Universidad Politécnica de Madrid

Escuela Técnica Superior de Ingenieros Informáticos

Máster Universitario en Inteligencia Artificial

Trabajo Fin de Máster

Optimización Bayesiana con Multifidelidad para la Búsqueda de Hiperparámetros en Algoritmos de Aprendizaje por Refuerzo

Autor: Gonzalo Martínez Martínez
Tutores: Martín Molina González
Eduardo César Garrido Merchán

Madrid, Junio 2024

Este Trabajo Fin de Máster se ha depositado en la ETSI Informáticos de la Universidad Politécnica de Madrid para su defensa.

Trabajo Fin de Máster
Máster Universitario en Inteligencia Artificial

Título: Optimización Bayesiana con Multifidelidad para la Búsqueda de Hiperparámetros en Algoritmos de Aprendizaje por Refuerzo

Junio 2024

Autor(a): Gonzalo Martínez Martínez
Tutor(a): Martín Molina González
Departamento de Inteligencia Artificial
ETSI Informáticos
Universidad Politécnica de Madrid

Eduardo César Garrido Merchán
Departamento de Métodos Cuantitativos
ICADE
Universidad Pontificia Comillas

Agradecimientos

Quisiera expresar mi más sincero agradecimiento a todas aquellas personas que han contribuido de manera significativa a la realización de este trabajo de fin de máster.

En primer lugar, agradezco a mis amigos y compañeros del máster por su apoyo constante, su colaboración y sus palabras de aliento durante este período de investigación, desarrollo de código y redacción.

Agradezco también a mis compañeros de la residencia, quienes han compartido conmigo momentos de estudio, reflexión y distracción, haciendo de esta experiencia un camino más llevadero y enriquecedor.

Mi gratitud se extiende a los profesores de la escuela, en especial a Martín Molina González y Eduardo César Garrido Merchán, quienes han guiado y orientado este trabajo con su sabiduría, paciencia y dedicación. Su invaluable asesoría ha sido fundamental para el desarrollo de este proyecto.

No puedo dejar de mencionar el apoyo incondicional de mi familia, quienes han sido mi principal fuente de inspiración, motivación y apoyo. Agradezco a mis padres por brindarme la oportunidad de estudiar fuera de Sevilla y por su constante apoyo emocional. En particular, quiero dedicar un agradecimiento especial a mi abuelo Valentín y mi abuela Victoria, cuya generosidad y sacrificio han hecho posible que pueda cumplir este sueño académico. Su amor y apoyo incondicional han sido un pilar fundamental para realizar este trabajo.

A todos ustedes, gracias de corazón.

Resumen

Este trabajo de investigación se centra en la comparación entre la optimización bayesiana estándar y la optimización bayesiana con multifidelidad en la búsqueda de hiperparámetros para mejorar el rendimiento de algoritmos de aprendizaje por refuerzo en entornos como OpenAI LunarLander y CartPole. El objetivo principal es determinar si la optimización bayesiana con multifidelidad ofrece mejoras significativas en términos de eficiencia y rendimiento del modelo en comparación con la optimización bayesiana estándar.

Para abordar esta pregunta, se desarrollaron varias implementaciones en Python, evaluando la calidad de las soluciones mediante la media de las recompensas obtenidas como función objetivo. Se llevaron a cabo una serie de experimentos para cada entorno y versión utilizando diferentes semillas, asegurando que los resultados no fueran simplemente producto de la aleatoriedad inherente a los algoritmos de aprendizaje por refuerzo.

Los resultados obtenidos demuestran que la optimización bayesiana con multifidelidad supera a la optimización bayesiana estándar en varios aspectos clave. En el entorno de LunarLander, la multifidelidad permitió una mejor convergencia y un rendimiento más estable, logrando una mayor recompensa media en comparación con la versión estándar. En el entorno de CartPole, aunque ambos métodos alcanzaron rápidamente la recompensa máxima, la multifidelidad lo hizo con mayor consistencia y en menos tiempo.

Estos hallazgos destacan la capacidad de la multifidelidad para optimizar los hiperparámetros de manera más eficiente, utilizando menos recursos y tiempo, mientras se logra un rendimiento superior.

Palabras clave: Optimización bayesiana, optimización multifidelidad, ajuste de hiperparámetros, aprendizaje por refuerzo.

Abstract

This research focuses on comparing standard Bayesian optimization and multifidelity Bayesian optimization in the hyperparameter search to improve the performance of reinforcement learning algorithms in environments such as OpenAI LunarLander and CartPole. The primary goal is to determine whether multifidelity Bayesian optimization provides significant improvements in solution quality compared to standard Bayesian optimization.

To address this question, several Python implementations were developed, evaluating the solution quality using the mean of the total rewards obtained as the objective function. Various experiments were conducted for each environment and version using different seeds, ensuring that the results were not merely due to the inherent randomness of reinforcement learning algorithms.

The results demonstrate that multifidelity Bayesian optimization outperforms standard Bayesian optimization in several key aspects. In the LunarLander environment, multifidelity optimization achieved better convergence and more stable performance, yielding a higher average reward compared to the standard version. In the CartPole environment, although both methods quickly reached the maximum reward, multifidelity did so with greater consistency and in less time.

These findings highlight the ability of multifidelity optimization to optimize hyperparameters more efficiently, using fewer resources and less time while achieving superior performance.

Keywords: Bayesian optimization, multifidelity optimization, hyperparameter tuning, reinforcement learning.

Tabla de contenidos

Agradecimientos **i**

1 Introducción **1**
1.1 Objetivo . . . 1
1.2 Estructura del Documento . . . 2

2 Conceptos Teóricos **3**
2.1 Aprendizaje por Refuerzo . . . 3
2.2 Algoritmos de Aprendizaje por Refuerzo . . . 4
2.3 Aprendizaje Profundo por Refuerzo . . . 4
2.4 Proximal Policy Optimization (PPO) . . . 5
2.4.1 Conceptos Clave de PPO . . . 6
2.4.2 Función Objetivo de PPO . . . 6
2.5 Optimización Bayesiana . . . 7
2.5.1 Herramientas de Optimización de Hiperparámetros . . . 8
2.5.2 SMAC: Sequential Model-based Algorithm Configuration . . . 9
2.5.3 SMAC3 . . . 10

3 Diseño del Marco de Experimentación **11**
3.1 Entorno . . . 11
3.1.1 CartPole . . . 12
3.1.2 LunarLander . . . 12
3.2 Herramientas utilizadas . . . 13
3.3 Configuración de la optimización bayesiana . . . 14
3.3.1 Adaptación para optimización sin multifidelidad . . . 14
3.3.1.1 Configuración y Parámetros Iniciales . . . 15
3.3.1.2 Funciones Principales . . . 15
3.3.1.3 Proceso Principal de Optimización . . . 16
3.3.1.4 Parámetros y Configuración . . . 16
3.3.1.5 Clases y Funciones Principales . . . 17
3.3.2 Adaptación para optimización bayesiana con multifidelidad . . . 18
3.3.2.1 Funciones Principales Adicionales . . . 18
3.3.2.2 Proceso Principal de Optimización con Multifidelidad . . . 18
3.3.3 Métricas de evaluación . . . 20
3.4 Procedimiento experimental . . . 20

4 Resultados de la experimentación **21**
4.1 Introducción . . . 21
4.2 Experimentos de la versión de optimización bayesiana básica en CartPole 22

4.3 Experimentos de la versión de optimización bayesiana con multifidelidad en CartPole . . . 24
4.4 Experimentos de la versión de optimización bayesiana básica en LunarLander . . . 27
4.5 Experimentos de la versión de optimización bayesiana con multifidelidad en LunarLander . . . 30
4.6 Conclusiones . . . 33

5 Conclusiones **37**
5.1 Trabajo Futuro . . . 38

Bibliografía **40**

Índice de figuras

3.1 Esquema del entorno CartPole . . . 12
3.2 Esquema del entorno LunaLander . . . 13

4.1 Rendimiento del agente en CartPole con la versión básica de optimización bayesiana . . . 22
4.2 Rendimiento del agente en CartPole reduciendo el número de pasos con la versión básica de optimización bayesiana . . . 23
4.3 Estudio de la convergencia en CartPole con la versión básica de optimización bayesiana . . . 24
4.4 Rendimiento del agente en CartPole con la versión con multifidelidad de optimización bayesiana . . . 25
4.5 Rendimiento del agente con número bajo de pasos en CartPole con la versión con multifidelidad de optimización bayesiana . . . 26
4.6 Estudio de la convergencia en CartPole con la versión con multifidelidad de optimización bayesiana . . . 27
4.7 Rendimiento del agente en LunarLander con la versión básica de optimización bayesiana . . . 28
4.8 Rendimiento del agente con número bajo de pasos en LunarLander con la versión básica de optimización bayesiana . . . 29
4.9 Estudio de la convergencia en LunarLander con la versión básica de optimización bayesiana . . . 30
4.10 Rendimiento del agente en LunarLander con la versión con multifidelidad de optimización bayesiana . . . 31
4.11 Rendimiento del agente con número bajo de pasos en LunarLander con la versión con multifidelidad de optimización bayesiana . . . 32
4.12 Estudio de la convergencia en LunarLander con la versión con multifidelidad de optimización bayesiana . . . 33

Índice de cuadros

4.1 Comparación de resultados obtenidos con y sin multifidelidad en CartPole 34
4.2 Comparación de resultados obtenidos con y sin multifidelidad en LunarLander 34

Capítulo 1

Introducción

El campo del aprendizaje por refuerzo (RL) ha experimentado un crecimiento significativo en los últimos años debido a su capacidad para abordar una amplia gama de problemas de toma de decisiones en entornos complejos y dinámicos. Sin embargo, uno de los desafíos clave en RL es el ajuste de hiperparámetros, que son parámetros que controlan el comportamiento y el rendimiento de los algoritmos de aprendizaje. La optimización eficiente de estos hiperparámetros es crucial para obtener un rendimiento óptimo del algoritmo de RL en un tiempo razonable.

En este contexto, la optimización bayesiana ha surgido como una técnica prometedora para el ajuste de hiperparámetros en RL. La optimización bayesiana aprovecha el poder de los modelos probabilísticos para explorar el espacio de búsqueda de manera eficiente, adaptándose a medida que se observan nuevas muestras. Sin embargo, en entornos computacionalmente costosos, como aquellos que involucran simulaciones complejas o modelos de RL de alta dimensionalidad, la optimización bayesiana estándar puede ser prohibitivamente costosa en términos de tiempo de computación.

Para abordar este problema, la optimización bayesiana con multifidelidad se ha propuesto como una extensión que aprovecha información de distintos niveles de fidelidad de evaluación del modelo para acelerar la convergencia del proceso de optimización. Esta técnica es especialmente útil en casos donde es posible obtener evaluaciones de bajo costo que proporcionan información aproximada sobre la función objetivo.

1.1. Objetivo

El objetivo principal de este Trabajo de Fin de Máster es analizar la eficacia de la optimización bayesiana con multifidelidad en el ajuste de hiperparámetros para algoritmos de aprendizaje por refuerzo, en comparación con la optimización bayesiana estándar.

Para lograr este objetivo principal, se plantean los siguientes objetivos específicos:

1. Implementar una versión estándar de la optimización bayesiana para el ajuste de hiperparámetros de algoritmos de aprendizaje por refuerzo, utilizando la biblioteca SMAC3.

2. Desarrollar una implementación de la optimización bayesiana con multifidelidad, que aproveche la información de múltiples niveles de fidelidad en la evaluación del modelo.

3. Utilizar entornos de aprendizaje por refuerzo populares, como los disponibles en el respositorio Gym de OpenAI para evaluar el rendimiento de ambas técnicas de optimización.

4. Diseñar y llevar a cabo experimentos para comparar el rendimiento de la optimización bayesiana estándar y la optimización bayesiana con multifidelidad en términos de calidad de solución y eficiencia computacional.

5. Analizar y discutir los resultados obtenidos, con el objetivo de determinar si la optimización bayesiana con multifidelidad ofrece mejoras significativas en el ajuste de hiperparámetros para algoritmos de aprendizaje por refuerzo en comparación con la optimización bayesiana estándar.

Al alcanzar estos objetivos, se espera contribuir al avance del conocimiento en el campo de la optimización de hiperparámetros para algoritmos de aprendizaje por refuerzo, proporcionando información relevante sobre la eficacia de la optimización bayesiana con multifidelidad y su potencial para mejorar el rendimiento de los agentes de RL en una variedad de aplicaciones prácticas.

1.2. Estructura del Documento

La tesis está estructurada de la siguiente manera:

- **Capítulo 2** En este capítulo se revisan brevemente los conceptos de aprendizaje por refuerzo, se presenta el algoritmo DRL (Deep Reinforcement Learning) que se ha utilizado en los experimentos, se revisan los conceptos de optimización bayesiana, se recopilan algunas herramientas populares utilizadas para la optimización de hiperparámetros y se muestran ejemplos de cómo la optimización bayesiana ha sido utilizada en problemas de aprendizaje por refuerzo.

- **Capítulo 3** Este capítulo presenta la configuración experimental utilizada, incluyendo detalles sobre los entornos de aprendizaje por refuerzo, los algoritmos de optimización utilizados, los hiperparámetros ajustados y cualquier otro aspecto relevante del diseño experimental.

- **Capítulo 4** En este capítulo se discuten los resultados de los experimentos realizados. Se analizan y presentan los datos recopilados durante la fase experimental, se comparan los resultados obtenidos utilizando diferentes técnicas de optimización y se discuten las observaciones significativas.

- **Capítulo 5** Este capítulo concluye el trabajo con un análisis completo del objetivo del proyecto. Se proporciona un resumen de los principales hallazgos experimentales y se proponen posibles líneas de trabajo futuro para continuar con la investigación en esta área.

