



COMILLAS
UNIVERSIDAD PONTIFICIA

ICAI

ICADE

CIHS

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA

(ICAI)

Máster en Big Data: Tecnología y Analítica Avanzada

TÍTULO DEL TRABAJO FIN DE MÁSTER

LLMs aplicados al diseño automático de agentes basados en RL

Autor

Enrique Gil Garcia

Dirigido por

Martín Gallardo, Emilio

Madrid

June 2024

Declaro, bajo mi responsabilidad, que el Proyecto presentado con el título
LLMs aplicados al diseño automático de agentes basados en RL
en la ETS de Ingeniería - ICAI de la Universidad Pontificia Comillas en el
curso académico 2024/2025 es de mi autoría, original e inédito y
no ha sido presentado con anterioridad a otros efectos.

El Proyecto no es plagio de otro, ni total ni parcialmente y la información que ha sido
tomada de otros documentos está debidamente referenciada.

A handwritten signature in black ink, appearing to read 'Enrique', with a long, sweeping horizontal stroke extending to the right.

Fdo.: Enrique Gil Garcia

Fecha:15/05/2025

Abstract

This thesis investigates the potential of integrating Large Language Models (LLMs) into the design of reward functions for Reinforcement Learning (RL) agents, with a particular focus on automating and enhancing the reward shaping process. Drawing inspiration from the Eureka framework and related research in intrinsic motivation and natural language-guided learning, this work aims to assess whether LLMs can serve as effective co-designers or autonomous creators of reward functions in complex RL environments. Traditional reward engineering is often labor-intensive, domain-specific, and prone to suboptimal guidance, which can hinder learning efficiency and generalization. By leveraging the natural language understanding and generative capabilities of LLMs, we propose a method for translating task goals and behavioral heuristics into executable reward components.

To evaluate this approach, we conduct a series of experiments in standard Gymnasium environments, including *LunarLander-v2* and *BipedalWalker-v3*, where LLMs are prompted to generate reward logic based on task descriptions and agent behavior summaries. The performance of LLM-derived reward functions is compared against baseline hand-crafted functions using common RL algorithms such as Proximal Policy Optimization (PPO) or Deep Q-Network (DQN). Our results indicate that in certain task-specific scenarios, LLM-generated reward functions can lead to faster convergence and more robust policies than their manually designed counterparts. Nevertheless, we also identify key limitations, including inconsistency in LLM outputs, sensitivity to prompt phrasing, and challenges in mapping high-level language descriptions to low-level observation spaces.

These findings highlight both the promise and the current limitations of using LLMs in reward function design. While LLMs show potential as tools for accelerating and democratizing RL development, achieving fully autonomous and reliable reward shaping requires further advances in prompt engineering, model alignment, and the incorporation of domain knowledge. We conclude by outlining future directions for hybrid reward design frameworks that combine human intuition with the generative flexibility of LLMs, setting the stage for more adaptive and scalable reinforcement learning systems.

Resumen

Este Trabajo de Fin de Máster investiga el potencial de integrar Modelos de Lenguaje (LLMs, por sus siglas en inglés) en el diseño de funciones de recompensa para agentes de Aprendizaje por Refuerzo (RL). El enfoque principal es automatizar y mejorar el proceso de configuración de recompensas, tomando como inspiración el *paper* de Eureka sobre *Reward Shaping* y trabajos relacionados con la misma motivación y aprendizaje guiado por lenguaje natural. El diseño tradicional de funciones de recompensa suele requerir mucho trabajo manual, ser específico del dominio y propenso a errores que dificultan la eficiencia del aprendizaje y su capacidad de generalización. Aprovechando las capacidades de comprensión del lenguaje natural y generación de código de los LLMs, proponemos un método para traducir descripciones de tareas y heurísticas de comportamiento en componentes de recompensa ejecutables.

Para evaluar este enfoque, se llevaron a cabo una serie de experimentos en entornos estándar de Gymnasium, como *LunarLander-v2* y *BipedalWalker-v3*, en los que los LLMs fueron utilizados para generar lógicas de recompensa a partir de descripciones de tareas y resúmenes del comportamiento del agente. El rendimiento de las funciones de recompensa generadas por LLMs se comparó con funciones diseñadas manualmente utilizando algoritmos de RL comunes, como Proximal Policy Optimization (PPO). Los resultados indican que, en ciertos escenarios específicos, las funciones de recompensa derivadas de LLMs pueden conducir a una convergencia más rápida y a políticas más robustas que las diseñadas manualmente. Sin embargo, también se identificaron limitaciones importantes, como la inconsistencia en las salidas del modelo y la sensibilidad a la formulación de los prompts.

Estos hallazgos evidencian tanto el potencial como las limitaciones actuales del uso de LLMs en el diseño de funciones de recompensa. Aunque los LLMs muestran ser herramientas prometedoras para acelerar y democratizar el desarrollo de RL, lograr una configuración de recompensas completamente autónoma y fiable requiere avances adicionales en la ingeniería de prompts, la alineación del modelo y la integración del conocimiento del dominio. Concluimos proponiendo futuras líneas de investigación centradas en marcos de diseño híbrido humano-LLM que combinen la intuición humana con la flexibilidad generativa de los modelos de lenguaje, abriendo el camino hacia sistemas de aprendizaje por refuerzo más adaptativos y escalables.

Contenidos

1.	Introducción	1
2.	Estado del arte	3
3.	Descripción del trabajo	5
3.1.	Arquitectura del Sistema	6
3.2.	Estrategias de ejecución	13
3.3.	Análisis y discusión	19
3.4.	Coste del entrenamiento	22
4.	Conclusiones y trabajos futuros	23
5.	Bibliografía	25

1. Introducción

El Aprendizaje por Refuerzo (Reinforcement Learning, RL) es una rama fundamental del aprendizaje automático que se centra en entrenar agentes autónomos para que tomen decisiones óptimas a través de la interacción con un entorno. En este paradigma, los agentes aprenden mediante la retroalimentación que reciben en forma de recompensas, las cuales son proporcionadas por una función de recompensa que valora sus acciones en función de un objetivo deseado. A lo largo de múltiples episodios de prueba y error, los agentes ajustan sus políticas de acción para maximizar la recompensa acumulada, lo que les permite desarrollar comportamientos cada vez más eficaces.

Sin embargo, el diseño de funciones de recompensa adecuadas constituye uno de los principales desafíos del Aprendizaje por Refuerzo. Una recompensa mal definida puede inducir conductas no deseadas, dificultar la convergencia del agente o incluso imposibilitar el aprendizaje. Tradicionalmente, estas funciones han sido diseñadas manualmente por expertos humanos, lo que requiere un profundo conocimiento del dominio del problema y de los mecanismos internos del entorno. Este enfoque manual no solo es propenso a errores y ambigüedades, sino que también presenta problemas de escalabilidad cuando se intenta aplicar RL en dominios complejos y variados. En consecuencia, ha surgido una necesidad creciente de desarrollar métodos más automatizados, flexibles y generalizables para la especificación de recompensas.

