APRENDIZAJE POR REFUERZO PARA LA MEJORA DE CALIDAD DE SERVICIO EN PROCESOS DE CODIFICACIÓN DE VÍDEO

IMPROVING QUALITY OF SERVICE FOR VIDEO CODING USING REINFORCEMENT LEARNING

TRABAJO FIN DE GRADO
CURSO 2020-2021

AUTOR
PEDRO SIMARRO GUERRA

DIRECTORES
KATZALIN OLCOZ HERRERO
LUIS MARÍA COSTERO VALERO

GRADO EN INGENIERÍA INFORMÁTICA

FACULTAD DE INFORMÁTICA

UNIVERSIDAD COMPLUTENSE DE MADRID

APRENDIZAJE POR REFUERZO PARA LA MEJORA DE CALIDAD DE SERVICIO EN PROCESOS DE CODIFICACIÓN DE VÍDEO

IMPROVING QUALITY OF SERVICE FOR VIDEO CODING USING REINFORCEMENT LEARNING

TRABAJO DE FIN DE GRADO EN INGENIERÍA INFORMÁTICA
DEPARTAMENTO DE ARQUITECTURA DE COMPUTADORES Y AUTOMÁTICA

AUTOR
PEDRO SIMARRO GUERRA

DIRECTORES
KATZALIN OLCOZ HERRERO
LUIS MARÍA COSTERO VALERO

CONVOCATORIA: SEPTIEMBRE 2021
CALIFICACIÓN: NOTA

GRADO EN INGENIERÍA INFORMÁTICA

FACULTAD DE INFORMÁTICA

UNIVERSIDAD COMPLUTENSE DE MADRID

DÍA DE MES DE AÑO

RESUMEN

El aprendizaje por refuerzo es uno de los tres grandes paradigmas de la inteligencia artificial junto al aprendizaje supervisado y al no supervisado. Su uso se ha extendido a lo largo de los años y se ha demostrado su eficacia en áreas como la robótica, el trading financiero o incluso en la simulación de videojuegos. Está enfocado principalmente a problemas de optimización, problemas que normalmente son difíciles de resolver y necesitan de un gran despliegue en recursos humanos y técnicos para encontrar una solución. Es por eso por lo que existe la posibilidad de crear un sistema de aprendizaje artificial como es el del aprendizaje por refuerzo que nos facilita la tarea, abaratando costes y automatizando el proceso.

En este trabajo se realizó un estudio sobre el aprendizaje por refuerzo y su aplicación en un ejemplo de uso concreto, la optimización de recursos de un procesador para la ejecución de *Kvazaar*, un codificador de vídeo de código abierto. La tarea consistió en crear un sistema agente-entorno típico del paradigma de aprendizaje por refuerzo usando las librerías RLLIB, una API que proporciona las herramientas crear agentes de aprendizaje por refuerzo y GYM, una librería para crear entornos. A partir de ahí, se discutieron los resultados de tal modo que se siguiese ajustando el modelo hasta llegar a nuestro objetivo anteriormente mencionado.

Palabras clave

Aprendizaje por refuerzo, recompensa, política, GYM, RLLIB, entorno, estado, acción, Kvazaar, multicore.

ABSTRACT

Reinforcement learning is one of the three most important artificial intelligence paradigms alongside supervised learning and non-supervised learning. Its use has been growing meaningfully in areas such as robotics, financial trading, or videogames simulation. It is thought towards optimization problems, problems that are usually hard to solve and need lots of computational and human resources. That is why creating a reinforcement learning environment helps saving costs and automate the process.

This project is an investigation about the reinforcement learning paradigm and its application on a specific use case, resources optimization of a CPU in Kvazaar's execution, an open-source video encoder. The main task was to develop an agent-environment system using RLLIB, an API that brings all the needed tools for creating agents and GYM, a library that standardizes environments. From that, results were discussed to adjust the model until getting the objective, the optimization of CPU resources to get the above-mentioned objective.

Keywords

Reinforcement learning, reward, policy, GYM, RLLIB, Kvazaar, state, action, environment. , multicore.

ÍNDICE DE CONTENIDOS

Capítulo 1 - Introducción

El aprendizaje por refuerzo, como en otras áreas de *machine learning*, basa su funcionamiento en la idea biológica del aprendizaje. Los humanos, por ejemplo, desde nuestra niñez, estamos expuestos al medio que nos rodea, estamos en constante interacción con el entorno. Esta interacción continua, nos da respuestas sobre cómo debemos actuar. A base de la prueba y el error en situaciones similares obtenemos información sobre cuáles son las mejores decisiones ante estas. El aprendizaje por refuerzo es, por tanto, esta idea llevada al ámbito computacional.

El aprendizaje por refuerzo utiliza un concepto fundamental para evaluar el aprendizaje, la recompensa. A cada acción se le asigna una recompensa en función de qué tan buena sea. El objetivo de este paradigma es por tanto maximizar esta recompensa, lo que, consecuentemente, indica que se han tomado las mejores decisiones. El agente, el ente o aprendiz encargado de realizar la toma decisiones, trabaja de manera autónoma en base a su experiencia utilizando el método de la prueba-error. Es tarea del experto indicar cuáles de estas decisiones son las mejores a través de la recompensa, de ahí que hablemos del concepto "refuerzo", premiamos aquellas acciones que son beneficiosas instando a que se tomen de nuevo en el futuro. Este comportamiento de prueba-error es similar a lo que hacemos nosotros los humanos: por ejemplo, al aprender a escribir, repetimos una y otra vez las letras hasta que las delineamos correctamente. En un entorno de aprendizaje por refuerzo, explicándolo a grandes rasgos, indicaríamos al agente cuáles son los mejores trazos para escribir las letras de manera que sean legibles.

Junto al aprendizaje supervisado y al no supervisado, el aprendizaje por refuerzo forma parte del gran grupo de los paradigmas de inteligencia artificial. Se separa de estos dos últimos por su objetivo; los dos anteriores sirven para categorizar situaciones, en el caso del aprendizaje supervisado, y encontrar patrones en datos en el caso del aprendizaje no supervisado [1]. Es por esto, que este paradigma abre una vertiente paralela y nueva, cuyas aplicaciones son muy diversas, como el aprendizaje locomotor en robots, jugar a videojuegos,

la simulación de situaciones en entornos científicos o su uso en sistemas de navegación.

En el capítulo 2 veremos una descripción básica de los conceptos previos para crear nuestro sistema. En el capítulo 3 veremos la implementación siguiendo el plan de trabajo establecido. En el capítulo 4 detallaremos los resultados obtenidos. Por último, en el capítulo 5 discutiremos las conclusiones extraídas de estos y plantearemos mejoras de cara a un trabajo futuro.

1.1 Motivaciones

El uso de este paradigma se ha ido acrecentando en la resolución de tareas de optimización. El aprendizaje por refuerzo escoge cuál es la mejor estrategia para conseguir la mayor recompensa. En nuestro caso, tenemos un problema de optimización de recursos hardware (más concretamente el CPU) sobre una tarea de codificación de vídeo. Veremos cuáles son las bases de este paradigma, así como la manera de adaptar las herramientas de las que disponemos para nuestro problema. Trataremos de generar un agente que aprenda cuáles son las mejores decisiones para codificar vídeos, intentando mantenernos en un rendimiento dado.

En este trabajo abordaremos el siguiente problema: queremos obtener cuál es la manera óptima de utilizar los núcleos de un procesador en una tarea de codificación de vídeo para conseguir un resultado específico, es decir cuántos núcleos necesitamos en cada momento sin malgastar recursos. Nuestro codificador de vídeo recibe una cantidad de núcleos que se van a utilizar para procesar un bloque de vídeo y arroja una cantidad de fotogramas por segundo o FPS como resultado de la operación. Sin embargo, no sabemos cuál es la mejor manera de seleccionar estos núcleos para obtener un rango aceptable de FPS (entre 20 y 30) sin utilizar núcleos de más. Es ahí cuando el aprendizaje por refuerzo nos sirve de utilidad. Podríamos realizar las pruebas a mano, probando todas las opciones para todos los posibles casos de vídeos, pero nos facilitaría mucho la tarea si creamos un sistema de aprendizaje por refuerzo que decida cuáles son las mejores acciones para conseguir tal objetivo y estudiar qué parámetros de este nos permiten optimizar el uso de núcleos. En este trabajo, por tanto, veremos cómo se modela este caso concreto en una

aplicación real de aprendizaje por refuerzo, ajustando sus parámetros para después discutir los resultados sobre su comportamiento.

1.2 Objetivos

Los objetivos del siguiente proyecto fueron los siguientes:

- Conocer las bases del paradigma del aprendizaje por refuerzo.
- Idear y diseñar un entorno de aprendizaje por refuerzo para nuestro problema real usando las herramientas RLLIB y GYM.
- Evaluar los resultados obtenidos con respecto al rendimiento buscado (20 a 30 FPS para Kvazaar) y discutir posibles mejoras del proyecto.
- Por último, afianzar todos los conocimientos adquiridos durante el Grado de Ingeniería Informática.

1.3 Plan de trabajo

Durante el desarrollo del proyecto, se han seguido las siguientes pautas para su realización:

1. Proceso de toma de contacto con el funcionamiento de Kvazaar para su aplicación posterior en el entorno de aprendizaje por refuerzo.
2. Realización de diferentes pruebas iniciales para idear la manera de comunicación entre Kvazaar y el agente del paradigma, que sirva de base para el entorno.
3. Creación propiamente dicha del entorno, ateniéndose a las bases de GYM y RLLIB.
4. Realización de pruebas finales medidoras del rendimiento de aprendizaje del entorno, y discusión de los resultados teniendo en cuenta nuestro objetivo.

Chapter 1 – Introduction

Reinforcement learning, as in other machine learning areas, base its fundamentals in the biological idea of learning. We, humans, since our childhood, are exposed to the external environment, we are constantly interacting with it. With the try and failure method, we obtain information about what the best actions for similar situations. Thus, reinforcement learning brings this idea in a computational way.

The reinforcement learning paradigm uses an essential concept for the evaluation process of learning, the reward. Each action is matched with a reward depending on its goodness. The paradigm has the objective to max the cumulative reward, indicating that good decisions were made. The agent, the entity or learner in charge of the decision-making process, works by its own getting experience using the try and failure method. The expert has then the task of telling the agent through the reward which of those decisions are the best. That is why we are talking about "reinforcement", we reward the most beneficial actions provoking that they will be chosen in the future. This try and failure behavior resembles the one human have: for example, when we learn to write, we repeat over and over until we draw the correctly. In a reinforcement learning environment, we would guide tell the agent which the best strokes are to write legible letters.

Alongside supervised learning and non-supervised learning, reinforcement learning belongs to the big group of three of artificial intelligence paradigms. It differs from them in its objective: supervised learning is used for categorization of situations and non-supervised learning, for data pattern spotting [1]. That is why this paradigm opens a new area of investigation parallelly, whose applications are very diverse, from movement learning in robots, playing videogames, simulation in scientific purposes to navigation systems.

In chapter 2, we will see a description of the basic concepts prior to the creation of our system. In chapter 3, we will go through the implementation following the established workplan. In chapter 4, we will detail the obtained

results. Finally, in chapter 5, we will discuss the conclusions extracted from them and we will propose future work ideas.

1.1 Motivations

The use of this paradigm has been growing exponentially it the last years. Reinforcement learning tries to get the best strategy to approach best rewards. In our case, we are facing a hardware resource optimization problem (more deeply a CPU optimization one) in a video codification task. We will see what the paradigm basis are and how we can adapt the currently available tools for our problem. We will develop an agent capable of deciding what is the best way of encoding videos, trying to stay in between a specified performance.

In this work, well will go through the following problem: we want to get the optimum way of managing the cores of a CPU during a video encoding task to obtain a specified performance. In other words, how many cores do we have to select in specific moment of time without wasting resources. Our video encoder takes as input number of cores value to process a video block and returns an FPS or frames per second value, as a result. However, we do not know what the best way is of selecting those cores to get an acceptable performance (between 20 and 30 FPS) without using too many resources. That is when reinforcement learning comes in action. We could make all the trails ourselves, for all options and all the possible use cases, but it would much easier if we built a reinforcement system capable of doing this work for us. Thus, in this work , we will see the modeling process of a real reinforcement learning use case. After, we will discuss the results of its behavior.

1.2 Objectives

This project objectives were the following:

- To know the basis of the reinforcement learning fundamentals.
- To design a reinforcement learning environment for our problem following RLLIB and GYM's instructions.
- To evaluate and discuss the results attaining the searched objective (20 to 30 FPS for Kvazaar).
- Finally, to prove all the knowledge from this Degree.

1.3 Workplan

During this project development, this workplan has been followed:

1. Contact making process with Kvazaar functionality for its application later in the environment.
2. Initial tests process to think up the way of communication between Kvazaar and the agent. This will be the base of the reinforcement learning environment.
3. To build the actual environment, using RLLIB and GYM's basis.
4. Final tests process to measure the learning rate of the environment, focusing on our objectives.

Capítulo 2 - Conceptos previos y herramientas de trabajo

En este capítulo abordaremos las nociones básicas sobre el aprendizaje por refuerzo, así como la descripción de las herramientas usadas que utilizamos en el proyecto, para poder comprender mejor su contexto.

2.1 Aprendizaje por refuerzo

Como ya habíamos planteado anteriormente, el aprendizaje por refuerzo tiene la idea básica de maximizar una recompensa para un determinado problema. Consiste en encontrar las mejores decisiones que nos consigan tal objetivo. Esta idea de escoger el mejor conjunto de decisiones tiene su definición formal como *Procesos de Decisión de Markov*.

Los Procesos de Decisión de Markov (MDP, por sus siglas en inglés *Markov Decision Processes*) definen de manera matemática el proceso de toma de decisiones en el que las acciones sobre un entorno influyen en las recompensas y situaciones futuras de este. Establecen la estructura básica del problema de aprendizaje por refuerzo. A continuación, veremos algunos de los conceptos básicos que formalizan los MDP.

El aprendiz o selector de decisiones se denomina **agente** y con lo que interactúa, **entorno**. Estos se comunican constantemente: el agente selecciona acciones y el entorno responde ante ellas con nuevas situaciones o estados, dándoles una **recompensa** numérica que el agente tratará de maximizar a lo largo del tiempo basándose en sus elecciones [1]. El agente, por tanto, interactúa un número indefinido de pasos. En cada paso, el agente recibe una representación del entorno llamada **estado**, y a partir de él selecciona una **acción**. Un instante en el tiempo después, el agente recibe una recompensa numérica como consecuencia de su acción y se encuentra en un nuevo estado del entorno. De esta forma, las múltiples interacciones consecutivas generan lo que se denomina **trayectoria**. En la siguiente figura se ve este proceso cíclico.

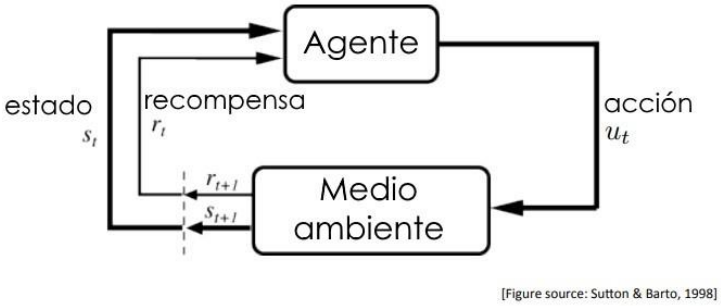


Figura 2-1: Modelo básico del comportamiento del aprendizaje por refuerzo (imagen de Sutton & Barto 1998).

En la Figura 2-1 vemos el funcionamiento básico del paradigma: el agente realiza una acción u sobre un estado s del entorno al que se le asocia una recompensa r en un momento t. En el instante siguiente $t+1$ el agente recibe el nuevo estado s_{t+1} y la recompensa nueva r_{t+1}. Esto se mantiene indefinidamente en el tiempo.

En los algoritmos de aprendizaje por refuerzo, existen funciones estimatorias llamadas **funciones de valor**. Estas funciones determinan qué tan bueno es el comportamiento de un agente en un determinado estado. Realmente, el decir qué tan buenas son las decisiones que toma el agente viene dado por el **retorno esperado**. El retorno esperado es una estimación de las recompensas que se esperan obtener en un momento dado. Estas funciones de valor definen la manera de actuar del agente, llamada **política.** Una política es un mapeo de los estados con las probabilidades de seleccionar cada una de las posibles acciones. Si el agente sigue una política π en un momento t, $\pi(a|s)$ es la probabilidad de tomar la acción a dado en un estado s. Los algoritmos de aprendizaje por refuerzo definen cómo cambia la política a través de la experiencia del agente.

Durante un entrenamiento del agente, decimos que existe una política π mejor o igual que otra π' si el retorno esperado de la primera es mayor o igual que la segunda. Es decir,

$$\pi \geq \pi' \Leftrightarrow v_{\pi}(s) \geq v_{\pi'}(s) \qquad [1]$$

Siempre hay al menos una política que es mejor que todas las demás, a esta la llamamos **política óptima** [1]. El objetivo final de los algoritmos de

aprendizaje por refuerzo es llegar a esta política óptima. En la ecuación anterior, $v_\pi(s)$ simboliza la función del valor que estima el retorno esperado comenzando desde un estado s siguiendo la política π.

Llegar a una política óptima en la mayoría de los problemas es bastante costoso computacionalmente y el tiempo y los recursos necesarios dependen sobre todo de cómo escogemos el algoritmo, y de cómo definimos los estados y recompensas. No es lo mismo, por ejemplo, tener un conjunto de acciones y estados continuos, que discretos, o utilizar algoritmos centrados en un único tipo de estos espacios, puesto que no rendirán de la misma manera. O, por ejemplo, tener un conjunto de recompensas únicamente positivas que una mezcla entre positivas y negativas, ya que podríamos indicarle al agente un comportamiento que no queremos. Además, todos estos algoritmos se pueden modificar a nuestro antojo. Muchas veces, incluso si el agente ha aprendido un entorno diseñado perfectamente, puede que haya tomado decisiones que a priori no se habían tenido en cuenta. La aleatoriedad está muy presente aquí.

Es por eso por lo que es necesario realizar un estudio previo del entorno antes de tomar estas decisiones ya que este campo del aprendizaje automático presenta muchas sutilezas. Sin embargo, es innegable el potencial y capacidades que presenta. Todo depende del tiempo que se le dedique al estudio de un caso y de la manera en la que se evalúan los comportamientos del agente. En los siguientes subcapítulos trataremos las herramientas que se han utilizado en el proyecto, que se basan en las premisas que acabamos de introducir.

2.2 RLLIB

RLLIB [2] es una librería de código abierto escrita en Python que consiste en una API escalable que nos brinda las herramientas necesarias para crear aplicaciones de aprendizaje por refuerzo. Funciona a través de Ray, un entorno universal para aplicaciones distribuidas. RLLIB realiza todas las abstracciones pertinentes para encapsular las tareas de aprendizaje por refuerzo usando la arquitectura de Ray como base. Soporta gran cantidad de algoritmos como A2C, DQN o PPO, este último el algoritmo utilizado en este proyecto.

El funcionamiento de RLLIB es sencillo: existen clases llamadas ***Trainers*** que modifican la política y tratan de optimizarla, cada cierto número de acciones sobre el entorno. Los datos de entrenamiento, es decir, estos pasos o acciones, son recogidos por ***Rollout Workers*** en bloques llamados ***sample batches***. Estos Rollout Workers pueden funcionar en paralelo. Cuando trabajamos con RLLIB, se gener5an redes neuronales de manera automática y subyacente. El Trainer realiza los cambios en los pesos de estas redes neuronales que se pasan a los Rollout Workers para continuar recopilando acciones usando los nuevos pesos. RLLIB usa actores de Ray para escalar el entrenamiento desde un único núcleo a miles de núcleos en un bloque [2]. Se puede modificar el paralelismo del entrenamiento cambiando el número de workers.

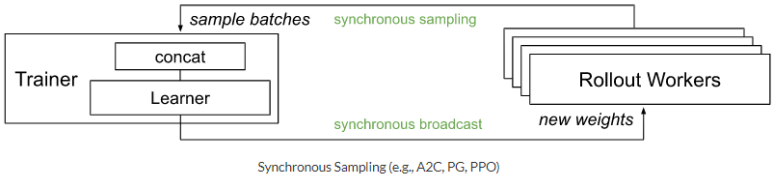


Figura 2-2: : Diagrama del entrenamiento en RLLIB (imagen extraída de la documentación de RLLIB).

Las políticas y los agentes en RLLIB son básicamente clases Python que definen cómo actúan estos en el entorno. Dependiendo del caso, se pueden definir desde uno a varios agentes actuando simultáneamente. Si existe más de un agente en el entorno, también existe la posibilidad de tener más de una política, hasta una por agente. En la Figura 2-3 vemos el esquema de los entornos en RRLIB. En los entornos vectoriales, múltiples agentes actúan sobre la misma política y en los entornos multiagente diferentes agentes actúan sobre más de una política, no necesariamente la misma. En los entornos GYM, más simples, solo hay un agente y una política. En nuestro caso utilizaremos un entorno GYM.

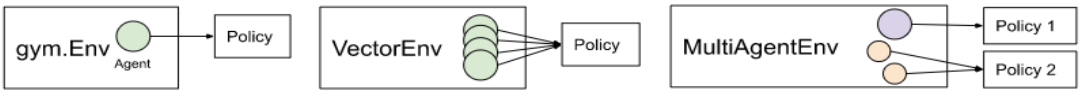


Figura 2-3: Políticas y agentes en RLLIB (imagen extraída de la documentación de RLLIB).

RLLIB ofrece una gran posibilidad de personalización ya que se pueden modificar prácticamente todos los aspectos del entrenamiento: la red neuronal, las distribuciones de acciones, la definición de políticas, el entorno o incluso el proceso de recolección de acciones. A continuación, se muestran aquellos componentes de la arquitectura que el usuario puede modificar.

