uela Técnica
erior
ngeniería de
comunicación

DISEÑO E IMPLEMENTACIÓN DE UN ALGORITMO NO SUPERVISADO PARA LA CONDUCCIÓN AUTÓNOMA EN UN CIRCUITO DE PRUEBAS VIRTUAL

AUTOR: FRANCISCO ALFREDO MORENO URREA

DIRECTORES: JAVIER VALES ALONSO Y

JUAN CARLOS JACOBO AARNOUTSE SÁNCHEZ

JULIO DE 2023

Agradecimientos

En primer lugar, quería dar las gracias a los directores de este Trabajo Fin de Máster, Dr. Juan Carlos Jacobo Aarnoutse Sánchez y Dr. Javier Vales Alonso por su atención, dedicación y aclaración de dudas sobre el trabajo durante el trascurso de este proyecto.

Por último, me gustaría agradecer a mis padres y mi hermana, por su paciencia, comprensión y apoyo incondicional que siempre me brindan. Y agradecer a Sara por su motivación constante, puesto que sin ellos esto no habría sido posible.

Muchas gracias.

Resumen

Francisco Alfredo Moreno Urrea. "Diseño e implementación de un algoritmo no supervisado para la conducción autónoma en un circuito de pruebas virtual", Cartagena julio 2023.

Desde la aparición del automóvil, se ha especulado con la idea de crear un vehículo motorizado que fuera capaz de realizar tareas como desplazarse de forma autónoma. Esta idea se ha visto frenada durante algún tiempo debido a las limitaciones de la tecnología de la época. Sin embargo, no ha sido hasta hace poco el momento en el que se ha producido un salto de calidad en la calidad del sector. Tesla ha sido una de las empresas que más ha apostado, y está más cerca del vehículo autónomo de nivel 5.

Esta idea nació originalmente para aumentar el nivel de seguridad vial debido a que los accidentes por salida de la vía originan entre el 30 % y el 40 % de las muertes relacionadas con el tráfico. Si el coche fuera capaz de detectar un accidente por salida de la vía antes de que se produzca y tomar medidas al respecto, estos datos podrían reducirse. Otra de las ideas principales sobre el desarrollo de vehículos autónomos, tiene el objetivo de aportar una experiencia más cómoda durante el viaje ya que el usuario no tiene necesidad de estar concentrado en el volante reduciendo el estrés.

Dependemos casi absolutamente de la tecnología disponible. Por ejemplo, un vehículo que circule por un circuito será rastreado para conocer las distancias mediante radares, tomando información sobre la vía, simulando sistemas reales de control de crucero adaptativo.

Suponiendo que todas las implementaciones funcionen correctamente, habremos diseñado un vehículo capaz de circular por un circuito por sí solo, sin intervención humana

Abstract

Francisco Alfredo Moreno Urrea. "Design and implementation of an unsupervised algorithm for autonomous driving on a virtual test track.", Cartagena July 2023.

Since car's appearance, there has been speculation about the idea of creating a motorized vehicle that would be capable of performing tasks such as autonomous travel. This idea has been stopped during some time because of the limitations on the technology of that time. However, It has not been until recently the moment in which there has been a leap in quality in sector's quality. Tesla has been one of the companies which has bet more, and is closer to the level 5 autonomous vehicle.

This idea was originally born in order to increase the level of road safety due to accidents caused by running off the road originating between 30 % and 40 % of traffic-related deaths. If the car would be able to detect a roadside accident before it happens and take action, this data could be reduced. Another of the main ideas about autonomous vehicle development, has the objective to bring a more comfortable experience while traveling because the user has no need about being focused in the steering wheel reducing the stress.

We depend almost absolutely on the available technology. For example, a vehicle going through a circuit will be traced to know the distances by radars, taking information about the track , simulating real adaptative cruising control systems.

If all the implementations work correctly, a vehicle capable of self-drive by a track by it's own will be developed.

Índice general

1. Introducción y objetivos **15**
1.1. El problema de la conducción autónoma. Niveles y objetivos . . . 15
1.1.1. Historia . . . 15
1.1.2. Niveles de automatización . . . 19
1.1.2.1. Nivel 0 . . . 20
1.1.2.2. Nivel 1 . . . 20
1.1.2.3. Nivel 2 . . . 20
1.1.2.4. Nivel 3 . . . 20
1.1.2.5. Nivel 4 . . . 20
1.1.2.6. Nivel 5 . . . 20
1.1.3. Partes Hardware automatización . . . 22
1.2. Machine Learning y tipos . . . 23
1.3. Reinforcement learning: objetivos y algoritmos principales . . . 28
1.3.1. Historia . . . 28
1.3.2. Objetivos . . . 29
1.3.3. Algoritmos principales . . . 31
1.3.3.1. Q-Learning . . . 31
1.3.3.2. NEAT . . . 31
1.4. NEAT Operación, funcionamiento y características principales . . . 33
1.4.1. Neuroevolución . . . 33
1.4.2. Algoritmos genéticos . . . 34
1.4.3. NEAT . . . 36
1.4.3.1. Historia . . . 36
1.4.3.2. Codificación . . . 36
1.4.3.3. Convenciones competidoras . . . 36
1.4.3.4. Mutaciones . . . 36
1.4.3.5. Reproducción . . . 37
1.4.3.6. Especiación . . . 37
1.4.4. Población inicial . . . 38
1.4.5. Parámetros de configuración . . . 38
1.5. Entorno de desarrollo y herramientas . . . 39

2. Trabajos relacionados **41**
2.1. Trabajos académicos . . . 41
2.2. Desarrollos relacionados . . . 42
2.2.1. Python . . . 42
2.2.2. Trackmania . . . 43
2.2.2.1. API . . . 44
2.2.2.2. OpenCV . . . 45
2.2.3. JavaScript . . . 46
2.2.4. C# . . . 49

3. Desarrollo y descripción del algoritmo en NEAT **51**
3.1. El algoritmo NEAT . . . 51

3.2. Adaptación del algoritmo . . . 51
3.2.1. Modo Consola . . . 53
3.2.2. Modo Interactivo . . . 54
3.2.3. Modo Supervisión . . . 57
3.2.3.1. Modo Entrenamiento . . . 58
3.2.3.2. Modo Repetición . . . 58
3.2.3.3. Modo Cargar . . . 59
3.2.4. Modo Monitorización . . . 60
3.3. Entrenamiento . . . 61
3.3.1. Archivo Car.py . . . 62
3.3.1.1. Establecer posición . . . 62
3.3.1.2. Establecimiento de variables . . . 63
3.3.1.3. Definir funciones . . . 64
3.3.1.4. Fitness . . . 68
3.3.2. Archivo PyCar.py . . . 70
3.3.2.1. Iniciar NEAT . . . 70
3.3.2.2. Iniciar juego . . . 71
3.3.2.3. Controles generados por la red neuronal . . . 71
3.3.2.4. Marcha atrás . . . 72
3.3.2.5. Actualización de las vueltas . . . 73
3.3.2.6. Recompensas, penalizaciones velocidad y giros . . . 73
3.3.2.7. Precisión algoritmo . . . 74
3.3.2.8. Representar datos en la generación . . . 75
3.3.3. Archivo de configuración . . . 76

4. Pruebas **79**
4.1. Sensores . . . 79
4.2. Ajustes en archivo de configuración . . . 82
4.3. Mapas . . . 82
4.4. Velocidad . . . 84
4.5. Ángulo de giro . . . 86
4.6. Distintas funciones . . . 87

5. Conclusiones y líneas futuras **89**
5.1. Conclusiones . . . 89
5.2. Conocimientos adquiridos . . . 89
5.3. Líneas futuras . . . 90

6. Bibliografía **91**

Índice de figuras

1.1. Carro autopropulsado de Da Vinci . . . 15
1.2. Vehículo a control remoto diseñado por Houdina . . . 16
1.3. Vehículo autónomo presentado en Futurama . . . 16
1.4. Prototipo Stanford Cart . . . 17
1.5. Prototipo NavLab 5 . . . 18
1.6. Prototipo taxi autónomo Ford . . . 18
1.7. Niveles de automatización de la conducción . . . 19
1.8. Vehículo Google nivel 5 . . . 21
1.9. Tipos de inteligencia artificial . . . 23
1.10. Comparación entre neuronas biológicas y artificiales . . . 24
1.11. Modelo de perceptrón monocapa . . . 24
1.12. Modelo de perceptrón multicapa . . . 25
1.13. Aprendizaje supervisado . . . 25
1.14. Aprendizaje no supervisado . . . 26
1.15. Aprendizaje por refuerzo . . . 27
1.16. La superstición de la paloma . . . 28
1.17. Experimento condicionamiento clásico Pávlov . . . 28
1.18. Diagrama de flujo aprendizaje reforzado . . . 30
1.19. Ejemplo gráfico Q-Tabla . . . 32
1.20. Evolución biológica de una especie . . . 33
1.21. Ejemplo genético . . . 35
1.22. Ejemplos de juegos realizados con Pygame. . . . 39

2.1. Configuración de los sockets del servidor . . . 43
2.2. Obtención de los datos de la API . . . 44
2.3. Código NEAT en Trackmania . . . 45
2.4. Aplicación Canny . . . 46
2.5. Conducción autónoma con JavaScript [29] . . . 47
2.6. Conducción autónoma con JavaScript usando Box2D . . . 48
2.7. Conducción autónoma con Unity . . . 49

3.1. Configuración velocidad variable . . . 51
3.2. Mapas usados en el algoritmo . . . 52
3.3. Salida original . . . 52
3.4. Código para guardar las generaciones . . . 53
3.5. Código para representar y guardar las gráficas de *fitness* . . . 53
3.6. Modo Consola . . . 53
3.7. Detalles tecla P . . . 55
3.8. Detalles función Tiempo de Parada . . . 55
3.9. Detalles teclas Tiempo de Parada . . . 56
3.10. Modo Supervisión . . . 57
3.11. Importar librerías . . . 57
3.12. Comprobar palabras clave . . . 57
3.13. Relación Palabras-Modo. . . . 58
3.14. Modo repetición . . . 59

3.15. Modo carga 59
3.16. Modo de monitorización 60
3.17. Ejemplo de modo monitor 61
3.18. Definición de posiciones según el mapa 62
3.19. Configuración de las variables iniciales 63
3.20. Configuración de las variables de las salidas 63
3.21. Configuración de variables del tiempo 63
3.22. Función posición vehículo 64
3.23. Establecer los puntos de colisiones 64
3.24. Comprobar posibles colisiones 65
3.25. Función para registrar la muerte 65
3.26. Función comprobar vuelta 66
3.27. Función para comprobar radar 67
3.28. Función para dibujar el radar 67
3.29. Función para obtener datos sensores 67
3.30. Función para elegir tipo de sensorización 68
3.31. Función para rotar la imagen 68
3.32. Función recompensa 69
3.33. Iniciar algoritmo NEAT 70
3.34. Iniciar el juego 71
3.35. Determinar ángulo de giro 72
3.36. Detectar marcha atrás 73
3.37. Obtener vueltas 73
3.38. Penalizar recompensas 74
3.39. Condición 1 - Acumulación de coches 74
3.40. Condición 2 - Mejorar precisión 74
3.41. Condición 3 - Buscar resultados 75
3.42. Actualización de los datos en la pantalla 75
3.43. Configuración de población y umbral 76
3.44. Configuración de activación 76
3.45. Parámetros de red 76
3.46. Configuración del umbral misma especie 76
3.47. Configuración estancamiento 77
3.48. Configuración reproducción 77

4.1. Dos sensores laterales 79
4.2. Dos sensores frontales. 79
4.3. Tres sensores. 80
4.4. Cuatro sensores. 80
4.5. Cinco sensores. 80
4.6. Diferentes ubicaciones de los dos sensores 81
4.7. Zonas criticas mapa 2. 82
4.8. Zonas criticas mapa 6. 83
4.9. Zonas criticas mapa 10. 84

Índice de tablas

4.1. Configuración de la velocidad 85
4.2. Evolución de tiempos por vuelta usando velocidad variable 85
4.3. Configuración del ángulo de giro 86
4.4. Comparativa entre resultados según la disponibilidad o no de las condiciones de precisión 88

Capítulo 1

Introducción y objetivos

1.1 El problema de la conducción autónoma. Niveles y objetivos

1.1.1. Historia

El concepto de coche autónomo comienza a finales del siglo XV a raíz de la idea del inventor Leonardo Da Vinci [1]. Da Vinci plasmó en unos bocetos la idea que tenía sobre el carro en cuestión. Este carro no tenía la finalidad de transportar personas como los posteriores, pero su mecanismo de autopropulsado era de lo más interesante. Su funcionamiento se basaba en la acción de los muelles ballesta que nivelaban el movimiento y otros muelles de espiral que conseguían generar el movimiento para así conseguir desplazar el vehículo durante varios metros. A su vez, disponía de un mecanismo similar a un diferencial que permitiría ajustar el ángulo de giro. Este proyecto fue realizado con éxito por el experto en la historia de la ciencia Paolo Galluzzi en 2004, junto a los científicos Carlo Peretti y Mark Roshlem.