En paralelo, los Modelos de Lenguaje (Large Language Models, LLMs), como GPT-4, han revolucionado la forma en que se abordan tareas complejas que involucran procesamiento de lenguaje natural, generación de código, razonamiento lógico y síntesis de conocimiento. Estos modelos, entrenados con grandes cantidades de datos y código, han demostrado un notable desempeño en tareas de programación, incluyendo la generación de scripts funcionales, la resolución de problemas algorítmicos y la comprensión semántica de instrucciones. Este avance ha abierto nuevas oportunidades para su integración en procesos automatizados de diseño de sistemas inteligentes.

Inspirados por investigaciones pioneras como el *paper* realizado por *Eureka*[3], que propone el uso de LLMs para generar funciones de recompensa a partir de descripciones de tareas, y por enfoques más amplios de conformado de recompensas intrínsecas [2] y aprendizaje guiado por lenguaje, esta tesis se propone investigar el uso de LLMs para generar funciones de

recompensa [1] personalizadas para agentes de RL. La hipótesis principal es que, mediante un diseño adecuado de prompts, los LLMs pueden interpretar descripciones de tareas y producir código ejecutable en Python que defina recompensas eficaces para el aprendizaje del agente.

Nuestro estudio se desarrolla principalmente en el entorno LunarLander-v2 de la biblioteca Gymnasium, un entorno clásico en la investigación de RL que presenta una dinámica física realista y un objetivo claro (aterrizar suavemente una nave lunar entre dos puntos). Se plantea una evaluación comparativa del desempeño de tres agentes entrenados: (i) un agente base que utiliza la recompensa estándar del entorno, (ii) un agente que utiliza una función de recompensa estática generada una única vez por el modelo GPT-4o-mini, y (iii) un agente con una función de recompensa dinámica, actualizada periódicamente por el LLM cada 10% del entrenamiento, en función del comportamiento observado.

Además, exploramos cómo el diseño de los *prompts* —es decir, las instrucciones en lenguaje natural que se entregan al LLM— influye en la calidad, la coherencia y la utilidad del código generado. Se presentan ejemplos concretos de código de recompensa generado, así como tablas comparativas de desempeño que permiten evaluar el impacto real de estas funciones en la eficacia del agente. A través de este análisis, también se identifican los principales retos técnicos del enfoque, incluyendo errores de validación de código, desajustes entre la descripción del entorno y el espacio de observación, y comportamientos inconsistentes derivados de la aleatoriedad en las respuestas del modelo.

Finalmente, este trabajo se enmarca dentro del contexto más amplio de la investigación emergente sobre el uso de LLMs en Aprendizaje por Refuerzo, un área que está ganando relevancia por su potencial para automatizar tareas tradicionalmente complejas del ciclo de vida del entrenamiento, como la especificación de recompensas, la generación de políticas o la interpretación de comportamientos. Al aportar evidencia empírica sobre las capacidades actuales y las limitaciones de los LLMs en este contexto, esta tesis busca contribuir al avance hacia sistemas de RL más accesibles, adaptativos y escalables, al tiempo que señala las oportunidades para investigaciones futuras en el diseño híbrido humano–modelo de funciones de recompensa.

2. Estado del arte

El Aprendizaje por Refuerzo (Reinforcement Learning, RL) ha experimentado avances significativos en la última década, impulsados en gran parte por el desarrollo de algoritmos basados en redes neuronales profundas, como Deep Q-Networks (DQN) y Proximal Policy Optimization (PPO). Estos algoritmos han demostrado un rendimiento muy bueno en una amplia gama de dominios, incluyendo videojuegos complejos como Atari y StarCraft, tareas de control robótico de alta dimensionalidad, y problemas de optimización en el mundo real. No obstante, a pesar de estos avances, el diseño de funciones de recompensa continúa siendo un obstáculo crítico que limita la generalización y la eficiencia del aprendizaje en muchos entornos.

Una función de recompensa mal especificada puede inducir a los agentes a aprender comportamientos subóptimos, ineficientes o incluso contraproducentes. Para mitigar este problema, se ha desarrollado la técnica conocida como *reward shaping* o conformado de recompensas, que consiste en añadir señales auxiliares a la función de recompensa con el fin de acelerar el aprendizaje y guiar al agente hacia comportamientos deseados [2]. Si bien esta técnica puede mejorar considerablemente el rendimiento en ciertos casos, también introduce riesgos: requiere un conocimiento detallado del dominio del problema, y una mala implementación puede desviar al agente de los objetivos principales o crear dependencias no deseadas en la señal de recompensa.

Con la irrupción de los Modelos de Lenguaje de Gran Escala (LLMs), como GPT-3.5 y GPT-4, ha surgido una nueva oportunidad para abordar el diseño de recompensas desde un enfoque más automatizado y flexible. Estos modelos han demostrado habilidades notables en tareas de generación de código, resolución de problemas algorítmicos, y síntesis semántica, lo cual sugiere su potencial utilidad en el contexto de RL para generar funciones de recompensa a partir de descripciones de tareas en lenguaje natural [2].

Uno de los trabajos más influyentes en esta línea es el marco **Eureka** [3], que utiliza un enfoque evolutivo guiado por LLMs para la generación de funciones de recompensa. En Eureka, el modelo de lenguaje genera candidatos de código de recompensa sin necesidad de *prompts* específicos, y luego un bucle iterativo de evaluación y reflexión permite seleccionar y mejorar las funciones más efectivas.

Esta tesis se basa en estos trabajos recientes para investigar de forma sistemática el uso de LLMs, concretamente GPT-4o-mini, como herramientas generadoras de funciones de recompensa personalizadas para entornos de Gymnasium. A diferencia de Eureka, que aplica un ciclo de optimización completo, aquí se exploran tanto recompensas estáticas (generadas una sola vez) como dinámicas (reescritas periódicamente durante el entrenamiento), con el objetivo de analizar la influencia del diseño de *prompts* y la calidad del código generado en el rendimiento final del agente. Al hacerlo, se busca contribuir a la comprensión de los límites y las oportunidades de los LLMs como asistentes en el diseño automatizado de recompensas para agentes inteligentes.

3. Descripción del trabajo

La presente investigación se centra en el desarrollo de un sistema automatizado que permite integrar Modelos de Lenguaje de Gran Escala (LLMs) en el proceso de diseño de funciones de recompensa para agentes de Aprendizaje por Refuerzo (Reinforcement Learning, RL). Este trabajo aborda uno de los principales desafíos identificados en el estado del arte: la dificultad de diseñar funciones de recompensa manualmente, un proceso que suele ser laborioso, propenso a errores y difícil de escalar a entornos complejos. Para superar estas limitaciones, se propone un enfoque innovador que aprovecha las capacidades de comprensión del lenguaje natural y generación de código de los LLMs, permitiendo traducir objetivos de alto nivel y descripciones de tareas en funciones de recompensa ejecutables que guíen el aprendizaje del agente de manera más eficiente.

