Universidad Politécnica
de Madrid
**Escuela Técnica Superior de
Ingenieros Informáticos**

Grado en Ingeniería Informática

Trabajo Fin de Grado

Seguimiento Líneas Mediante Técnicas Robótica Evolutiva

Autor: Eduardo Miralles Ciordia
Tutor: Javier de Lope Asiain

Madrid, junio 2025

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

Trabajo Fin de Grado

Grado en Ingeniería Informática

Título: SEGUIMIENTO LÍNEAS MEDIANTE TÉCNICAS ROBÓTICA EVOLUTIVA

Junio 2025

Autor: Eduardo Miralles Ciordia

Tutor:
Javier de Lope Asiain
Inteligencia Artificial
ETSI Informáticos
Universidad Politécnica de Madrid

Agradecimientos

Quiero expresar mi agradecimiento a Javier, tutor de este Trabajo de Fin de Grado, por su guía y apoyo durante todo el desarrollo del proyecto. Su orientación ha sido clave tanto en los aspectos técnicos como metodológicos.

También quiero agradecer a la UPM haberme dado la formación y las herramientas necesarias para afrontar este reto. Estos años han sido una etapa fundamental de crecimiento personal y de aprendizaje.

Quiero hacer una mención especial a mi equipo de rugby, por enseñarme la importancia del esfuerzo colectivo, la disciplina y el compromiso, valores que también he aplicado en este trabajo y aplicaré el resto de mi vida.

A mis amigos fuera y dentro del ámbito universitario, gracias por estar siempre ahí, por sus palabras de ánimo y por ayudarme a desconectar cuando más lo necesitaba.

Y, sobre todo, a mi familia, por su paciencia, su apoyo constante y su ayuda incondicional durante todos estos años. Gracias por confiar en mí incluso en los momentos más difíciles, por su respaldo económico cuando fue necesario y por darme siempre la tranquilidad para centrarme en lo que más necesitaba. Nada de esto habría sido posible sin ellos.

¡Muchas gracias a todos!

Resumen

Este TFG (Trabajo de Fin de Grado) se centra en el desarrollo de un controlador autónomo para robots móviles, utilizando el algoritmo evolutivo *NEAT* (*NeuroEvolution of Augmenting Topologies*). El objetivo principal es diseñar un sistema robótico capaz de seguir una línea de forma eficiente utilizando *NEAT*. Dicho sistema combina el procesamiento de imágenes captadas por una cámara integrada y la información proporcionada por sensores de tipo *SONAR* como entrada a la red neuronal para poder seguir la línea de manera eficiente.

El proyecto se lleva a cabo en un entorno de simulación, empleando la plataforma *CoppeliaSim*. En ella se modela un robot virtual y una escena que incluye todos los componentes necesarios para la tarea. El controlador se entrena, incrementando la calidad de los individuos a través de generaciones. Se utiliza la información aportada por las entradas del robot para ajustar los pesos y estructuras de las redes y así optimizar su comportamiento. La calidad se define a través del *fitness* de los individuos, calculado por una función de evaluación personalizada.

La implementación técnica se realiza en el lenguaje de programación Python, integrando librerías como *neat-python* para la evolución de redes neuronales y *OpenCV* para el procesamiento de imágenes. El desarrollo abarca tanto el diseño del sistema como su entrenamiento, validación y análisis de resultados. También se documentan las decisiones tomadas durante el proceso, los retos encontrados y las posibles mejoras para el futuro. Este enfoque permite explorar la aplicación práctica de algoritmos evolutivos en entornos de robótica autónoma, pudiendo contribuir al estudio de técnicas de control adaptativo y aprendizaje automático en tareas de navegación complejas.

Para complementar el análisis del comportamiento evolutivo del controlador, se ha desarrollado código para procesar los datos. Se analizan los puntos de control guardados durante el entrenamiento, extrayendo información estadística útil por generación. Esta información facilita una mejor evaluación del rendimiento del algoritmo ya que permite identificar patrones de aprendizaje, posibles estancamientos o momentos de extinción de especies durante el proceso evolutivo.

Abstract

This Bachelor's Thesis focuses on the development of an autonomous controller for mobile robots using the *NEAT* algorithm (*NeuroEvolution of Augmenting Topologies*). The main objective is to design a system capable of efficiently following a line by combining image processing from an onboard camera and data from sonar-type sensors.

The project is carried out in a simulated environment using the *CoppeliaSim* platform. A virtual robot and scenes with paths are modeled within it, including all the necessary components for the task. The controller is trained through a genetic evolution process, adjusting the weights and structures of neural networks by giving a *fitness* value to individuals. This *fitness* value is used to optimize its behavior based on a custom function, giving the neuronal network an objective to train towards.

The technical implementation is done in the Python programming language, integrating libraries such as *neat-python* for neural network evolution and *OpenCV* for image processing. The development covers system design, training, validation, and results analysis. Decisions made during the process, challenges encountered, and potential future improvements are also documented. This approach enables the exploration of practical applications of evolutionary algorithms in autonomous robotics, possibly contributing to the study of adaptive control techniques and machine learning in complex navigation tasks.

To complement the analysis of the controller’s evolution, code has been developed to process the data generated throughout training. Checkpoints saved during training are studied, extracting stadistical information per generation. This system allows for a better evaluation of the algorithm’s performance and helps identify learning patterns, potential stagnation, or species extinction events during the evolutionary process.

Tabla de contenido

1. Introducción

El desarrollo de sistemas de control autónomos representa uno de los principales desafíos en el campo de la robótica e inteligencia artificial. Estos sistemas requieren la capacidad de percibir el entorno, tomar decisiones en tiempo real y adaptarse a condiciones inesperadas sin intervención humana. Una de los comportamientos más comunes en el campo de la robótica es el seguimiento de líneas, ya que permite comprobar la eficacia de un sistema para realizar comportamientos básicos.

En la mayoría de sistemas robóticos actuales, los comportamientos que un robot puede ejecutar han sido tradicionalmente diseñados mediante arquitecturas estáticas. En otras palabras, los ingenieros que desarrollan dichos sistemas definen de forma explícita las reglas, algoritmos y respuestas ante cada situación posible. Estos desarrollos suelen estar compuestos por varios módulos estructurados, siguiendo flujos de control predecibles y secuencias de decisión bien definidas.

Por ejemplo, en un robot tradicional el módulo encargado de la navegación podría incluir un algoritmo de planificación de trayectorias como *A*, Dijkstra* o *RRT* y unas reglas fijas para evitar obstáculos. Aunque estos enfoques han demostrado ser eficaces en entornos controlados, su principal limitación radica en que no pueden adaptarse fácilmente a cambios inesperados o condiciones no contempladas en el diseño. Si el entorno cambia de forma significativa, es necesario reprogramar el sistema o incluso rediseñar el algoritmo.

Frente a estas limitaciones, las redes neuronales y la inteligencia artificial están empezado a ser utilizadas en la robótica. A diferencia de las arquitecturas estáticas, estos métodos permiten que el robot aprenda comportamientos a partir de un entrenamiento en lugar de recibir instrucciones explícitas. El aprendizaje puede realizarse por imitación, por refuerzo, o como se hace en este proyecto, mediante evolución de redes neuronales sin supervisión directa.

NEAT (*NeuroEvolution of Augmenting Topologies*) es un algoritmo utilizado para evolucionar redes neuronales, permitiendo mejorar a lo largo de las generaciones la aptitud de los genotipos. A diferencia de otros enfoques tradicionales que requieren una arquitectura fija y un conjunto de datos etiquetados, *NEAT* parte de redes simples mínimas. A lo largo del entrenamiento, el algoritmo complejiza la red mediante mutaciones estructurales, permitiendo la generación automática de soluciones adaptadas a tareas específicas.

La incorporación de redes neuronales evolutivas representa un cambio drástico en la generación de controladores robóticos, ya que el comportamiento deseado emerge del entrenamiento y no se codifica explícitamente. Este enfoque permite encontrar soluciones no triviales e incluso contra-intuitivas que los diseñadores podrían no habrían considerado. Al permitir que el robot explore su espacio de soluciones y, por lo tanto, adaptarse a través del tiempo, se favorece la robustez y previsibilidad ante entornos desconocidos.

El uso de un entorno simulado es muy relevante cuando se trabaja con técnicas de inteligencia artificial y aprendizaje evolutivo, ya que permite superar muchas de las limitaciones asociadas al desarrollo físico de robots. Entrenar un controlador desde cero, especialmente mediante algoritmos evolutivos, requiere de una alta cantidad de evaluaciones iterativas para encontrar soluciones eficientes. En un entorno físico no solo sería lento y costoso, sino también arriesgado y repetitivo para los diseñadores.

En cambio, los simuladores proporcionan un entorno tridimensional realista donde se puede ensayar todo, incluso los fallos o condiciones adversas de manera controlada. Permiten reproducir las condiciones de prueba necesarias e incluso acelerar y automatizar la simulación. Por estas razones, utilizar un simulador no solo reduce los costes iniciales, sino que se logra obtener un prototipo funcional y robusto antes de pasar a la validación física. Esta implementación es necesaria al utilizar inteligencia artificial, ya que permite experimentar libremente con configuraciones y algoritmos sin comprometer la integridad del hardware.

Objetivo Principal

El objetivo principal de este trabajo es el diseño e implementación de un controlador autónomo, capaz de seguir una línea azul, utilizando redes neuronales entrenadas mediante *NEAT*. El enfoque se centra en lograr un comportamiento funcional y adaptable del robot dentro del entorno de simulación *CoppeliaSim*. Este proyecto busca demostrar que es posible alcanzar un controlador robusto y eficiente exclusivamente a través de técnicas evolutivas, optimizando el proceso de desarrollo mediante simulación realista.

Objetivos Específicos

A lo largo del trabajo, aparte del objetivo principal, se abordarán los siguientes objetivos:

- Analizar el funcionamiento de la técnica *NEAT* y de su aplicación a tareas de control robótico.
- Diseñar y construir un entorno de simulación y un modelo del robot virtual.
- Desarrollar e integrar un sistema de control que siga una línea eficientemente.
- Validar la eficacia del sistema en diferentes escenarios realizando pruebas.
- Generar documentación y resultados para su posterior estudio y validación.

2. Estado del Arte

Este capítulo presenta un estudio de las principales líneas de investigación y desarrollo relacionadas con el control autónomo de robots móviles. Se hace énfasis en las aplicaciones que involucran las redes neuronales y navegación sensoriales. La robótica y la inteligencia artificial son campos muy relevantes hoy en día y es crucial combinarlas para el avance de la tecnología, siendo muy útiles para la automatización de tareas en múltiples ámbitos de la vida cotidiana, tecnológica y laboral. El avance en algoritmos de aprendizaje ha permitido que los sistemas sean capaces de aprender comportamientos complejos por sí mismos, provocando una revolución en cómo se plantea el diseño de controladores.

2.1. Papel de la Robótica e Inteligencia Artificial

La integración de la robótica y la inteligencia artificial en la industria y los procesos cotidianos, laborables y sociales en las últimas décadas ha sido gradual. A lo largo de los años, ambos sectores se han convertido en pilares fundamentales en la tecnología actual. Estas dos disciplinas, aunque distintas en su origen, han generado una nueva rama: la automatización inteligente de tareas utilizando sistemas robóticos.

La robótica se centra en el diseño, construcción y control de máquinas capaces de interactuar con el mundo físico. Su uso es ampliamente reconocido en entornos industriales, donde los brazos robóticos han reemplazado tareas repetitivas en cadenas de montaje. Así, se mejora la eficiencia de la fábrica y se reducen considerablemente los riesgos laborales. Por ejemplo, los robots de ensamblaje de *Toyota* y los sistemas de logística automatizada de *Amazon* han transformado la producción y distribución de bienes a escala global, reduciendo costes e incrementando el rendimiento.

Otro campo emergente es el de la movilidad autónoma. Ciudades como Singapur y Phoenix (Estados Unidos) han realizado pruebas con flotas de autobuses autónomos y taxis sin conductor, como los desarrollados por *Waymo*. Estos vehículos cuentan con sensores y procesamiento de imágenes para navegar de forma segura en el tráfico urbano. En el sector doméstico, la robótica también ha encontrado su lugar con dispositivos como *Roomba*, una aspiradora autónoma, y asistentes de movilidad para personas mayores y con discapacidades.

La inteligencia artificial se refiere al desarrollo de sistemas capaces de imitar funciones cognitivas humanas, como el aprendizaje, la toma de decisiones y el razonamiento. Gracias a los avances en redes neuronales, se ha convertido en una tecnología presente en muchos aspectos de la vida laboral y social. Un ejemplo notable son los modelos de lenguaje generativo, como *chatGPT*, desarrollado por *OpenAI*, o *Gemini*, desarrollado por *Google DeepMind.* Estas herramientas son capaces de generar texto, resumir documentos y asistir en casi cualquier tarea, siendo utilizadas por millones de personas en labores educativas, creativas y profesionales. Dichos modelos de lenguaje destacan por estar diseñados para que aprendan por su cuenta; cuanto más se utilicen, mejores resultados se obtendrán.

Otros campos de aplicación incluyen el reconocimiento facial, presente en los *iPhone,* y los sistemas de recomendación en plataformas como *Netflix* y *Spotify.* En los últimos años, se han empezado a realizar modelos en el ámbito sanitario, pudiendo generar diagnósticos detectando patologías a partir de imágenes médicas. La inteligencia artificial también está transformando sectores como el financiero, con sistemas de predicción que son capaces de realizar varias tareas según una serie de parámetros. Estos avances, aunque graduales, han cambiado la manera en que se plantean muchos problemas y retos.

La convergencia entre la robótica y la inteligencia artificial ha abierto un amplio abanico de oportunidades. Robots que solo podían ejecutar comandos predefinidos ahora son capaces de aprender de su entorno, pudiéndose adaptar a situaciones cambiantes. Así, se mejora su desempeño con el tiempo gracias al uso de algoritmos evolutivos. Esta combinación ya se explora en vehículos autónomos, robots quirúrgicos y agricultura de precisión, permitiendo que no haya una persona supervisando los trabajos. En conjunto, estas técnicas están dotando a las máquinas de una capacidad creciente de razonamiento y adaptación, provocando un comportamiento inteligente.

2.2. Retos Actuales en el Control Autónomo

Uno de los principales desafíos es la dificultad de operar en entornos impredecibles, donde las condiciones cambian constantemente y no pueden ser anticipadas durante la programación. Por ejemplo, en el sector logístico, empresas como *Amazon Robotics* utilizan robots móviles para desplazar paquetes entre los almacenes. Si las rutas se bloquean por cajas mal colocadas, personas que se cruzan u obstáculos que no deberían encontrarse ahí, el sistema debe adaptarse en tiempo real. Este tipo de entornos requieren controladores que pueden aprender y reaccionar ante situaciones nunca vistas, lo cual sigue siendo un reto significativo para arquitecturas clásicas.

La colaboración segura y fluida entre robots y humanos es otra gran barrera tecnológica. En sectores como la industria automotriz, existen robots que trabajan junto a operarios humanos en fábricas como las de *Ford* o *BMW*. Los controladores de estos sistemas autónomos deben anticipar movimientos humanos, responder ante errores y garantizar la seguridad de los trabajadores.

Muchos de los controladores autónomos actuales funcionan correctamente en escenarios específicos, pero fallan al cambiar de entorno. Un robot entrenado para navegar en una fábrica concreta puede no funcionar en otra distribución distinta o con diferentes condiciones de iluminación. Este problema, conocido como *reality gap*, ocurre en múltiples aspectos tecnológicos, estando presente en el salto de un entorno virtual a la vida real. En un entorno virtual no se garantiza que el entrenamiento generado funcione correctamente en un entorno real, como una ciudad o una fábrica.

Google Research ha realizado un estudio sobre el *reality gap*, recopilando información de la cantidad de pruebas que se han hecho a lo largo de los años. En la figura 1 se representa un gráfico del porcentaje de tipos de entrenamiento de un proyecto. Se puede observar que, a partir del millón de muestras, la ratio de simulaciones reales con respecto a las de solo simulación aumenta considerablemente. Según *Google Research*, a partir de esta cantidad de muestreo, invertir más tiempo de entrenamiento en simulación no implica casi ninguna mejora del *reality gap*.

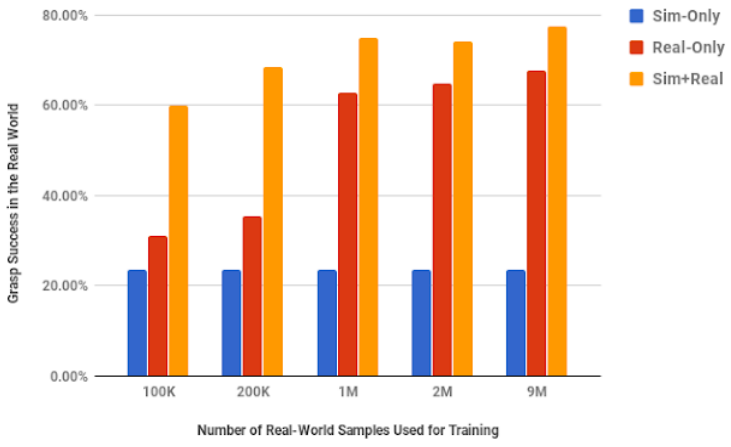

Figura 1: Gráfico, de *Google Research*, que indica el número de pruebas realizadas

2.3. Evolución de la Robótica e Inteligencia Artificial

La robótica evolutiva es una rama de la inteligencia artificial que aplica algoritmos inspirados en la evolución biológica para desarrollar el comportamiento y la estructura de robots. Estos métodos se utilizan principalmente para optimizar controladores o el diseño físico de robots sin intervención directa del programador. Una de sus principales ventajas es que permite encontrar soluciones no evidentes a problemas complejos, adaptándose a entornos variables. Además, el hecho de que solo haya que definir una arquitectura base y una función de evaluación inicial permite automatizar gran parte del proceso de diseño, liberando al programador para otras tareas mientras el sistema evoluciona por si mismo.

Uno de los enfoques más estudiados es el uso de algoritmos genéticos *(Holland, 1975).* Los algoritmos genéticos trabajan con poblaciones de posibles soluciones, aplicando operadores como selección, cruce y mutación. Estos operadores se ven aplicados al control de robots móviles, tanto en simulación como en entornos reales. Otro enfoque importante es la coevolución, donde dos o más poblaciones evolucionan de forma interdependiente y paralela, como en el trabajo de *Karl Sims (1994),* que desarrolló simulaciones de criaturas con morfologías y comportamientos emergentes, siendo convergentes en casi cualquier objetivo que se le presentaba al controlador.