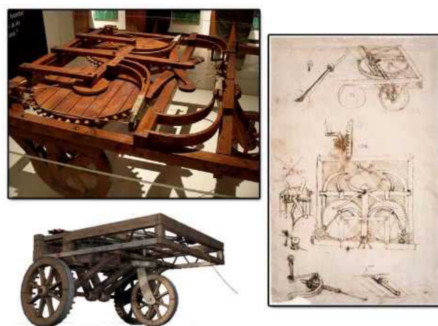


Figura 1.1: Carro autopropulsado de Da Vinci

A principios del siglo XX, un ingeniero eléctrico estadounidense llamado Francis Houdina traspasó el plano teórico y diseñó un vehículo autónomo, aunque su idea era que el automóvil fuera controlado a distancia. Su primer prototipo fue mostrado al público en Manhattan en el año 1925, en el que recorrió 19 kilómetros. Sin embargo, un choque con otro automóvil impidió que recorriera más distancia. Este control remoto utilizaba radiofrecuencia y permitía que el vehículo utilizara su motor, circulando e incluso tocando el claxon sin necesidad de tener un conductor a bordo del coche. El vehículo siguió fabricándose hasta la década de los 30, cuando finalmente la empresa que lo producía, Chandler (nombre del vehículo que le pusieron), desapareció debido a la poca demanda de este producto y a las regulaciones necesarias para su funcionamiento.

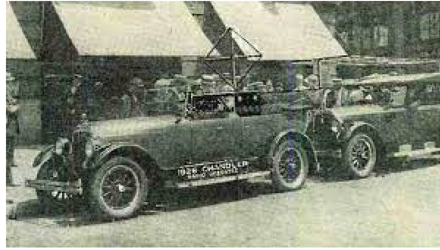


Figura 1.2: Vehículo a control remoto diseñado por Houdina

Años más tarde, durante la Gran Depresión (1939), se celebró una feria de muestras llamada Futurama [2]. Estaba enmarcada dentro de la Exposición Universal de Nueva York. En dicha feria se mostraron los avances científicos y técnicos aplicados a cómo sería el futuro 20 años después. En este escenario, el diseñador industrial estadounidense Norman Bel Geddes presentó la idea de coches eléctricos sin necesidad de conductor, circulando por carreteras que suministraban energía a los vehículos que circulaban por ellas. Dichos vehículos presentaban visualmente un diseño muy futurista para la época.

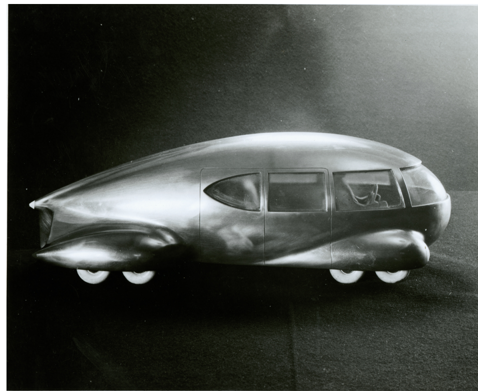


Figura 1.3: Vehículo autónomo presentado en Futurama

No fue hasta 1958 cuando este modelo propuesto en la Exposición Universal se hizo realidad. Se presentó un automóvil con sensores integrados, con los cuales podía detectar la corriente que fluía a través de un cable incrustado en la carretera.

Unos años después, en pleno auge de la carrera espacial, se desarrolló el carro “Stanford Cart” por parte de James Adams. Este estaba equipado con cámaras y estaba programado para detectar y seguir una línea en el suelo de forma autónoma.El objetivo de este carro era aterrizar vehículos en la Luna.Uno de los avances que proporcionó este proyecto fue el uso de cámaras en los vehículos autónomos, elemento vital en los vehículos autónomos recientes.

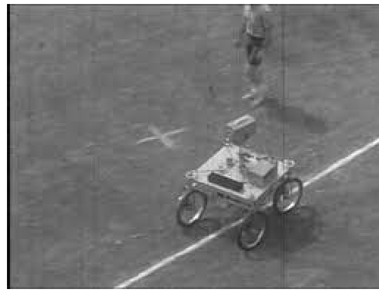


Figura 1.4: Prototipo Stanford Cart

Ya en 1977, en los laboratorios Tsukuba Mechanical Engineering de Japón, mejoraron esta idea con un sistema de cámaras que transmitía los datos a un ordenador para que este pudiera procesar imágenes de la carretera. Este se considera el primer vehículo realmente automatizado, ya que realizaba el transporte de pasajeros de forma totalmente autónoma, sin intervención del usuario. En este proyecto se llegaron a alcanzar velocidades de 30 Km/h.

Entre las décadas de los 80 y 90, Japón, EEUU y Europa iniciaron proyectos para brindar soluciones a los problemas de tráfico. Uno de los enfoques principales fue utilizar la conducción autónoma para disminuir la cantidad de accidentes, aumentar la eficiencia del flujo de tráfico y optimizar el combustible [3]. Una de las soluciones que propuso Japón en un proyecto es la utilización de las comunicaciones entre los vehículos implicados para sincronizar los movimientos.

Mientras tanto, en Alemania, se siguió apostando por alcanzar una velocidad mayor que la establecida en Japón. Fue en el año 1986 cuando se logró por primera vez controlar el acelerador y el freno al mismo tiempo, utilizando comandos de ordenador basados en un procesamiento en tiempo real de las imágenes. Este proyecto fue realizado por la Universidad de Múnich y alcanzó una velocidad máxima de 100 Km/h [4].

En Europa, también se realizó el denominado Proyecto Eureka PROMETHEUS (PROgraMme for a European Traffic of Highest Efficiency and Unprecedented Safety). Este fue el mayor proyecto de I+D relacionado con los automóviles sin conductor. En él, unificaron las ideas establecidas por Japón y Alemania mencionadas anteriormente. En el proyecto destacan el realizado en París en 1994 y el proyecto realizado en Munich y Copenhague en 1995.

El proyecto realizado en París fue conocido como “VaMP”. Realizado con un Mercedes 500 SEL, fue capaz de recorrer más de 1000 kilómetros.

De forma coetánea, en la Universidad Carnegie Mellon (Estados Unidos), comenzaron a construir vehículos autónomos, integrando, en esta ocasión, redes neuronales para el procesamiento de imágenes y controles de dirección. Esto suponía un gran avance con respecto a proyectos anteriores debido a que se permitía una mayor libertad a la hora del aprendizaje por parte del algoritmo. En 1995, llevaron dicho vehículo a la carretera y lo llamaron NavLab 5. En esta ocasión, recorrieron cerca de 5000 km con un nivel de autonomía del 98 % del tiempo, siendo el ser humano quien controlaba la velocidad y el frenado para el control de seguridad. A este proyecto se le llamó “No Hands across America”.

Figura 1.5: Prototipo NavLab 5

Mientras Estados Unidos orientaba su enfoque hacia un mundo más militar (creación de vehículos terrestres no tripulados para navegar por caminos fuera de línea y evitar obstáculos, DARPA), Japón, de la mano de Toyota, se orientaba más hacia mantener una velocidad programada y detectar si hay un obstáculo delante para reducir la velocidad. Esto se denominó “Sistema de control de crucero adaptativo”.

Desde 2009, muchas otras compañías se han centrado en desarrollar y optimizar las características autónomas para los nuevos vehículos. Desde entonces, la financiación ha pasado del sector público (Eureka) al sector privado (Google, Tesla, Uber, etc.). Desde ese año, han ido alcanzando distintos niveles, hasta llegar al nivel 5 (en 2020). Actualmente, hay diversas compañías que están en fase de pruebas en este nivel, desde Ford (probando taxis robotizados) hasta Coca-Cola (en Suecia se transporta la mercancía a los almacenes minoristas de alimentos), pasando por Amazon (utilizando drones para el reparto a clientes de Prime) y Tesla (con su Tesla D, capaz de estacionar sin intervención del usuario) [5].

Figura 1.6: Prototipo taxi autónomo Ford

1.1.2. Niveles de automatización

Los niveles de automatización de un vehículo indican el grado de interacción con el usuario. Existen 6 niveles, siendo el nivel 0 el grado en el que se necesita más la intervención del usuario debido a la nula automatización. Estos niveles fueron creados por la Sociedad de Ingenieros Automotrices (SAE) en 2014 en el estándar SAE J 3016. Esta clasificación se basa en cuatro aspectos fundamentales:

- Responsabilidad del movimiento, ya sea la dirección o la velocidad.
- Detección y respuesta ante estímulos específicos (objetos u obstáculos).
- Acción ante situaciones de fallo.
- Capacidad del sistema de conducción para adaptarse a situaciones desfavorables.

Siguiendo la clasificación de la NHTSA[6], también contamos con 6 niveles de automatización, al igual que la SAE[7, 8].

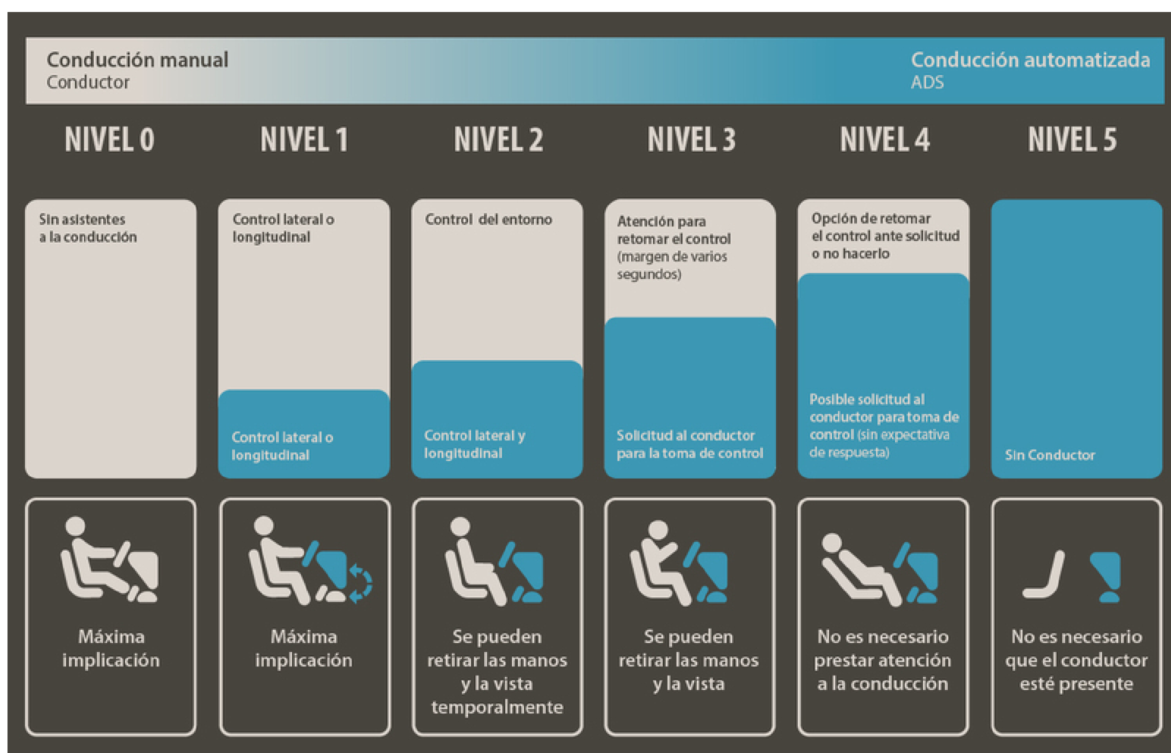


Figura 1.7: Niveles de automatización de la conducción

1.1.2.1. Nivel 0

No hay automatización de la conducción. Este es el nivel más básico. Todas las tareas recaen sobre el conductor. En este nivel se incluyen todos los coches convencionales, ya sean económicos o antiguos, que no cuentan con sistemas de automatización.