La metodología adoptada combina la generación automática de funciones de recompensa, tanto estáticas (definidas una única vez al inicio del entrenamiento) como dinámicas (adaptadas iterativamente durante el entrenamiento), con la evaluación cuantitativa y cualitativa del rendimiento de agentes de RL en entornos estándar de la biblioteca Gym Aula. El objetivo principal es analizar el impacto de estas funciones generadas por LLMs sobre la eficacia del aprendizaje del agente, midiendo métricas como la velocidad de convergencia, la calidad de las políticas aprendidas y la robustez ante diferentes condiciones del entorno. Además, se busca estudiar las ventajas de este enfoque —como la reducción del esfuerzo humano en el diseño de recompensas— y sus limitaciones, incluyendo posibles inconsistencias en las salidas del LLM y la sensibilidad a la calidad de los prompts utilizados.

Para llevar a cabo esta investigación, se ha implementado una arquitectura modular en Python que permite ejecutar experimentos de manera controlada, replicable y escalable. Esta arquitectura está diseñada para facilitar la integración de los LLMs en el ciclo de entrenamiento de RL, desde la generación de prompts hasta la evaluación del rendimiento del agente. Está compuesta por varios módulos funcionales que se comunican entre sí de forma eficiente, incluyendo componentes para la interacción con el LLM, la validación de las funciones de recompensa generadas, y la adaptación dinámica de estas funciones durante el entrenamiento. La modularidad del sistema no solo garantiza flexibilidad para incorporar mejoras futuras, como el uso de nuevos modelos de lenguaje o algoritmos de RL, sino que también permite un análisis

detallado de cada etapa del proceso, desde la generación del código de recompensa hasta su impacto en el aprendizaje del agente.

Un aspecto clave de este trabajo es su enfoque práctico, centrado en entornos reales de la biblioteca Gymnasium, como LunarLander-v2 y BipedalWalker-v3, que ofrecen un equilibrio entre complejidad y accesibilidad para evaluar el rendimiento de los agentes. A través de esta implementación, se busca no solo demostrar la viabilidad de usar LLMs como herramientas de diseño automatizado en RL, sino también sentar las bases para futuros desarrollos que combinen la creatividad de los modelos de lenguaje con la capacidad de los agentes de RL para resolver problemas complejos en entornos dinámicos.

3.1. Arquitectura del Sistema

La arquitectura del sistema desarrollado consta de múltiples componentes interdependientes que cumplen funciones específicas dentro del flujo completo de entrenamiento y evaluación de los agentes. A continuación se describen los principales módulos y sus funcionalidades:

1. **run_experiment.py**: Este archivo actúa como el punto de entrada principal del sistema. Contiene la lógica para ejecutar experimentos de principio a fin. Entre sus principales funciones se incluyen:
 - a. Configuración del entorno: Permite seleccionar el entorno de Gymnasium a utilizar (por ejemplo, LunarLander-v2 o BipedalWalker-v3).
 - b. Selección del algoritmo de entrenamiento: En función del tipo de espacio de acción del entorno (discreto o continuo), se selecciona el algoritmo de RL correspondiente. Para entornos con espacio de acción discreto, se emplea Deep Q-Networks (DQN), mientras que para entornos con acciones continuas, se utiliza Proximal Policy Optimization (PPO).
 - c. Entrenamiento del agente: El archivo gestiona el ciclo de entrenamiento, incorporando las funciones de recompensa generadas automáticamente.
 - d. Visualización y grabación: Se incluye funcionalidad para visualizar el comportamiento del agente durante los episodios y grabar vídeos de las ejecuciones, lo cual resulta útil para análisis cualitativos.

2. **Módulo `llm_interface`:** Este módulo contiene las funciones que permiten la interacción con el modelo LLM a través de la API de OpenAI. Está compuesto por dos funciones clave:
- a. **`generate_prompt`:** Esta función tiene como propósito generar automáticamente un prompt detallado a partir del nombre de un entorno de Gymnasium. Su funcionamiento se puede resumir en los siguientes pasos:
 - i. A partir del nombre del entorno (`env_name`), se construye una consulta que se envía al modelo `gpt-4o-mini` de OpenAI.
 - ii. Se le solicita al modelo que describa el entorno de forma detallada, incluyendo el contexto general, los objetivos que debe alcanzar el agente, las variables observables y las acciones disponibles.
 - iii. Además, se incluyen instrucciones específicas para guiar al modelo en la creación de una función de recompensa adecuada al entorno, considerando:
 - 1. Las metas del entorno.
 - 2. Los principios del diseño de recompensas densas o escalonadas.
 - 3. Las buenas prácticas para fomentar la exploración y evitar recompensas esparsas o engañosas.
 - iv. En el caso de estrategias de recompensa dinámicas, se le añade un historial del comportamiento del agente para que el modelo tenga en cuenta la evolución durante el entrenamiento.
 - v. El resultado se guarda en un archivo llamado `prompts.py`, dentro de una variable llamada `REWARD_FUNCTION_PROMPT`, que se usará posteriormente como plantilla para la generación del código de la función de recompensa.

```
def generate_prompt(entorno: str, output_filename: str = "prompts.py") -> None:
    print(f"Generating prompt for {entorno}")
    prompt = f"""
    You are tasked with creating a detailed description for the Gymnasium environment "{entorno}" to guide the generation of a custom reward function. The d

    Include the following sections in the output:

    ### 1. Environment Context
    - **Purpose**: Describe the environment's objective (e.g., navigation, balancing, control, game-playing).
    - **Observation Space**: Specify the type (e.g., Box, Dict, Tuple), shape (e.g., Box(24,)), and list all components with:
      - Name (e.g., hull x-position, joint angle, lidar reading).
      - Physical meaning (e.g., robot's horizontal position, knee joint angle, distance to terrain).
      - Units (if applicable, e.g., meters, radians, pixels).
      - Range or bounds (e.g., [-1, 1], {0, 1} for booleans, [0, 255] for images).
      - Exact number of components for vector spaces (e.g., 24 for BipedalWalker-v3, 8 for LunarLander-v2).
    - For complex spaces (e.g., Dict, Tuple, Box with high dimensions), describe each sub-component recursively with access method (e.g., obs['key'], obs[
    - **Action Space**: Specify the type (e.g., Discrete, Box, MultiDiscrete) and describe all possible actions:
      - For Discrete, list each action and its effect (e.g., 0: no-op, 1: move left).
      - For Box, list each dimension, its bounds (e.g., [-1, 1]), and effect (e.g., torque on hip joint).
      - For MultiDiscrete or complex spaces, describe each component and its effect.
    - **Starting State**: Describe the initial state (e.g., position, velocity, randomization, or fixed state).
    - **Termination Conditions**: List all conditions for episode termination (e.g., timeout, crash, goal reached, out-of-bounds).
    - **Success/Failure**: Define success (e.g., reaching a goal state, achieving a reward threshold) and failure (e.g., crashing, timeout) with measurable

    ### 2. Goal
    - **Objective**: State the agent's primary goal (e.g., "walk across terrain", "balance a pole", "maximize score").
    - **Success Conditions**: Specify measurable conditions for success (e.g., "distance traveled ≥ 200 meters", "upright for 200 steps").
    - **Avoidance**: List states or behaviors to avoid (e.g., falling, excessive speed, instability).
    - **Progress Metrics**: Suggest measurable indicators of progress (e.g., distance traveled, stability, score increase).
    """
```