En el campo de la robótica móvil, la evolución de redes neuronales ha permitido lograr avances significativos. En vez de entrenar redes a través de métodos supervisados, las redes se entrenan para controlar directamente el comportamiento del robot. Así, se permite generar adaptaciones rápidas a tareas como navegación, evasión de obstáculos y reconocimiento de patrones sensoriales. Por ejemplo, el trabajo de *Floreano y Mondada (1996)* mostró que un robot podía aprender a seguir paredes mediante un controlador neuronal evolucionado. Este trabajo era altamente modulable, permitiendo cambiar el funcionamiento del controlador con tan solo cambiar su configuración o modificar el algoritmo.

Por otro lado, la inteligencia artificial ha evolucionado rápidamente en los últimos años. A principios de siglo, los algoritmos para muchos sistemas eran toscos y sin capacidad de aprendizaje. Al introducir conceptos como redes neuronales, se fueron adaptando hasta conseguir lo que ahora conocemos como *LLMs* o *Large Language Models.*

Los algoritmos evolutivos como *NEAT* se utilizan a diario para casos más complejos. Por ejemplo, la empresa *Virtual Box* quiere utilizar un controlador para generar horarios adaptables a los autobuses de la *EMT* en Madrid. Utilizando datos ya presentes, están desarrollando una red neuronal evolutiva que consigue prevenir los picos de viajeros. Así, se permite incrementar o reducir la frecuencia de autobuses en tiempo real, reduciendo el estrés de los usuarios y gastos innecesarios de la *EMT*.

2.4. Implementaciones Actuales y sus Clasificaciones

Actualmente, existen varios campos como la logística, el transporte, la atención médica y la exploración general que han empezado a utilizar sistemas autónomos. Muchas empresas líderes han empezado a apostar por integrarlos en sus procesos productivos, generando un gran impacto en todos los niveles de la vida cotidiana. La tabla 1 presenta ejemplos de aplicaciones actuales relevantes de la robótica autónoma y las redes neuronales. Se han escogido según su utilidad y su impacto social, económico y laboral.

Proyecto/Empresa	Descripción
Boston Dynamics-Spot	Robot cuadrúpedo utilizado en inspecciones, rescates y vigilancia
Tesla-Autopilot	Sistema de conducción autónoma en vehículos comerciales
Amazon Robotics (*Kiva*)	Robots logísticos que mueven productos en centros de distribución
Starship Technologies	Robots autónomos de reparto en entornos urbanos
DJI Drones autónomos	Drones con navegación autónoma usados en agricultura, rescate y fotografía aérea.
Intuitive Surgical	Sistema quirúrgico asistido por robots con algunas funciones autónomas.
Softbank (*Robot Pepper*)	Robot social interactivo utilizado en atención al cliente
Virtual Box	Sistema de predicción en tiempo real de la flota de autobuses de la EMT

Tabla 1: Proyectos y Empresas que utilizan la robótica autónoma

Los sistemas de conducción autónoma, robótica colaborativa industrial y servicios de atención a personas destacan como las áreas con mayor impacto en la sociedad. Proyectos como *Tesla, ABB* o *Boston Dynamics* muestran una madurez tecnológica elevada, ya que ya están siendo utilizados actualmente en muchas empresas y ciudades. Por otro lado, iniciativas más orientadas a la interacción humana, como *Intuitive Surgical* y *Pepper*, aún tienen desafíos que resolver. Por ejemplo, la precisión y conocimiento que se necesitan para poder realizar una intervención quirúrgica es un desafío mucho más complejo que requiere más información y tiempo de entrenamiento.

3. Fundamento Teórico

El desarrollo de sistemas autónomos inteligentes requiere la integración de la inteligencia artificial y la robótica. Este proyecto se apoya en tres fundamentos teóricos: las redes neuronales artificiales, los algoritmos evolutivos y la navegación basada en sensores. La conjunción de estos tres campos permite diseñar agentes capaces de aprender y adaptarse a entornos dinámicos, además de reducir la necesidad de programación explícita y conocimiento previo del entorno. A continuación, se describen con mayor profundidad los conceptos teóricos del desarrollo del trabajo.

3.1. Redes neuronales para Sistemas de Control

Las redes neuronales artificiales (*RNA*) son modelos computacionales inspirados en el funcionamiento del cerebro biológico. Su estructura, formada por neuronas interconectadas, les permite representar relaciones no lineales complejas entre entradas y salidas. Esta capacidad de modelado ha demostrado ser especialmente útil en tareas de clasificación, regresión, control y toma de decisiones.

En el ámbito del control autónomo, las *RNA* permiten conectar directamente percepciones sensoriales a acciones motoras, como ruedas, brazos mecánicos o frenos. Esta propiedad es útil en robótica móvil, donde las condiciones externas cambian constantemente y es inviable generar modelos precisos *a priori*. Las redes neuronales permiten aprender a partir de la experiencia y los conocimientos obtenidos del sistema, permitiendo la adaptación a contextos nuevos sin tener que modificar el controlador.

Otra ventaja de las *RNA* es su capacidad de entrenamiento. Pueden iniciar su entrenamiento sin experiencia previa, y también tienen la posibilidad de reutilizar genotipos entrenados previamente en entornos más simples. Así, se pueden usar controladores ya entrenados como punto de partida en escenas más complejas. Esto se traduce en una reducción significativa del tiempo de entrenamiento y en una mejor estabilidad de aprendizaje.

En este proyecto, las redes neuronales actúan como el núcleo del sistema de control del robot. A diferencia de enfoques tradicionales basados en un aprendizaje supervisado, se emplea un esquema de entrenamiento evolutivo, eliminando la necesidad de disponer de grandes volúmenes de datos. Esta manera de funcionamiento permite que las redes evolucionen libremente basándose en una función de evaluación (*fitness*), descubriendo patrones para la tarea de seguimiento de línea y evasión de obstáculos.

3.2. Algoritmos Genéticos y Neuroevolución

Como se ha mencionado previamente, en este proyecto se utiliza el algoritmo *NEAT* (*NeuroEvolution of Augmenting Topologies*). Aunque existen otros algoritmos evolutivos y variantes de *NEAT*, no se han escogido por las siguientes razones:

- ***HyperNEAT*** extiende la utilidad de *NEAT* al permitir la evolución de redes de gran escala mediante codificación indirecta basada en geometría. Aunque es un algoritmo muy potente en problemas espaciales complejos (como control de brazos robóticos articulados o procesamiento de imágenes a gran escala), su implementación y configuración inicial son notablemente más complejas. En el caso de este proyecto, donde se busca un comportamiento reactivo en un entorno sensorial más básico, la expresividad de *NEAT* es suficiente y más eficiente.
- **Otros algoritmos genéticos clásicos**, como los que operan solo sobre pesos fijos de redes con topología definida, requieren que el programador determine la arquitectura de la red antes de entrenar. Esto impone una fuerte restricción sobre la adaptabilidad del sistema y limita la exploración estructural, aspectos críticos al no conocer la mejor arquitectura para el problema.
- ***CMA-ES*** (*Covariance Matrix Adaptation Evolution Strategy*) es un algoritmo muy potente para optimizar funciones en espacios continuos, pero no permite la evolución de estructuras complejas como grafos neuronales. Además, su comportamiento se degrada en espacios de alta dimensión si no se controla adecuadamente la parametrización.

Aunque existan implementaciones de estos algoritmos en algunos lenguajes, *NEAT* es el único algoritmo que permite su implementación en Python y en *CoppeliaSim*. Su funcionamiento es estable, bien documentado y potente para casos como el planteado en este proyecto. Estas ventajas han permitido una integración fluida en el sistema de entrenamiento y análisis, sin necesidad de herramientas externas o adaptaciones costosas. *NEAT* presenta varias ventajas fundamentales que lo hacen especialmente adecuado para el problema abordado:

- **Evolución simultánea de estructura y pesos**: *NEAT* parte de redes mínimas y va aumentando progresivamente su complejidad estructural mediante la adición de nodos y conexiones. Esto permite que la arquitectura de la red se adapte de forma natural a la dificultad del problema, sin necesidad de definirla *a priori*.
- **Preservación de innovación mediante historial genético**: Cada mutación estructural se registra con identificadores únicos, llamados *innovation numbers*, lo que facilita el alineamiento genético entre individuos y permite un cruce eficiente entre topologías diferentes.
- **Especiación**: *NEAT* organiza la población en especies según su compatibilidad genética. Esto protege a las innovaciones recientes frente a la presión selectiva inmediata y promueve la exploración de soluciones diversas sin caer rápidamente en mínimos locales.
- **Adecuación a tareas de control continuo**: Debido a que la salida de las redes puede ser directamente conectadas a acciones motoras, *NEAT* es especialmente efectivo en entornos de simulación donde se requiere control en tiempo real.

Por tanto, *NEAT* tiene una serie de ventajas sobre los otros algoritmos. Se ha elegido por ser el más adecuado al contexto del proyecto académico, ya que se cuenta con recursos computacionales limitados y se requiere flexibilidad, escalabilidad y facilidad de interpretación de resultados. En resumen, *NEAT* representa un punto de equilibrio ideal entre simplicidad, potencia adaptativa y control evolutivo, siendo una herramienta idónea para evolucionar redes neuronales en tareas de navegación robótica.

3.4. Entornos de simulación en robótica

El uso de entornos de simulación es muy importante en la robótica, ya que permite diseñar sistemas sin riesgos físicos, sin dedicar un espacio grande para la simulación o sin invertir grandes sumas de dinero, acelerando el proceso de prueba y error. Existen varios simuladores utilizados en desarrollo siendo algunos de los más conocidos:

- ***Gazebo*:** Ampliamente usado en proyectos que integran ROS (Robot Operating System), con gran soporte para simulaciones físicas realistas.
- ***Webots*:** Plataforma orientada a la educación y a la investigación, con buena interfaz gráfica y soporte para diferentes robots.
- ***PyBullet*:** Centrado en simulación física, con soporte para aprendizaje por refuerzo.
- ***CoppeliaSim*** (antes *V-REP*): Destaca por su flexibilidad, soporte de múltiples sensores, motores físicos y su API para integración con lenguajes como Python.

En este proyecto se ha escogido *CoppeliaSim* por su facilidad para definir escenarios personalizados, la integración directa con scripts en Python y la compatibilidad con simulaciones en tiempo real. Además, permite modelar sensores de cámara y simular la interacción con el entorno de forma precisa, lo cual es fundamental para una simulación precisa y lo más parecida a la realidad. A diferencia de otros entornos más orientados a la física avanzada, *CoppeliaSim* ofrece una solución más práctica y centrada en la evolución rápida de controladores.

3.5. Navegación Basada en Sensores

La navegación basada en sensores se entiende como la capacidad de un sistema robótico para desplazarse por su entorno, tomando decisiones de movimiento en tiempo real. Esta capacidad implica no solo el seguimiento de trayectorias predefinidas, sino también la detección y evasión de obstáculos, la adaptación a cambios en el entorno y la planificación reactiva. Los sensores a bordo de los robots son una de las bases para lograr un comportamiento autónomo.

A diferencia de los métodos que dependen exclusivamente de mapas estáticos o sistemas de localización complejos, la navegación sensorial permite obtener información del ambiente en tiempo real. Esta manera de obtener información permite operar al robot incluso en entornos desconocidos, permitiendo al controlador tener aún mayor autonomía. En el robot *P3DX* simulado, se ha implementado un sistema de percepción basado en dos tipos de sensores:

- **Cámara *RGB***: empleada para la detección de líneas en el suelo, aportando datos visuales sobre la trayectoria deseada. Mediante técnicas de procesamiento de imagen (como la segmentación en el espacio *HSV* y el cálculo del centroide del contorno), se determina la desviación de la línea respecto al eje central del robot.
- **Sensores de proximidad tipo *SONAR***: utilizados para detectar la presencia de obstáculos cercanos. Estos sensores proporcionan medidas de distancia basadas en la reflexión de ondas ultrasónicas, permitiendo inferir si el robot se encuentra próximo a objetos como paredes.

La combinación de las fuentes de información se conoce como fusión sensorial, un enfoque que mejora considerablemente la percepción del entorno al integrar múltiples perspectivas. Esta integración permite compensar las limitaciones individuales de cada sensor. Por ejemplo, la sensibilidad de la cámara a las condiciones de iluminación complementa el rango limitado de los sonares, mejorando la fiabilidad del sistema.

En este trabajo, los datos sensoriales se utilizan como entradas directas a la red neuronal del controlador, que toma decisiones motoras en función de esta información. La cámara otorga dos fuentes de información: la localización de la línea con respecto a la cámara y una variable que indica si está el robot encima de la línea. En cambio, existen 16 sensores SONAR en el robot, aportando información útil de los objetos alrededor del robot. Estas 18 entradas se utilizan para que el sistema evolucione, detecta correctamente la línea y adapte su comportamiento ante obstáculos, giros cerrados, o condiciones inesperadas.

La combinación de sensores visuales y ultrasónicos proporciona una base sólida para desarrollar comportamientos complejos, como la anticipación a colisiones, reencuentro de trayectorias tras perder la línea, y ajuste dinámico de la velocidad. Esta arquitectura multi-sensorial es especialmente útil para entornos simulados, ya que se pueden realizar variaciones de los sensores para comprobar que combinación es la correcta antes de utilizarlos en la realidad.

3.6. Limitaciones y Desafíos en Control Autónomo

A pesar de los avances tecnológicos recientes, el control autónomo de robots sigue presentando múltiples retos que deben ser considerados en el diseño e implementación de los controladores. Estos desafíos emergen tanto de las características propias del entorno como de las limitaciones inherentes a los algoritmos empleados y a la tecnología sensorial disponible.

Uno de los principales obstáculos es la incertidumbre ambiental. Los robots móviles operan en entornos donde las condiciones pueden ser impredecibles. Factores como la iluminación, la presencia de obstáculos móviles, las superficies irregulares o incluso condiciones meteorológicas (en casos reales) afectan de forma directa a la percepción sensorial y la capacidad de decisión del agente. En este sentido, la simulación resulta una herramienta valiosa, ya que permite crear entornos controlados donde se puede aislar el impacto de estas variables y evaluar de forma precisa el comportamiento del sistema.

Otro obstáculo es la imprecisión de los sensores, que introducen ruido o valores nulos que pueden deteriorar la calidad del controlador. Una red neuronal entrenada con datos ruidosos o inconsistentes puede aprender comportamientos erróneos o ineficaces. Por esta razón, hay que asegurarse de que los componentes usados, sean simulados o reales, funcionen correctamente y devuelvan consistentemente resultados correctos. Es importante comprobar también si se usa un sistema simulado en un sistema real o viceversa, ya que los sensores y cámaras en un entorno pueden no funcionar de la misma manera en otro entorno.

La generalización del comportamiento aprendido es otro gran desafío. Un controlador entrenado para seguir una línea en un entorno puede no funcionar correctamente si se traslada a un escenario diferente. Esto se debe a que las redes neuronales pueden sobreajustarse al entorno de entrenamiento, perdiendo su capacidad de adaptación ante nuevos estímulos. Este fenómeno, conocido como *overfitting*, se agrava en simulaciones si no se introducen suficientes variaciones estructurales durante el entrenamiento. El *reality gap*, que, como se ha dicho previamente, es la diferencia entre el entorno simulado y el entorno físico real, puede provocar que un sistema funcional en simulación no lo sea en la realidad, aunque todo funcione igual *a priori*.

Una de las mayores limitaciones es la definición de la función de evaluación de *fitness*. Si esta no representa adecuadamente el comportamiento deseado, el sistema evolutivo puede converger hacia soluciones localmente óptimas, pero alejadas del comportamiento esperado. Por ejemplo, si se recompensa únicamente la permanencia sobre la línea sin penalizar el retroceso o la velocidad negativa, la red neuronal aprende a mantenerse estática o retroceder. Si no se tiene experiencia o conocimiento sobre el funcionamiento del control autónomo, la correcta implementación del comportamiento puede ser laboriosa.

Desde el punto de vista computacional, los métodos basados en evolución presentan una carga significativa. La evaluación de múltiples individuos por generación, durante decenas o cientos de generaciones, implica un alto consumo de recursos. A mayor complejidad, se invertirá más tiempo para converger en el comportamiento deseado, incrementando aún más los costes computacionales. Este fenómeno se puede mitigar incorporando una serie de técnicas, como ajustar los parámetros de los algoritmos o aplicando gradualmente los comportamientos deseados.

A pesar de estas limitaciones, la combinación de neuroevolución y navegación sensorial representa una alternativa potente y flexible para el desarrollo de controladores autónomos adaptativos. La capacidad del sistema para aprender de la experiencia, mejorar con el tiempo, y adaptarse sin necesidad de modelos explícitos lo convierten en una estrategia especialmente adecuada para muchas tareas. Superar los retos mencionados es esencial para avanzar hacia sistemas capaces de operar en entornos reales con eficacia.

4. Desarrollo

El sistema propuesto ha sido desarrollado con el objetivo de simular y entrenar un robot móvil autónomo, capaz de seguir una línea y evitar obstáculos de forma eficaz utilizando redes neuronales. Para ello, se ha construido una arquitectura modular escrita en Python e integrada con el simulador *CoppeliaSim* a través de una *API*, siendo diseñada para ser flexible y eficiente en ciclos de entrenamiento prolongados. Esta sección describe en detalle cómo se ha construido el entorno de desarrollo, el modelo simulado del robot, las herramientas utilizadas y el flujo general de trabajo.

4.1. Configuración Entorno

El desarrollo se ha llevado a cabo en un entorno de simulación mixto: mientras que el comportamiento físico y sensorial del robot se simula en *CoppeliaSim*, la lógica de control y el entrenamiento de las redes neuronales se ejecutan en Python. Esta separación permite aprovechar las fortalezas de ambos entornos: el motor físico de *CoppeliaSim* reproduce con alta fidelidad las condiciones del mundo real, mientras que Python aporta flexibilidad y potencia para el entrenamiento mediante algoritmos evolutivos y procesamiento de imágenes.

La comunicación entre ambos entornos se gestiona a través de la *ZeroMQ Remote API*, utilizando el archivo *robotica.py* como interfaz de alto nivel. Esto permite mantener un ciclo de entrenamiento eficiente, en el que cada individuo de la población evolutiva se prueba directamente en simulación y se evalúa mediante una función de *fitness* personalizada.