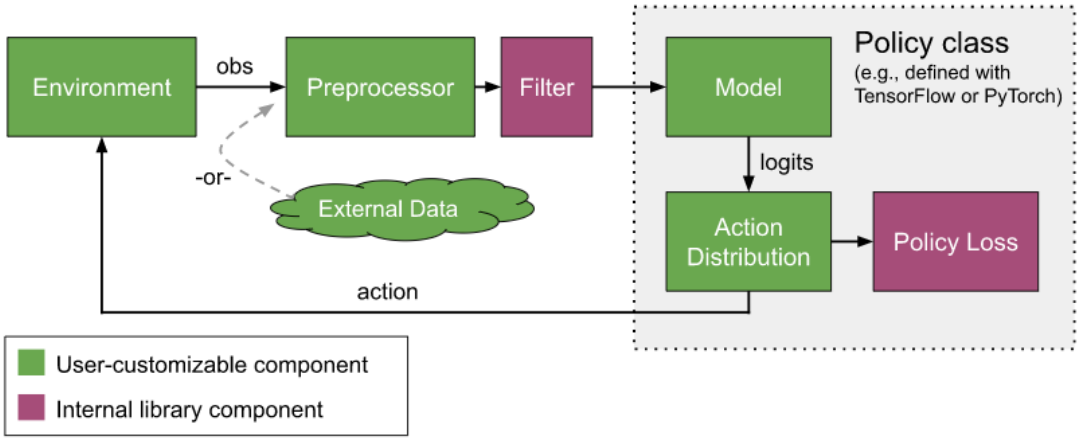


Figura 2-4: Arquitectura de RLLIB (imagen extraída de la documentación de RLLIB).

2.2.1 El Algoritmo PPO

El algoritmo *PPO*[1] (*Proximal Policy Optimization* en inglés), utilizado en este proyecto, está desarrollado por OpenAI y pertenece al grupo de los algoritmos de Descenso de Gradiente. Pretende simplificar el algoritmo *TRPO* (*Trust Region Policy Optimization*) así como brindar más paralelismo a la hora de ejecutar entrenamientos. Se ha demostrado su versatilidad y robustez en gran cantidad de problemas y por ello, es la opción utilizada en este proyecto.

2.2.2 Definición formal: objetivo y mejora frente a TRPO

Antes de formalizar el algoritmo, veamos sus ideas iniciales. Los algoritmos de descenso de gradiente pretenden optimizar la siguiente función:

$$L^{PG}(\theta) = \widehat{E_t}\left[log\pi_\theta(a_t \mid s_t)\,\widehat{A_t}\right] \qquad [3]$$

Esta es la **función de pérdida de la política** y viene definida por una serie de parámetros θ. Su valor viene dado por el estimado de la multiplicación entre el logaritmo de la política π y una "ventaja" (A_t) que es el resultado de la función de valor de la acción a para un estado s aplicando la política π en un momento t , es decir el retorno esperado al aplicar la acción a sobre el estado s . La política π se implementa como una red neuronal que indica el comportamiento que debe tomar el agente, si la ventaja es positiva, significa que las acciones

1 HTTPS://OPENAI.COM/BLOG/OPENAI-BASELINES-PPO/

tomadas por el agente dan un retorno mejor de lo esperado respecto al retorno medio, y por tanto el descenso de gradiente será positivo, instando a la elección de estas acciones en un estado similar. Por el contrario, si la ventaja es negativa, ocurrirá exactamente lo contrario [3].

El problema de esta función es la posibilidad de "destrucción" de la política al realizar múltiples descensos de gradiente en una misma muestra de acciones. Por ello, surgió el algoritmo TRPO, que trata de impedir cambios en la política demasiado alejados a la actual.

Se define una proporción $r_t(\theta)$ de probabilidad entre las acciones de una política y su anterior. Si esta proporción es mayor que 1, significa que las acciones son más probables en la política actual que en la anterior y si está entre 0 y 1, son más probables en la anterior.

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \quad [3]$$

Ahora si se multiplica esta proporción con la anterior ventaja, tenemos la función que maximiza TRPO:

$$\begin{gathered} maximizar \quad \widehat{E_t}\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}\widehat{A_t}\right] \\ con \ \ \widehat{E_t}\left[KL\left[\pi_{\theta_{old}}(\cdot\,|s_t), \pi_\theta(\cdot\,|s_t)\right]\right] \leq \alpha \end{gathered} \quad [3]$$

La diferencia con el primero objetivo es que el logaritmo de la política ahora está cambiado por la proporción $r_t(\theta)$. Además, TRPO añade una restricción de Kullback-Lieber que limita al paso de gradiente para que no se aleje demasiado de la política actual. Por ello, se sabe que la política siempre se mantiene en una "región de confianza" funcional, de ahí el nombre del algoritmo. Sin embargo, añadir esta restricción supone un mayor gasto computacional y en ocasiones da lugar a comportamientos no deseados del agente [3].

PPO por tanto trata de simplificar este último algoritmo intentando mantener sus características. Para ello, hace uso del siguiente objetivo:

$$L^{CLIP}(\theta) = \widehat{E_t}\left[min\left(r_t(\theta)\widehat{A_t}, clip(r_t(\theta), 1+\epsilon, 1-\epsilon)\widehat{A_t}\right)\right] \quad [3]$$

Lo que hace este objetivo es calcular el estimado de un mínimo entre dos términos: el primer término es el mismo que se encuentra en TRPO, y el segundo utiliza la proporción $r_t(\theta)$ truncada entre $1 + \varepsilon$ y $1 - \varepsilon$, siendo ε un hiperparámetro del algoritmo. Este truncamiento se denomina en inglés *clip*. Dependiendo del signo de la ventaja, esta función *clip* se comporta de manera diferente:

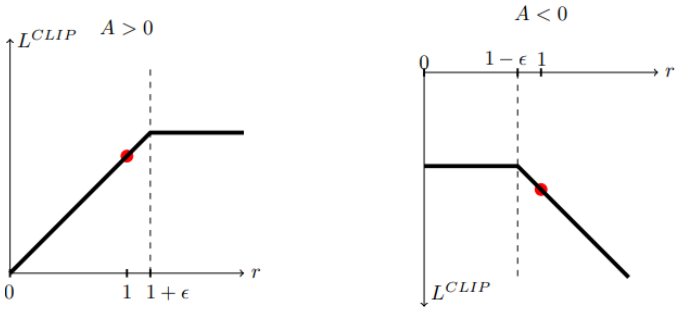


Figura 2-5: Funcionamiento de la función clip de PPO (Schulman et al., 2017)

En la figura anterior, en la parte izquierda, cuando la proporicón r se vuelve demasiado alta, es decir cuando la acción es más posible en la política actual vemos que, si la acción tomada tiene un rendimiento superior al esperado (ventaja A positiva), la función de pérdida se vuelve constante a partir de $1 + \varepsilon$. De esta forma, se trunca el objetivo para prevenir que la actualización de la política no tome un paso demasiado alejado.

De una manera similar, como se observa en la parte de la derecha cuando la acción es muy poco probable en la política actual (proporción r baja), y la ventaja A es negativa, es decir, la acción tomada tiene un rendimiento peor de lo esperado, se trunca la función para evitar cambios muy bruscos en la política.

Este truncamiento permite resolver errores en la actualización de la política. Por ejemplo, si ocurre el caso en el que el algoritmo selecciona una acción haciendo que la política sea peor, esta operación de truncamiento incita a tomar acciones diferentes para corregir el error cometido. Además, PPO elimina el uso de la restricción de KullBack-Lieber usado en TRPO, lo que agiliza el procesamiento del algoritmo.

Finalmente, el **objetivo final** de PPO es el siguiente:

$$L^{CLIP+VF+S}(\theta) = \widehat{E_t}[L_t^{CLIP} - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t)] \quad [3]$$

Añade un par de términos adicionales con c_1 y c_2 como hiperparámetros. El primer término L_t^{VF} es el error cuadrático medio de la función de valor, y el segundo se denomina entropía, encargada de aumentar la exploración del agente [3]. Este término provoca que la política se comporte más aleatoriamente a medida que el resto del objetivo deja de evolucionar.

Finalmente, el funcionamiento de PPO es sencillo, por un lado, los agentes recopilan las pasos o acciones, calculan las estimaciones de sus valores y realizan subdivisiones de las muestras para que posteriormente cada cierto número de pasos (definido en un parámetro llamado ***minibatch***), se realiza un descenso de gradiente estocástico en la anterior función de pérdida.

La ventaja que tiene este algoritmo frente a anteriores versiones es la posibilidad de realizar múltiples iteraciones de optimización, lo cual se veía imposible antes de definir este algoritmo por la posibilidad de arruinar la política debido a la sobreestimulación de la actualización de esta. Además, permite optimizar el uso de las muestras, ayudando a aprender más con cada paso de entrenamiento. Por último, la posibilidad de ejecutar múltiples recolecciones de muestras simultáneamente por varios agentes, opción que no existía antes, ofrece una mejora sustancial en la riqueza de información obtenida por el algoritmo para actualizar la política.

2.3 Entornos en GYM

GYM [4] es una librería de código abierto escrita en Python que permite la definición de entornos de aprendizaje por refuerzo. Se ha demostrado su utilidad en conjunción con RRLIB en la creación de entornos para videojuegos como *Pong* o *Atari*, o en tareas de aprendizaje de movimiento en robots. Además, ofrece una gran versatilidad gracias a que su estructura sirve de base para prácticamente cualquier entorno. Además, la librería contiene muchos ejemplos para familiarizarse y aprender acerca del aprendizaje por refuerzo.

La motivación de usar *GYM* es simple, el aprendizaje por refuerzo tiene dos problemas importantes como se detallan en su sitio oficial [4]: en primer

lugar, se necesitan mejores *benchmarks* a la hora de evaluar las implementaciones y, en segundo lugar, a pesar del auge del aprendizaje por refuerzo debido a su simplicidad teórica y al rendimiento ofrecido en tareas complicadas, no existe estandarización en los entornos usados en las publicaciones. GYM trata de solucionar ambos problemas.

Para poder utilizar esta librería, basta con instalar el paquete de *Pip* correspondiente en nuestro entorno de trabajo. Además, en el repositorio en Github de OpenAI, se encuentran los ejemplos ya mencionados con los que podemos ejecutar sesiones de entrenamiento para ver los resultados.

Sabemos que, en el paradigma del aprendizaje por refuerzo, el entorno se comunica con el agente con cada una de sus acciones. Un entorno en GYM devuelve cuatro datos al agente [4]:

- Una **observación** o **estado**: es una representación de una observación del entorno. Por ejemplo, un *frame* de un videojuego o la posición de un robot.
- Una **recompensa**: un valor en punto flotante que indica la recompensa obtenida al realizar la acción anterior dado el estado en el que se encontraba el entorno.
- Una **señal de terminación** booleana llamada "*done*": indica si el entorno ha terminado un episodio. Llamamos episodio a cualquier tarea bien definida con principio y final. Por ejemplo, escapar de un laberinto, o completar un puzle.
- Un **diccionario de información** del entorno: aquí se puede recopilar información interesante acerca del entorno. Este elemento es opcional ya que el agente no utiliza esta información para el aprendizaje.

2.3.1 Espacios en GYM

Es importante definir cómo se representan el conjunto de estados y de acciones de nuestro entorno. GYM define una serie de espacios que encapsulan la estructura de estos. Existen diferentes tipos de espacios como *Discrete* o *Box*, entre otros. *Discrete* define un espacio discreto mientras que *Box*

representa un espacio continuo de números reales acotado y que puede ser desde una a varias dimensiones. Por ejemplo, el famoso entorno "*Cartpole-v0*"

```
print(env.observation_space.high)
#> array([ 2.4       ,         inf,  0.20943951,         inf])
print(env.observation_space.low)
#> array([-2.4       ,        -inf, -0.20943951,        -inf])
```

Figura 2-7: intervalos del espacio de observaciones de Cartpole-v0 (imagen extraída de OpenAI).

```
import gym
env = gym.make('CartPole-v0')
print(env.action_space)
#> Discrete(2)
print(env.observation_space)
#> Box(4,)
```

Figura 2-6: espacios de Cartpole-v0 (imagen extraída de OpenAI).

representa el entorno de un péndulo que se coloca sobre un carrito accionado por un mecanismo que se puede mover de izquierda a derecha. Su objetivo es mantenerse en equilibrio gracias al movimiento del mecanismo una cantidad de tiempo fijada sin que el carro se salga de un espacio acotado. Este entorno presenta un espacio discreto de acciones con dos acciones (mover el carro a la izquierda o a la derecha) y un espacio continuo de una dimensión con 4 números que representa la posición del péndulo y su inclinación. En la Figura 2-6 y la Figura 2-7 se ve el código asociado a la definición de estos espacios.

Para crear un entorno en GYM basta con implementar una clase Python que herede de la clase *Gym.Env* y definir los siguientes 5 métodos iniciales:

- **__init__(self)**: Este es el constructor de la clase, y aquí, entre otras cosas, definiremos los espacios de observaciones y de acciones.
- **reset(self):** este método se llama al principio del entrenamiento, así como al finalizar un episodio. En él se debe reiniciar todos aquellos atributos a un valor inicial, como el estado y la recompensa. Debe devolver el estado inicial.
- **step(self, action):** se llama cada vez que el entorno da un paso en el entrenamiento. Es decir, se recibe la acción realizada por el agente y se actualizan tanto el estado como la recompensa. Devuelve el estado nuevo, la recompensa obtenida, la señal de *done* y el diccionario de información opcional que puede utilizarse para guardar datos secundarios del estado del entorno, mencionado anteriormente.
- **render(self, mode="human"):** este método devuelve una representación visual del entorno. Existen varios modos de renderizado como *human*

(renderizado por terminal) o *rgb_array* (se devuelve un array que representa los píxeles RGB de una imagen).

- **seed(self, seed=None):** aquí se inicializa la semilla para generar números aleatorios.

```
import gym

class Entorno(gym.Env):

    def __init__(self):
        ## Definir espacio de acciones y observaciones
        ## Crear otros elementos necesarios para el entorno
        ## Creación de semilla para números aleatorios
        pass

    def reset(self):
        ## Resetear otros atributos
        ## Resetear estado y recompensa
        return self.state

    def step(self, action):
        ## Realizar una acción en el entorno
        ## Actualizar estado y recompensa

        return self.state, self.reward, self.done, self.info

    def render(self, mode="human"):
        ## Tarea de renderizado, devolver la representación visual del entorno
        pass


    def seed(self, seed=None):
        ## Creación de la semilla
        pass
```

Figura 2-8: Estructura básica de un entorno en GYM.

Una vez definido nuestro entorno bastaría con registrarlo en nuestro espacio de trabajo para poder utilizarlo con Ray y RLLIB. Más adelante en la parte de implementación se detallará cómo se realiza.

Capítulo 2.4 Kvazaar

Kvazaar[2] es un codificador de vídeo de código abierto desarrollado en la universidad de Tampereen en Finlandia. Está disponible para las plataformas de Windows y Linux. Implementa el estándar HEVC (*High Efficiency Video*

2 HTTPS://GITHUB.COM/ULTRAVIDEO/KVAZAAR

Coding) conocido como H.265. Su funcionamiento básico consiste en procesar una secuencia de vídeo en bruto no comprimida y devolver la misma secuencia de forma comprimida según un estándar dado. En el proyecto nos centramos en la codificación de secuencias de vídeo en bruto (formato *yuv*).

Kvazaar presenta diferentes características que lo hacen sobresalir frente a otros codificadores, de las cuales destacamos las siguientes:

- Su desarrollo en código abierto (licencia LGPL2).
- Implementación muy completa del estándar de codificación HEVC.
- La capacidad de ejecución multihilo.
- La capacidad de codificación en tiempo real (dependiendo de la máquina y los parámetros de codificación).

En este proyecto se utilizó una versión modificada que nos permite seleccionar de **manera dinámica el número de hilos** que utiliza el programa, sin embargo, su instalación es la misma que la versión original.

Para poder instalar esta versión de Kvazaar en Linux, primero deberemos tener instalados los siguientes paquetes:

```
git, automake, autoconf, libtool, m4, build-essential
```

Después haremos lo siguiente en una terminal:

```
$ git clone https://github.com/luismacostero/malleable_kvazaar
$ cd malleable_kvazaar
$ autoreconf -fiv
$ ./configure --prefix=$(pwd)
$ make
$ make install
```

Esta ejecución de comandos compila la versión modificada de Kvazaar y la instala. El argumento `--prefix` permite indicar la ruta de instalación, se puede obviar si se quiere instalar como aplicación del sistema.

2.3.2 Ejecución de Kvazaar

Aunque Kvazaar presenta multitud de opciones de codificación en función de la calidad del archivo de salida, compresión y velocidad, nosotros usaremos un conjunto fijo a lo largo de todas las ejecuciones para mantener

una base robusta en nuestros resultados. Una ejecución típica del programa es la siguiente:

```
$ cd bin/
$ ./kvazaar --input <ficheroEntrada.yuv> --output <ficheroSalida>
--preset=ultrafast --qp=22 --owf=0 --threads=<nThs>
```

Donde cada parámetro corresponde a:

- `--input`: fichero de entrada.
- `--output`: fichero de salida, o `/dev/null` si no se quiere guardar.
- `--preset`: este parámetro define de una sola vez otros parámetros. Influye directamente en la compresión, la calidad y el tiempo de codificación. Existen diferentes opciones disponibles, pero en el proyecto se usó la opción `ultrafast` que nos da un buen compromiso entre los anteriores factores.
- `--qp`: determina la calidad del fichero de salida. Cuanto más bajo es el valor, mejor calidad tendrá. Esto obviamente incrementa los recursos y el tiempo necesarios para codificar un vídeo. El valor 22 es típico aquí.
- `--owf`: número de frames que se codifican simultáneamente. Se usó el valor 0.
- `--threads`: número de hilos que se crean al lanzamiento de la ejecución. No determina el número de hilos que se usan en la codificación.

Una vez lanzada una ejecución, Kvazaar permite establecer de manera dinámica el número de hilos que se utilizan en la codificación de un bloque de vídeo. Por defecto, cada bloque tiene 24 *frames*, pero se puede modificar cambiando la variable de entorno `NUM_FRAMES_PER_BATCH`. En cada bloque, Kvazaar requiere del número de hilos por entrada estándar y devuelve el resultado de FPS obtenidos por salida estándar.

Lectura del número de hilos:

Para indicar el número de hilos, Kvazaar esperará la entrada de una línea con el formato

```
nThs:N\n
```

donde N es el número de hilos a utilizar entre 1 y el número máximo indicado al lanzar la ejecución (parámetro `--threads` de Kvazaar).

Escritura de los FPS obtenidos:

Por cada bloque procesado, Kvazaar mostrará los FPS obtenidos en el formato

```
FPS:19.459454\n
```

Además, una vez acabada la ejecución, Kvazaar mostrará el mensaje END, seguido de los FPS globales obtenidos en toda la ejecución. Hay que tener en cuenta que al contrario que las otras medidas de FPS, esta medida sí incluye los tiempos de espera para recibir el número de hilos, por lo que puede ser utilizado para medir el tiempo de reacción del servidor. El formato de este mensaje es:

```
END\n
Overall_FPS:9.9047345\n
```

```
$ NUM_FRAMES_PER_BATCH=100 ./kvazaar --input
QuarterBackSneak1_1280x720_30p.yuv
--output /dev/null --preset=ultrafast --qp=22 --owf=0 --threads=20 2>/dev/null
nThs:3
FPS:19.459319
nThs:20
FPS:31.498376
nThs:2
FPS:13.935963
nThs:8
END
Overall_FPS:9.908793
```

Figura 2-9: Ejemplo de ejecución de Kvazaar.

Capítulo 3 - Implementación

En el siguiente capítulo se abordará el plan de trabajo desarrollado para la realización de la implementación del proyecto. Se verá paso a paso los métodos utilizados para crear el entorno de aprendizaje por refuerzo para nuestro problema, así como la creación y ejecución de entrenamientos. Todo el código fuente se encuentra disponible en el repositorio de Github accesible desde https://github.com/psimarro/TFG.

3.1 Primeros pasos

Antes de la creación del entorno en GYM, fue necesario establecer la manera en que el agente de aprendizaje por refuerzo se comunicaría con Kvazaar. Era pertinente encapsular la ejecución de las codificaciones de vídeo para poder integrarla en el entorno. Para ello, se creó unos archivos de prueba como primeras tareas (se pueden encontrar en los directorios *tarea-1* y *tarea-2* del repositorio) y que servirían como base de la función *step* del entorno, encargada de realizar una acción sobre él.

3.1.1 Tarea 1

En esta primera tarea se desarrolló un controlador en Python capaz de ejecutar un **subproceso** a partir del proceso principal. Esto se realiza gracias al módulo *subprocess* de Python.

Este módulo nos permite lanzar cualquier comando como parámetro usando la sintaxis de línea de comandos tradicional. En esta tarea, el subproceso consistió en un programa básico escrito en C++ que recibe por entrada estándar un número y devuelve por salida estándar la suma de los números hasta ese número. El **controlador** se encarga de lanzar la instrucción:

```
proc = subprocess.Popen(["./controlado"], stdin=subprocess.PIPE,
stdout=subprocess.PIPE, universal_newlines=True, bufsize=1)
```

Que ejecuta el comando del programa en C++ compilado cuyos argumentos se escriben en una lista. La instrucción devuelve la información del subproceso en una variable. Los parámetros *stdin* y *stdout* crean tuberías para

manejar la entrada y salida estándar del subproceso. Se puede acceder a estas como atributos de la variable *proc* generada por *Popen*.