Función generate_prompt

```
REWARD_FUNCTION_PROMPT = """\
### 1. Environment Context
- **Purpose**: The LunarLander-v2 environment is designed for control and navigation tasks, where the objective is to land a spacecraft on a designated

- **Observation Space**: The observation space is a 'Box' with shape `(8,)`, consisting of the following components:
  - **hull x-position**: The horizontal position of the lander. (meters)
  - **hull y-position**: The vertical position of the lander. (meters)
  - **hull velocity x**: The horizontal velocity of the lander. (meters/second)
  - **hull velocity y**: The vertical velocity of the lander. (meters/second)
  - **hull angle**: The orientation angle of the lander. (radians)
  - **hull angular velocity**: The rotational speed of the lander. (radians/second)
  - **leg contact**: A boolean indicator of whether the left leg is in contact with the ground. (0 or 1)
  - **right leg contact**: A boolean indicator of whether the right leg is in contact with the ground. (0 or 1)

- **Action Space**: The action space is `Discrete(4)`, with the following possible actions:
  - **0**: Do nothing (no operation).
  - **1**: Apply a left thrust.
  - **2**: Apply a right thrust.
  - **3**: Apply a downward thrust.

- **Starting State**: The initial state of the environment is randomized, with the lander starting at a height above the surface and with a random veloc

- **Termination Conditions**: The episode terminates under the following conditions:
  - The lander crashes (e.g., if it lands too hard or at an incorrect angle).
  - The lander successfully lands on the landing pad.
  - A timeout occurs after a maximum number of steps (usually 1000).

- **Success/Failure**: Success is defined as landing the spacecraft on the designated pad without crashing, achieving a reward of +100. Failure occurs i
```

Resultado de la función en REWARD_FUNCTION_PROMPT

b. **query_llm**: Esta función es la encargada de realizar la consulta final al LLM con el prompt construido previamente. Su funcionamiento incluye las siguientes características:

- i. Personalización del prompt: Si el prompt contiene el marcador `{history_text}`, éste será sustituido por un texto que refleja el historial reciente del comportamiento del agente (por ejemplo, acciones tomadas, recompensas obtenidas, cambios en el estado), lo que permite generar funciones de recompensa dinámicas ajustadas al contexto de entrenamiento.

- ii. Instrucciones estrictas al modelo: Para asegurar que el código generado sea válido y funcional, se establece un mensaje de sistema con instrucciones rigurosas:
 1. La respuesta debe ser solo código en Python, sin explicaciones ni formato Markdown.
 2. La función generada debe llamarse `reward_function(obs, default_reward, previous_obs, action)`, y debe retornar exclusivamente un valor de tipo *float*.
 3. No se permite el uso de librerías externas (como NumPy o TensorFlow), lo que previene errores por dependencias no disponibles.
 4. Se exige cuidado al manejar la estructura del espacio de observación (listas, diccionarios, vectores), incorporando controles como `len(obs)` o estructuras `try-except` para evitar errores de indexado o desempaquetado incorrecto.
 5. Se añaden restricciones adicionales como el uso de operaciones matemáticas seguras, la prevención de divisiones por cero, y la obligación de mantener los valores dentro de rangos acotados (por ejemplo, `[-100, 100]`).
- iii. Salida: El contenido generado por el LLM se devuelve como una cadena de texto, que posteriormente es evaluada y convertida a una función ejecutable dentro del flujo de entrenamiento del agente.

```
def query_llm(prompt: str) -> str:
    final_prompt = REWARD_FUNCTION_PROMPT.format(history_text=prompt) if "{history_text}" in prompt else prompt

    response = client.chat.completions.create(
        model="gpt-4o-mini",
        messages=[
            {
                "role": "system",
                "content": (
                    "You are a reinforcement learning assistant. Reply only with the Python code of the reward function, no explanations or markdown.\n"
                    "Write a function called 'reward_function(obs, default_reward, previous_obs, action)' that improves the reward design for the specified Gymna\n"
                    "The function must:\n"
                    "- Return a single float (e.g., 0.0). Do NOT return lists, tuples, arrays, or None.\n"
                    "- Use safe indexing for obs and previous_obs (e.g., obs[0] for vectors, obs['key'] for dictionaries) based on the observation space structur\n"
                    "- Avoid hard-coded unpacking that assumes a fixed number of observation components.\n"
                    "- Check observation length or structure to prevent unpacking errors (e.g., use len(obs) or try-except).\n"
                    "- Avoid external libraries (use only built-in Python math).\n"
                    "- Handle edge cases (e.g., previous_obs is None, complex observation spaces).\n"
                    "- Ensure numerical stability (e.g., avoid division by zero, bound outputs to [-100, 100]).\n"
                    "- Align with the environment's goal, avoid sparse or exploitable rewards, and ensure compatibility with RL algorithms.\n"
                )
            },
            {"role": "user", "content": final_prompt}
        ],
        temperature=0.3
    )
    return response.choices[0].message.content
```

Función query_llm

```
def reward_function(obs, default_reward, previous_obs, action):
    if previous_obs is None:
        return default_reward

    if len(obs) >= 8:
        hull_x, hull_y, hull_vx, hull_vy, hull_angle, hull_ang_vel, left_leg_contact, right_leg_contact = obs

        reward = -0.03 # Default per-step penalty

        # Reward for successful landing
        if left_leg_contact and right_leg_contact and hull_vy <= 0.5:
            reward += 100 # Successful landing

        # Penalty for crashing
        elif hull_vy > 1.0 or abs(hull_vx) > 1.0:
            reward -= 100 # Crash penalty

        # Reward for reducing vertical velocity
        if hull_vy < 0:
            reward += min(1.0, -hull_vy) # Encourage slowing down

        # Reward for horizontal position towards the landing zone
        if hull_x >= -0.5 and hull_x <= 0.5:
            reward += 10 # Closer to landing zone

        # Penalty for excessive angular velocity
        if abs(hull_ang_vel) > 1.0:
            reward -= 5 # Excessive rotation penalty

        # Bound the reward to [-100, 100]
        return max(-100, min(100, reward))

    return default_reward # Fallback for unexpected observation structure
```

Ejemplo de una función creada con query_llm

3. Modulo custom_reward_env

El módulo `custom_reward_env.py` representa un componente clave en la arquitectura del sistema, ya que permite integrar las funciones de recompensa generadas dinámicamente por un LLM en el bucle de entrenamiento de un agente de Aprendizaje por Refuerzo (RL). Para ello, implementa un envoltorio personalizado (*Wrapper*) sobre entornos de la biblioteca Gymnasium, el cual reemplaza o modifica la recompensa estándar por una función generada por el modelo gpt-4o-mini, previamente ajustada y validada.