4.1.1. Coppelia Robotics

CoppeliaSim, anteriormente conocido como *V-REP*, es una plataforma de simulación robótica que permite modelar, simular y controlar robots en entornos tridimensionales complejos. Su arquitectura abierta, basada en scripts de control internos llamados *Lua* y *APIs* externas, como *ZeroMQ* o *ROS*, facilita la integración con lenguajes como Python, Matlab o C#. En este proyecto se ha utilizado para simular un entorno cerrado con una trayectoria de línea azul y obstáculos simples. El robot simulado es una versión virtual del *Pioneer P3DX*, un robot móvil ampliamente utilizado en investigación.

Una ventaja clave de *CoppeliaSim* es su sistema de *step-based simulation*, que permite ejecutar la simulación paso a paso con total control. Cada *step* equivale a 50 ms, lo que se traduce en 20 *steps* por segundo y 1.200 por minuto. Esta granularidad ha permitido establecer métricas temporales precisas durante el entrenamiento, como la cantidad de tiempo que un robot permanece alineado con la línea o el número de *steps* que pasa atascado frente a una pared. Estas métricas se han utilizado como parte del cálculo de *fitness* y han sido una parte esencial para este proyecto.

La clase *Coppelia* definida en *robotica.py* encapsula la lógica de arranque y parada del simulador, así como la optimización del rendimiento. El único ajuste que se ha cambiado es el parámetro *displayEnabled*, que desactiva el renderizado gráfico durante el entrenamiento y así reduce el tiempo de simulación de 60 a 45 segundos.

4.1.2. Modelo Robot y Sensores

El robot simulado es una versión virtual del *Pioneer P3DX*, equipado con los siguientes elementos funcionales:

- **2 motores de tracción diferencial**, que permiten controlar el movimiento del robot ajustando la velocidad de cada rueda por separado. Cada rueda puede tener velocidad positiva o negativa de manera independiente.
- **16 sensores ultrasónicos de tipo *SONAR*** ubicados alrededor del cuerpo del robot. Estos sensores proporcionan medidas de distancia a obstáculos cercanos, lo que permite realizar maniobras de evasión y navegación segura.

- **Cámara *RGB* frontal**, utilizada para detectar la línea azul que el robot debe seguir. Las imágenes capturadas se procesan en tiempo real usando *OpenCV* para calcular la alineación del centro del robot respecto a la línea.
- **Sensores *LIDAR***, que no ha sido utilizado directamente en este proyecto, pero se puede contemplar si se quiere expandir con más funcionalidad.

Esta combinación sensorial, integrada mediante la clase *P3DX* en *robotica.py*, permite simular un entorno sensorial realista con bajo coste computacional. Los sensores *LIDAR* son un sistema de medición y detección de objetos mediante láseres, mientras que los sensores *SONAR* usan ultrasonidos para detectar contornos de objetos. Se decidió utilizar los *SONAR* por encima de los *LIDAR* porque no se necesita saber la distancia precisa de las paredes, solo detectarlas y diferenciarlas de otros objetos. Ya que los sensores *SONAR* no utilizan tanta capacidad de procesamiento, el proyecto se puede ejecutar en una mayor cantidad de ordenadores.

Si en el futuro se quiere implementar más funcionalidad al controlador, como seguir una bola, se pueden usar los sensores *LIDAR* para realizar la tarea. No se ha modificado en *robotica.py* nada de la funcionalidad de estos sensores, con lo que bastaría con llamar la función correspondiente al objeto del robot para utilizarlos.

4.1.3. Herramientas y Librerías Extras

El sistema completo combina múltiples tipos de software previamente implementados para ofrecer una solución modular, extensible y eficiente. A continuación, se describen las principales herramientas empleadas:

1) ***API* personalizado**: Una parte fundamental del sistema es el archivo *robotica.py*, desarrollada por el tutor. Esta clase actúa como la *API* de conexión entre el simulador *CoppeliaSim* y la aplicación Python principal, desarrollada en *VSCode*. Su diseño encapsula las operaciones básicas de control y lectura de sensores del robot, conteniendo dos clases principales:
   - *Coppelia:* Controla el arranque, parada y estado de la simulación. También ajusta parámetros de rendimiento gráfico para acelerar las ejecuciones durante el entrenamiento evolutivo.
   - *P3DX:* Representa el robot simulado y proporciona métodos para obtener distancias de los sensores ultrasónicos, capturar imágenes desde la cámara *RGB* y establecer velocidades individuales para las ruedas.

   Esta *API* ha sido esencial para mantener el código limpio y modular, permitiendo separar la lógica de control de manejo de la simulación. Gracias a esta interfaz, se ha conseguido reducir el tiempo de integración, permitiendo dedicarle más tiempo al controlador y evitar errores derivados del manejo, ya que es una *API* que ha sido testeada al 100%

2) ***neat-python**:* Implementa el algoritmo *NEAT*. En este proyecto se ha usado para entrenar redes que transforman entradas sensoriales en comandos de movimiento. La configuración establece 18 entradas y 2 salidas, activación tanh, sin capas ocultas iniciales, y un enfoque de evolución agresiva.

3) ***OpenCV*:** Utilizada en la clase principal *avoid.py* para procesar las imágenes procedentes de la cámara. La clase segmenta la línea azul y extrae características como el centroide de la línea, su proporción en la imagen, y el ángulo respecto al eje del robot.

4) ***matplotlib*, *numpy*, *os*, *pickle*, *glob:*** Utilizadas principalmente en la clase auxiliar *treatment.py* para analizar los resultados del entrenamiento. Estas herramientas permiten cargar automáticamente puntos de control (*checkpoints*), visualizar la evolución del *fitness*, detectar estancamiento evolutivo y representar la diversidad de especies con gráficos detallados

5) ***Treatment.py*:** Clase auxiliar creada para poder procesar los *checkpoint* generados por la librería *NEAT*. Con solo correr el comando "py treatment.py" en el directorio que se encuentra el archivo, ejecuta automáticamente el procesamiento de datos. La información que se saca es:
    - *Fitness* medio, mejor y peor por generación
    - Evolución por especies
    - Generación de gráficos para una mejor visualización de información
    - Sacar toda información en el archivo *robot-info*

4.2. Diseño Experimental y Estrategias de Optimización

Esta sección describe los métodos y decisiones técnicas implementadas para optimizar el desarrollo, evaluación y análisis del sistema. Dado el alto coste computacional del entrenamiento evolutivo, es fundamental extraer información útil durante el proceso y comprobar la eficacia del controlador. Se han definido varias estrategias para lidiar con estos problemas, permitiendo acelerar los ciclos de entrenamientos sin comprometer la calidad de los resultados y facilitando la interpretación de manera intuitiva el rendimiento evolutivo de los individuos.

4.2.1. Optimización del tiempo de Modelaje

Durante la fase de entrenamiento y evaluación de los individuos, se identificó que la simulación en tiempo real puede volverse muy costosa si no se implementan mecanismos de control sobre la duración de los entrenamientos. Por este motivo, se cambiaron varios ajustes para optimizar el tiempo de modelaje, logrando una reducción significativa del tiempo de cómputo sin afectar a la calidad del aprendizaje.

El primer ajuste consistió en limitar la duración máxima de cada evaluación a 1200 *steps*, lo que equivale a 60 segundos reales de simulación. Esta restricción permite detener la ejecución de un individuo una vez superado el umbral, evitando las ejecuciones innecesariamente largas. Se ha optado por este valor al comprobar que el robot, con un comportamiento ideal, es capaz de completar una vuelta entera a ambos circuitos de prueba. Las evaluaciones con tiempos superiores no aportaban mejoras relevantes, suponiendo un gasto ineficiente de recursos.

En segundo lugar, se implementó una detención anticipada cuando el individuo presentaba un comportamiento ineficiente o estancado. Si el robot permanecía quieto, giraba sobre sí mismo o mostraba una velocidad inapropiada durante más de 4 segundos, se finalizaba la evaluación de manera automática. Este criterio permite filtrar individuos no prometedores de manera temprana, reduciendo considerablemente el tiempo de entrenamiento.

Por último, se optimizó el sistema aún más desactivando la visualización gráfica de la simulación. El parámetro *displayEnabled* en el archivo *robotica.py* se estableció a *False*, desactivando la visualización del entorno en tiempo real. Este cambio permitió acortar la duración de cada simulación de 60 a 45 segundos, manteniendo los 60 segundos de tiempo lógico simulado. Este cambio representa una reducción del 25% del tiempo total por individuo. El cambio de este parámetro no afecta a la calidad del entrenamiento, ya que toda la información relevante es procesada internamente y no depende de la representación gráfica.

En conjunto, estos tres ajustes reducen el tiempo total de simulación hasta en un 60%, permitiendo realizar más generaciones y pruebas en menos tiempo. Este cambio ha sido especialmente útil durante el desarrollo del sistema, donde se han ejecutado más de 20 simulaciones completas. Por ejemplo, una simulación estándar que antes podía tardar 15 días, se ha completado en poco más de 6. Esta reducción de tiempo ha sido esencial para poder realizar el proyecto dentro de los plazos establecidos y ha demostrado ser una decisión clave en la viabilidad de la solución propuesta.

4.2.2. Generación de información

Durante la ejecución y el entrenamiento del controlador, se genera una cantidad significativa de información. Esta información es extremadamente útil para el análisis posterior del comportamiento del robot y la evolución de la red neuronal e incluye:

- **Datos de rendimiento** de cada individuo por generación: valor de *fitness*, alineación con la línea, velocidad media o evasión de obstáculos, entre otros.
- **Historial de *checkpoints* evolutivos**, que permiten reanudar entrenamientos y comparar genotipos en distintos puntos del proceso. Se producen en cada generación.

Toda esta información es analizada posteriormente por la clase definida en el archivo *treatment.py*, que se encarga de procesar los *checkpoints* guardados. Esta herramienta personalizada extrae métricas clave**,** facilitando una comprensión más profunda del proceso evolutivo. Además, se generan gráficos que representan visualmente la evolución del desempeño del sistema, permitiendo identificar patrones de mejora, momentos de estancamiento o incluso la extinción de especies evolutivas. Esto convierte a la generación de información en una herramienta muy útil tanto para la validación del controlador como para la mejora del sistema. Los gráficos que se generan son:

- Media del *fitness*
- Mejor genotipo
- Peor genotipo
- Combinación de todos los valores.

4.2.3. Pruebas y Validación

Para validar la efectividad del controlador evolutivo, se han llevado a cabo múltiples pruebas dentro del entorno simulado de *CoppeliaSim*, utilizando dos escenas distintas. El primer mapa es una escena simple con una línea circular, presente en la figura 2, mientras que la segunda escena más compleja es una línea que serpentea. Este segundo mapa es el de la figura 3.

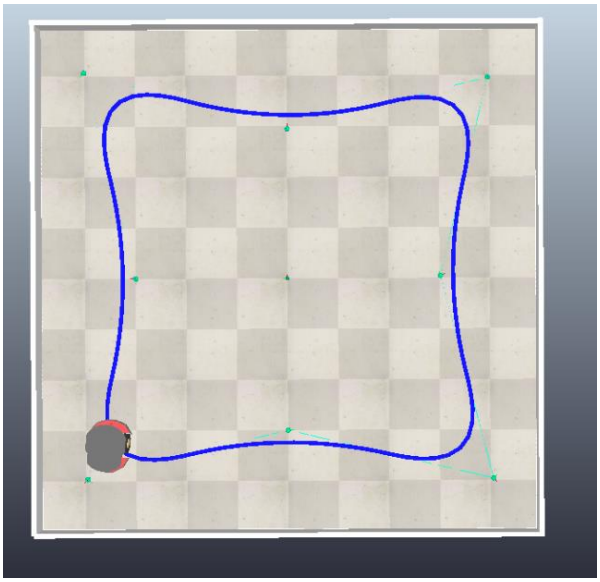

Figura 2: Escena con línea circular

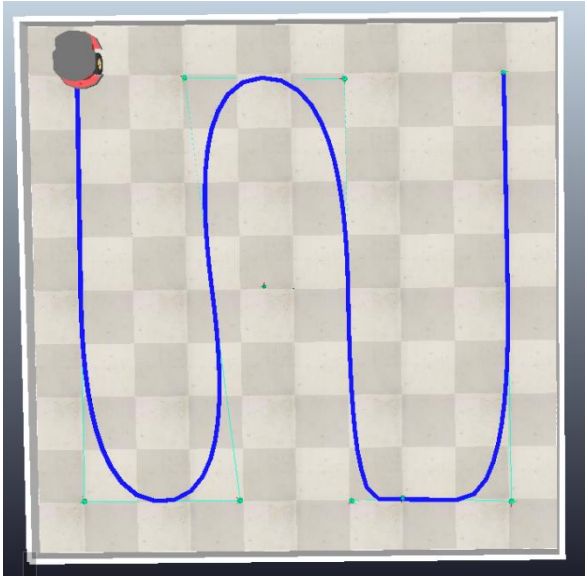

Figura 3: Escena con camino serpenteante

Durante el entrenamiento, se realizaron evaluaciones por generación y se guardaron *checkpoints* que permitieron revisar el progreso evolutivo. Los *checkpoints* tienen dos ventajas principales: la primera es que permite retocar los ajustes entre generaciones, ajustando y modificando el comportamiento sin necesidad de iniciar el entrenamiento desde 0 de nuevo; la segunda es que permite entrenar el mejor genotipo de un circuito en otro, permitiendo que aprenda a tratar el nuevo circuito sin necesidad de volver a aprender comportamientos erróneos.

La validación del controlador incluye:

- **Medición del *fitness*** para acelerar la simulación. El valor máximo teórico de *fitness* que puede obtener un controlador es de 3600.
- **Comparación entre especies**: se observa la diversidad y evolución paralela de topologías diferentes. Si se extinguen las especies presentes, se termina el entrenamiento.
- **Reproducción del comportamiento del mejor individuo** en entornos no vistos durante el entrenamiento, demostrando la calidad del controlador
- **Análisis cuantitativo y gráfico**: utilizando el módulo *treatment.py*, se pueden obtener gráficos de evolución de las estadísticas por especie. Si se observa un estancamiento durante más de 30 generaciones, es una indicación de que el controlador no va a aprender más.
- **Finalización automática del entrenamiento** si no se llega al valor máximo de *fitness* de 3600, hay estancamiento o se extinguen todas las especies. El límite se ha establecido en 150 generaciones. Si no se cumplen ninguna de las condiciones, se termina la ejecución y se devuelve el mejor genotipo hasta entonces.

El *fitness* máximo de 3600 se obtiene a partir de los valores implementados en la función *Calculate_fitness*. El individuo ideal mantiene siempre la línea centrada, durante los 60 segundos de la simulación. Por lo tanto, ya que en cada *step* de simulación se suman 2 puntos si el individuo mantiene la línea centrada, el individuo obtiene 2400 (2 * 1200), más los 1200 por mantenerse durante los 60 segundos de simulación, logrando el total de 3600 de *fitness*, el máximo que se puede obtener. Se entrará en más detalle de qué comportamientos aportan o restan *fitness* en el apartado 4.3.4.1.

Estas validaciones han permitido demostrar que el controlador implementado es capaz de aprender de manera autónoma una política eficaz de navegación. El controlador, como se mostrará a continuación, presenta una mejora progresiva y estable a lo largo de las generaciones, adaptándose mejor al objetivo principal de seguimiento de líneas.

4.2.4. Automatización del Proceso de Entrenamiento

Durante el desarrollo del sistema, se ha implementado una gestión automatizada del flujo de simulaciones. Esta automatización permite que el sistema entrene durante largos periodos de tiempo sin supervisión directa, siendo especialmente útil dada la alta carga computacional de *NEAT*.

Uno de los métodos utilizados es el uso de *checkpoints*. Los *checkpoints* son archivos intermedios que almacenan el estado completo de la población evolutiva. Incluyen datos como los mejores individuos, la generación a que pertenecen, el historial del *fitness* y el número de especies presentes. En este proyecto se ha configurado la generación de un *checkpoint* tras cada generación completada, utilizando el método proporcionado por la librería *neat-python*.

Estos archivos permiten reanudar el entrenamiento exactamente desde el punto en que fue interrumpido, evitando pérdidas de progreso en caso de cierre inesperado, interrupción en el suministro eléctrico o pausa manual. Aunque el sistema no contempla un reinicio desde cero en caso de fallo, sí está preparado para gestionar la extinción completa de especies. Si ocurre una extinción masiva (provocada por genotipos demasiado cercanos en su *fitness* durante más de 20 generaciones), *NEAT* conserva automáticamente el mejor genotipo encontrado hasta ese momento y finaliza el entrenamiento. Así evita que se pierdan soluciones válidas y acelera futuras iteraciones en el proceso.

4.3. Evolución Controlador

4.3.1. Diseños de la Detección de Imágenes

Uno de los elementos más críticos para lograr un seguimiento eficaz de la línea ha sido el procesamiento visual, encargado de interpretar los datos procedentes de la cámara del robot. A medida que el proyecto fue evolucionando, también lo hizo la forma en que se realizaba esta detección. Inicialmente, se empleaban enfoques generales que solo verificaban la presencia de la línea, pero con el tiempo se incorporaron análisis más estructurados, como la estimación de coordenadas y métricas centradas en la posición relativa de la línea con respecto al robot.

4.3.1.1. Primera Versión: Detección Línea Completa

La primera implementación visual se centraba exclusivamente en detectar la presencia de píxeles azules en el campo de visión de la cámara, sin importar su posición o distribución. La imagen se segmentaba en una región y se aplicaba una máscara binaria que destacaba las zonas azules. Si la cantidad de píxeles superaba un cierto umbral, se consideraba que la línea estaba presente. Este método era muy simple y permitía verificar si el robot detectaba algo azul, pero no ofrecía ninguna noción de alineación.

Otro problema fundamental es que tenía un *mask* muy simple. Al querer detectar la línea entera, los individuos no conseguían distinguir tipos distintos de tramos de línea con respecto al robot. Por lo tanto, provocaba que el sistema creyese que estaba siguiendo correctamente la línea cuando no lo hacía, dando falsos positivos. Como no se penalizaba la mala alineación ni se distinguía entre una línea centrada o periférica, surgieron las soluciones engañosas de robots que giraban o se mantenían cerca de la línea sin seguir ningún patrón útil.