1.1.2.2. Nivel 1

Asistencia al conductor. En este nivel, el vehículo comienza a contar con algún sistema de asistencia, ya sea longitudinal (aceleración o frenado) o lateral (desplazamiento hacia un lado u otro). El conductor realiza el resto de las habilidades. Todos los coches que tienen un sistema de control de velocidad de crucero adaptativo entran en este nivel. Se establece una velocidad programada, pero el conductor sigue siendo responsable de manejar el volante. También se incluyen en este nivel los coches que tienen sistemas de estacionamiento asistido, ya que actúan sobre la dirección y no sobre la velocidad.

1.1.2.3. Nivel 2

Automatización parcial de la conducción. El vehículo cuenta con sistemas de asistencia tanto longitudinal como lateral al mismo tiempo. Sin embargo, la detección de obstáculos y otras tareas siguen siendo responsabilidad del usuario. Actualmente, este es el sistema más extendido. En este nivel se incluyen los vehículos con un piloto automático temporal para autopistas, como el Nissan Qashqai con ProPilot, los vehículos con sistemas de asistencia en atascos de tráfico y los coches con estacionamiento asistido que actúan sobre la dirección y la velocidad.

1.1.2.4. Nivel 3

Automatización condicionada de la conducción. Este nivel ofrece asistencia tanto en el movimiento longitudinal como en el lateral e incluye mecanismos para la detección de obstáculos. Sin embargo, el usuario debe intervenir en situaciones no contempladas por el sistema. Comercialmente, este nivel es difícil de encontrar, aunque se podría considerar el Tesla Model S con el sistema Autopilot 2.0.

1.1.2.5. Nivel 4

Automatización elevada de la conducción. Este nivel ofrece la asistencia del nivel anterior y añade un sistema de respaldo para reaccionar en caso de emergencia sin necesidad de la intervención del usuario. Comercialmente, no existen coches de este tipo, aunque hay prototipos de varios fabricantes que aún tienen volante y pedales, un conductor humano y un botón de pánico para desactivar el sistema de automatización y que el humano pueda retomar el control del vehículo.

1.1.2.6. Nivel 5

Automatización completa de la conducción. Este es el máximo nivel de automatización, donde no se requiere la intervención del usuario en ningún momento, ya que el vehículo es capaz de detectar y sortear obstáculos. También se incluyen sistemas de respaldo para situaciones de emergencia.

En la actualidad, existen algunos vehículos que podrían considerarse en este nivel según la definición, como el coche autónomo de Google sin conductor.

Cada uno de los niveles presenta un riesgo en la conducción. En niveles bajos de automatización,

Figura 1.8: Vehículo Google nivel 5

existe el riesgo de distracción, donde el conductor puede atribuir al vehículo características que no posee. Por otro lado, en niveles altos de automatización, surge el problema de otorgar a la inteligencia artificial plenos poderes, lo que implica enfrentarse a situaciones que no controla y no saber cómo actuar, mientras el conductor no tiene mecanismos para retomar el control.

1.1.3. Partes Hardware automatización

Para que un vehículo pueda ser automatizado, va a necesitar distintos dispositivos y sensores para poder recolectar la información y poder procesarla correctamente. Entre otros destacan:

- **Ordenador central:** Cerebro de este sistema, recibe la información procedente de los sensores y de las cámaras para procesarlo y tomar la siguiente decisión relativa al movimiento del coche

- **Radares:** Dispositivos utilizados principalmente en el sector aéreo y naval. Tienen como función principal la detección de objetivos y obstáculos. En este proyecto hemos incluido este hardware.

- **Sensores de ultrasonidos:** Dispositivo similar al anterior, utiliza ondas sonoras a altas frecuencias para la detección de obstáculos. Utilizados, principalmente, para el aparcamiento o para detectar objetos a baja velocidad.

- **Cámara:** Dispositivos por los cuales podemos generar una imagen del entorno que rodea al vehículo y poder pasárselo al ordenador central para que procese la información. Utilizado para detección de señales de tráfico, marcas viales, etc.

- **LIDAR:** Dispositivo que genera una visión total del entorno que rodea al vehículo, esta proyección de millones de haces de luz genera una mejor detección de objetos que, junto a la información procedente de las cámaras, se puede realizar un mapa 3D.

- **GPS o GALILEO:** Sistema de posicionamiento por satélite en el que nos permite conocer la posición exacta del vehículo en cada instante.

Para nuestro proyecto solo necesitaremos radares, para calcular la distancia a los objetivos y obstáculos y un ordenador central para que pueda procesar esta información.

1.2 Machine Learning y tipos

Para entender qué es el aprendizaje máquina, primero hay que contextualizar esta tecnología. El aprendizaje máquina es un subtipo de la inteligencia artificial (IA).

La inteligencia artificial es un campo perteneciente a la ciencia informática que está dedicado a la resolución de diversos problemas cognitivos, comúnmente atribuidos a la inteligencia humana, ya sea el aprendizaje mediante reconocimiento de patrones o por experiencia propia, como en el caso de la resolución de problemas [9]. Dentro de esta disciplina se derivan otros campos de la ciencia informática, como el “Machine Learning” y “Deep Learning”.

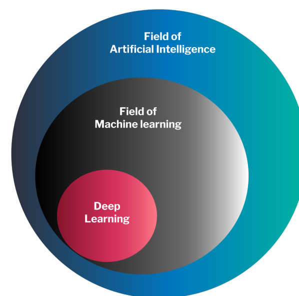


Figura 1.9: Tipos de inteligencia artificial

Se denomina aprendizaje máquina el campo dentro de la inteligencia artificial encargado de otorgar capacidades a un ordenador para realizar una determinada tarea sin haber necesitado programarlo expresamente. Este conjunto de algoritmos es capaz de aprender datos registrados y hacer predicciones basados en ellos. Este tipo de aprendizaje se suele implementar cuando la programación del problema resulta complicada y tediosa. La clave de la precisión de estos modelos reside principalmente en la disposición de una gran cantidad de datos y que sean de buena calidad.

Dentro del aprendizaje máquina se encuentra el denominado “aprendizaje profundo” o “Deep Learning”, en el cual se busca comprender mejor los datos del entrenamiento. Esto se logra porque el algoritmo busca identificar relaciones entre los distintos elementos de la muestra y poder crear predicciones con el mayor índice de acierto posible.La clave de éxito en este tipo de problemas reside en la parte de software, para realizar la limpieza de los datos y entrenar correctamente al algoritmo, y parte de hardware,para realizar un mayor número de operaciones en menos tiempo (en este aspecto fue de vital importancia la introducción de la GPU para el procesamiento general).

El aprendizaje profundo se caracteriza también por la presencia de un capas de algoritmos,las cuales se conocen como nodos. Estas redes empezaron a desarrollarse a partir de finales de los años 50.

En 1958,el psicólogo estadounidense Frank Rosenblatt se basó tanto en los estudios del funcionamiento del cerebro humano realizados por Santiago Ramón Y Cajal y Charles Scott Sherrintong como en la idea expuesta por Warren McCulloch y Walter Pitts sobre la posibilidad de crear redes neuronales artificiales para proponer el perceptrón monocapa [10][11].

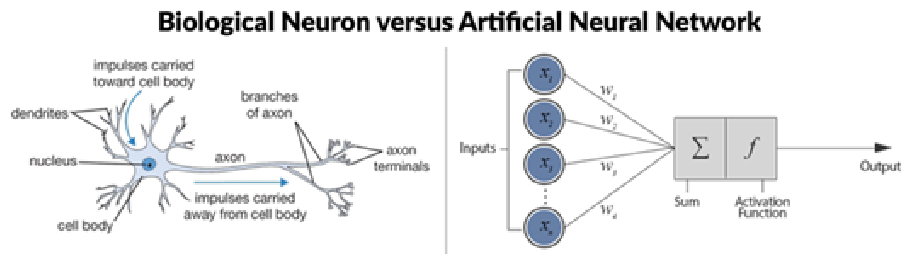


Figura 1.10: Comparación entre neuronas biológicas y artificiales

El perceptrón monocapa está compuesto por una única capa de neuronas, las cuales se encuentran localizadas en la capa de salida. Este tipo de redes neuronales necesitan un tiempo de cómputo muy bajo debido a que resuelven problemas para objetos linealmente separables con resultados binarios (1s y 0s), por lo tanto, no requieren demasiados cálculos.

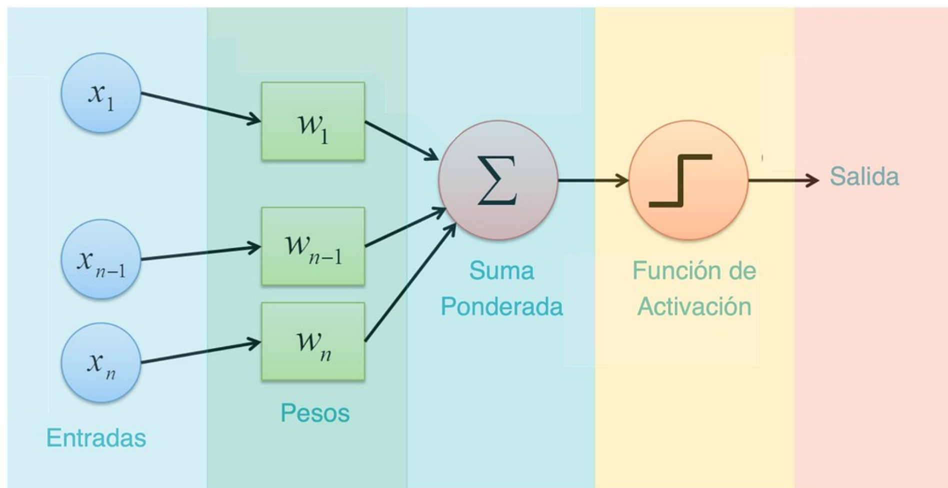


Figura 1.11: Modelo de perceptrón monocapa

Posteriormente, esta teoría fue ampliada a una red neuronal multicapa en 1986 por el psicólogo estadounidense David Everett Rumelhart. Estas redes estaban compuestas por más de una capa de neuronas,las cuales se dividían en capas intermedias y capas de salida. Tiene una estructura similar a la del perceptrón monocapa si consideramos que este no posee capas ocultas [12].

Este tipo de red neuronal se ejecuta en dos etapas, la que va desde la capa de entrada hacia la capa de salida, según la función de activación y la capa que tiene la ruta inversa, que va propagando el error entre el valor real obtenido y el valor esperado. La función de esta segunda etapa es modificar los pesos.

El aprendizaje puede dividirse en cuatro tipos fundamentales:

- Aprendizaje supervisado: Este tipo de aprendizaje se basa en la respuesta deseada a partir de una determinada entrada. En otras palabras, esta técnica se construye utilizando dos conjuntos de datos: uno destinado a entrenar al algoritmo, proporcionando ejemplos de lo que es cada cosa, y otro conjunto reservado para indicarle al algoritmo qué es exactamente cada foto.

