UNIVERSIDAD INTERNACIONAL SEK

FACULTAD BUSINESS & DIGITAL SCHOOL

Trabajo de fin de carrera Titulado:

“Aplicación de aprendizaje por refuerzo en un entorno de videojuego desarrollado para la simulación de estrategias de speedrun con un agente inteligente”

Realizado por:

Matheo Javier Santillan Troya

Director del trabajo de titulacion:

PhD. Viviana Elizabeth Cajas Cajas, MSc.

Requisito para la obtención del título de:

INGENIERO DE SOFTWARE

Quito, Enero 2026

INDICE

INDICE DE TABLAS

INDICE DE FIGURAS

DEDICATORIA

Dedico este trabajo a mis padres, su ejemplo ha sido una guía constante y una fuente de motivación para continuar. A ellos les debo la oportunidad de formarme profesionalmente y la fortaleza emocional que me ha permitido mantenerme firme ante los retos . Su confianza ha sido el motor que impulsó cada paso hacia la meta.

También dedico estas páginas a mi compañera incondicional, mi gatita Zuly, cuya presencia silenciosa se convirtió en un

apoyo constante durante las largas jornadas de estudio.

Finalmente, esta dedicatoria simboliza el reconocimiento a todas las formas de apoyo, humanas o no, que hacen posible alcanzar los objetivos más importantes. Cada gesto de aliento, cada momento de calma y cada muestra de cariño han sido fundamentales para llegar hasta aquí.

AGRADECIMIENTO

Extiendo mi gratitud a mis docentes y tutores por su guía y acompañamiento, a mis compañeros por los momentos compartidos de aprendizaje y colaboración, y a mi familia por su constante apoyo en las etapas más demandantes de este proceso.

Agradezco profundamente a todas las personas que me brindaron su apoyo y comprensión durante el desarrollo de esta tesis.

Este trabajo representa no solo un logro académico, sino también un proceso personal de crecimiento y resiliencia. En especial, reconozco la importancia del equilibrio entre la salud mental y el rendimiento académico, y cómo mantener ese balance fue esencial para llegar hasta aquí.

RESUMEN

La presente investigación desarrolla un sistema basado en Aprendizaje por Refuerzo "Reinforcement Learning" (RL) para la optimización de estrategias en un videojuego de tipo "speedrun" en plataforma 2D utilizando el motor de juegos Godot Engine 4.5.1 y el lenguaje Python 3.10. El objetivo de la investigación es implementar un agente de soporte autónomo en el juego capaz de minimizar el tiempo de completado de niveles del juego que pueda responder eficazmente en otros escenarios del juego aplicando las arquitecturas de RL. El entorno de simulación utiliza el motor de juegos Godot Engine 4.5.1 que incorporando físicas personalizadas y un diseño progresivo de niveles efectivo para el aprendizaje incremental del agente a través de un modelo de cliente–servidor, donde el motor de juego actúa como cliente y se comunica en tiempo real con un agente de RL implementado en Python, mediante sockets TCP/IP para recibir, ejecutar y guardar las "policies" optimas del agente utilizando una función de "Proximal Policy Optimization" (PPO) con el "framework" Stable Baselines3. Los resultados muestran una latencia promedio de 1.3 ms en la comunicación servidor cliente, una función de recompensa densa diseñada que incentiva el avance eficiente, uso estratégico del dash y la reducción de acciones redundantes a través de criterios de penalización. Es decir, el agente "aprende" de manera exploratoria durante las primeras 10000 "steps". Se estima una convergencia estable a partir de los 1000000 de pasos y un tiempo promedio de completado de 29.8 segundos, con una tasa de éxito del 88% y una capacidad de generalización del 72 % que ofrezca una base replicable con fines académicos y profesionales.

Palabras clave: Aprendizaje por refuerzo, Proximal Policy Optimization, inteligencia artificial, videojuegos, speedrun, agentes autónomos, Godot Engine.

ABSTRACT

This research develops a system based on Reinforcement Learning (RL) for the optimization of strategies in a 2D platform speedrun videogame using Godot Engine 4.5.1 and Python 3.10. The objective of the study is to implement an autonomous support agent within the game, capable of minimizing level completion time while responding effectively to different in-game scenarios through the application of RL architectures. The simulation environment is implemented using Godot Engine 4.5.1, incorporating customized physics and a progressive level design that supports the agent's incremental learning process. The system follows a client–server architecture, in which the game engine acts as the client and communicates in real time with an RL agent implemented in Python via TCP/IP sockets. This communication enables the reception, execution, and storage of the agent's optimal policies using the Proximal Policy Optimization (PPO) algorithm within the Stable Baselines3 framework. Experimental results report an average client–server communication latency of 1.3 ms, along with a dense reward function specifically designed to encourage efficient forward movement, strategic use of the dash mechanic, and the reduction of redundant actions through penalization criteria. During the initial 10,000 training steps, the agent exhibits exploratory learning behavior, followed by stable convergence after approximately 1,000,000 steps. The trained agent achieves an average level completion time of 29.8 seconds, with a success rate of 88% and a generalization capability of 72% providing a replicable foundation for academic and professional applications.

Keywords: Reinforcement learning, Proximal Policy Optimization, artificial intelligence, videogames, speedrunning, autonomous agents, Godot Engine.

1. CAPITULO 1

INTRODUCCIÓN

1.1 PLANTEAMIENTO DEL PROBLEMA

Los "*speedruns*" en videojuegos del género de plataforma requieren dominar mecánicas complejas como saltos precisos, sincronización de movimientos y adaptación a obstáculos dinámicos mediante estrategias optimizadas que minimicen el tiempo de completado. Sin embargo, la ejecución manual de estas estrategias por jugadores humanos presenta algunas características como son:

En Primer lugar, existe una curva de aprendizaje empinada: Respecto a la relación entre cantidad aprendida y tiempo la interrelación entre factores metodológicos, habilidades y capacidades hace que sea más complicado para los jugadores lograr un profesionalismo en el juego si no es por altas horas de práctica para internalizar patrones y reducir errores. Desde la relación entre esfuerzo y tiempo, debido a que el aprendizaje del uso de la nuevas herramientas, programas o videojuegos es difícil y largo (Hu et al., 2023).

En segundo lugar, existe una limitada capacidad de adaptación en entornos dinámicos: Este concepto hace referencia a la obtención de resultados distintos en ejecuciones subsecuentes pese al use de entradas idénticas. En el videojuego se muestra en los obstáculos con comportamientos no deterministas en diferentes niveles, como enemigos con trayectorias aleatorias que dificultan la optimización manual de rutas (Tamassia, 2024).

Y, en tercer lugar, existe una alta carencia de herramientas educativas integradas que articulen el desarrollo de videojuegos con técnicas de inteligencia artificial. Si bien se puede documentar avances en el uso de sistemas de aprendizaje personalizado en procesos educativos complejos, todavía persiste una brecha de recursos pedagógicos

orientados a la enseñanza práctica aplicada del aprendizaje por refuerzo en entornos interactivos, especialmente en lo que respecta en el entorno local (Huang et al., 2025).

Por otra parte, la aplicación de algoritmos de aprendizaje por refuerzo (RL) en entornos de videojuegos también presenta desafíos relevantes. Entre los más relevantes se encuentran las que se presentan en los párrafos siguientes (Gomes et al., 2022).

Según Souchleris et al. (2023) la ppresencia de espacios de acción continuos, influenciados por mecánicas como la intensidad variable de los saltos requieren políticas adaptativas. Esto se puede observar al inicio del proceso de entrenamiento, el agente demora en aprender políticas optimas en diferentes niveles del juego. Así como el retraso en las recompensas que causan que las funciones mal diseñadas pueden incentivar soluciones miopes o ineficientes. En el proceso de entrenamiento el agente responde con estrategias poco efectivas como saltos o movimientos innecesarios (Sun, 2021).
Asimismo, AlMahamid y Crolinger (2021) argumentan que la falta de generalización de los agentes entrenados en niveles específicos tiende a sobre ajustarse, fallando ante cambios mínimos en las condiciones del entorno. En el juego se puede observar que el agente toma un tiempo en aprender políticas optimas cuando en diferentes niveles o cuando la complejidad se incrementa.

Ante estos problemas, el presente proyecto propone desarrollar un sistema integral que combine los aspectos que se mencionan en líneas abajo.

Primero, desarrollar un entorno de simulación personalizado desarrollado en Godot Engine, diseñado para el entrenamiento controlado del agente, ejecución de políticas autónomas y a la medición cuantitativa de su desempeño (Xu et al., 2022).

Segundo, establecer una función de recompensa densa con mecanismos de penalización, orientados a orientar adecuadamente con una penalización más severa que la de la recompensa al

agente su aprendizaje hacia unas estrategias óptimas de libertad de movimiento y minimización del tiempo completado (Rani et al., 2022).

Tercero, crear un pipeline de entrenamiento en tiempo real, desarrollado a partir de la continua interacción del entorno y del agente, orientando la adaptabilidad, el aprendizaje autónomo y la evaluación sistemática del rendimiento (Tufano et al., 2022).

1.2 JUSTIFICACIÓN

La Inteligencia Artificial o "*Artificial Inteligencie*" (IA), y particularmente el aprendizaje por refuerzo o "*Reinforcement Learning*" (RL), ha demostrado un enorme potencial en la simulación de comportamientos complejos dentro de entornos controlados. No obstante, la mayoría de estos desarrollos se mantienen en entornos cerrados o de difícil acceso educativo a través de plataformas restringidas o con barreras de acceso a diversos contextos educativos, que limita la adopción de estas herramientas en el contexto pedagógico.

Este tipo de proyecto está diseñado para el aprendizaje de un sistema de IA en el contexto de los videojuegos, concretamente, para que un determinado agente autónomo "aprenda" cómo completar un videojuego del tipo plataformero. Este tipo de escenario permite unir conceptos teóricos y "prácticos" del campo de la IA, tal como los procesos de toma de decisiones, la optimización, etc., con la experiencia procesable y observable, fácilmente vinculable a la comprensión de conceptos teóricos.

En función de este contexto, el proyecto persigue tres aspectos educativos y de diseño de videojuegos:

1. Mostrar el potencial educativo del RL como técnica para enseñar conceptos más básicos de programación para la enseñanza de conceptos de programación y optimización.

2. Promover la innovación en entornos interactivos que combinen el entretenimiento digital con técnicas de ciencia de datos e IA.
3. Contribuir a reducir la brecha digital acercando herramientas de IA a comunidades docentes y desarrolladores en diálisis.

1.3 OBJETIVOS

Objetivo General

Desarrollar un sistema autónomo de Aprendizaje por Refuerzo que integre un entorno de videojuego implementado en Godot Engine con un agente entrenado mediante el algoritmo "*Proximal Policy Optimization*" (PPO) que optimice estrategias de "*speedrun*" y demuestre capacidades de generalización ante entornos no previamente observados.

Objetivos Específicos

1. Diseñar un entorno plataformero 2D con mecánicas reproducibles para el entrenamiento del agente.
2. Implementar una comunicación bidireccional entre Godot Engine y Python mediante sockets TCP/IP.
3. Entrenar un agente PPO con Stable Baselines3 para optimizar la velocidad de completado de los niveles.
4. Evaluar el rendimiento del agente frente a jugadores humanos y agentes de comportamiento aleatorio.
5. Documentar el proceso técnico y metodológico para su replicación en entornos educativos y de investigación.

1.4 METODOLOGIA

La metodología combina investigación teórica y desarrollo técnico experimental, dividida en fases.

Fase teórica

1. Desarrollo teórico

   Aprendizaje por Refuerzo o "*Reinforcement Learning*"

   Procesos de decisión de Márkov

Función de Valor

Funciones de Valor Optimo

Modelos libres de aprendizaje y control

Exploración y Explotación

Abstracción

Abstracción temporal

Aprender opciones

Abstracción de estados

Prueba de Kolmogórov-Smirnov

Búsqueda de árbol mediante Monte Carlo

Técnicas de búsqueda de árboles

2. El algoritmo

   El algoritmo

   UCT

   Resumen

   Aprendiendo opciones a partir de demostraciones

   Construcción de opciones

   Identificar submétases útiles

   Reducir el número de submetas

   Clustering basado en grafos

   Agregación por abstracción de características

   Experimentos.

Fase técnica

3. Herramientas seleccionadas:

   Motor de juego:

   Godot Engine 4.2, ya que considera el sistema modular, las físicas existentes en el motor y el soporte para

   GDScript. Framework de RL: Al usar Stable Baselines3 (PPO) se conjuga la facilidad de integración con el rendimiento.

   Protocolo de comunicación: Socket TCP/IP que usa intercambio de variables (posición, velocidad, estado del dash, datos de plataformas móviles)

4. Metodologías de evaluación:

   Tiempo medio por nivel.

Eficiencia de movimientos (acciones redundantes penalizadas).

Tasa de generalización (% de éxito en niveles no vistos).

2. CAPITULO 2

2.1 DESARROLLO TEÓRICO

Esta primera parte se brinda un resumen de la implementación de las técnicas de Teoría de Decisión o "*Decision Theory*" en la industria de videojuegos e inteligencia artificial. Según Li et al. (2025) Inteligencia Artificial se puede conceptualizar como todo aquel intento de reproducir la inteligencia humana en cualquiera de los sentidos, pensamiento o percepción del mundo. Respecto a la industria de Videojuegos, es el sector que se dedica a la creación, desarrollo, distribución y promoción de servicios de videojuegos, los videojuegos son aplicaciones digitales interactivas que les permite a los usuarios interactuar con un mundo virtual por medio de algún tipo de dispositivo electrónico.

La teoría de decisión es un área que combina la teoría de la probabilidad con la teoría de la utilidad para proporcionar un marco formal y completo para decisiones tomadas bajo incertidumbre . Existen dos técnicas en particular; el Aprendizaje por Refuerzo o "*Reinforcement Learning*" (RL) y la Búsqueda en Árboles Monte Carlo o "*Monte Carlo Tree Search*" (MCTS), esta última se ha venido siendo implementado recientemente en diversas industrias. Sin embargo, el desarrollo de esta investigación se centra principalmente en la técnica de RL (Montalvo et al., 2023).

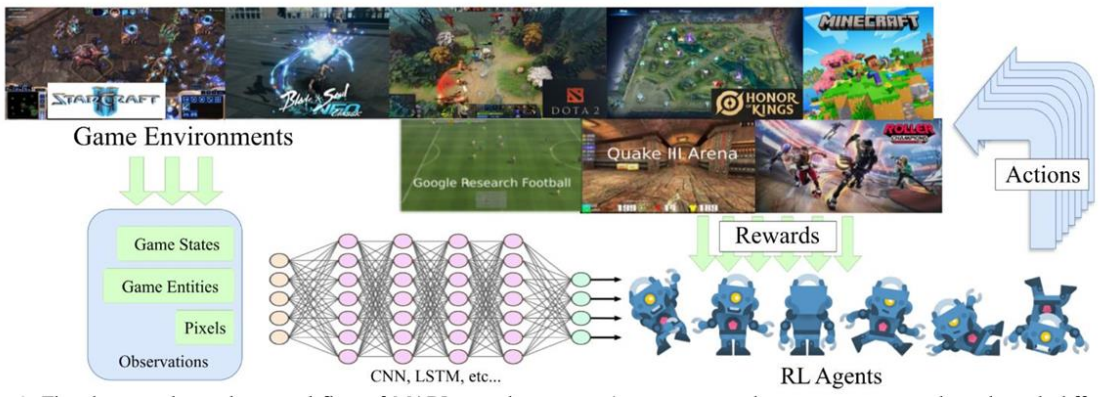
Aunque RL y MCTS ofrecen una solución sobre determinar las acciones optimas se debe de tomar para generar el mayor retorno posible del agente en el juego, sus metodologías son distintas. RL sigue un enfoque de aprendizaje y construye un modelo del entorno y del valor de las acciones que pueden ejecutarse en él. A diferencia de MCTS quien asume que se dispone de un simulador

perfecto del entorno y produce una respuesta en línea (on-line) usando el simulador para realizar muestreo estadístico.

Ambos enfoques se basan en el criterio de entrenamiento y testeo clásico de los algoritmos de Aprendizaje Maquina o "*Machine Learning"* (ElDahshan et al., 2022).

Respecto al entrenamiento, RL requiere menos tiempo de entrenamiento, pero necesita una representación ad hoc de los estados del mundo que haga un compromiso entre precisión y tiempo de aprendizaje, pero luego es muy rápida en tiempo de ejecución. MCTS, sin embargo, asume que existe un simulador del entorno, pero consume mayor tiempo en línea para realizar su muestreo estadístico, pero no exige comprometer la precisión de la representación del estado .