Esta implementación fue útil como base, pero rápidamente resultó insuficiente para guiar conductas complejas. La primera versión de la detección se utilizó muy poco tiempo, y solo en la primera versión de *fitness*. Al comprobar su ineficiencia, rápidamente fue reemplazada por la segunda versión.

4.3.1.2. Segunda Versión: Detección por coordenadas Cx y Cy.

La segunda versión introdujo un enfoque más geométrico mediante el cálculo del centroide de los contornos detectados. Utilizando momentos de imagen, se determinaban las coordenadas *Cx* (horizontal) y *Cy* (vertical) del centro de masa de la línea azul. En esta implementación se usó la primera segmentación real de espacio, permitiendo una base geométrica muy útil para las siguientes versiones.

Este cambio permitió obtener una referencia posicional de la línea en el campo visual del robot, lo cual resultó esencial para decisiones de dirección. Por ejemplo, si *Cx* tenía un valor negativo, significaba que la línea estaba a la izquierda y el robot podía girar hacia la derecha para recentrarse. Si *Cy* aumentaba positivamente, significaba que la línea giraba 90 grados, por lo que el robot podía prevenirlo y adaptarse de manera óptima.

Aunque representó un avance significativo, esta versión todavía presentaba ambigüedades: el valor de *Cx* dependía del tamaño y forma del contorno visible, lo que podía variar por ruido o perspectiva. Además, los valores de *Cy* no aportaban información relevante y solo obscurecían el entrenamiento, ya que *Cx* indicaba correctamente los giros bruscos. Esta versión de la cámara fue utilizada en las versiones de fitness 4.3.2.1 y 4.3.2.2.

4.3.1.3. Tercera Versión: Offset Linea respecto Centro Robot

Esta versión transformaba la información posicional obtenida en un único parámetro cuantificable: el *line_offset*, que mide la distancia entre *Cx* y el centro de la imagen. En otras palabras, se calculaba la desviación de la línea con respecto al centro físico del robot. Este valor se obtenía a partir de la posición central del robot con respecto a la posición de la línea de la cámara. Dicho *offset* resultaba en un valor entre 0 y 1, indicando si el robot estaba mal o bien alineado respectivamente.

Este parámetro tiene múltiples ventajas con respecto a la segunda versión. Al ser un valor numérico continuo y normalizado, es fácilmente interpretable por la red neuronal. Puede ser usado directamente como entrada para la red neuronal y como recompensa en la función de *fitness*, demostrando su alta flexibilidad. Finalmente, permitía penalizar desviaciones y recompensar alineaciones de forma precisa.

Esta implementación no solo aportaba una descripción objetiva del entorno visual, sino que también se adaptaba naturalmente a situaciones con curvas, cambios de trayectoria o pérdidas temporales de línea. Se ha utilizado en las versiones de fitness 4.3.2.3 y 4.3.2.4

4.3.2. Diseños de la Función de *Fitness*

La función de *Fitness* es la parte más importante del aprendizaje de las redes neuronales, indicando qué comportamientos son correctos y aportando una calidad a los individuos. Una implementación errónea o poco robusta puede provocar variaciones en las respuestas o individuos que convergen a un objetivo incorrecto.

4.3.2.1 Primera Versión: Fitness por Potencias

Esta versión se centraba en recompensar al individuo según si estaba en la línea o no (a través de una variable que simplemente detectaba si la línea estaba presente en la cámara). Calculaba el *fitness* usando el tiempo que pasaba en la línea a través de recompensas que se incrementaban exponencialmente. En otras palabras, el individuo generaba un valor de calidad dependiendo de su tiempo en la línea al cuadrado. Por ejemplo, si el robot estaba 30 segundos siguiendo la línea, obtenía un valor de 900.

En esta versión se ha implementado también un sistema de penalización básica. Se contaba el tiempo que estaba el robot dando vueltas sobre sí mismo, retrocediendo o se quedaba atascado en una pared y le restaba al *fitness* una puntuación equivalente al tiempo empleando estas actividades al cuadrado. Por ejemplo, si un robot llevaba diez segundos atascado en una pared, se veía penalizado por 100. Dichos valores eran demasiado altos al principio y se fueron reduciendo a través de los entrenamientos que se hicieron. Así, se daba más importancia al *fitness* positivo de seguir líneas, que se mantuvo igual en todas las pruebas.

Esta implementación permitió iniciar el entrenamiento, pero no fomentaba el comportamiento deseado. Al ganar o perder puntuación, la diferencia entre valores eran elevadas, provocando que *NEAT* no supiese distinguir correctamente entre comportamientos [*Figura 4*]. Se creía que el problema era el uso de tiempo en vez de *steps* y no el uso de ganancia de *fitness* exponencial, por lo que se decidió cambiar el tiempo por los *steps* acumulativos en la segunda versión.

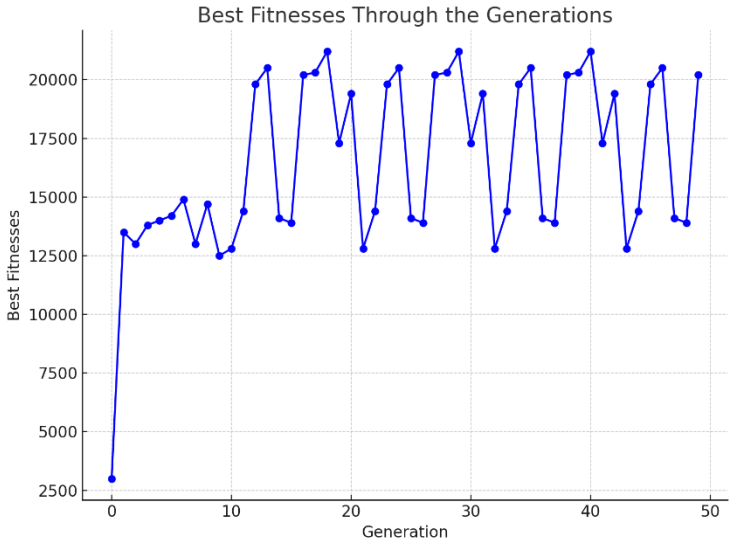

Figura 4: valores extremos por función exponencial

4.3.2.2 Segunda Versión: Steps Exponenciales

Al considerase que el problema de los valores extremos se originaba del tiempo y no por la obtención de *fitness* exponencial, se decidió utilizar los *steps* de la simulación. Cada vez que el individuo realizaba una acción negativa o positiva, un contador se incrementaba y se llamaba a la función de *fitness.* Si el individuo paraba de realizar dicha acción, el contador reseteaba a 0. Esto provocaba que, cuanto más tiempo el individuo realizaba una acción, más penalizado o premiado era.

Con esta versión de *fitness*, el aprendizaje siempre tendía a generar una respuesta local. Por "localidad", o "valle", se entiende el algoritmo evolutivo que ha encontrado un objetivo que cumple parcialmente el objetivo principal, pero no es capaz de conseguir el objetivo principal por mucho quese innove [*Figura 5*]. Independientemente del número de entrenamientos que se realicen, el aprendizaje siempre va a tender a una respuesta local en vez de cumplir el objetivo principal. Por ejemplo, los mejores genotipos de la Figura 5 simplemente divagaban por el mapa a máxima velocidad sin tener en cuenta la línea.

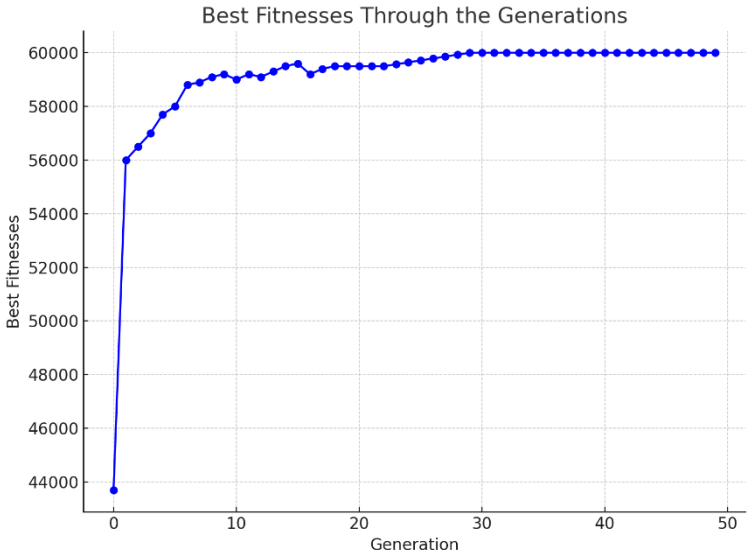

Figura 5: genotipo con respuesta local

4.3.2.3 Tercera Versión: Fitness por 'factor_alineación'

Al observar que los valles generados por la versión 4.3.2.2 se debían a que los individuos no lograban mantenerse correctamente en la línea, se introdujo en el cálculo del *fitness* el factor de alineación. Dicho factor tenía un valor de 0 a 1, que indicaba la distancia de la línea al centro de la cámara. Si la línea se encontraba en el centro, tenía un valor de 1, y si la línea se encontraba en el extremo de la imagen, tenía un valor de 0. Por ejemplo, en la *Figura 6*, el factor de alineación tendría un valor de 0.80.

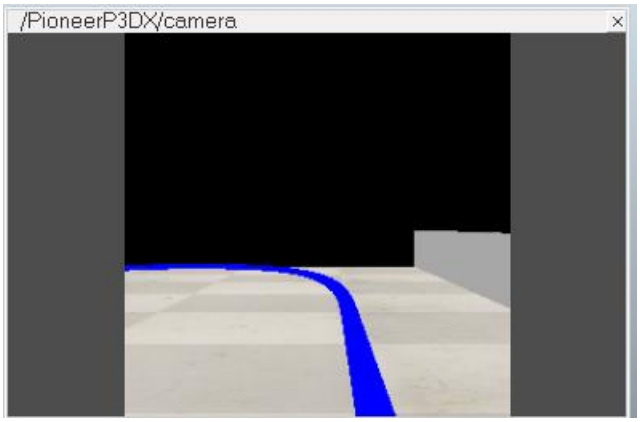

Figura 6: Línea con factor de 0.80

Este factor de alineación se utilizó para dar más información a *NEAT* de lo bien alineado que estaba el individuo con la línea. En el cálculo de calidad, los *steps* acumulados por seguir la línea eran multiplicados por el factor de alineación, provocando que, que a mayor alineación del individuo, obtuviera mayor cantidad de *fitness*. Se puede observar en la figura 7 que el problema de respuestas locales mejoró ligeramente. El crecimiento era muy lento, con algunos picos importantes; pero positivo hasta la generación 35, donde se detectaron comportamientos erráticos que resultaban en una inhabilidad del controlador en aprender comportamientos correctos.

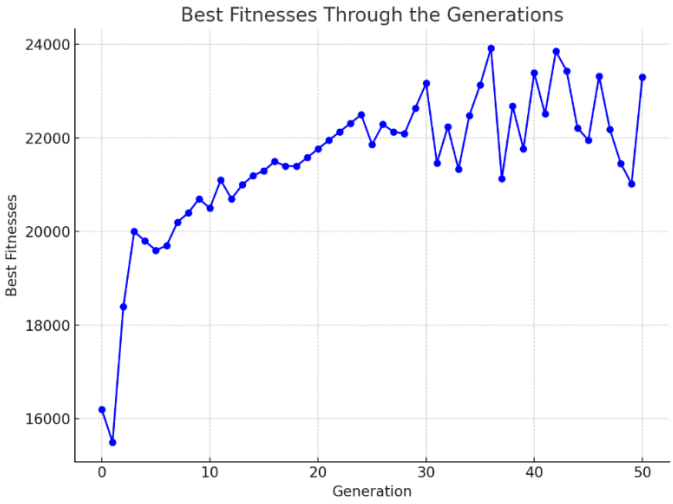

Figura 7: Crecimiento positivo hasta encontrar una respuesta local

Aunque se reiniciase el entrenamiento, los individuos no aprendían a seguir la línea. Los primeros individuos premiaban explorar el escenario, generando un objetivo que no era el esperado, ya que los individuos preferían seguir paredes en vez de a la línea. Después de investigar y estudiar otros proyectos que implementaban *NEAT*, se llegó a la conclusión de que los *fitness* generados eran demasiado grandes, provocando que la solución ideal para el aprendizaje no fuese la esperada. El problema fundamental consistía en que la diferencia entre genotipos era tan grande que la red neuronal no era capaz de discernir qué acciones provocaban qué cálculo de *fitness*.

En esta versión se implementó la finalización prematura de individuos que no tenían comportamientos correctos. Así, a partir de esta versión, los individuos no ganaban *fitness* de manera activa por el tiempo que el robot estaba siendo simulado. También, el entrenamiento era mucho más corto, ya que no hace falta utilizar el minuto entero de simulación en un individuo con comportamiento erróneo, reduciendo drásticamente el tiempo en las primeras generaciones.

4.3.2.4 Cuarta Versión: Steps Lineales

Para que los saltos entre generaciones no fuesen tan dispares, y por lo tanto obscureciendo a la red neuronal del objetivo, se decidió cambiar el cálculo del fitness con respecto a los *steps*. En las versiones anteriores, la función de *fitness* utilizaba un contador de *steps* que incrementaba exponencialmente de valor con cada acción. Se creía que esta manera de tratar con la información de simulación premiaría las buenas acciones en vez de obscurecerlas, pero no fue el caso. En esta versión y en la quinta se eliminó la ganancia de *fitness* por la duración de los individuos para forzar a la red neuronal a solo ganar *fitness* por intentar seguir la línea correctamente.

Para aclarar a *NEAT* como tenía que actuar, se decidió restar de manear lineal los *steps*. Ahora, por cada paso de simulación, se resta o suma un valor constante en vez de un contador. Así, la red neuronal tiene más claro que acciones provocan un comportamiento positivo o negativo de manera más clara. Se puede observar en la figura 8 de que, aunque haya algo de comportamiento errático en el *fitness*, los individuos aprenden un comportamiento que, aunque erróneo, era mucho mejor que las otras poblaciones. Este compartimiento no era correcto ya que el mejor individuo seguía la línea, pero a una cierta distancia. El robot siempre estaba siempre desplazándose a la derecha de la línea como se observa en la figura 9.

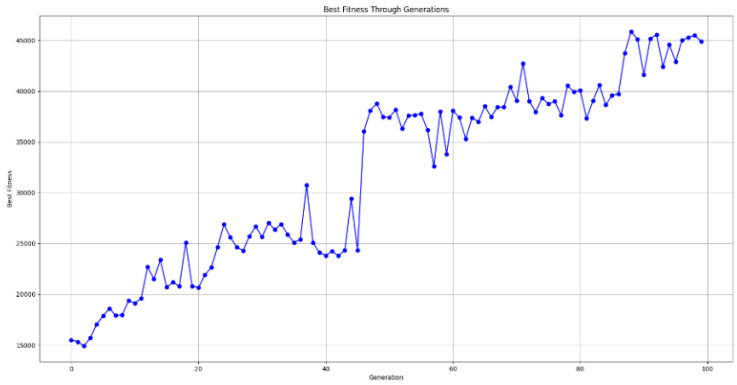

Figura 8: Aprendizaje positivo pero erróneo

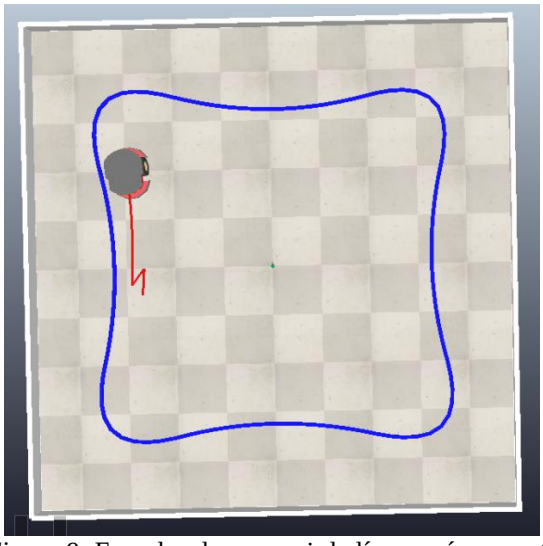

Figura 9: Error local por seguir la línea erróneamente

4.3.2.5 Quinta Versión: Fitness por 'line_offset'

Al realizar una investigación más a profundo en el diseño de métodos de *fitness* óptimos, se descubrió que *NEAT* prefiere trabajar con números enteros y con diferencias mínimas entre individuos. Dichos parámetros son suficientes para que el algoritmo aprenda a seguir líneas, pero con la ventaja de diferenciar mejor los individuos en base a su calidad. *NEAT* sabe que el individuo con valor 999 es peor que el individuo con *fitness* 1000, por lo que las diferencias entre comportamientos pueden ser mínimas.

Una ventaja de utilizar números enteros y diferencias de *fitness* mínimos es que permite llegar al mejor individuo antes. Este cambio permite que se invierta menos tiempo entrenando y por lo tanto detectar antes los fallos o comportamientos correctos. Otra ventaja que tiene diseñar el método de *fitness* así es que se puede categorizar comportamientos según la puntuación que tiene el individuo:

- *Fitness* menos de 0: El individuo tiene un comportamiento indeseado, como atascarse contra una pared, ir marcha atrás o no seguir la línea.
- *Fitness* entre 0 y 1199: El individuo ha empezado a moverse por la línea o por el mapa, obteniendo un comportamiento indeseado hacía el final que provoque la terminación prematura de la ejecución
- *Fitness* entre 1200 y 2399: Aunque contenga algo de comportamiento indeseado, el individuo está empezando a explorar ya que tiene al menos 1200 de *fitness* por duración de simulación. Tiene tramos que sigue correctamente la línea, pero hay partes que se pierde.
- *Fitness* entre 2400 y 2999: El individuo sigue la línea correctamente en la mayoría de la simulación. Puede haber momentos, especialmente en curvas, que pierda la línea un poco. El robot aun así consigue encontrarla, provocando que continúe a seguir la línea correctamente.
- *Fitness* entre 3000 y 3600: Los individuos siguen perfectamente la línea, ya que estará presente siempre en la cámara. Ya que el robot tiene que mantener la línea en el 5% interior de la imagen, es complicado que un individuo llegue a este rango de *fitness* en pocas generaciones.

En vez de tener un valor de alineación que multiplique los *steps* del individuo en la línea, se obtiene la calidad dependiendo del *offset* de la línea con respecto al centro de la cámara. Cuando la línea está en el centro de la cámara, el *offset* tiene un valor de 0, mientras que, si la línea está en el borde de la cámara, tendrá un valor máximo de 110. Se ha dividido la imagen en 5 franjas, que se reparten de dentro a fuera.