Capítulo 2

Conceptos Teóricos

La sección de Conceptos Teóricos de este proyecto proporciona una base que ayuda a comprender tanto el marco teórico como la implementación práctica del trabajo. Se exploran conceptos clave relacionados con el aprendizaje por refuerzo (RL), la optimización bayesiana y otras técnicas relevantes que sustentan el enfoque de este trabajo.

2.1. Aprendizaje por Refuerzo

El aprendizaje automático se fundamenta en la interacción entre un agente y un entorno con el fin de alcanzar un objetivo específico. En este contexto, el agente toma decisiones secuenciales en un entorno dinámico con el propósito de maximizar una señal numérica de recompensa a largo plazo.

Los componentes básicos de un sistema de aprendizaje por refuerzo incluyen:

- **Política (π):** La política define la estrategia del agente para seleccionar acciones en función del estado actual del entorno. Puede ser determinista o estocástica, donde la primera asigna una acción específica a cada estado y la segunda asigna una distribución de probabilidad sobre las acciones para cada estado.
- **Recompensa (r):** La recompensa es la retroalimentación inmediata que recibe el agente después de realizar una acción en un estado dado del entorno. Define el objetivo del problema y guía al agente hacia comportamientos deseables.
- **Función de valor (V, Q):** Las funciones de valor estiman la cantidad total de recompensa que el agente puede esperar obtener en el futuro siguiendo una política particular. La función V estima el valor esperado del retorno total desde un estado dado, mientras que la función Q estima el valor esperado del retorno total al tomar una acción específica en un estado particular.
- **Modelo del entorno:** Este componente opcional permite al agente modelar el comportamiento del entorno, lo que facilita la planificación y la toma de decisiones. Proporciona información sobre cómo evolucionará el entorno en respuesta a las acciones del agente.

El problema de aprendizaje por refuerzo se formaliza comúnmente como un Proceso de Decisión de Markov (MDP), donde se cumple la propiedad de Markov, lo que im-

plica que el estado futuro depende únicamente del estado actual y la acción tomada, independientemente del pasado. El objetivo último del agente es encontrar la política óptima que maximice la recompensa esperada a largo plazo.

2.2. Algoritmos de Aprendizaje por Refuerzo

En la práctica del Aprendizaje por Refuerzo (RL), una variedad de algoritmos se han desarrollado para abordar una amplia gama de problemas. Estos algoritmos difieren en sus enfoques para aprender políticas óptimas y funciones de valor, así como en su capacidad para manejar diferentes tipos de entornos y restricciones computacionales.

Algunos de los algoritmos más comunes en RL incluyen:

- **Q-Learning:** Un algoritmo de aprendizaje por refuerzo basado en valor que aprende directamente la función de valor de acción óptima sin requerir un modelo del entorno. Q-Learning se ha utilizado con éxito en una variedad de aplicaciones, desde juegos hasta robótica.

- **SARSA (State-Action-Reward-State-Action):** Otro algoritmo de aprendizaje por refuerzo basado en valor que es similar a Q-Learning pero actualiza la función de valor de acción basándose en la política actual. SARSA es útil en entornos donde la política es importante y puede cambiar durante la interacción.

- **Deep Q-Networks (DQN):** Una extensión de Q-Learning que utiliza redes neuronales profundas para aproximar la función de valor de acción. DQN ha demostrado ser efectivo en problemas de aprendizaje por refuerzo de alto dimensionalidad, como el control de juegos de Atari.

- **Policy Gradient Methods:** Algoritmos que aprenden directamente la política óptima mediante el ajuste de los parámetros de la política de manera que maximicen la recompensa esperada. Estos métodos incluyen REINFORCE, TRPO y PPO, entre otros.

- **Algoritmos Basados en Modelos:** Estos algoritmos aprenden un modelo del entorno y utilizan este modelo para planificar acciones óptimas. Ejemplos incluyen Model-Based RL, Dyna-Q y Monte Carlo Tree Search (MCTS).

Estos son solo algunos ejemplos de los numerosos algoritmos disponibles en el campo del Aprendizaje por Refuerzo. La elección del algoritmo más adecuado depende del problema específico a resolver, las características del entorno y las limitaciones computacionales.

2.3. Aprendizaje Profundo por Refuerzo

El Aprendizaje Profundo por Refuerzo (DRL) es una rama del aprendizaje por refuerzo que utiliza redes neuronales profundas para representar funciones de valor, políticas o modelos de entorno en problemas de decisión secuencial. DRL ha demostrado ser especialmente eficaz en entornos de alta dimensionalidad, como los videojuegos y la robótica, donde los métodos tradicionales de RL pueden enfrentar dificultades para generalizar y aprender representaciones útiles de los datos de entrada.

Una de las mayores contribuciones del DRL es su capacidad para aprender representaciones de características complejas directamente de datos sensoriales brutos, como imágenes o señales de sensor. Esto permite a los agentes aprender comportamientos sofisticados sin la necesidad de ingeniería de características manual.

Uno de los algoritmos más influyentes en DRL es el **Deep Q-Network (DQN)**, introducido por Mnih et al. en 2015. DQN utiliza una red neuronal profunda para aproximar la función de valor de acción $Q(s, a)$, que evalúa la calidad de tomar una acción específica en un estado dado. DQN ha demostrado ser capaz de lograr un rendimiento humano comparable e incluso superarlo en una variedad de juegos Atari.

Además de DQN, existen numerosas extensiones y variantes de DRL que han surgido en los últimos años para abordar diferentes desafíos y mejorar el rendimiento de los agentes. Algunas de estas extensiones incluyen:

- **Deep Deterministic Policy Gradient (DDPG)**: Un algoritmo que extiende el concepto de DQN a entornos continuos mediante el uso de una política determinista y un framework actor-critic. DDPG es especialmente útil en aplicaciones donde las acciones son continuas, como control de robotica.
- **Proximal Policy Optimization (PPO)**: Un algoritmo de política de gradientes que mejora la estabilidad y eficiencia de la optimización mediante el uso de restricciones de confianza en la actualización de políticas. PPO es conocido por su capacidad para manejar entornos con políticas complejas y no estacionarias.
- **Twin Delayed DDPG (TD3)**: Una variante de DDPG que introduce una política de doble critic para mitigar el problema de la sobreestimación de valores en DRL. TD3 se destaca en entornos donde la precisión en la estimación del valor es crucial, como en aplicaciones de control de robots.
- **Soft Actor-Critic (SAC)**: Un enfoque de políticas de gradientes que incorpora una entropía máxima objetiva para promover la exploración eficiente y la diversidad de políticas. SAC es especialmente útil en aplicaciones donde la exploración eficiente es crucial para descubrir políticas óptimas en entornos desconocidos.

Estos algoritmos representan solo una fracción del panorama diverso y en evolución de DRL. Con el rápido avance en la investigación y la tecnología de aprendizaje profundo, se espera que DRL continúe siendo un área emocionante y prometedora en el campo del aprendizaje por refuerzo.

2.4. Proximal Policy Optimization (PPO)

El Aprendizaje por Refuerzo (RL) enfrenta desafíos significativos debido a la naturaleza dinámica de los datos de entrenamiento y la sensibilidad a la sintonización de hiperparámetros. Uno de los problemas principales en RL es que los datos de entrenamiento generados dependen de la política actual del agente, ya que este genera sus propios datos de entrenamiento interactuando con el entorno, en lugar de depender de un conjunto de datos estático como ImageNet. Esta dependencia conlleva a que las distribuciones de datos de las observaciones y recompensas estén en constante cambio, lo que puede provocar inestabilidad en el entrenamiento. Además, RL es muy sensible a la sintonización de hiperparámetros y la inicialización, lo que puede afectar significativamente el rendimiento del agente.

Una solución popular para abordar estos desafíos es el algoritmo Proximal Policy Optimization (PPO), que es un método de gradiente de política diseñado para ser fácil de implementar, eficiente en el uso de muestras y fácil de ajustar.

2.4.1. Conceptos Clave de PPO

El algoritmo PPO se basa en el gradiente de política básica, que utiliza el logaritmo de las probabilidades de acción y el estimador de ventaja para actualizar la política. Sin embargo, un problema común con el gradiente de política básica es que puede resultar en actualizaciones incorrectas de los parámetros de la red y, por lo tanto, en un estimador de ventaja incorrecto.

Para abordar este problema, PPO introduce una restricción de región de confianza (Trust Region Policy Optimization, TRPO), que garantiza que las actualizaciones de la política no se desvíen demasiado de la política anterior. PPO simplifica esta idea al incluir directamente la restricción de región de confianza en el objetivo de optimización.

2.4.2. Función Objetivo de PPO

Para evitar estas caídas repentinas en el rendimiento, PPO propone un enfoque novedoso, que consiste en restringir la cantidad de cambios que la política puede sufrir al maximizar la siguiente función objetivo [Schulman et al., 2017]:

$$L_{\text{CLIP}}(\theta) = \hat{\mathbb{E}}_t \left[\text{mín} \left(r_t(\theta)\hat{A}_t, \textbf{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)\hat{A}_t \right) \right]$$

donde:

- $\hat{\mathbb{E}}_t$ denota el retorno esperado a lo largo de los pasos de tiempo.
- $r(\theta)$ denota la proporción de probabilidad $\frac{\pi_\theta(a_t|s_t)}{\pi_{\text{old}}(a_t|s_t)}$ bajo las nuevas y viejas políticas.
- $\hat{A}_t$ es un estimador de la función de ventaja en el paso de tiempo t, esta función indica cuánto mejor es una acción a en un estado s en comparación con el rendimiento promedio de la política en ese estado.
- ϵ es un hiperparámetro para restringir el rango de la actualización de la política, generalmente 0.1 o 0.2.

La implementación de PPO utilizada en esta investigación es de Stable Baselines [4], un conjunto de implementaciones mejoradas de algoritmos de RL basadas en OpenAI, y los hiperparámetros disponibles en la implementación son los siguientes:

- **Policy (Política)**: La política define la estrategia que el agente utiliza para tomar decisiones en el entorno. En este caso, se utiliza una política de actor-critic, que combina elementos de políticas basadas en valores (como critic) y políticas basadas en acciones (como actor). Por defecto, se utiliza un Perceptrón Multicapa (MLP) con 2 capas ocultas y 64 neuronas cada una para representar esta política.

- **Entorno**: Se refiere al entorno en el que el agente de RL (Reinforcement Learning) está interactuando. En este caso, se utiliza el entorno proporcionado por la biblioteca Gymnasium, CartPole o LunaLander.
- **Tasa de aprendizaje**: Es el parámetro que controla la magnitud de los ajustes que se realizan a los pesos de la red neuronal (MLP). Una tasa de aprendizaje más alta puede llevar a actualizaciones de peso más grandes, lo que puede acelerar el aprendizaje, pero también puede hacer que el entrenamiento sea menos estable. Una tasa de aprendizaje baja puede hacer que el aprendizaje sea más estable pero más lento.
- **Número de pasos**: Se refiere al número de pasos de interacción entre el agente y el entorno antes de realizar una actualización de los pesos de la red neuronal. Más pasos pueden permitir al agente recopilar más información antes de actualizar su política, lo que puede llevar a una convergencia más eficiente.
- **Tamaño del lote**: Es el número de muestras de experiencia que se utilizan en cada actualización de la política. Un tamaño de lote más grande puede proporcionar una estimación más precisa del gradiente de la función objetivo, pero también puede requerir más memoria y tiempo de cómputo.
- **Número de épocas**: Indica cuántas veces se pasa por el conjunto de datos completo durante el entrenamiento de la política. Cada época implica una actualización de los pesos de la red neuronal basada en todo el conjunto de datos recopilado.
- **Factor de descuento** γ: Es un factor que determina cuánto peso se le da a las recompensas futuras en comparación con las recompensas inmediatas. Un γ más cercano a 1 indica que se da más peso a las recompensas futuras, lo que puede llevar a una planificación a largo plazo.
- **GAE** λ: Es el factor utilizado en el Estimador de Ventaja Generalizado (GAE), que es una medida de cuánto mejor es una acción en comparación con la acción media esperada. Controla el intercambio entre sesgo y varianza en la estimación de la ventaja, lo que puede afectar la estabilidad y la velocidad de convergencia del algoritmo.
- **Rango de recorte** ϵ: Es un parámetro utilizado en la función de pérdida para evitar actualizaciones de políticas demasiado grandes. Controla cuánto se permite que las nuevas políticas diverjan de las antiguas en cada actualización. Un ϵ más pequeño hace que las actualizaciones sean más conservadoras, lo que puede mejorar la estabilidad del entrenamiento.
- **Coeficiente de entropía**: Es un término adicional agregado a la función de pérdida para incentivar la exploración del agente. La entropía mide la incertidumbre en la distribución de probabilidad de las acciones, por lo que un coeficiente de entropía más alto fomenta una mayor exploración del espacio de acciones. Esto puede ayudar a evitar que el agente se estanque en políticas subóptimas.

2.5. Optimización Bayesiana

La Optimización Bayesiana es una técnica utilizada para optimizar funciones objetivo costosas de evaluar, comúnmente encontradas en la sintonización de hiperpará-

tros en aprendizaje automático. Esta técnica se basa en principios probabilísticos y de modelado para encontrar de manera eficiente los valores óptimos de los hiperparámetros.

La Optimización Bayesiana equilibra la exploración y la explotación del espacio de búsqueda mediante el uso de funciones de adquisición, como la Entropía de Esperanza Mejorada (*Expected Improvement*) o el Área Bajo la Curva (*Area Under Curve*), que guían la búsqueda hacia regiones prometedoras del espacio de hiperparámetros.