3.1.2 Tarea 2

Esta tarea es una prolongación de la primera tarea. En este caso, se sustituyó el comando del archivo compilado en C++ por la ejecución de una codificación de vídeo en Kvazaar. El funcionamiento de esta tarea es similar a la anterior: se pasa el comando de Kvazaar como argumento de la instrucción *Popen* en forma de lista de argumentos:

```
comando = [kvazaar_path, "--input", vid_path, "--output", "/dev/null", "--
preset=ultrafast", "--qp=22", "--owf=0", "--threads=" + str(nCores)]

# creamos subproceso
proc = subprocess.Popen(comando, stdin=subprocess.PIPE, stdout=subprocess.PIPE,
universal_newlines=True, bufsize=1, env={'NUM_FRAMES_PER_BATCH': '24'})
```

Como vemos, el comando es una lista con los argumentos de una ejecución típica de Kvazaar y se crean las tuberías para la entrada y salida estándar. El argumento *env* permite establecer un diccionario con variables de entorno para el subproceso. En este caso la variable de entorno *NUM_FRAMES_PER_BATCH* no es realmente necesaria puesto que Kvazaar funciona por defecto a ese valor 24.

Una vez definido el subproceso podemos comunicarnos con Kvazaar utilizando las tuberías *stdin* y *stdout* generadas por el subproceso de esta forma:

```
def input_handler(stdin, stdout):
	random.seed() #generamos la semilla para randoms
	while(True):
		s = "nThs:" + str(random.randint(1, nCores))
		print(s)
		stdin.write(s + "\n")
		output = stdout.readline()
		print(output.strip())
		if(output.strip() == "END") :
			break
```

En este método creamos un bucle en el que, en cada iteración, mandamos una cantidad de núcleos aleatoria para codificar un bloque. El bucle termina cuando se recibe por la salida del subproceso la cadena *END*, que simboliza la finalización de la codificado del vídeo.

Con *stdin.write()* escribimos por entrada estándar lo que haríamos por teclado. Y con *stdout.readline()* leemos la salida de Kvazaar. De esta forma tenemos la comunicación básica con Kvazaar que usaremos en el entorno.

3.2 Integración con GYM

3.2.1 Estructura del módulo del entorno

Una vez se estableció la manera de comunicación con Kvazaar, ya se pudo empezar a crear el entorno utilizando la infraestructura de GYM. Para ello, se creó el entorno como un módulo instalable a través de *Pip* usando la siguiente estructura de directorios:

```
kvazaar_gym
│   README.md
│   setup.py
│
└──kvazaar_gym
    │   __init__.py
    │
    └──envs
           kvazaar_env.py
           __init__.py
```

El archivo *setup.py* establece el nombre y las dependencias del módulo final.

El archivo *kvazaar_gym/kvazaar_gym/__init__.py* realiza el registro del entorno en GYM y le da un alias que se utilizará posteriormente para referirnos a él:

```
from gym.envs.registration import register

register(
    id='kvazaar-v0',
    entry_point='kvazaar_gym.envs:Kvazaar',
)
```

En el directorio *envs/* se sitúan todos los entornos GYM del módulo. En este caso solo tenemos uno, el entorno de Kvazaar llamado *kvazaar_env.py*. El último *__init__.py* situado en el directorio *envs/* importa la clase del entorno.

3.2.2 Entorno de Kvazaar

Como habíamos introducido en el capítulo 1, para crear un entorno en GYM es necesario escribir una clase Python que hereda de la clase *Gym.Env* e implementar los 5 métodos básicos. No vamos a detallar cómo se instanció cada uno de ellos, pero sí entraremos en la definición del método más importante, el método *step*:

```
def step(self, action):
        if self.done:
            pass
        else:
            assert self.action_space.contains(action)
            action += 1

            output = self.call_kvazaar(action)

            self.info["kvazaar"] = "running"
            #Check if kvazaar is done
            if(output == "END"):
                self.reset_kvazaar()
                output = self.call_kvazaar(action)
                self.info["kvazaar"] = "END"
                #self.done = True

            ##update metrics
            self.total_steps += 1
            self.episode_steps += 1
            self.video_usage[self.vid_selected['name']] += 1

            self.calculate_state(output=output)

            #log this num step and the video used
            if self.logger:
                self.logger.info("Step {} : fps {}, action  {}".format(str(self.total_steps),
                                                                       str(self.info.get("fps")),
                                                                       str(action)))

            if self.total_steps == self.max_steps:
                #reached end of traininng
                self.log_metrics()

            if self.episode_steps == 50 :
                  self.done = True

        try:
            assert self.observation_space.contains(self.state) #check if new state is valid
        except AssertionError:
            print("INVALID STATE", self.state)

        return [self.state, self.calculate_reward(), self.done, self.info]
```

Figura 3-1: Implementación del método step del entorno.

Como vemos, el método *step* llama a *call_kvazaar(action)*, un método auxiliar que realiza prácticamente lo mismo que en la tarea 2 cambiando el

comportamiento aleatorio de esta última: recoge la acción a realizar en el paso (acción generada por el agente), un número entero, y devuelve una cadena con la salida. Más adelante, en el método *calculate_state(output)*, se actualiza el estado dependiendo de esa salida. El método *calculate_reward()* devuelve la recompensa asociada al estado. Este método *step* devuelve como ya dijimos anteriormente, el estado, la recompensa, el diccionario de información, y la variable *done*.

Cabe destacar que el entorno reinicia Kvazaar cada vez que se termina de procesar un vídeo, es decir, cuando se recibe la salida *END* se vuelve a ejecutar otro comando de Kvazaar con un vídeo nuevo. Además, la variable *done* se vuelve verdadera cuando el número de pasos llega a 50, lo que supone un episodio. Se escogió este valor puesto que realmente nuestro entorno no tiene un estado final bien definido como podría ser escapar de un laberinto o ganar una ronda de un juego. Podríamos pensar que acabar con la codificación de un vídeo terminaría el episodio, pero no todos los vídeos tienen el mismo número de bloques y es más difícil optimizar la política del agente cuando existen estados finales que no se consiguen con el mismo número de pasos. Por ello, se fijó un número de pasos a este valor para obtener resultados más consistentes.

3.2.3 Recompensas y Espacios de observaciones y acciones

Todo entorno de GYM debe establecer su conjunto de estados u observaciones, así como el de acciones. En nuestro caso, sabemos que Kvazaar devuelve el número de FPS que se consiguen al codificar un bloque de un vídeo. Esta sería nuestra observación, sin embargo, tras probar diferentes espacios de observaciones, se concluyó en la toma de un espacio discreto en el que cada estado corresponde a un intervalo de FPS. Esto es porque para nuestro estudio tener un número de FPS muy cercano a otro no supone una diferencia significativa en el rendimiento. Por ejemplo, un resultado de 20 FPS es prácticamente el mismo a uno de 21.

Para el espacio de acciones, tenemos un espacio discreto correspondiente al número de núcleos que se le asignan a Kvazaar. RLLIB utiliza ciertos recursos para ejecutar la tarea de actualización de la política, por tanto, para aislar el rendimiento de Kvazaar y evitar así interferencias entre los distintos

procesos, se dividió el conjunto de núcleos disponibles entre 2, la primera mitad de núcleos se destinaron a la ejecución de Kvazaar y la otra mitad, a las tareas de RLLIB y Ray. Por tanto, el espacio de acciones tendrá tantas acciones como la primera mitad de núcleos.

En cuanto a las recompensas, cada estado tiene una recompensa diferente para indicarle al agente que cada estado es distinto a cualquier otro. Se probaron distintos conjuntos de recompensas inicialmente, pero se llegó a la conclusión de que una mezcla entre recompensas negativas y positivas nos daba una capacidad de aprendizaje más rápida, puesto que las recompensas negativas aceleran la toma de decisiones [5].

Estado	Intervalo de FPS	Recompensa
0	[0, 10)	-25
1	[10,16)	-20
2	[16, 20)	-15
3	[20, 24)	1
4	[24, 27)	2
5	[27, 30)	5
6	[30, 35)	0
7	[35, 40)	-5
8	[40, ∞)	-10

Tabla 3-1: Definición de estados y recompensas.

Como se puede observar en la anterior tabla, la manera de establecer las recompensas define lo que queremos que aprenda el agente. Los estados 3, 4 y 5 son los estados objetivo, y por ende sus recompensas son positivas, instando al agente a estar en esos estados para conseguir la mayor recompensa posible. El resto de los estados tienen recompensas negativas acordes a qué tan bueno es un estado respecto a otro. Los estados que tienen un intervalo de FPS superior al de los intervalos objetivo (6, 7 y 8), tienen una recompensa mayor que los que están por debajo (0, 1, 2). De esta forma, siempre intentaremos que el agente se mantenga en valores por encima de 20 FPS, que consideramos un valor mínimo aceptable teniendo en cuenta que 24 FPS es lo estándar para una codificación en tiempo real.

3.3 Lanzamiento de entrenamientos

Una vez definido nuestro entorno se pueden lanzar entrenamientos creando un agente de Ray. Cada agente tiene una configuración específica dependiendo del algoritmo utilizado. En nuestro caso utilizamos una extensión de la configuración base de PPO, ya que se cambió el tamaño de *batch*, y el tamaño de *minibatch* que indican el número de muestras que toma el agente antes de actualizar la política y el número de subdivisiones de estas muestras anteriores para realizar el descenso de gradiente, respectivamente. Además, se modificó el número de *workers* de RLLIB a 1, puesto que solo queremos tener una instancia de Kvazaar por entrenamiento.

Para lanzar un entrenamiento es necesario registrar antes el entorno en Ray de la siguiente manera:

```
select_env = "kvazaar-v0"
register_env(select_env, lambda config: Kvazaar(kvazaar_path=kvazaar_path,
                                                vids_path=vids_path_train,
                                                cores=kvazaar_cores,
                                                mode=kvazaar_mode,
                                                logger=video_logger,
                                                num_steps=batch*n_iters,
                                                batch=batch,
                                                kvazaar_output=False,
                                                rewards_map=rewards
                                                        ))
```

La variable *select_env* se corresponde al alias que usamos a la hora de registrar el entorno en GYM. El método *register_env*, importado del módulo *ray.tune.registry* registra el entorno con el alias introducido y la configuración dada mediante un objeto de la clase del entorno. En este caso, nuestra configuración del entorno presenta multitud de opciones, entre ellas, la ruta del ejecutable de Kvazaar, el intervalo de núcleos para Kvazaar, el número de pasos totales del entrenamiento, o un mapeo de las recompensas con los estados de observaciones.

Para definir el agente y su configuración realizamos lo siguiente:

```
config = ppo.DEFAULT_CONFIG.copy()
config["log_level"] = "WARN"
config["num_workers"] = 1
config["train_batch_size"] = batch
config["rollout_fragment_length"] = batch
config["sgd_minibatch_size"] = mini_batch
config["callbacks"] = MyCallBacks
```

```
agent = ppo.PPOTrainer(config, env=select_env, logger_creator=ray_results_logger)
```

La variable *agent* guardará el agente creado con la configuración dada por *config* que como vemos usa la configuración base de PPO cambiando los parámetros que queramos a continuación. El *logger_creator* es un logger de los resultados del entrenamiento personalizado para que se guarden en la ruta *resultados/* del proyecto.

Finalmente, para lanzar el entrenamiento se usa el siguiente bucle:

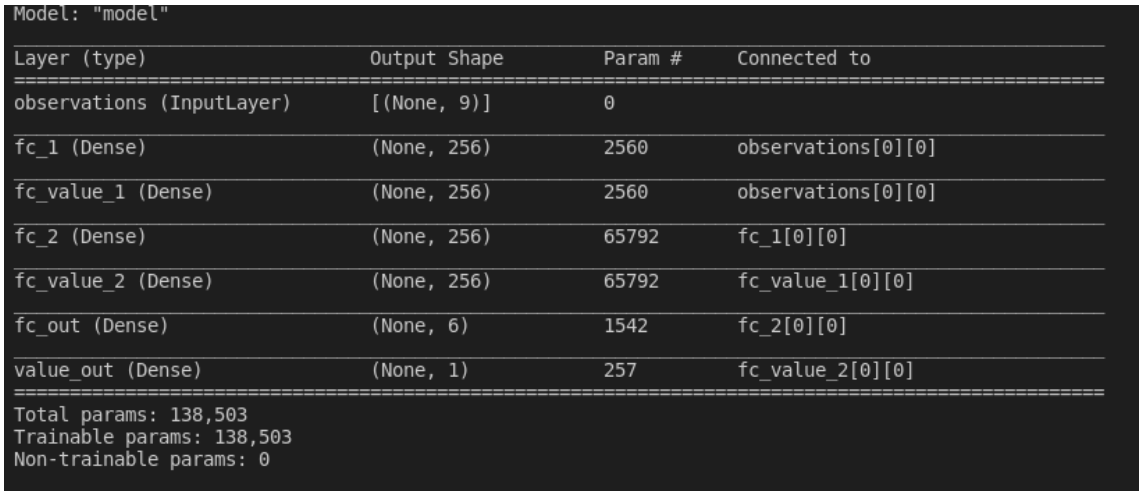
```
# train a policy with RLlib using PPO
for n in range(n_iters):
        try:
                result = agent.train()
                chkpt_file = agent.save(chkpt_root)

                print(status.format(
                                n + 1,
                                result["episode_reward_min"],
                                result["episode_reward_mean"],
                                result["episode_reward_max"],
                                result["episode_len_mean"],
                                chkpt_file
                                ))
        except RayError:
                agent.stop()
                print("Training stopped.")
                exit()
        except KeyboardInterrupt:
                agent.stop()
                print("Training stopped.")
                exit()

# examine the trained policy
policy = agent.get_policy()
model = policy.model
print(model.base_model.summary())
```

Se realizan *n_iters* iteraciones de entrenamiento y se guardan los resultados de estas con *agent.save(chkpt_root)* en lo que se denomina *checkpoints*: cada iteración tendrá un checkpoint asociado. Más adelante estos resultados se pueden utilizar para seguir con un entrenamiento o para ejecutar una agente con la política aprendida.

Al final del entrenamiento se muestra un resumen del modelo de la política que nos brinda información acerca de la red neuronal subyacente que se genera con la cantidad de parámetros que tiene:



```
Model: "model"
_________________________________________________________________________________________________
Layer (type)                 Output Shape              Param #     Connected to
=================================================================================================
observations (InputLayer)    [(None, 9)]               0
_________________________________________________________________________________________________
fc_1 (Dense)                 (None, 256)               2560        observations[0][0]
_________________________________________________________________________________________________
fc_value_1 (Dense)           (None, 256)               2560        observations[0][0]
_________________________________________________________________________________________________
fc_2 (Dense)                 (None, 256)               65792       fc_1[0][0]
_________________________________________________________________________________________________
fc_value_2 (Dense)           (None, 256)               65792       fc_value_1[0][0]
_________________________________________________________________________________________________
fc_out (Dense)               (None, 6)                 1542        fc_2[0][0]
_________________________________________________________________________________________________
value_out (Dense)            (None, 1)                 257         fc_value_2[0][0]
=================================================================================================
Total params: 138,503
Trainable params: 138,503
Non-trainable params: 0
```

Figura 3-2: Resumen del modelo generado.

Como se puede observar en la figura, la red neuronal generada está formadas por varias capas con diferentes tipos de neuronas. Presenta una capa de observaciones inicial con forma *[(None, 9)]* que se corresponde con los 9 estados que habíamos definido y suponen la entrada de la red. En la capa *value_out,* vemos la salida de la red neuronal con forma *(None, 1)* que se corresponde con la acción que toma el agente, que es una cada vez. El resto de las capas son intermedias entre las dos. Aquí vemos la conveniencia de utilizar RLLIB y GYM para desarrollar nuestro entorno ya que las capas intermedias se generan automáticamente. El número de parámetros es la suma de parámetros de cada una de las capas. Este valor influye directamente en la velocidad de convergencia de la red, así como en el gasto computacional para conseguirla.

3.4 Callbacks personalizados

Como hemos visto en la configuración del agente, RLLIB nos brinda la oportunidad de modificar los *callbacks* del agente. Estos *callbacks* son tareas que se ejecutan en determinados momentos del entrenamiento, como al finalizar un episodio o al terminar un paso. Podemos modificar aspectos del entrenamiento como cambiar la salida por pantalla o añadir **más métricas** al entrenamiento para obtener más información acerca de este. Estas métyicas se pueden visualizar luego gráficamente usando la herramienta Tensorboard de TensorFlow. La documentación de RLLIB presenta un ejemplo del que se tomó la información para crear los nuestros [2].

Para crear los *callbacks* personalizados es necesario crear una clase Python que herede la clase *DefaultCallbacks* del módulo *ray.rllib.agents.callbacks* y luego pasar la clase a la configuración del agente. Esta clase tiene una serie de métodos que se deben implementar:

```
class MyCallBacks(DefaultCallbacks):
    def __init__(self, **kwargs):
        ...

    def on_episode_start(self, worker: RolloutWorker, base_env: BaseEnv,
                         policies: Dict[str, Policy],
                         episode: MultiAgentEpisode, **kwargs):
        ...

    def on_episode_step(self, worker: RolloutWorker, base_env: BaseEnv,
                        episode: MultiAgentEpisode, **kwargs):
        ...


    def on_episode_end(self, worker: RolloutWorker, base_env: BaseEnv,
                       policies: Dict[str, Policy], episode: MultiAgentEpisode,
                       **kwargs):
        ...

    def on_sample_end(self, worker: RolloutWorker, samples: SampleBatch,
                      **kwargs):
       ...

    def on_train_result(self, trainer, result: dict, **kwargs):
       ...


    def on_postprocess_trajectory(
            self, worker: RolloutWorker, episode: MultiAgentEpisode,
            agent_id: str, policy_id: str, policies: Dict[str, Policy],
            postprocessed_batch: SampleBatch,
            original_batches: Dict[str, SampleBatch], **kwargs):

                ...
```

Los que nos son de más utilidad son *on_episode_start, on_episode_end, on_episode_step*, que se lanzan al iniciar un episodio, al terminarlo y tras cada paso respectivamente. Para crear métricas nuevas, se utiliza un atributo del objeto *episode* de tipo *MultiAgentEpisode* que se denomina *user_data*. Se trata de un diccionario en el que podemos añadir nuevas entradas con valores que queramos medir. Una vez añadidas estas métricas, Ray hará un cálculo automático de su valor mínimo, medio y máximo que se mostrarán en 3 gráficas diferentes en Tensorboard. En esta implementación se vio interesante guardar la media de FPS por episodio y por batch (bloque de pasos determinado), el número de errores por episodio y *batch* y el porcentaje de errores por episodio y por batch. Consideramos como error aquel paso cuya acción da un resultado de FPS por encima o por debajo del rango objetivo.

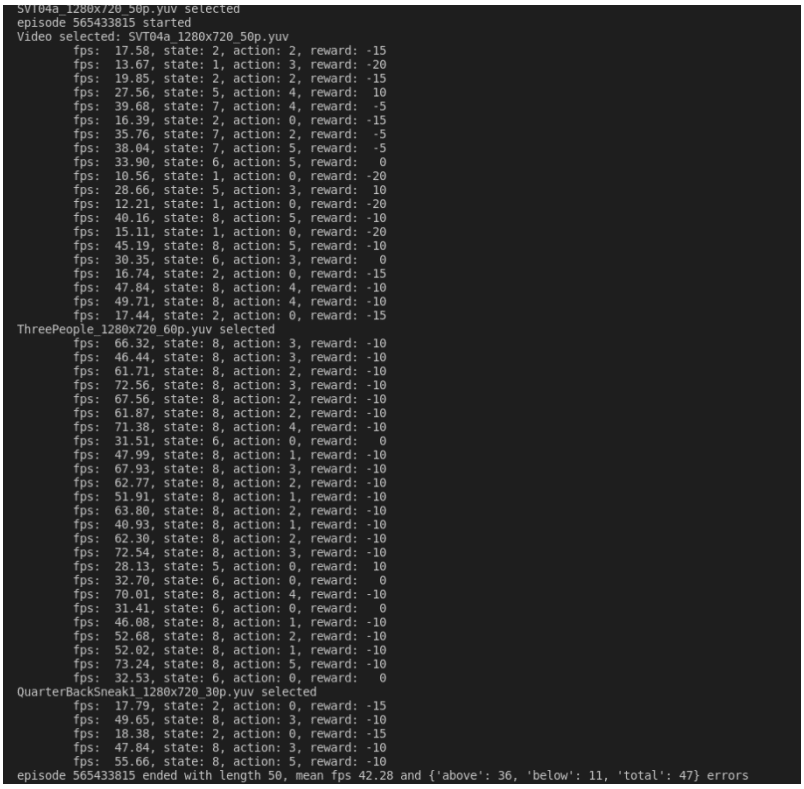


Figura 3-3: Callbacks personalizados. Gracias a ellos se puede modificar la salida por pantalla del entrenamiento.

En la figura anterior se ve un ejemplo de la ejecución de nuestra implementación. Cada línea representa el estado y la recompensa obtenidos tras realizar una acción. Como vemos, cuando un vídeo ha terminado de procesarse, se selecciona uno nuevo.

3.5 Guía de uso del código fuente

3.5.1 Instalación

Antes de utilizar el código fuente en Linux es necesario tener instalados previamente los paquetes *python3*, *python3-venv*, *python3-pip*. Para instalar las dependencias necesarias del proyecto haremos lo siguiente:

- Podemos crear primero un entorno virtual de Python con:

```
$ python3 -m venv --system-site-packages ./venv
$ source venv/bin/activate
```

- Y luego,

```
$ pip install --upgrade pip
$ pip install -r requirements.txt
```

- O directamente utilizar las 2 sentencias anteriores para instalar sin usar entorno virtual.

Para instalar el entorno basta con hacer

```
$ pip install -e kvazaar_gym/
```

3.5.2 Ejecución de scripts

Todos los scripts ejecutables se encuentran en el directorio *src/* del proyecto y **se ejecutan desde la raíz del proyecto**. A continuación, detallaremos la utilización de cada uno de ellos, así como su configuración requerida.