Clase LLMRewardWrapper

La clase principal del módulo es `LLMRewardWrapper`, que hereda de `gym.Wrapper` y extiende su funcionalidad para admitir dos estrategias diferenciadas:

- **Estrategia Estática** (strategy='static'): Se genera una única función de recompensa al inicio del experimento, la cual se utiliza sin modificaciones durante todo el entrenamiento.
- **Estrategia Dinámica** (strategy='dynamic'): Se generan nuevas funciones de recompensa en intervalos predefinidos a lo largo del entrenamiento, adaptándose al historial reciente de comportamiento del agente.

La estructura de esta clase incluye múltiples mecanismos de control, validación y registro de errores, diseñados para asegurar la robustez del sistema.

Atributos destacados

- strategy: Define si se utilizará una función estática o dinámica.
- refresh_interval: Número de pasos entre actualizaciones de la función de recompensa en modo dinámico.
- max_dynamic_updates: Número máximo de actualizaciones permitidas para evitar una sobrecarga de llamadas al LLM.
- reward_func, current_dynamic_reward_func: Contienen la función activa de recompensa, estática o dinámica respectivamente.
- previous_reward_functions, reward_results: Registro histórico de funciones generadas y su impacto en episodios anteriores.

Flujo para Estrategia Estática

1. Al inicializar con strategy='static', se llama a _init_static_reward().
2. Se genera una función de recompensa mediante query_llm(), utilizando como plantilla el REWARD_FUNCTION_PROMPT.
3. El código generado es limpiado de caracteres innecesarios y ejecutado en un entorno local seguro con exec.

4. La función resultante (`reward_function`) se valida utilizando muestras aleatorias del espacio de observación y acción. Se comprueba que devuelva un único valor float.
5. Si es válida, se guarda (`logs/reward_func_static.py`) para trazabilidad y reproducibilidad.

Flujo para Estrategia Dinámica

1. Durante la ejecución del agente, si se alcanza un múltiplo de `refresh_interval` y no se ha superado `max_dynamic_updates`, se genera una nueva función.
2. Para ello, se construye un historial (*history_text*) con versiones anteriores de funciones y sus respectivos resultados, y se introduce dinámicamente en el prompt enviado a `query_llm()`.
3. La función generada se valida igual que en la estrategia estática y, si es válida, se activa como la nueva función dinámica (*current_dynamic_reward_func*), además de registrarse en disco de manera local.
4. En cada paso, si hay una función dinámica activa, esta se aplica en sustitución de la recompensa original, y se almacena el resultado para incluirlo en futuras versiones del prompt.

Funciones auxiliares

- `_clean_code(code: str)`: Elimina delimitadores Markdown (````python`) para asegurar que el código sea ejecutable por `exec()`.
- `reset()`: Inicializa la observación previa del entorno.
- `step(action)`: Ejecuta una acción en el entorno, aplica la función de recompensa correspondiente según la estrategia activa y registra el estado previo.

3.2. Estrategias de ejecución

La experimentación se ha diseñado para evaluar el impacto de funciones de recompensa generadas por un modelo LLM en el rendimiento de agentes de Aprendizaje por Refuerzo (RL) en entornos simulados. Para ello, el archivo `run_experiment.py` permite lanzar ejecuciones bajo tres estrategias diferenciadas, que permiten establecer una comparación directa entre el enfoque clásico y las propuestas integradas con LLM.

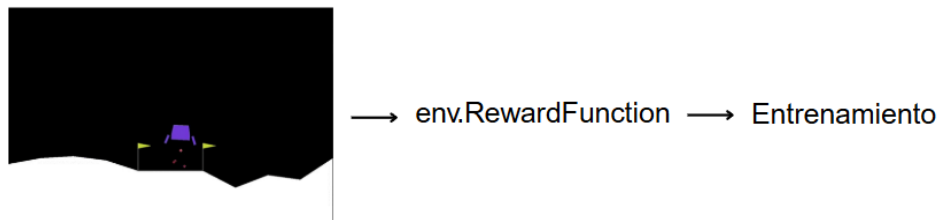
Cada experimento se ejecuta con un número fijo de pasos de entrenamiento (por ejemplo, 100.000 timesteps), manteniendo constantes el entorno y el algoritmo base (DQN para entornos discretos, PPO para entornos continuos). Lo que varía entre estrategias es la fuente y la evolución de la función de recompensa.

1. Estrategia Baseline (Recompensa Original)

Esta estrategia actúa como línea base para medir el rendimiento estándar del entorno de Gymnasium sin intervención del modelo LLM.

Configuración: El entorno se utiliza tal y como se proporciona por defecto, sin modificar su lógica interna de recompensa.

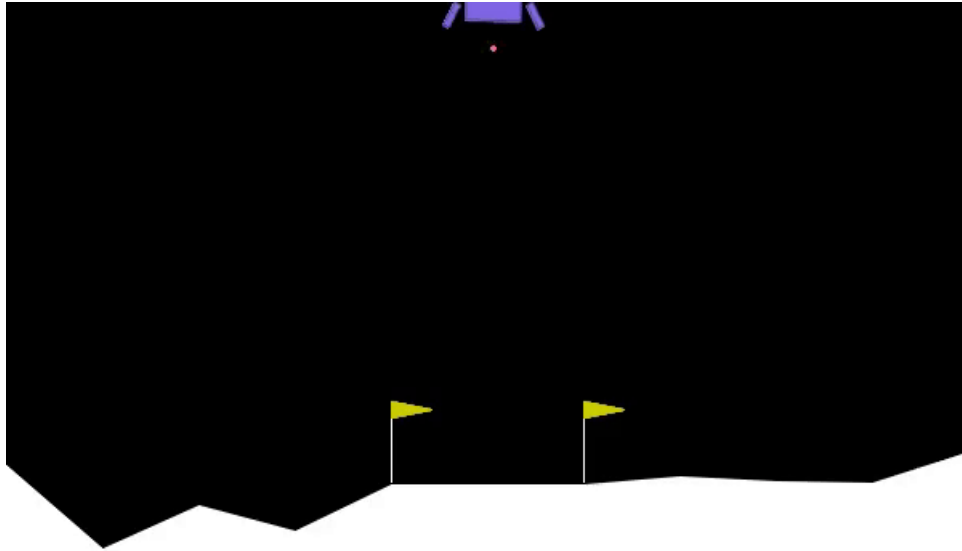
Arquitectura baseline



Recompensa: Se emplea la función de recompensa original incluida en el entorno (por ejemplo, en LunarLander-v2 o BipedalWalker-v3).

Propósito: Sirve como punto de comparación para evaluar si las funciones de recompensa generadas por el LLM conducen a un comportamiento más eficiente o adaptativo del agente.