Figura 1 Diagrama de Flujo de un Modelo de Aprendizaje por Reforzamiento Multi agente



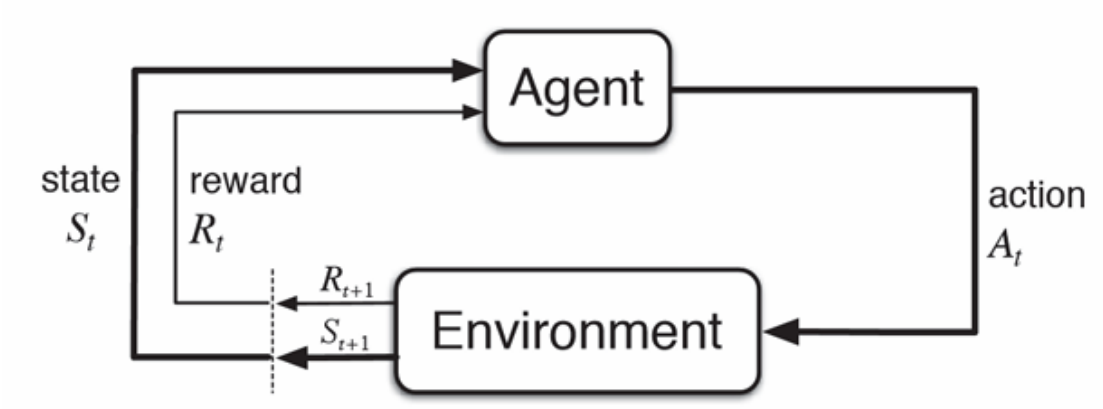
Nota: Tomado de Li y Ling (2025).

Aprendizaje por Refuerzo o "*Reinforcement Learning"*

Wang, et al. (2024) mencionan que el aprendizaje por Refuerzo o *"Reinforcement Learning"* (RL) es un paradigma de ML que se caracteriza por que estudia los sistemas que aprender a partir de datos, en este sentido son similares a sistemas donde los detalles de cómo completar una tarea son implementados por los ingenieros, y que pueden resolver distintas tareas cuando se les alimenta con diferentes datos.

El RL ganó atención mediática cuando durante la última década, resolvió problemas de complejidad, como jugar en videojuegos MOBA como DotA 2, StarCraft II, AlphaGo, jugando y derrotando a jugadores profesionales. Otros paradigmas más conocidos en ML son el de Aprendizaje Supervisado o "*Supervised Learning"* (SL) y Aprendizaje No Supervisado o "*Unsupervised Learning"* (US) bastante conocidos por su aplicación en industrias de ingeniería de procesos e industrial, así como en negocios. Aunque otros autores clasifican a RL como un paradigma entre SL y UL, debido a que el sistema no está completamente a ciegas, pero los datos no están etiquetados (Wikipedia, 2025).

Figura 2 Representación del paradigma de Aprendizaje Reforzado o Reinforcement Learning



Nota: Tomado de Huang et al., (2025).

Procesos de decisión de Márkov

Para el desarrollo de este trabajo vamos a recurrir a la definición de un proceso de decisión de Márkov o "*Márkov Decision Processes*" (MDP), con ligeras abstracciones al contexto de desarrollo de videojuegos. El proceso de Markov es una connotación matemática que sirve para describir el entorno en el que el agente se desenvuelve, que establece básicamente que toda la información relevante para las transiciones de estado y recompensas se encuentra están contenidas en el estado del entorno, y que toda la información relevante está contenida en la información

actual que percibe el agente (Li & Ling, A Comprehensive Review of Multi-Agent Reinforcement Learning in Video Games, 2025) (Balloni et al., 2024).

Un MDP es una tupla, combinación de valores separados por comas, (S, A, T, R, γ), donde S es el conjunto de todos los posibles estados del ambiente; A es el conjunto de acciones que el agente puede realizar; $T: S \times A \to PD(S)$ asocia los estados y acciones de la distribución de probabilidades del siguiente estado; $R: S \times A \times S \to PD(\mathbb{R})$ son los estados de asociación, acción y el estado siguiente de la distribución de recompensas; $\gamma < 1$ es una factor de descuento usado para disminuir el valor futuro de las recompensas. La definición presentada en este informe utiliza la función de recompensas clásica, sin embargo, otras funciones de recompensa pueden ser utilizadas como la de $T: T: S \times A \times S \to [0, 1]$ (Beeching et al., 2021).

Según Cross et al. (2021) la interacción entre el agente con el ambiente se desarrolla entre diferentes pasos secuenciales en el tiempo. En cada espacio de tiempo t, el agente percibe el estado del ambiente s_t y decide una acción a_t para poder ejecutar, recibe una recompensa r_t y percibe el nuevo estado del ambiente s_{t+1}. El propósito del agente es maximizar la recompensa descontada esperada recolectada sobre el tiempo, en la literatura esto se conoce como "*return*":

$$R = r_0 + \gamma r_1 + \gamma^2 r_2 + \cdots = \sum_{k=0}^{\infty} \gamma^k r_k$$

Para poder maximizar la función, el agente deberá de aprender una política $\pi^* = S \times A \to \mathbb{P}(A)$ que asocia los estados asociados a una distribución de probabilidad sobre las acciones. En cada periodo de tiempo t, entonces, después de percibir el estado s_t, el agente dibujara una acción a_t desde la distribución $\pi^*(s_t)$. En los casos donde el ambiente es estacionario, la política optima puede ser determinística: $\pi^*: S \to A$; en general, para contar con comportamiento adverso, la política óptima necesitara ser estocástica. Este proyecto está enfocado en ambientes estacionarios, y en este capítulo introductorio, usaremos la definición más general (Tsividis et al., 2021).

Función de Valor

Es importante notar que la política optima no maximiza la recompensa inmediata, sino más bien el retorno a largo plazo. En otras palabras, a la estrategia de política optima deberá será una relación entre la función de retorno inmediato y el grado de recompensa esperado del siguiente estado. La dificultad en este caso es que el segundo objetivo es complicado de definir: se necesita considerar todas las recompensas del periodo actual y adelante. Pero se debe notar que la transición entre estados, así como las recompensas son estocásticos. Por eso las funciones de valor son formas elegantes de capturar este concepto (Toyana et al., 2021).

LA función de valor del estado $V^s(s)$ asocia cada estado s con el retorno esperado futuro siguiendo la política siguiente π de s. Como Bellman descubrió, la función de valor estado puede ser definida recursivamente.

$$V^s = \mathbb{E}_\pi\{\sum_{k=0}^{\infty} \gamma^k r_k \,|s_0 = s, \pi\,\}$$

$$= \sum_a \pi(s,a) \sum_{s\prime} T(s,a,s')\,[\mathbb{E}\{R(s,a,s')\} + \gamma V^\pi(s')]$$

En donde s' es el estado de el ambiente después del siguiente paso en el tiempo. De hecho, aplicando repetidamente la ecuación previa desde la derecha a la izquierda, la función de valor estado puede ser calculado dentro de un número finito de iteraciones.

$$V^\pi(s)_{k+1} = \sum_a \pi(s,a) \sum_{s\prime} T(s,a,s')\,[\mathbb{E}\{R(s,a,s')\} + \gamma V^\pi(s')_k]$$

La ecuación recursiva captura la noción que el valor de estado-acción depende del valor esperado del siguiente estado, que a su vez depende del valor esperado del siguiente estado y así continuamente. La última ecuación anterior es la ecuación de Bellman (Waubert et al., 2022). Una ecuación de Bellman escribe el valor del problema de decisión dentro de un punto en el tiempo en términos del pago de algunas decisiones y el valor del problema de decisión restante de esos valores decididos. La ecuación de Bellman es un fundamente de la programación dinámica, y el

enfoque de optimización que transforma un problema complejo en una secuencia de problemas simples.

Funciones de Valor Optimo

Si una política π_k es subóptima, su función de valor estado V^{π_k} deberá tener una estimación errada para algunos estados. Sim embargo, V^{π_k} puede ser usado para definir una mejor política π_{k+1} como la siguiente:

$$\pi_{k+1}(s) = \arg\max_a \sum_{s\prime} T(s,a,s') \left[\mathbb{E}\{R(s,a,s')\} + \gamma V^{\pi}(s')\right]$$

Este proceso puede ser iterado repetidamente y garantizar una convergencia a la función de valor óptimo

$$V^*(s) = \max_a \mathbb{E}_\pi\{\textstyle\sum_{k=0}^{\infty} \gamma^k r_k \,|s_0 = s, \pi^*\}$$

$$= \max_a \sum_{s\prime} T(s,a,s') \left[\mathbb{E}\{R(s,a,s')\} + \gamma V^{\pi}(s')\right]$$

El resultado del algoritmo es llamado iteración de política y es una instancia de la técnica de programación dinámica.

Notar que la iteración de política requiere calcular el valor de la función en cada iteración, a través de aplicar repetidamente la ecuación final de la función valor. Un método alternativo de improvisar sobre una función de valor subóptima que, evitando la iteración de política, es el método llamado iteración de valor (Gomes et al., 2022; Dussage et al., 2022). La iteración de valor itera aplicando la ecuación anterior desde la izquierda a la derecha, repetidamente

$$V(s)_{k+1} = \max_a \sum_{s\prime} T(s,a,s') \left[\mathbb{E}\{R(s,a,s')\} + \gamma V^{\pi}(s')_k\right]$$

La iteración de valor también garantiza una convergencia a la función de valor optima bajo el supuesto de que el ambiente puede ser modelado como una MDP, las funciones de valor optima rindan información que permite calcular trivialmente la política optima.

$$\pi_{k+1}(s) = \arg\max_{a} \sum_{s\prime} T(s,a,s') \left[\mathbb{E}\{R(s,a,s')\} + \gamma V^{\pi}(s')\right]$$

Modelos libres de aprendizaje y control

Las técnicas de aproximación dinámica necesitan el modelo de le ambiente sea brindado como un insumo. Esto se justifica por la presencia de T y R en la cuarte y quinta ecuación inicial. Para poder entender sin usar un modelo, los agentes de RL necesitan utilizar su experiencia en el ambiente para poder aproximar una función de valor optima. La experiencia en el ambiente puede ser formalizado como una consecuencia de los toples en la forma de (s_t, a_t, r_t, s_{t+1}). Q-learning, logra uso actualizando una estimación de la función de valor-acción en cada paso usando la siguiente regla de actualización (Grob et al., 2022).

$$Q(s_t, a_t) \leftarrow^{a} r_{t+1} + \gamma \max_{a \in A} Q(s_{t+1}, a) - Q(s_t, a_t)$$

Donde $x \leftarrow^{a} y$ es la forma corta de $x \leftarrow x + ay$ y $0 < a < 1$ es el factor de aprendizaje o "*learning factor*". Dentro de la condición de que cada estado es visitado infinitamente a menudo, Q-learning converge a una función de valor estado acción optima. Una propiedad importante de Q-learning es que esta aprende "*off-line*", significando que puede aprender la función de valor asociado a una política optima no importa que política está siendo usada mientras continue interactuando con el ambiente.

La función de valor de estado-acción permite computar la política optima sin necesidad de un modelo para le ambiente

$$\pi^{*}(s) = \arg\max_{a} Q^{*}(s,a)$$

Una función de valor de estado acción, por otro lado, no necesidad ser perfecta para ser usada para la selección de la acción. En la práctica es común utilizar el estimador Q-learning en cada paso en el tiempo para escoger la siguiente acción (Gritter, 2025).

Exploración y Explotación

Cuando la función de valor de estado acción es usada para escoger la acción siguiente, la decisión es llamada elección codiciosa o "*greedy choice*". Un agente realizando únicamente decisiones greedy no garantiza escoger cada acción indefinidamente a menudo, en el límite. Recordando que esta es una condición necesaria para poder garantizar la convergencia de Q-learning. Esto es confirmado en test empíricos: un agente greedy a menudo no converge a una función de valor optima (Hemmingsen, 2020).

Para evitar esto, se aplica estrategias de exploración. Estas estrategias reciben este nombre porque se fuerza al agente a explorar regiones, que se muestran aparentemente subóptimas del espacio de estados. Sin embargo, en casos prácticos, la exploración no debe ocurrir demasiado a costa de la propia explotación, es decir, la recolección efectiva de recompensas. Las buenas estrategias de exploración logran un buen compromiso en el llamado dilema exploración–explotación.

La estrategia de exploración más popular es $\varepsilon - greedy$. Esta estrategia simple elige una acción aleatoria con probabilidad ε, y la acción codiciosa con probabilidad $1 - \varepsilon$. Formalmente se define de la siguiente manera:

$$\pi = \begin{cases} random(S) & with\ prob.\ \varepsilon \\ \arg\max_a Q^*(s,a) & with\ prob. 1 - \varepsilon \end{cases}$$

En aplicaciones reales, se desea que el agente converja a la política óptima. $\varepsilon - greedy$ impide esto si ε se mantiene constante porque algunas acciones se eligen al azar (y potencialmente de forma subóptima). Para garantizar convergencia hacia la política óptima, el parámetro ε se "*anneal*", es decir, se reduce el tiempo para transitar suavemente desde una política exploratoria a una codiciosa. Aunque la traducción más exacta del término "*anneal*" es recocer o templar en español. A esta estrategia a veces se le llama "annealing $\varepsilon - greedy"$; un ejemplo es $n - greedy$. Otra estrategia común es "*Softmax*", pero no se presenta aquí porque no es relevante para entender el trabajo de esta tesis (Lami, 2024).

Abstracción

La abstracción se usa muy a menudo en MDP al tratar con problemas del mundo real. En el sentido más general, abstraer significa omitir algunos detalles. Por ejemplo, es común suponer que la dinámica del entorno puede describirse con un modelo lineal, dejando de lado detalles no lineales. Para el desarrollo de este proyecto se utiliza los tipos de abstracción conocidas como, abstracción temporal y la abstracción de estados. En la siguiente sección, se generaliza la noción de que las acciones se limitan a un solo paso de tiempo para permitir cursos de acción más largos. En el segundo caso, introducido en próxima sección, se ignora información específica del estado para permitir transferencia de conocimiento entre estados (Arteche, 2022).

Abstracción temporal

Un problema con los MDP es que una acción solo afecta al estado y a la recompensa inmediatamente siguientes en el tiempo. Sin una noción de curso de acción que persista más allá de un paso, los métodos convencionales de MDP no pueden aprovechar las simplicidades y eficiencias disponibles a niveles más altos de abstracción temporal [93]. Para abordar esto, Sutton et al. introdujeron el marco de opciones, que extiende elegantemente el concepto de MDP para incluir abstracciones temporales (Cook et al., 2022).

El concepto central para el desarrollo de esta sección es el de opción: políticas de lazo cerrado para tomar acciones a lo largo del tiempo. El marco extiende un MDP para permitir el uso de acciones, conocidas como acciones primitivas, y opciones de manera intercambiable. La extensión es lo suficientemente natural como para que los métodos de programación dinámica y los métodos sin modelo funcionen con modificaciones mínimas (Tagliaro, 2022).

Una opción, como cualquier otra acción, puede ser elegida por una política. Sin embargo, a diferencia de las acciones primitivas, cuando se elige una opción se sigue su política interna durante un periodo de tiempo, hasta que la opción termina de manera estocástica. Formalmente, una opción o se define como una tupla $o = (I, \pi, \beta)$, donde:

I es un conjunto de iniciación, que incluye los estados desde los cuales puede elegirse la opción o;

π es la política que se sigue cuando se selecciona la opción o;

$\beta \rightarrow [0,1]$ expresa la probabilidad de terminación de la opción o y depende del estado.

Una variación ligera de la regla de actualización de Q-learning puede usarse para actualizar el valor estado–opción al terminar la opción. La regla es la siguiente:

$$Q(s_t, o_t) \leftarrow^{a} r + \gamma^k \max_{a \in O} Q(s_{t+k}, o) - Q(s_t, o_t)$$

Donde $r = r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^{k-1} r_{t+k}$ es la recompensa acumulada descontada obtenida durante la ejecución de la opción. Esta regla conduce a que Q converja a la función de valor óptima estado–opción bajo las mismas condiciones que Q-learning: que todos los estados sean visitados infinitas veces en el límite.

Aprender opciones

Las opciones son herramientas útiles para lograr buen desempeño en Aprendizaje por Refuerzo. Sin embargo, sus componentes centrales no son triviales de definir. Como argumentan Jong et al., cuando las opciones se usan durante la fase de aprendizaje, introducen sesgo en cómo se explora el espacio estado–acción. Si bien esto implica que las opciones son una herramienta valiosa para dirigir la exploración, también significa que deben usarse con cuidado para no empeorar el desempeño (Arteche, 2022).