- `train_kvazaar.py [--r]`: este script crea un entrenamiento del agente. Guarda los resultados de este en un directorio llamado *resultados/*. Si se utiliza la opción `--r o ---restore`, se restaura el entrenamiento cuyo nombre se encuentra en el archivo de configuración y se entrena a partir de él.
- `learned_kvazaar.py [-h] -v <video_path> -p <checkpoints_path>`: se crea un agente que recupera una política indicada en el path de checkpoints y ejecuta un episodio sobre el vídeo cuya ruta se indica como parámetro. La ruta de los checkpoints se encuentra en rutas del tipo *resultados/<entrenamiento>/checkpoints*. Además, guarda un archivo con extensión *csv* con el resultado de los FPS y las recompensas obtenidos para cada paso.
- `baseline.py -v <video> (-c <cores> | -r)`: similar al anterior script, pero aquí no se utiliza una política aprendida. Se genera un agente que toma o acciones aleatorias si se usa la opción `-r` o un número de núcleos fijo con la opción `-c`. También se guarda un archivo *csv* como en el caso anterior.
- `plotter.py`: este script es un generador de gráficas con los resultados de los archivos *csv* de los otros 2 scripts anteriores. Usa las librerías *matplotlib* y *pandas*.

El resto de scripts son auxiliares y no se utilizan como ejecutables:

- `custom_callbacks.py`: módulo que crea la clase para los callbacks personalizados.
- `common_tasks.py`: módulo auxiliar que encapsula tareas usadas en otros módulos.

3.5.3 El archivo de configuración

Existe un archivo de configuración situado en *src/config.ini* necesario para la ejecución de los scripts ejecutables (salvo *plotter.py)*. En él se guarda la configuración de distintos parámetros y se divide en 2 secciones:

- Sección *common:* sección común que se usa en *trained_kvazaar.py, learned_kvazaar.py, baseline.py:*
  - *rewards:* ruta del archivo de recompensas
  - *kvazaar*: ruta del ejecutable de Kvazaar
  - *cores*: par de enteros separados por comas que indica el conjunto de cores para Kvazaar.
- Sección *train*: únicamente se utiliza para el entrenamiento y consta de:
  - *batch*: número de pasos que se van a realizar antes de actualizar la política.
  - *mini_batch*: número de pasos para el descenso de gradiente. Debe ser menor que *batch* y preferiblemente una potencia de 2 divisora de este.
  - *videos*: ruta del directorio de vídeos de entrenamiento.
  - *mode*: modo de selección de vídeos para el entrenamiento. Se puede escoger entre *random* y *rotating*, para selección aleatoria de vídeos o rotatoria respectivamente.
  - *iters*: número de iteraciones del entrenamiento. Una iteración corresponde a una vuelta de *batch* pasos.

- *name*: nombre del entrenamiento. Este nombre es el que se usa para crear el subdirectorio de resultados del entrenamiento.Resultados

Capítulo 4 - Resultados

4.1 Metodología de pruebas

Nuestro entorno de pruebas ha sido el servidor Esfinge ofrecido por la Facultad de Informática. Sus especificaciones son las siguientes:

- Procesador Intel Xeon E5-2670
- Frecuencia 2.6 GHz
- Número de cores: 16, de los cuales 8 se destinaron a la ejecución de Kvazaar, y los otros 8 para el resto de los procesos del proyecto.
- RAM: 64 GB

El entrenamiento final de los que se obtuvieron estos resultados tuvo como configuración la siguiente:

- 130000 pasos de entrenamiento
- 1024 pasos de bloque *batch*
- 32 pasos para el descenso de gradiente, que es un valor estándar en aprendizaje por refuerzo [6]
- Sets de estados y recompensas definidos en el apartado 3.2.3
- Selección de vídeos aleatoria
- Set de vídeos con diferentes dificultades de procesamiento, para crear un entorno lo más realista y diverso posible, teniendo en cuenta las capacidades de nuestro entorno de pruebas. Se han utilizado un total de 11 vídeos diferentes, todos de resolución 1280x720 píxeles con diferentes tasas de fotogramas por segundo oscilando entre 20 y 60. De estos vídeos, 6 han utlizaron para el conjunto de entrenamiento, y 5 para para el conjunto de testeo.

4.2 Análisis de las métricas del entrenamiento

Gracias a la herramienta Tensorboard[3] de TensorFlow y a los callbacks personalizados, podemos ver la información de las métricas recopiladas de manera gráfica a través de un navegador. Así podemos intuir el comportamiento del agente y observar si nuestro diseño es el correcto.

En primer lugar, veamos los resultados obtenidos en el entrenamiento para las métricas personalizadas. En las siguientes imágenes veremos gráficas correspondientes a cada una de ellas y que son de interés para comprobar si el entrenamiento ha sido exitoso. Estas gráficas muestran el valor de estas métricas en un momento dado del entrenamiento y se dividen en grupos de tres: cada métrica presenta un valor mínimo (min), medio (mean) y máximo (max), con sus correspondientes gráficas. Las gráficas han sido obtenidas a través de Tensorboard.

Métricas para los FPS

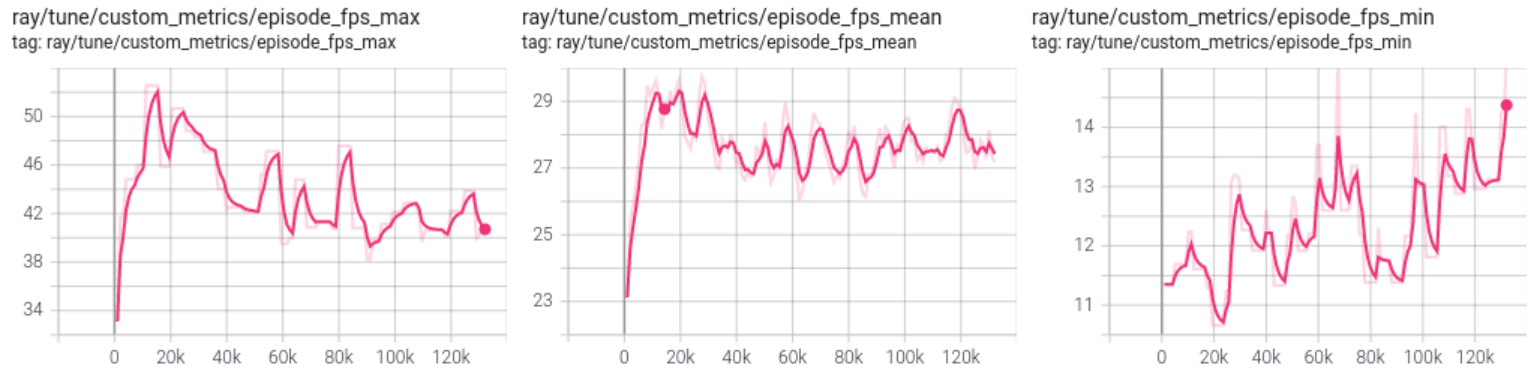


Figura 4-1: Evolución de los FPS medios por episodio.

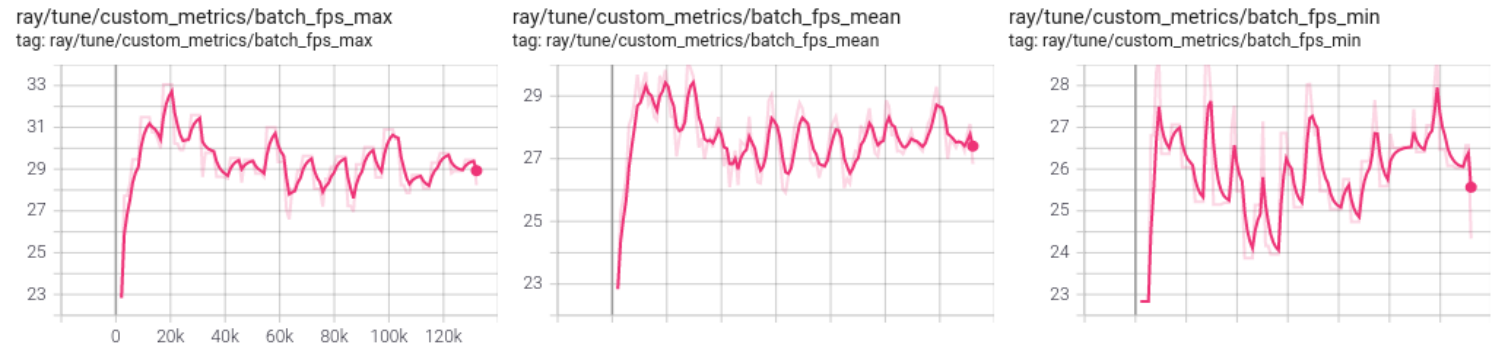


Figura 4-2: Evolución de los FPS medios por batch.

3 https://www.tensorflow.org/tensorboard?hl=es-419

En primer lugar, tenemos la evolución de los FPS medios durante el entrenamiento (Figura 4-1 y Figura 4-2). En el eje horizontal tenemos el número de pasos del agente y en el eje vertical tenemos el valor de FPS obtenidos.

La gráfica *episode_fps_mean* muestra las medias hasta el momento de los valores medios de FPS conseguidos por episodio. Es decir, cada 50 pasos (un episodio), se guarda una media de FPS de esos 50 pasos, y cuando se llega a una iteración completa (un batch o 1024 pasos) se realiza una media de todas esas medias y es el valor que vemos en las gráficas. La gráfica *episode_fps_max* muestra por tanto la media máxima de FPS en un episodio conseguida hasta el momento. La gráfica *episode_fps_min*, de manera análoga, muestra la media de FPS en un episodio mínima hasta el momento.

Por otro lado, la gráfica *batch_fps_mean* representa las medias de FPS en un batch completo. En realidad, esta es prácticamente la misma que la de *episode_fps_mean* ya que, al final, hacer unas medias del valor medio de FPS en unos episodios y una media de un bloque entero es redundante. Sin embargo, el uso de esta métrica se vuelve interesante en *batch_fps_max* y *batch_fps_min*, ya que estas muestran las medias de FPS máximas y mínimas hasta el momento en un *batch*. Con estas dos últimas, se puede visualizar más a largo plazo la tendencia del agente, puesto que sus contrapartes de episodio reflejan mucho más las diferencias de los vídeos. Al tener, por ejemplo, una media máxima de FPS en un *batch* completo, se suavizan aquellas medias de episodios que son muy bajas o altas con respecto al objetivo.

Como vemos, en líneas generales las medias de FPS (gráficas mean) se mantienen en el rango objetivo (20 a 30). Si nos fijamos en las gráficas *batch_fps_max* y *batch_fps_min,* que muestran de mejor manera la tendencia del agente, vemos que estas se van lentamente aproximando al rango objetivo con el paso del tiempo, los máximos disminuyendo hacia él, y los mínimos aumentando. Sus contrapartes de episodio (*episode_fps_max, episode_fps_min)* muestran cambios más bruscos, pero también se observa el mismo patrón de acercamiento al objetivo.

En definitiva, podemos decir que el agente, durante el entrenamiento, mantiene un comportamiento positivo con respecto a nuestro objetivo: a largo

plazo buscamos que nuestro agente se mantenga el máximo tiempo posible en el objetivo, por tanto, el acercamiento de los mínimos y máximos hacia él resulta un comportamiento lógica ante este objetivo.

Métricas para los errores

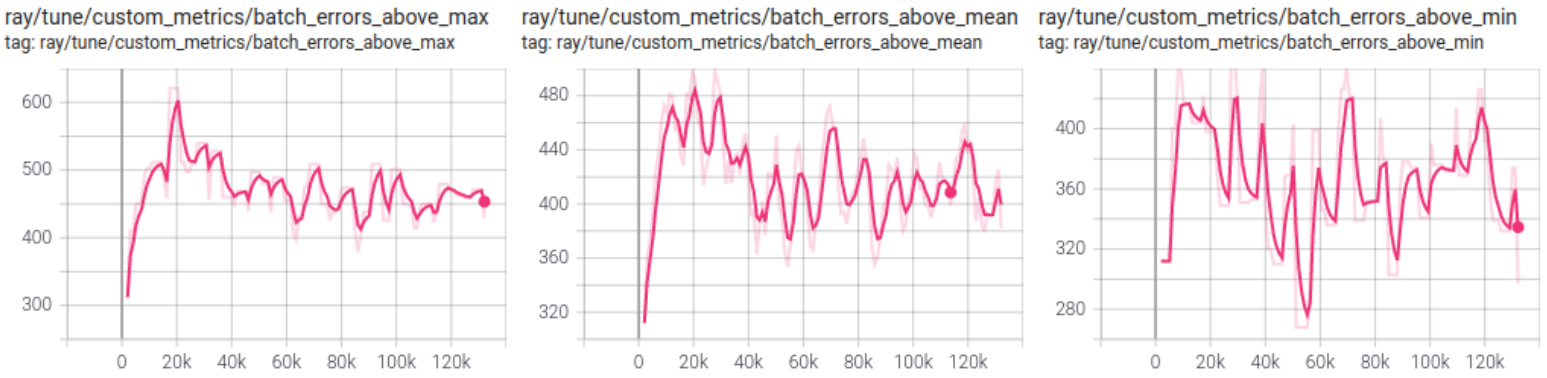


Figura 4-3: Errores por encima del objetivo por batch.

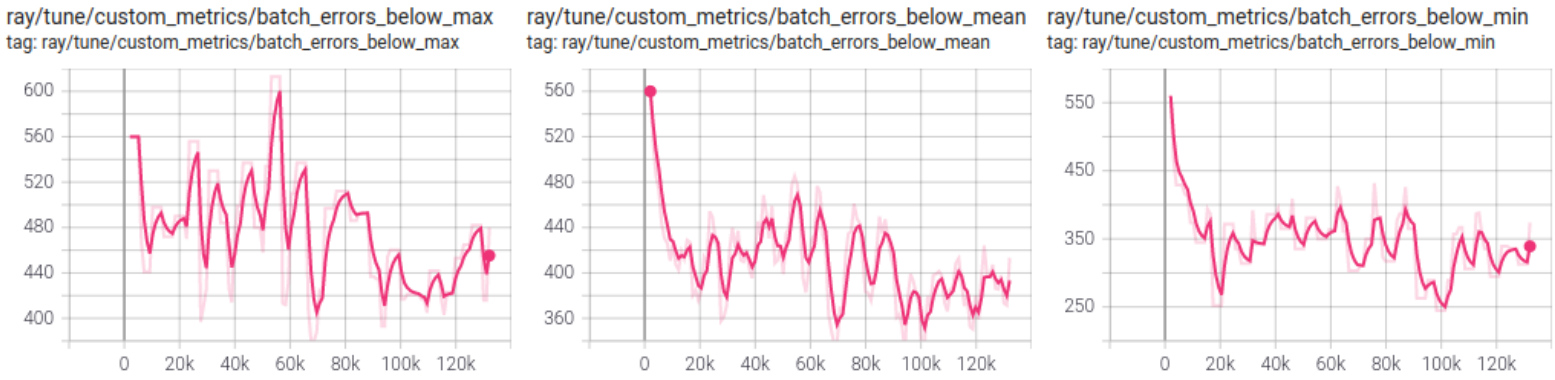


Figura 4-5: Errores por debajo del objetivo por batch.

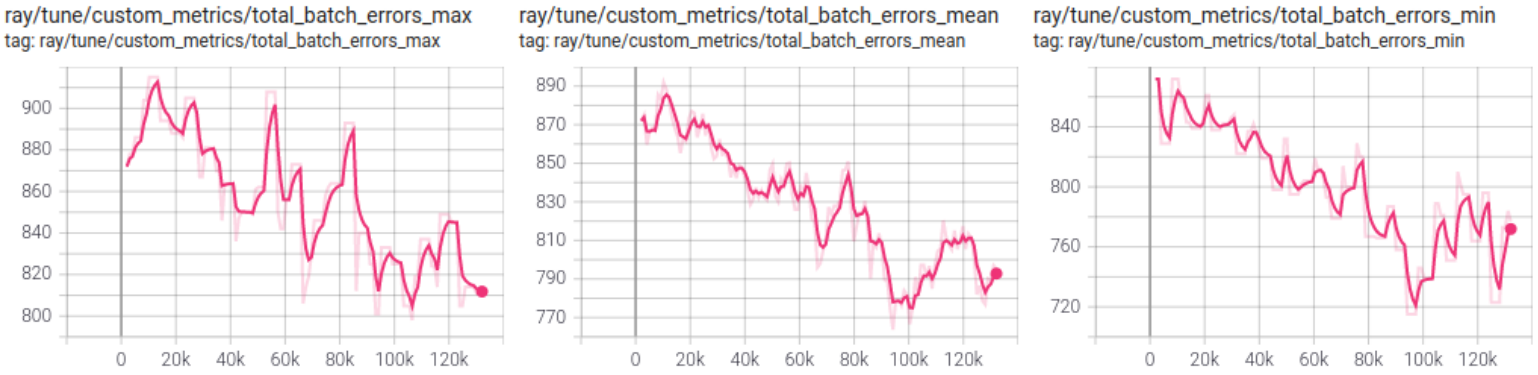


Figura 4-4: Errores totales por batch.

Figura 4-6: Ratio (en %) de errores por batch.

Los errores por *batch* se recopilan en cada iteración (*batch* número de pasos o 1024 pasos) y muestran la cantidad de veces que el agente está por encima (Figura 4-3) o por debajo (Figura 4-5) del rango objetivo. La Figura 4-4 y la Figura 4-6 muestran la suma de estos errores y la evolución del porcentaje de estos en un *batch*, respectivamente. Estas gráficas muestran la tendencia a largo plazo del agente en cuanto a las veces que no está en el objetivo. Como vemos, en general, la media de la suma de errores (Figura 4-4) y el porcentaje de estos van disminuyendo con el tiempo, justo lo que queremos. A medida que el agente va aprendiendo, buscamos que se mantenga en el objetivo el mayor tiempo posible y si esta métrica va disminuyendo significa que el agente está tomando buenas decisiones.

Si nos fijamos en los errores por encima del objetivo (Figura 4-3), el agente no parece disminuirlos en gran medida, en comparación con los errores por debajo (Figura 4-5) que sufren una drástica bajada al principio del entrenamiento. Este comportamiento es destacable, ya que al definir nuestras recompensas en el punto 3.2.3, nos interesaba que los valores por encima del objetivo predominaran sobre los valores por debajo, y esto se refleja aquí : el agente se apresura en reducir los errores por debajo puesto que los valores de FPS asociados a estos errores dan las peores recompensas.

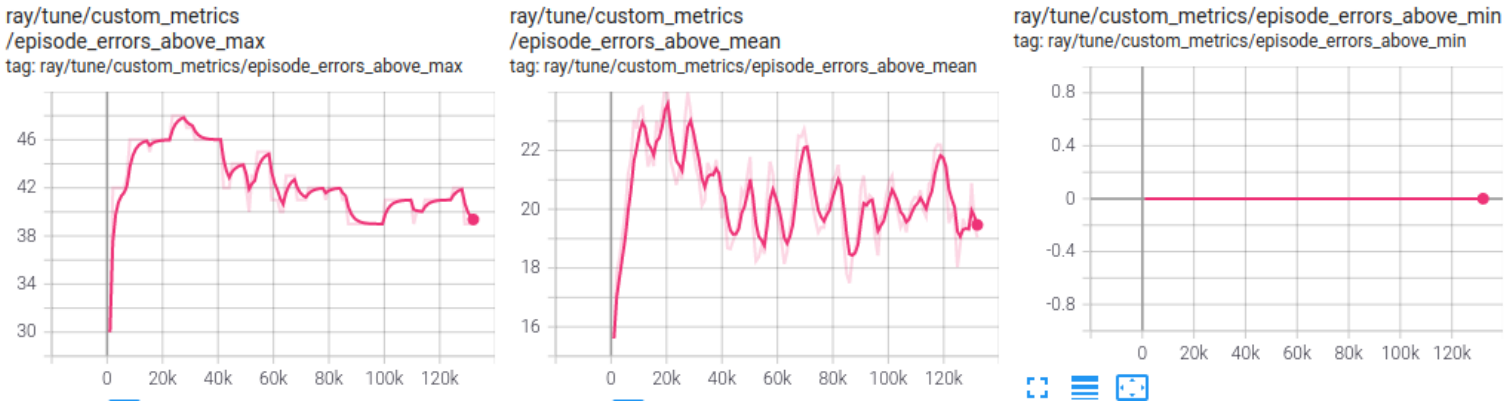


Figura 4-8: Errores por encima del objetivo por episodio.

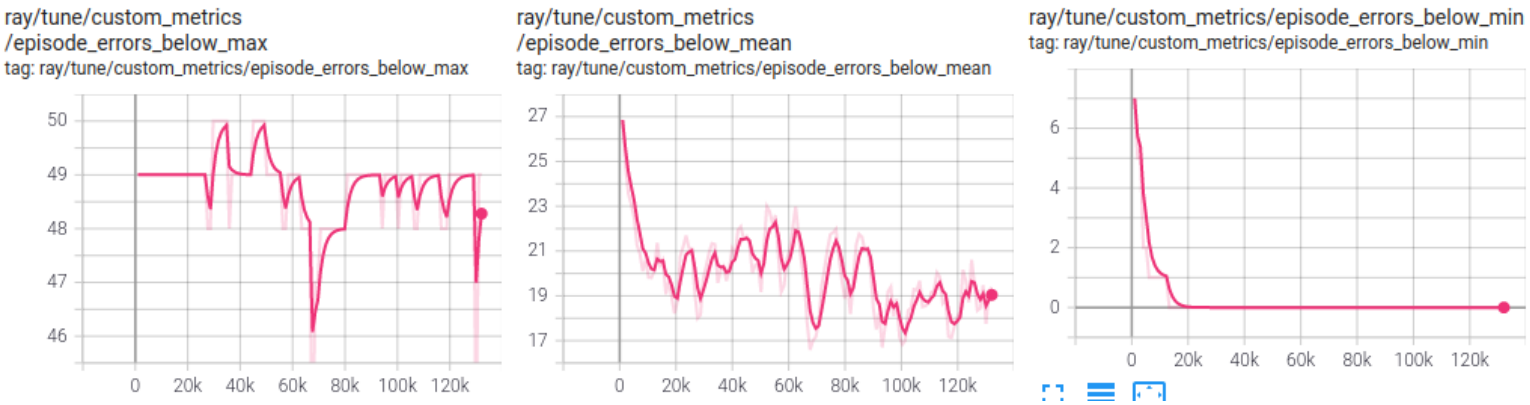


Figura 4-7: Errores por debajo del objetivo por episodio.