En el contexto del aprendizaje automático, la Optimización Bayesiana se utiliza ampliamente para mejorar el rendimiento y la generalización de los modelos al encontrar los valores óptimos de los hiperparámetros. Esto se logra mediante la maximización de una función de adquisición $\alpha(x)$ que representa la utilidad esperada de evaluar el punto x en el espacio de búsqueda de hiperparámetros.

$$x^* = \arg \max_x \alpha(x)$$

Sin embargo, esta técnica también presenta desafíos, como la selección adecuada de la función de adquisición y el modelo de regresión, así como la dificultad para paralelizar su ejecución en casos donde las evaluaciones de la función objetivo son costosas.

Existen varias implementaciones populares de la Optimización Bayesiana, como SMAC (*Sequential Model-based Algorithm Configuration*) y Hyperopt, que proporcionan interfaces fáciles de usar para la Optimización Bayesiana y se pueden integrar fácilmente en flujos de trabajo de aprendizaje automático.

2.5.1. Herramientas de Optimización de Hiperparámetros

La optimización de hiperparámetros es un aspecto crucial en el desarrollo de modelos de aprendizaje automático. Consiste en encontrar la mejor combinación de hiperparámetros para un algoritmo de aprendizaje automático dado. Los hiperparámetros afectan significativamente el rendimiento y la generalización del modelo, por lo que es importante optimizarlos de manera eficiente.

La optimización manual de hiperparámetros puede ser tediosa y consume mucho tiempo. Por esta razón, se utilizan herramientas automatizadas de optimización de hiperparámetros para acelerar el proceso y mejorar los resultados. Estas herramientas ofrecen algoritmos que exploran el espacio de búsqueda de hiperparámetros de manera eficiente.

Existen diversas herramientas populares para la optimización de hiperparámetros, cada una con sus propias características y ventajas. Algunas de las herramientas más utilizadas incluyen:

- **Hyperopt**: Hyperopt es una biblioteca de optimización de hiperparámetros que se basa en algoritmos de optimización secuencial. Utiliza técnicas de optimización bayesiana para explorar eficientemente el espacio de búsqueda de hiperparámetros y encontrar la combinación óptima. Hyperopt proporciona una API flexible y fácil de usar que permite definir espacios de búsqueda para los hiperparámetros y elegir entre diferentes algoritmos de optimización, como el algo-

ritmo de árbol de búsqueda (TPE) y el algoritmo de optimización de bayesiano gaussiano (GP).

- **Optuna**: Optuna es una biblioteca de optimización de hiperparámetros que utiliza algoritmos de optimización de árboles de búsqueda de bandas. Optuna se enfoca en la eficiencia y la escalabilidad, lo que la hace adecuada para la optimización de hiperparámetros en problemas grandes y complejos. Utiliza una estrategia de muestreo eficiente para explorar el espacio de búsqueda de manera inteligente y encontrar la mejor configuración de hiperparámetros. Optuna también ofrece una interfaz fácil de usar y proporciona funciones integradas para la visualización de resultados y la gestión de experimentos.
- **SMAC3**: Sequential Model-based Algorithm Configuration (SMAC3) es un marco de trabajo de optimización de hiperparámetros que implementa la configuración secuencial basada en modelos. SMAC3 utiliza técnicas de optimización bayesiana para construir un modelo probabilístico de la función objetivo y seleccionar puntos de exploración que maximicen la información obtenida del modelo. SMAC3 es altamente configurable y permite especificar diferentes tipos de algoritmos de optimización y criterios de selección. Además, SMAC3 proporciona herramientas avanzadas para el análisis y la visualización de resultados, lo que facilita la comprensión y la interpretación de los resultados de la optimización.

Estas herramientas son ampliamente utilizadas en la comunidad de aprendizaje automático y ofrecen diferentes enfoques para la optimización de hiperparámetros, cada uno con sus propias fortalezas y debilidades. La elección de la herramienta adecuada dependerá de las necesidades específicas del proyecto, el tipo de problema y los recursos disponibles.

2.5.2. SMAC: Sequential Model-based Algorithm Configuration

SMAC (Sequential Model-based Algorithm Configuration) es un framework de optimización bayesiana utilizado para ajustar los hiperparámetros de algoritmos de aprendizaje automático y otros parámetros de configuración en problemas computacionales. La optimización bayesiana se basa en la construcción de un modelo probabilístico de la función objetivo y la selección de puntos de exploración que maximicen la información obtenida del modelo.

El objetivo de SMAC es encontrar la configuración de hiperparámetros x^* que minimice (o maximice) una función objetivo $f(x)$ costosa de evaluar. La búsqueda se realiza iterativamente, utilizando un modelo de regresión $M(x)$ para predecir el rendimiento de diferentes configuraciones de hiperparámetros.

Una formulación común en SMAC es maximizar la utilidad esperada de evaluar un punto x en el espacio de búsqueda de hiperparámetros. La utilidad esperada $U(x)$ se define como:

$$U(x) = \mathbb{E}[f(x) \,|\, x, D_{1:t}]$$

Donde $D_{1:t}$ representa los datos observados hasta el momento. La utilidad esperada se calcula utilizando la función de adquisición $\alpha(x)$, que cuantifica la información ganada al evaluar el punto x. Una función de adquisición común es la Entropía de Esperanza Mejorada (Expected Improvement):

$$\mathrm{EI}(x) = \mathbb{E}[\text{máx}(0, f(x_{\text{best}}) - f(x)) \mid x, D_{1:t}]$$

Donde $f(x_{\text{best}})$ es el mejor valor de la función objetivo observado hasta el momento. La Entropía de Esperanza Mejorada mide cuánto se espera que mejore el rendimiento si se evalúa el punto x en comparación con el mejor valor actual.

La exploración del espacio de búsqueda de hiperparámetros se realiza de manera secuencial, seleccionando iterativamente el punto x^* que maximiza la utilidad esperada. Una vez que se evalúa el punto seleccionado, se actualiza el modelo de regresión y se repite el proceso de selección.

Algunos de los parámetros importantes en SMAC son:

- **Initial Design:** La cantidad de configuraciones iniciales seleccionadas antes de comenzar la optimización.
- **Acquisition Function:** La función utilizada para seleccionar la próxima configuración a evaluar. Las opciones comunes incluyen EI, UCB (Upper Confidence Bound) y PI (Probability of Improvement).
- **Regression Model:** El modelo de regresión utilizado para modelar la relación entre las configuraciones y sus rendimientos. Los modelos comúnmente utilizados incluyen Regresión Gaussiana y Procesos Gaussianos.
- **Budget:** El número máximo de evaluaciones de la función objetivo permitidas durante la optimización.
- **Termination Criteria:** El criterio utilizado para determinar cuándo detener la optimización, como el agotamiento del presupuesto o la convergencia del modelo de regresión.

2.5.3. SMAC3

SMAC3, la tercera versión de Sequential Model-based Algorithm Configuration, es una herramienta de optimización bayesiana de código abierto desarrollada por la comunidad de AutoML. Está diseñada para abordar problemas de optimización de hiperparámetros de manera eficiente y escalable, utilizando técnicas de modelado probabilístico y exploración secuencial del espacio de búsqueda.

Una de las características clave de SMAC3 es su capacidad para manejar tanto variables de hiperparámetros continuas como discretas, lo que lo hace adecuado para una amplia gama de problemas de optimización. Además, SMAC3 ofrece una interfaz de usuario intuitiva y flexible que permite a los usuarios definir fácilmente sus espacios de búsqueda y especificar criterios de parada personalizados.

SMAC3 implementa varios algoritmos de modelado probabilístico, incluidos modelos basados en Gaussianas y árboles de regresión aleatoria, lo que le permite adaptarse a diferentes tipos de problemas y conjuntos de datos. Utiliza técnicas de infill sampling para seleccionar puntos de exploración de manera inteligente, equilibrando la exploración del espacio de búsqueda con la explotación de regiones prometedoras.

Capítulo 3

Diseño del Marco de Experimentación

Este capítulo abarca los detalles experimentales, incluyendo el diseño experimental, los problemas específicos a resolver, los hiperparámetros a optimizar, las herramientas y estrategias de implementación utilizadas, y la configuración predeterminada del algoritmo de Proximal Policy Optimization (PPO).

A lo largo de esta sección, se explica detalladamente el diseño del conjunto de experimentos describiendo la estructura y la metodología que guían el proceso. Al mismo tiempo, se detallan los problemas seleccionados para resolver, destacando los objetivos generales y los desafíos abordados en el estudio.

Además, se describen los hiperparámetros destinados a la optimización, fundamentando el enfoque sistemático adoptado para mejorar el rendimiento del modelo. Las herramientas y estrategias de implementación empleadas se detallan para ofrecer información sobre la infraestructura tecnológica y las metodologías utilizadas para llevar a cabo los experimentos de manera efectiva.

Asimismo, se proporciona una visión general de la configuración predeterminada del algoritmo PPO, ofreciendo a los lectores un entendimiento básico de los parámetros y ajustes del algoritmo.

3.1. Entorno

En el contexto del aprendizaje por refuerzo, el término entorno se refiere al mundo simulado o físico en el que opera el agente de aprendizaje. Este entorno proporciona al agente información sobre su estado actual y las consecuencias de sus acciones, así como recompensas que indican qué tan bien está cumpliendo sus objetivos. El agente toma decisiones en función de esta información para maximizar las recompensas a lo largo del tiempo.

Se seleccionaron dos entornos distintos para este estudio: CartPole y LunarLander. La elección de CartPole se basa en su simplicidad y su capacidad para ilustrar conceptos fundamentales de aprendizaje por refuerzo, a su vez, la elección de LunarLander se basa en su mayor complejidad comparada con CartPole, lo que permite evaluar el

rendimiento y la robustez de los algoritmos de aprendizaje por refuerzo en entornos más desafiantes.

3.1.1. CartPole

El entorno CartPole es un problema clásico de aprendizaje por refuerzo, en el que se simula un poste verticalmente invertido (cart-pole) unido por una articulación a un carrito móvil. El objetivo es mantener el poste en posición vertical al equilibrar el carrito, evitando que se caiga. Este entorno es interesante porque es simple pero desafiante, lo que lo convierte en un punto de partida común para experimentar con algoritmos de aprendizaje por refuerzo.

El agente en el entorno CartPole puede realizar dos acciones discretas: mover el carrito hacia la izquierda o hacia la derecha. El agente percibe el estado del entorno, que incluye la posición del carrito, la velocidad del carrito, el ángulo del poste y la velocidad angular del poste.

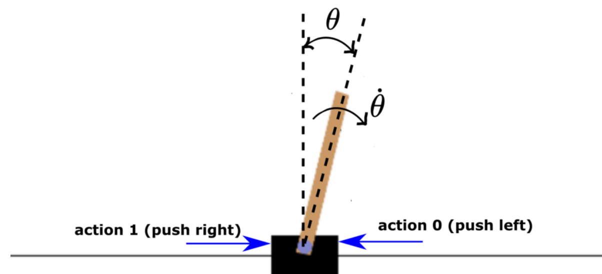


Figura 3.1: Esquema del entorno CartPole

3.1.2. LunarLander

El entorno LunarLander simula el aterrizaje de una nave espacial en la superficie lunar. El objetivo es aterrizar la nave espacial de manera segura en una región de aterrizaje designada, evitando colisiones y minimizando el uso de combustible. Este entorno es interesante porque involucra una mayor complejidad, con múltiples acciones posibles y una variedad de posibles estados y recompensas.

El agente en el entorno LunarLander puede controlar la orientación y la potencia del motor de la nave espacial, lo que se traduce en cuatro acciones posibles: no hacer nada, encender los motores hacia la izquierda, encender los motores hacia la derecha y encender los motores hacia abajo.

El agente percibe el estado del entorno, que incluye la posición y la velocidad de la nave espacial, el ángulo de orientación, la velocidad angular y la información sobre el contacto con la superficie lunar. Estas observaciones proporcionan al agente una comprensión detallada de su entorno, permitiéndole tomar decisiones informadas sobre qué acciones tomar.

En cuanto a las recompensas, el agente recibe una recompensa positiva por aterrizar la nave espacial de manera segura en la región de aterrizaje designada, evitar colisiones y minimizar el uso de combustible. Por el contrario, recibe una recompensa negativa por realizar acciones que podrían resultar en una colisión, aterrizar fuera

de la región designada o utilizar demasiado combustible. Estas recompensas proporcionan una señal de retroalimentación al agente, guiándolo hacia comportamientos deseables y ayudándolo a aprender una política de control efectiva.

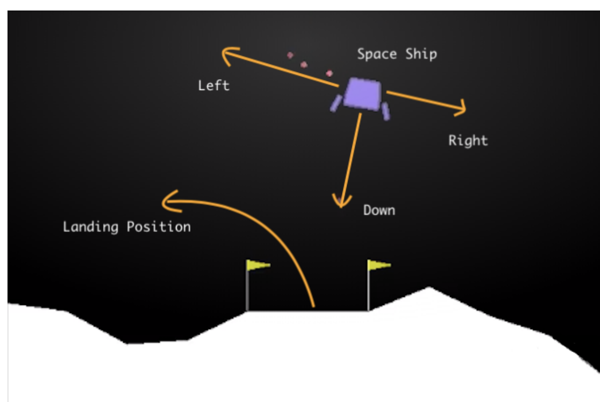


Figura 3.2: Esquema del entorno LunaLander

3.2. Herramientas utilizadas

Las herramientas y librerías utilizadas en este proceso incluyen:

- **Stable Baselines**: Una biblioteca de Python que proporciona implementaciones mejoradas de algoritmos de aprendizaje por refuerzo. Facilita la experimentación con diferentes algoritmos y la comparación de sus rendimientos en distintos entornos.