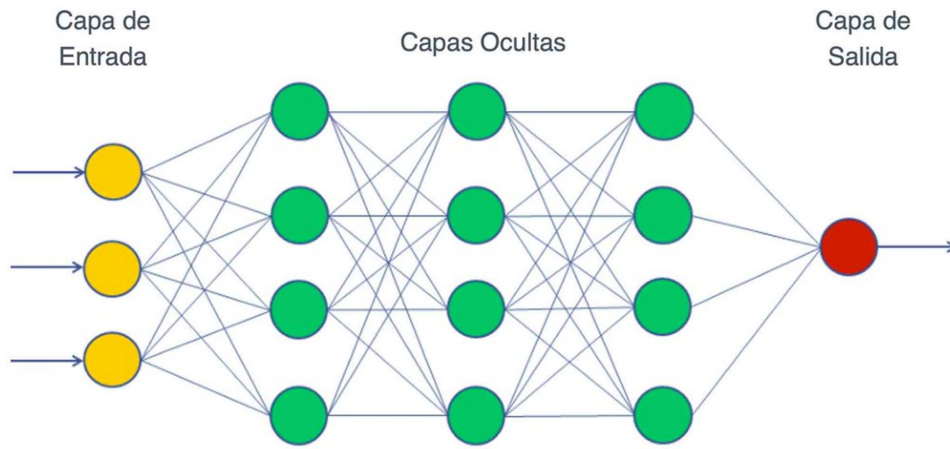


Figura 1.12: Modelo de perceptrón multicapa

Una vez que el algoritmo ha sido entrenado, es capaz de analizar un objeto en base a lo que ha aprendido anteriormente y asociarle una etiqueta.

Algunos ejemplos de algoritmos de este tipo de aprendizaje son las redes neuronales, las máquinas de soporte vectorial, los clasificadores bayesianos y los árboles de decisión, entre otros.

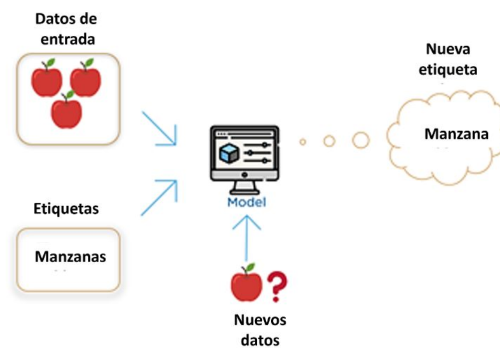


Figura 1.13: Aprendizaje supervisado

Dependiendo del tipo de problema y del tipo de etiqueta, nos encontraremos ante dos modelos distintos: clasificación o regresión.

El modelo de clasificación producirá como salida una etiqueta dentro de un conjunto finito de posibilidades. Cada posibilidad representa un tipo de etiqueta en el que se ha dividido el problema en cuestión. Puede ser binario si el conjunto de respuestas posibles son 2 (por ejemplo, si un correo es SPAM o no). También puede ser multiclase, lo cual implica clasificar más de 2 opciones (por ejemplo, reconocimiento de emociones).

Por otro lado, el modelo de regresión se centra en producir una salida numérica real ante un determinado problema. Por ejemplo, en función del mes en el que nos encontremos y de la temperatura, se puede estimar cuántos aires acondicionados se venderán.

- Aprendizaje no supervisado: Este tipo basa su aprendizaje en la obtención de patrones en los datos,que, a diferencia del anterior aprendizaje, no se encuentran etiquetados.

  Este algoritmo analiza y agrupa en clústeres los conjuntos de datos de los que dispone en la fase de entrenamiento sin etiquetar. La forma que tiene de agruparlos es buscando patrones ocultos o similitudes.

  Una vez entrenado correctamente será capaz de asociar el nuevo dato de entrada a un determinado clúster.

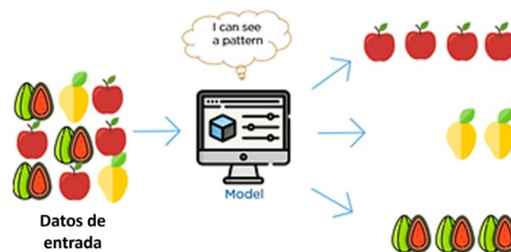


Figura 1.14: Aprendizaje no supervisado

  Ejemplos de algoritmos de este tipo de aprendizaje son, K-Means, PCA, etc.

- Aprendizaje semisupervisado: El aprendizaje semisupervisado, como indica su nombre, es un tipo de aprendizaje que mezcla conceptos de los dos anteriores. Utiliza una pequeña muestra para etiquetar los datos y luego utiliza dicho etiquetado para entrenar un nuevo algoritmo no supervisado. De esta manera, se puede mejorar considerablemente la precisión del aprendizaje.

  Este tipo de aprendizaje se usa ampliamente cuando resulta complicado obtener un conjunto de datos etiquetados lo suficientemente grande. Los ejemplos más conocidos son Autoencoders y TSVM.

- Aprendizaje por refuerzo: Este tipo de aprendizaje se basa en obtener una salida de la red neuronal y realizar las operaciones necesarias para maximizar la recompensa acumulada. Esta recompensa se obtiene a través de una señal de refuerzo denominada *fitness*.

El aprendizaje por refuerzo recibe un dato desconocido y sin etiquetar como entrada, y en función de la respuesta obtenida por el algoritmo, se calificará de una manera u otra. Esta calificación servirá como base para mejorar predicciones futuras.

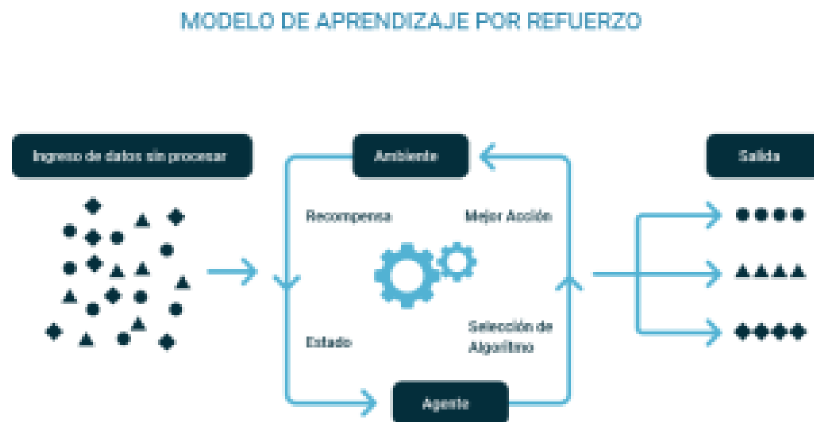


Figura 1.15: Aprendizaje por refuerzo

Este tipo de aprendizaje por refuerzo se utiliza en diversos contextos, incluyendo el aprendizaje de videojuegos y la conducción autónoma. En el aprendizaje por refuerzo, los algoritmos buscan maximizar una recompensa acumulada a través de interacciones con un entorno. Algunos algoritmos comúnmente utilizados en el aprendizaje por refuerzo son Q-Learning y algoritmos basados en procesos de decisión de Markov. Además, en la conducción autónoma, se han aplicado técnicas de NeuroEvolution of Augmenting Topologies (NEAT), que es un método de algoritmo genético para evolucionar redes neuronales.

Para la conducción autónoma se ha comprobado que un aprendizaje del tipo supervisado es posible [13]. Sin embargo, para el éxito de este aprendizaje se necesitan una gran base de datos, que contengan datos de calidad y que sea lo más variada posible para poder entrenar la red neuronal correctamente, si no es así se podría provocar un *underfitting*, que generaría en un comportamiento nefasto de la red.

Para evitar este problema, este proyecto se enfocará mediante el uso de un aprendizaje de tipo reforzado, dejaremos que sea el propio algoritmo el que vaya entrenando correctamente sin tener la necesidad de disponer de una gran base de datos.

1.3 Reinforcement learning: objetivos y algoritmos principales

El aprendizaje por refuerzo,como se explicó en el apartado anterior,es una de las diferentes formas que tiene un algoritmo de aprender para determinar sus acciones futuras.

1.3.1. Historia

Este tipo de aprendizaje está inspirado en la psicología conductista propuesta por el psicólogo estadounidense Burrhus Frederic Skinner en 1948 mediante el experimento de "La superstición de la paloma"[14][15], en el cual se utilizaron palomas como ejemplo. El estudio comenzó con palomas hambrientas, las cuales recibían comida a intervalos regulares, independientemente de su respuesta ante esta situación.

Figura 1.16: La superstición de la paloma

Pasado un tiempo, Skinner observó que cada una de las palomas adoptaba un comportamiento evidente y diferente. Este comportamiento de las aves se debía a que creían que realizando dichos actos provocaría la aparición de comida, aunque no era así. Skinner ofreció una explicación argumentando que este fenómeno era consecuencia directa del último comportamiento realizado por el pájaro antes de recibir la comida, el cual reforzaría posteriormente la idea de que si realizaba ese comportamiento recibiría la recompensa. Este procedimiento es de condicionamiento clásico, similar al que realizó Iván Pávlov con los perros y la salivación [16].

Figura 1.17: Experimento condicionamiento clásico Pávlov

1.3.2. Objetivos

Este tipo de aprendizaje tiene como objetivo tomar ciertas decisiones en función de lo que le genere una mayor recompensa. Este tipo de lenguaje máquina equivaldría a un infante aprendiendo una nueva tarea, por la que si lo hace correctamente será recompensado y si lo hace incorrectamente será penalizado.

Dentro de este aprendizaje aparecen diversos conceptos para comprender mejor el algoritmo. Estos son:

- Agente: se trata de un componente software. Es el encargado de tomar decisiones inteligentes e interactuar con el entorno mediante la realización de acciones. Dependiendo de la acción realizada, el agente recibe un tipo de recompensa u otro. En el caso del presente proyecto, el agente correspondería al coche.

- Entorno: este concepto esta relacionado con el ambiente donde interactúa el agente. El entorno establece las limitaciones y las reglas posibles a cada momento. Puede ser real o simulado. En este caso, será un entorno simulado ya que lo implementaremos a través de Pygame.

- Acciones: es el conjunto de movimientos que el agente puede realizar en un determinado momento en el entorno. En nuestro caso, se refieren a la velocidad y el ángulo de giro.

- Estado: representa la posición del agente en un determinado instante de tiempo. Cuando el agente realiza una acción, el entorno proporciona una recompensa y un nuevo estado.

- Transición: es el paso o cambio de un estado a otro.

- Probabilidad de transición: es la posibilidad que tiene un agente para elegir el próximo estado en función del estado en el que se encuentra.

- Recompensa: puede ser positiva o negativa (en este caso sería castigo). Gracias a este concepto, podemos guiar al agente y determinar si está desempeñándose bien o mal.

En el caso en que nos encontramos, el agente partirá de un estado inicial y tomará una acción, lo cual influirá en el entorno. En la siguiente iteración, el ambiente devolverá al agente un nuevo estado con la recompensa obtenida, para el caso de que la recompensa sea positiva, se estará reforzando este comportamiento de cara al futuro, en caso de una recompensa negativa, se estará penalizando y el agente buscará una manera alternativa de actuar.

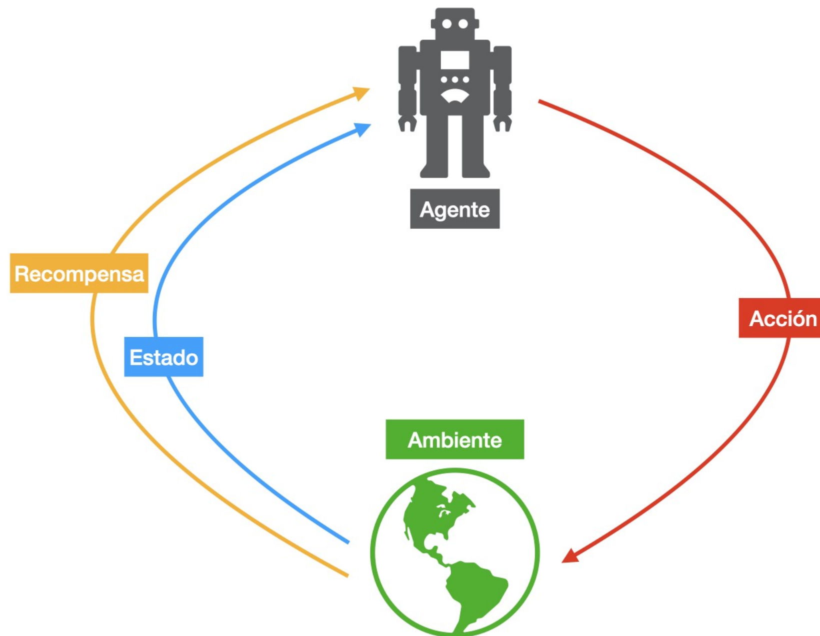


Figura 1.18: Diagrama de flujo aprendizaje reforzado

Este esquema, en el que se apoya el aprendizaje reforzado, es el proceso de decisión de Markov[17]. *"Los estados futuros son dependientes del presente pero independientes de los estados pasados"*

Esta referencia se puede expresar matemáticamente de la siguiente forma:

$$P[S_{t+1}|S_t] \leftarrow P[S_{t+1}|S_1, ..., St]$$

Donde:

- S_1 se designa al estado actual.
- S_{t+1} corresponde al siguiente estado.
- $S_1, ..., St$ corresponden a los estados tomados hasta llegar al estado actual.