La franja de 0 a 22 es la central (11 píxeles a cada lado del centro de la cámara), mientras que el resto de las franjas eran de 22 a 44, de 44 a 66, de 66 a 88 y de 88 a 110. Dependiendo de la franja en que se encuentre la línea en ese *step*, el individuo recibía un *fitness*. De 0 a 22, la franja done la línea estaba más centrada, obtenía 5, mientras que la franja de 88 a 110, la menos alineada de todas, obtenía sólo un *fitness* de 1. Por ejemplo, en la figura 6 el offset tendría un valor de 24, obteniendo así una puntuación de 4 en ese *step*.

Dicho diseño de *fitness* consiguió que los *fitness* pasaran de magntiud 4-5 a magnitud 2-3, ayudando a que la red neuronal tuviese aún más claro cuál era el objetivo final del entrenamiento. Como se puede observar en la figura 10, los individuos aprendían rápido al principio, produciendo un valle a partir de la generación 28. Dicha respuesta local consistía en que el robot no avanzase por la línea, sino que simplemente se alineaba con la línea y se quedaba quieto sin avanzar.

Aun así, los problemas principales de la versión 4.3.2.4 se arreglaron. El enteramiento llegaba antes a la respuesta, aunque fuese la errónea, mientras que la puntuación del *fitness* era más fácil de entender; al ser números enteros sumados linealmente cada *step*, se podía calcular fácilmente cómo obtenía el individuo el valor de *fitness*. Finalmente, el error más importante fue arreglado; ahora el robot intenta desplazarse encima de la línea en vez de a una distancia a la derecha.

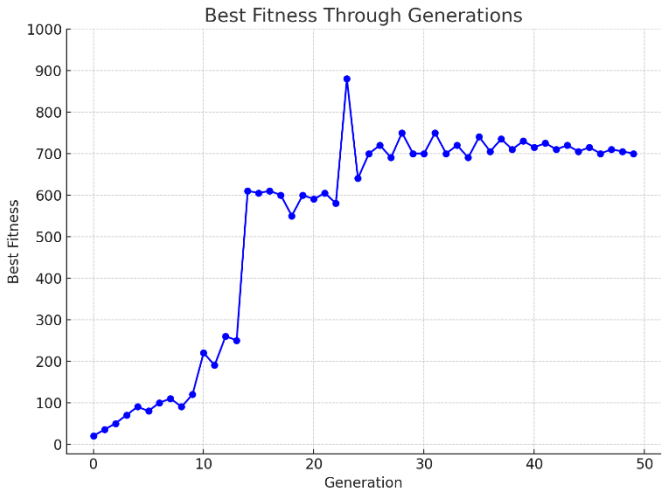

Figura 10: Fitness con Offset y no con Alineación

4.3.3. Diseños de Archivo de Configuración

El rendimiento de un algoritmo evolutivo como *NEAT* no depende únicamente del diseño del *fitness* o del procesamiento visual, sino también de los parámetros evolutivos. Estos parámetros definen el comportamiento genético y reproductivo de la población y están declarados en el archivo *neat_config*. La configuración del algoritmo es muy importante ya que regulan aspectos clave como tasas de mutación, tamaño de población, elitismo y selección por supervivencia. Estos aspectos cambian drásticamente el aprendizaje del individuo.

Durante el desarrollo del proyecto, se probaron distintas configuraciones con el objetivo de encontrar un balance entre diversidad genética, eficiencia evolutiva y estabilidad del aprendizaje. Algunas versiones favorecían una rápida exploración del espacio de soluciones, mientras que otras priorizaban la conservación de los mejores genomas para mejorar la precisión. Es importante recalcar que los parámetros hay que ajustarlos entre implementación e implementación del *fitness*. Esto se debe a que, por ejemplo, un alto elitismo en una implementación puede provocar estancamiento y en otro, ser el número ideal.

Aun así, hay que tener en cuenta de que algunas implementaciones son superiores a otras, independientemente del entrenamiento a que se apliquen. Un cierto nivel de mutación, un elitismo y *survival threshold* medio y una población alta tienden a ser las mejores opciones para la mayoría de implementaciones de *fitness.* Es buena práctica tener una configuración estándar y modificarla ligeramente entre implementaciones para conseguir un mejor resultado más rápido.

Ya que la configuración desempeña un papel tan grande en el entrenamiento de los individuos, se decidió encontrar la configuración estándar lo antes posible. La primera versión de configuración se utilizó en la primera versión de la función de cálculo de *fitness*. Antes de realizar la segunda implementación, se decidió realizar pruebas distintas con la configuración para ver como variaba el comportamiento del robot. A partir de la tercera versión del *fitness*, se utilizó la versión final de la configuración.

Como mucho, se cambiaron el tamaño de elitismo, *survival threshold* e individuos en una generación para comprobar los valores óptimos de dicha implementación. Hay que tener en cuenta que los cambios aplicados en una versión se hicieron sobre la versión anterior. Por ejemplo, los cambios realizados en la versión 4.3.3.3 se hicieron sobre la versión 4.3.3.2.

4.3.3.1. Primera Versión: Valores por Defecto

En la página oficial de *neat_python* indican la documentación del formato y contenido del archivo de configuración, indicando los valores que vienen por defecto. La configuración inicial tiene, entre otros, los siguientes parámetros:

- pop_size = 50
- elitism = 2
- survival_threshold = 0.2
- Tasas de mutación muy bajas

Esta versión presenta una buena base para empezar a generar casos específicos, ya que permitía observar el funcionamiento básico del algoritmo. Aun así, presentaba limitaciones importantes. La población pequeña reducía la diversidad genética y la falta de elitismo provocaba que los mejores genomas no se conservasen correctamente. La baja mortalidad de peores individuos reducía el número de reproductores con comportamientos correctos, mientras que la baja variación entre individuos era un problema muy grande por la baja mutación.

Estos ajustes generaban una evolución lenta y altamente volátil, con frecuentes pérdidas de buenos comportamientos. Esta primera versión de la configuración fue rápidamente modificada al comprobar que era demasiado básica. En las siguientes versiones, se fue modificando sección por sección los problemas indicados para poder solucionarlos.

4.3.3.2. Segunda Versión: Altos Niveles de Mutación

En esta versión se priorizó la exploración agresiva del espacio de soluciones, modificando los siguientes parámetros:

- weight_mutate_rate de 0.7 a 0.8
- bias_mutate_rate de 0.7 a 0.8
- Activation_mutation_rate de 0.0 a 0.2
- Aggregation_mutation_rate de 0.0 a 0.2

El objetivo era aumentar la tasa de innovación estructural y permitir la aparición rápida de nuevas conexiones, nodos y combinaciones. Esta configuración generaba una gran diversidad de soluciones, lo cual fue útil para etapas tempranas donde se buscaba romper simetrías y explorar comportamientos innovadores. Sin embargo, esta variabilidad también introducía inestabilidad evolutiva, con generaciones enteras sin progreso o incluso empeorando considerablemente el comportamiento ya aprendido. Además, al contar con el elitismo y *survival threshold* por defecto, muchas soluciones prometedoras se perdían tras una sola generación, empeorando aún más la situación

4.3.3.3 Tercera Versión: Poblaciones Grandes

En esta versión se experimentó con aumentar drásticamente el tamaño poblacional, cambiando los siguientes parámetros:

- *pop_size* de 50 a 200 individuos
- *elitism* de 2 a 20
- *survival_threshold* de 0.2 a 0.4

La lógica detrás de estos cambios era maximizar la diversidad, imponiendo una cierta presión selectiva. Esta configuración permitía observar cómo se distribuían naturalmente los comportamientos en una población grande. Al tener los niveles de mutación tan altas, se esperaba que cuadriplicando el tamaño de individuos en una generación (e incrementando el elitismo por 10 y duplicando el *survival threshold*) pudiese generar nuevas soluciones sin obscurecer el entrenamiento.

El resultado fue interesante desde el punto de vista exploratorio, ya que mostraba un crecimiento gradual sin perder mucha información. Aun así, esta implementación mostró varios problemas prácticos:

- **Muchas generaciones sin avance** debido a la ausencia de selección.
- **Reducción de presión evolutiva**, lo cual impedía fijar características útiles.
- **Alto costo computacional**, ya que cada evaluación implicaba 200 simulaciones por generación.

Esta configuración fue descartada por su bajo rendimiento relativo y alto tiempo de cómputo por generación.

*4.3.3.4 Cuarta Versión: Elitismo y **Survival_Threshold** elevado*

Esta variación de la implementación incorporó una combinación de preservación de élite y selección competitiva. Los ajustes cambiados fueron:

- *Elitism* de 20 a 10: Así, no se generaban respuestas locales con tanta facilidad
- *survival_threshold* de 0.4 a 0.3125: reproduciéndose solo el 31.25% mejor de cada especie.
- *pop_size* de 200 a 160: Mejor tiempo computacional, pero manteniendo alta diversidad.

Esta configuración logró el mejor balance observado**:**

- El elitismo aseguraba que los mejores genomas se mantuvieran activos sin forzar una respuesta local
- El umbral de supervivencia imponía una presión evolutiva clara**,** reduciendo estancamientos y permitiendo aun así suficiente variación.
- La gran población permitía mantener la exploración sin perder los mejores resultados.

Además, se observaron mejores resultados en menor cantidad de generaciones, y una notable estabilidad en las especies dominantes. Dichas especies lograban progresar consistentemente hacia mejores soluciones, aunque fuesen las erróneas en el momento de realizar las pruebas con esta configuración.

4.3.4. Estructura Final de Fitness, Detección de Imágenes y Configuración

Después de múltiples iteraciones, ajustes y pruebas, se definió una estructura final para los tres componentes. Esta estructura representa la combinación más eficiente entre precisión, estabilidad evolutiva y tiempo de entrenamiento. Esta estructura fue la utilizada para obtener los resultados finales del proyecto.

4.3.4.1 Estructura Final de Fitness

La versión final del *fitness* es una evolución directa de la quinta versión, pero con los siguientes mejoras y cambios:

- **Puntuación por tiempo**: Se introdujo de nuevo la bonificación adicional por la duración que el robot permanecía activo en el mapa. Así se incentivan recorridos prolongados sin comportamientos incorrectos. Este *fitness* se añade al final de la simulación en vez de por cada *step*.
- **Penalización por poca velocidad**: Se penaliza al robot si avanzaba con una velocidad promedio demasiado baja. También se acaba la simulación en el caso de que dure más de 4 segundos.
- **Simplificación de la comprobación del** *line_offset*: Se redujeron los puntos de muestreo en la imagen para verificar el alineamiento de la línea. Originalmente se analizaban cinco regiones horizontales (22, 44, 66, 88, 110), pero se redujo a dos (55 y 110). Si la línea está en la franja 0-55, obtiene 2 de *fitness* en ese *step*. Si está en la franja 56-110, obtiene solo 1.
- **Penalización por giros bruscos**: Se modificó la penalización directa por giros excesivos, manteniendo el criterio de corte temprano. Un individuo se ve penalizado por giros bruscos solo si no está en la línea, permitiendo filtrar individuos no viables sin castigar movimientos válidos en trayectorias complejas. Aun así, si el robot gira sobre sí mismo durante más de 4 segundos, el entrenamiento sigue parándose

Dichos cambios se han realizado por varios motivos. Se observó dos respuestas locales en las pruebas de la quinta versión del *fitness.* En el primer comportamiento, los individuos se quedaban quietos mirando a la línea en vez de avanzar por ella, mientras que el segundo tenía problemas graves al realizar curvas más bruscas. La estructura del *fitness* final se puede observar en la figura 11.

```python
def calculate_fitness(avg_speed, line_offset, on_line, stuck, turn_amount):
    fitness = 0
    abs_line_offset = abs(line_offset)

    # Follow line correctly
    if on_line and avg_speed > 0.2:
        if(abs_line_offset < 56):
            fitness += 2
        else:
            fitness += 1

    # Penalize wondering
    if not on_line:
        fitness -= 4
        if(turn_amount > 1.5):
            fitness -= 4

    # Obstacle avoidance
    if stuck:
        fitness -= 2

    # Moving backwards or no movement
    if avg_speed <= 0:
        fitness -= 2
    # Barely positive speed
    elif abs(avg_speed) <= 0.2:
        fitness -= 2

    return fitness
```

Figura 11: Estructura Final de *Fitness*

Al penalizar los individuos por poca velocidad, fuerza a la red neuronal a aprender que los individuos tienen que tener al menos una velocidad mínima positiva. Este comportamiento se refuerza aún más por el tiempo de ejecución, ya que al cortarse los individuos a los 4 segundos si están quietos, retroceden o van lentos, la función de *fitness* no aporta una mayor puntuación. Con estos dos cambios, el entrenamiento siempre genera individuos que tengan una velocidad positiva.

Al simplificar la comprobación del *offset* y modificar la penalización por giros bruscos, provoca que el robot aprenda más rápido a seguir la línea. El primer motivo es que el sistema de *fitness*, al tener un sistema tan complejo de entrenamiento, generaba valles u objetivos locales. Por ejemplo, hubo un entrenamiento el cual el robot prefería seguir una línea recta y en vez de girar correctamente en la curva, daba un giro de 180 grados y volvía a seguir la línea en sentido opuesto.

El segundo motivo es que el robot se veía penalizado activamente por realizar giros bruscos si estaba siguiendo la línea. Al ser interpretado por la red neuronal que el robot no podía realizar giros fuertes en ningún momento, los individuos aprendían que era preferible perder la línea a intentar corregirse. Al solo penalizar los giros bruscos si no está el individuo en la línea, permite a la red neuronal corregirse para poder seguir la línea mejor. Por último, ya que es menos compleja la detección del *offset*, la red neuronal tarda menos en aprender el comportamiento correcto.

En conjunto, estas mejoras presentan un balance positivo entre exploración y eficiencia, promoviendo comportamientos más útiles para la navegación real. Ahora el robot sigue la línea correctamente con velocidad positiva en un tiempo de entrenamiento razonable.

4.3.4.2 Estructura Final de Detección de Imágenes

La detección visual también alcanzó una versión final más óptima. Esta estructura toma como base la tercera versión, manteniendo su enfoque, pero con detalles más refinados. Se captura una única región inferior de la imagen para evaluar la posición horizontal del centro de masa de la línea. Esta posición se compara con el centro de la imagen en vez de con el centro del robot, produciendo el *line_offset*, el valor que indica qué tan alineado está el robot con la línea. Ahora también produce un segundo parámetro, que simplemente indica si el robot está encima o no de la línea.

Estos dos parámetros se usan directamente como entradas a la red neuronal, también alimentando el cálculo del *fitness*. El procesamiento fue simplificado para reducir el tiempo de simulación, ofreciendo estabilidad y rapidez. Así, se permite una evaluación clara y continua sin procesar múltiples capas de imagen o contornos.

4.3.4.3 Estructura Final de Configuración

La configuración final de *NEAT* refleja el aprendizaje obtenido en todas las versiones anteriores, estableciendo una estructura robusta que producen resultados predecibles:

- **Población grande y estable**: con valor de 150, favorece la diversidad genética sin comprometer la velocidad de entrenamiento.
- **Preservación de élite** con valor de 8 asegura que los mejores individuos sobrevivan entre generaciones sin generar respuestas locales.
- **Selección exigente**: *survival_threshold* con valor de 0.28 o 28%
- Tasas de mutación estructural y de pesos reducidas ligeramente. Se ha establecido *bias_mutate* de 0.8 a 0.7, *Aggregation_mutation_rate* de 0.2 a 0.0 y *Activation_mutation_rate* de 0.2 a 0.1.
- **Compatibilidad y especiación** controladas: *compatibility_threshold* ha subido de 3.0 a 3.5

5. Resultados y conclusiones

Esta sección recoge los resultados obtenidos durante el desarrollo y entrenamiento del controlador, así como las conclusiones extraídas del análisis del comportamiento evolutivo. Se presentan los distintos escenarios de prueba, las métricas utilizadas, el progreso del proceso evolutivo, y un estudio detallado de las conductas emergentes. También se reflexiona sobre las fortalezas del sistema, sus limitaciones, y la posibilidad de reutilizar genotipos entrenados en nuevos entornos. Finalmente, se trata la extrapolación del mejor genotipo a distintos casos más complejos que simplemente seguir una línea.

5.1. Escenarios de Prueba y Efectividad de Controlador

Para validar la robustez del controlador, se han utilizado dos mapas diferentes. El primero, el más simple, representa una línea circular, ideal para pruebas de comportamiento básico y comprobación de alineación. Dicho circuito destaca por sus curvas y rectas idénticas; da igual en que parte del circuito se encuentre el robot que siempre va a encontrarse con los mismos tramos. El segundo circuito, el más complejo, es de tipo serpenteante, con múltiples curvas y secciones más exigentes. Este circuito destaca por un circuito ligeramente más largo, que tiene curvas de distintos ángulos internos y rectas de distintas longitudes.

Ambos escenarios fueron creados para cubrir distintos comportamientos y obligar al robot a aprender tanto el seguimiento básico como adaptarse a trayectorias variables. Además, se emplearon estos circuitos para observar el comportamiento del mejor genotipo frente a nuevas condiciones sin y con reentrenamiento. Se realizó el entrenamiento inicial del controlador en el primer circuito, obteniendo el *fitness máximo* de 3600, terminándose el entrenamiento antes de la generación máximo de 150.

Se puede observar que la trayectoria del mejor genotipo en la figura 12 no sigue perfectamente la línea azul. Esta discrepancia de trayectoria se debe a que la cámara está colocada en la parte delantera superior del robot y no entre las ruedas, por lo que el centro del robot no coincide plenamente con el eje de simetría. Por esto, se provoca un ligero desajuste entre la trayectoria del robot con respecto a la línea que no afecta en absoluto la capacidad del robot en seguir la línea.