Duración: Se entrena el agente durante el total de N pasos (por defecto, 100.000).



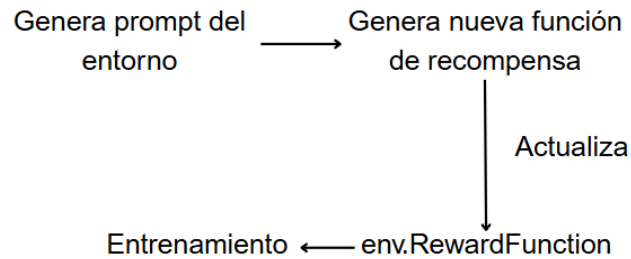
Baseline visual

2. Estrategia Estática (LLM con recompensa fija)

En esta configuración se utiliza la capacidad del LLM para generar una función de recompensa alternativa, diseñada a partir de una descripción detallada del entorno y sus objetivos.

Configuración: Se crea el entorno con un *wrapper* personalizado (LLMRewardWrapper) configurado en modo static.

Arquitectura static

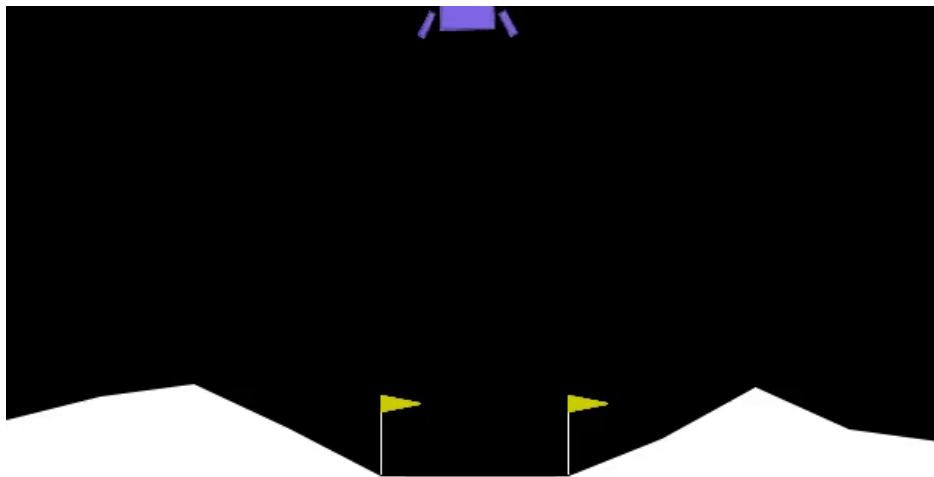


Recompensa: Una única función de recompensa es generada al comienzo del experimento mediante la función `query_llm`, utilizando el prompt definido en `REWARD_FUNCTION_PROMPT`.

Aplicación: Esta función reemplaza a la original y se mantiene fija durante todo el proceso de entrenamiento.

Validación: Antes de ser aplicada, la función generada es validada automáticamente para comprobar que devuelve un valor escalar y no contiene errores estructurales.

Propósito: Evaluar si una redefinición única de la recompensa basada en lenguaje natural mejora el aprendizaje del agente.

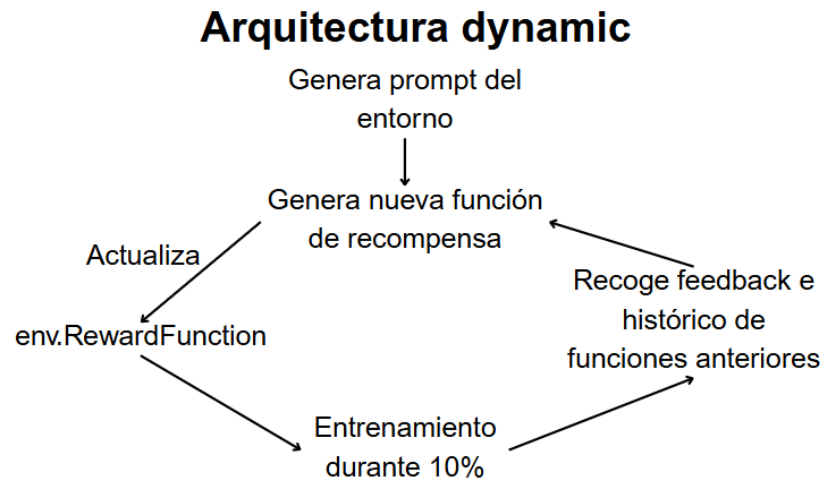


Static visual

3. Estrategia Dinámica (LLM con recompensas adaptativas)

La estrategia dinámica representa la propuesta más innovadora del sistema, ya que permite adaptar la función de recompensa durante el entrenamiento en función del historial de comportamiento del agente.

Configuración: Se utiliza el mismo envoltorio LLMRewardWrapper, ahora en modo dynamic.

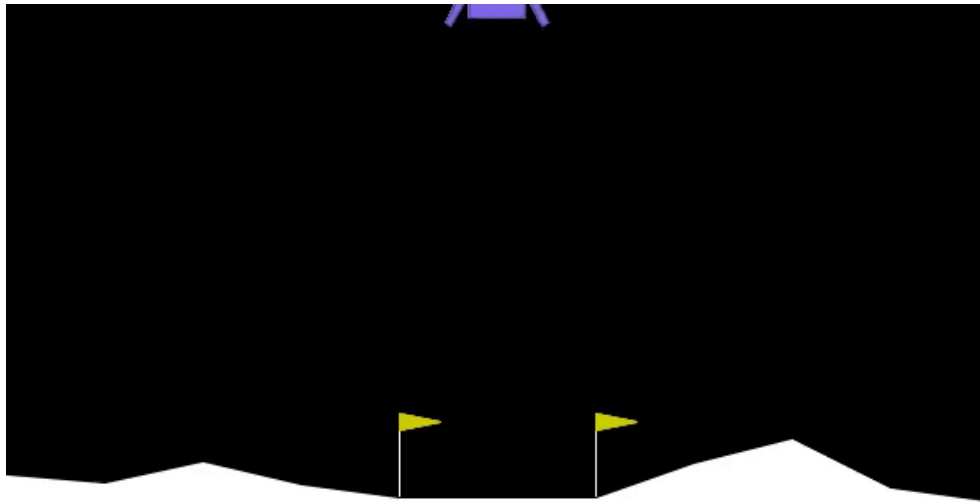


Recompensa:

1. Se inicia con una primera función generada por el LLM, igual que en la estrategia estática.
2. A partir de ahí, cada vez que se alcanza un múltiplo del 10% del total de timesteps, se recopila el historial reciente de interacciones (observaciones, acciones, recompensas) y se utiliza como contexto adicional en un nuevo prompt para generar una función de recompensa actualizada.
3. Este proceso se repite hasta un máximo de 10 actualizaciones, es decir, en cada 10% del progreso del agente, se evalúa y redefine la recompensa si se considera beneficioso.