Gracias a que únicamente se necesitará conocer el estado actual para la decisión se reduciría drásticamente el coste computacional necesario para obtener el siguiente estado.

1.3.3. Algoritmos principales

Dentro de este tipo de aprendizaje, existen multitud de enfoques que pueden ser usados para resolver de manera óptima los problemas. Algunos se resolverán mediante los procesos de decisión de Markov y otros mediante algoritmos genéticos. A continuación, se detallará un algoritmo de cada tipo: Q-Learning y NEAT, respectivamente.

1.3.3.1. Q-Learning

Este tipo de aprendizaje reforzado, se basa en la existencia de una tabla denominada Q-tabla, en la que se almacena la recompensa esperada si el agente realiza una determinada acción en un estado concreto. El entorno está dividido por una serie de casillas, donde el objetivo es maximizar la recompensa (el valor Q), esta se obtiene en función del siguiente estado que alcance. Dependiendo del estado al que se mueva y con la acción que realice, obtendría una recompensa diferente. Esta recompensa se actualizará siguiendo la siguiente regla:

$$Q(s,a) = r(s,a) + \gamma max_a * Q(s^{'},a)$$

Donde:

- s se designa al estado actual.
- $s^{'}$ corresponde al siguiente estado.
- $r(s,a)$ denota la recompensa inmediata recibida por la realización de la acción a
- γ es el factor de descuento que controla el efecto de la recompensa para un futuro distante
- max_a representa la recompensa máxima posible en el siguiente estado

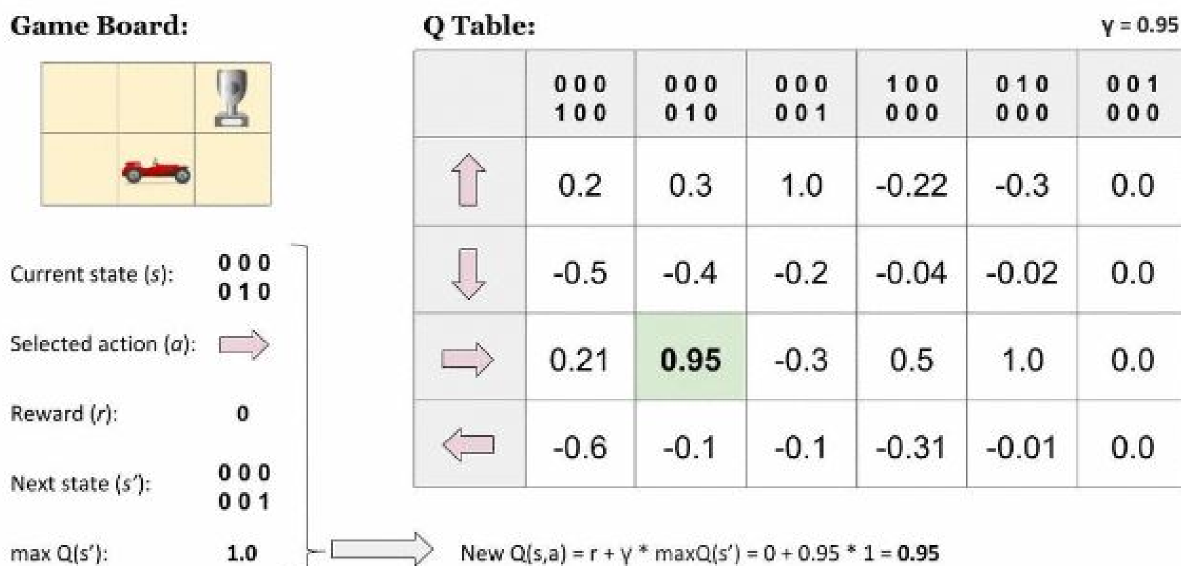
Como se puede apreciar, el valor Q dependerá del valor Q del estado futuro, con lo que el valor de Q del estado s resultará ser:

$$Q(s,a) \leftarrow \gamma Q(s^{'}),a) + \gamma^2 Q(s^{''}),a) + \gamma^n Q(s^{''\cdots n}),a)$$

Donde el valor $\gamma \in [0,1]$. Si $\gamma = 1$, entonces todas las recompensas futuras tendrán una contribución a la recompensa total. En el caso opuesto, si $\gamma = 0$, no se consideran las recompensas futuras. Se comienza el proceso de aprendizaje utilizando valores arbitrarios y, después de aprender de las recompensas positivas y negativas, se busca converger hacia un óptimo.

1.3.3.2. NEAT

NEAT,acrónimo de NeuroEvolución de Topologías Aumentada, es un método evolutivo. Dicho algoritmo evolutivo utiliza redes neuronales artificiales, comenzando con unas redes neuronales simples y, a lo largo de realizar generaciones, se irá modificando dicha red, aumentando la complejidad y aumentando también las conexiones entre ellas. Debido a este aumento de dificultad son conocidos como un algoritmo de complejización.



	000 100	000 010	000 001	100 000	010 000	001 000
⇧	0.2	0.3	1.0	-0.22	-0.3	0.0
⇩	-0.5	-0.4	-0.2	-0.04	-0.02	0.0
⇨	0.21	**0.95**	-0.3	0.5	1.0	0.0
⇦	-0.6	-0.1	-0.1	-0.31	-0.01	0.0

Figura 1.19: Ejemplo gráfico Q-Tabla

El principal beneficio de este tipo de algoritmos es que al ir incrementando la red neuronal lentamente, evita que su comportamiento cambie de forma drástica.

En la implementación de NEAT, usando el lenguaje de programación de alto nivel Python, mantendremos una población de individuos, donde cada uno de los individuos contendrá dos lista de genes, la primera se denominará ´´genes de nodo” y la segunda ´´genes de conexión”.

Mientras que los genes de nodo identifica a una sola neurona, los genes de conexión identifica la conexión entre dos neuronas, especificando la dirección y el sentido de la conexión, el peso de la misma, si se encuentra habilitada y el número de innovación, que nos permitirá encontrar los genes correspondientes durante el cruce.

La diferencia entre ambas soluciones reside en que la salida del Aprendizaje Reforzado es una política, con su estrategia y su algoritmo, mientras que la salida de NEAT es una red neuronal.

Para este proyecto nos decantaremos por la utilización de NEAT ya que esta funciona bien para escenarios de control simples, con una red de políticas que genera determinadas acciones según las entradas de los sensores. También nos decantaremos por NEAT debido a que a priori, es un problema fácil de evaluar, ya que la medición del rendimiento lo catalogaremos en funcíon de la distancia recorrida[18].

1.4 NEAT Operación, funcionamiento y características principales

Para comprender correctamente qué es el algoritmo NEAT, es importante destacar que se trata de un algoritmo de tipo genético (GA) utilizado para generar redes neuronales artificiales (ANN) mediante evolución (lo que corresponde a la técnica conocida como neuroevolución).

A continuación, se irán describiendo los conceptos mencionados arriba y subrayados.

1.4.1. Neuroevolución

La neuroevolución parte del campo de la computación evolutiva. Esta constituye una rama del aprendizaje máquina y se utiliza para encontrar soluciones a problemas en los que los espacios de búsqueda son extensos y no lineales [19]. Los algoritmos evolutivos (EA) se basan en la evolución biológica de las especies, publicada por Charles Darwin en 1859. En ella se introdujeron teorías científicas como que las poblaciones evolucionan a lo largo de generaciones mediante un proceso conocido como selección natural. Este tipo de computación evolutiva se aplica comúnmente en los sectores de vida artificial, robótica evolutiva y juegos. En el presente proyecto lo utilizaremos para el sector de juegos y vida artificial, ya que el principal beneficio de los algoritmos evolutivos es su aplicación más amplia en comparación con los algoritmos que utilizan un aprendizaje de tipo supervisado.

Las EA utilizan los mismos mecanismos que publicó Darwin. Estos son:

1. Reproducción
2. Mutación
3. Cruce
4. Selección

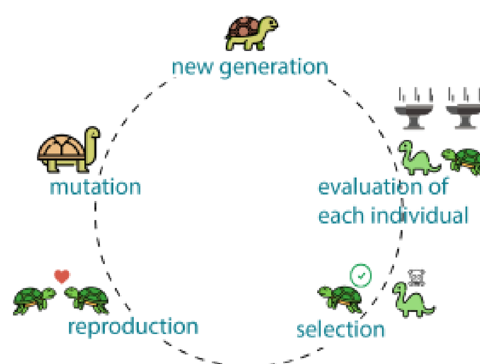


Figura 1.20: Evolución biológica de una especie

Una de las características que poseen los EA, son que se mantienen un conjunto de entidades, los cuales representan las posibles soluciones, estos se cruzan entre si y compiten de tal forma que solo los que obtengan los mejores resultados prevalecerán en el tiempo. Esta evolución constante provocará que se obtengan mejores soluciones.

En los EA, existe la misma terminología que en la de la teoría de la evolución. Estos conceptos son:

- **Individuos:**Entidades que representan las soluciones al problema.
- **Población:**Conjunto de individuos.
- **Cruce:**Mezcla de información de dos o más cromosomas.
- **Mutación:**Cambio aleatorio en algún cromosoma.
- **Selección:**Elección de los cromosomas que prevalecerán en la siguiente generación.

Dentro de los EA existen varios algoritmos que tienen como base esta idea de los EA, estos algoritmos son:

- **Algoritmos Genéticos (AG):** Este algoritmo codifica el conjunto de individuos de una población en cromosomas (cadenas binarias) y evoluciona a través de iteraciones. Los individuos son evaluados según una función de adaptación (*fitness*). Si el objetivo del programa es maximizar los resultados, los cromosomas con mejores valores de *fitness* se cruzarán entre sí.
- **Programación Evolutiva (PE):** Esta estrategia de optimización, similar a los Algoritmos Genéticos, modifica la representación de los individuos. Los individuos se representan mediante ternas, que consisten en el valor actual, un símbolo del alfabeto utilizado y el valor del nuevo estado.
- **Estrategias Evolutivas (EE):** Este algoritmo evolutivo realiza la recombinación y selección de individuos de manera imparcial y determinista. Utiliza vectores de números reales para codificar las posibles soluciones de un problema numérico en particular.

De estas 3 paradigmas, elegiremos para realizar este proyecto el que corresponde a los AG, este será detallado a continuación.

1.4.2. Algoritmos genéticos

Este tipo de algoritmo consiste en una heurística inspirada en la teoría de la selección natural. El proceso de selección se inicia con la elección de individuos con mejores aptitudes de una población; posteriormente, se mezclarán y producirán descendientes que heredarán las características de los padres y formarán parte de una nueva generación.

El objetivo de heredar las características de los padres es claro: tener una mayor probabilidad de sobrevivir.

Dicho proceso se realiza de forma reiterativa hasta alcanzar una generación que cumpla con los objetivos de aptitud más altos posibles

Dentro de los AG aparecen 5 etapas[20], de las cuales comparte varias con los EA:

- **Población inicial:** Este proceso comienza con un conjunto de individuos, conocidos como po-

blación inicial. Cada individuo representa una solución del problema a resolver. Cada individuo contiene una serie de cromosomas, los cuales albergan un conjunto de variables de tipo booleano denominadas genes.