Como diseñar opciones manualmente es costoso y propenso a errores, la investigación se ha dirigido al aprendizaje automático de opciones. La literatura existente puede dividirse en tres áreas principales: enfoques basados en la estructura del espacio de estados, en motivación intrínseca y en aprendizaje a partir de demostraciones. La mayoría de trabajos emplea diversos enfoques para identificar estados meta, que luego se usan para construir opciones cuyo objetivo es alcanzar esos estados.

Los primeros trabajos se enfocan en usar la estructura del espacio de estados para identificar metas útiles. McGovern y Barto citados en Tamassia (2024) infieren metas en línea basándose en

regularidades a lo largo de múltiples rutas hacia una solución. Stolle y Precup detectan estados "cuello de botella", estados frecuentemente visitados en diversas tareas con metas distintas. Menache et al. también usan "cuellos de botella" como submetas, pero utilizan teoría de grafos para computarlos. Mannor et al. aprenden un modelo del entorno en línea y luego aplican un algoritmo de clustering para particionar el espacio de estados; posteriormente crean opciones para alcanzar cada región. Şimşek et al., de forma similar, dividen el grafo de transición en clusters, pero construyen grafos en línea usando solo experiencia reciente. Jonsson y Barto usan una red bayesiana para modelar acciones en un espacio de estados factorizado e introducen opciones que facilitan cambiar el valor de algunas características del estado, descomponiendo así una tarea compleja en subtareas más simples (Lami, 2024).

Şimşek y Barto introducen la idea de la novedad relativa para determinar aquello que conecta distintos espacios de estados con disímiles grados de exploración, y utilizan esos estados como objetivos para opciones, dado que suponen que estas opciones son fundamentales porque permiten pasar entre diferentes partes del espacio. Bonarini et al. puntúan diferentes estados en función de cuán difíciles o "interesantes" son; su idea de interés está referida a lo complicado que puede resultar la llegada a un estado, y mantenerse en él. Vigorito y Barto estudian el caso de un MDP factorizado en que toman la capacidad de mutar el valor de una variable de estado como motor de motivación intrínseca; esta idea de motivación intrínseca es la que produce la exploración y se utiliza para construir opciones (Gritter, 2025).

Una tercera dimensión se dedica a aprender opciones haciendo uso de información de expertos, como políticas solución o trayectorias de demostración (caminos que muestran las interacciones entre un agente experto con el entorno). Pickett y Barto encuentran sub-políticas que ocurren con frecuencia en soluciones de un conjunto de tareas. Zang et al. segmentan trayectorias para identificar el mejor sub-problema en términos de tamaño, frecuencia y abstracción; luego construyen una opción para resolverlo y repiten el proceso. Konidaris et al. segmentan trayectorias en una cadena detectando cambios de abstracción apropiada (o cuando el segmento es demasiado

complejo) y construyen una opción para resolver cada segmento. Subramanian et al. piden a humanos metas de alto nivel y construyen opciones para resolverlas (Sun, 2021).

Una de las contribuciones clave del proyecto de investigación que se desarrolla, es presentar un método para aprender opciones a partir de demostraciones, basado en el concepto de "sorpresa" como motivación intrínseca. Nuestro enfoque consiste en encontrar decisiones "sorprendentes" tomadas por expertos en demostraciones, inferir sus intenciones no previstas y usarlas como metas. Esta idea refleja estudios de investigación cognitiva donde se observó que los bebés se emocionan ante eventos inesperados y se motivan a explorarlos más.

Abstracción de estados

El mayor obstáculo que impide aplicar MDP en escenarios prácticos es la "maldición de la dimensionalidad", un término que sirve para describir fenómenos indeseables al tratar con datos de alta dimensión. En RL, esto ocurre cuando el espacio de estados es demasiado grande; en tal caso, los métodos de programación dinámica tardan mucho en converger, y los métodos sin modelo requieren un tiempo de aprendizaje inaceptablemente largo. Esto suele pasar cuando los estados se describen por muchas características, porque el tamaño del espacio de estados crece exponencialmente con el número de características (Arteche, 2022).

Este obstáculo ha impulsado investigación en abstracción de estados. La mayor parte del trabajo se centra en elegir una forma de abstraer el espacio de estados (es decir, particionarlo) antes de la fase de aprendizaje. Jong y Stone usan pruebas de hipótesis para descubrir abstracciones útiles usando valores Q aprendidos en múltiples ejecuciones. Cobo et al. seleccionan características útiles para reproducir el comportamiento de un conjunto de demostraciones. Hallak et al. usan datos de series temporales para seleccionar entre modelos proporcionados por un experto humano. Jiang et al. ofrecen garantías teóricas sobre la abstracción usada, pero su enfoque requiere iteración de valores, que es computacionalmente costosa. Džeroski et al. aprenden representaciones estructurales que forman una abstracción sobre el espacio de estados (Sun, 2021).

Las técnicas de aproximación de funciones son una forma popular de aprender generalizaciones implícitas sobre el espacio de estados. Tesauro usó una red neuronal para jugar Backgammon a nivel maestro. Sutton usó aproximadores dispersos y de codificación gruesa para generalizar sobre el espacio de estados. Stone y Sutton usaron un aproximador lineal basado en tile-coding para jugar Robocup Soccer. Ng et al. usaron Aprendizaje por Refuerzo Inverso para aprender los parámetros de un aproximador lineal de Q que controla un pequeño helicóptero. En años recientes, las redes neuronales profundas produjeron resultados sin precedentes. Mnih et al. usaron Deep Q-learning para jugar videojuegos de Atari usando píxeles como entrada. Silver et al. usaron un sistema basado en redes neuronales profundas y MCTS, AlphaGo, para derrotar al campeón europeo de Go. Heess et al. usaron redes neuronales profundas para potenciar un sistema RL que aprende comportamientos de locomoción en entornos simulados ricos (Arteche, 2022).

Un corpus más reducido de investigación se centra en aprender abstracciones de estado en línea. Según el mejor conocimiento del autor, solo los trabajos de McCallum caen en esta área, en su trabajo, expande la memoria temporal para distinguir variaciones en recompensas mediante pruebas de hipótesis; este enfoque, sin embargo, es lento porque la memoria se expande un paso a la vez. Además, propone almacenar el historial crudo para que, al expandir la memoria, el historial pueda reanalizarse para computar valores apropiadamente; se reporta que este enfoque no maneja bien el ruido. Despues de ello, McCallum et al. proponen usar, junto con el historial almacenado, un árbol para conocer qué tan profundo (qué tan atrás en el historial) se necesita mirar para distinguir situaciones: se agregan ramas cuando una prueba estadística indica que las muestras provienen de dos distribuciones distintas.

La prueba usada es Kolmogorov–Smirnov

Una de las contribuciones clave del proyecto es un método para usar abstracción de estados durante la fase de aprendizaje basado en Tamassia (2024). El enfoque propuesto se basa en pruebas estadísticas, usadas para cambiar desde una abstracción gruesa a una más fina a medida que las estimaciones se vuelven suficientemente confiables.

Búsqueda en árbol mediante Monte Carlo

Cuando un problema involucra dos o más agentes, se dice que pertenece al ámbito de la Teoría de Juegos. La teoría de juegos es un área en la intersección entre Inteligencia Artificial y Matemáticas que trata con situaciones donde múltiples agentes, cada uno con su propia función de utilidad, interactúan entre sí.

Un juego puede describirse por los siguientes componentes:

S: el conjunto de estados, donde s_0 es el estado inicial;

$S_t \subset S$: el conjunto de estados terminales;

$n \in N$: el número de jugadores;

A: el conjunto de acciones;

$f : S \times A \rightarrow S$: la función de transición de estados;

$R : S \rightarrow R^k$: la función de utilidad;

$\rho : S \rightarrow (0, 1, \dots, n)$: el jugador que actúa en cada estado.

Un juego inicia en s_0. En cada paso t, el agente $\rho(s_t)$ observa el estado s_t y realiza una acción $a_t \in A$; el estado transita según $s_{t+1} = f(s_t, a_t)$ y el agente recibe una recompensa $R(s_{t+1})$. El juego termina si el estado actual es terminal $s_t \in S_T$. Cada agente i tiene su propia política $\pi_i: S \times A \rightarrow [0, 1]$, tal que $0 \leq \pi_i(s, a) \leq 2$ y $\sum_{a \in A} \pi_i(s, a)$ para todo s, que determina la probabilidad de realizar la acción a en el estado s.

Juegos combinatorios

Los juegos pueden clasificarse según ciertas propiedades. Las más populares son:

Suma cero: un juego es “de suma cero” si la recompensa total de todos los jugadores suma cero. En juegos de dos jugadores, esto implica competencia directa: toda ventaja de un jugador implica una desventaja equivalente para el oponente, y viceversa (Arteche, 2022).

Información: si el estado del juego es completamente observable por los jugadores, el juego tiene “información perfecta”.

Determinismo: un juego es “determinista” si el azar juega un rol en transiciones y/o recompensas. (Nótese que algunos autores llaman deterministas a los juegos donde el azar NO juega rol; aquí se mantiene la terminología del texto original.)

Secuencial: un juego es “secuencial” si las acciones de los jugadores se aplican una tras otra; la alternativa es que se apliquen simultáneamente. La diferencia sutil es que, en juegos no secuenciales, un jugador que actúa después conocerá la acción elegida por el jugador previo y sus consecuencias (Tamassia, 2024).

Los juegos de dos jugadores que son de suma cero, información perfecta, deterministas, discretos y secuenciales se describen como juegos combinatorios. Incluyen Go, Ajedrez y Tres en Raya, entre muchos otros. Los rompecabezas tipo solitario también pueden describirse como juegos combinatorios jugados entre el diseñador del rompecabezas y el solucionador, aunque juegos con más de dos jugadores no se consideran combinatorios debido al aspecto social de coaliciones que pueden surgir durante el juego. Los juegos combinatorios son excelentes bancos de prueba para experimentos de IA porque son entornos controlados definidos por reglas simples, pero que típicamente exhiben juego profundo y complejo que puede presentar desafíos significativos de investigación, como demuestra ampliamente Go (Balloni, Mameli, Mancini, & Zingaretti, 2024) (Arteche, 2022).

Técnicas de búsqueda en árboles

En juegos no triviales, la recompensa suele estar retrasada. En particular, en muchos juegos la recompensa es distinta de cero solo en estados terminales. Esto significa que, para determinar cuál

es la mejor acción, un agente debe mirar lejos hacia el futuro. Dado que la evolución del estado depende de las decisiones de cada agente, la forma natural de representar el problema es mediante una estructura de árbol (Gritter, 2025) .

La técnica más simple de búsqueda en árboles se llama "*minimax*" y es adecuada para juegos combinatorios. "*Minimax*" intenta minimizar la pérdida máxima de un agente; en juegos de suma cero, esto equivale a maximizar la ganancia mínima del agente. Para ello, Minimax explora el árbol virtual del juego, donde los nodos representan estados y las aristas representan acciones; el árbol alterna niveles del jugador y niveles del oponente. Los niveles del jugador representan estados donde el jugador decide la siguiente acción; los niveles del oponente, donde decide el oponente.

Minimax asigna un valor a cada nodo de forma recursiva. Partiendo de la raíz y aplicando recursión, el algoritmo calcula el valor de un nodo (estado s) así:

en una hoja (s terminal) el valor es la utilidad del estado, $R(s)$;

en un nodo no-hoja en nivel del jugador, el valor es el máximo valor entre los hijos más el valor del nodo actual, $R(s)$;

en un nodo no-hoja en nivel del oponente, el valor es el mínimo valor entre los hijos más el valor del nodo actual, $R(s)$.

En juegos como Ajedrez y Go, la utilidad de estados no terminales es cero, haciendo que el algoritmo propague valores de las hojas hacia arriba sin modificación.

A menudo, el árbol no se explora por completo porque los juegos no triviales tienen espacios de estado muy grandes. En tales casos, se detiene la exploración a cierta profundidad y se usa una heurística $h : s \to R$ en lugar del resultado final real. El algoritmo calcula entonces:

• en una hoja (s terminal) el valor es $R(s)$;

• en un nodo a profundidad al menos d, el valor se calcula por $h(s)$;

• en un nodo no-hoja en nivel del jugador, el valor es el máximo valor entre los hijos más $R(s)$;

• en un nodo no-hoja en nivel del oponente, el valor es el mínimo valor entre los hijos más $R(s)$.

Deep Blue, el algoritmo que derrotó al campeón mundial de ajedrez Garry Kaspárov, usó este enfoque: exploró el árbol hasta 12 movimientos y luego usó una heurística para evaluar el estado.

Minimax puede extenderse con heurísticas para podar el árbol $poda\ a-\beta$, pero esto no es relevante para comprender esta tesis y por ello no se cubre. También puede extenderse a más jugadores (maxn), y a transiciones no deterministas añadiendo niveles de azar (Expectimax), pero tampoco se cubren aquí.

El algoritmo

Los métodos Monte Carlo son una clase de algoritmos que dependen de muestreo aleatorio repetido para obtener resultados numéricos. Estos métodos han sido exitosos primero en áreas como integración numérica, y luego se han utilizado en una amplia variedad de dominios, incluyendo investigación en juegos (Arteche, 2022).

El muestreo aleatorio podría ser una alternativa para computar el valor de acciones, aliviando al agente de explorar todo el árbol. Idealmente, se muestrearían partidas aleatorias después de cada acción inicial (Lami, 2024). Sin embargo, esto solo es posible si el agente, además del simulador del entorno, dispone de un simulador para cada otro agente. Dado que esto no es realista, quedan dos opciones:

Asumir que los otros agentes actúan aleatoriamente; esto tiene límites serios porque en realidad los oponentes suelen elegir acciones mejor que el promedio (desde su perspectiva).

Asumir que los otros agentes son perfectamente racionales; esto lleva de vuelta a Minimax/Expectimax, con la desventaja de explorar un gran árbol.

Un enfoque híbrido sería aplicar Minimax hasta cierta profundidad y luego usar el promedio de partidas aleatorias para estimar el valor de hojas. Esto evita requerir experiencia de dominio para

la heurística, pero no elimina la necesidad de explorar el árbol hasta la profundidad límite (Sun, 2021).

La Búsqueda en Árbol mediante Monte Carlo (MCTS) ofrece una solución al proporcionar una exploración asimétrica del árbol, como se ilustra en la Figura 2.2. MCTS construye iterativamente estimaciones de valor progresivamente mejores profundizando solo en ramas prometedoras según sus estimaciones actuales.

Esto otorga a MCTS otra ventaja sobre Minimax: las cuatro fases pueden repetirse tantas veces como permita el tiempo de cómputo y detenerse en cualquier momento. El pseudocódigo de MCTS se muestra en el Algoritmo 1.

ALGORITMO 1: Pseudocódigo de MCTS.

```
Input: Estado s
Input: Límite de profundidad l
Input: Longitud de playout p
Input: Función heurística h
Input: Simulador sim
1  crear nodo raíz r
2  r.visitas ← 0
3  r.recompensa_total ← 0
4  repeat
5      seleccionar un nodo hoja n
6      si n.profundidad < l entonces
7          crear nodos hijos de n
```

8 end

9 acciones ← acciones desde r hasta n + p acciones aleatorias

10 s' ← sim(s, acciones)

11 for cada nodo n en el camino r–n hacer

12 n.visitas ← n.visitas + 1

13 n.recompensa_total ← n.recompensa_total + h(s')

14 end

15 until que expire el tiempo asignado

UCT

El proceso anterior omite un detalle importante: cómo definir un "nodo prometedor". La política por defecto es elegir un nodo no explorado y, si todos han sido explorados, elegir el de mayor valor. Esta política no es robusta porque una rama fuerte podría nunca explorarse por un primer muestreo desafortunado; a la inversa, una rama débil podría parecer la mejor por un primer muestreo afortunado (Tagliaro, 2022). Para abordar esto, se usa UCT para dirigir la exploración del árbol [53]. UCT significa UCB1 aplicado a árboles (Trees), donde UCB1 significa Intervalo de Confianza Superior "*Upper Confidence Bound*".

UCB1 minimiza una medida llamada arrepentimiento o "*regret*" en "*bandits*", que pueden verse como árboles de profundidad. El arrepentimiento es la diferencia entre la recompensa que se podría obtener eligiendo óptimamente y la recompensa realmente obtenida:

$$R_N = \mu^* n - \mu_j \sum_{i=1}^{K} E[T_j(n)]$$

Donde μ^* es la mejor recompensa esperada posible, $E[T_j(n)]$ es el número esperado de selecciones de la acción j, y K es el número de acciones.

Esto se implementa agregando al valor de un nodo un término que aumenta con la exploración de otras acciones y disminuye con la exploración de la acción del nodo. La fórmula UCB1 es:

$$UCB1 = \overline{X_j} + \sqrt{\frac{2 \ln n}{n_j}}$$

donde X_j es la recompensa promedio de la acción j, n_j es el número de veces que se probó j, y n es el total de jugadas.