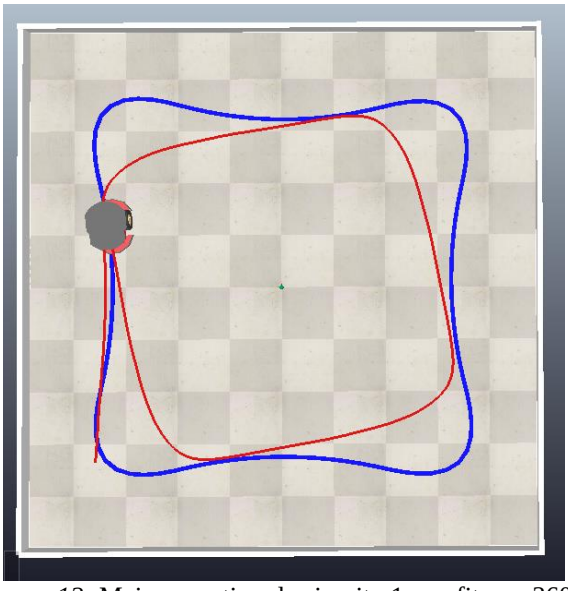

Figura 12: Mejor genotipo de circuito 1 con fitness 3600.

Para comprobar la robustez del controlador, se probó el mejor genotipo del primer circuito en el segundo. Siendo la primera vez que el genotipo experimenta el segundo circuito, el robot no demostró el comportamiento adecuado. Como se puede observar en la figura 13, el robot no consiguió si quiera seguir la línea, quedándose atascado el robot en la pared trasera.

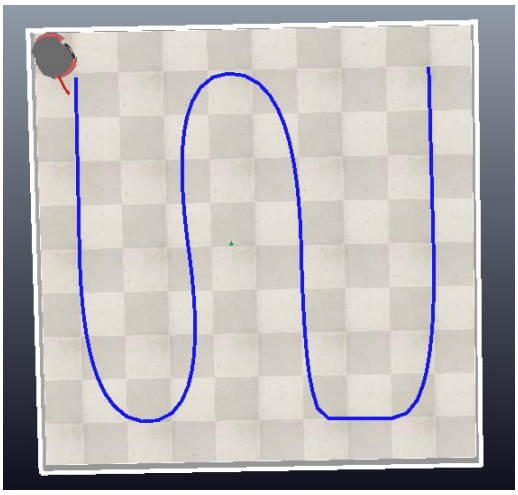

Figura 13: Mejor genotipo de circuito 1 en el circuito 2

Este comportamiento es perfectamente normal. Al ser un escenario distinto, el controlador no sabe cómo iniciar correctamente los parámetros de la red neuronal, provocando un comportamiento no deseado. Para comprobar la capacidad de reentrenamiento de *NEAT*, se decidió extender el aprendizaje del mejor genotipo del primer circuito en el segundo. El reentrenamiento obtuvo una puntuación de mejor genotipo en la primera generación de 2105, incrementando la calidad del genotipo hasta 2861 en la generación 37, donde no se obtuvo ninguna mejora en el *fitness* durante más de 30 generaciones. El controlador por lo tanto había encontrado una respuesta local, que se puede observar en la figura 14. Obviamente el comportamiento es erróneo, ya que no consigue correctamente seguir la línea a partir del segundo giro.

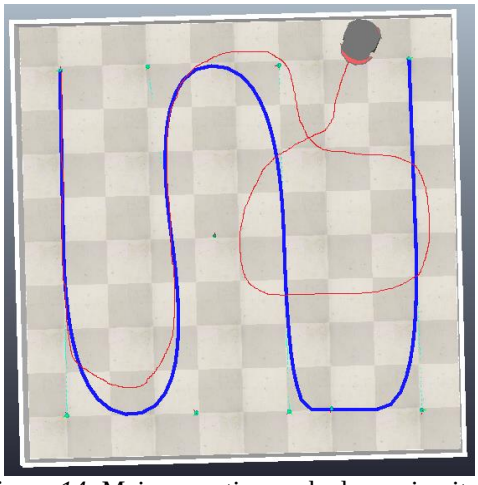

Figura 14: Mejor genotipo probado en circuito 2

La respuesta local obtenida en el reentrenamiento se debe al *overfitting*. El sobreajuste de la red neuronal se debe a la variación de las curvas con respecto a lo que ya estaba entrada la red neuronal. Como se ha descrito anteriormente, las curvas en el primer circuito son iguales y con poca curvatura, por lo que el controlador no ha tenido que aprender a realizar distintos tipos de giros. Al tener tres tipos de curvas distintas, con direcciones y ángulos internos distintos, la red neuronal no ha tenido la capacidad suficiente de reajustar los nodos correspondientes a la decisión de giros. Por lo tanto, *NEAT* no ha conseguido adaptar el controlador para que el robot consiga completar el circuito.

Para comprobar si empezando el entrenamiento desde 0 se conseguía generar el comportamiento deseado, se reinició el entrenamiento sin ningún tipo previo de conocimiento. Este segundo entrenamiento en el circuito mejoró mucho la situación, ya que al final se consiguió que el mejor genotipo terminase el circuito sin ajustar ningún parámetro. El individuo obtuvo un *fitness* de 3307, siendo el comportamiento el de la figura 15.

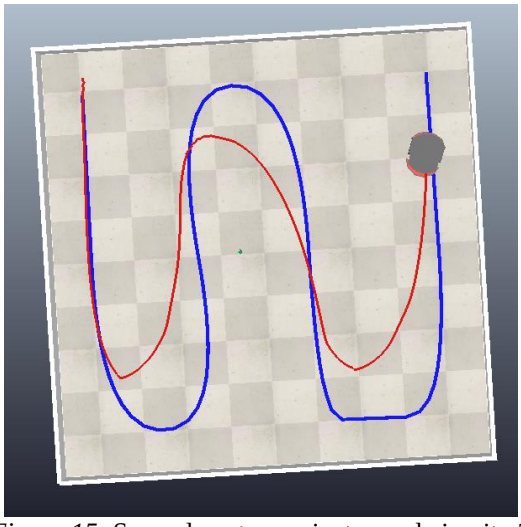

Figura 15: Segundo entrenamiento en el circuito 2

A partir de la generación 97, el controlador no mejoró la calidad durante más de 30 generaciones. Se concluyó que, al no haber mejora durante un periodo tan largo, el controlador no tendría capacidad de mejora en la calidad de los individuos. Por lo tanto, el *fitness* máximo que iba a obtener el controlador sería el del mejor genotipo de la generación 97. El *fitness* obtenido en el entrenamiento es menor que el máximo obtenible de 3600, debiéndose a que no era capaz el mejor individuo de mantener centrada la línea.

El hecho de que el controlador no pueda obtener la puntuación máxima procede del propio circuito y no de la implementación de la red neuronal. Al tener giros tan complicados, la línea es imposible mantenerla siempre centrada en la cámara, provocando que se obtenga un *fitness* inferior al esperado. Aun así, teniendo en cuenta la complejidad del circuito, es buena señal que el robot haya conseguido seguir la línea en todo momento con un *fitness* superior a 3300. Este entrenamiento demuestra que el controlador es apto para seguir líneas incluso en escenas no idílicas donde no se puede obtener un *fitness* teóricamente perfecto.

Aunque el reentrenamiento del primer mejor genotipo en el segundo circuito debería haber tenido los mismos resultados que el entrenamiento desde cero, se puede observar que no fue el caso. Como ya se ha mencionado previamente, la razón principal es el *over fitting*, ya que, al ser entrenada durante tantas generaciones en un circuito más fácil, por mucho entrenamiento que se realice en un escenario más complicado no se conseguirá que aprendan nuevos comportamientos.

El sobre ajuste se puede remediar de varias maneras. La primera es forzando que el entrenamiento dure menos y por lo tanto que la red neuronal no sea tan rígida, teniendo una mayor capacidad de aprendizaje. La segunda es implementando una función de *fitness* menos restrictiva o premiando aún más los comportamientos positivos, permitiendo también que tenga una mayor capacidad de reacción y de cambio ante situaciones nuevas.

5.2. Métricas de Rendimiento

La evaluación del sistema de entrenamiento se ha llevado a cabo mediante el análisis de métricas extraídas a través del módulo auxiliar *treatment.py*. Las métricas obtenidas permiten no solo cuantificar el rendimiento de los individuos, sino también comprender la dinámica evolutiva, identificar posibles cuellos de botella y validar la eficacia de la configuración empleada. Las principales métricas utilizadas son las siguientes:

- **Fitness promedio por generación**: Mide la media de aptitud de los individuos de una generación. Es útil para evaluar el progreso global de la población y detectar si existe una tendencia de mejora continua, estabilización o decrecimiento. Esta métrica también ayuda a ajustar parámetros como la tasa de mutación o el tamaño poblacional, ya que una población estancada puede indicar una falta de exploración adecuada del espacio de soluciones.
- **Fitness máximos y mínimos**: Representan, respectivamente, el mejor y peor individuo de los genotipos de elitismo. El valor máximo es especialmente importante, ya que *NEAT* no busca optimizar la media sino encontrar individuos excepcionales. El comportamiento del mejor fitness a lo largo del tiempo permite verificar si la evolución está progresando hacia soluciones más óptimas, o si ha alcanzado un estancamiento local. Por otro lado, el fitness mínimo permite detectar si existen individuos cuya calidad se mantiene excesivamente baja, lo cual podría indicar un mal ajuste de la presión selectiva o una supervivencia excesiva de genotipos poco prometedores.
- **Duración media por individuo**: Aunque no está directamente relacionada con el *fitness*, el análisis del tiempo de simulación promedio por individuo revela información crítica sobre la robustez de los controladores. Por ejemplo, un aumento sostenido en la duración media puede indicar que los individuos están sobreviviendo más tiempo sin quedarse bloqueados, lo que es señal de una mayor adaptabilidad. Esta métrica se calcula indirectamente a partir del número de *steps* antes de que se active una condición de parada (por giro excesivo, parada prolongada o colisión).
- **Historial de especies**: A través del sistema de especiación incorporado en *NEAT*, cada individuo pertenece a una especie que agrupa topologías similares. Se registran la cantidad de especies activas por generación, su media de fitness y su evolución temporal. Esta información permite observar el surgimiento, persistencia o extinción de especies, lo cual es fundamental para preservar la diversidad genética del sistema. Un número muy bajo de especies puede ser síntoma de una convergencia temprana, mientras que un exceso puede dificultar la consolidación de avances.

Estas métricas no solo sirven para evaluar el rendimiento funcional del controlador, sino también como una guía para ajustar la configuración del sistema evolutivo. Por ejemplo, si durante más de 30 generaciones los mejores individuos no superan un umbral de *fitness* predefinido (por ejemplo, 2400 puntos, siendo el umbral de éxito 3600), es indicativo de que la función de fitness, la estructura de entrada, o los parámetros de *NEAT* deben ser revisados. Una ausencia de mejora sostenida en los mejores individuos puede deberse a una falta de presión selectiva, un elitismo insuficiente, o una configuración inadecuada del *compatibility threshold* que impida la formación de especies estables.

Aun así, hay que tener en cuenta que si el umbral predefinido de *fitness* es muy cercano al máximo (en este caso, un *fitness* superior a 3300), el problema seguramente sea la complejidad de la escena. En algunas ocasiones, si la escena contiene trayectorias complicadas, es imposible obtener el umbral máximo predefinido. Estas implementaciones pueden ser las mejores soluciones del objetivo principal sin tener el comportamiento teórico idílico.

Aunque el foco principal del análisis suele situarse en los mejores fitness de cada generación (ya que son los individuos que se centra *NEAT* para resolver el problema), tanto la media como el peor fitness ofrecen información crítica para valorar el estado evolutivo de la población. Por ejemplo:

- **Fitness mínimo creciente**: Indica que los peores individuos están mejorando generación tras generación, lo que sugiere que el algoritmo no solo está aprendiendo en los extremos, sino en todo el espectro poblacional.
- **Media estable con mejor fitness creciente**: Señala que las especies están progresando positivamente.
- **Media creciente con desviación estándar baja**: Sugiere una evolución estable y controlada, ideal para fases finales del entrenamiento donde se busca consolidar un comportamiento eficaz.

En conjunto, estas métricas proporcionan una visión integral del entrenamiento evolutivo, permitiendo decisiones informadas sobre cuándo intervenir, modificar la configuración o reiniciar el proceso.

5.3. Análisis Comportamiento

Tras varias versiones del *fitness*, detección de imágenes y configuración *NEAT*, se obtuvo un controlador capaz de generar dos genotipos para recorrer con éxito los escenarios. La simulación del primer escenario duró algo más de seis días. El reentreno del mejor genotipo en el segundo circuito duró un poco más de 2 días, mientras que el entrenamiento desde 0 en el segundo circuito duró 7 días. El primer controlador consiguió el *fitness* máximo de 3600, no pudiendo completar correctamente el segundo circuito, independientemente de si se reentrenaba o no. El otro controlador sí que consiguió recorrer todo el circuito 2, aunque con un *fitness* de 3307. El segundo controlador también fue capaz de recorrer perfectamente el primer circuito con la misma facilidad que el primer controlador.

5.3.1. Comportamiento en Circuito 1 y 2

Los comportamientos generados por la evolución han sido siempre analizados de manera visual. Aunque los datos estuviesen escritos en el archivo *robot info*, con más de 10 sectores de información por generación, es muy complicado estudiar el patrón completo del entrenamiento. Por lo tanto, generar gráficos con los datos relevantes ha sido un aspecto clave para el análisis del comportamiento.

En la figura 16 se observa el gráfico de los mejores individuos de cada generación durante el entrenamiento del primer escenario. Se puede observar una tendencia positiva, terminando en 3600 de *fitness* en la generación 91, con un comportamiento poco errático. Existen 5 zonas importantes a destacar en este gráfico, cada una mostrando un comportamiento distinto:

- **Generación 0 a 16:** Los individuos aprenden que comportamientos son erróneos y cuales son correctos. Aquí, el controlador realiza distintas pruebas con velocidades hasta que aprende que los individuos que tienen una velocidad positiva y no los que giran sobre sí mismos, tienen una velocidad negativa o se quedan quietos son los individuos correctos.
- **Generación 17 a 39:** Los individuos aprenden a seguir el tramo inicial de recta. Muchos de ellos se estrellan contra la primera pared, no sabiendo que hacer exactamente con el obstáculo.
- **Generación 40 a 60:** En esta zona, el controlador aprende a evadir obstáculos y a empezar a girar correctamente en la primera curva, aunque no correctamente.
- **Generación 61 a 73**: donde el controlador ya no produce casi ningún comportamiento erróneo. Aquí, el controlador empieza a seguir correctamente la línea con poca alineación, pero presenta un comportamiento correcto
- **Generación 74 a 91:** No hay presente ningún individuo con comportamientos erróneos, siendo el aprendizaje principal alinearse correctamente con la línea. La última generación obtiene un individuo con el *fitness* máximo de 3600.

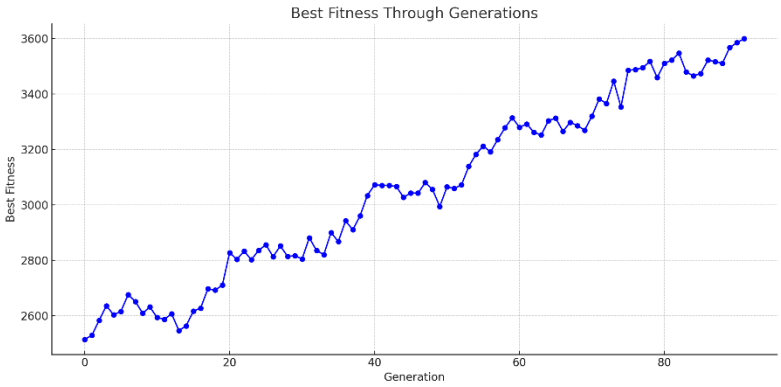

Figura 16: Mejores individuos del entrenamiento con circuito 1

Los peores individuos, representados en la figura 17, indican un crecimiento lineal y positivo, demostrando que los parámetros de presión selectiva son correctos. Si hubiesen sido inferiores, se habrían reproducido los individuos con peores comportamientos, reduciendo drásticamente el crecimiento. Si hubiesen sido mayores, se habría generado una respuesta local, estancando la evolución.

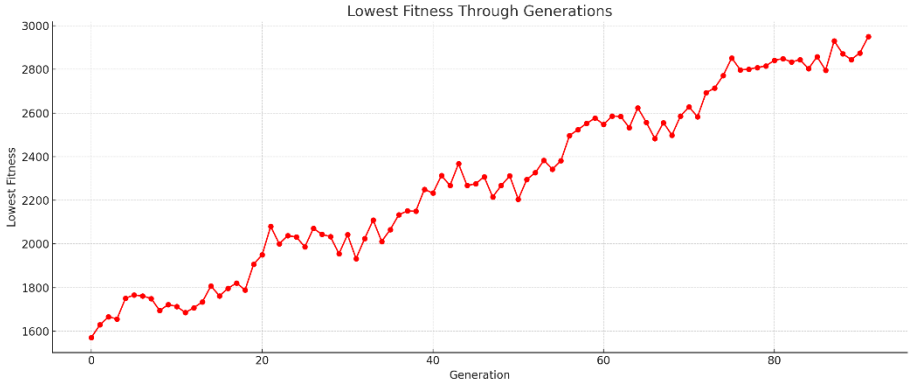

Figura 17: Peores individuos del entrenamiento con circuito 1

De la misma manera que los peores individuos, la media de *fitness* en la figura 18 indica un crecimiento muy prometedor; no hay ni estancamientos notables ni problemas de aprendizaje. Esto demuestra un balance de población, mutación y elitismo correcto para el controlador. Gracias a la media, se puede observar una desviación típica media, afianzando aún más que la configuración de *NEAT* para este controlador eran los correctos.

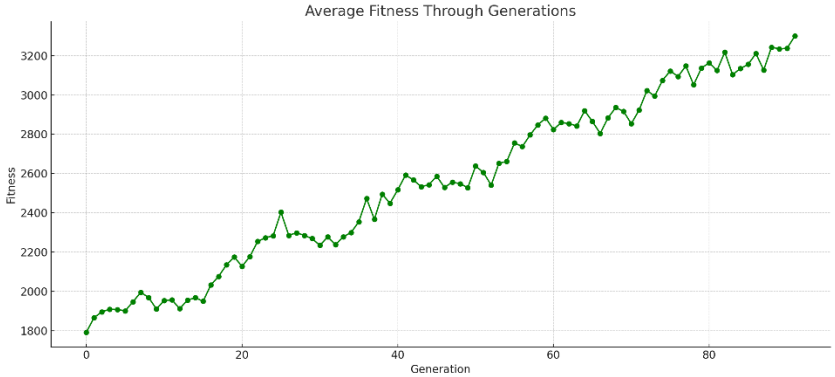

Figura 18: Media de los individuos del entrenamiento con circuito 1

A lo largo del proceso se han identificado distintas fases de exploración, especialización y mejora progresiva. Estas zonas están presentes solo si se realiza el entrenamiento con el primer circuito, ya que, aunque se presenten 5 zonas distintas también en el segundo, no presentan los mismos comportamientos. Como se verá a continuación, ya que el segundo circuito es más complejo, la red neuronal obtuvo otro enfoque de aprendizaje distinto al primer circuito.