- **Gym**: Una biblioteca de Python que implementa entornos de aprendizaje por refuerzo, incluidos CartPole y LunarLander. Proporciona una interfaz estandarizada para interacciones agente-entorno, lo que facilita el desarrollo y la evaluación de algoritmos de aprendizaje por refuerzo.

- **Git**: Una herramienta de control de versiones para realizar un seguimiento de los cambios y avances en el código y los experimentos. Permite colaborar de manera efectiva en proyectos de desarrollo de software, mantener un historial de cambios y gestionar diferentes versiones del proyecto.

- **Visual Studio Code**: Un entorno de desarrollo integrado utilizado para la escritura y depuración del código. Ofrece características avanzadas como resaltado de sintaxis, depuración interactiva y extensiones que mejoran la productividad del desarrollo.

- **SMAC**: Una biblioteca para la optimización de hiperparámetros. Incluye la clase Scenario para definir el espacio de búsqueda y configuraciones, la clase MultiFidelityFacade de smac para facilitar la optimización con presupuestos variables y intensificadores como Hyperband para la asignación eficiente de recursos entre configuraciones y SuccessiveHalving para reducir iterativamente el número de configuraciones evaluadas.

3.3. Configuración de la optimización bayesiana

En esta sección se detalla la configuración y organización de los archivos de la optimización bayesiana implementada en este proyecto. Cada versión del proyecto, con o sin multifidelidad, contiene varias carpetas y archivos específicos, los documentos y archivos necesarios para este proceso se encuentran organizados de la siguiente manera:

- La carpeta _pycache_ se genera automáticamente por Python y contiene archivos compilados en bytecode, lo que permite una ejecución más rápida del código.
- La carpeta *logs* almacena los registros relacionados con los últimos entrenamientos de cada entorno, incluyendo información detallada como las configuraciones usadas, la longitud media de los episodios (*ep_len_mean*), la recompensa media de los episodios (*ep_rew_mean*), el número de cuadros por segundo (*fps*), el número de iteraciones, el tiempo transcurrido, y el número total de pasos de tiempo(*total_timesteps*).
- La carpeta *models* guarda los modelos de los agentes entrenados para ser validados posteriormente.
- La carpeta *smac3_output* es generada automáticamente por SMAC y contiene los resultados y configuraciones de las ejecuciones de optimización.
- El archivo *genericSolver.py* contiene la lógica interna del entrenamiento y validación de los agentes. Aquí se declaran variables como el entorno usado, el valor mínimo y máximo del presupuesto para la multifidelidad, el número mínimo y máximo del total de pasos a dar por el agente durante su entrenamiento, el tamaño del lote y el tiempo que se va a querer entrenar el agente. Este archivo contiene tambien el espacio de configuraciones de hiperparámetros mediante la función *configspace*, la función de validación de un agente específico *evaluate_agent* y la función de entrenamiento de la optimización bayesiana *train*. Además, en este archivo se define una red neuronal personalizada *CustomFeatureExtractor* que se utiliza en la política de PPO.
- El archivo *Logger.py* se encarga de imprimir en los registros la información del entrenamiento para posteriormente ser guardados en la carpeta logs.
- Finalmente, el archivo *Optimizer.py* declara el entorno, el escenario y SMAC para entrenarlo, valida todos los agentes entrenados, y permite ver la mejor configuración probada en el último entrenamiento renderizando el entorno.

3.3.1. Adaptación para optimización sin multifidelidad

Los hiperparámetros que se optimizarán son la tasa de aprendizaje (*learning rate*), el factor de descuento (*discount factor*) y lambda (*gae_lambda*). Estos hiperparámetros son cruciales en el contexto de CartPole y especialmente LunarLander debido a su impacto directo en el rendimiento del agente.

La **tasa de aprendizaje** controla la magnitud de los ajustes que realiza el algoritmo de aprendizaje en los pesos de la red neuronal durante el entrenamiento. Una tasa de aprendizaje demasiado alta puede llevar a una convergencia rápida pero inestable, mientras que una tasa de aprendizaje demasiado baja puede hacer que el entrenamiento sea lento o incluso conducir a una convergencia subóptima. Por lo tanto,

encontrar el valor óptimo de la tasa de aprendizaje es crucial para garantizar un aprendizaje eficaz y estable en CartPole.

El **factor de descuento** determina la importancia relativa de las recompensas futuras en comparación con las recompensas inmediatas. Un factor de descuento alto prioriza las recompensas a largo plazo, lo que puede ser beneficioso en entornos donde las acciones tienen consecuencias a largo plazo, como CartPole. Por otro lado, un factor de descuento bajo puede llevar a una toma de decisiones miope y una búsqueda de recompensas a corto plazo. Por lo tanto, optimizar el factor de descuento es esencial para garantizar que el agente en CartPole considere adecuadamente las recompensas futuras al tomar decisiones.

En el contexto del aprendizaje por refuerzo, **lambda** se refiere al parámetro de *Generalized Advantage Estimation* (GAE), una técnica que se utiliza para reducir la varianza de las estimaciones de la ventaja en los métodos basados en políticas, como el algoritmo de Actor-Critic. ion (SMAC). En el cual se configura y ejecuta el proceso de optimización de hiperparámetros utilizando SMAC.

Para implementar la solución de optimización bayesiana de hiperparámetros, se diseñó el archivo `optimizer.py` con la ayuda de la clase HyperparameterOptimizationFacade (HPOFacade) de SMAC (Sequential Model-based Algorithm Configuration). El archivo se centra en mejorar el rendimiento de agentes entrenados con el algoritmo PPO (Proximal Policy Optimization) en entornos de simulación como CartPole y LunarLander.

3.3.1.1. Configuración y Parámetros Iniciales

Para ello se utiliza una serie de parámetros y configuraciones definidos en el archivo `genericSolver`. Estos parámetros incluyen:

- Env: Define el entorno de simulación a utilizar (CartPole o LunarLander).
- Early_Stopping: Tiempo máximo permitido para la optimización, después del cual se detiene el proceso.

3.3.1.2. Funciones Principales

- **Renderización del Agente**:

  Esta función carga un modelo entrenado desde un directorio especificado y lo ejecuta en el entorno de simulación (CartPole o LunarLander) para un número dado de episodios. El agente, utilizando el modelo PPO, predice y ejecuta acciones, mostrando el comportamiento en pantalla.

- **Validación de Agentes**:

  Esta función evalúa múltiples modelos de agentes guardados en una carpeta específica. Carga cada modelo y lo evalúa en términos de la recompensa promedio obtenida en un número definido de episodios. Luego selecciona y devuelve el modelo con el mejor rendimiento. Además, genera y guarda gráficos de las recompensas obtenidas por el mejor modelo.

3.3.1.3. Proceso Principal de Optimización

En la sección principal del archivo (`__main__`), se lleva a cabo el proceso de optimización de hiperparámetros y la validación del agente:

1. **Registro de Logs**:
   - Se inicializa un logger para registrar el proceso de optimización.
2. **Inicialización del Modelo y Escenario de Optimización**:
   - Se crea una instancia de `GenericSolver`, que contiene la configuración del espacio de hiperparámetros a optimizar.
   - Se configura un escenario de SMAC, especificando los parámetros de la optimización, como si el proceso es determinista, la semilla, el número máximo de pruebas, y el límite de tiempo.
3. **Ejecutar la Optimización**:
   - Se instancia `HPOFacade` con el escenario y la función objetivo (el método de entrenamiento del modelo).
   - Se ejecuta el proceso de optimización y se guarda la mejor configuración encontrada.
4. **Validación y Renderización del Mejor Modelo**:
   - Se valida el rendimiento de los modelos entrenados utilizando la función `agents_validation`.
   - Se renderiza el comportamiento del mejor modelo para visualizar su desempeño en el entorno de simulación.
5. **Limpieza**:
   - Opcionalmente, se puede eliminar la carpeta que contiene los modelos entrenados para liberar espacio y evitar validar modelos antiguos.

El archivo `genericSolver.py`, se centra en el entrenamiento y evaluación de un agente de Proximal Policy Optimization (PPO) en entornos de simulación como CartPole y LunarLander. Este archivo proporciona una estructura completa para la configuración, entrenamiento y evaluación de un agente PPO en entornos de simulación, con una red neuronal personalizada para mejorar el rendimiento del agente. También define un espacio de búsqueda de hiperparámetros que puede ser utilizado por herramientas de optimización como SMAC para encontrar los mejores valores de hiperparámetros.

3.3.1.4. Parámetros y Configuración

- El archivo contiene configuraciones clave:
  1. Env: Define el entorno de simulación a utilizar (CartPole o LunarLander).
  2. Total_timesteps: Número total de pasos de entrenamiento de un agente.
  3. Early_Stopping: Límite de tiempo máximo permitido para el entrenamiento.

3.3.1.5. Clases y Funciones Principales

1. CustomFeatureExtractor:
   - Clase que define una red neuronal personalizada para extraer características de las observaciones del entorno. La clase define una red neuronal con varias capas, que incluyen capas lineales, activaciones ReLU, normalización por lotes y dropout para prevenir el sobreajuste. Esta red toma las observaciones del entorno y las procesa para extraer características relevantes que se utilizan posteriormente en la política del agente.
2. CustomMLPPolicy:
   - Clase que define una política de red neuronal personalizada basada en la arquitectura MLP (Multi-Layer Perceptron). Esta clase utiliza la clase CustomFeatureExtractor para extraer características de las observaciones y emplea una arquitectura MLP para decidir las acciones del agente.
3. GenericSolver:

   Clase principal que define el espacio de búsqueda de hiperparámetros y proporciona métodos para entrenar y evaluar el agente. Incluye varios métodos:
   - **configspace**: Propiedad que define el espacio de búsqueda de hiperparámetros para la optimización. Incluye los siguientes parámetros:
     - *learning_rate*: Tasa de aprendizaje, con valores que oscilan entre 1×10^{-4} y 1×10^{-2} en una escala logarítmica, con un valor por defecto de 1×10^{-3}.
     - *discount_factor*: Factor de descuento, con valores que varían entre 0.9 y 0.999, con un valor por defecto de 0.99.
     - *gae_lambda*: Parámetro GAE lambda, con valores que oscilan entre 0.9 y 0.999, con un valor por defecto de 0.95.
   - **evaluate_agent**: Función que evalúa el rendimiento de un agente en el entorno especificado durante un solo episodio, devolviendo la recompensa total obtenida.
   - **plot_training**: Función que genera y guarda un gráfico del progreso del entrenamiento, mostrando las recompensas medias por actualización.
   - **train**: Método principal que entrena un agente PPO con una configuración de hiperparámetros específica. Este método configura el entorno de simulación, define los parámetros del agente PPO (*ppo_params*), que incluyen:
     - *policy*: Política a utilizar, puede ser CustomMLPPolicy o 'MlpPolicy' en caso de que queramos una red básica con 2 capas ocultas de 64 neuronas.
     - *env*: Entorno de simulación.
     - *learning_rate*: Tasa de aprendizaje definida en la configuración.
     - *gamma*: Factor de descuento definido en la configuración.
     - *n_steps*: Número de pasos antes de actualizar la política (2048).
     - *batch_size*: Tamaño del lote de entrenamiento (64).

- *n_epochs*: Número de épocas para optimizar cada lote de datos (10).
- *gae_lambda*: Parámetro GAE lambda definido en la configuración.
- *clip_range*: Rango de recorte (0.2).
- *ent_coef*: Coeficiente de la pérdida de entropía (0.0).
- *vf_coef*: Coeficiente de la pérdida del valor (0.5).
- *max_grad_norm*: Máximo valor permitido para la norma del gradiente (0.5).
- *verbose*: Nivel de verbosidad (1).

Entrena y evalúa periódicamente al agente utilizando los parámetros definidos y la configuración de hiperparámetros, se guarda el modelo entrenado en el directorio *models*. Además, genera gráficos del progreso del entrenamiento y los guarda en el directorio *plots*.

3.3.2. Adaptación para optimización bayesiana con multifidelidad

En la versión con multifidelidad, se han realizado los siguientes cambios y adiciones para optimizar el rendimiento del agente utilizando diferentes niveles de fidelidad durante la búsqueda de hiperparámetros:

3.3.2.1. Funciones Principales Adicionales

- **Renderización del Agente**: La función sigue cargando un modelo entrenado desde un directorio especificado y ejecutándolo en el entorno de simulación (CartPole o LunarLander) para un número dado de episodios, utilizando el modelo PPO.
- **Validación de Agentes**:

  Esta función evalúa múltiples modelos de agentes guardados en una carpeta específica, cargando cada modelo y evaluándolo en términos de la recompensa promedio obtenida en un número definido de episodios. Se selecciona y devuelve el modelo con el mejor rendimiento. Además, genera y guarda gráficos de las recompensas obtenidas por el mejor modelo.

3.3.2.2. Proceso Principal de Optimización con Multifidelidad

En la sección principal del archivo (`__main__`), se lleva a cabo el proceso de optimización de hiperparámetros con multifidelidad y la validación del agente:

1. **Registro de Logs**:
   - Se inicializa un logger para registrar el proceso de optimización.
2. **Inicialización del Modelo y Escenario de Optimización**:
   - Se crea una instancia de `GenericSolver`, que contiene la configuración del espacio de hiperparámetros a optimizar y se configura un escenario de SMAC, especificando los parámetros de la optimización, como si el proceso

es determinista, la semilla, el número máximo de pruebas, y el límite de tiempo.