- **Función *fitness*:** Es una función que clasifica a cada uno de los individuos de la población. Esta clasificación se realiza para cuantificar numéricamente los resultados obtenidos en la generación de cada individuo. Luego, se utiliza esta clasificación para determinar qué individuo es el más apto y reproducirlo.

- **Selección:** En esta fase, se utiliza el valor obtenido en la función *fitness* para establecer qué individuo tiene mejores genes y permite que pasen a las siguientes generaciones. A este proceso se le conoce como selección elitista.

- **Cruce o *crossover*:** Es la etapa en la cual dos padres seleccionados previamente intercambian genes en un determinado cromosoma, lo que resulta en la creación de una nueva descendencia.

- **Mutación:** Es otra forma de generar nueva descendencia. En este caso, no depende de los progenitores, ya que es una variación espontánea en un determinado gen. La probabilidad de que esto ocurra se determina en el algoritmo. Esta etapa es fundamental, ya que evita que el algoritmo converja prematuramente.

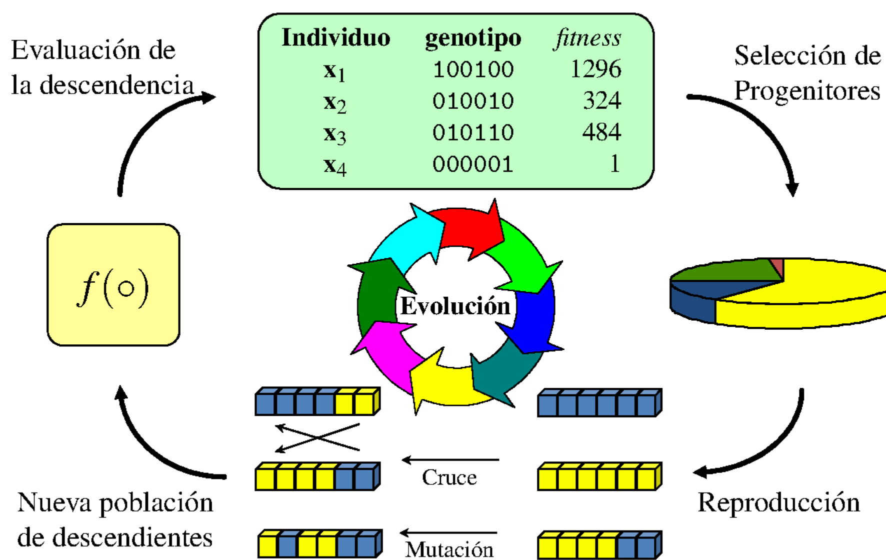


Figura 1.21: Ejemplo genético

1.4.3. NEAT

1.4.3.1. Historia

Este algoritmo fue desarrollado en el año 2002 por Kenneth O. Stanley y Risto Miikkulainen en la Universidad de Texas. En términos generales, propusieron la implementación de un algoritmo genético para evolucionar poblaciones en redes neuronales, con el objetivo de llegar a una solución óptima. Estas modificaciones se realizaban tanto en los pesos como en la topología de las redes.

El objetivo de NEAT también es mantener el tamaño de las redes lo más pequeño posible. Para lograr esto, comienza la evolución utilizando poblaciones con topologías sencillas. Luego, agrega conexiones y nuevas neuronas a lo largo de la ejecución. Este proceso es transparente para el usuario en esta librería.

Existen diversos métodos para codificar y representar la topología en una red neuronal. En el caso de NEAT, se opta por utilizar una codificación binaria o directa, ya que es necesario rastrear la evolución de la topología en la red. Si se utilizara una codificación indirecta, podría haber una pérdida de información en la topología al combinar individuos.

1.4.3.2. Codificación

En la codificación directa, el número de nodos, conexiones y pesos de un genotipo se representan literalmente. Cada enlace se detalla mediante un parámetro “KEY”. Si un parámetro KEY tiene un valor de (-1, 0), significa que habrá una conexión directa entre el nodo -1 y el nodo 0 [21].

1.4.3.3. Convenciones competidoras

Este concepto significa tener más de una manera de representar una misma solución al problema de optimización de pesos en la red neuronal. Esto se produce cuando los genomas no tienen la misma codificación. El cruzamiento es el responsable de producir descendientes defectuosos. NEAT aboga por realizar sinapsis artificiales basadas en los marcadores históricos, de manera que se pueda modificar la topología sin perder la pista de los genes en la simulación.

1.4.3.4. Mutaciones

Respecto a este apartado, existen tres tipos diferentes de mutaciones, algunas afectan a la tipología de la red (estructurales) y otras no (no estructurales).

La primera de ellas corresponde a la mutación en los pesos. Esta mutación es no estructural, ya que solo modifica el valor del peso en las conexiones. La probabilidad de que esto suceda estará predefinida en el archivo de configuración.

La siguiente a mencionar es la relativa a los nodos. En este caso sí es estructural, ya que se añaden nodos. Cuando se agrega un nodo, a este se le asigna un peso de 1 para que no tenga tanta afectación en el sistema. Cuando se agrega un nodo entre dos nodos, se tiene que tener en cuenta que el enlace que figuraba antes entre ambos nodos debe ser eliminado para poder construir dos enlaces: uno del nodo origen al nodo creado y otro del nodo creado al nodo destino.

Por último, está la mutación en enlaces. En este tipo de mutación se produce la creación o eliminación de los enlaces entre los nodos de los genomas, con lo cual, al igual que en las mutaciones de los nodos,

afecta a la tipología de la red.

1.4.3.5. Reproducción

Parte del éxito del algoritmo reside en respetar la misma estructura en la fase de reproducción entre dos genes, aunque existe la posibilidad de que haya una modificación en los pesos debido a que son derivados del mismo gen ancestro en algún punto de generaciones pasadas. Al ser codificación directa, rastrear el origen histórico de un gen requiere muy poca computación, es aquí cuando al nuevo gen que se le incrementa el número de innovación global.

Para realizar una recombinación entre genomas es necesario alinear conforme al numero de innovación, considerando únicamente los genes de las conexiones. Puede llegar a ocurrir que un genotipo no disponga de ese número de innovación en concreto, en ese caso se verá si está dentro de un rango determinado, si lo está se considerará una desarticulación y si no lo está se considerará un exceso.

Cuando se conforma un genoma hijo, los genes son seleccionados de manera aleatoria por cualquiera de los padres, siempre y cuando se encuentren alineados. Para el caso de que no lo estén, serán incluidos los del padre que se considere más apto y, si por un casual, no hay un padre más apto que el otro, este se realizará de forma aleatoria.

1.4.3.6. Especiación

Es frecuente, que al agregar nodos a una red, este baje su desempeño, ya que, como se ha comentado antes, se establece por defecto un peso igual a 1. Es improbable que, al añadir un nuevo nodo, mejore en esa generación las prestaciones, por ello es necesario dejar unas generaciones de margen para que los pesos se vayan optimizando. El problema que ocurre al bajar el desempeño es que es probable que este individuo no sobreviva, por ello es necesario proteger de alguna manera la estructura de la red, para dar tiempo a que se optimice.

De esta idea nace agrupar por especies las redes,de manera que con esa agrupación, se permitirá a una red obtener el tiempo necesario para posteriormente competir con el grueso de la población.

Esta especiación requiere de una función de compatibilidad, la cual nos permita distinguir si dos genomas se deben de encontrar en la misma especie o no.

Esta compatibilidad se obtiene mediante el cálculo de la distancia de compatibilidad, expresada matemáticamente de la siguiente forma:

$$d = c_1 * (E/N) + c_2 * (D/N) + c_3 * avg(W)$$

Donde

- d: es la distancia de compatibilidad.
- E: es el número de genes que se encuentran en exceso.
- D: se corresponde con el número de genes que se encuentran desarticulados.

- W: es el peso de la red.
- c_i: coeficiente que nos sirve para ajustar la importancia de cada factor.
- N: es el número de genes que tiene el genoma más grande.

Una vez obtenida la distancia de compatibilidad, podremos compararla contra un umbral de compatibilidad definido en el archivo de configuración. Si ese genoma no es compatible con ningún representante de las especies definidas, se creará una nueva especie.

Existe un método llamado "*fitness sharing*" que vela porque una especie no tome todo el control del entrenamiento; para ello, penalizará a aquellas especies que dispongan de muchos individuos.

1.4.4. Población inicial

Como se ha comentado antes, NEAT, a diferencia de otros algoritmos de neuroevolución, busca minimizar las redes. Para ello, establece los individuos de su población inicial con pocas neuronas, las cuales están interconectadas entre sí sin tener capas de neurona ocultas.

1.4.5. Parámetros de configuración

Los parámetros de configuración se encuentran recogidos en un archivo de configuración con formato de texto. En él, se pueden variar los algoritmos NEAT para obtener distintas configuraciones.

En este documento, se pueden modificar las probabilidades de mutaciones (ya sean de nodos, enlaces o modificaciones en el peso), los valores de los pesos, el número de neuronas (entrada, ocultas y de salida), establecer un umbral de supervivencia o establecer el elitismo, entre otras opciones.

1.5 Entorno de desarrollo y herramientas

Para el desarrollo de este proyecto, se han utilizado diversas herramientas. A continuación, se detallará los software y librerías elegidas para realizar el programa:

- Python: es un lenguaje de programación de alto nivel de código abierto. Es altamente popular debido a su gran versatilidad para realizar proyectos de distintos ámbitos, como desarrollo de páginas web, desarrollo de aplicaciones e inteligencia artificial.

- Pygame: es una biblioteca multiplataforma escrita en Python para desarrollar videojuegos en 2D.

(a) Juego genérico de plataformas

(b) Wolfenstein

Figura 1.22: Ejemplos de juegos realizados con Pygame.

- Neat-python: es una implementación en Python del algoritmo de ”Neuroevolución de las topologías de aumento”desarrollado por Kenneth O. Stanley para evolucionar redes neuronales arbitrarias.

- Overleaf: es una herramienta en línea para la elaboración de documentos científicos utilizando el sistema de composición de textos conocido como LaTeX.

- Visual Studio Code: es un editor de código desarrollado por Microsoft. Es muy popular debido a que tiene un entorno de trabajo altamente personalizable. Se pueden instalar extensiones que ayudan al programador, como la comparación de dos archivos en busca de similitudes y diferencias, o la extensión ***”TODO TREE”***, que permite realizar anotaciones sobre implementaciones o modificaciones futuras en el código.

- GIMP: es un programa libre y gratuito de edición de imágenes en forma de mapa de bits. Se utiliza para el dibujo de los circuitos empleados en este proyecto.

Capítulo 2

Trabajos relacionados

Para la realización de este trabajo final de estudios, se ha llevado a cabo una búsqueda de proyectos relacionados con la conducción autónoma en los que se aplica el protocolo NEAT, tanto académicamente como fuera de este campo.

A continuación, se detallarán los trabajos encontrados relacionados con la conducción autónoma.

2.1 Trabajos académicos

Dentro del mundo académico alojado en el motor de búsqueda de Google, que se centra en contenido y bibliografías científico-académicas, se han encontrado diversos proyectos relacionados con la conducción autónoma.

Principalmente, estos se centran en la comparación entre la realización de este proyecto utilizando aprendizaje por refuerzo y la utilización de algoritmos genéticos. Una vez concluidos, llegaron a la conclusión de que sería muy beneficioso para el algoritmo combinarlos, de manera que se utilizara NEAT para acelerar la fase inicial. De esta manera, se alcanzaba una generalización de varios órdenes de magnitud en comparación con el uso de técnicas como la retropropagación (que requieren grandes cantidades de conjuntos de datos para generar buenas soluciones). Sin embargo, una vez que la red alcanza ciertos niveles de habilidades cognitivas para la conducción, se podría utilizar un aprendizaje por refuerzo como Deep Q-learning [22] [23].

Con esta combinación, podemos observar que el código se puede dividir en dos fases:

- **Fase inicial:** Utilizada para la obtención de experiencia por parte del vehículo, en la que el agente tiene como objetivo principal no ser castigado por las acciones cometidos.