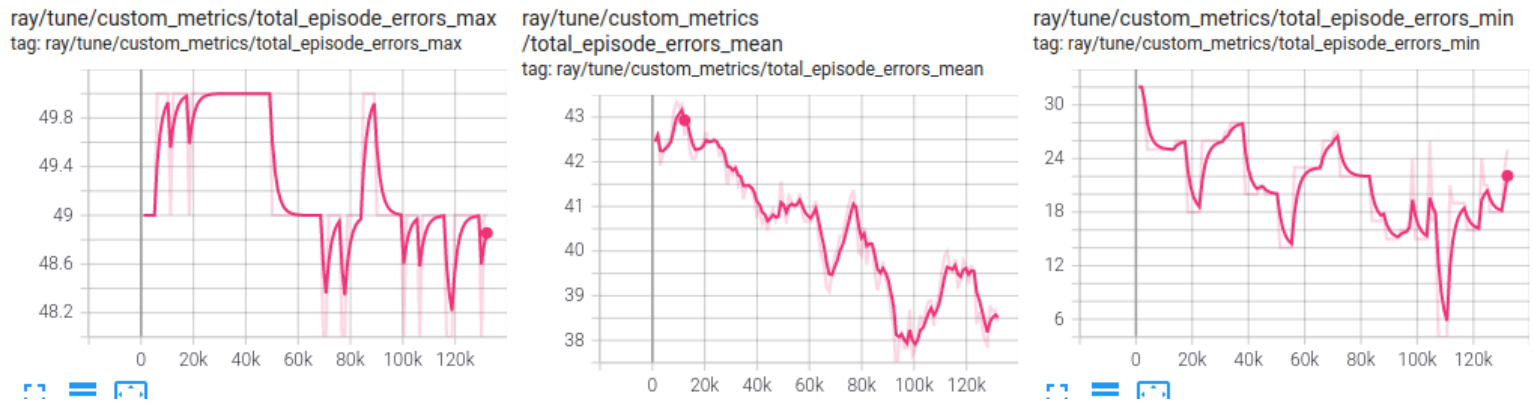


Figura 4-9: Errores totales por episodio.

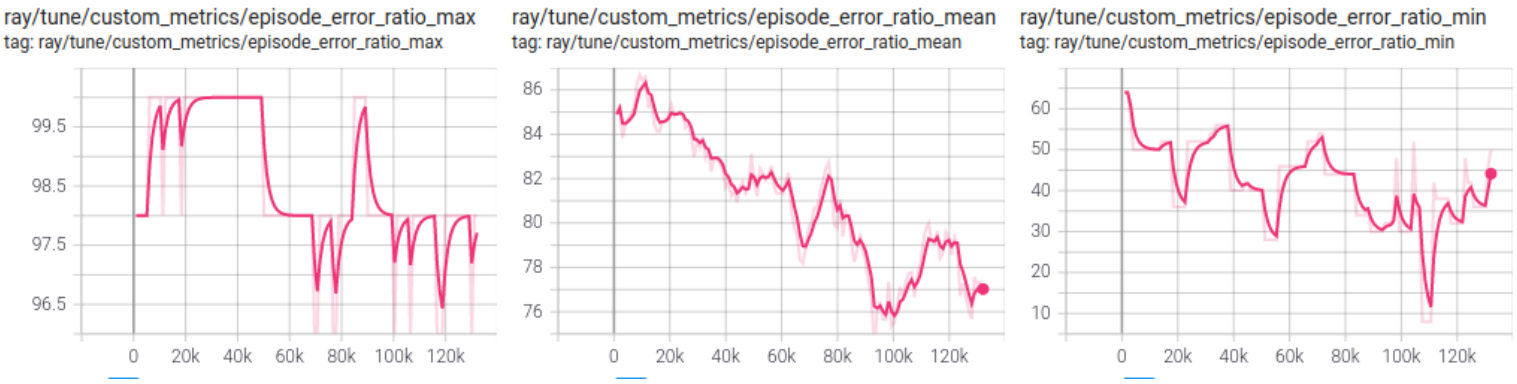


Figura 4-10: Ratio (en %) de errores por episodio.

En cuanto a los errores por episodio (Figura 4-8 a Figura 4-10) tenemos un comportamiento similar a las gráficas para *batch*. Estas gráficas se construyen a partir de los errores obtenidos en cada episodio (50 pasos). Como vemos tanto el ratio de errores (Figura 4-9) como los errores totales (Figura 4-10) tienden a disminuir con el paso del tiempo. Este comportamiento es correcto en un entrenamiento exitoso por lo explicado anteriormente en los errores para *batch*. En estas gráficas vemos también más fuertemente los extremos del entrenamiento, como por ejemplo en las gráficas de errores por encima y por debajo del objetivo (Figura 4-8 y Figura 4-7, respectivamente), cuyos mínimos son 0. Esto puede ocurrir ya que puede haber algún episodio en el que no hubo errores. En las gráficas para *batch*, al englobar muchos episodios a la vez, este hecho se pierde.

Métricas para la política

Entre las métricas de la evolución de la política (Figura 4-11) nos interesan las de la entropía y las de las funciones de pérdida, tanto la función de pérdida para la política (*policy_loss)*, la pérdida total (*total_loss)* y la pérdida de la función de valor (*vf_loss)*.

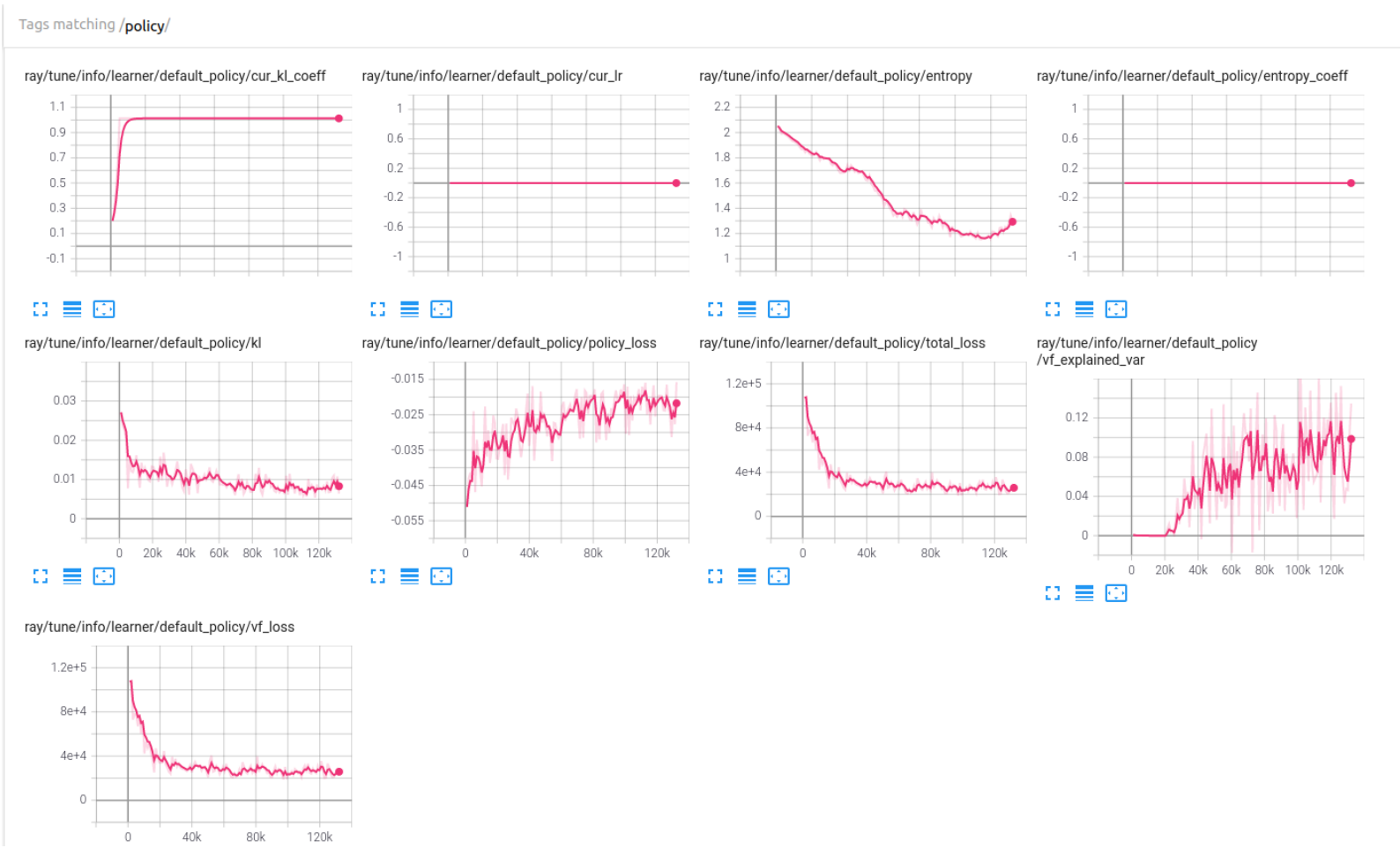


Figura 4-11: Métricas generales para la política.

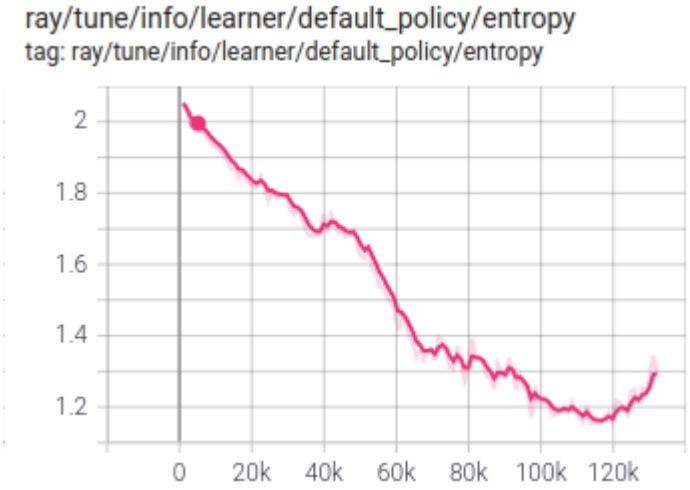


Figura 4-12: Evolución de entropía de la política.

La entropía (Figura 4-12) mide la aleatoriedad de las decisiones del agente, es decir, qué tan seguro está de las decisiones que toma [7]. En un entrenamiento exitoso esta métrica debe disminuir de manera progresiva y sin saltos bruscos a lo largo del tiempo y es justo lo que ocurre en nuestro caso. En los últimos pasos del entrenamiento vemos una tendencia al alza que quiere decir que el agente empieza a tomar ciertas decisiones más aleatorias para provocar una mayor exploración en el entrenamiento, como vimos en la introducción de PPO (2.2.1).

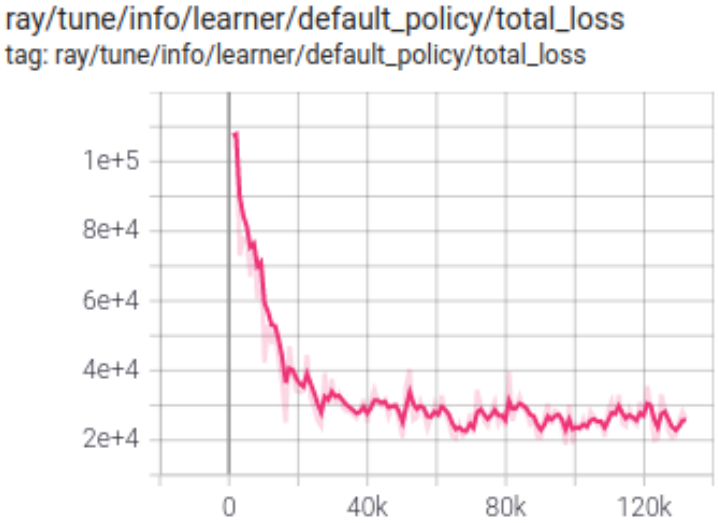


Figura 4-13: Evolución de la función de pérdida total.

La función de pérdida total (Figura 4-13) que se corresponde al objetivo de PPO que introdujimos en el punto 2.2.1 debe disminuir a lo largo de tiempo, puesto que el objetivo de PPO es minimizar la pérdida de la política.

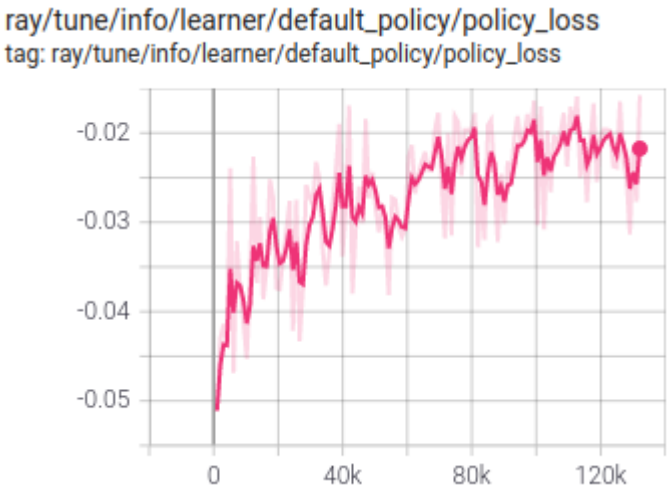


Figura 4-14: Evolución de la pérdida de la política.

La función de pérdida (*policy_loss)* suele fluctuar mucho en los entrenamientos, pero siempre por debajo del valor 1 [7].

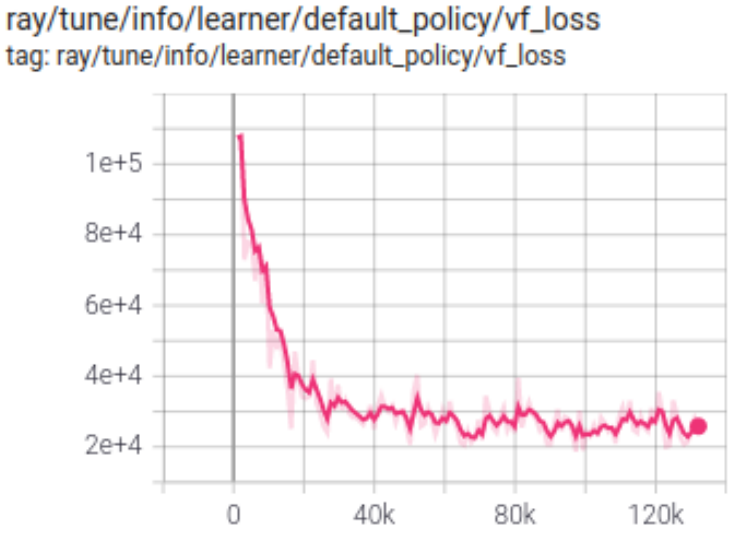


Figura 4-15: Evolución de la función de valor.

La función de valor, que indica cómo es capaz de predecir el agente el valor de cada estado debe disminuir a medida que las recompensas se estabilizan [7] como ocurre en la Figura 4-15.

Métrica para las recompensas

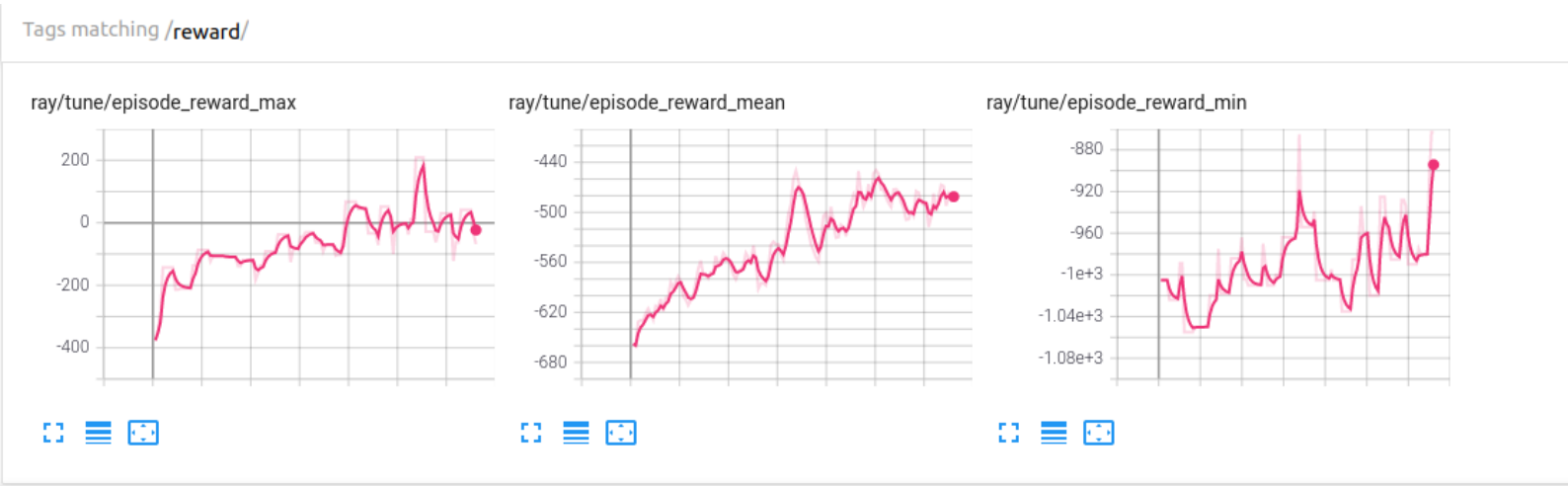


Figura 4-16: Evolución de las recompensas.

Por último, tenemos las gráficas más importantes (Figura 4-16), las recompensas. Estas muestran la evolución de la recompensa acumulada por episodio a lo largo del entrenamiento. Cada iteración o *batch* número de pasos, se recogen las recompensas acumuladas obtenidas en cada episodio y se realiza una media de ellas, valor que se muestra en *episode_reward_mean*. Las gráficas *episode_reward_max* y *episode_reward_min* muestran los valores de recompensas acumuladas máximos y mínimos conseguidos hasta un determinado momento, respectivamente. En un entrenamiento exitoso, estas tres gráficas deben aumentar gradualmente, puesto que este es el objetivo del paradigma, maximizar la recompensa. Como vemos, las gráficas crecen gradualmente a lo largo del tiempo, lo que nos indica que el agente está tomando decisiones correctas.

Conclusiones acerca de las métricas de entrenamiento

Dadas las métricas expuestas anteriormente, el agente siguió una estrategia a priori correcta: los FPS se mantienen en el rango objetivo lo máximo posible, los errores disminuyen con el paso del tiempo, y la política se comporta de manera exitosa. Viendo los resultados mostrados a continuación podremos saber a ciencia cierta si el comportamiento del agente es el esperado.

4.3 Evaluación de resultados

Para evaluar los resultados obtenidos, tenemos en cuenta tres métricas: el número de FPS arrojados por Kvazaar en un bloque, la recompensa acumulada a lo largo de la codificación, y el uso de CPUS. Si el agente toma decisiones correctas los FPS deben mantenerse en el rango objetivo el máximo número de pasos posible y el uso de CPUS debe poder justificarse. Para poder comparar los resultados obtenidos por el agente entrenado, se decidió utilizar unos resultados base o *baselines* usando un número fijo de núcleos con 1, 4 y 8 núcleos para Kvazaar así como una versión usando acciones totalmente aleatorias. Veremos a continuación los resultados obtenidos para estas métricas sobre diferentes vídeos utilizados en nuestra implementación.

QuarterBackSneak1

Veamos primero cómo es el comportamiento de los diferentes escenarios para un vídeo presente en el entrenamiento, QuarterBackSneak1. Este vídeo presenta un nivel de dificultad elevado de procesamiento ya que, cómo se puede observar en la línea de 8 núcleos (Figura 4-6), incluso utilizando el máximo número de núcleos posibles, los FPS nunca suben de 30, como sí ocurre en para otros vídeos que veremos posteriormente.

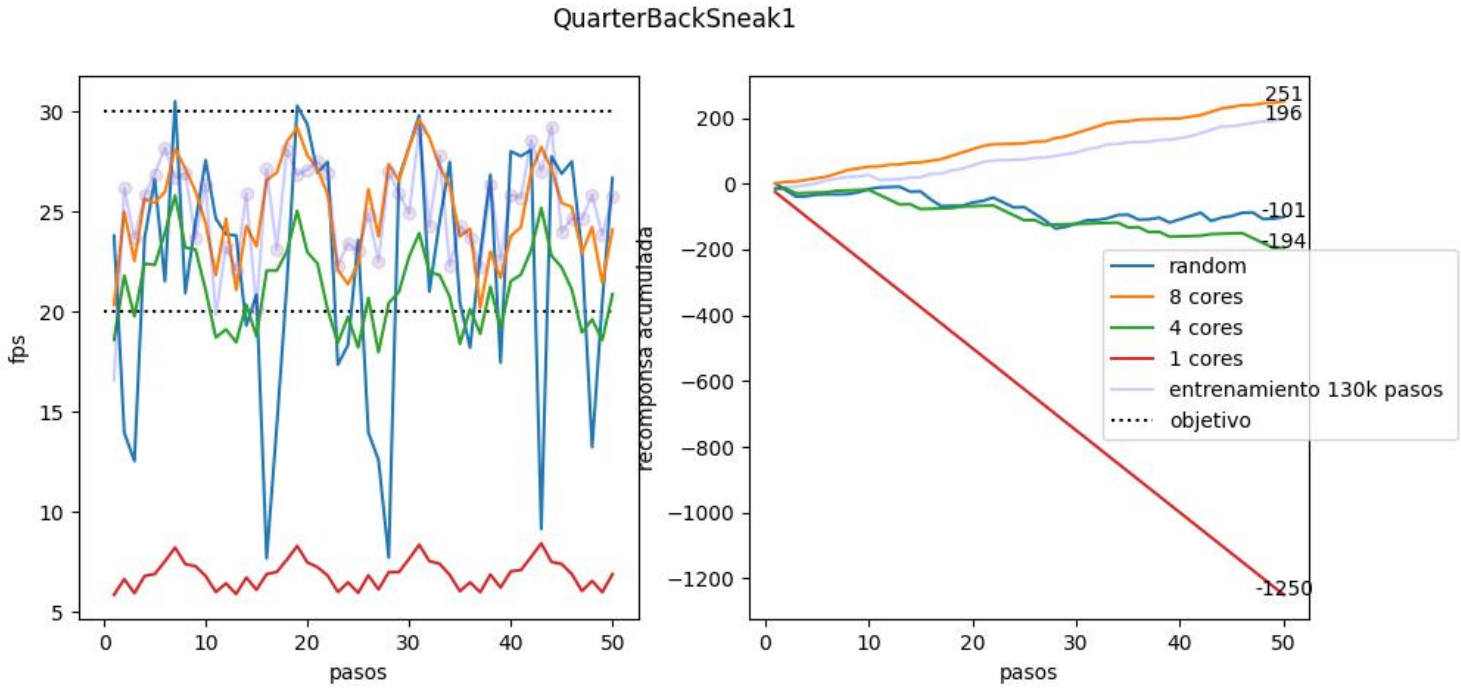


Figura 4-17: Resultados base para QuarterBackSneak1.