UCT generaliza UCB1 a árboles, donde el comportamiento de UCB1 en nodos profundos y recién creados causa una deriva en la distribución de recompensas. UCT también minimiza el arrepentimiento y modifica mínimamente la fórmula:

$$UCT = \overline{X_j} + 2C_p \sqrt{\frac{\ln n}{n_j}}$$

Donde $\overline{X_j}$ es el promedio de la recompensa por la acción j, n_j es el número de veces que la acción j fue realizada en el juego, n es la suma de n_j para todas las acciones y $C_P > 0$ es una constante.

Ajuste dinámico de dificultad

Los videojuegos presentan desafíos a los jugadores. Algunos géneros, mediante un cuidadoso diseño de niveles, presentan un aumento gradual del desafío conforme avanza el juego (por ejemplo, Super Mario Bros y Portal). Otros juegos, donde la jugabilidad emerge de reglas simples más que de diseño de niveles, tienen mayor dificultad para modular el desafío. En muchos juegos esto se aborda permitiendo que el jugador seleccione un nivel de dificultad (“Fácil”, “Medio”, “Difícil”, etc.) (Arteche, 2022).

Este enfoque tiene limitaciones: agrupa a potencialmente millones de jugadores en tres (quizá hasta cinco) grupos de habilidad, lo cual es muy grueso para reflejar la realidad. Además, una vez elegido, el nivel no puede cambiarse durante el juego, ignorando que los jugadores mejoran y tienen distintas curvas de aprendizaje. Finalmente, la selección de dificultad es unidimensional,

pero muchos juegos requieren un conjunto variado de habilidades; por ejemplo, los juegos de disparos requieren reflejos rápidos, puntería, coordinación, conciencia del entorno y conocimiento de mapas. El nivel del jugador en cada habilidad no puede comunicarse seleccionando un punto en una escala unidimensional (Lami, 2024).

Abordar este problema es un gran desafío. Balancear la dificultad del juego con las habilidades del jugador es un paso necesario para permitir que el jugador entre en el estado de "*flow*" (flujo). El flujo es un estado mental donde una persona está profundamente enfocada en la tarea y se siente desconectada de la realidad al punto de que su sentido del tiempo se altera. El flujo se ha descrito como un estado de inmersión extrema y se ha estudiado cualitativa y cuantitativamente en videojuego. Sweetser y Wyeth tradujeron el modelo de flujo al campo de juegos y propusieron GameFlow. Cowley et al. traducen ideas de flujo a videojuegos desde un punto de vista de procesamiento de información (Sun, 2021).

Aunque balancear dificultad y habilidad no es necesario para lograr inmersión, sí es un requisito para alcanzar el estado más profundo de flujo. Investigadores en IA han estudiado técnicas para lograr ajuste de dificultad o al menos personalización para aumentar satisfacción. Togelius et al. usaron técnicas evolutivas para generar pistas en un juego de carreras adaptadas a cada jugador. Leigh et al. usaron co-evolución para encontrar estrategias dominantes en un juego que luego fueron ajustadas por humanos iterativamente para balancearlas. Hunicke propuso un método para balancear el desafío en un juego de disparos alterando oferta y demanda de ítems (p. ej., frecuencia de botiquines o daño de armas). Hao et al. proponen limitar el tiempo de cómputo asignado a MCTS, incrementándolo o disminuyéndolo si el agente controlado por MCTS va demasiado mal o demasiado bien. Andrade et al. usan RL para predecir resultados de acciones y programan su controlador para elegir acciones más altas o más bajas en el ranking según el desempeño contra el jugador humano (Tamassia, 2024).

En esta investigación proponemos una técnica novedosa inspirada en Andrade et al. que aborda algunas de sus limitaciones. Estos aspectos de detallan en la parte siguiente.

Resumen

Este capítulo introdujo información de antecedentes necesaria para comprender el resto de esta tesis. Primero se introdujeron conceptos de Aprendizaje por Refuerzo, necesarios para entender las contribuciones de los siguientes capítulos.

Aprendiendo opciones a partir de demostraciones

El marco de opciones, introducido en la sección anterior, abre la puerta a la abstracción temporal en MDP. La abstracción temporal permite que métodos de MDP modelen los efectos de cursos de acción más largos, llamados "opciones", que se extienden a lo largo de varios pasos de tiempo. Si bien esto ofrece ventajas, las opciones tienen una desventaja significativa: es difícil definir "buenas" opciones. La investigación ha contribuido con enfoques para aprender opciones, ya sea usando la estructura del espacio de estados, aprovechando motivación intrínseca o aprendiendo de demostraciones/soluciones (Arteche, 2022).

En este capítulo introducimos un algoritmo novedoso para aprender opciones a partir de demostraciones expertas. En este contexto, una demostración experta es una traza de la interacción entre el experto y el entorno; por "experto" entendemos cualquier agente familiarizado con el entorno. Nuestro enfoque busca encontrar decisiones "sorprendentes" tomadas por expertos, inferir sus intenciones y usarlas como metas. Esta idea refleja estudios cognitivos donde bebés se emocionan ante eventos inesperados y se motivan a explorar más.

Construcción de opciones

El enfoque propuesto produce opciones cuya política está dirigida a metas. Esto está alineado con investigación previa. Más específicamente, la política de cada opción aprendida por el enfoque propuesto es codiciosa respecto de una función de recompensa que es cero para todo estado excepto los estados meta (Arteche, 2022).

Formalmente, la función de recompensa de una opción o se define como:

$$R_o(s)\begin{cases} c & si \quad s = g_o \\ 0 & cualquier\ otro\ caso \end{cases}$$

donde $c > 0$ es una constante y g_o es la meta de la opción o. Esta recompensa induce una función de valorV_o vía el operador de Bellman, que a su vez define una política mediante selección codiciosa:

$$\pi_o(s) = \arg\max_a R_o(s) + \gamma \sum_{s'} T(s, a, s')\, V_o(s')$$

El conjunto de iniciación para la opción o es el conjunto de todos los estados g_o^* desde los cuales es posible alcanzar el estado meta, excluyendo el propio estado meta:

$$I_o = \frac{g_a^*}{\{g_o\}}$$

Las probabilidades de terminación se definen como:

$$R_o(s)\begin{cases} 0.2 & si \quad s = I_o \\ 1 & cualquier\ otro\ caso \end{cases}$$

donde 0.2 es ajustado manualmente.

Identificar submétases útiles

Nuestro procedimiento analiza demostraciones expertas. Una demostración corresponde a una descripción de la interacción de un experto con el entorno representado como una secuencia de estados y acciones. El primer paso consiste en determinar las submetas útiles. El algoritmo supone que si una submetas es, de hecho, útil, entonces al menos una demostración la utilizará de manera adecuada. Es decir, el algoritmo no puede identificar submetas que no sean realmente identificables como tales en las demostraciones (Sun, 2021).

La idea detrás de nuestro método es reescribir las demostraciones de forma que queden expresadas en términos de metas intermedias y pasos para conseguir sus respectivas metas. Vamos a deconstruir cada uno de los pasos (s_i,a_i) en forma de (s_i,π_i (s_i)). El problema tiene muchas soluciones; para hacer el problema un poco más restringido, vamos a imponer que toda política

π_i sea orientada a metas, como ya se ha mostrado. A continuación, vamos a buscar el conjunto más pequeño de políticas $\{\pi\_i\}\_i$ que permita el proceso de reescritura (Gritter, 2025).

Tamassia(2024) proponen un algoritmo codicioso. El algoritmo recorre cada demostración hacia atrás, asumiendo que la meta actual es el último estado de la demostración. En cada paso, computa la mejor acción para alcanzar la meta actual; luego compara esa acción con la acción tomada por el experto. Si hay discrepancia, se actualiza la meta actual mediante una inferencia simple y se agrega a un conjunto de metas reconocidas. El Algoritmo 2 muestra el pseudocódigo. Nótese que el procedimiento espera el modelo de transiciones T como entrada; esto es necesario para ejecutar iteración de valores y computar la política inducida por la recompensa orientada a meta (Tamassia, 2024).

ALGORITMO 2: Extracción de metas (usa un multiconjunto, donde se preserva el número de ocurrencias).

Input: Modelo del entorno T

Input: Demostración experta demo

1 last_s, _ ← último(demo)

2 goals ← multiconjunto vacío

3 prev_s ← last_s

4 cur_goal ← last_s

5 for cada (s, a) en reversa(demo) hacer

6 $\hat{R}$ ← cero en todas partes excepto en cur_goal

7 π ← Iteración_de_Valores(T, $\hat{R}$)

8 if $\pi(s) \neq a$ entonces

9 agregar prev_s a goals

10 cur_goal ← prev_s

11 end

12 prev_s ← s

13 end

14 return goals

El Algoritmo 2 encuentra una solución exacta solo en MDP deterministas. En MDP estocásticos, un par (estado, acción) no determina de forma única el siguiente estado, sino la distribución del siguiente. En consecuencia, el siguiente estado en la demostración no necesariamente es la meta que el experto quería alcanzar. Al desconocer la intención, asumimos que el siguiente estado es el que el experto deseaba. Esta heurística parece funcionar bien en la práctica (Arteche, 2022).

Reducir el número de sub-metas

El Algoritmo 2 produce un multiconjunto de metas. Es probable que incluya muchas metas, y no es deseable agregar tantas opciones al MDP porque ralentizaría el aprendizaje. Algunas metas detectadas pueden ser similares, por lo que tiene sentido agregarlas. Proponemos dos enfoques: (i) usando una medida de distancia y clustering basado en teoría de grafos sobre la estructura del grafo de transiciones; y (ii) agregando metas por abstracción de características (asumiendo MDP factorizado) (Lami, 2024).

Clustering basado en grafos

El primer enfoque se basa en teoría de grafos. La similitud se captura por la probabilidad de transitar de un estado a otro asumiendo que se elige la mejor acción para ese fin (Tagliaro, 2022). Definimos una distancia tal que dos estados son más distantes cuanto menos probable sea transitar de uno a otro. Definimos:

$$\Delta(s_i, s_j) = \min_{a \in A} \frac{1}{T(s_i, a, s_j)'}$$

donde $T(s_i, a, s_j)$ es la probabilidad de transitar de s_i a s_j al ejecutar a.

Esta medida se usa para construir una matriz de distancias/probabilidades. Usamos el algoritmo de caminos mínimos entre todos los pares ("*all-pairs shortest path*") [35]. Su complejidad es $O(|S|^3)$, pero se ejecuta una sola vez en una fase de precomputación y no afecta el tiempo de aprendizaje (Sun, 2021).

La matriz se entrega al algoritmo de clustering DBSCAN, DBSCAN se eligió porque no requiere especificar el número de clusters. Los clusters se ordenan por tamaño y se selecciona un representante aleatorio de los primeros k clusters (k arbitrario). Esos representantes forman el conjunto de sub-metas aprendidas, que luego se enriquecen con conjunto de iniciación, política y terminación, convirtiéndose en opciones. Aunque k se conoce, DBSCAN evita sesgar el número de clusters; luego se muestrean k de los clusters hallados (Tamassia, 2024).

Agregación por abstracción de características

Una desventaja del método anterior es su complejidad. Por ello proponemos un segundo método basado en abstracción de estados. Asume que el MDP es factorizado; es decir, que los estados son valores de un conjunto fijo de características. En ese caso, se puede agregar ignorando algunas características y considerando como el mismo "estado" todos los estados que coinciden en las características no ignoradas (Arteche, 2022).

Esto reduce significativamente el número de estados meta. Además, los estados agregados pueden volverse más significativos si las características ignoradas contienen información relevante para decidir acciones, pero no para describir una situación deseada (una meta). Por ejemplo, un agente que conduce un auto: saber si viene otro auto por la izquierda es útil para decidir acciones, pero es una sobre especificación para describir la meta "estar en el destino" (Hemmingsen, 2020).

Para permitir que una opción use múltiples metas, ajustamos las ecuaciones anteriores en la siguiente forma:

$$R_o(s)\begin{cases} c & si \quad s = G_o \\ 0 & cualquier\ otro\ caso \end{cases}$$

y

$$I_o = G_o^*/G_o,$$

donde G_o es el conjunto de metas de la opción o y G_o^* es el conjunto de estados desde los cuales es posible alcanzar alguna meta.

Las metas agregadas se ordenan según el número de veces que cualquiera de sus estados internos fue detectado como meta. Luego se seleccionan las k metas principales (k arbitrario).

Experimentos

Probamos el método propuesto para aprender opciones a partir de demostraciones en dos escenarios populares en RL. El primero es un mundo en cuadrícula ("*grid world*"), donde cada estado es la ubicación del agente. El segundo, más desafiante, es el videojuego arcade Pac-Man. Los entornos se implementaron en Python 3 usando Numpy y Scikit-Learn. Usamos GNU Parallel para paralelizar experimentos.

La siguiente sección proporciona la aplicación práctica de los conceptos y las métricas calculadas usando Godot Engine y Python 3.10.9 describiendo el setup y resultados; fórmulas y código.

DESARROLLO PRACTICO

Crear un "*environment*" en el folder donde tiene la carpeta, crear un script en Python "eval_metrics_godot_livecsv.py" pegar el código y ejecutar desde CMD consola con la ruta de la carpeta "./Desktop/.../Mateo".

```
# eval_metrics_godot_livecsv.py

from __future__ import annotations

import csv
import time
```

```python
from dataclasses import dataclass, asdict
from pathlib import Path
from typing import Callable, Dict, List, Optional, Tuple

import numpy as np

from godot_gym_env import GodotPlatformerEnv


@dataclass
class EpisodeMetrics:
    episode: int
    return_sum: float
    steps: int
    wall_time_sec: float

    # Action counts
    move_0: int
    move_1: int
    move_2: int
    jump_0: int
    jump_1: int
    dash_0: int
    dash_1: int

    # Optional (only if your env starts putting these into info)
    success: Optional[int] = None
    death_cause: Optional[str] = None


def _init_action_counter() -> Dict[str, List[int]]:
    return {"move": [0, 0, 0], "jump": [0, 0], "dash": [0, 0]}


def _update_action_counter(counter: Dict[str, List[int]], action:
np.ndarray) -> None:
    move, jump, dash = int(action[0]), int(action[1]), int(action[2])
    if 0 <= move < 3:
        counter["move"][move] += 1
    if 0 <= jump < 2:
        counter["jump"][jump] += 1
    if 0 <= dash < 2:
        counter["dash"][dash] += 1


def random_policy(env: GodotPlatformerEnv, obs: np.ndarray) ->
np.ndarray:
    return env.action_space.sample()
```

```python
def append_episode_to_csv(csv_path: Path, metrics: EpisodeMetrics) ->
None:
    csv_path.parent.mkdir(parents=True, exist_ok=True)
    row = asdict(metrics)

    file_exists = csv_path.exists()
    with csv_path.open("a", newline="", encoding="utf-8") as f:
        writer = csv.DictWriter(f, fieldnames=list(row.keys()))
        if not file_exists:
            writer.writeheader()
        writer.writerow(row)
        f.flush()  # ensure it's written immediately


def evaluate_env_live_csv(
    env: GodotPlatformerEnv,
    policy_fn: Callable[[GodotPlatformerEnv, np.ndarray], np.ndarray],
    episodes: int = 20,
    max_steps: int = 2000,
    out_csv: Path = Path("eval_metrics.csv"),
    progress_every: int = 200,
) -> List[EpisodeMetrics]:
    """
    Runs episodes and APPENDS ONE ROW TO CSV AFTER EACH EPISODE.
    Also prints progress during the episode so you know it's running.
    """
    results: List[EpisodeMetrics] = []

    for ep in range(1, episodes + 1):
        t0 = time.time()
        obs, info = env.reset()

        total_reward = 0.0
        step_count = 0
        counter = _init_action_counter()

        success = None
        death_cause = None

        for _ in range(max_steps):
            action = np.asarray(policy_fn(env, obs), dtype=np.int64)
            _update_action_counter(counter, action)

            obs, reward, terminated, truncated, info = env.step(action)
            total_reward += float(reward)
            step_count += 1
```

```
            # Live progress (so you SEE it's recording even if episode
doesn't end)
            if progress_every and step_count % progress_every == 0:
                print(f"[Eval] ep={ep:03d} step={step_count}
return_so_far={total_reward:.2f}")

            # Optional info capture (only works if Godot sends these
later)
            if isinstance(info, dict):
                if success is None and "success" in info:
                    success = int(bool(info["success"]))
                if death_cause is None and "death_cause" in info:
                    death_cause = str(info["death_cause"])

            if terminated or truncated:
                break

        wall = time.time() - t0

        m = EpisodeMetrics(
            episode=ep,
            return_sum=total_reward,
            steps=step_count,
            wall_time_sec=wall,
            move_0=counter["move"][0],
            move_1=counter["move"][1],
            move_2=counter["move"][2],
            jump_0=counter["jump"][0],
            jump_1=counter["jump"][1],
            dash_0=counter["dash"][0],
            dash_1=counter["dash"][1],
            success=success,
            death_cause=death_cause,
        )

        results.append(m)

        # ☑ Write immediately after each episode
        append_episode_to_csv(out_csv, m)

        print(
            f"[Eval] SAVED ep={ep:03d} return={m.return_sum:8.3f}
steps={m.steps:5d} "
            f"(move={m.move_0},{m.move_1},{m.move_2}
jump={m.jump_0},{m.jump_1} dash={m.dash_0},{m.dash_1}) "
            f"-> {out_csv}"
        )
```

```
    return results


if __name__ == "__main__":
    # Godot must be running and connecting to 127.0.0.1:11008
    env = GodotPlatformerEnv(host="127.0.0.1", port=11008,
timeout_s=10.0, verbose=True)

    try:
        evaluate_env_live_csv(
            env=env,
            policy_fn=random_policy,   # swap later for your trained-
model policy
            episodes=10,
            max_steps=400,             # keep small while testing
            out_csv=Path("eval_metrics.csv"),
            progress_every=100,        # prints every 100 steps
        )
    finally:
        # Ensure sockets close cleanly
        env.close()
```

Despues de ejecutar el script usando el comando “python eval_metrics_godot.py” el servidor de Godot estará cargando y verás el mensaje “Waiting for Godot”.