- **Fase de recompensas:** Utilizada para obtener la mayor cantidad de recompensa posible.

Por otro lado, también se hace hincapié en que todas las pruebas se han llevado a cabo en un entorno simulado, por lo que es posible que no funcione con la misma eficacia en un entorno real, puesto que no integra los cambios de carriles ni las señales de tráfico. Para integrarlos en un proyecto, se deberá desarrollar un sistema en paralelo que tenga en cuenta esos factores. Una de las posibles soluciones es la instalación de cámaras en el vehículo y, con ellas, poder distinguir los carriles y las señales [24] [25].

2.2 Desarrollos relacionados

En la búsqueda de trabajos relacionados con esta tecnología, se ha encontrado un amplio abanico en GitHub. Este sistema de control de versiones (GIT) permite gestionar proyectos, alojar páginas web y alojar código.

En el próximo apartado se detallarán las distintas tecnologías y lenguajes de programación más utilizados en relación con la conducción autónoma y los algoritmos genéticos.

2.2.1. Python

Para esta tecnología, hay multitud de proyectos alojados en GitHub. También se encuentran alojados en la web Replit, donde se puede ejecutar directamente el código y visualizar los resultados.

Estos proyectos han servido de base para la consecución del trabajo de fin de estudios realizado por mí. Tal y como se explicará en el apartado de Pruebas, uno de los trabajos contenía un estudio sobre el número óptimo de sensores en este tipo de problemas.

El código realizado por NeuralNine [26], mostraba una modificación de los códigos anteriores. Dicha modificación consistía en establecer una limitación de velocidad máxima, lo que resultaba en mejores resultados, ya que requería menos generaciones para realizar una vuelta completa.

Otro de los algoritmos a mencionar se trata del propuesto por Jackie Barman [27]. El código de este proyecto nos servirá de base para realizar el que nos atañe, ya que se ha elegido debido a que servía de base para futuros trabajos. Otra de las ventajas principales de este proyecto residía en que estaba adaptado para dos salidas: una para girar hacia la izquierda y otra para girar hacia la derecha. Esto se modificará para unificar ambas salidas y que, en función del valor numérico, decida la dirección del giro.

2.2.2. Trackmania

Uno de los proyectos utilizaba el protocolo NEAT para realizar un aprendizaje en conducción autónoma utilizando el videojuego Trackmania. En él, se podía observar cómo se realizaba una simulación con un vehículo que intentaba recorrer la mayor distancia posible antes de ser eliminado.

Este videojuego permite la creación de mapas por parte del usuario, lo que permite entrenar a un algoritmo de manera semi-realista. La integración entre el videojuego y las instrucciones se llevaba a cabo mediante la librería Openplanet[1].

Gracias a esta librería, podemos introducir instrucciones al coche generado. Esto se puede lograr utilizando los datos recopilados por la API de la librería o también realizando cálculos en función de los datos obtenidos directamente desde la cámara, utilizando la librería de Python OpenCV.

Openplanet es una libreria escrita en C/C++ que sirve de pasarela entre la aplicación y el cliente, los cuales están conectados utilizando *sockets*.

```
auto sock_serv = Net::Socket();
if (!sock_serv.Listen("127.0.0.1", 9000)) {
    print("Could not initiate server socket.");
    return;
}
print(Time::Now + "Waiting for incomming connection...");
```

Figura 2.1: Configuración de los sockets del servidor

El objetivo de esta librería es convertir al juego en un servidor y que el cliente pueda establecer conexión con el, pudiendo recoger los datos obtenidos en el juego a través de la API o bien inyectando código en el juego.

Para que el vehículo pueda moverse se utilizará un mando de videojuegos virtual, para ello es necesario instalar el controlador para la virtualización de juegos conocido como *Nefarius*.[2]. Una vez instalado ya podríamos mandar al juego instrucciones de movimiento.

Como hemos mencionado anteriormente, este proyecto se puede enfocar desde dos perspectivas diferentes: según los datos obtenidos por la API y los obtenidos por la cámara que se encuentra en la parte superior del vehículo.

[1] https://openplanet.dev/
[2] https://vigem.org/projects/ViGEm/How-to-Install/

2.2.2.1. API

Una vez establecida la conexión entre el juego y el cliente remoto (en este caso Python), podemos empezar a enviar datos para analizarlos utilizando los *sockets* (agrupación IP y puerto). Los datos con mayor relevancia dentro de la API son la posición del coche y los valores de aceleración y giro.

```
while(cc)
{
    CTrackMania@ app = cast<CTrackMania>(GetApp());
    CSmArenaClient@ playground = cast<CSmArenaClient>(app.CurrentPlayground);
    CSmArena@ arena = cast<CSmArena>(playground.Arena);
    auto player = arena.Players[0];
    CSmScriptPlayer@ api = cast<CSmScriptPlayer>(player.ScriptAPI);

    auto race_state = playground.GameTerminals[0].UISequence_Current;

    // Sending data
    cc = send_data_float(sock, api.Speed);
    send_data_float(sock, api.Distance);
    send_data_float(sock, api.Position.x);
    send_data_float(sock, api.Position.y);
    send_data_float(sock, api.Position.z);
    send_data_float(sock, api.InputSteer);
    send_data_float(sock, api.InputGasPedal);
    if(api.InputIsBraking) send_data_float(sock, 1.0f);
    else send_data_float(sock, 0.0f);
```

Figura 2.2: Obtención de los datos de la API

El siguiente paso es la aplicación del algoritmo NEAT. Para ello, el primer paso consistirá en realizar una captura de pantalla mediante la librería *"ImageGrab"*. Posteriormente, se aplicarán distintas funciones para detectar dónde se encuentran los muros, que junto a la función de activación de la red neuronal, dará unos valores de salida (dirección, gas y freno).

Una vez obtenido los valores, los incluiremos en el mando virtual para que sean trasladados al juego y el vehículo adapte la dirección y la velocidad, con el fin de obtener mejores resultados.

```
    # screenshot
    img = ImageGrab.grab()

    img = mod_shrink_n_measure(img, image_width, image_height, no_lines)

    try:
        img = img / 255.0
    except:
        img = img

    # speed
    speed = data['speed']*3.6
    distance = data['distance']

    img.append(speed)

    # net
    output = np.array(net.activate(img))

    brake = output[2]
    gas = output[1]
    steer = output[0]

    steer = steer * 2 - 1

    gamepad.left_joystick_float(x_value_float=steer, y_value_float=0)
    gamepad.right_trigger_float(value_float=gas)
    gamepad.left_trigger_float(value_float=brake)
    gamepad.update()

    # stop = time.time()
    # print(stop - start) # reaction time
    finish = time.time()-begin
    if finish > no_seconds or (finish > kill_seconds and speed < kill_speed):
        gamepad.reset()
        gamepad.update()
        break

genome.fitness = distance
```

Figura 2.3: Código NEAT en Trackmania

2.2.2.2. OpenCV

Para el caso de querer realizar entrenamiento con los datos de la cámara,usaremos una biblioteca libre de visión artificial originalmente desarrollada por Intel denominada OpenCV (Open Computer Vision).

El mecanismo es muy similar al utilizado en el apartado anterior, ya que se siguen los siguientes pasos:

- Obtener captura de pantalla de Trackmania.
- Procesado de imagen usando OpenCV.
- Enviar los datos de vuelta a Trackmania.

Para el procesado de imagen, se utilizará la función de Canny [28].

Una vez obtenido el mapa procesado, el siguiente paso será calcular la distancia entre el centro del vehículo y los limites de pista. Para ello, se utilizarán los píxeles.

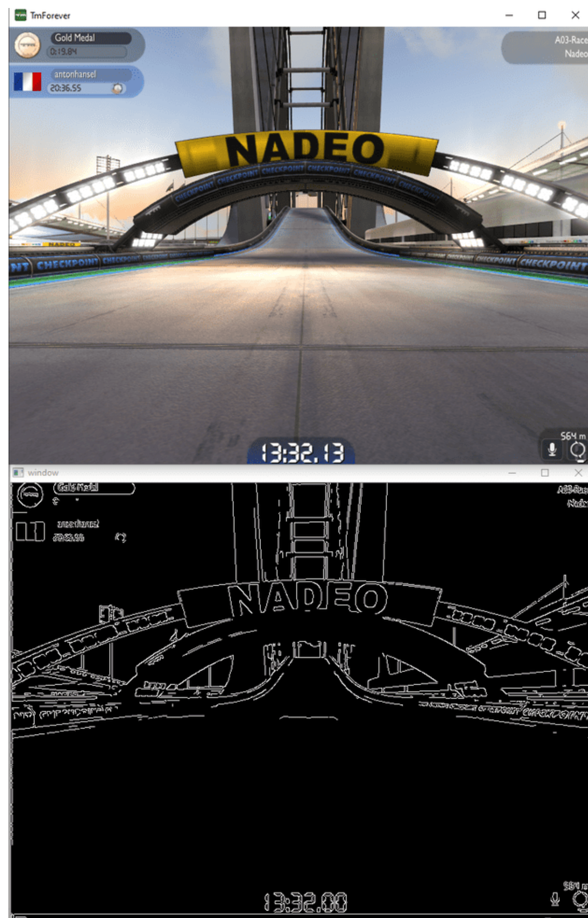


Figura 2.4: Aplicación Canny

Posteriormente, aplicaremos esos valores a la función de activación de la red neuronal. Finalmente, incluiremos esos valores en el mando virtual y enviaremos los datos de vuelta al juego para que pueda continuar con el entrenamiento.

2.2.3. JavaScript

Utilizando el lenguaje de programación orientado a objetos JavaScript también se han encontrado proyectos relacionados con la conducción automática. Usando este lenguaje de programación orientado a objetos, se han encontrado varios ejemplos que han implementado el algoritmo de NEAT sin necesidad de incluir librerías externas. Para ello es necesario dividir el código en una serie de archivos para hacer más manejable el proyecto.

- **Main.js:** Este es el archivo principal del proyecto, en él se establecerán las posiciones que ocupan en el circuito los obstáculos, representados mediante coches y se llevará a cabo las animaciones. La forma de almacenar la mejor generación se realiza utilizando la propiedad de JavaScript *"localStorage"*, esta propiedad permite acceder a datos almacenados en la sesión del navegador.

- **Controls.js:** En este fichero se detalla aquellos movimientos que puede realizar el vehículo, relacionándolas con las teclas del teclado.

- **Car.js:** En este archivo se incluyen los atributos que caracterizan al coche, como pueden ser el tamaño del vehículo, los atributos tales como ángulos y velocidades, así como la posición donde se encuentra en ese instante.

- **Road.js:** Aqui se define el circuito por el que circulará el vehículo, se ha optado por un circuito infinito en el que no se configuran curvas. De tal manera que el coche circulará toda la simulación.

- **Sensors.js:** Para la detección correcta de obstáculos, se ha implementado esta función. Al igual que realizamos en nuestro proyecto, se despliegan una serie de sensores, que nacen en el centro del vehículo, cuya finalidad es calcular la distancia que existe entre el coche y el obstáculo.

- **Networks.js:** Como este proyecto carece de librerías auxiliares es necesaria la implementación manual del algoritmo de NEAT. En este archivo se configura la red neuronal, las mutaciones que pueden aparecer y como resultado, obtenemos la salida de la red neuronal que permitirá al coche realizar la siguiente acción.

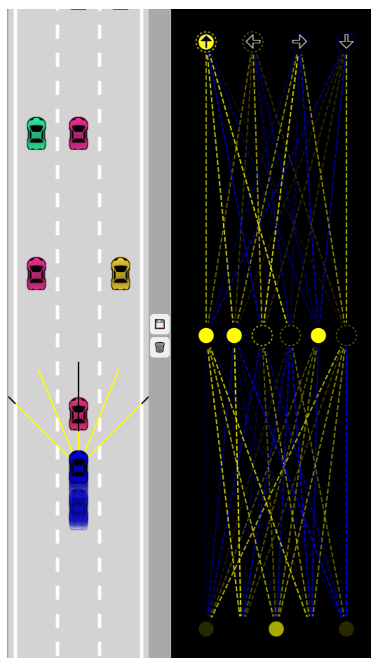


Figura 2.5: Conducción autónoma con JavaScript [29]

Por otro lado, otro de los proyectos que cabe mencionar, es el realizado por Rafael Matsunaga y David Bau [30]. En dicho proyecto se utiliza la biblioteca ”Box2D”, que contiene un motor físico en dos dimensiones que permite representar una especie de coche.