Como vemos (Figura 4-17), el baseline para 8 núcleos es el que arroja mejor recompensa acumulada, muy seguido por nuestra política, y como se puede observar en la gráfica de FPS siempre se mantiene en la franja objetivo.

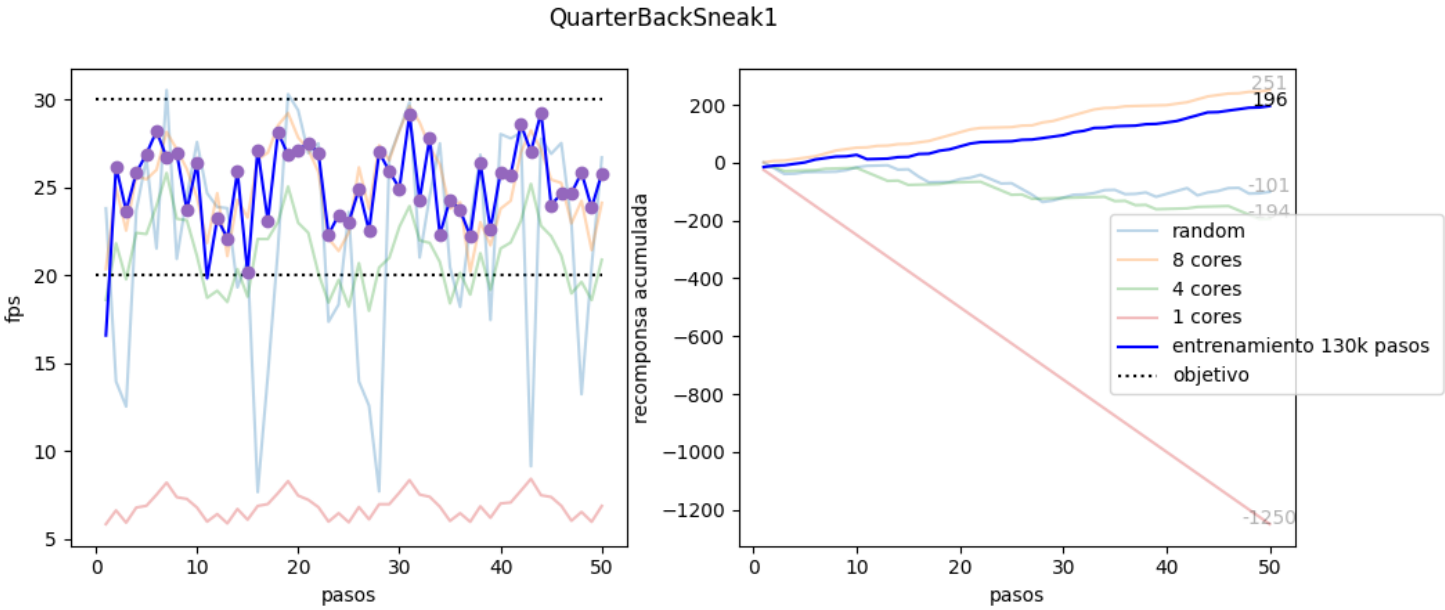


Figura 4-18: Resultados de la política aprendida para QuarterBackSneak1.

Si nos centramos en la gráfica del entrenamiento (Figura 4-18), vemos que los FPS son ideales, estamos prácticamente siempre en el objetivo y además las recompensas son muy similares al baseline de 8 núcleos.

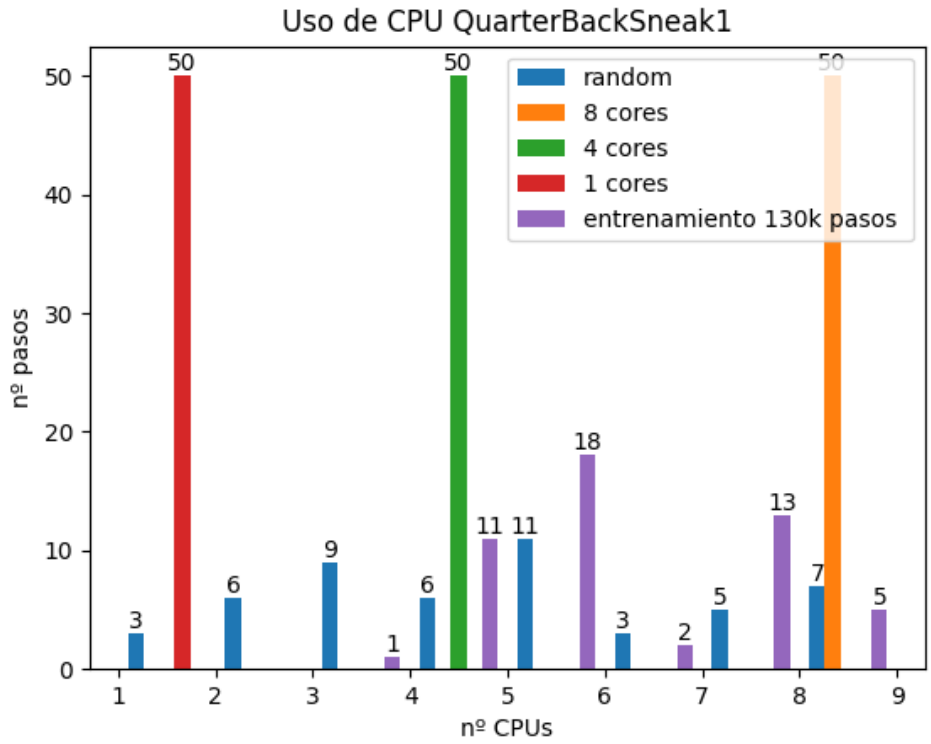


Figura 4-19: Uso de CPUS para QuarterBackSneak1

El anterior histograma (Figura 4-19) que representa el uso de CPUS se entiende de la siguiente manera. Cada intervalo del eje horizontal representa las posibles acciones, el intervalo (1,2) representa la acción de 1 núcleo, el (2,3), la acción 2 núcleos, así sucesivamente. El eje vertical representa la cantidad de pasos en los que se ha tomado esa acción. Vemos como el agente entrenado utiliza muchos menos núcleos que la versión de 8 núcleos fijos. Esto supone un mejor resultado puesto que el agente entrenado es capaz de obtener un resultado similar, además de ser el mejor posible, utilizando muchos menos núcleos, justificando su uso de núcleos aparentemente más errático. Vemos como el comportamiento de las gráficas del entrenamiento (punto 4.2) queda reflejado en la evaluación: el entrenamiento es exitoso.

E_FourPeople

E_FourPeople se encuentra en el conjunto de vídeos de testeo que no se usan para el entrenamiento. Viendo los resultados de este vídeo y los posteriores entenderemos cómo reacciona el agente entrenado ante situaciones nuevas.

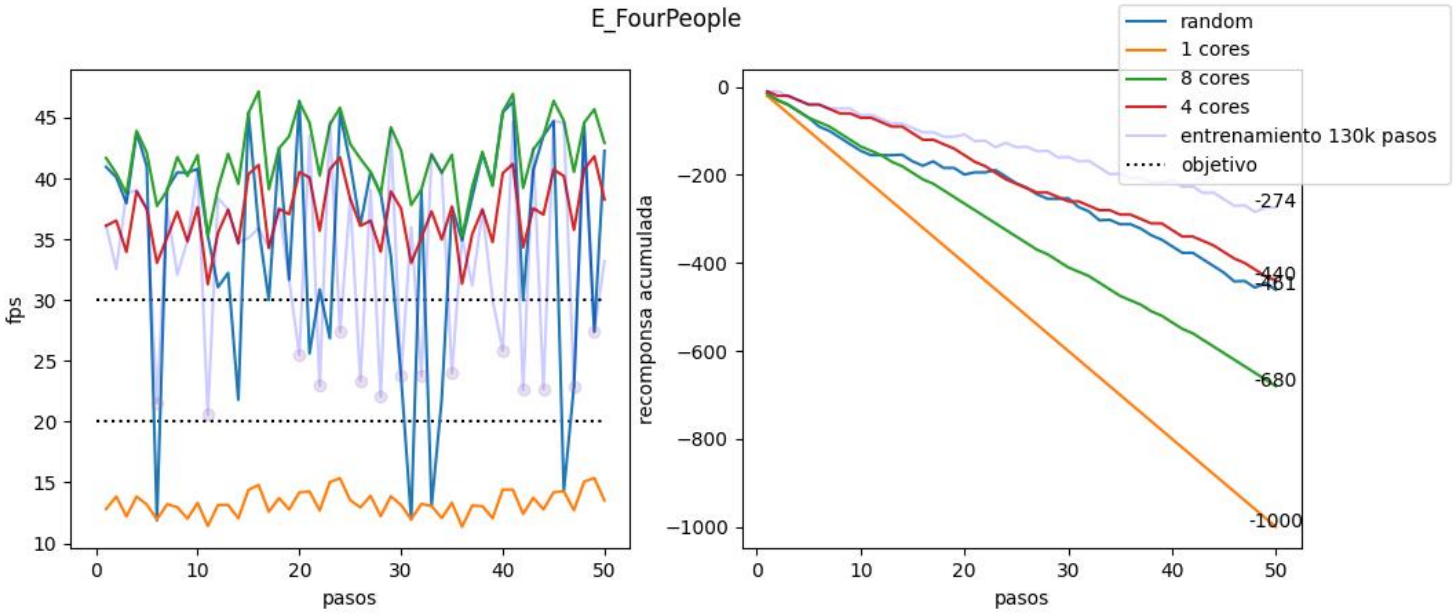


Figura 4-20: Resultados base para E_FourPeople.

Viendo los resultados base para este vídeo (Figura 4-20), y comparándolos con los anteriores vídeos, podemos observar que este vídeo tiene una dificultad menor de procesamiento puesto que Kvazaar sí que es capaz ahora de superar los 30 FPS para el mismo número de hilos usado anteriormente. En este caso, ninguno de los resultados base es bueno teniendo en cuenta nuestro objetivo, o están por debajo del objetivo o por encima.

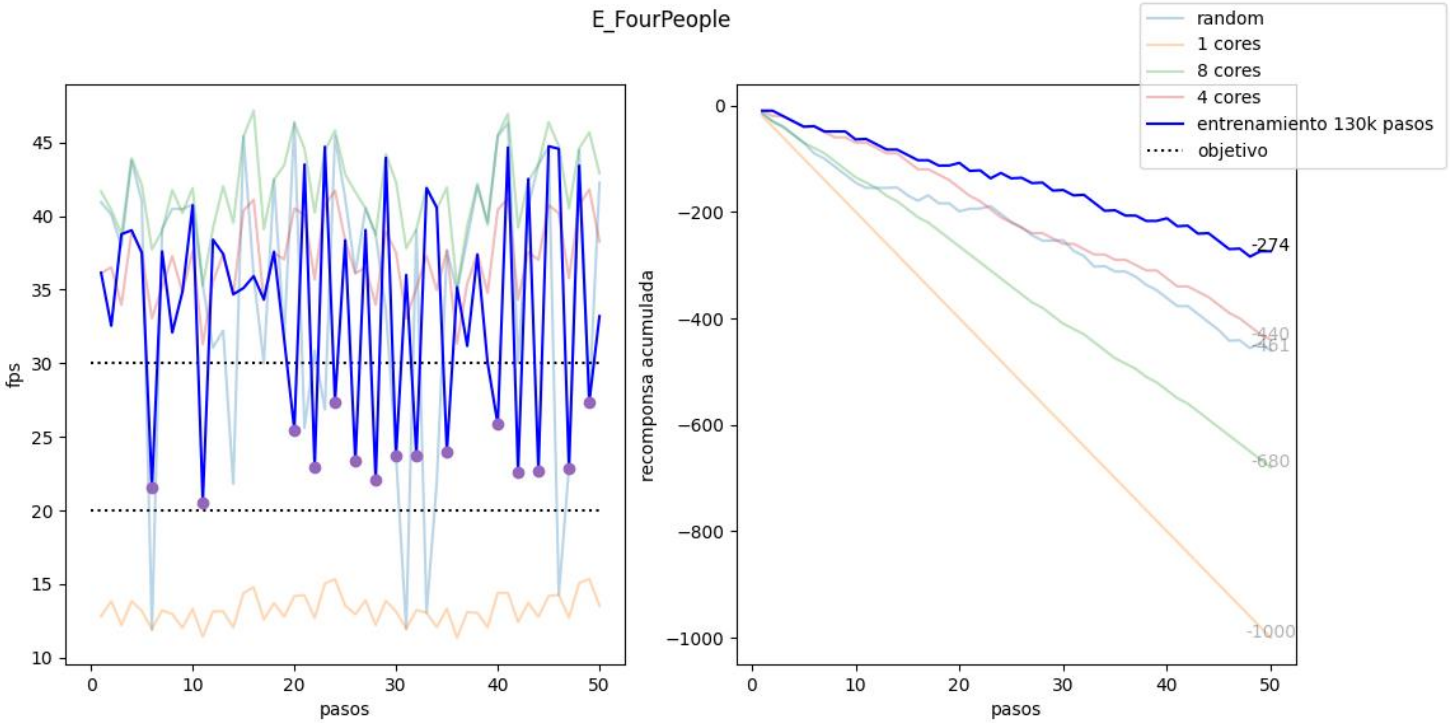


Figura 4-21: Resultados de la política aprendida para E_FourPeople

Pero si vemos el resultado para el entrenamiento (Figura 4-21) vemos que el agente es capaz de tomar decisiones que arrojan valores dentro del objetivo. Es cierto que existe mucha fluctuación de FPS, pero el agente es capaz de rectificar y volver a un valor óptimo. Además, la función de recompensa acumulada es la mejor de todas.

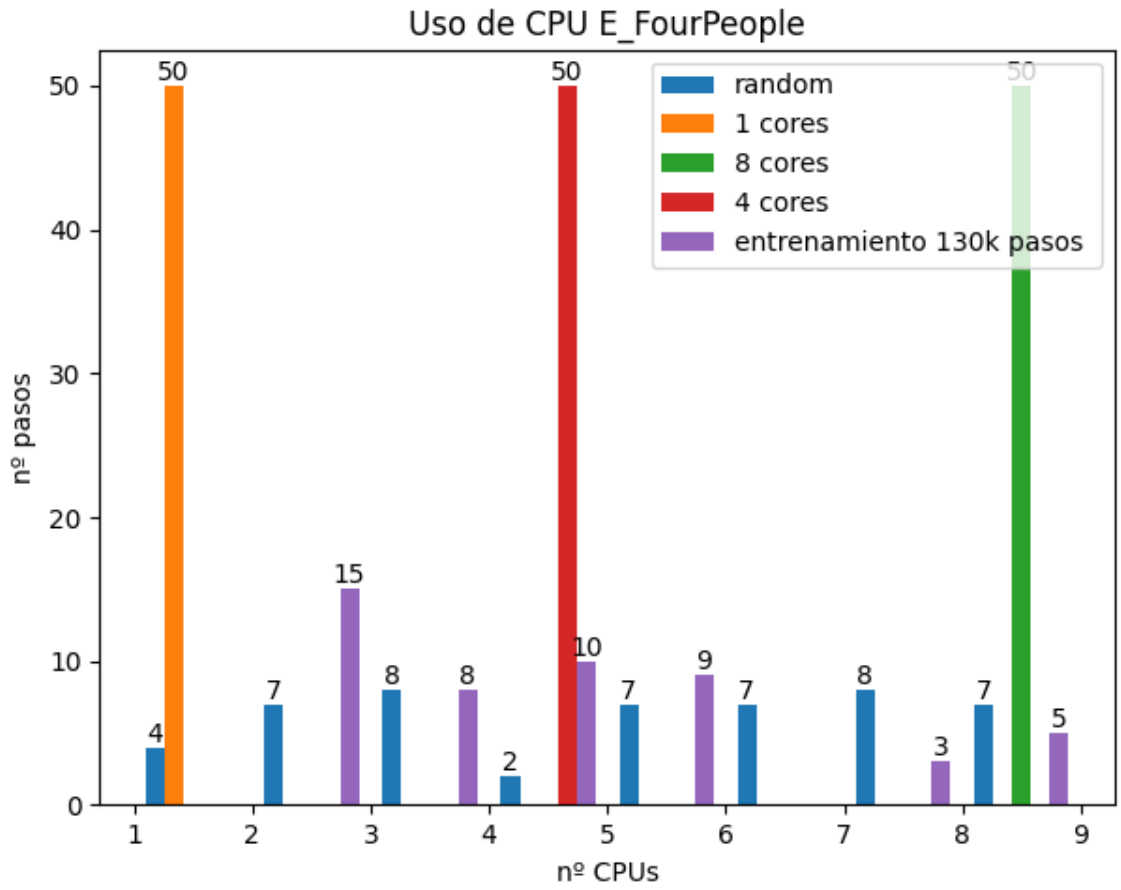


Figura 4-22: Uso de CPUS para E_FourPeople

En cuanto al uso de CPUS (Figura 4-22), no se ve exactamente una decisión preferida por parte del agente, pero, en cualquier caso, no se están utilizando los valores más altos (8 y 9 núcleos) muchas veces, lo cual es siempre positivo.

OldTownCross

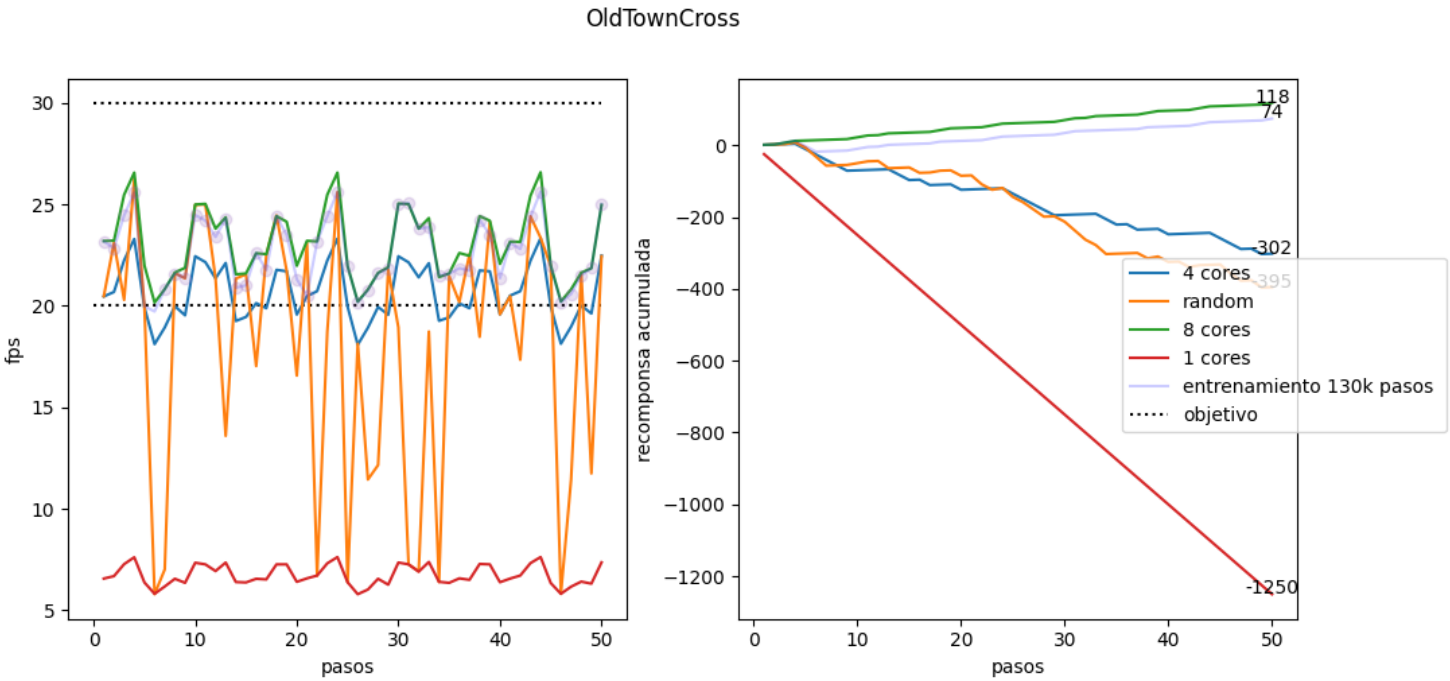


Figura 4-23: Resultados base para OldTownCross.

Este vídeo presente en el conjunto de testeo es incluso más difícil de procesar que QuarterBackSneak1, ya que, para 8 núcleos, tanto los FPS (Figura 4-23) como la mejor recompensa acumulada son menores que en dicho vídeo

Sin embargo, centrándonos en la política aprendida (Figura 4-25), esta se comporta realmente bien para este vídeo. Los FPS se mantiene prácticamente siempre en el objetivo y la recompensa acumulada es positiva y muy similar a la de 8 núcleos.