El entrenamiento en el segundo circuito fue ligeramente más largo que en el primero; en vez de 91 generaciones para obtener el mejor genotipo, se tardaron 97 generaciones. En la figura 19 se muestran los mejores genotipos del segundo entrenamiento. Se puede observar que, comparado con los mejores genotipos del primer entrenamiento, este es ligeramente más errático. Por ejemplo, las zonas de generaciones 16 a 22 y 60 a 80 muestran picos y bajadas no tan prominentes ni durante tanto tiempo como el entrenamiento del primer circuito.

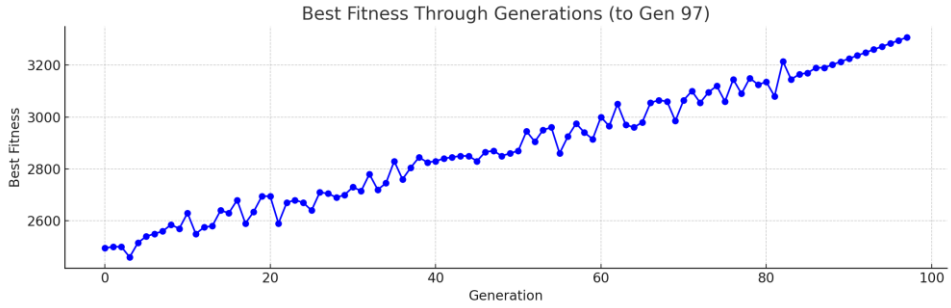

Figura 19: Mejores individuos del entrenamiento del circuito 2

En este entrenamiento se observan también cinco zonas de comportamiento, aunque con un aprendizaje ligeramente distinto al del circuito 1. Las zonas indicadas son las siguientes:

- **Generación 0 a 21:** Como en el circuito 1, los individuos aprenden que comportamientos son erróneos y cuales son correctos. El controlador realiza distintas pruebas con velocidades hasta que aprende que individuos tienen comportamiento erróneo y correcto
- **Generación 22 a 38:** Los individuos aprenden a seguir el tramo inicial de recta, produciendo un fallo en el comportamiento. En vez de estrellarse contra la pared como en el entrenamiento previo, los genotipos aprendieron a dar una vuelta de 180 grados y seguir avanzando por el mismo tramo de línea por el que habían venido.
- **Generación 40 a 50:** En esta zona, el controlador aprende a girar en la primera curva, lentamente desaprendiendo el comportamiento generado en la última sección.
- **Generación 51 a 82**: En esta sección, el controlador ya no produce ningún comportamiento erróneo. El controlador empieza a seguir correctamente la línea con poca alineación, aprendiendo a tomar correctamente las tres líneas de la escena.
- **Generación 83 a 97:** En esta sección, el aprendizaje se centra en alinearse correctamente con la línea. El robot ya tiene un comportamiento correcto en las líneas, centrándose en tomar las curvas correctamente. La última generación obtiene un individuo con el *fitness* máximo de 3307.

La figura 20 indica los peores individuos del entrenamiento del circuito 2. Este gráfico muestra algunos aspectos que no son obvios en la figura 19 y que son importantes de recalcar. Aunque el *survival threshold* sea correcto, se generan demasiados picos y valles en algunas zonas que podrían haberse evitado y por tanto haberse generado el genotipo antes. Por ejemplo, las zonas de la generación 20 a la 35 y de la generación 60 a 71 podrían haberse evitado si el cálculo de *fitness* fuese aún más claro con el objetivo.

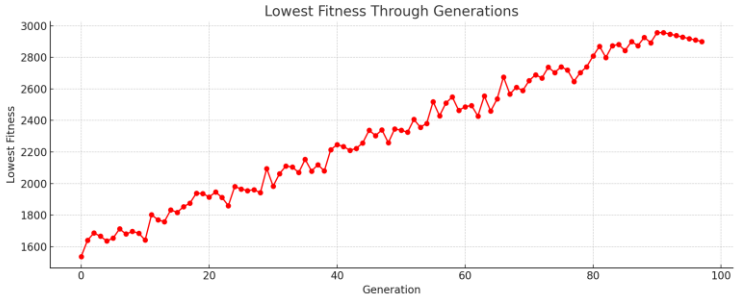

Figura 20: Peores individuos del entrenamiento del circuito 2

La desviación estándar es ligeramente mayor y más volátil que en el entrenamiento del primer circuito. Si se compara la información presente en la figura 21, que indica la media de *fitness* de los genotipos, con la información de la figura 20, es claro ver que el crecimiento no ha sido ni tan gradual como en el primer entrenamiento. Esto se puede explicar por el salto de complejidad, provocando que los genotipos de las generaciones tuviesen que explorar más soluciones y por lo tanto generando individuos más dispares.

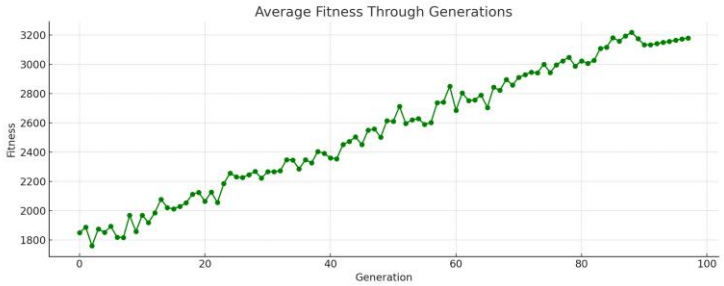

Figura 21: Media de los individuos del entrenamiento del circuito 2

Se puede observar que, como en el primer entrenamiento, hay cinco tipos de individuos dependiendo del comportamiento que tienen. Entre sesión y sesión de entrenamiento, habrá ligeras variaciones en cuando ocurren estos cambios en las generaciones por culpa de la mutación. Las mismas etapas se van a ver en todo momento si se ejecuta el entrenamiento tanto en el primer como en el segundo circuito.

Al tener dos genotipos ganadores, uno por cada circuito, se pueden comparar el comportamiento entre ellos para comprobar cuál es superior. Como se ha dicho ya previamente, el mejor genotipo del primer circuito no sigue correctamente la línea en el segundo circuito. Se probó el mejor genotipo del segundo circuito en el primero, observando que realiza sin ningún problema el primer circuito, demostrando que puede desplazarse correctamente por ambos circuitos. Por lo tanto, el segundo genotipo es superior al primero al ser más capaz de seguir las líneas en más circuitos.

5.3.2. Fortalezas y Limitaciones del Controlador

Entre las fortalezas destacan:

- Capacidad de alineación precisa con la línea.
- Velocidad positiva sostenida.
- Reacción eficiente ante obstáculos estáticos cercanos.
- Capacidad de aprendizaje con mapas más complejas.

Como limitaciones, se detectaron:

- Tiempo oscilante de entrenamiento; al depender de mutaciones puede tardar mucho en encontrar el genotipo ganador.
- Comportamiento oscilante en zonas con casos muy especiales o con líneas complejas no tratados en función de *fitness*.
- Necesidad de reentrenar desde el principio ya que la arquitectura generada en el primer circuito es demasiado rígida para poder ser reutilizada en el segundo circuito.

5.3.3. Generalización a Nuevos Mapas

Se ha probado el controlador en escenas no planteadas en el objetivo principal del *TFG*. En el primer caso, se realizó una escena con caminos que se cruzan, presente en la figura 22. En el segundo, se generó otra escena en la cual la línea se bifurca, pasando a veces el robot por caminos estrechos, como la figura 23. Todas las pruebas con los circuitos nuevos se han realizado con el genotipo entrenado del circuito 2, ya que se considera el mejor individuo de los dos.

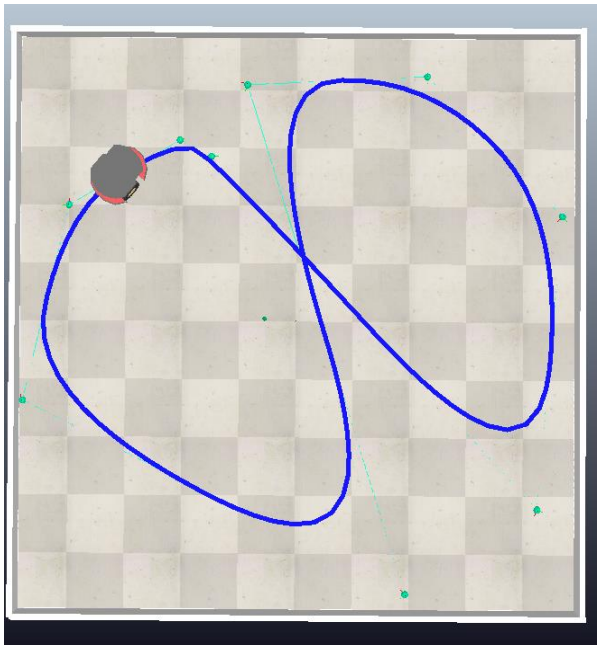

Figura 22: Circuito con caminos que se cruzan

En el caso de la escena con cruces, el resultado fue mixto. Ya que la percepción de imágenes está diseñada para detectar contornos en el 5% inferior de la imagen, y por lo tanto líneas con el mismo sentido del movimiento del robot, no está diseñado para que tome decisiones con cruces. El método de procesamiento de imágenes no sabe qué hacer cuando el robot se topa con una línea perpendicular, por lo que ocurren dos casos distintos:

- En la mayoría de casos, el robot simplemente sigue hacia delante, sin tomar ninguno de los caminos del cruce.
- En el resto de casos, el robot toma el camino de la izquierda, ignorando el camino hacia delante y el de la derecha del cruce.

Se tiene una hipótesis de porque tiene el robot este comportamiento. Al llenar el 5% de la imagen inferior al toparse con el cruce, el robot interpreta que la línea en realidad está en la franja de *offset* de menos de 55, por lo que está alineado y sigue hacia delante. Ya que el mejor genotipo utilizado ha sido el del circuito 2, dos de los giros son a la izquierda mientras que solo hay uno hacia la derecha. Como la línea también está también presente en el offset de -55 a -110, el controlador a veces toma la decisión de que la línea está en el extremo izquierdo (porque ha sido entrenado con más curvas a la izquierda) por lo que gira en ese sentido.

Este comportamiento se solucionaría con facilidad cambiando dos cosas importantes del funcionamiento actual de la versión de generación de *fitness*. Primero, se añadiría un booleano extra en la función de detección de imágenes. Dicho booleano sería falso en todos los casos excepto cuando detecte el cruce. En la función de cálculo de *fitness*, se añadiría un caso extra, activándose solo cuando el booleano devuelto por la detección de imágenes sea verdadero. Dependiendo de la funcionalidad que se le quiera añadir, el controlador daría puntuación por seguir hacia adelante, tomar el camino de la izquierda o el de la derecha. Con reentrenamiento y estos dos cambios, el controlador tomaría siempre el mismo camino en vez de tomar la decisión de manera aleatoria.

.

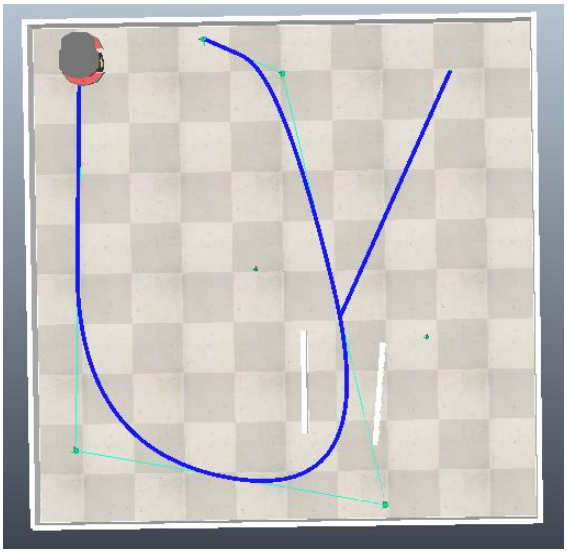

Figura 23: Circuito con bifurcaciones y que pasan por sitios estrechos

En el caso de la figura 23, el robot siempre toma el camino de la izquierda en las bifurcaciones. Si se quisiese cambiar dicho comportamiento, se debería cambiar el sistema de compensación del *fitness*. Ya que el valor del *offset* es positivo si la línea está a la derecha (yendo de 0 a 110) y negativo a la izquierda (yendo de -110 a 0), con simplemente premiar a los individuos que tomen el offset positivo es suficiente

El problema con la escena de la figura 23 son los caminos estrechos. El robot a veces se queda quieto en medio del pasillo, ya que los sensores detectan las paredes a los lados y no sabe qué hacer. Una posible solución para cambiar este comportamiento es reducir las distancias de los *readings* de los sonar de 0.1 a 0.05. Así, los sensores no se activarían y el robot pasaría por el pasillo sin problemas.

Los resultados fueron positivos en estructuras similares (circuitos cerrados), pero menos eficientes en entornos con bifurcaciones, caminos estrechos y cruces. Esto indica que, si bien el controlador actual es sólido, la generalización aún depende del grado de similitud entre escenarios. Si se añaden los cambios descritos anteriormente, el comportamiento del robot sería más fiable.

No obstante, el hecho de que el mismo genotipo pueda seguir líneas en mapas distintos sin necesidad de reentrenamiento es una señal clara de éxito evolutivo y robustez del mejor genotipo obtenido. Estos casos extras han demostrado que el robot, con el entrenamiento actual, sabría tomar decisiones incluso en situaciones no planteadas en el objetivo principal. También demuestra que la implementación actual es lo suficientemente modular que modificando o añadiendo ciertos parámetros se puede cambiar el comportamiento deseado con facilidad.

5.5. Lecciones Aprendidas

Este Trabajo de Fin de Grado ha demostrado que es posible diseñar y entrenar un controlador autónoma robusto y eficaz utilizando técnicas de inteligencia artificial evolutiva. Concretamente, se ha validado la capacidad del algoritmo *NEAT* para resolver tareas de navegación como el seguimiento de líneas, incluso en entornos simulados con obstáculos y trayectorias no triviales. Se ha hecho un especial énfasis en la optimización del proceso de entrenamiento, lo que ha permitido reducir significativamente el coste computacional sin comprometer la calidad de aprendizaje. Esta eficiencia se ha logrado gracias a técnicas como la detención anticipada de individuos ineficaces, la desactivación del componente gráfico y la automatización del flujo de simulaciones.

Una de las lecciones más destacadas ha sido la importancia de iterar continuamente en el diseño de la función de *fitness*. A lo largo del proyecto, se han probado múltiples versiones, cada una con mejoras orientadas a evitar comportamientos no deseados y facilitar la convergencia hacia soluciones útiles. Se ha podido comprobar como una función de evaluación bien ajustada puede ser determinante en la estabilidad del aprendizaje.

Asimismo, se ha reafirmado la relevancia de acompañar el entrenamiento evolutivo con herramientas de análisis cuantitativo. La implementación de *treatment.py* ha permitido visualizar gráficamente la evolución del *fitness*, detectar patrones de estancamiento y comprender la dinámica de las especies. Esta capacidad analítica ha sido clave para la tomar decisiones fundamentadas, tanto en la configuración de parámetros como la interpretación de resultados obtenidos.

El proyecto ha permitido consolidar conocimientos fundamentales en áreas como la inteligencia artificial, simulación robótica, diseño de algoritmos evolutivos y análisis de datos. Ya que se ha tenido que integrar múltiples disciplinas, se ha reforzado competencias como la planificación, la depuración sistemática y la documentación técnica rigurosa. También, se ha aprendido a afrontar la incertidumbre inherente a los sistemas evolutivos. A diferencia de los enfoques deterministas, trabajar con evolución implica aceptar cierto grado de imprevisibilidad, fomentando una mentalidad más flexible y adaptiva.

Por último, este proyecto ha sido una oportunidad para reflexionar sobre el estado actual de la inteligencia artificial y su capacidad para transformar varios sectores. Desarrollar un sistema autónomo desde cero, capaz de tomar decisiones en tiempo real, ha permitido valorar con mayor profundidad la sofisticación de tecnologías modernas. Asimismo, se ha adquirido una mayor apreciación hacia los grandes modelos de lenguaje (*LLMs*) como *chatGPT* o *Gemini*, reconociendo el largo proceso de iteración, entrenamiento y validación que subyace tras su aparente simplicidad. La observación de comportamientos emergentes en sistemas con comportamientos simples ha expuesto lo complejo que puede ser modelar la inteligencia, y como incluso soluciones modestas pueden producir propiedades inesperadas si no se entrenan adecuadamente o se cambia el ambiente en el que trabajan.

5.6. Futuros Trabajos

Este proyecto abre diversas posibilidades para trabajos futuros que pueden expandir, reinterpretar o adaptar los resultados obtenidos. Por ejemplo, se podría adaptar el proyecto al mundo físico, siendo una posible continuación al trasladar el controlador entrenado en simulación a un robot físico real. Así, se puede evaluar su desempeño ante el *reality gap*, y diseñar estrategias de transferencia de comportamiento, como la técnica *Domain Randomization.*

Se puede también extender el proyecto para que pueda realizar múltiples objetivos. A partir de la arquitectura ya establecida, podrían incorporarse nuevos comportamientos en la función de *fitness*, como el tratamiento de las bifurcaciones, la búsqueda de línea efectiva si se pierde y los cruces de líneas. Otro comportamiento añadido podría ser el de seguimiento de una esfera en el escenario, requeriríendo un rediseño del entorno y del sistema de recompensas y penalizaciones.

Otro proyecto futuro podría ser la exploración con coevolución. En este tipo de controladores, múltiples robots colaboran o compiten por un objetivo compartido, permitiendo estudiar comportamientos emergentes como la coordinación, la competencia por recursos o la formación de estrategias colectivas. Este trabajo tendría que reinterpretar la información procedente de los robots y de la escena, generando un sistema que permita modificar y expandir una red neuronal mucho más compleja.

También se puede realizar una integración con algoritmos de aprendizaje por refuerzo. Se podría estudiar la hibridación entre neuroevolución y aprendizaje por refuerzo profundo (*Deep Reinforcement Learning*), evaluando qué combinación permite mejores resultados en términos de convergencia, generalización y adaptabilidad.