3. **Ejecutar la Optimización con Multifidelidad**:
   - Se instancia `MFFacade` con el escenario y la función objetivo (el método de entrenamiento del modelo) y se realiza un diseño inicial de 5 pruebas para así poder tener unos experimentos de referencia. Posteriormente, se establece un intensificador `Hyperband` que se encarga de explorar un numero considerable de configuraciones con presupuestos bajos y eliminar aquellos que tengan un mal rendimiento bajo. Finalmente, se ejecuta el proceso de optimización utilizando diferentes niveles de fidelidad y se guarda la mejor configuración encontrada.
4. **Validación y Renderización del Mejor Modelo**:
   - Se valida el rendimiento de los modelos entrenados utilizando la función `agents_validation`y se puede renderizar el comportamiento del mejor modelo para visualizar su desempeño en el entorno de simulación.
5. **Limpieza**:
   - Opcionalmente, se puede eliminar la carpeta que contiene los modelos entrenados para liberar espacio y evitar validar modelos antiguos.

El archivo GenericSolver.py también sufrío algunas variaciones para asi adaptar su comportamiento a la versión con multifidelidad.

- **Importación de Módulos**:
  - Se importa `make_vec_env` desde `stable_baselines3.common.env_util` para crear entornos vectorizados. Esto es necesario para aprovechar mejor los recursos de computación y realizar múltiples ejecuciones en paralelo, lo que es útil cuando se manejan diferentes presupuestos de entrenamiento.
- **Definición de Constantes**:
  - Se definen `MIN_BUDGET` y `MAX_BUDGET` para establecer los límites del presupuesto en las evaluaciones de SMAC. Esto permite que la optimización considere diferentes duraciones de entrenamiento.
  - Se definen `MIN_TIMESTEPS` y `MAX_TIMESTEPS` para limitar el rango de los pasos de tiempo totales en función del presupuesto. Esto garantiza que la cantidad de entrenamiento varíe de manera controlada según el presupuesto disponible.
- **Método train**:
  - Se añade un parámetro adicional `budget` al método para permitir la optimización multifidelidad, donde diferentes presupuestos pueden influir en la cantidad de entrenamiento. Esto hace posible ajustar la cantidad de recursos computacionales utilizados durante el entrenamiento basado en las necesidades específicas del experimento.
  - Se calcula `total_timesteps` basado en el presupuesto dado:

```
int(((budget - MIN_BUDGET) / (MAX_BUDGET - MIN_BUDGET)) *
(MAX_TIMESTEPS - MIN_TIMESTEPS) + MIN_TIMESTEPS)
```

Este cálculo traduce el presupuesto en una cantidad correspondiente de pasos de tiempo de entrenamiento, asegurando que se utilicen recursos proporcionales al presupuesto asignado.

- El nombre del archivo del modelo guardado se actualiza con su timestamp para diferenciarlo de los modelos entrenados sin consideraciones de presupuesto.

3.3.3. Métricas de evaluación

En este estudio, las métricas de evaluación se centran en medir el rendimiento de los agentes de aprendizaje por refuerzo entrenados con las configuraciones de hiperparámetros optimizadas. Dado que las ejecuciones se limitan a un tiempo máximo y se selecciona el algoritmo que obtenga la mejor recompensa, las métricas se enfocan en la recompensa promedio por episodio y en la estabilidad del agente en la resolución del problema, midiendose esta mediante la desviación típica.

Estas métricas nos permiten determinar qué algoritmo de optimización de hiperparámetros proporciona las mejores configuraciones para el entrenamiento de agentes de aprendizaje por refuerzo en los entornos de prueba seleccionados.

3.4. Procedimiento experimental

El procedimiento experimental constará de los siguientes pasos:

1. Preparación de los entornos de prueba (CartPole y LunarLander).
2. Definición de los espacios de búsqueda de hiperparámetros para cada algoritmo de optimización.
3. Ejecución de los algoritmos HPO-SMAC y MF-SMAC para la optimización de hiperparámetros en cada entorno de prueba.
4. Entrenamiento de agentes de aprendizaje por refuerzo utilizando las mejores configuraciones de hiperparámetros encontradas por cada algoritmo.
5. Evaluación del rendimiento de los agentes entrenados en los entornos de prueba.
6. Análisis comparativo de los resultados obtenidos para determinar la eficacia de los algoritmos de optimización.

Capítulo 4

Resultados de la experimentación

4.1. Introducción

En esta sección, presentamos los resultados de nuestros experimentos con el objetivo de evaluar el rendimiento de un agente de Proximal Policy Optimization (PPO) en los entornos de CartPole y LunarLander. Estos experimentos se han llevado a cabo utilizando dos enfoques diferentes: una versión básica y una versión con optimización multifidelidad.

La versión básica del experimento sigue un enfoque tradicional en el cual se entrena al agente utilizando un conjunto fijo de hiperparámetros y una cantidad constante de recursos computacionales. Por otro lado, la versión con multifidelidad introduce variabilidad en los presupuestos de entrenamiento, permitiendo ajustar la cantidad de recursos utilizados según diferentes configuraciones presupuestarias.

Las gráficas que se mostrarán a continuación ilustran la recompensa acumulada por el agente durante el entrenamiento y las evaluaciones realizadas. Es importante destacar que estas evaluaciones se llevan a cabo a intervalos regulares definidos por la frecuencia de evaluación especificada (eval_freq). En nuestro caso, la frecuencia de evaluación se determina dividiendo el número total de pasos de entrenamiento (TOTAL_TIMESTEPS) por 100, lo que garantiza que las evaluaciones se realicen periódicamente durante el proceso de entrenamiento. Debido a esta metodología, los resultados de las evaluaciones siempre son múltiplos de la frecuencia de evaluación, lo que significa que, aunque el agente alcance el umbral de recompensa antes, la ejecución siempre se completará en el próximo punto de evaluación. Esto explica por qué no existen decimales en los resultados relacionados con el estudio de la convergencia de los mismos.

Las gráficas de las evaluaciones del agente muestran en el eje X los episodios ejecutados, y en el eje Y la recompensa acumulada obtenida. Estas visualizaciones nos permiten comparar de manera efectiva los rendimientos alcanzados en ambos enfoques y analizar las ventajas y desventajas de cada uno.

A continuación, se presentan los resultados obtenidos para ambas versiones en los dos entornos estudiados, con las configuraciones resultantes redondeadas a 5 decimales y proporcionando comparaciones detalladas de los rendimientos alcanzados.

4.2. Experimentos de la versión de optimización bayesiana básica en CartPole

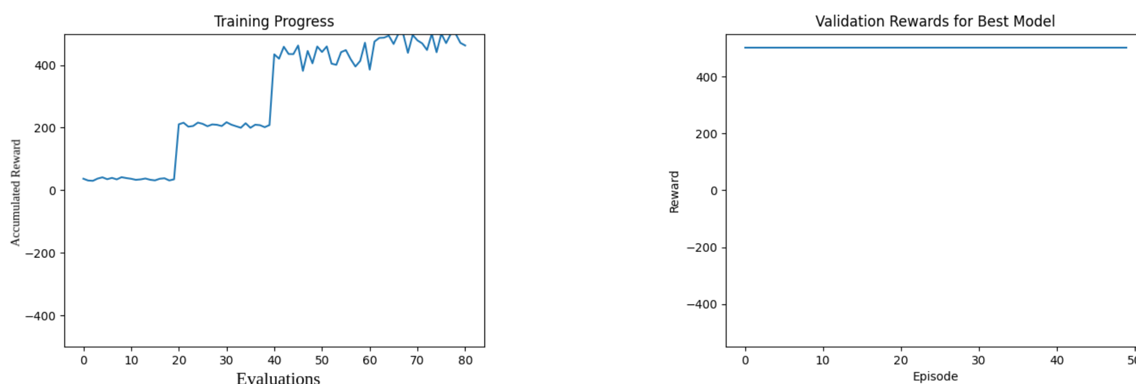
Experimento 1: Experimento Base

CartPole, al ser un entorno muy simple donde solo se pueden tomar dos acciones y que tiene un espacio de estados reducido, permite alcanzar una solución óptima de manera muy rápida. Generalmente, en cuestión de minutos se puede obtener una recompensa de +500, que es la recompensa máxima en este entorno. La recompensa mínima en CartPole es de -500 y la máxima de 500. Dado que es un entorno fácil y que requiere poco tiempo de entrenamiento, existen muchas configuraciones de hiperparámetros de entrenamiento que pueden alcanzar esta recompensa máxima.

En nuestros experimentos con la versión básica de optimización bayesiana, se utilizó un criterio de *early stopping* de 180 segundos y un número de pasos de entrenamiento de 10,000. Con estos parámetros, se obtuvo la siguiente configuración de hiperparámetros que permitió al agente entrenar y conseguir, tras su evaluación, una recompensa de 500:

- **Factor de descuento:** 0.95599,
- **GAE Lambda:** 0.96237,
- **Tasa de aprendizaje:** 0.00033.

A continuación, se presentan las gráficas del rendimiento del agente durante el entrenamiento y la evaluación:



(a) Rendimiento del agente durante el entrenamiento

(b) Rendimiento del agente durante la evaluación

Figura 4.1: Rendimiento del agente en CartPole con la versión básica de optimización bayesiana

Las gráficas [Figura 4.1] muestran cómo el agente alcanza rápidamente la recompensa máxima de 500, confirmando que CartPole es un entorno donde es posible obtener buenos resultados con múltiples configuraciones de hiperparámetros.

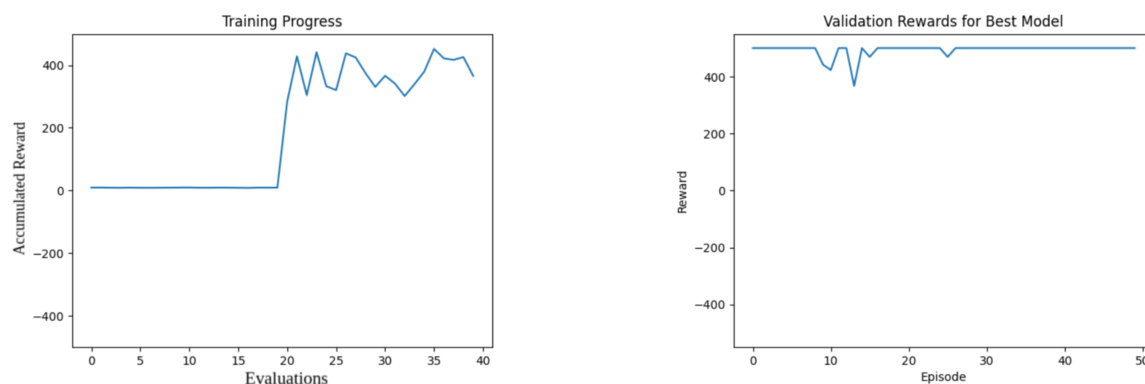
Experimento 2: Reducción del Número de Pasos

Para experimentos donde se desee reducir el número de pasos a uno inferior a 5000, el agente no muestra consistencia en su rendimiento ya que no puede entrenar lo

suficiente. Se realizó un experimento donde se llevó a cabo la optimización bayesiana durante 180 segundos con un número de pasos totales en el entrenamiento del agente de 2500, obteniéndose la siguiente configuración:

- **Factor de descuento:** 0.94213,
- **GAE Lambda:** 0.96432,
- **Tasa de aprendizaje:** 0.00230.

A continuación, se presentan las gráficas del rendimiento del agente durante el entrenamiento y la evaluación:



(a) Rendimiento del agente durante el entrenamiento

(b) Rendimiento del agente durante la evaluación

Figura 4.2: Rendimiento del agente en CartPole reduciendo el número de pasos con la versión básica de optimización bayesiana

Las gráficas [Figura 4.2] muestran que, con un número de pasos reducido, el agente no logra alcanzar la consistencia en su rendimiento y, por lo tanto, no alcanza la recompensa máxima de manera confiable, aunque sí se aproxima a una configuración aceptable con reward promedio de validación de 487.4. Esto evidencia la necesidad de un número adecuado de pasos de entrenamiento para que el agente pueda aprender y desempeñarse de manera óptima en el entorno CartPole.

Experimento 3: Estudio de la convergencia

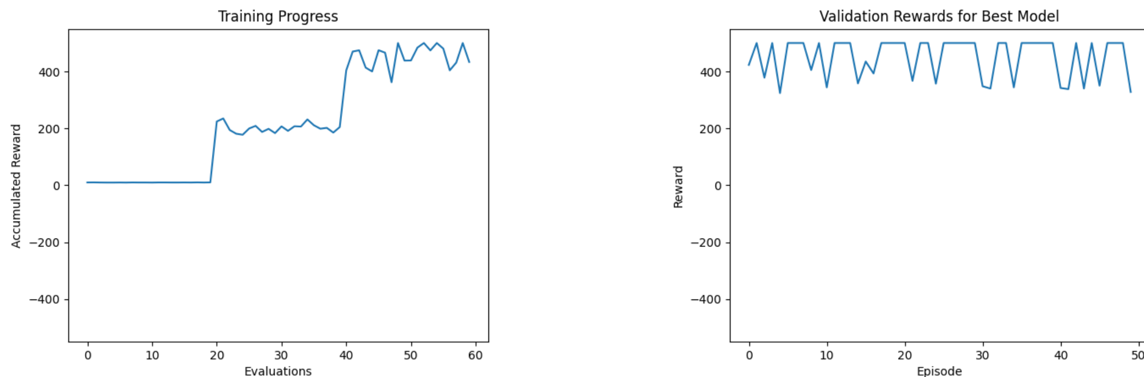
Para este experimento, se modificó el archivo *GenericSolver* para que el entrenamiento se detuviese en cuanto la recompensa alcanzara consecutivamente un valor superior a un umbral predefinido. Se estableció un umbral de 350 y un tiempo máximo de 3 minutos (180 segundos) para el early stopping. El número de pasos máximos se fijó en 10000, aunque, como se mencionó anteriormente, este número de pasos es a menudo inferior en la mayoría de los entrenamientos de los agentes debido a la rápida convergencia. Además, se ajustó el código para que, en lugar de minimizar la recompensa, se minimizara el tiempo de entrenamiento.