De forma similar a QuarterBackSneak1, el uso de CPUS (Figura 4-24) es sustancialmente mejor para el agente entrenado, ya que se consigue un rendimiento óptimo usando menos núcleos que en el caso de 8 núcleos.

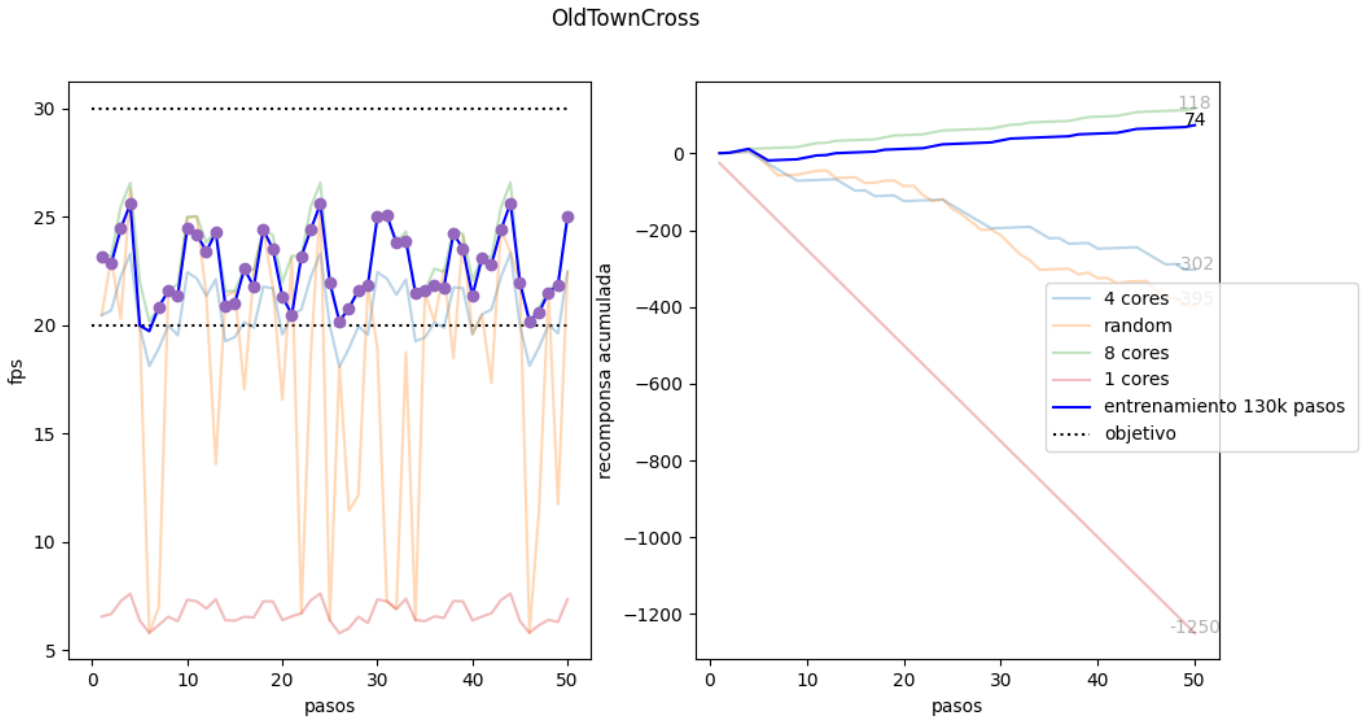


Figura 4-25: Resultados de la política entrenada para OldTownCross.

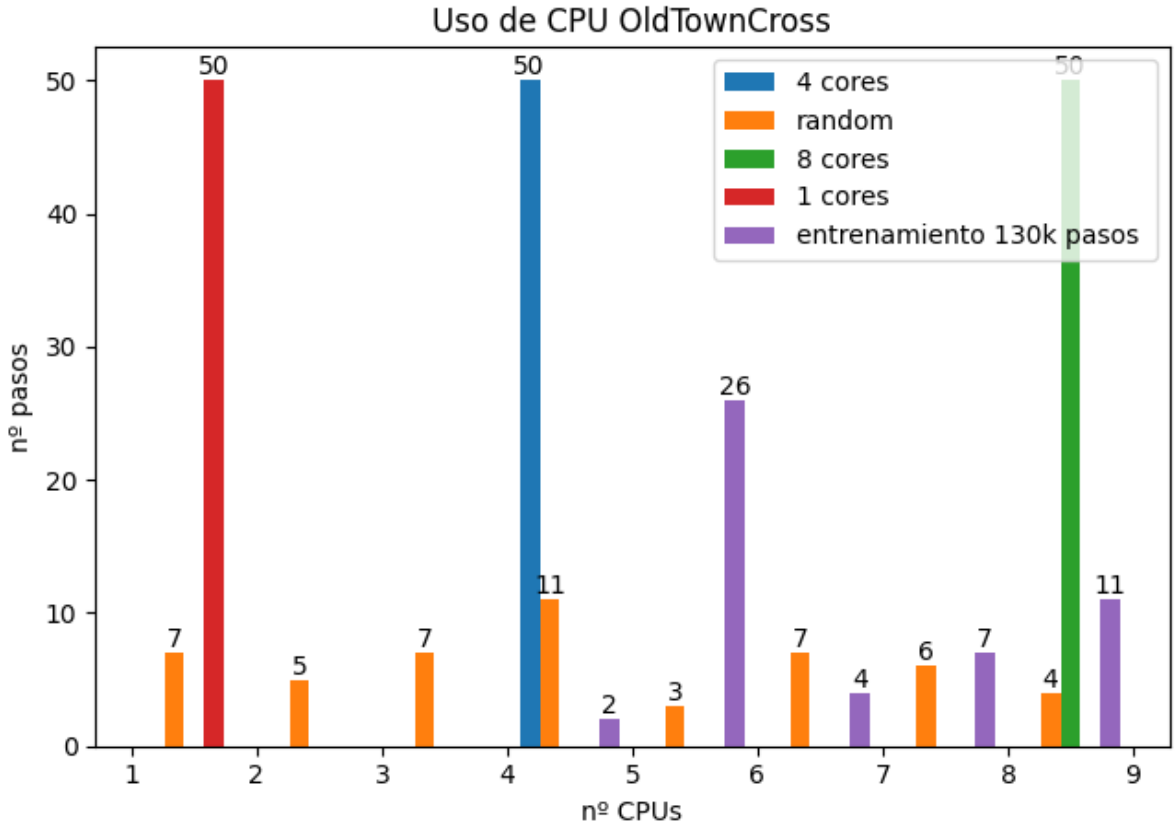


Figura 4-24: Uso de CPUS para OldTownCross.

BT709Parakeets

En este caso, con Parakeets, otro vídeo de testeo, tenemos unos resultados ligeramente diferentes. La baseline de 8 cores no es la mejor en este caso, sino que es la de 4 núcleos es la que mejor recompensa acumulada ofrece. Sin embargo, con 4 u 8 núcleos, a Kvazaar le es difícil mantenerse en el rango objetivo (Figura 4-26).

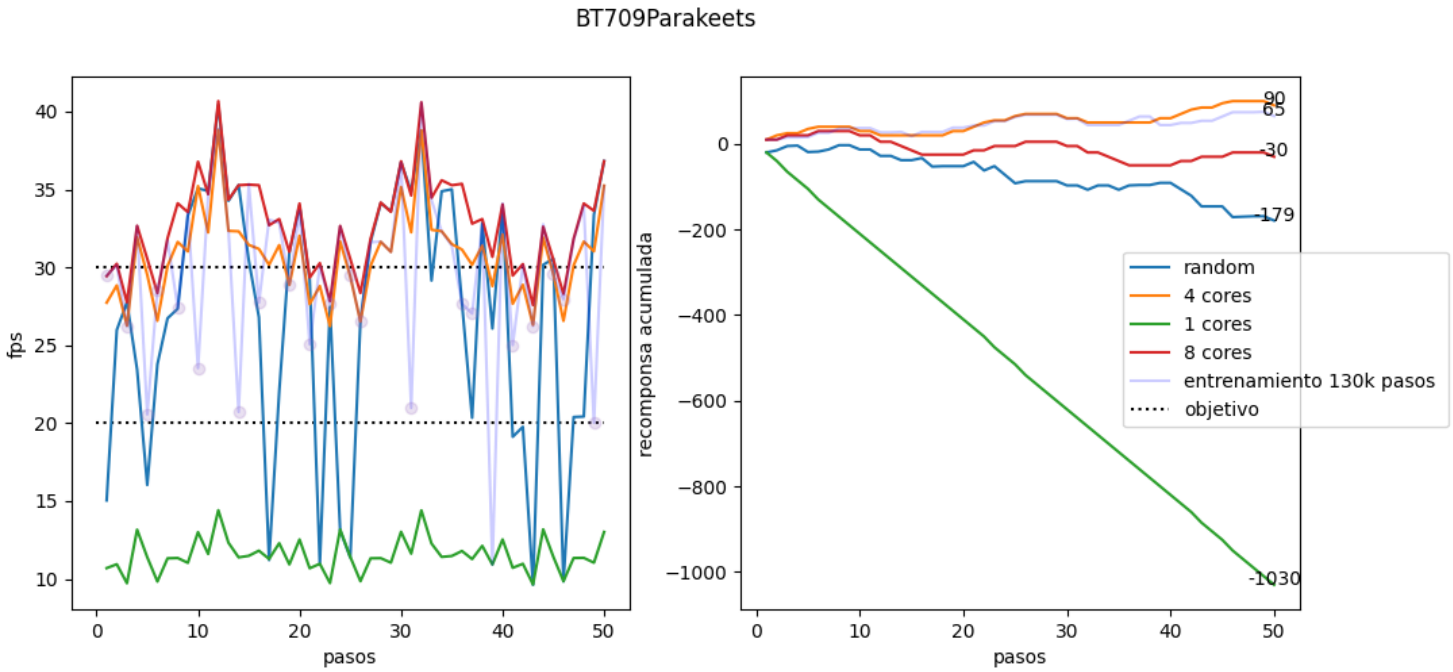


Figura 4-26: Resultados base para Parakeets.

En este caso, si comparamos la política con la baseline de 4 núcleos (Figura 4-28 y Figura 4-27), a pesar de que esta última ofrece un recompensa acumulada mejor, el agente entrenado se mantiene más veces en el rango objetivo. El agente entrenado es capaz de rectificar una acción que genera un estado fuera del objetivo, la mayoría de las veces volviendo de unos FPS altos. Así consigue mantenerse prácticamente, la mitad de los pasos en el objetivo, y la otra mitad se encuentran por encima. Este rendimiento es justo lo que queríamos al definir las recompensas en el punto 3.2.3.

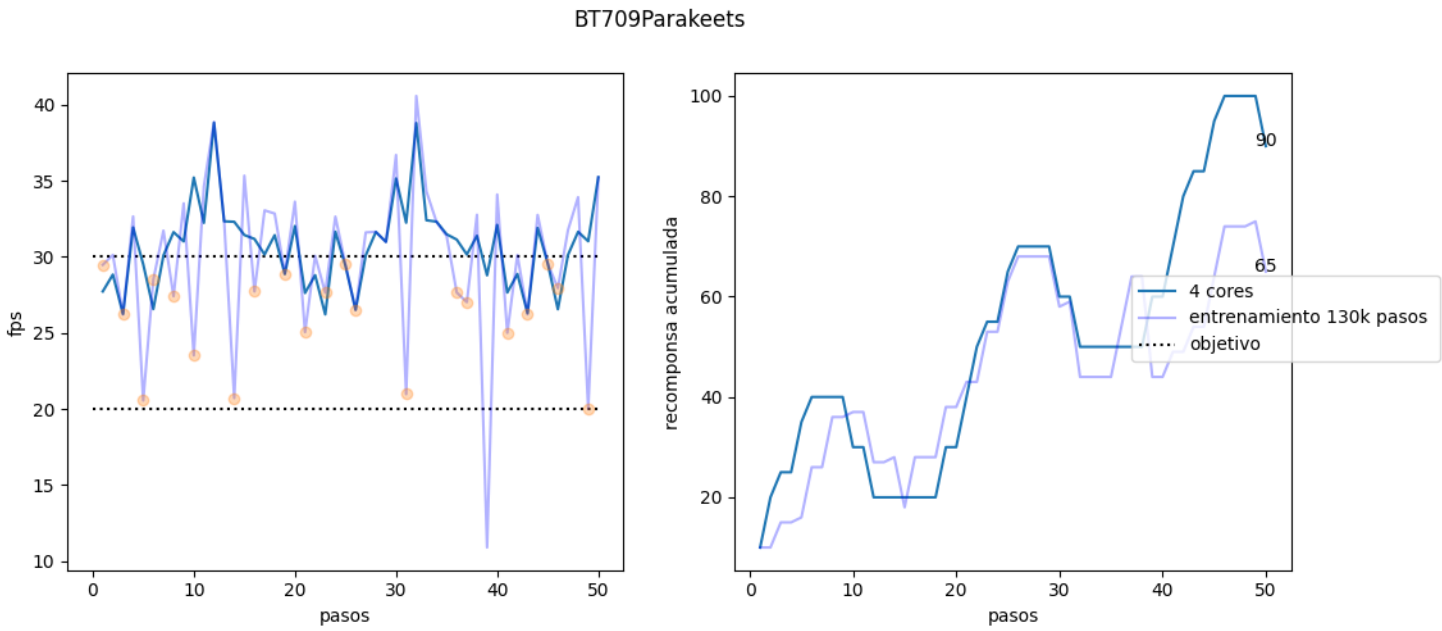


Figura 4-28: Resultados para la política entrenada para Parakeets.

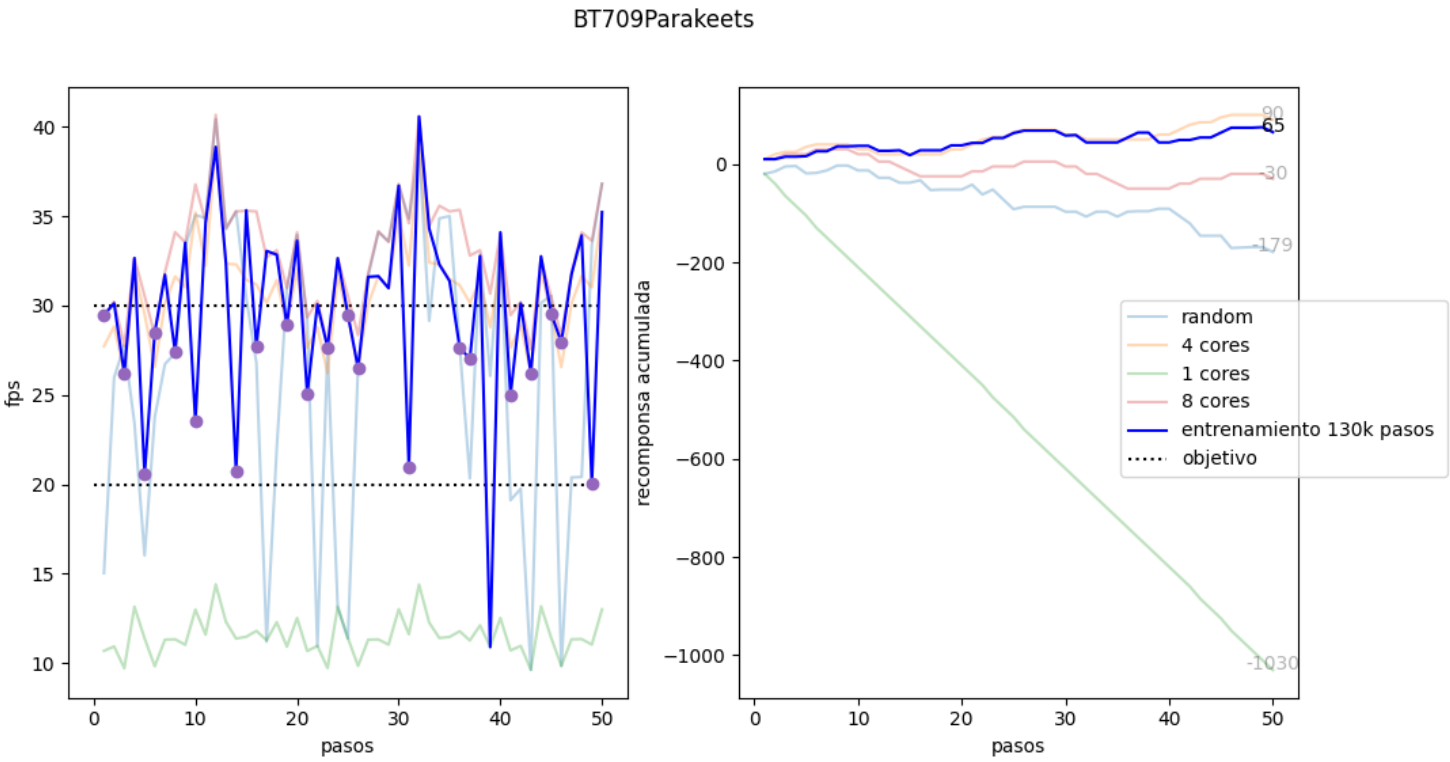


Figura 4-27: Comparación entre la política entrenada y la baseline de 4 núcleos para Parakeets.

En este caso, en cuanto al uso de CPUS (Figura 4-29) el agente se comporta de manera similar a E_FourPeople: el agente no parece decantarse por una acción concreta. Si intentamos explicar esto, llegamos a la conclusión de que el agente aprende una política en la que se le presentaron diferentes

tipos de vídeos, por tanto, las acciones que debió tomar son las mejores posibles para todos esos vídeos. Es por eso, que existe tanta variabilidad en el uso de CPUS, ya que el agente no sabe qué tipo de vídeo se le está presentando, únicamente conoce una cantidad de FPS. Sin embargo, y de manera análoga a E_FourPeople, el uso de CPUS se mantiene en la mitad (4 o 5 núcleos) la mayoría de las veces.

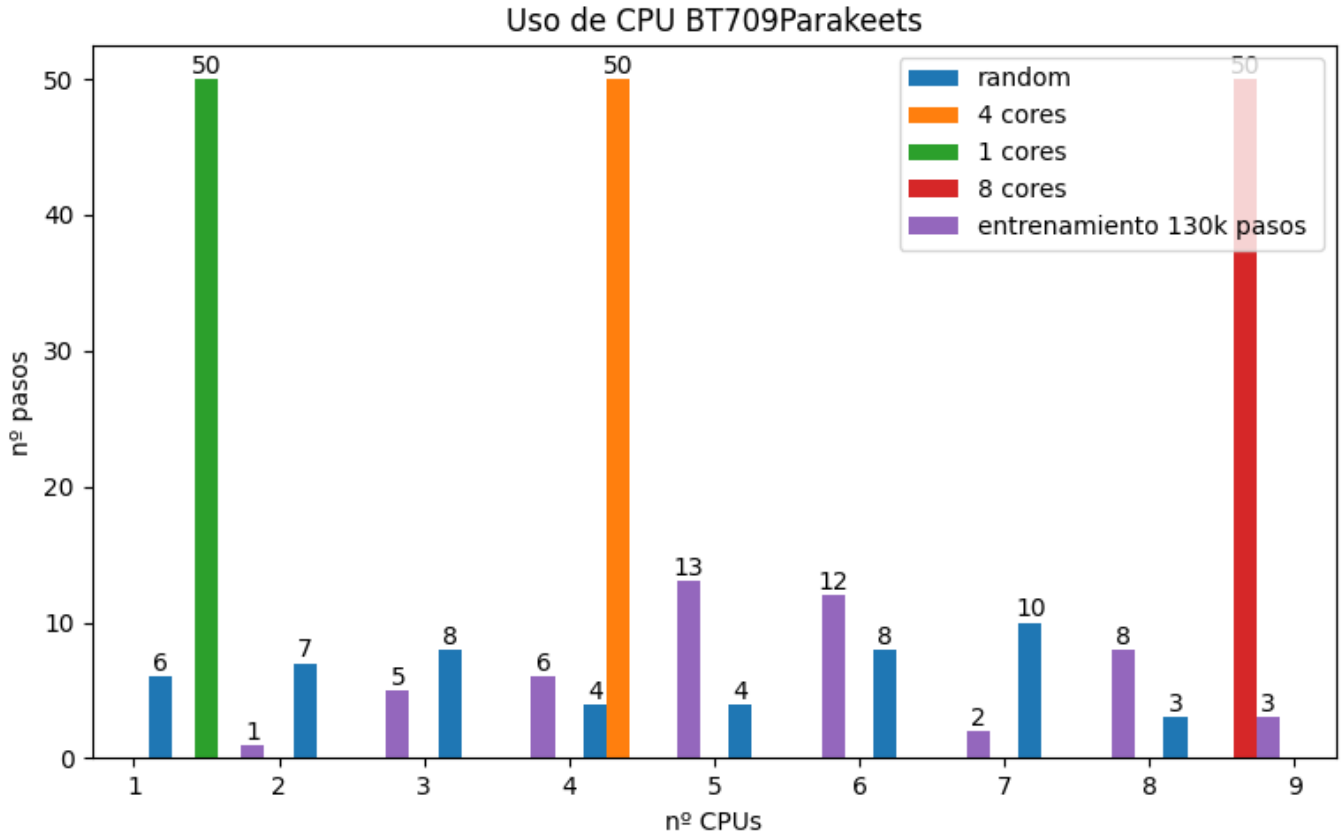


Figura 4-29: Uso de CPU para Parakeets.