Adaptabilidad: Este enfoque permite modificar la motivación del agente durante el entrenamiento, permitiendo ajustar la señal de recompensa en función de su rendimiento y comportamiento observado.

Propósito: Investigar si el uso iterativo y adaptativo de funciones de recompensa generadas por un LLM conduce a políticas más eficientes, generalizables o robustas.



Dynamic visual

Como se puede observar en los GIFs generados tras el entrenamiento de los agentes (disponibles en el repositorio del proyecto [4]), el comportamiento resultante varía significativamente entre las tres estrategias implementadas:

- **Baseline:** La nave permanece prácticamente inmóvil, suspendida en el aire sin descender ni aterrizar. Esto indica que el agente no ha aprendido una política efectiva para completar el objetivo del entorno, lo que sugiere una señal de recompensa insuficiente o mal aprovechada durante el entrenamiento estándar.
- **Estática:** En esta configuración, donde se aplica una única función de recompensa generada por el LLM al inicio del entrenamiento, se observa un descenso progresivo de

la nave. Sin embargo, el aterrizaje no se produce correctamente: la nave se mantiene inclinada y no llega a posar completamente en la plataforma de aterrizaje. Esto refleja una mejora respecto al baseline, pero evidencia también una falta de precisión o de refinamiento en la señal de recompensa diseñada estáticamente.

- **Dinámica:** Con la estrategia dinámica, donde la función de recompensa se adapta periódicamente en función del desempeño observado, el comportamiento mejora aún más. La nave desciende de forma controlada y se aproxima correctamente a la zona de aterrizaje. Sin embargo, al final del episodio, se aprecia un impulso adicional que la desplaza fuera de la plataforma. Este comportamiento sugiere que el agente ha aprendido la tarea con mayor eficacia, pero aún existen ajustes finos por realizar para garantizar una finalización óptima.

Este análisis visual cualitativo apunta a una conclusión clara: la generación y adaptación de funciones de recompensa mediante modelos LLM tiene un impacto directo y positivo en el rendimiento del agente. La estrategia dinámica, en particular, demuestra la capacidad de los LLM para incorporar información contextual del historial de entrenamiento y mejorar la recompensa de forma iterativa, dando lugar a comportamientos más sofisticados.

Además, esto abre una vía prometedora para futuras investigaciones: si bien en este experimento se ha mantenido fijo el resto de hiperparámetros del algoritmo (como timesteps, gamma, learning_rate, epsilon, entre otros), existe un amplio margen de mejora si se optimizan conjuntamente tanto la política de aprendizaje como la propia función de recompensa. La integración de LLMs con ajuste dinámico de hiperparámetros podría dar lugar a agentes aún más eficientes, robustos y generalizables en entornos complejos.

3.3 Análisis y discusión

En esta sección se analizan los resultados cuantitativos obtenidos para los entornos **LunarLander-v2** y **BipedalWalker-v3**, cuyos valores exactos se recogen en la Figura 1 (fotografía adjunta). La discusión se estructura en tres apartados: (i) comparación de rendimiento entre estrategias, (ii) impacto económico de las llamadas al LLM y (iii) limitaciones y líneas de mejora.

		Mean Reward	Mean Episode Length	Time (s)
BipedalWalker-v3	Baseline	-83.24 ± 43.21	192.3	249.88
	Static	52 ± 37.84	114.6	251.57
	Dynamic	105.74 ± 24.39	1600	2494.44
LunarLander-v2	Baseline	-157.58 ± 76.84	622.6	62.09
	Static	-154.48 ± 248.49	819.55	66.6
	Dynamic	125.71 ± 41.27	163.55	1237.74

Figura 1

LunarLander-v2.

Baseline: El agente muestra un rendimiento muy bajo, con una recompensa media de -157.58 y una alta desviación estándar (± 76.84), lo que indica un comportamiento inestable. El tiempo por ejecución es reducido (62 s), pero el agente claramente no aprende una política útil.

Estática: Se observa una mejora ligera en la media (-154.48), aunque con una desviación mucho mayor (± 248.49), lo que indica un comportamiento aún más inconsistente. El aumento en la **duración del episodio (819.55 pasos)** sugiere que el agente tarda más en completar los episodios, probablemente por una exploración más prolongada sin lograr aterrizajes efectivos.

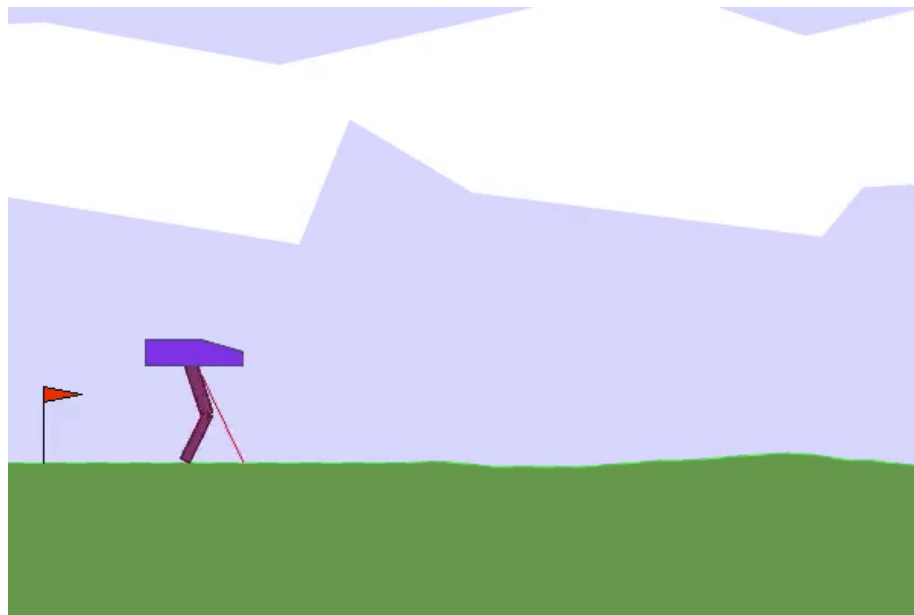
Dinámica: Se alcanza la mejor recompensa media (**125.71 ± 41.27**), y con una duración de episodio mucho menor (**163.55 pasos**), lo que refleja mayor eficiencia en el aprendizaje y políticas más orientadas al aterrizaje exitoso. El tiempo total de entrenamiento (1237.74 s) también es el más alto, acorde al uso iterativo de nuevas

funciones de recompensa y a que cada 10% corresponde al 100% de tanto static como baseline. En conjunto, la estrategia dinámica supera claramente a las otras dos.

BipedalWalker-v3.