Este trabajo asienta una base sólida y versátil sobre la que pueden construirse múltiples líneas de investigación y desarrollo. La estructura modular del sistema, el uso de algoritmos evolutivos y la integración con entornos de simulación permiten su expansión hacia nuevas tareas más complejas. Además, la arquitectura es compatible con enfoques más avanzados, como la coevolución multiagente permitiendo explorar sistemas más adaptativos, colaborativos y generalizables. Todo ello convierte este proyecto en una plataforma flexible para desarrollar soluciones más complejas dentro del campo de la robótica autónoma e inteligencia artificial.

5.7. Conclusiones

Se ha conseguido desarrollar e implementar con éxito un sistema de navegación autónoma mediante neuroevolución, capaz de seguir una línea utilizando información visual y sensorial. El sistema ha sido validado en dos circuitos de complejidad diferente, demostrando tanto la viabilidad del enfoque como sus actuales límites.

En el circuito 1 el controlador fue capaz de alcanzar un comportamiento óptimo en un número reducido de generaciones, obteniendo el valor máximo de *fitness* de 3600. Este mapa permitió comprobar la eficacia del sistema en un entorno con curvas suaves. En cambio, el circuito 2 que tiene trayectos más irregulares y curvas pronunciadas, exigió mayores capacidades de adaptación. Al comprobar que el mejor genotipo del primer circuito no interpretaba correctamente la escenografía del segundo mapa, se realizó un reentrenamiento del mejor individuo en el segundo circuito.

El reentrenamiento duró menos de 40 generaciones, observando cierta transferencia de comportamiento, pero también una pérdida de eficacia ante nuevas condiciones. El entrenamiento desde 0 en el segundo circuito ofreció resultados mucho más robustos, aunque requirió más del doble de generaciones en producirse que el reentrenamiento. Este entrenamiento generó un *fitness* de 3307, que, aunque no fuese el máximo obtenible, sí que era capaz de seguir la línea y terminar el circuito.

Estas observaciones reflejan tanto el potencial como las limitaciones del enfoque del controlador. Por un lado, el sistema es capaz de aprender desde 0 comportamientos complejos, pero por otro, la generalización a nuevos entornos no está garantizada. Esto evidencia una sensibilidad estructural en los genotipos generados por *NEAT*: una red adaptada a un entorno específico puede no converger eficazmente al ser expuesta a estímulos distintos. Es importante asegurar una generalización del controlador y por tanto capaz de ser utilizado para reentrenamiento. Así, es posible reducir el tiempo invertido en adaptarlo.

Para facilitar el reentrenamiento y mejorar la transferencia entre escenarios, sería necesario incorporar mecanismos adicionales al controlador. Por ejemplo, se podría preservar mejor la diversidad estructural en las primeras etapas o introducir capas ocultas iniciales que favorezcan una representación más general. También sería útil diseñar una arquitectura aún más modulable, permitiendo bloques funcionales reutilizables mientras solo se optimizan las partes necesarias para el nuevo escenario. Finalmente, una función de cálculo de *fitness* menos restrictiva y una mayor población por generación permitiría a *NEAT* no generar redes neuronales que se sobre ajusten a mapas, permitiendo una mayor generalización y por lo tanto una mayor capacidad de reentrenamiento.

Desde un punto de vista más global, este proyecto representa una prueba de concepto válida sobre la utilidad de la inteligencia artificial evolutiva en el diseño de comportamientos autónomos. El sistema desarrollado ha demostrado capacidad de adaptación, aprendizaje y mejora progresiva incluso en condiciones complejas. Al mismo tiempo, se han identificado con claridad los retos clave del enfoque: la sensibilidad a los entornos de entrenamiento, la necesidad de configuraciones precisas y el alto coste computacional de las simulaciones prolongadas.

En conjunto, este *TFG* ha servido como una prueba de concepto sólida de que la inteligencia evolutiva puede ser una herramienta efectiva para diseñar comportamientos autónomos en robótica. A la vez, ha dejado en evidencia los retos asociados a la escalabilidad, generalización y estabilidad de soluciones evolutivas. Se pretende que este trabajo abra la puerta a mejoras futuras que hagan los sistemas más flexibles, adaptables y reutilizables.

6. Análisis de Impacto

En este capítulo se analiza el impacto de los resultados obtenidos durante la realización del trabajo. Además, se relacionan los resultados con los Objetivos de Desarrollo Sostenible (ODS) de la Agenda 2030. También se destacan decisiones tomadas durante el desarrollo que se vinculan directamente con la consideración del impacto.

Desde un punto de vista personal, la realización de este TFG ha supuesto un gran avance en distintos aspectos, como la organización de tiempo, planificación y disciplina. Ya que el entrenamiento de una población podía durar más de 6 días, se tenía que organizar muy bien el tiempo disponible para poder realizar el proyecto a tiempo. La complejidad del proyecto ha permitido profundizar en áreas clave como la inteligencia artificial, la robótica, la programación avanzada en Python, y el uso de simuladores como *CoppeliaSim*. Además, el trabajo ha fomentado la planificación de experimentos, la evaluación crítica de resultados y la capacidad de resolver problemas de forma autónoma.

El desarrollo de una solución autónoma basada en técnicas de neuroevolución ha contribuido significativamente a la percepción de la inteligencia artificial como puesto de trabajo. Ahora que se han estudiado las bases del aprendizaje automático y de la robótica, se ha obtenido un mayor respeto hacia los avances automatizados. Asimismo, se ha favorecido el aprendizaje de buenas prácticas de programación, documentación y análisis de resultados, que son útiles en cualquier ámbito de trabajo que se escoja.

En el contexto empresarial, las técnicas aplicadas en este proyecto tienen un potencial claro de transferencia hacia el desarrollo de soluciones comerciales basadas en vehículos autónomos, robots de reparto o sistemas de inspección automatizada. El uso de algoritmos como *NEAT* permite desarrollar controladores capaces de adaptarse a distintos entornos sin requerir programación explícita, lo cual reduce costes y tiempo de implementación.

Ya que se está incrementando la necesidad de obtener sistemas de automatización de vehículos potentes y seguros, estas técnicas de aprendizaje se pueden considerar muy útiles. Tener información sobre los mejores caminos, velocidad y horario para maximizar la eficiencia siempre es clave para cualquier aplicación. Por lo tanto, aunque nunca se implementase la automatización de ciertos campos en las empresas, aplicar un aprendizaje evolutivo en muchos procesos será de los avances más importantes en el futuro.

Además, el enfoque modular del sistema, incluyendo una API desarrollada para comunicar Python con *CoppeliaSim* y una herramienta de análisis evolutivo personalizada, puede servir como base para futuros desarrollos en sectores como la logística, transporte público, la agricultura o la robótica de servicio. Este tipo de soluciones permiten mejorar la eficiencia operativa y reducir riesgos laborales ya que se automatizan tareas repetitivas o peligrosas. Si se integrasen correctamente estas técnicas en la vida laboral, la calidad de vida incrementería.

El impacto social de este tipo de tecnologías es significativo. Por un lado, la automatización de tareas puede liberar a las personas de trabajos monótonos, permitiendo una reorientación hacia labores más creativas como el arte, cinematografía o música. Por otro lado, es fundamental tener en cuenta el posible efecto negativo de la automatización en términos de sustitución de empleo. Como la revolución industrial, habrá muchos trabajos que se verán reducidos o completamente eliminados del mercado laboral, provocando que gente que lleve años especializándose en su trabajo se vuelvan irrelevantes.

En este trabajo se ha considerado este aspecto apostando por el uso de simuladores como herramienta educativa. Al ser una plataforma abierta, el sistema desarrollado puede utilizarse con fines educativos en instituciones académicas, facilitando la enseñanza de técnicas de inteligencia artificial, evolución biológica y control de robots. De este modo, se contribuye a la formación de futuros profesionales, ampliando el acceso al conocimiento en áreas de gran demanda. Estos profesionales, en vez de realizar los trabajos monótonos, supervisarían el aprendizaje de la red neuronal, generando nuevos trabajos y volviendo a haber otras nuevas oportunidades para las nuevas generaciones.

Desde el punto de vista económico, el sistema propuesto presenta beneficios tanto en términos de eficiencia como de escalabilidad. Al permitir la evolución automática de comportamientos, se evita la necesidad de diseñar manualmente algoritmos de control complejos. Esto implica una reducción de los tiempos de desarrollo y de los costes asociados al despliegue de robots autónomos. Ya que, con solo diseñar un controlador relativamente sencillo y que se puede usarse una y otra vez, solo habría un coste inicial de implementación. Estos costes se verían rápidamente cubiertos al tener en cuenta que se puede dejar el sistema encendido durante el tiempo necesario, permitiendo a los trabajadores realizar otras tareas de manera paralela.

Además, al utilizar software de código abierto (*OpenCV*, *NEAT*, *CoppeliaSim*, etc.), el proyecto reduce las barreras de entrada para pequeñas empresas o centros de investigación con recursos limitados. Así, se fomenta la democratización de la tecnología con tecnologías que son *open source* y por lo tanto gratis para todo el mundo. Al dejar que las implementaciones desarrolladas sean accedidas por todo el mundo de manera gratuita, permite generar una base de información para todo el mundo. Dicha información iría creciendo de manera exponencial, provocando que sean mucho más fácil implementar los controladores o refinarlos.

Sin embargo, también deben tenerse en cuenta los costes energéticos derivados del entrenamiento de modelos evolutivos, especialmente si se utilizan simulaciones masivas o prolongadas. En este trabajo, se ha mitigado este impacto mediante la optimización de simulaciones (limitando la duración, anticipando comportamientos no válidos y desactivando elementos gráficos innecesarios). De todos modos, si se quiere realizar tareas más complejas o que tarden más en converger en el objetivo puede ser un coste extremadamente elevado.

Hay dos razones por el cual este proyecto podría generar un impacto medioambiental. Por un lado, su aplicación real podría tener un efecto positivo, especialmente si se integra en robots diseñados para optimizar rutas de transporte. Así se consigue reducir desplazamientos innecesarios o controlar procesos industriales de forma más eficiente. Todo ello contribuiría a un menor consumo de recursos, reducción de emisiones y menos gastos de recursos naturales.

Por otro lado, hay que tener en cuenta el impacto energético asociado al entrenamiento evolutivo, que puede ser muy elevado. Aunque básicas, las medidas tomadas durante este proyecto para reducir el impacto energético han sido muy efectivas y pueden aplicarse a la mayoría de otros entrenamientos, Siempre pudiendo expandir o modificar los ajustes para reducir aún más el impacto. Estas medidas permitieron disminuir el tiempo de entrenamiento hasta en un 60%, contribuyendo a una ejecución más sostenible. Si las empresas y usuarios fuesen conscientes de estas disminuciones, también podrían reducir considerablemente el impacto al medio ambiente al implementar sus controladores.

Culturalmente, este proyecto se enmarca dentro de una tendencia creciente hacia la integración de la inteligencia artificial, aprendizaje automático y robótica en la vida cotidiana. Aun así, el desarrollo de robots capaces de aprender comportamientos sin intervención humana directa plantea retos éticos. Por ejemplo, el temor generado por ciencia ficción sobre los robots, como el libro *Yo, Robot* de Isaac Asimov o series como *Love, Death and Robots* genera un rechazo público hacía estos avances. Para disminuir el roce que podrían generar estas implementaciones, hay que implementar y desarrollar nuevos avances con la moralidad como uno de los pilares más importantes.

Hay que tener en cuenta que también abre nuevas oportunidades para la creatividad, el diseño colaborativo y la participación en entornos automatizados. Estas nuevas oportunidades tienen que ser descritas por el mundo científico como innovadoras y positivas para que el público sea más aceptante. Como todo, la cultura es un aspecto de la sociedad humana que no se puede cambiar de la noche a la mañana y hay que ir lentamente introduciendo estos avances en la vida cotidiana.

La utilización de entornos de simulación como *CoppeliaSim* y la apertura del código desarrollado favorecen la cultura de la colaboración, el aprendizaje entre pares y la transparencia en el desarrollo tecnológico. El enfoque educativo y accesible del sistema también lo convierte en una herramienta útil para fomentar vocaciones científicas y técnicas en niveles de institutos y universitarios. Con este aprendizaje, se tiene como objetivo que las nuevas generaciones tengan un mayor conocimiento sobre la robótica y la inteligencia artificial, reduciendo el miedo hacía los casos de un posible futuro oscuro donde la robótica y la inteligencia artificial controlan la humanidad.

A lo largo del desarrollo del TFG se tomaron decisiones específicas orientadas a reducir el impacto negativo de la implementación:

- Limitación temporal de simulaciones para reducir consumo energético.
- Priorización del software libre para fomentar la accesibilidad y reutilización.
- Documentación clara y modular del sistema para favorecer su adopción académica.
- Análisis constante de comportamientos no deseados para evitar soluciones no éticas o poco eficientes.
- Indicar las ventajas y desventajas para que el público tenga una mayor comprensión sobre la robótica y la inteligencia artificial.

7. Bibliografía

[1] S.D. News, "Moove and Waymo Expand Urban Mobility Partnership", *Self Drive News,* Dec. 06, 2024. https://selfdrivenews.com/moove-and-waymo-expand-urban-mobility-partnership/
[2] T. Greenawalt, "Descubre los 8 robots que están transformando los centros logísticos de Amazon de última generación," *ES About Amazon,* Oct. 14, 2024. https://www.aboutamazon.es/noticias/trabajar-en-amazon/8-robots-transformando-los-centros-logisticos
[3] "Que es la inteligencia artificial en las finanzas | IBM," *www.ibm.com*, Feb. 19, 2024. https://www.ibm.com/es-es/topics/artificial-intelligence-finance
[4] "Neuroevolution of augmenting topologies," *Wikipedia,* Dec.10, 2020 https://en.wikipedia.org/wiki/Neuroevolution_of_augmenting_topologies

[5] "CoppeliaSim User Manual," *manual.coppeliarobotics.com.*
[6] "Welcome to NEAT-Python's documentation! – NEAT-Python 0.92 documentation," *Readthedocs.io,* 2015. https://neat-python.readthedocs.io/en/latest/
[7] OpenCV, "OpenCV library," *Opencv.org,* 2019 https://opencv.org/
[8] Matplotlib, "Matplotlib: Python Plotting – Matplotlib 3.1.1 Documentation," *Matplotlib.org,* May 30, 2024 https://matplotlib.org/

[9] B. Siciliano and O. Khatib, *Springer handbook of Robotics.* Berlin: Springer, 2016
[10] K.O. Stanley and R. Miikkulainen, "Evolving Neural Networks through Augmenting Topologies," *Evolutionary Computation,* vol. 10, no. 2, pp. 99 – 127, Jun. 2002, doi: https://doi.org/10.1162/106365602320169811.
[11] S. Khamesian and H. Malek, "Hybrid self-attention NEAT: a novel evolutionary self-attention approach to improve the NEAT algorithm in high dimensional inputs," *Evolving Systems,* vol. 15, no. 2, pp. 489-503, Jun. 2023, doi: https://doi.org/10.1007/s12530-023-09510-3.
[12] K. Bousmalis and S. Levine, "Closing the simulation-to-reality gap for Deep robotic learning," *research google,* Oct. 2917. https://research.google/blog/closing-the-simulation-to-reality-gap-for-deep-robotic-learning/

[13] Tech With Nijola, "Snake learns with neuroevolution (implementing NEAET from scratch in C++)", *Youtube,* Oct. 04, 2023. https://www.youtube.com/watch?v=lAjcH-hCusg
[14] Underpower Jet, "Evolving Nueoral Networks NEAT with 3D Cars + Tutoral," *Youtube,* Jul. 16, 2017. https://www.youtube.com/watch?v=X8ieBGMjtzM
[15] Pranav Bhounsule, Playlist: "CoppeliaSim 4.3.0 Tutorial," *Youtube,* Jun. 20, 2022. https://www.youtube.com/watch?v=g73pygOBuA4&list=PLc7bpbeTIk74RR8V1yCYnF5MT9KH2D98b

[16] Kshitij Aucharmal (2023) "Flappy Bird Learning using NEAT," https://github.com/kshitijaucharmal/FlappyNEAT
[17] McIntyre, Alan and Kallada, Matt and Miguel, Cesar G. and Feher de Silva, Carolina and Netto, Marcio Lobo (2024) "neat-python," https://github.com/CodeReclaimers/neat-python
[18] Javier de Lopez Asiain (2025) "Robotica," https://github.com/jdlope/robotica
[19] Yudarw (2024) "Coppeliasim-proyects-with-zmqRemoteApi," https://github.com/yudarw/coppeliasim-projects-with-zmqRemoteApi

.

8. Anexo

EDUARDO MIRALLES CIORDIA

Entrega Final.pdf

Turnitin Memoria Final
TFG ETSIINF (Moodle PP)
Universidad Politecnica de Madrid

Document Details

Submission ID
trn:oid:::1:3268745828

Submission Date
Jun 4, 2025, 10:20 PM GMT+2

Download Date
Jun 4, 2025, 10:54 PM GMT+2

File Name
10871_EDUARDO_MIRALLES_CIORDIA_Entrega_Final_83714_1502031922.pdf

File Size
1.9 MB

49 Pages
20,187 Words
115,962 Characters

1% Overall Similarity

The combined total of all matches, including overlapping sources, for each database.

Filtered from the Report

- Bibliography
- Quoted Text

Top Sources

0% Internet sources
0% Publications
1% Submitted works (Student Papers)

Top Sources

0% Internet sources
0% Publications
1% Submitted works (Student Papers)

Top Sources

The sources with the highest number of matches within the submission. Overlapping sources will not be displayed.

1 Student papers
Universidad Politécnica de Madrid <1%

2 Student papers
Instituto Tecnologico de Costa Rica <1%

3 Student papers
Universidad de Antioquia <1%

4 Student papers
University of Strathclyde <1%

Este documento esta firmado por

	**Firmante**	CN=tfgm.fi.upm.es, OU=CCFI, O=ETS Ingenieros Informaticos - UPM, C=ES
	Fecha/Hora	Wed Jun 04 23:21:16 CEST 2025
	Emisor del Certificado	EMAILADDRESS=camanager@etsiinf.upm.es, CN=CA ETS Ingenieros Informaticos, O=ETS Ingenieros Informaticos - UPM, C=ES
	Numero de Serie	561
	Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)