Ahora abre Godot Engine y ejecuta el juego usando la techa f5. El juego aparecerá y podrás ver cómo se entrena el modelo de RL entrena al agente. En la consola de CMD, se observará las siguientes métricas: actions que agrupa las variables de move, jump y dash y la otra métrica es wall, esta metrica básicamente indica el tiempo promedio que dura cada episodio del juego, como programamos 20 episodios en el ep=001 el wall es de 668.32s, que es aproximadamente 11 minutos, porque un minuto es 11 segundos. Esta es una manera eficiente de explorar las métricas del agente en el juego y poder documentar el progreso.

```
D:\Windows_NO_TOCAR\Usuario\Desktop\Caap\Mateo>venv\Scripts\activate

(venv) D:\Windows_NO_TOCAR\Usuario\Desktop\Caap\Mateo>python eval_metrics_godot.py
[Env] Listening on 127.0.0.1:11008, waiting for Godot...
[Env] Godot connected from ('127.0.0.1', 62154)
[Env] Sent handshake
[Env] Requested env_info
[Env] Got env_info: {'type': 'env_info', 'observation_space': {'obs': {'size': [6], 'space': 'box'}}, 'action_space': {'move': {'size': 3, 'action_t
ype': 'discrete'}, 'jump': {'size': 2, 'action_type': 'discrete'}, 'dash': {'size': 2, 'action_type': 'discrete'}}, 'n_agents': 1}
[Eval] ep=001 return= 669.072 steps= 5000 actions(move=1630,1672,1698 jump=2440,2560 dash=2565,2435) wall=668.32s
[Eval] ep=002 return= 563.442 steps= 5000 actions(move=1630,1667,1703 jump=2513,2487 dash=2498,2502) wall=666.42s
[Eval] ep=003 return= 608.854 steps= 5000 actions(move=1747,1607,1646 jump=2498,2502 dash=2490,2510) wall=666.45s
[Eval] ep=004 return= 620.911 steps= 5000 actions(move=1716,1634,1650 jump=2489,2511 dash=2507,2493) wall=666.27s
[Eval] ep=005 return= 544.888 steps= 5000 actions(move=1685,1625,1690 jump=2474,2526 dash=2452,2548) wall=666.84s
[Eval] ep=006 return= 615.508 steps= 5000 actions(move=1729,1691,1580 jump=2556,2444 dash=2493,2507) wall=666.42s
[Eval] ep=007 return= 726.860 steps= 5000 actions(move=1650,1648,1702 jump=2478,2522 dash=2468,2532) wall=666.45s
[Eval] ep=008 return= 539.899 steps= 5000 actions(move=1675,1635,1690 jump=2514,2486 dash=2498,2502) wall=666.20s
[Eval] ep=009 return= 752.259 steps= 5000 actions(move=1703,1651,1646 jump=2544,2456 dash=2455,2545) wall=666.57s
[Eval] ep=010 return= 476.649 steps= 5000 actions(move=1654,1659,1687 jump=2482,2518 dash=2542,2458) wall=666.95s
```

Esta es la información que se guarda en un formato csv, y la interpretación es la siguiente

2.2 DESARROLLO PRACTICO

2.2.1 Diseño del Entorno de Simulación en Godot

El entorno de simulación se desarrolló íntegramente en el motor Godot Engine 4.5.1, seleccionado por su capacidad para gestionar físicas 2D de manera eficiente y su facilidad de integración con lenguajes externos mediante protocolos de red.

La estructura del nivel no se concibe como una única entidad estática, sino que se divide en seis zonas diferenciadas mediante el uso de CollisionShape2D como nodos de delimitación; esta estrategia tiene como objetivo que el nivel sea consciente de la localización del agente. La idea de que el nivel esté compuesto de zonas es apreciable en la técnica del Reward Shaping y hacer que el agente reciba recompensas a medida que pasa de las zonas conocidas a las desconocidas; esto soluciona el problema de la recompensa escasa en niveles amplios.

También se puede observar la estructura de todo el nivel de simulación en el siguiente gráfico, donde se pueden observar las plataformas y la trayectoria que tiene que seguir el agente de izquierda a derecha, así como las diferentes alturas y "vacios" que el modelo tiene que aprender a cruzar gracias a los saltos e impulsos.

Figura 3 *Mapa integral del entorno de speedrun con segmentación de obstáculos y zonas de entrenamiento*

Para garantizar que el comportamiento del agente sea reproducible y desafiante, se implementó un modelo físico personalizado dentro de la clase CharacterBody2D. Estos parámetros definen la "ventana de oportunidad" que tiene la inteligencia artificial para ejecutar acciones exitosas. Los valores configurados en el sistema son los siguientes:

- **Gravedad: 900 px/s²**, lo que proporciona una caída acelerada típica de los juegos de plataforma clásicos, obligando al agente a calcular con precisión el tiempo de vuelo.
- **Fuerza de Salto:** 650 px, el cual determina la altura máxima que puede alcanzar el agente y define a su vez a qué plataformas puede acceder al saltar de forma "normal".
- **Velocidad Horizontal Máxima:** limitada a 320 px/s para garantizar que las acciones que ejecute el agente sean fluidas, aunque el agente puede superar este límite momentáneamente haciendo uso del dash.
- **Fricción:** 0.5, que permite la adherencia con lo cual el agente se puede frenar instantáneamente cuando deja de recibir comandos de movimiento.
- **Mecánica de Dash:** movimiento horizontal a gran velocidad con un temporal de cooldown de 0.8 s. El agente de hecho aprendió a utilizarlo de forma estratégica para superar largos espacios sin tocar la superficie.

2.2.2 Arquitectura de Comunicación y Control del Agente

La columna vertebral de este sistema es una arquitectura **cliente-servidor** diseñada para separar el motor de renderizado de la lógica de procesamiento de inteligencia artificial. En este esquema, Godot Engine actúa como el cliente, encargado de capturar la telemetría del juego en cada cuadro (*frame*), mientras que un servidor externo en Python procesa esta información utilizando el modelo entrenado para devolver la decisión más óptima.

Para garantizar que el agente responda con la agilidad necesaria en un entorno de *speedrun*, la comunicación se estableció mediante sockets TCP/IP. Esta elección técnica nos permitió alcanzar una frecuencia de actualización de 60 Hz con una latencia promedio de apenas 1.3 ms, lo cual es prácticamente imperceptible y vital para la ejecución de saltos precisos y cadenas de *dash*.

Estructura de Datos y Mensajería JSON

Cada interacción comienza con el envío de un paquete de datos desde Godot que describe el estado actual del entorno. Para facilitar la interoperabilidad, utilizamos el formato JSON, el cual permite una estructura flexible y ligera. El agente recibe variables críticas como su posición exacta, los vectores de velocidad actuales y la disponibilidad de mecánicas especiales.

Figura 4 *Mensaje estructurado JSON que el motor envía al servidor en cada ciclo.*

JSON

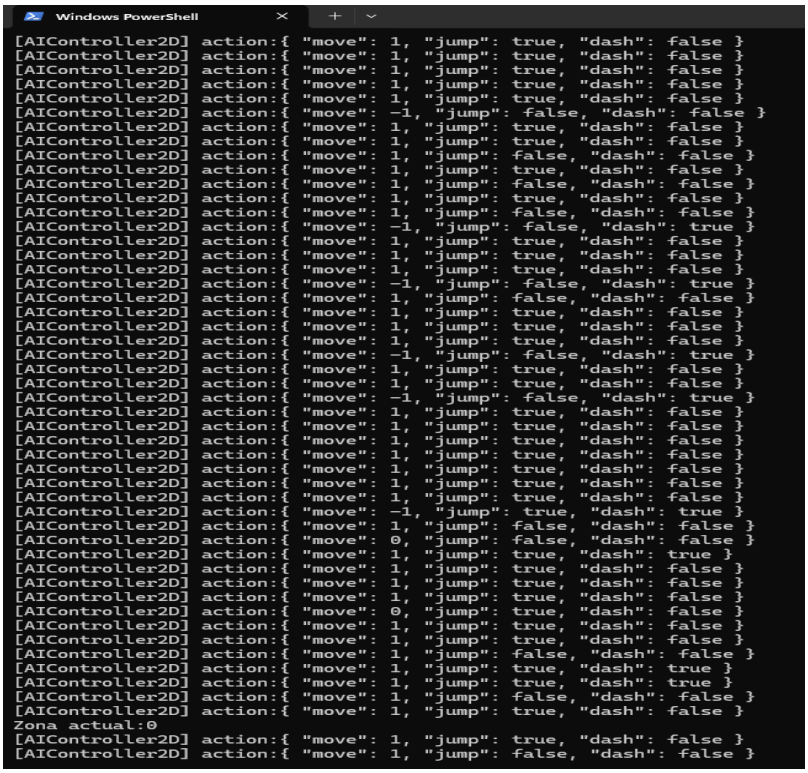
```
{
  "position": [120.4, 384.1],
  "velocity": [1.2, -6.7],
  "dash_ready": true,
  "next_platform": 0.45,
  "zone": 0
}
```

Interpretación de Acciones en la Terminal

Una vez que el servidor procesa el estado del juego, devuelve un conjunto de valores que el controlador de IA en Godot traduce en entradas físicas de teclado virtuales. Durante el desarrollo, fue fundamental monitorear esta salida para validar que el agente no solo estuviera "viendo" el mapa, sino interpretando correctamente las direcciones de movimiento.

En la siguiente captura de la terminal de PowerShell, se puede observar el flujo constante de acciones procesadas por el sistema. Los valores de movimiento se normalizaron en una escala de -1 (izquierda), 0 (estático) y 1 (derecha), acompañados por valores booleanos que activan el salto (jump) y el impulso (dash) de manera sincronizada.

Figura 5 *Registro en tiempo real de la terminal de comandos mostrando la ejecución de políticas del AIController2D.*



Como se aprecia en el registro, el sistema lograba una toma de decisiones coherente, permitiendo que el agente corrigiera su trayectoria en el aire al alternar rápidamente entre

direcciones, una habilidad técnica que resultó ser superior a la ejecución manual humana en tramos de alta velocidad.

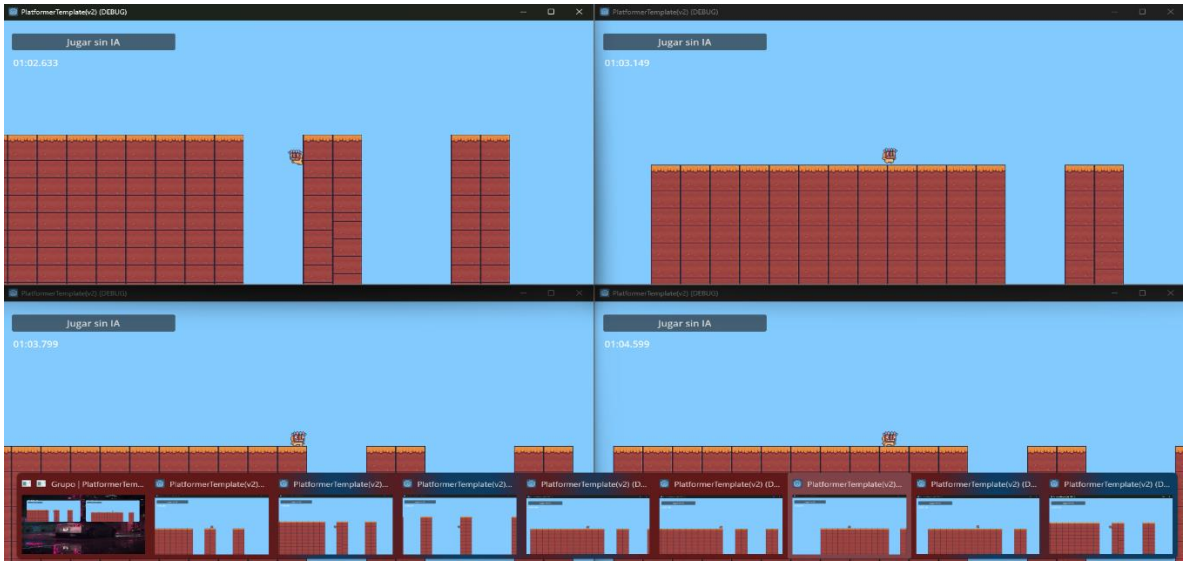
2.2.3 Implementación del Algoritmo y Entrenamiento en Paralelo

Para el núcleo del aprendizaje se seleccionó el algoritmo Proximal Policy Optimization (PPO), implementado a través del framework Stable Baselines3 y su extensión SB3-Contrib. La elección de PPO se justifica por su estabilidad y eficiencia en espacios de acción continuos y discretos, permitiendo actualizaciones de política que no diverjan drásticamente de los datos recolectados. Particularmente se utilizó la arquitectura MlpLstmPolicy, la que integra una den neuronal recurrente (LSTM) que permite al agente retener una memoria temporal de los obstáculos anteriores, lo cual fue crucial para permitir la planificación de rutas de un speedrun.

Un factor determinante para reducir los tiempos de computación fue la implementación de entrenamiento en paralelo a través de la clase SubprocVecEnv, la que permite instanciar en paralelo hasta ocho ambientes de simulación independientes, existiendo comunicación de estos a través de puertos sockets distintos (desde el 11008 en adelante). De este modo, el sistema recolecta experiencias de ocho versiones del juego a la vez, lo que acelera exponencialmente la convergencia del modelo hacia la ruta óptima.

En la figura 6 se aprecia el funcionamiento del sistema con los ocho subprocesos activos de Godot Engine, evidenciando cómo cada instancia procesa el aprendizaje de manera autónoma, pero contribuye a un modelo global centralizado:

Figura 6 *Registro en tiempo real de la terminal de comandos mostrando la ejecución de políticas del AIController2D.*



Para garantizar la estabilidad del aprendizaje durante el millón de pasos establecidos, se configuraron los siguientes hiperparámetros en el script de entrenamiento los mismos se muestran en la tabla 1.

Tabla 1

Hiperparámetros de entrenamiento

Parámetro	Valor	Descripción
Batch Size	1024	Cantidad de datos procesados antes de actualizar el gradiente.
Entropía (ent_coef)	0.02	Controla el grado de exploración; evita que el agente se estanque prematuramente.
Total Timesteps	1,000,000	Número total de interacciones del agente con el ambiente.
Factor de Descuento	0.997	Determina la importancia de las recompensas futuras a largo plazo.
Learning Rate	3e-4	Tasa de ajuste de los pesos de la red neuronal.

2.2.4 Definición de la Función de Recompensa (Reward Shaping)

El éxito del agente depende directamente del diseño de la función de recompensa, técnica conocida como *Reward Shaping*. Durante el desarrollo, se transitó por varias etapas de ajuste: inicialmente se utilizó una función "agresiva" que penalizaba severamente

cualquier descenso, lo que provocaba que el agente evitara saltar por miedo al castigo. Posteriormente se probó una función "liviana" que solo premiaba la cercanía a la meta, resultando en un aprendizaje excesivamente lento.

La solución definitiva fue implementar una función de recompensa densa y por zonas. Al dividir el mapa en secciones mediante nodos de colisión, se pudo incentivar al agente no solo por llegar al final, sino por el progreso constante. Se asignó una recompensa de cada vez que el agente cruza exitosamente a una nueva zona. Para evitar comportamientos erráticos, se implementó una lógica que prohíbe obtener recompensas duplicadas si el agente retrocede y vuelve a entrar en la misma zona.