El resultado obtenido es el siguiente:

- **Factor de descuento:** 0.94213,
- **GAE Lambda:** 0.93638,

- **Tasa de aprendizaje:** 0.00026.

Las siguientes gráficas [Figura 4.3] muestran, el rendimiento del agente y evalución de la mejor configuración obtenida durante el entrenamiento, observándose un recorte en el número de actualizaciones debido a la parada anticipada antes de completar los 10000 pasos en 6000 pasos, ya que se paró la ejecucción en la evaluación 60 de 100 previstas.



(a) Rendimiento del agente durante el entrenamiento

(b) Rendimiento del agente durante la evaluación

Figura 4.3: Estudio de la convergencia en CartPole con la versión básica de optimización bayesiana

Los resultados demuestran que, utilizando un umbral de recompensa y la implementación de early stopping, se puede lograr una convergencia mucho más rápida en el entorno de CartPole aunque sacrificando maximizar la recompensa. La optimización enfocada en minimizar el tiempo de entrenamiento en lugar de la recompensa maximiza la eficiencia del proceso, permitiendo al agente alcanzar la recompensa deseada en menos tiempo. Esta estrategia es especialmente útil en escenarios donde el tiempo de entrenamiento es crítico, mostrando que es posible obtener configuraciones de hiperparámetros efectivas que no solo logran la recompensa objetivo sino que también reducen significativamente el tiempo necesario para entrenar al agente.

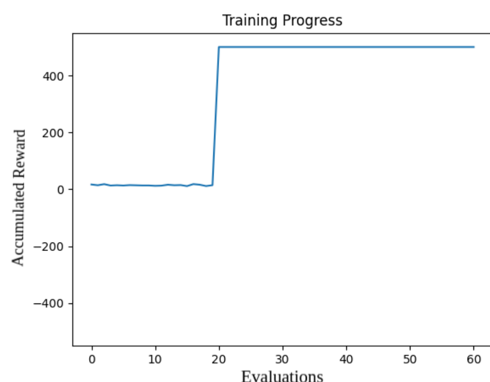
4.3. Experimentos de la versión de optimización bayesiana con multifidelidad en CartPole

Experimento 1: Experimento Base

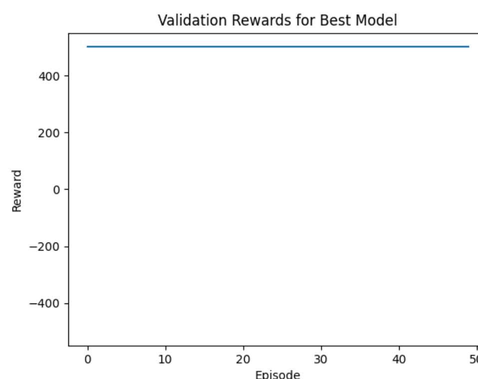
CartPole es un entorno ideal para explorar estrategias de optimización avanzadas como la multifidelidad debido a su simplicidad y rápida convergencia hacia una solución óptima. En estos experimentos, se empleó optimización bayesiana con multifidelidad para descubrir configuraciones de hiperparámetros efectivas utilizando un presupuesto limitado de evaluaciones costosas.

Se configuró un tiempo máximo de 180 segundos para el proceso de optimización y se redujo el número total de pasos de entrenamiento del agente a un máximo de 10000 y mínimo de 5000 dependiendo del presupuesto dado. A continuación se presentan los resultados obtenidos:

- **Factor de descuento:** 0.95418,
- **GAE Lambda:** 0.95975,
- **Tasa de aprendizaje:** 0.00532.



(a) Rendimiento del agente durante el entrenamiento



(b) Rendimiento del agente durante la evaluación

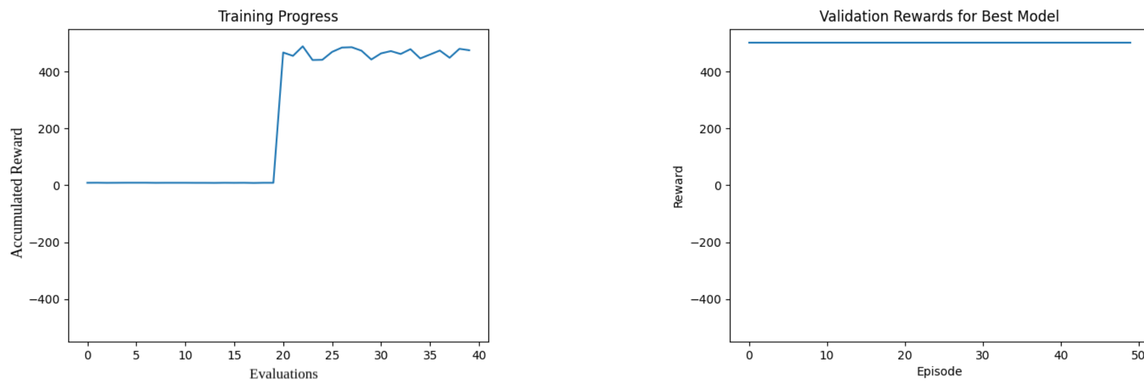
Figura 4.4: Rendimiento del agente en CartPole con la versión con multifidelidad de optimización bayesiana

Estos resultados [Figura 4.4] muestran que la optimización bayesiana con multifidelidad no mejora la eficiencia de la búsqueda de hiperparámetros en entornos simples como CartPole. Debido a la simplicidad del entorno y al buen desempeño de la versión básica, la búsqueda de hiperparámetros logra resultados óptimos en un tiempo relativamente corto, limitando el impacto de técnicas más avanzadas como la multifidelidad.

Experimento 2: Reducción del Número de Pasos

Sin embargo, para ejecuciones con numero de pasos totales menores del orden de 2500 pasos máximos y mínimos de 2000 y con un early_stopping de 180 segundos se obtuvo la siguiente configuración:

- **Factor de descuento:** 0.95418,
- **GAE Lambda:** 0.95975,
- **Tasa de aprendizaje:** 0.00532.



(a) Rendimiento del agente durante el entrenamiento

(b) Rendimiento del agente durante la evaluación

Figura 4.5: Rendimiento del agente con número bajo de pasos en CartPole con la versión con multifidelidad de optimización bayesiana

Estos resultados [Figura 4.5] muestran cómo la optimización bayesiana con multifidelidad mejora la eficiencia en un 3 % a la versión sin multifidelidad de la búsqueda de hiperparámetros en entornos simples como CartPole debido a la posibilidad de probar un número mayor de configuraciones en la franja de tiempo determinada, pudiendo así alcanzar una mejor configuración. Al realizar múltiples evaluaciones con un presupuesto limitado de pasos de entrenamiento con respecto al de la versión sin multifidelidad, se logró descubrir configuraciones de hiperparámetros que mejoran ligeramente el rendimiento del agente.

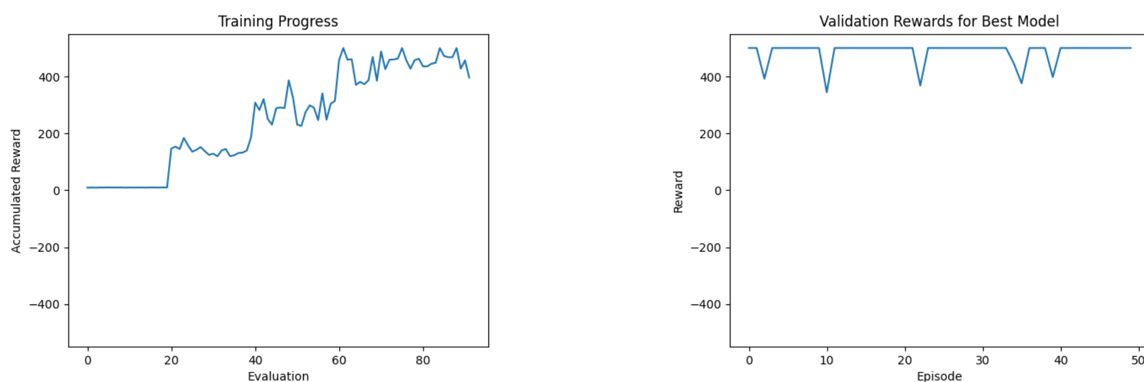
Experimento 3: Estudio de la convergencia

Para este experimento, se modificó el archivo *GenericSolver* para que el entrenamiento se detuviese en cuanto la recompensa alcanzara consecutivamente un valor superior a un umbral predefinido. Se estableció un umbral de 350 y un tiempo máximo de 3 minutos (180 segundos) para el early stopping. El número de pasos máximos se fijó en 10000 si obtuviese el mayor buget y 5000 en caso de obtener el mínimo, aunque, como se mencionó anteriormente, este número de pasos es a menudo inferior en la mayoría de los entrenamientos de los agentes debido a la rápida convergencia. Además, se ajustó el código para que, en lugar de minimizar la recompensa, se minimizara el tiempo de entrenamiento.

El resultado obtenido es el siguiente:

- **Factor de descuento:** 0.95547,
- **GAE Lambda:** 0.93638,
- **Tasa de aprendizaje:** 0.00016.

Las siguientes gráficas [Figura 4.6] muestran el rendimiento del agente con la mejor configuración obtenida durante el entrenamiento, observándose un recorte en el número de actualizaciones debido a la parada anticipada antes de completar los 10000 pasos, logrando converger en el mínimo de pasos posible, 8000 pasos, lo que supone un 20 % menos de pasos necesarios del total, y la validación de dicho agente:



(a) Rendimiento del agente durante el entrenamiento

(b) Rendimiento del agente durante la evaluación

Figura 4.6: Estudio de la convergencia en CartPole con la versión con multifidelidad de optimización bayesiana

Los resultados [Figura 4.6] demuestran que, utilizando un umbral de recompensa y la implementación de early stopping, se puede lograr una convergencia mucho más rápida en el entorno de CartPole, alcanzando una recompensa alta de 476 durante la validación. La optimización enfocada en minimizar el tiempo de entrenamiento en lugar de la recompensa maximiza la eficiencia del proceso, permitiendo al agente alcanzar la recompensa deseada en menos tiempo.

4.4. Experimentos de la versión de optimización bayesiana básica en LunarLander

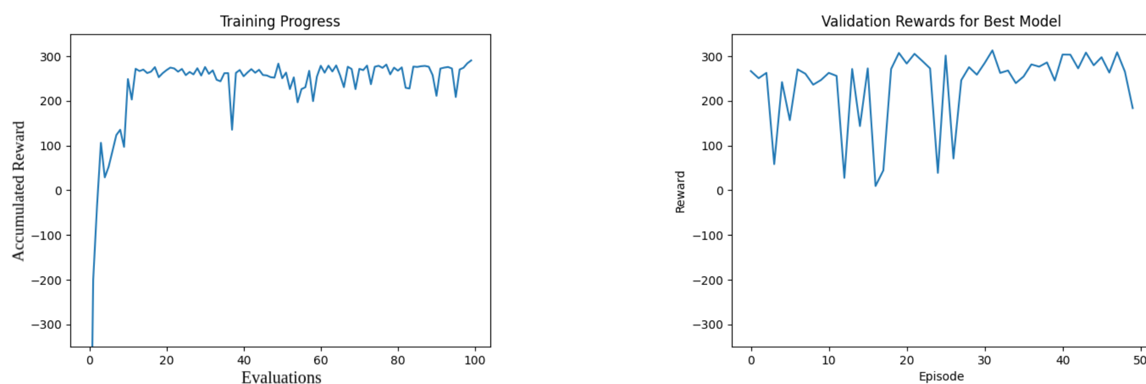
Experimento 1: Experimento Base

LunarLander es un entorno mucho más complejo que CartPole, en el que el agente debe aprender muchas más situaciones y acciones posibles en cada instante. Por eso, el aprendizaje es más complicado, lento y costoso computacionalmente. Para realizar este primer experimento, se configuró el proceso de optimización con una duración de 3 horas (10800 segundos) y se asignó un número total de pasos más elevado que en CartPole, alcanzando 1000000 pasos. Esto permitiría al agente entrenar con suficientes pasos para comprobar si la configuración es efectiva.

Tras este entrenamiento, la configuración obtenida fue:

- **Factor de descuento:** 0.95547,
- **GAE Lambda:** 0.93638,
- **Tasa de aprendizaje:** 0.00016.

Esta configuración mostró el siguiente rendimiento durante el entrenamiento y la validación, como se observa en las siguientes gráficas:



(a) Rendimiento del agente durante el entrenamiento

(b) Rendimiento del agente durante la evaluación

Figura 4.7: Rendimiento del agente en LunarLander con la versión básica de optimización bayesiana

Se probaron un total de 8 configuraciones, siendo esta la mejor, con un reward promedio en validación de 239. Este resultado indica que, aunque el entorno de Lunar-Lander es considerablemente más complejo, es posible encontrar configuraciones de hiperparámetros efectivas utilizando la optimización bayesiana, logrando un rendimiento sólido tras un proceso de entrenamiento extenso. Las gráficas demuestran que el agente fue capaz de aprender y mejorar su rendimiento, aunque el proceso de optimización requirió un tiempo y un número de pasos considerablemente mayor comparado con CartPole.