SVT04a

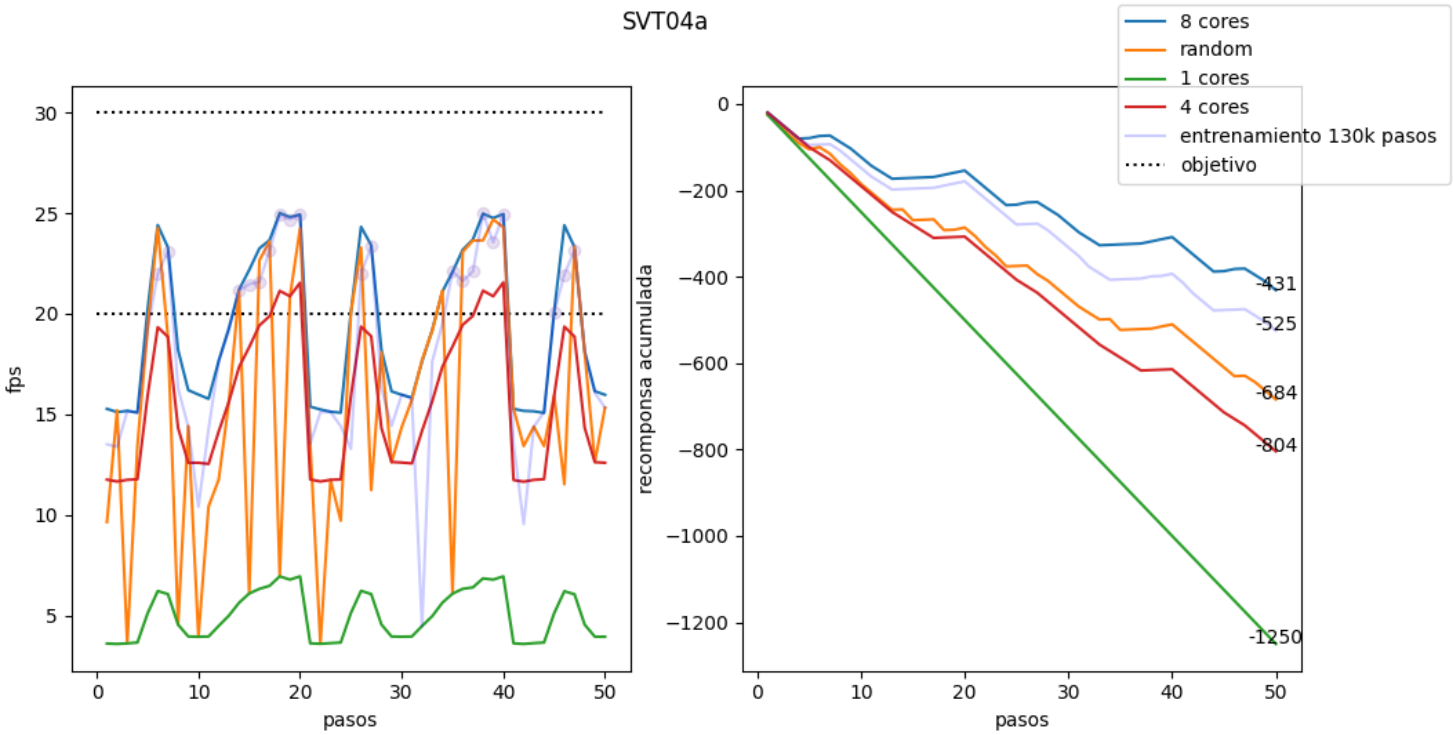


Figura 4-30: Resultados base para SVT04.

Este es sin duda el vídeo más complicado de procesar de los vídeos de testo, la base de 8 núcleos prácticamente no puede mantenerse en el objetivo y se queda por debajo (Figura 4-30), dando por tanto un rendimiento inadecuado. Sin embargo, está bien visualizar cómo se comporta el agente entrenado ante una situación tan adversa.

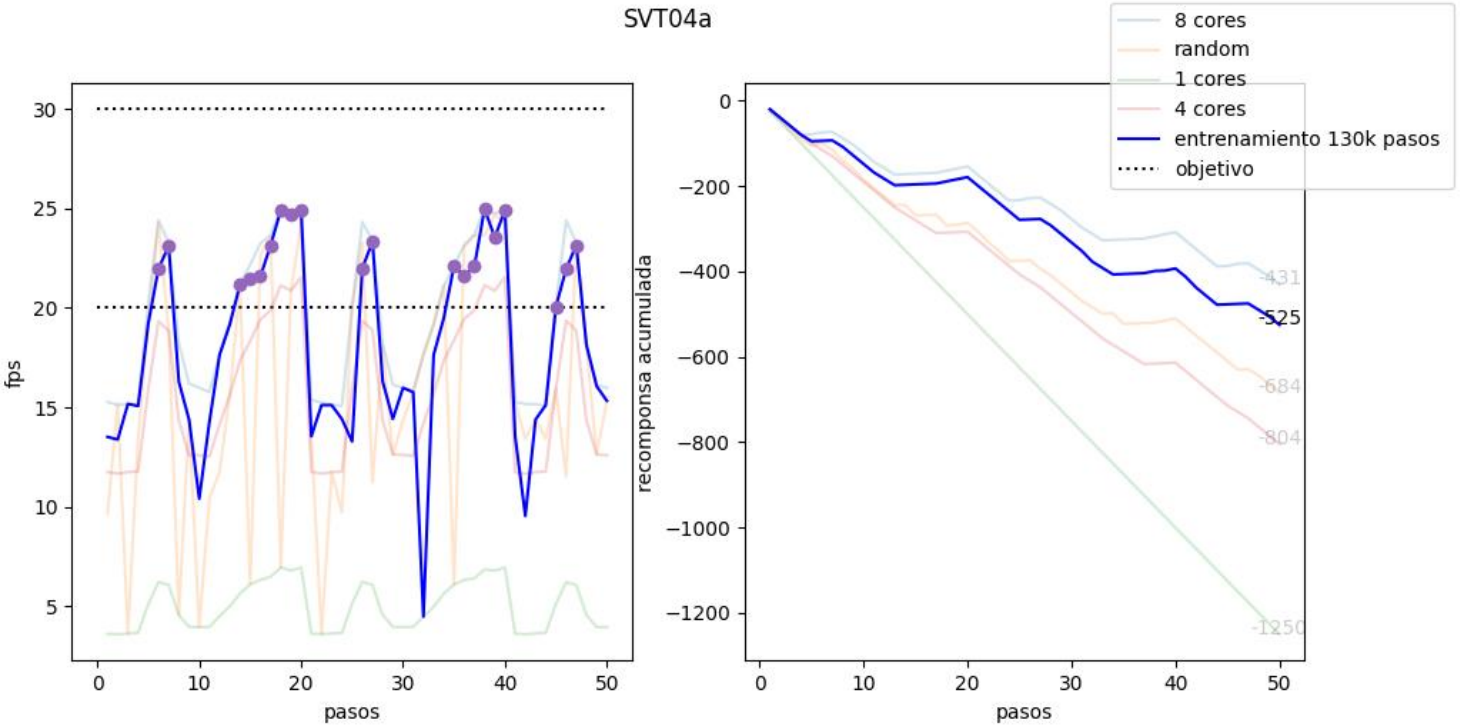


Figura 4-31: Rendimiento del agente entrenado para SVT04.

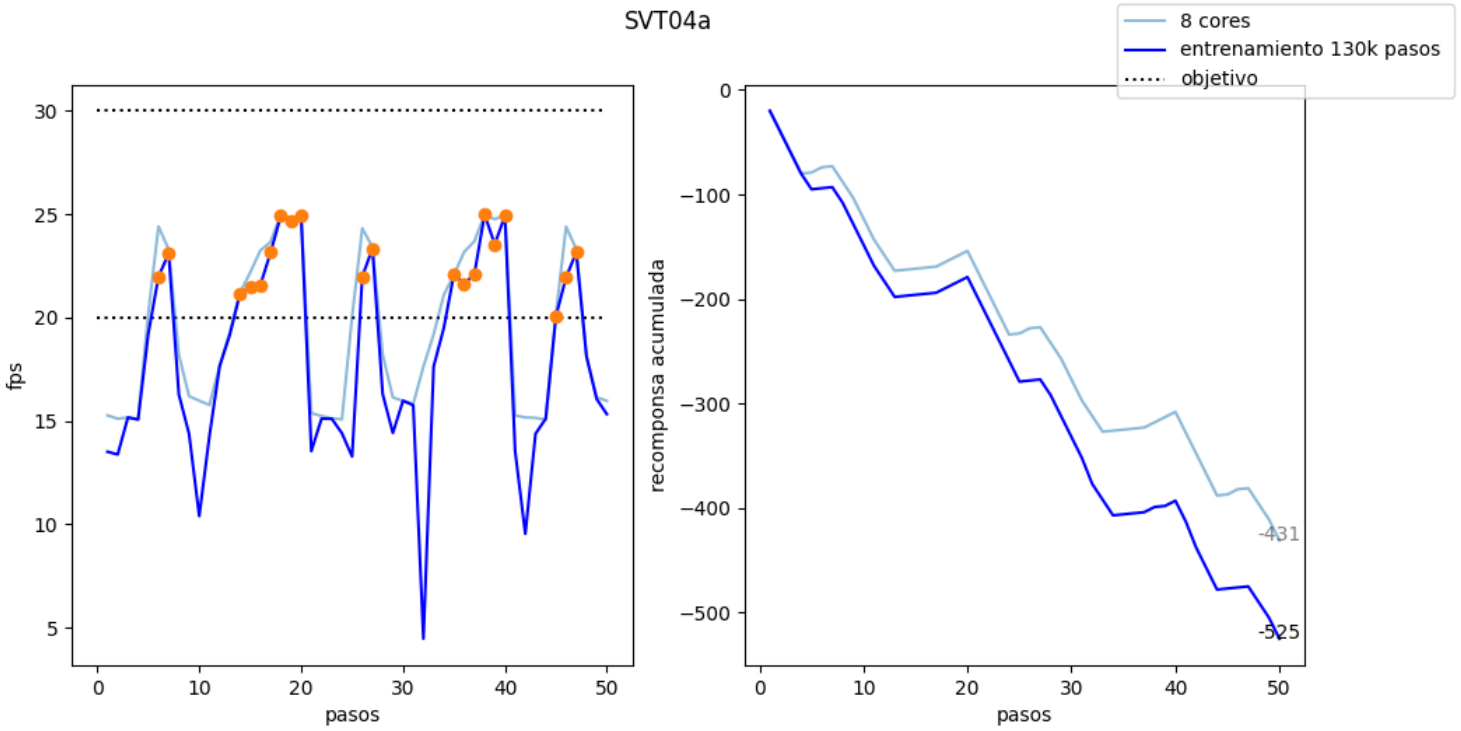


Figura 4-32: Comparación del rendimiento con 8 núcleos y con agente entrenado(SVT04).

Si vemos el rendimiento del agente entrenado aquí (Figura 4-31 y Figura 4-32), este es similar al de 8 núcleos: al agente le resulta muy complicado mantenerse en el objetivo y muchas veces se queda por debajo de él. Además, presenta alguna bajada importante de FPS, lo cual es normal ante semejante vídeo y teniendo en cuenta que el agente no toma decisiones perfectas. No obstante, el uso de CPUS (Figura 4-33) es bastante más contenido que usando 8 núcleos fijos. Se demuestra que el agente es capaz de mantener un rendimiento similar si bien malo, pero optimizando el uso de núcleos.

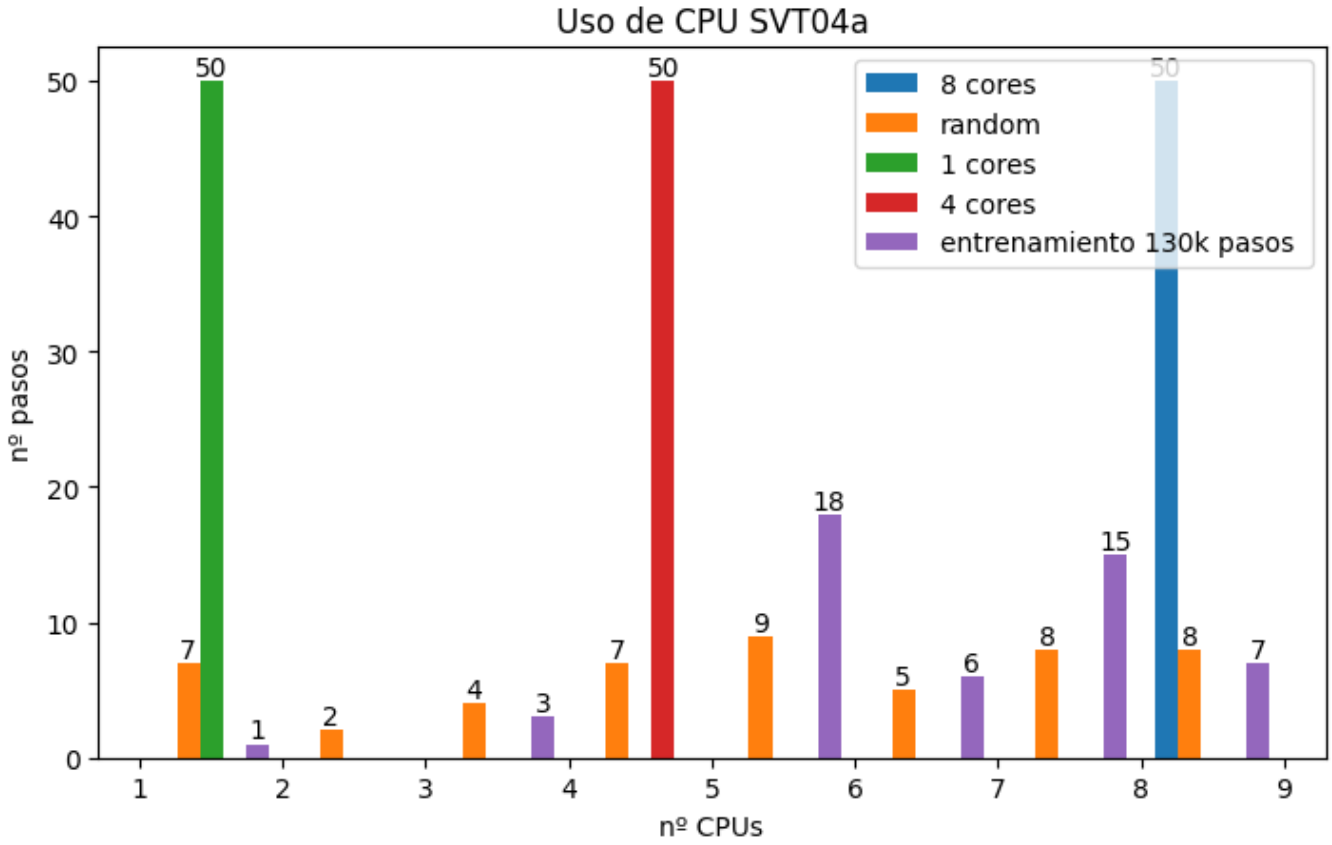


Figura 4-33: Uso de CPU para SVT04a

Capítulo 5 - Conclusiones y trabajo futuro

Antes de concluir, es necesario indicar las dificultades encontradas a lo largo del proceso de creación de este proyecto. En primer lugar, el paradigma de aprendizaje por refuerzo presenta, como ya habíamos planteado en la introducción, sutilezas que ralentizan el diseño y resolución de un problema. Definir unos conjuntos de estados y recompensas óptimos, así como los mejores parámetros del algoritmo pueden llegar a ser tareas complicadas si no se hacen de manera organizada. Es cierto que, en este proyecto, al haber sido una primera toma de contacto con las bases de este paradigma, han surgido impedimentos propios de la inexperiencia. Sin embargo, tomando decisiones paso a paso, se resolvieron. Un ejemplo de esto fue la idea de no probar dos cambios en el entorno a la vez para evitar conocer las causas de posibles fallos en el futuro.

En segundo lugar, las capacidades de la máquina usada siempre pueden dar lugar a limitaciones a la hora de recolectar suficientes resultados. El hecho de compartir un servidor de pruebas entre otros estudiantes siempre genera conflictos.

Sin embargo, viendo los resultados obtenidos en nuestras pruebas, podemos deducir que nuestro agente ha aprendido de manera positiva. En líneas generales, podemos asegurar que nuestro agente entrenado es capaz de realizar algún hecho de los siguientes: mantenerse muy próximo a comportamientos fijos de uso de CPU sin tomar decisiones desorbitadas (como ocurre en el caso de E_FourPeople) o mejorarlos, es decir, mantenerse en nuestro objetivo de 20 a 30 FPS utilizando menor número de núcleos (OldTownCross). Además, el agente es capaz de rectificar sus acciones cuando son incorrectas. En E_FourPeople, por ejemplo, el agente disminuye el rendimiento para mantenerse en el objetivo cuando es necesario. Cuando es imposible lograr el objetivo de fotogramas por segundo, como SVT4a el agente, nunca está entre los peores resultados. Esto demuestra que nuestro diseño, si bien posiblemente no sea el óptimo, ha encaminado al agente a tomar las decisiones que queríamos.

Como trabajo futuro, se pueden plantear las siguientes líneas de investigación y mejoras:

- Ampliar la capacidad de recolección de datos, aumentando el número de agentes haciendo que cada uno tenga una instancia de Kvazaar activa. Esto mejoría la velocidad de aprendizaje.
- Realizar tareas más exhaustivas en cuanto a la elección de hiperparámetros del algoritmo. Existen muchas combinaciones posibles aun no barajadas que podrían refinar el entorno.
- Optar por otros conjuntos de recompensas y estados. Jugar con estas entradas puede mejorar el rendimiento del agente.
- Comparar el rendimiento de PPO con otros algoritmos. Sabemos que PPO tiene una tendencia a converger muy lentamente. Además, si bien funciona perfectamente con nuestro espacios de observaciones y acciones, existen algoritmos que cumplen mejor su trabajo, como DQN.

En definitiva, el aprendizaje por refuerzo ha demostrado su valor en nuestro problema. Está claro que es muy difícil conseguir un rendimiento perfecto, y menos si se trata de entornos autómatas no supervisados que dependen mucho de la aleatoriedad. Sin embargo, hemos podido comprobar que ante un problema real como es el caso de la optimización de recursos en un tarea específica, este paradigma da lugar a decisiones que seguramente no se nos habrían ocurrido. Esto demuestra su potencial, tanto por los resultados obtenidos como por las posibilidades futuras que hemos planteado. Estos resultados podrían ser tomados en un futuro como la norma a seguir para ejecutar este software en concreto, lo que, si se extendiese a grandes corporaciones, ahorraría mucho en costes de electricidad y hardware.

Finalmente, el proceso de desarrollo de este proyecto ha resultado, a título personal, un desafío, así como una experiencia muy enriquecedora ya sea por afianzar los conocimientos obtenidos durante el Grado de Ingeniería Informática como por tener contacto con herramientas actuales de *machine learning*.

Chapter 5 – Conclusions and future work

Before concluding this project, it is important to summarize the difficulties encountered during its creation. First, the reinforcement learning paradigm has its drawbacks that slow solving problems. It can be hard to define good rewards, states and hiperparameters of the algorithm if it is not done properly. It is true that, several problems have been faced during the implementation since it was the first contact with the paradigm. However, making choices step by step helped solving them. For example, taking one change at a time in the environment would avoid not knowing the cause of future problems.

Secondly, the machine used can have its limitations. Using a shared server among other students for testing purposes can be challenging sometimes.

However, if we look to the results, we can deduce that our agent has learned in a positive way. In general, our agent can do one of the following things: maintain a similar behavior of CPU usage compared to the fixed CPU situations without making absurd decisions (E_FourPeople) or improving it, keeping the FPS rate in between our objective using a smaller number of CPUs (OldTownCross). In fact, the agent can rectify from its bad decisions like in E_FourPeple reducing the FPS rate to stay in the objective when necessary. If reaching the objective is not possible, the agent works as in fixed CPU cases, and never in the worst results category. This can mean that we told the agent to make the decisions we wanted, even if they probably were not the best ones.

As future work, different investigation ideas come to mind:

- Extending sample collecting by using more agents in parallel. Each one could have a Kvazaar process that could lead to better learning speeds.
- Studying other sets of the algorithm hyperparamers that could tune our environment.
- Choosing other states and rewards can increase the environment performance.
- Comparing the PPO algorithm with other ones. It is known that PPO has a slow convergence rate and other algorithms can have better

performance with the same observation and action spaces, like DQN.

The reinforcement learning has proven its value in our problem. It is hard to get perfect results, especially in non-supervised automatic environments that depend greatly on randomness. However, in a real use case scenario as this, the paradigm gives us decisions that probably would never have been possible by ourselves. These results have the potential to be used in the future as the way of acting for big corporations that could use similar software and could give big savings in hardware and electricity costs.

Personally, this project has been both a challenge and an enriching experience because of all the knowledge acquired during this Degree that has been deployed here and the possibility of using current machine learning tools.

BIBLIOGRAFÍA

1. **Sutton, Richard S. y Barto, Andrew G.** *Reinforcement learnign: an introduction.* s.l. : The MIT Press, 2020.

2. **Documentación de RLLIB. [En línea] https://docs.ray.io/en/latest/rllib.html.**

3. Kim, Tyler. Understanding Proximal Policy Optimization. [En línea] https://towardsdatascience.com/understanding-and-implementing-proximal-policy-optimization-schulman-et-al-2017-9523078521ce.

4. Documentación de GYM. [En línea] https://gym.openai.com/docs/.

5. Bonsai. Deep Reinforcement Learning Models: Tips & Tricks for Writing Reward Functions. [En línea] 2017. https://medium.com/@BonsaiAI/deep-reinforcement-learning-models-tips-tricks-for-writing-reward-functions-a84fe525e8e0.

6. Brownlee, Jason. A gentle introduction to mini-batch gradient Descent and how to configure batch size. [En línea] 2019. https://machinelearningmastery.com/gentle-introduction-mini-batch-gradient-descent-configure-batch-size/.

7. Tactics, Aurelian. Understanding PPO Plots in Tensorboard. [En línea] 2018. https://medium.com/aureliantactics/understanding-ppo-plots-in-tensorboard-cbc3199b9ba2.

8. Documentación de Python. [En línea] https://docs.python.org/3/.