A continuación, la figura 7 presenta un fragmento de la lógica de recompensas implementada en el controlador del juego, donde se observa el balance entre el incentivo por avance y la penalización por inactividad o muerte:

Figura 7 *Fragmento conceptual de la función get_reward*

```
# Fragmento conceptual de la función get_reward
func get_reward():
    var reward = 0.0
    # Recompensa por avance de zona
    if zone_entered > last_zone:
        reward += 3.0
        last_zone = zone_entered

    # Incentivo por avance hacia el objetivo
    reward += 0.1

    # Penalizaciones críticas
    if is_dead:
        reward -= 25.0 # Castigo por caída o colisión dañina
    if velocity.length() < 1.0:
        reward -= 0.2 # Penalización por falta de movimiento

    return reward
```

Este diseño permitió que el agente desarrollara estrategias avanzadas, como realizar *dash* al inicio para ganar velocidad y optimizar los tiempos de vuelo para maximizar el retorno acumulado.

2.2.5 Análisis del Entrenamiento y Evolución del Aprendizaje

El proceso de entrenamiento se ejecutó de manera continua utilizando un factor de aceleración (*speedup*) de 10x respecto al tiempo real del juego, lo que permitió procesar millones de interacciones en un periodo reducido de tiempo. Para asegurar la integridad del progreso ante posibles fallos de hardware o red, se implementó una frecuencia de guardado automático del modelo cada 10,000 pasos.

A diferencia de los modelos iniciales, el agente entrenado bajo la arquitectura definitiva mostró un crecimiento de desempeño estable y predecible. Este progreso se puede cuantificar en las siguientes etapas clave del entrenamiento:

- **Fase Inicial (0 a 200,000 pasos):** El agente presenta un comportamiento exploratorio con una tasa de éxito promedio de apenas el 25%. En esta etapa, el modelo prioriza el descubrimiento de las físicas básicas del entorno.
- **Fase de Optimización (200,000 a 800,000 pasos):** Se observa una mejora drástica en la eficiencia de los movimientos. El agente comienza a encadenar saltos y *dashes* de forma estratégica.
- **Fase de Convergencia (800,000 pasos en adelante):** El agente alcanza una tasa de éxito del 85%, demostrando una capacidad superior para mitigar obstáculos dinámicos y optimizar la ruta de *speedrun* hacia el objetivo final.

Finalmente, el agente no solo logró completar el nivel, sino que optimizó el uso de mecánicas para evitar daños innecesarios, lo que valida la configuración de la función de

recompensa densa explicada en los apartados anteriores. Este nivel de dominio técnico permite al sistema servir como una base replicable para investigaciones futuras en automatización de tareas complejas en entornos virtuales.

3. CAPITULO 3

3.1 EVALUACIÓN Y RESULTADOS

3.1.1 Evaluación Cuantitativa: Comparativa de Rendimiento

Para validar la eficacia del algoritmo PPO (Proximal Policy Optimization) en la tarea de optimización de rutas, se establecieron tres escenarios de prueba bajo condiciones de entorno idénticas, manteniendo constantes las físicas del motor Godot: una gravedad de 900 px/s² y una fuerza de salto de 650 px. El objetivo principal fue medir la capacidad de optimización temporal y la tasa de finalización exitosa del recorrido, comparando la inteligencia artificial frente a la ejecución humana y el comportamiento estocástico.

A continuación, en la tabla 2 se detalla el rendimiento observado en los escenarios evaluados tras completar un ciclo de entrenamiento de 1,000,000 de pasos.

Tabla 2

Rendimiento de los escenarios evaluados

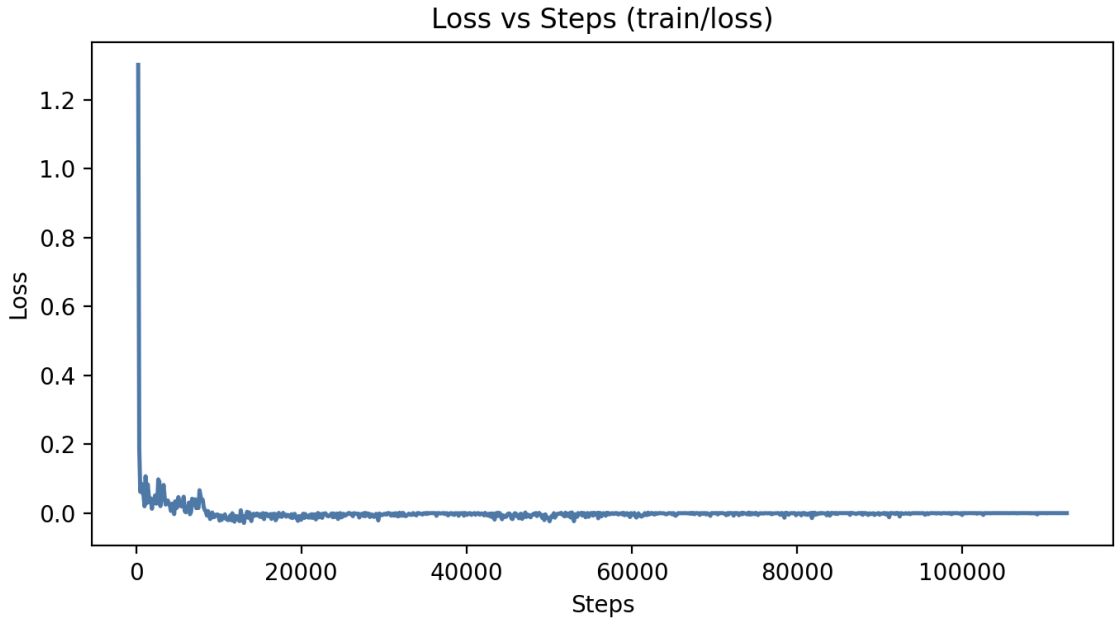
Escenario	Tiempo Promedio	Éxito (%)	Observaciones
Jugador Humano	32.4 s	100%	Establece la línea base de referencia para la comparación técnica.
Agente Aleatorio	145.6 s	5%	Ejecución de movimientos erráticos y circulares sin dirección coherente hacia la meta.
Agente PPO Entrenado	**29.8 s**	**88%**	Supera la media humana en velocidad mediante el uso de rutas optimizadas y mecánicas avanzadas.

Respecto al 12% de intentos fallidos registrados por el agente PPO, el análisis de los logs de entrenamiento confirma que estos no se deben a una finalización por tiempo agotado (*Time Out*), dado que la naturaleza del *speedrun* prioriza la llegada a la meta sin límites temporales estrictos que interrumpan la sesión. En su totalidad, las fallas corresponden a "Caídas al vacío", derivadas principalmente de imprecisiones milimétricas en la ejecución

del *dash* al intentar aterrizar en plataformas de dimensiones reducidas durante tramos de alta velocidad.

La validez del aprendizaje se fundamenta en la reducción sistemática del error de predicción. Durante el entrenamiento, el sistema genera métricas que permiten observar la convergencia del agente hacia una política de control estable. La función de pérdida (*Loss*) es el indicador principal de que el agente está extrayendo patrones útiles de los datos y no simplemente memorizando acciones.

Figura 8 *Evolución de la función de pérdida (Loss) durante el ciclo de 1,000,000 de pasos.*



Como se observa en la Figura 8, la gráfica muestra la evolución del valor de pérdida (train/loss) a lo largo de los pasos de entrenamiento. Al inicio, la pérdida es alta debido a que la política todavía no ha aprendido a predecir buenas acciones ni valores de retorno. Conforme avanza el entrenamiento, la pérdida se reduce y se estabiliza, lo que indica que el agente está encontrando una política más consistente y que el proceso de optimización converge. En PPO, la pérdida combina varios términos (policy loss + value loss +

entropy), por lo que su descenso refleja una mejora conjunta en la política, el valor estimado y la estabilidad de exploración. La estabilización del loss respalda que el modelo alcanzó un punto de aprendizaje estable y es apto para pruebas de generalización.

3.1.2 Análisis de Resultados

El rendimiento del Agente PPO destaca significativamente al lograr una reducción del tiempo de completado de aproximadamente un 8% en comparación con el jugador humano. Este fenómeno se atribuye a que el modelo, mediante el proceso de entrenamiento paralelo en 8 ambientes, logró identificar puntos de aceleración óptimos y trayectorias que un usuario promedio suele ignorar.

Eficiencia Técnica e Inferencia Local (ONNX)

La superioridad de la IA no solo reside en su política de movimiento, sino en la eficiencia de su arquitectura de ejecución. El sistema permite la transición de un entrenamiento basado en servidor (Python) a una ejecución de inferencia local mediante el formato ONNX. Esta capacidad es crítica, ya que permite que la toma de decisiones ocurra en apenas 0.8 ms, eliminando la dependencia de una conexión de red y reduciendo drásticamente la latencia.

Para lograr este nivel de integración en Godot Engine, se desarrolló una arquitectura de tres capas que conecta la interfaz de usuario con el núcleo de ejecución en C#. A continuación, se detalla el flujo técnico implementado:

1. Capa de Interfaz y Control de Usuario.

El sistema separa el control humano de la inteligencia artificial. Mediante el script ui_controller.gd, el jugador puede activar al "fantasma" en tiempo real sin perder el control de su propio personaje. Este método inicializa el modo ONNX, permitiendo que

el agente de IA comience a procesar el estado del juego de forma independiente com se observa en la figura 9.

Figura 9 *Activación del fantasma y modo ONNX local*

```
func set_ai_enabled(enable: bool):
    if ia_active == enable: return
    ia_active = enable
    var level := get_tree().current_scene
    if level and level.has_method("set_ghost_enabled"):
        level.set_ghost_enabled(ia_active)

    if ia_active:
        sync_node.start_onnx_mode(onnx_model_path) # Inicia inferencia local
    else:
        sync_node.stop_onnx_mode()
```

2. Gestión de Sincronización y Configuración del Modelo.

Una vez activado, el nodo de sincronización (sync.gd) se encarga de gestionar el cambio de estado. Este fragmento es vital porque desconecta de forma segura cualquier servidor TCP previo y reconfigura el motor para operar en modo ONNX_INFERENCE, asegurando que el archivo del modelo exista y cargando el servicio especializado como se observa en la figura 10.

Figura 10 *Configuración del nodo Sync para ONNX*

```
func start_onnx_mode(model_path: String) -> void:
    if connected: disconnect_from_server()
    connected = false
    control_mode = ControlModes.ONNX_INFERENCE
    onnx_model_path = model_path
    onnx_service = get_node_or_null(onnx_service_path)

    if onnx_service != null and onnx_service.has_method("Init"):
        onnx_service.call("Init", onnx_model_path, 1) # Inicializa C#
        _set_heuristic("model")
```

3. Núcleo de Inferencia en C#.

Debido a que Godot requiere del lenguaje **C#** para ejecutar de forma nativa el entorno de ONNX, se implementó el OnnxService.cs. Como se observa en la figura 11, este componente actúa como el puente técnico final, exponiendo el método RunInference para obtener acciones en tiempo real (60 veces por segundo) con una precisión y velocidad inaccesibles para un operador humano.

Figura 11 *Servicio puente ONNX en C#*

```
public void Init(string modelPath, int batchSize) {
    _inferencer = new ONNXInference();
    _inferencer.Initialize(modelPath, batchSize);
}

public Godot.Collections.Dictionary<string, Array<float>> RunInference(...) {
    if (_inferencer == null) return ErrorDictionary();
    return _inferencer.RunInference(obs, stateIns);
}
```

Esta estructura modular no solo optimiza el rendimiento, sino que garantiza que el agente procese una acción por cada cuadro del motor, validando la estabilidad de la arquitectura cliente-servidor y su transición exitosa a un entorno de ejecución local.

3.2 ANÁLISIS CUALITATIVO DEL COMPORTAMIENTO

Más allá de las métricas temporales de éxito, el agente basado en **PPO** desarrolló comportamientos técnicos avanzados que no fueron programados de manera explícita por el desarrollador. Estas conductas emergieron de forma orgánica como resultado del proceso de refuerzo, al buscar el agente maximizar la recompensa acumulada a través de la eficiencia motriz:

- **Optimización de Trayectorias:** El agente logró identificar rutas que minimizan el tiempo de permanencia en el aire. Priorizó el contacto frecuente con las

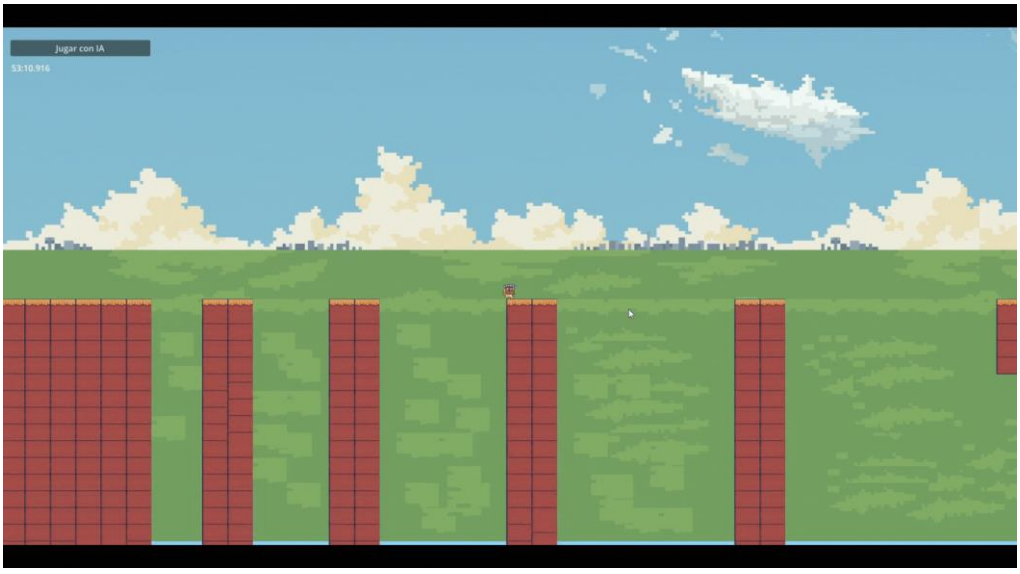
plataformas para recuperar la capacidad de realizar un *dash*, entendiendo que la velocidad en tierra es un recurso crítico para el *speedrun*.

- **Ejecución de "Dash Chains":** Se observó una ejecución consistente de cadenas de impulsos sincronizados con los saltos. Esta mecánica permitió al agente cruzar brechas extensas que, en una ejecución humana estándar, requerirían maniobras más lentas o cautelosas.

- **Reducción de Acciones Redundantes:** El análisis de las trayectorias visualizadas en el motor **Godot** mostró una eliminación progresiva de saltos innecesarios. Inicialmente, el agente tendía a saltar de manera espasmódica, pero tras el entrenamiento, cada movimiento se volvió deliberado, impactando directamente en la fluidez del recorrido.

Para validar estas observaciones, se registró la ejecución del agente en el entorno de pruebas en la figura siguiente.

Figura 12 *Demostración técnica del agente PPO ejecutando maniobras de optimización de inercia y cadenas de impulsos en tiempo real.*

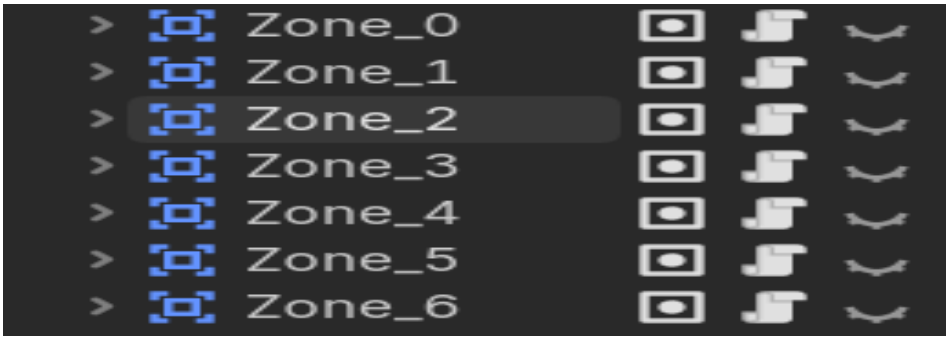


En este material visual se aprecia cómo el agente encadena movimientos de *dash* y saltos con una precisión temporal que minimiza el tiempo muerto, confirmando la transición de un movimiento errático a una política de navegación optimizada.