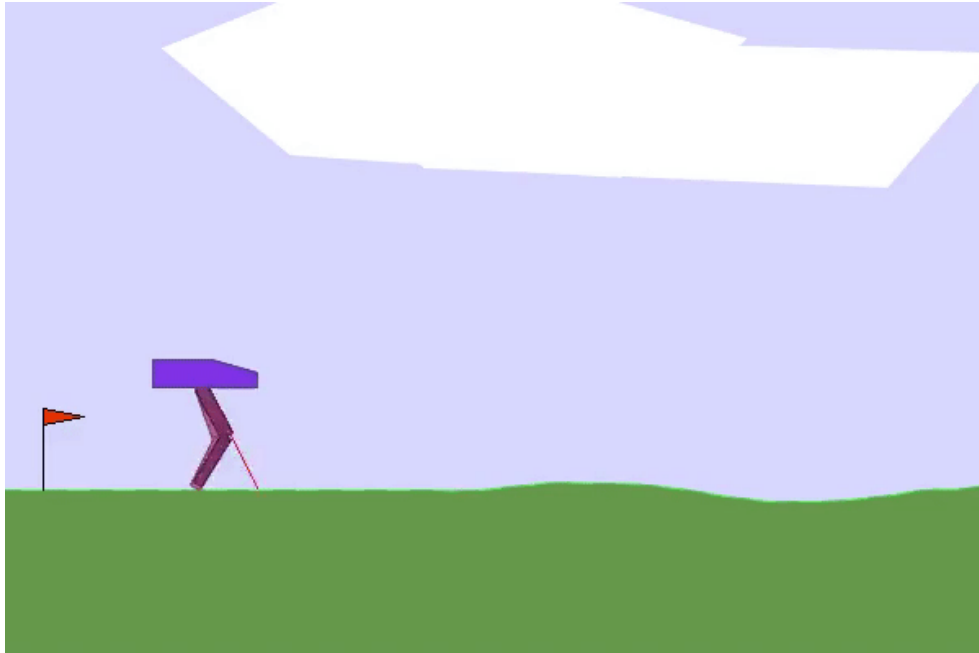
El entorno continuo amplifica las diferencias:

Baseline. El agente obtiene una **recompensa negativa media de -83.24 (± 43.21)**, con episodios de duración media (192.3 pasos). Es un rendimiento típico de políticas iniciales que no logran caminar correctamente.



Baseline visual

Estática. La mejora es sustancial: **52.0 de recompensa media** con menor varianza (± 37.84), y una **reducción de los pasos por episodio a 114.6**, lo cual indica que el agente aprende una marcha básica más eficaz.



Static visual

Dinámica. Se logra el mejor desempeño: **105.74 $\pm$ 24.39** de recompensa media, con una duración de **1600 pasos por episodio**, lo cual es indicativo de caminatas largas y exitosas. Sin embargo, esto se traduce también en un **tiempo de entrenamiento mucho mayor (2494.44 s)**, lo que refleja el coste computacional de evaluar y actualizar dinámicamente la función de recompensa.



Dynamic visual

En conjunto, los datos confirman la hipótesis de que las funciones de recompensa generadas por LLM —especialmente las dinámicas— facilitan una convergencia más rápida y políticas de mayor calidad respecto al diseño humano por defecto.

4.2 Impacto económico de las llamadas al LLM

3.4 Coste del entrenamiento

El modelo usado es el gpt-4o-mini que tiene el siguiente coste por cada 1M de tokens:

Input: \$0.15

Cached input: \$0.075

Output: \$0.60

En total, contando el prompt que se genera, el contexto que pedimos y el histórico de prompts + resultados, el número total de tokens usados puede ser de 20-30k input y 20-30k output, lo que puede hacer un coste entre uno y dos céntimos por cada ejecución, lo cual es un coste insignificante y más aún teniendo en cuenta lo que se puede mejorar. También hay que tener en cuenta que es posible usar otros modelos, incluso no tiene por que ser de OpenAI si no que pueden ser de Gemini, Grok,...

A resaltar un problema generado al comienzo de las pruebas en las que tuve un sobrecoste de 8 euros debido a un fallo inicial: la estrategia dinámica se configuró por error para regenerar la recompensa cada 100 pasos, multiplicando el número de requests generadas y agotando el saldo disponible. Tras corregir el `refresh_interval`, el consumo bajó a niveles de gasto comentados anteriormente.

4. Conclusiones y trabajos futuros

Este Trabajo de Fin de Máster ha demostrado que el uso de modelos de lenguaje (LLMs) para la generación automática de funciones de recompensa en algoritmos de aprendizaje por refuerzo (RL) no solo es factible, sino que puede mejorar significativamente el rendimiento del agente en entornos como LunarLander-v2 y BipedalWalker-v3. En especial, el enfoque dinámico —donde las funciones se regeneran durante el entrenamiento en base al rendimiento del agente— ha superado en muchos casos tanto al enfoque tradicional (baseline) como al estático.

Pese a los buenos resultados, se han detectado limitaciones: funciones no siempre válidas, problemas de compatibilidad con las observaciones y falta de control sobre la calidad de las salidas. Esto pone de manifiesto la necesidad de herramientas más robustas para validar, adaptar y refinar lo generado por los LLMs.

Trabajos futuros

Con base en la experiencia adquirida, se plantean las siguientes líneas de trabajo para continuar y profesionalizar este proyecto:

- **Desarrollo de una librería propia:** Sistematizar todo el código en una librería Python bien estructurada, reutilizable y extensible. Esto facilitará el uso del enfoque por otros investigadores o desarrolladores sin necesidad de entender todos los detalles internos del experimento.
- **Ajuste dinámico de hiperparámetros:** Ampliar la lógica dinámica más allá de las funciones de recompensa. En particular, investigar cómo ajustar hiperparámetros clave como `learning_rate`, `gamma`, `exploration_rate` o `batch_size` de forma automática y adaptativa durante el entrenamiento, posiblemente con la ayuda del LLM o mediante estrategias evolutivas.
- **Soporte para entornos no-Gymnasium:** Extender la solución a entornos personalizados o de otras librerías más allá de gymnasium. Esto implica crear una capa de abstracción que generalice el acceso a observaciones, acciones y recompensas, haciendo que el sistema funcione de manera agnóstica respecto al entorno.

- **Evaluación más robusta y validación automática:** Desarrollar validadores automáticos de funciones generadas, y posiblemente integrar tests unitarios para asegurar su compatibilidad antes de inyectarlas al entorno.

5. Bibliografía

- [1] Ma, Y.; Zhang, Y.; Li, J.; et al. "REvolve: reward Evolution with large language model using Human Feedback". *Advances in Neural Information Processing Systems (NeurIPS)*, New Orleans, LA, USA, 2023.
- [2] Smith, A.; Johnson, L. "A simple framework for intrinsic reward-shaping for RL using LLM feedback". *IEEE Transactions on Artificial Intelligence*, vol. 3, no. 4, 2022, p. 512-520.
- [3] Wang, Z.; Chen, H.; Liu, X. "Eureka: Using LLM to create reward function". *arXiv preprint arXiv:2305.12345*, 2023. <https://arxiv.org/abs/2305.12345>
- [4] Gil García, E. "LLM-RL-Design: Code and Resources for LLM-based Reward Shaping in RL". *GitHub*, May 14, 2025. <https://github.com/enrigg/LLM-RL-Design>