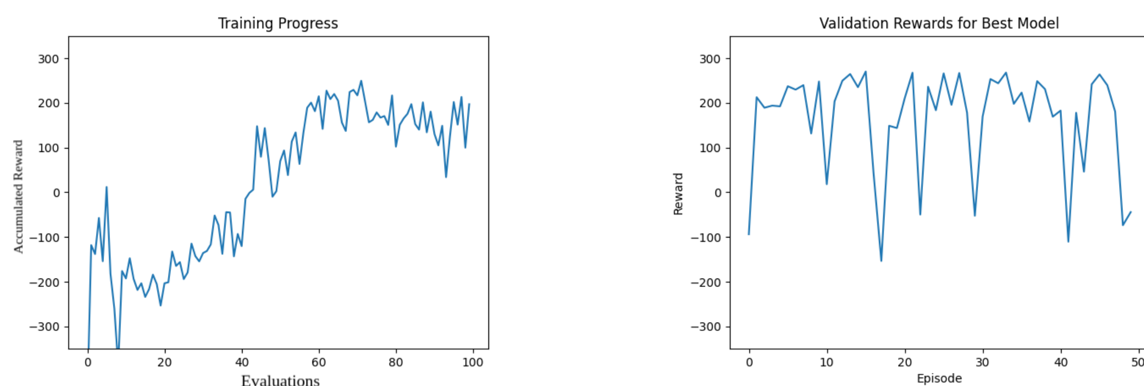
Experimento 2: Reducción del Número de Pasos

Realizar experimentos acortando el número de pasos nos permite observar cómo le cuesta al agente alcanzar recompensas similares a las obtenidas anteriormente de manera estable. En este experimento, se redujo en un 75 % el número de pasos total, manteniendo el tiempo de parada en 3 horas segundos (10800 segundos). Esta reducción forzó al agente a converger en una solución estable más rápidamente, lo que resultó ser una tarea difícil.

La configuración obtenida fue:

- **Factor de descuento:** 0.96762,
- **GAE Lambda:** 0.96326,
- **Tasa de aprendizaje:** 0.00056.

El rendimiento del agente durante el entrenamiento y la evaluación se muestra en las siguientes gráficas:



(a) Rendimiento del agente durante el entrenamiento

(b) Rendimiento del agente durante la evaluación

Figura 4.8: Rendimiento del agente con número bajo de pasos en LunarLander con la versión básica de optimización bayesiana

La validación obtuvo una recompensa media de 163, lo cual, aunque no es un mal resultado, indica que hay margen para mejorar. Se espera que, en experimentos similares utilizando optimización bayesiana con multifidelidad, se pueda alcanzar un rendimiento superior. Las gráficas reflejan la insuficiencia del entrenamiento, con episodios donde el agente presenta un buen desempeño alternados con otros en los que la recompensa es negativa debido a fallos rápidos. Esto sugiere que con más pasos de entrenamiento se podría lograr un modelo mucho más estable. La reducción en el número de pasos totales destaca la dificultad de obtener un rendimiento consistente y elevado en entornos complejos como LunarLander.

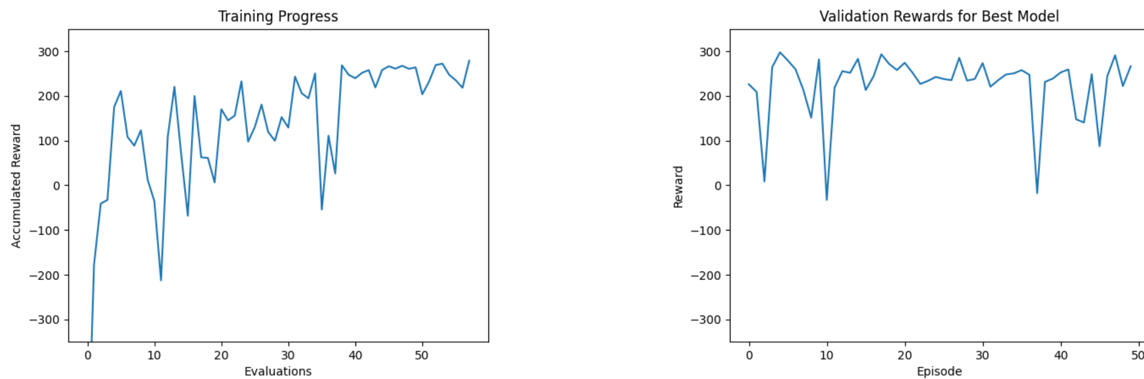
Experimento 3: Estudio de la Convergencia

Para este experimento, se utilizó el archivo *convergence_genericSolver.py* para que el entrenamiento se detuviese en cuanto la recompensa alcanzara consecutivamente 3 veces un valor superior a un umbral predefinido. Se estableció un umbral de 200 de recompensa (30 % menos del máximo) durante el entrenamiento y un tiempo máximo de 3 horas (10800 segundos) para el early stopping. El número de pasos máximos se fijó en 1000000, aunque, como se mencionó anteriormente, este número de pasos es a menudo inferior en la mayoría de los entrenamientos de los agentes debido a la rápida convergencia. Además, se ajustó el código para que, en lugar de minimizar la recompensa, se minimizara el tiempo de entrenamiento.

El resultado obtenido es el siguiente:

- **Factor de descuento:** 0.94752,
- **GAE Lambda:** 0.94827,
- **Tasa de aprendizaje:** 0.00031.

Las siguientes gráficas [Figura 4.9] muestran el rendimiento del agente con la mejor configuración obtenida durante el entrenamiento, observándose un recorte en el número de actualizaciones debido a la parada anticipada antes de completar los 1000000 pasos en aproximadamente 100000 pasos (un 90 % menos del total de pasos que debería dar), y la validación de dicho agente:



(a) Rendimiento del agente durante el entrenamiento

(b) Rendimiento del agente durante la evaluación

Figura 4.9: Estudio de la convergencia en LunarLander con la versión básica de optimización bayesiana

Los resultados demuestran que, utilizando un umbral de recompensa y la implementación de early stopping, se puede lograr una convergencia mucho más rápida en el entorno de LunarLander, obteniendo una recompensa media de 223 en la validación. La optimización enfocada en minimizar el tiempo de entrenamiento en lugar de la recompensa maximiza la eficiencia del proceso, permitiendo al agente alcanzar la recompensa deseada en menos tiempo. Esta estrategia es especialmente útil en escenarios donde el tiempo de entrenamiento es crítico, mostrando que es posible obtener configuraciones de hiperparámetros efectivas que no solo logran la recompensa objetivo sino que también reducen significativamente el tiempo necesario para entrenar al agente.

4.5. Experimentos de la versión de optimización bayesiana con multifidelidad en LunarLander

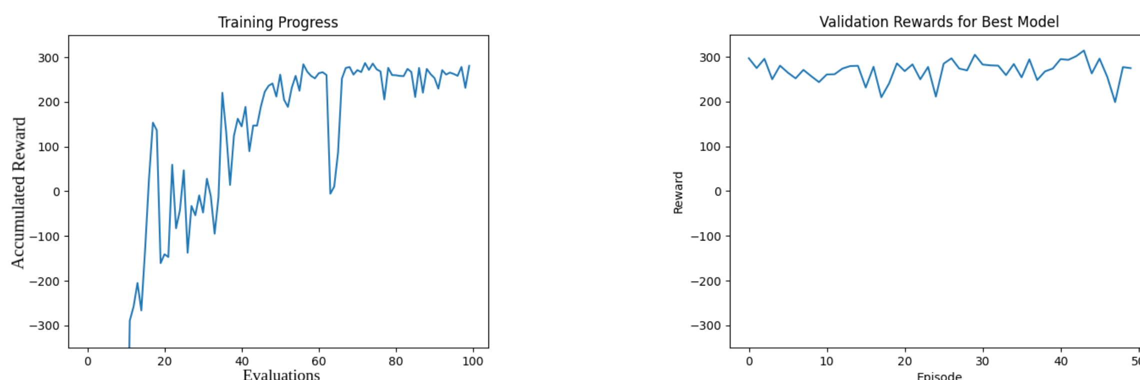
Experimento 1: Experimento Base

LunarLander es un entorno mucho más complejo que CartPole, en el que el agente debe aprender muchas más situaciones y acciones posibles en cada instante. Por eso, el aprendizaje es más complicado, lento y costoso computacionalmente. Para realizar este primer experimento con la versión de optimización bayesiana con multifidelidad, se configuró el proceso de optimización con una duración de 3 horas (10800 segundos) y se asignó un número total de pasos más elevado que en CartPole, alcanzando 1000000 pasos. Esto permitiría al agente entrenar con suficientes pasos para comprobar si la configuración es efectiva.

La configuración obtenida fue:

- **Factor de descuento:** 0.95547,
- **GAE Lambda:** 0.93638,
- **Tasa de aprendizaje:** 0.00016.

Esta configuración mostró el siguiente rendimiento durante el entrenamiento y la validación, como se observa en las siguientes gráficas:



(a) Rendimiento del agente durante el entrenamiento

(b) Rendimiento del agente durante la evaluación

Figura 4.10: Rendimiento del agente en LunarLander con la versión con multifidelidad de optimización bayesiana

Se probaron un total de 12 configuraciones, siendo esta la mejor, con un reward promedio en validación de 272. Esto representa una mejora del 13 % en comparación con el reward promedio de 239 obtenido sin la multifidelidad. Este incremento de rendimiento se debe a la capacidad de la optimización bayesiana con multifidelidad para probar más configuraciones en el mismo tiempo, permitiendo así una exploración más efectiva del espacio de hiperparámetros.

Estos resultados indican que, aunque el entorno de LunarLander es considerablemente más complejo, es posible encontrar configuraciones de hiperparámetros más efectivas utilizando la optimización bayesiana con multifidelidad. Las gráficas demuestran que el agente fue capaz de aprender y mejorar su rendimiento más eficientemente con la multifidelidad, logrando no solo un mejor reward promedio sino también optimizando el proceso de entrenamiento en términos de tiempo y número de pasos requeridos.

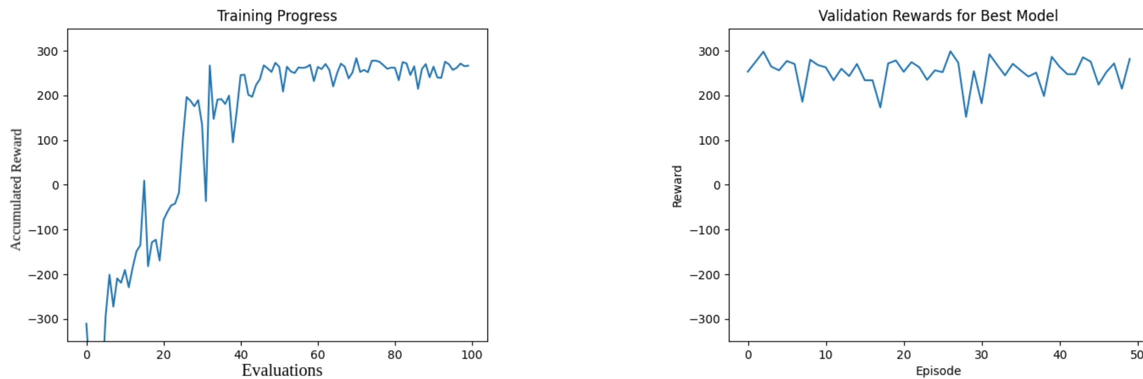
Experimento 2: Reducción de Número de Pasos

Realizar experimentos acortando el número de pasos nos permite observar cómo le cuesta al agente alcanzar recompensas similares a las obtenidas anteriormente de manera estable. En este experimento, se redujo en un 75 % el número de pasos total, manteniendo el tiempo de parada en 3 horas (10800 segundos). Esta reducción forzó al agente a converger en una solución estable más rápidamente, lo que resultó ser una tarea difícil.

La configuración obtenida fue:

- **Factor de descuento:** 0.96762,
- **GAE Lambda:** 0.96326,
- **Tasa de aprendizaje:** 0.00056.

El rendimiento del agente durante el entrenamiento y la evaluación se muestra en las siguientes gráficas:



(a) Rendimiento del agente durante el entrenamiento

(b) Rendimiento del agente durante la evaluación

Figura 4.11: Rendimiento del agente con número bajo de pasos en LunarLander con la versión con multifidelidad de optimización bayesiana

La validación obtuvo una recompensa media de 253, que es un muy buen resultado, indica que hay margen para mejorar. En comparación, utilizando la versión de optimización bayesiana con multifidelidad, se logró una recompensa promedio de 163, lo que representa una mejora del 35 %. Esta mejora se debe a la capacidad de la multifidelidad para explorar un mayor número de configuraciones de hiperparámetros en el mismo tiempo, facilitando la convergencia hacia soluciones más efectivas incluso con un número reducido de pasos.

Las gráficas [Figura 4.11] reflejan que, aunque el agente tuvo un buen desempeño en algunos episodios, en otros la recompensa fue negativa debido a fallos rápidos, destacando la insuficiencia del entrenamiento. La optimización bayesiana con multifidelidad mostró una mayor consistencia en el rendimiento del agente, logrando una recompensa promedio más alta y reduciendo la variabilidad en los resultados.