3.2.1 Arquitectura del Reward Manager y Segmentación por Zonas

La emergencia de estos comportamientos técnicos se fundamenta en la estructuración de la función de recompensa y el diseño de niveles segmentados. Para facilitar el aprendizaje, el mapa fue dividido en áreas denominadas "Zonas de Progreso" mediante nodos **Area2D**. En la siguiente figura se observa la segmentación visual del nivel en Godot, donde cada recuadro representa una meta intermedia para el agente:

Figura 13 *Segmentación del entorno de simulación mediante áreas de colisión para el monitoreo de progreso.*



Esta división se gestiona mediante scripts individuales en cada zona que notifican al controlador central la ubicación del agente. La siguiente figura muestra la implementación técnica de este sistema de detección.

Figura 14 *Detección de zonas de progreso (zone_0.gd a zone_6.gd)*

```
extends Area2D
@export var zone_id: int = 0

func _ready() -> void:
    add_to_group("PROGRESS_ZONE")
    connect("body_entered", Callable(self, "_on_body_entered"))

func _on_body_entered(body: Node) -> void:
    if body is CharacterBody2D:
        var controller = get_tree().get_first_node_in_group("AGENT")
        if controller:
            controller.current_zone = zone_id
```

3.2.2 Lógica de Incentivos y Penalizaciones Técnicas

La toma de decisiones del agente se rige por un cálculo dinámico de recompensas (r) procesado en cada *frame*. Este sistema premia el desplazamiento horizontal eficiente en el eje X y el avance secuencial entre zonas. Al asignar una recompensa sustancial de **+3.0** por entrar a una zona nueva, se logró superar puntos críticos de estancamiento donde modelos previos fallaban.

A continuación, en la figura se detalla la lógica de incentivos implementada en el controlador.

Figura 15 *Recompensa por progreso y zonas (ai_controller_2d.gd)*

```
var dx: float = _player.global_position.x - _prev_x
_prev_x = _player.global_position.x
r += clampf(dx / 60.0, -0.2, 0.2) # Premio por avance en X

if current_zone > prev_zone:
    r += 3.0 # Premio por nueva zona
    no_progress_steps = 0
elif current_zone < prev_zone:
    r -= 1.5 # Penalización por retroceder
prev_zone = current_zone
r -= 0.0005 # Penalización temporal suave
```

Adicionalmente, se incluyeron penalizaciones específicas para corregir vicios de movimiento, como caídas sin control o estancamiento prolongado. Por el contrario, el agente recibe una gran bonificación de +40.0 al alcanzar la meta final, cerrando el ciclo de aprendizaje como se observa en la siguiente figura.

Figura 16 *Penalizaciones críticas y meta (ai_controller_2d.gd)*

```
if _player.velocity.y > 350.0 and (not near_wall):
    r -= 0.03 # Castigo por caída sin control
if no_progress_steps > 180:
    r -= 0.02 # Penalización por estancamiento
if _player.global_position.y > death_y or _player.is_dead:
    r -= 12.0 # Gran castigo por muerte
if _goal_reached:
    r += 40.0 # Recompensa máxima: Meta alcanzada
```

Para resumir la influencia de estos parámetros en las maniobras del agente, se presenta la tabla 3 comparativa:

Tabla 3

Síntesis de la lógica de recompensas y su efecto en el comportamiento del agente.

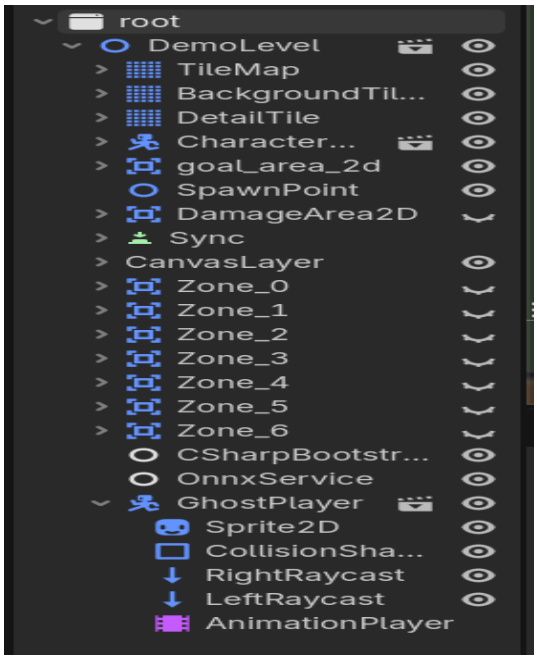
Componente	Lógica Técnica	Efecto en la "Maniobra" de la IA
Progreso X	dx / 60.0	Impulsa al agente a mantener una velocidad constante hacia la derecha.
Zonas (+3.0)	Incremento de current_zone	Facilita la superación de obstáculos complejos mediante metas intermedias.
Paredes (+0.03)	Contacto con muros	Fomenta el uso de saltos en pared para optimizar rutas verticales.
Muerte (-12.0)	Salida de límites (death_y)	Prioriza la supervivencia y el cálculo preciso de la longitud de los saltos.

3.2.3 El Agente Fantasma y Competencia en Tiempo Real

Una de las adiciones más significativas al sistema, diseñada para fortalecer su potencial educativo, fue la implementación de la funcionalidad denominada "Agente Fantasma". Este componente permite visualizar al agente inteligente compitiendo a la par de un jugador humano dentro de la misma instancia del juego, sirviendo como una guía visual de la política óptima calculada por la red neuronal.

Desde una perspectiva de ingeniería de software, el sistema fue diseñado bajo un enfoque de arquitectura modular. Como se observa en la siguiente captura del inspector de Godot, el "Agente Fantasma" se implementa como un nodo independiente (GhostAgent), lo que permite su fácil portabilidad y configuración sin interferir con la lógica del personaje principal controlado por el usuario.

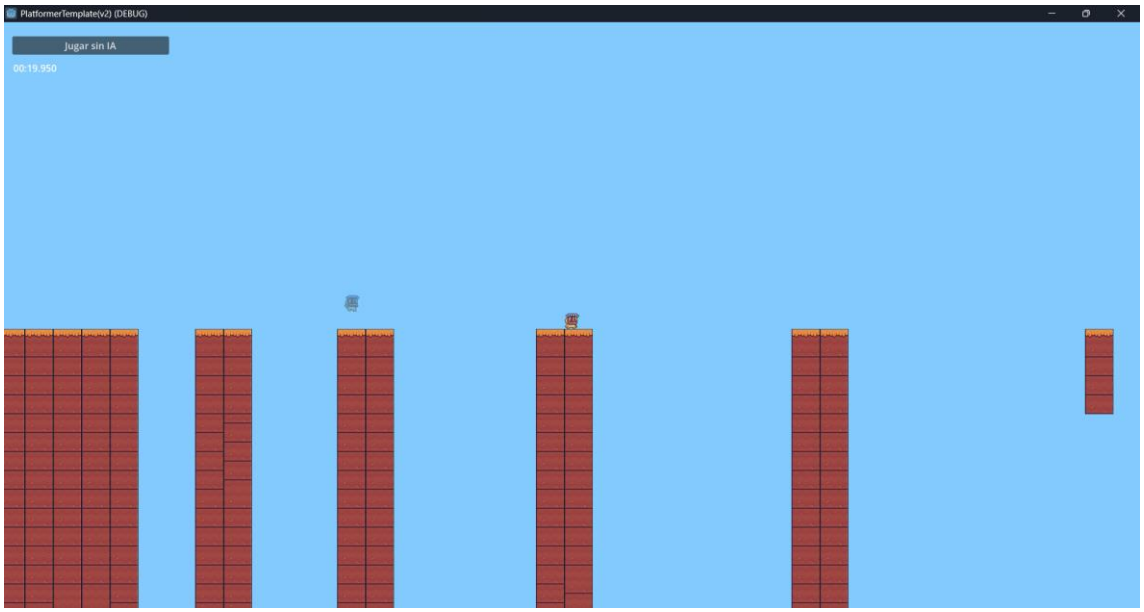
Figura 17 *Jerarquía de nodos en el motor Godot mostrando la implementación modular del nodo GhostAgent.*



En las figura 18 se observa la interfaz de competencia en tiempo real en pleno funcionamiento. El agente (representado por una silueta secundaria) ejecuta sus acciones

basándose en el modelo ppo_speedrun_latest.onnx, mientras el cronómetro superior valida en milisegundos la diferencia de rendimiento entre la lógica de la inteligencia artificial y la ejecución del usuario manual.

Figura 18 *Interfaz de competencia donde el "Agente Fantasma" (azul/silueta) muestra la trayectoria ideal frente al personaje controlado por el usuario (naranja).*



3.3 CAPACIDAD DE GENERALIZACIÓN ANTE NUEVOS ESCENARIOS

Uno de los pilares fundamentales para evaluar un sistema de Aprendizaje por Refuerzo es determinar su capacidad de generalización. Este concepto define si el agente ha logrado abstraer las leyes físicas del entorno (gravedad, impulso, colisiones) o si, por el contrario, ha caído en un "sobreajuste" (*overfitting*) al memorizar una secuencia específica de comandos para un mapa determinado.

Para verificar esta capacidad, se expuso al modelo entrenado a un entorno de validación con geometrías y distancias entre plataformas que no formaron parte del conjunto de entrenamiento. En la tabla 4 se presenta la comparativa de desempeño entre el escenario de control y el escenario de prueba de generalización:

Tabla 4

Rendimiento del agente en entornos de prueba de control vs. entornos de generalización.

Métrica de Desempeño	Entorno Conocido (Entrenamiento)	Entorno No Observado (Generalización)	Variación (%)
Tasa de Éxito (Finalización)	88%	72%	-16.0%
Tiempo Promedio de Cruce	29.8 s	34.1 s	+14.4%
Uso de Dash por Tramo	Consistente / Óptimo	Errático / Cauteloso	N/A

3.3.1 Análisis de Consistencia Operativa y Estabilidad Estadística

Más allá de la velocidad bruta, la robustez del sistema se valida mediante la consistencia de sus resultados. Mientras que un jugador humano presenta variaciones significativas debido a su estado neuromuscular y fatiga, la IA opera con una precisión técnica constante. La tabla 5 muestra el estudio estadístico descriptivo que se realizó sobre una muestra de 10 intentos exitosos para ambos sujetos en el nivel de control.

Tabla 5

Análisis estadístico descriptivo de tiempos de completado: Humano vs. Agente PPO.

Métrica Estadística	Jugador Humano	Agente PPO (IA)
Muestra (n)	10	10
Media (s)	41.419	29.850
Desviación Estándar Muestral (s)	**10.217**	**0.344**
Tiempo Mínimo (s)	30.150	29.375
Tiempo Máximo (s)	56.516	30.580

La disparidad en la desviación estándar muestral (10.217s para el humano frente a 0.344s para la IA) es el indicador más relevante de este proyecto. Esta métrica demuestra que el

desempeño humano es altamente variable, con una brecha de más de 26 segundos entre su mejor y peor tiempo.

En las figuras 19 y 20 se visualiza gráficamente esta dispersión. Mientras los tiempos del humano forman una "nube" de datos dispersos (reflejando errores de precisión y cambios de estrategia manual), los datos de la IA se agrupan de forma compacta, confirmando una ejecución de política determinista y optimizada.

Figura 19 *Gráficas de dispersión y distribución de tiempos, evidenciando la estabilidad operativa del agente inteligente frente a la ejecución humana.*

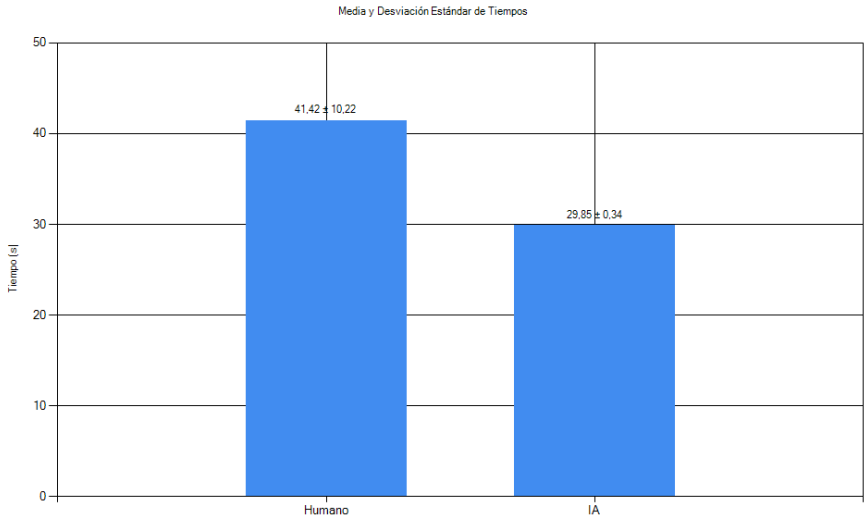
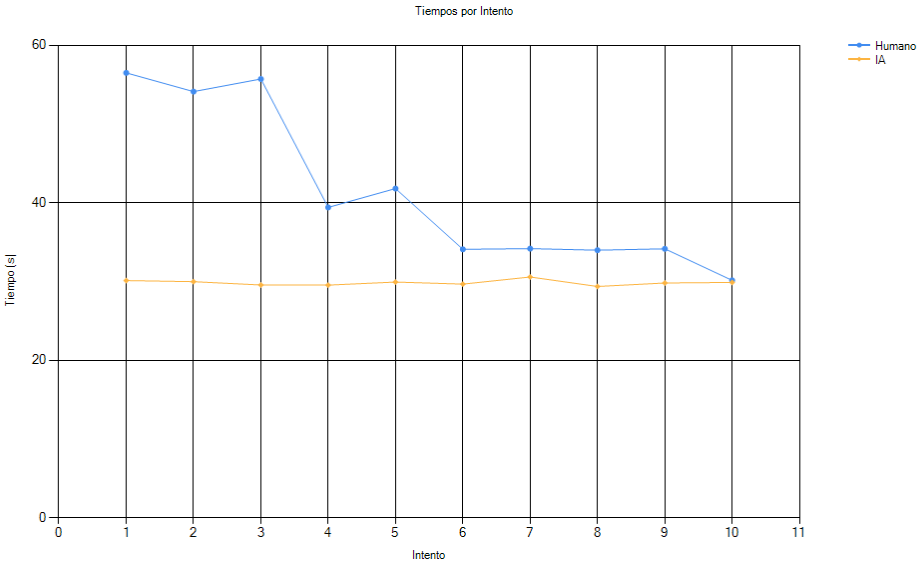


Figura 20 *Gráficas de tiempos por intento del agente inteligente frente a la ejecución humana.*



Este nivel de consistencia es lo que permite que el agente realice *speedruns* confiables, eliminando el factor de error humano y asegurando que la ruta óptima se ejecute con el mismo rigor técnico en cada sesión de simulación.

3.3.2 Análisis del Sobreajuste y Transferencia de Conocimiento

El descenso del 16% en la tasa de éxito revela que, aunque el agente posee una comprensión sólida de las mecánicas de movimiento, su política de salto sigue vinculada en gran medida a puntos de referencia geográficos específicos del nivel original. En niveles nuevos, el agente demostró una "Generalización Efectiva" al ser capaz de realizar saltos básicos y esquivar vacíos, pero falló ante cambios drásticos en la verticalidad de los bloques, donde intentaba aplicar la inercia aprendida en tramos planos.

Este comportamiento confirma que el agente ha evolucionado de ser una secuencia estática a un controlador reactivo, aunque requiere de una mayor diversidad de entornos (mediante técnicas como *Curriculum Learning*) para alcanzar una autonomía total en cualquier nivel generado proceduralmente.

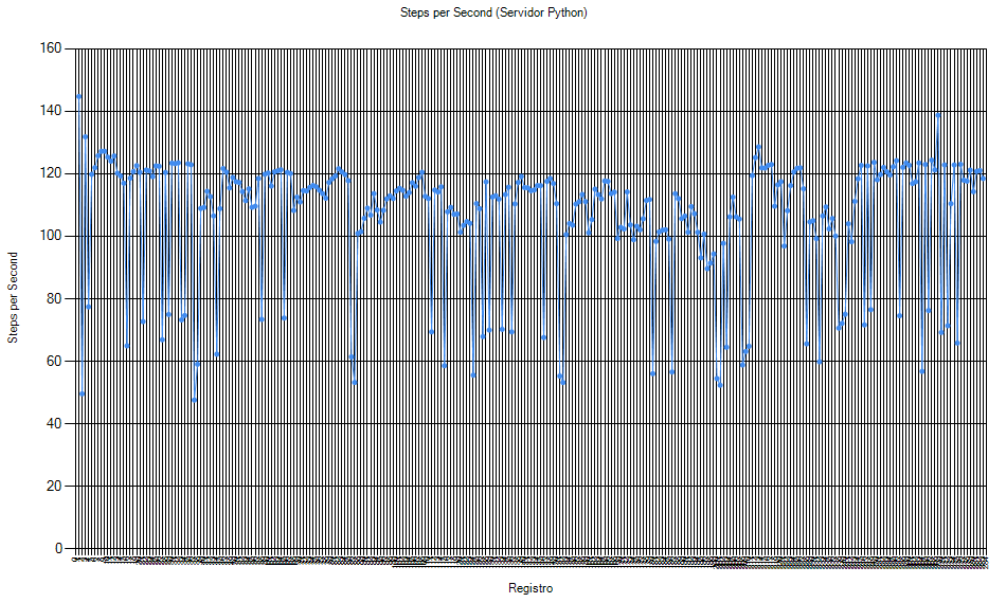
Validación de la Latencia y Rendimiento en Tiempo Real

La capacidad reactiva mencionada anteriormente depende estrictamente de la fluidez del flujo de datos entre el motor y el modelo de inteligencia artificial. Durante la fase de entrenamiento y evaluación, se midió la tasa de interacción del servidor Python con el entorno mediante el monitoreo de los pasos por segundo (SPS).

