencias a realizar:

- `-m, --model`: ruta al archivo `.tflite` en el que se encuentra el modelo de *Tensorflow Lite* (cuantizado o no) sobre el que ejecutaremos las inferencias.
- `-s, --steps`: pasos de inferencia que queremos ejecutar. Si damos valor a `--episodes` el número de pasos indicado no tendrá efecto.
- `-e, --episodes`: número de episodios completos de inferencia a ejecutar. Si indicamos su valor, el de `--steps` queda sin efecto.
- `-o, --output`: ruta a un archivo `.csv` en el que guardaremos los datos relativos a la ejecución de las inferencias (tiempos, pasos por episodio, recompensas...).

Cuando ejecutamos cualesquiera de los dos *scripts* en primer lugar se crea el intérprete para el modelo de *Tensorflow Lite* indicado. Seguidamente se crea un entorno con `wrap_deepmind` con *Pong-v0* como base, y de aquí será de donde se toman las iteraciones. Ahora, se itera mientras no hayamos completado el número total de episodios (o mientras no hayamos completado el número total de pasos en caso de no haber indicado un número de episodios a ejecutar) y en cada paso de iteración se toma una imagen

⁸https://github.com/javigm98/Mejorando-el-Aprendizaje-Automático/blob/main/exported_models/rollout_coral.py

⁹https://github.com/javigm98/Mejorando-el-Aprendizaje-Automático/blob/main/exported_models/rollout_tflite.py

del entorno, se coloca como tensor de entrada del intérprete del modelo, se invoca al modelo y se obtiene el valor del tensor de salida. De la salida de la política, se toma el índice con el valor más alto y esa será la siguiente acción, que se realiza sobre el entorno, obteniéndose así una nueva observación y comenzando nuevamente el proceso (básicamente es la misma idea que se sigue en el *script rollout.py* de *RLLib*).

Bibliografía

- [1] Imad Dabbura. «Gradient Descent Algorithm and Its Variants». En: *Towards Data Science* (2017). URL: <https://towardsdatascience.com/gradient-descent-algorithm-and-its-variants-10f652806a3>.
- [2] Vincent Dumoulin y Francesco Visin. «A guide to convolution arithmetic for deep learning». En: *arXiv* (2016). URL: <https://arxiv.org/pdf/1603.07285v1.pdf>.
- [3] Jonathan Hui. «RL — Proximal Policy Optimization (PPO) Explained». En: *Medium* (2018). URL: <https://jonathan-hui.medium.com/rl-proximal-policy-optimization-ppo-explained-77f014ec3f12>.
- [4] Renu Khandelwal. «A Basic Introduction to TensorFlow Lite». En: *Towards Data Science* (2020). URL: <https://towardsdatascience.com/a-basic-introduction-to-tensorflow-lite-59e480c57292>.
- [5] Ue Kiao. «Calculate output size of Convolution». En: *OpenGenus IQ* (2021). URL: <https://iq.opengenus.org/output-size-of-convolution/>.
- [6] Mehryar Mohri, Afshin Rostamizadeh y Ameet Talwalkar. *Foundations of Machine Learning (second edition)*. MIT Press, 2018.
- [7] Paco Nathan. «Distributed Computing with Ray: Intro to RLlib: Example Environments». En: *Medium* (2020). URL: <https://medium.com/distributed-computing-with-ray/intro-to-rllib-example-environments-3a113f532c70>.
- [8] Sumit Saha. «A comprehensive Guide to Convolutional Neural Networks». En: *Towards Data Science* (2018). URL: <https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53>.
- [9] Manas Sahni. «8-Bit Quantization and TensorFlow Lite: Speeding up mobile inference with low precision». En: *HeartBeat Fritz AI* (2018). URL: <https://heartbeat.fritz.ai/8-bit-quantization-and-tensorflow-lite-speeding-up-mobile-inference-with-low-precision-a882dfcafbbd>.
- [10] Sabyasachi Sahoo. «Deciding optimal kernel size for CNN». En: *Towards Data Science* (2018). URL: <https://towardsdatascience.com/deciding-optimal-filter-size-for-cnns-d6f7b56f9363>.
- [11] Kaz Sato. «What makes TPUs fine-tuned for deep learning?». En: *Google Cloud Blogs* (2018). URL: <https://cloud.google.com/blog/products/ai-machine-learning/what-makes-tpus-fine-tuned-for-deep-learning>.
- [12] John Schulman y col. «Proximal Policy Optimization Algorithms». En: *arXiv* (2017). URL: <https://arxiv.org/pdf/1707.06347.pdf>.
- [13] John Schulman y col. «Trust Region Policy Optimization». En: *arXiv* (2017). URL: <https://arxiv.org/pdf/1502.05477.pdf>.

- [14] Sagar Sharma. «Policy Networks vs Value Networks in Reinforcement Learning». En: *Towards Data Science* (2018). URL: <https://towardsdatascience.com/policy-networks-vs-value-networks-in-reinforcement-learning-da2776056ad2>.
- [15] Abhishek Suran. «Proximal Policy Optimization (PPO) With TensorFlow 2.x». En: *Towards Data Science* (2020). URL: <https://towardsdatascience.com/proximal-policy-optimization-ppo-with-tensorflow-2-x-89c9430ecc26>.
- [16] Richard S. Sutton y Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 2014.
- [17] Jordi Torres. «Deep Q-Network (DQN)-I OpenAI Gym Pong and Wrappers». En: *Towards Data Science* (2020). URL: <https://towardsdatascience.com/deep-q-network-dqn-i-bce08bdf2af>.
- [18] Serdar Yegulalp. «What is TensorFlow? The machine learning library explained». En: *InfoWorld* (2019). URL: <https://www.infoworld.com/article/3278008/what-is-tensorflow-the-machine-learning-library-explained.html>.