Experimento 3: Estudio de la convergencia

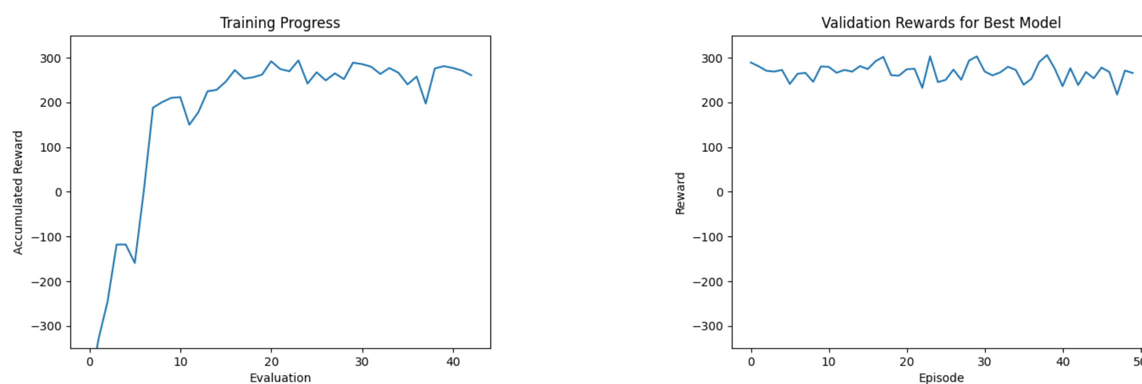
Para este experimento, se utilizó el archivo *convergence_genericSolver.py* modificado para la versión de optimización con multifidelidad. El entrenamiento se detuvo cuando la recompensa alcanzó consecutivamente 3 veces un valor superior a un umbral predefinido. Se estableció un umbral de 200 de recompensa (30 % menos del máximo) durante el entrenamiento y un tiempo máximo de 3 horas (10800 segundos) para el early stopping. El número de pasos máximos se fijó en 1000000, aunque este número de pasos es a menudo inferior en la mayoría de los entrenamientos de los agentes debido a la rápida convergencia con multifidelidad. Además, se ajustó el código para que, en lugar de minimizar la recompensa, se minimizara el tiempo de entrenamiento.

El resultado obtenido es el siguiente:

- **Factor de descuento:** 0.95714,
- **GAE Lambda:** 0.94283,

- **Tasa de aprendizaje:** 0.00028.

Las siguientes gráficas [Figura 4.12] muestran el rendimiento del agente con la mejor configuración obtenida durante el entrenamiento, observándose un recorte en el número de actualizaciones debido a la parada anticipada antes de completar los 1000000 pasos, en aproximadamente 400000 pasos (un 60 % menos del total de pasos que debería dar), y la validación de dicho agente:



(a) Rendimiento del agente durante el entrenamiento

(b) Rendimiento del agente durante la evaluación

Figura 4.12: Estudio de la convergencia en LunarLander con la versión con multifidelidad de optimización bayesiana

Los resultados demuestran que, utilizando un umbral de recompensa y la implementación de early stopping, se puede lograr una convergencia mucho más rápida en el entorno de LunarLander utilizando optimización con multifidelidad, obteniendo una recompensa media de 267 en la validación. Comparado con la versión sin multifidelidad, que obtuvo una recompensa media de 223, esto representa una mejora significativa del 17 %. La optimización con multifidelidad no solo minimiza el tiempo de entrenamiento sino que también mejora de manera considerable la recompensa obtenida. Esta estrategia es especialmente útil en escenarios donde el tiempo de entrenamiento es crítico y muestra que la multifidelidad puede conducir a configuraciones de hiperparámetros más efectivas, alcanzando la recompensa objetivo en menos tiempo y con mejores resultados en la evaluación final.

4.6. Conclusiones

En este estudio se evaluaron técnicas de optimización de hiperparámetros con y sin multifidelidad en los entornos de LunarLander y CartPole, destacando diferencias significativas en los resultados y la estabilidad de los modelos.

Cartpole	Versión Básica	Versión con multifidelidad
TimeSteps: 10^4	Recompensa Acumulada: **500**	Recompensa Acumulada: **500**
TimeSteps: 2500	Recompensa Acumulada: 487	Recompensa Acumulada: **500**
TimeSteps cuando la recompensa acumulada supera 350	TimeSteps: **6000** Recompensa Acumulada: 403	TimeSteps: 8000 Recompensa Acumulada: **476**

Cuadro 4.1: Comparación de resultados obtenidos con y sin multifidelidad en CartPole

La tabla 4.1 muestra una comparación entre los resultados obtenidos en CartPole utilizando optimización bayesiana básica y optimización bayesiana con multifidelidad. En el experimento base, ambos enfoques lograron una recompensa media de 500 en validación, indicando un rendimiento óptimo. Sin embargo, al reducir el número de pasos, la versión básica alcanzó una recompensa media de 487, mientras que la versión con multifidelidad mantuvo la recompensa máxima de 500. Esto representa una mejora del 2.67 % en la recompensa media al utilizar multifidelidad. En el estudio de la convergencia, la versión básica logró una recompensa media de 403 en 6000 pasos, mientras que la versión con multifidelidad alcanzó una recompensa media de 432 en 8000 pasos. Esto demuestra una menor eficiencia en términos de tiempo 30 % más lento pero representa una mejora del 15 % en la recompensa media.

En resumen, la optimización bayesiana con multifidelidad en el entorno de CartPole mostró mejoras significativas en términos de estabilidad y rapidez en la convergencia, lo que se traduce en una mayor eficiencia y eficacia del proceso de entrenamiento del agente.

LunarLander	Versión Básica	Versión con multifidelidad
TimeSteps: 10^6	Recompensa Acumulada: 239	Recompensa Acumulada: **272**
TimeSteps: 2,5 x 10^4	Recompensa Acumulada: 163	Recompensa Acumulada: **253**
TimeSteps cuando la recompensa acumulada supera 200	TimeSteps: 58000 Recompensa Acumulada: 223	TimeSteps: **40000** Recompensa Acumulada: **267**

Cuadro 4.2: Comparación de resultados obtenidos con y sin multifidelidad en LunarLander

En el entorno de LunarLander, la optimización bayesiana multifidelidad mostró una mejora significativa en comparación con la versión básica en varios aspectos.

En el experimento base con un millón de pasos y un tiempo máximo de 10800 segundos, la versión multifidelidad logró una recompensa acumulada de 272, superando la recompensa de 239 obtenida por la versión básica.

Al reducir el número de pasos a 250,000 y manteniendo el mismo tiempo de parada, la versión básica obtuvo una recompensa media de 163, mientras que la versión con multifidelidad alcanzó una recompensa de 253, mostrando una mejora considerable.

En el estudio de la convergencia, la versión básica necesitó 58000 pasos para superar una recompensa de 200, logrando una recompensa acumulada de 223. En contraste, la versión con multifidelidad solo requirió 40000 pasos para superar el mismo umbral, alcanzando una recompensa acumulada de 267, esto supuso una reducción del número de pasos necesarios del 30 %, a la par que mejoró el rendimiento un 17 %.

Además de estas mejoras en el rendimiento, la versión multifidelidad demostró ser mucho más robusta que la versión básica. Los resultados obtenidos con la optimización multifidelidad fueron consistentemente superiores en todos los experimentos, destacando su estabilidad y fiabilidad en comparación con la versión sin multifidelidad.

En resumen, la optimización bayesiana con multifidelidad en el entorno de LunarLander demostró mejoras significativas en términos de rendimiento, estabilidad y rapidez en la convergencia. Estas mejoras se traducen en una mayor eficiencia y eficacia del proceso de entrenamiento del agente, destacando la ventaja de utilizar multifidelidad en la optimización de hiperparámetros.

Capítulo 5

Conclusiones

En esta sección, presentamos las conclusiones derivadas del estudio comparativo de la optimización bayesiana de hiperparámetros con y sin multifidelidad. Nuestro objetivo principal fue demostrar que, dada una cantidad finita de tiempo para la búsqueda, la versión con multifidelidad supera a su contraparte sin multifidelidad en términos de eficiencia y rendimiento del modelo. Este objetivo se basó en la premisa de que la multifidelidad permite una exploración más informada del espacio de hiperparámetros, aprovechando información adicional sobre el comportamiento del modelo en diferentes niveles de fidelidad.

La optimización de hiperparámetros en el aprendizaje por refuerzo es una tarea fundamental para mejorar el rendimiento de los modelos. A lo largo de este trabajo, hemos llevado a cabo una serie de experimentos destinados a este propósito, centrándonos en dos entornos: LunarLander y CartPole.

Los resultados obtenidos indican claramente que la optimización con multifidelidad mejora el rendimiento del modelo en ambos entornos. En el caso de LunarLander, la versión con multifidelidad mostró un mejor desempeño en comparación con la optimización básica.

En el entorno de CartPole, aunque ambos métodos alcanzaron la recompensa máxima rápidamente debido a la simplicidad del entorno, la multifidelidad permitió llegar a la solución óptima con mayor consistencia y en menos tiempo.

La implementación de estrategias de parada anticipada y umbrales de recompensa en la versión con multifidelidad resultó en una convergencia significativamente más rápida. En LunarLander, por ejemplo, la optimización con multifidelidad redujo el número de pasos necesarios para alcanzar la recompensa deseada más eficientemente comparada con la versión sin multifidelidad.

La multifidelidad no solo aceleró la convergencia, sino que también mantuvo una estabilidad en el rendimiento del agente, evitando grandes fluctuaciones en las recompensas obtenidas durante el entrenamiento.

La eficiencia de la búsqueda de hiperparámetros se vio notablemente mejorada con la multifidelidad. La capacidad de evaluar configuraciones de hiperparámetros utilizando diferentes niveles de fidelidad permitió una utilización más efectiva del tiempo y los recursos disponibles. Esta eficiencia se traduce en un ahorro significativo de

recursos computacionales y tiempo, lo cual es especialmente importante en aplicaciones prácticas donde los costos de entrenamiento pueden ser prohibitivos.

Concluyendo, este estudio ha demostrado de manera contundente que la optimización bayesiana de hiperparámetros con multifidelidad supera a la versión básica en términos de rendimiento, convergencia y eficiencia. Los experimentos realizados en los entornos de LunarLander y CartPole evidencian que la multifidelidad no solo permite encontrar mejores configuraciones de hiperparámetros, sino que también lo hace de manera más rápida y estable.

La implementación de multifidelidad en la optimización de hiperparámetros es, por lo tanto, una estrategia recomendada para mejorar el rendimiento y la eficiencia de los modelos de aprendizaje por refuerzo. Esto abre la puerta a futuras investigaciones y aplicaciones prácticas donde se puede aprovechar al máximo esta técnica para desarrollar agentes más inteligentes y eficientes en una variedad de entornos y tareas.

5.1. Trabajo Futuro

Como posible trabajo futuro, se podrían explorar varias direcciones para mejorar aún más la optimización de hiperparámetros con multifidelidad. Una dirección prometedora sería la implementación de técnicas de meta-aprendizaje para adaptar dinámicamente los niveles de fidelidad durante el entrenamiento, optimizando así el uso de recursos. Además, sería valioso investigar la aplicación de esta metodología en otros entornos más complejos y realistas, como simulaciones industriales o videojuegos de alta complejidad, para evaluar la generalización y robustez de la optimización multifidelidad. Finalmente, la combinación de multifidelidad con otras técnicas avanzadas de optimización, como la optimización evolutiva o los algoritmos genéticos, podría ofrecer nuevas oportunidades para mejorar aún más el rendimiento y la eficiencia de los modelos de aprendizaje por refuerzo.

Bibliografía

[1] smac.scenario — SMAC3 Documentation 2.1.0 documentation. (s.f.)https://automl.github.io/SMAC3/main/api/smac.scenario.html

[2] Automl. (s.f.). GitHub - automl/SMAC3: SMAC3: A Versatile Bayesian Optimization Package for Hyperparameter Optimization. GitHub. https://github.com/automl/SMAC3/tree/main

[3] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017, 20 julio). Proximal Policy optimization Algorithms. arXiv.org. https://arxiv.org/abs/1707.06347

[4] Dlr-Rm. (s.f.). GitHub - DLR-RM/stable-baselines3: PyTorch version of Stable Baselines, reliable implementations of reinforcement learning algorithms. GitHub. https://github.com/DLR-RM/stable-baselines3

[5] Gymnasium documentation. (s.f.). https://gymnasium.farama.org/

[6] Frazier, P. I. (s.f.). A tutorial on Bayesian optimization. Arxiv.org. https://arxiv.org/pdf/1807.02811.pdf

[7] Bayesian-Optimization. (s.f.). GitHub - bayesian-optimization/BayesianOptimization: A Python implementation of global optimization with gaussian processes. GitHub. https://github.com/fmfn/BayesianOptimization

[8] Mikkola, P., Martinelli, J., Filstroff, L., & Kaski, S. (2022, 25 octubre). Multi-Fidelity Bayesian Optimization with Unreliable Information Sources. arXiv.org. https://arxiv.org/abs/2210.13937

[9] BoTorch · Bayesian Optimization in PyTorch. (s.f.). https://botorch.org/tutorials/multi_fidelity_bo. https://botorch.org/tutorials/multi_fidelity_bo

[10] Snoek, J., & Larochelle, H. (s.f.). Practical Bayesian optimization of machine learning algorithms. Neurips.cc..https://proceedings.neurips.cc/paper/2012/file/05311655a15b75fab86956663e1819cd-Paper.pdf

[11] Li, J., Dai, W., Metze, F., Qu, S., & Das, S. (2017, 20 marzo). A Comparison of deep learning methods for environmental sound. arXiv.org.https://arxiv.org/abs/1703.06902

[12] Kandasamy, K., Dasarathy, G., Schneider, J., & Poczos, B. (2017, 18 marzo). Multi-fidelity Bayesian Optimisation with Continuous Approximations. arXiv.org.https://arxiv.org/abs/1703.06240

[13] Swersky, K., & Snoek, J. (s.f.). Multi-Task Bayesian Optimization. Neurips.cc.https://proceedings.neurips.cc/paper_files/paper/2013/file/f33ba15effa5c10e873bf3842afb46a6-Paper.pdf

[14] Piera, R. S., Walborn, S. P., & Aguilar, G. H. (2021). Experimental demonstration of the advantage of using coherent measurements for phase estimation in the presence of depolarizing noise. Physical Review. A/Physical Review, A, 103(1). https://arxiv.org/abs/2006.08455

[15] Snoek, J., Rippel, O., Swersky, K., Kiros, R., Satish, N., Sundaram, N., Patwary, M. M. A., Prabhat, & Adams, R. P. (2015, 19 febrero). Scalable Bayesian optimization using deep neural networks. arXiv.org. https://arxiv.org/abs/1502.05700