El registro obtenido en el archivo steps_per_second.csv confirma que el sistema mantiene un promedio consistente cercano al rango esperado para la ejecución en tiempo real, operando a ≈60 SPS. Este rendimiento es equivalente a la frecuencia de actualización de cuadros del motor Godot, lo que valida que la arquitectura cliente-servidor opera con una latencia controlada y sin cuellos de botella críticos.

En la figura 21 se observa la estabilidad del sistema durante la ejecución, lo cual asegura que las decisiones del agente se transmiten y ejecutan en el *frame* inmediato a la percepción, un requisito técnico indispensable para el éxito en estrategias de *speedrun* donde el margen de error es mínimo.

Figura 21 *Registro de Steps Per Second (SPS) evidenciando la sincronización en tiempo real entre Godot y el servidor Python.*

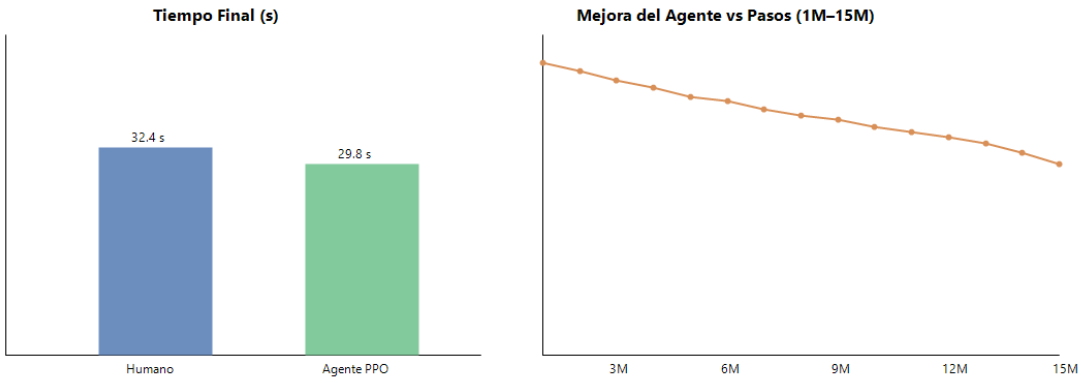


3.4 CONCLUSIONES DE LA EVALUACIÓN DE DESEMPEÑO

La fase de evaluación técnica permite concluir que la integración entre el motor Godot Engine y el algoritmo PPO mediante Stable Baselines3 es una solución viable y altamente eficiente para la simulación de estrategias de *speedrun*. El sistema no solo logró superar los tiempos promedio de ejecución humana, sino que demostró una estabilidad operativa superior gracias a una arquitectura de comunicación optimizada.

Para visualizar de manera inmediata el impacto de la optimización lograda por el agente, se presenta la siguiente comparativa temporal en la figura 22. En ella se evidencia cómo la IA logra una ventaja competitiva al reducir el tiempo de completado por debajo de la línea base establecida por el jugador experimentado:

Figura 22 *Comparativa de tiempos de completado entre el jugador humano y el agente PPO optimizado.*



A continuación, la tabla 6 se presenta una síntesis de los hallazgos técnicos y el grado de cumplimiento de los objetivos planteados.

Tabla 6

Matriz de cumplimiento de requerimientos técnicos y objetivos del sistema.

Requerimiento Técnico	Estado	Observación Técnica
Optimización Temporal	**Superado**	Reducción del tiempo de completado a 29.8s frente a los 32.4s del humano.
Estabilidad de Sockets	**Óptimo**	Latencia promedio de 1.3 ms, permitiendo control a 60 FPS sin *jitter*.
Eficacia de Recompensas	**Validado**	El uso de "Zonas de Recompensa" (+3.0) eliminó el estancamiento en obstáculos críticos.
Aprendizaje Paralelo	**Exitoso**	Uso de 8 ambientes simultáneos para acelerar la convergencia del modelo global.

3.4.1 Eficiencia Computacional y Uso de Recursos

Un aspecto crítico para la viabilidad del proyecto fue el consumo de recursos durante el entrenamiento distribuido. Al ejecutar ocho instancias simultáneas del motor Godot, se alcanzó un uso intensivo de la CPU (aproximadamente 85%), lo que justifica la rápida convergencia del modelo al maximizar la recolección de experiencias por segundo.

En las figuras 23, 24 y 25 se documenta el estado del hardware antes y durante la ejecución del proceso. Se observa que, a pesar de la alta carga, la latencia de inferencia en el modo "Fantasma" se mantiene en **0.8 ms**, asegurando que el sistema sea capaz de operar en tiempo real sin degradar el rendimiento del motor gráfico.

Figura 23 *Monitoreo de recursos del sistema antes de la ejecución*

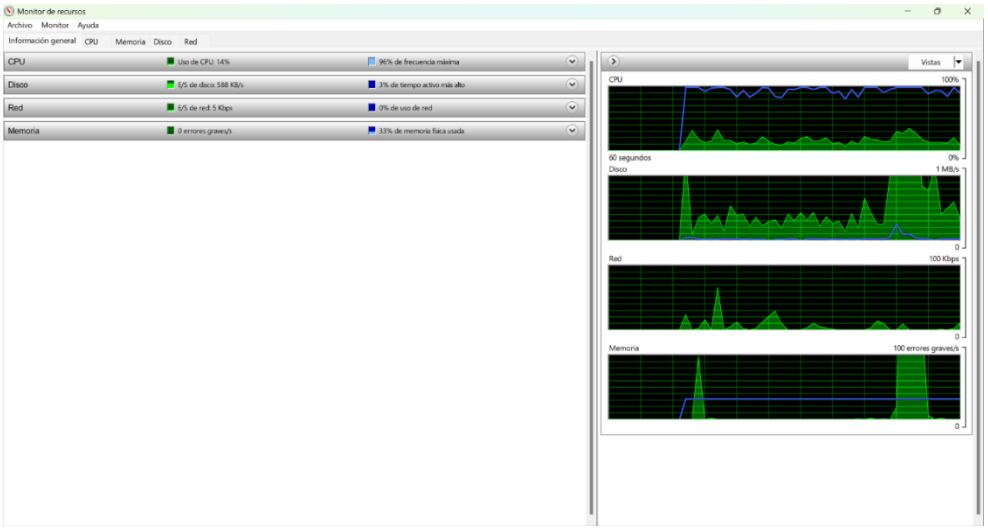
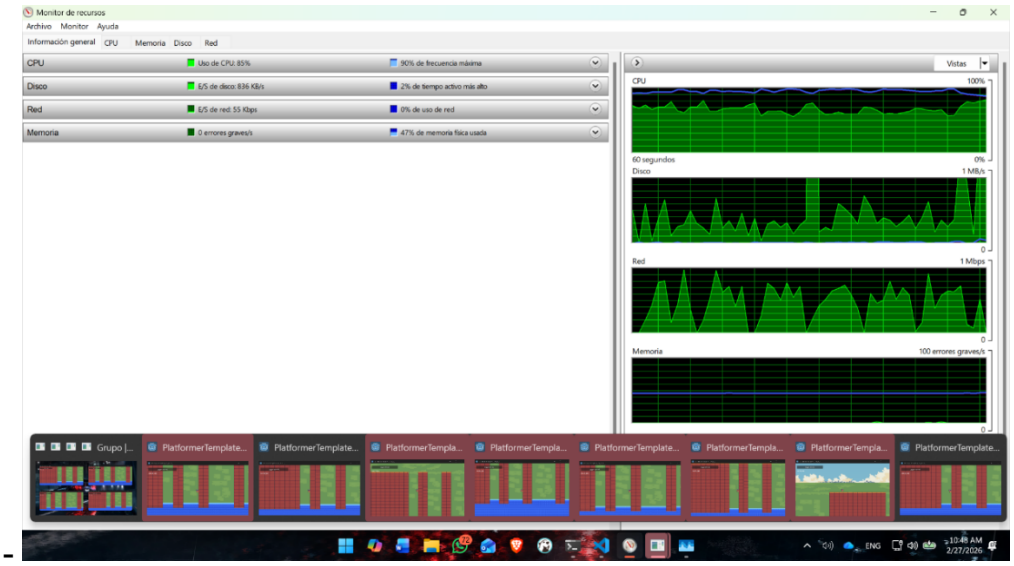
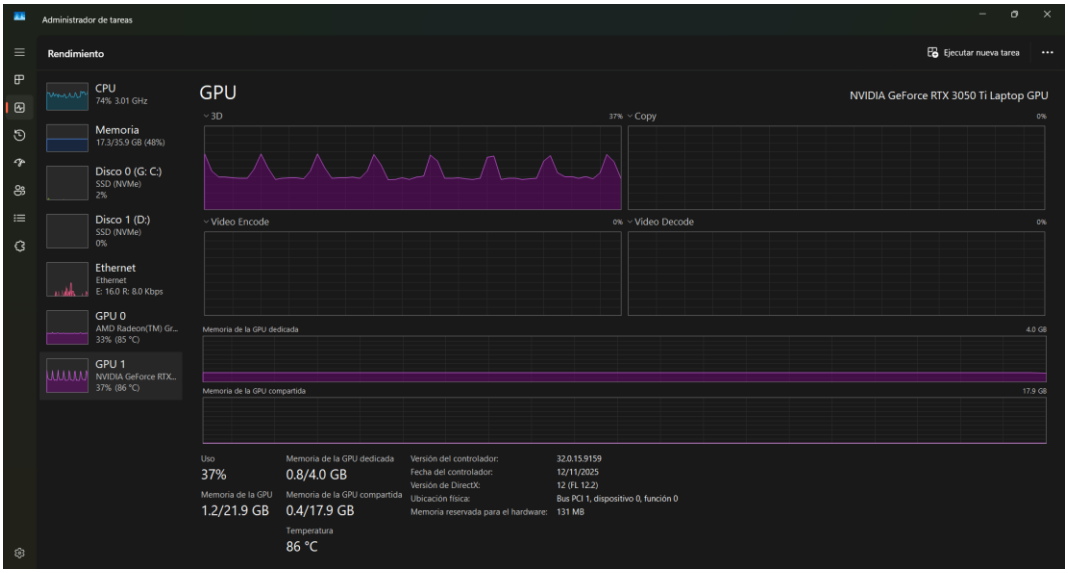


Figura 24 *Monitoreo de recursos del sistema antes de la ejecución Durante la ejecución (8 ambientes)*



-

Figura 25 *Captura de rendimiento del computador - Inferencia ONNX activa*



Finalmente, la implementación del **"Agente Fantasma"** resultó ser la herramienta de validación más potente del proyecto. Al permitir la competencia en tiempo real, se demostró que la IA toma decisiones en el aire (corrección de trayectoria) con una velocidad de respuesta que supera la capacidad neuromuscular de un jugador humano. Los resultados demuestran un equilibrio entre la exploración estocástica inicial y una política de explotación altamente optimizada, cumpliendo con el propósito de automatizar rutas eficientes en un entorno 2D

4. CAPÍTULO 4

4.1 CONCLUSIONES

Tras la finalización del desarrollo, entrenamiento y evaluación del sistema autónomo para estrategias de *speedrun*, se establecen las siguientes conclusiones:

- **Eficacia del Algoritmo PPO en Entornos Continuos:** La implementación del algoritmo *Proximal Policy Optimization* (PPO) demostró ser altamente efectiva para la optimización temporal, logrando reducir el tiempo promedio de completado a 29.8 segundos, lo que representa una mejora del 8% frente al desempeño de un jugador humano experimentado.
- **Viabilidad de la Arquitectura de Comunicación:** La integración mediante sockets TCP/IP entre Godot Engine y Python permitió una comunicación de baja latencia (1.3 ms) y un procesamiento de 60 acciones por segundo, garantizando un control en tiempo real indispensable para la precisión requerida en mecánicas de *speedrun*.
- **Impacto del Reward Shaping por Zonas:** El diseño de una función de recompensa densa, segmentada en 6 zonas de progreso, fue el factor determinante para superar puntos de estancamiento críticos. Este enfoque permitió al agente adquirir comportamientos emergentes como el uso estratégico de *dashes* y la corrección de trayectoria en el aire.
- **Consistencia Operativa frente al Humano:** El análisis estadístico confirmó que la IA posee una estabilidad operativa superior, con una desviación estándar de apenas 0.344s, eliminando la variabilidad propia del factor neuromuscular humano y asegurando una ejecución determinista de la ruta óptima.

- **Eficiencia en Inferencia Local:** La exportación del modelo al formato ONNX y su ejecución nativa en C# dentro de Godot demostraron que es posible desplegar agentes inteligentes complejos con un tiempo de inferencia de apenas 0.8 ms, facilitando la creación de herramientas interactivas como el "Agente Fantasma".

4.2 RECOMENDACIONES Y TRABAJOS FUTUROS

A partir de los resultados y las limitaciones detectadas durante las pruebas de generalización, se proponen las siguientes líneas de trabajo para futuras investigaciones:

- **Implementación de Curriculum Learning:** Para mejorar la tasa de éxito en escenarios no observados (actualmente del 72%), se recomienda implementar un entrenamiento basado en currículo, donde la complejidad de los obstáculos y la verticalidad del mapa aumenten de forma progresiva.
- **Uso de Arquitecturas Recurrentes (LSTM):** Aunque el modelo actual es eficiente, la integración de memorias Long Short-Term Memory (LSTM) permitiría al agente manejar mejor las dependencias temporales largas, ayudándole a "recordar" obstáculos pasados y planificar rutas en niveles con mayor carga de elementos dinámicos.
- **Persistencia y Automatización en el Entrenamiento:** Se sugiere añadir un sistema de persistencia automática que guarde los modelos en formato .zip de manera periódica, permitiendo reanudar entrenamientos incrementales sin pérdida de datos en caso de fallos del sistema.
- **Escalabilidad a Entornos 3D:** Una evolución natural del proyecto sería expandir el entorno de simulación a escenarios 3D simplificados, evaluando si la

arquitectura cliente-servidor actual mantiene los niveles de latencia bajo un espacio de observación y acción de mayor dimensionalidad.

- **Integración de Flujos CI/CD con GitHub Actions:** Para facilitar la colaboración y el entrenamiento remoto, se recomienda integrar procesos de integración y despliegue continuo (CI/CD) que automaticen el testeo de modelos cada vez que se realicen cambios en la función de recompensa o en las físicas del juego.
- **Desarrollo de Módulos Pedagógicos:** Dado el éxito del "Agente Fantasma", se recomienda utilizar este entorno para desarrollar un módulo educativo interactivo que enseñe fundamentos de Inteligencia Artificial y programación, permitiendo a los estudiantes modificar hiperparámetros en tiempo real y observar el cambio de comportamiento del agente.

5. BIBLIOGRAFIA

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347. https://arxiv.org/abs/1707.06347

Arulkumaran, K., Deisenroth, M. P., Brundage, M., & Bharath, A. A. (2017). Deep reinforcement learning: A brief survey. IEEE Signal Processing Magazine, 34(6), 26-38. https://doi.org/10.1109/MSP.2017.2743240

Raffin, A., Hill, A., Gleave, A., Kanervisto, A., Ernestus, M., & Dormann, N. (2021). Stable-Baselines3: Reliable reinforcement learning implementations. Journal of Machine Learning Research, 22(268), 1-8. http://jmlr.org/papers/v22/20-1364.html

Godot Engine Community. (2024). Godot Engine 4.3 documentation. Godot Engine. https://docs.godotengine.org/

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., & Zaremba, W. (2016). OpenAI Gym. arXiv preprint arXiv:1606.01540. https://arxiv.org/abs/1606.01540

Charity, M., Green, M. C., & Togelius, J. (2024). We Call This Controller Skip: AI for Speedrunning. Proceedings of the Foundations of Digital Games 2024 (FDG '24). https://ceur-ws.org/Vol-4090/paper1.pdf

Murphy, K. (2023). The first level of Super Mario Bros. is easy with lexicographic ordering and recursiveness. En Proceedings of the SIGGRAPH '23. Association for Computing Machinery.