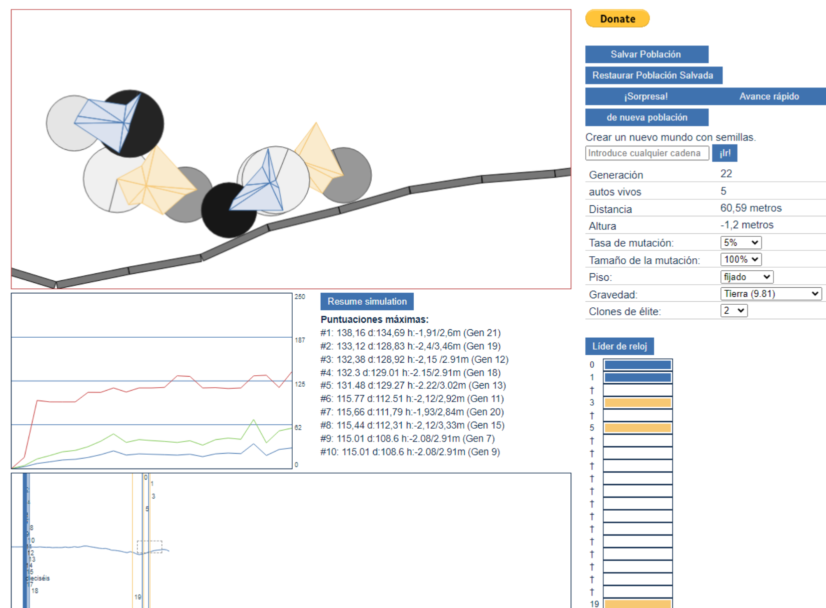


Figura 2.6: Conducción autónoma con JavaScript usando Box2D

La peculiaridad de este proyecto reside en la flexibilidad, ya que se pueden configurar varios parámetros, como la tasa de mutación o el elitismo, de forma dinámica. También posee una gráfica en la que se presentan los mejores resultados hasta la simulación actual, en la que se indica la distancia recorrida, el tiempo empleado y el número de generación en el que se han obtenido.

2.2.4. C#

Otro de los proyectos encontrados ha sido utilizando como motor UNITY.

Unity es un motor de desarrollo de videojuegos escrito en C#. Es muy popular debido a que permite a los desarrolladores trabajar con un entorno visual y proporciona herramientas y recursos como animaciones y físicas, lo cual permite una representación más realista que la que proporciona Pygame. Otra de las ventajas de este motor de videojuegos es que permite desarrollar juegos multiplataforma, capaces de ejecutarse en PCs, consolas y móviles.

Mayoritariamente, se utiliza Unity en el mundo de la conducción autónoma para realizar el entrenamiento mediante la visión por computadora, utilizando sensores de radar y LIDAR. Sin embargo, también podemos encontrar proyectos que utilizan este motor de desarrollo para realizar un entrenamiento basado en aprendizaje por refuerzo, más concretamente utilizando algoritmos genéticos. Uno de los proyectos más completos en este sentido es el realizado por el estudiante de la UPM Diego Berrocal Gutiérrez, en su proyecto “Inteligencia computacional para el guiado de vehículos autónomos” [31][32].

En el proyecto se incluye la física del vehículo (añadiendo un sistema de suspensión delantero y un sistema de frenos) y componentes sensoriales que se distribuirán según el número de sensores totales y el campo de visión 2.7.

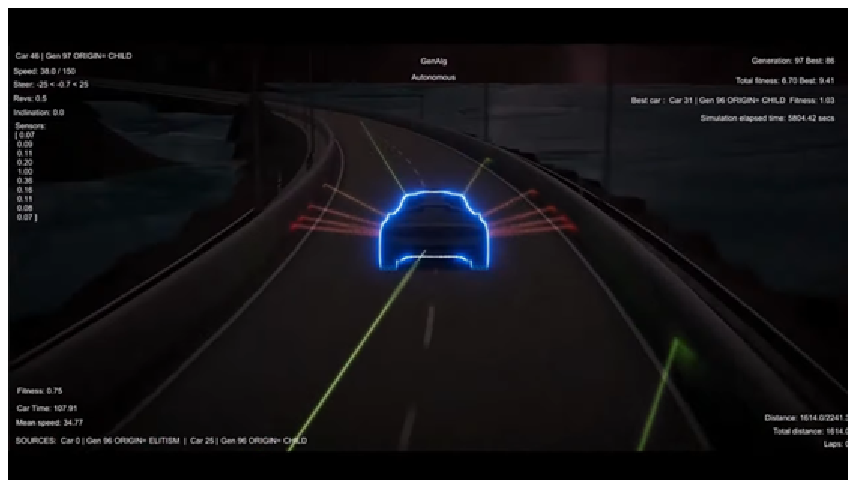


Figura 2.7: Conducción autónoma con Unity

Capítulo 3

Desarrollo y descripción del algoritmo en NEAT

3.1 El algoritmo NEAT

Como hemos visto en en el capítulo correspondiente a NEAT. NEAT pertenece al grupo de los algoritmos evolutivos que utilizan las combinaciones de codificación genética directa e indirecta para evolucionar la topologia como los pesos de las redes neuronales. La flexibilidad y la capacidad de adaptación le permiten generar topologías complejas y resolver una amplia gama de problemas sin la necesidad de conocimiento previo del dominio[33].

3.2 Adaptación del algoritmo

En el algoritmo del cual se partía,se presentaban una serie de limitaciones, las cuales se han ido subsanando durante la evolución y desarrollo del mismo. En este apartado,se detallarán las limitaciones encontradas y se presentarán las soluciones que han mejorado el rendimiento.

El principal problema que nos encontrábamos residía en que en el código original no se tenía en cuenta la posibilidad de tener una velocidad variable, independientemente de si el coche se encontraba en la entrada de una curva o en una recta larga, con lo que siempre se movería a la misma velocidad. Esta suposición claramente no es realista y se ha decidido darle al vehículo la posibilidad de ir modificando la velocidad en función de sus necesidades.

```
output = nets[index].activate(car.get_data())
if output[1] < 0 :
        car.speed -= abs(output[1])
        car.number_times_braking += 1
elif output[1] > 0:
        if car.speed <= 10:
            car.speed += abs(output[1])
            car.number_times_accelerated += 1
```

Figura 3.1: Configuración velocidad variable

Esta combinación de velocidad y ángulos de giro constantes hace que, aunque se encuentre el caso óptimo, este solo será correcto para el circuito en cuestión y no para otros.Por ejemplo, si se entrena el algoritmo en un circuito con curvas suaves, cuando se traslade el código a otro mapa con curvas cerradas, el coche se saldrá de la curva.

En relación a los mapas, se han añadido más y se han creado nuevos partiendo de la base del mapa original (utilizando el programa de edición GIMP), hasta llegar a un total de 10 mapas distintos, para realizar las pruebas en dichos circuitos. De manera que podemos confirmar si el algoritmo está funcionando bien.

Cabe añadir que se podrían crear nuevos mapas en cualquier momento y que se espera que la red pueda conducir correctamente en ellos a partir del entrenamiento realizado.

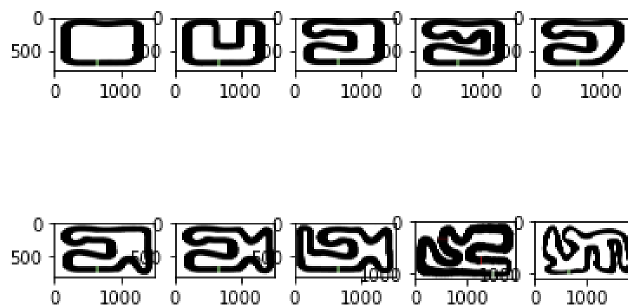


Figura 3.2: Mapas usados en el algoritmo

Mientras que, en el código que nos sirve de base, solo había un archivo de configuración, se han creado 6 archivos de configuración más que difieren en el número de entradas,correspondiendo cada entrada a un sensor de proximidad. Esto aparecerá explicado en el apartado de Pruebas.

Se han considerado configuraciones con 2, 3...., hasta 11 sensores dispuestos en distintas configuraciones geométricas, siempre simétricas respecto al eje principal del vehículo, lo que suma un total de los 7 archivos de configuración distintos. Gracias a ellos, se ha podido realizar distintas pruebas para poder comparar qué combinación es la mejor.

También se ha visto que, a pesar de tener 2 salidas, no estaba optimizado, puesto que una salida se destinaba para girar a la izquierda y la otra para girar a la derecha.Ahora seguimos teniendo 2 salidas, solo que una se encarga de controlar el giro y la otra controla la velocidad.

```
# Input my data and get result from network
for index, car in enumerate(cars):
    output = nets[index].activate(car.get_data())
    i = output.index(max(output))
    if i == 0:
        car.angle += 10
    else:
        car.angle -= 10
```

Figura 3.3: Salida original

Otra de las limitaciones residía en que el código no estaba diseñado para poder ejecutar los datos de una generación en el futuro, ya sea a modo de repetición o para continuar el entrenamiento a partir de los datos de la generación en cuestión. Esto es debido a que no se utilizaba la función *checkpoint* proporcionada por la biblioteca NEAT. Esta biblioteca permite guardar el estado del modelo y todos los datos relacionados con la tipología de la red.

```
p.add_reporter(neat.Checkpointer(1,5))
```

Figura 3.4: Código para guardar las generaciones

En esta imagen podemos apreciar cómo se guardan las poblaciones y otros aspectos del estado de la simulación, como que cada vez que termina una generación se guardan los datos obtenidos en intervalos de tiempo de 5 segundos.

Utilizando este módulo de NEAT, se han creado 3 opciones distintas para ejecutar el código:

- Entrenamiento: esta es la que tenía originalmente.
- Repetición: pensada para reproducir la generación deseada una y otra vez.
- Cargar y continuar: muy útil por si hay problemas técnicos, ya que permite seguir entrenando el algoritmo. Esto se detallará más dentro del modo supervisión explicado en el siguiente punto.

Finalmente, se ha añadido la representación del *fitness* y de la especiación del programa. Estos se representarán gráficamente cuando se cumpla el objetivo del umbral.

```
visualize.plot_stats(stats, ylog=True, view=True, filename="feedforward-fitness.svg")
visualize.plot_species(stats, view=True, filename="feedforward-speciation.svg")
```

Figura 3.5: Código para representar y guardar las gráficas de *fitness*

Se han añadido al código una diversidad de modos que, en función de las necesidades del usuario, puede elegir.También es cierto que se pueden ejecutar distintos modos en paralelo.

Este modo solo se puede usar con el modo monitor.

3.2.1. Modo Consola

Para ejecutar este modo, se deben configurar una serie de líneas de código en el archivo.

```
import os
os.environ["SDL_VIDEODRIVER"] = "dummy"
os.environ['SDL_AUDIODRIVER'] = 'dsp'
```

Figura 3.6: Modo Consola

Con estos comandos podemos evitar que la librería Pygame intente usar un dispositivo de visualización real y, en su lugar, SDL usará un controlador ficticio.
SDL es el acrónimo de Simple DirectMedia Layer, que es una biblioteca de desarrollo multiplataforma diseñada para acceder a bajo nivel al hardware de audio, gráficos, teclado, etc.

[0]https://wiki.libsdl.org/SDL2/FAQUsingSDL

Gracias a esto, podemos ejecutar el código en otras plataformas, como Google Colab, que no disponen de tarjeta de vídeo ni salida de sonido. Cabe mencionar que, aunque se podría ejecutar el código en Google Colab, aparecería un mensaje de error en cada generación.

Este modo solo se puede usar con el modo monitor.

3.2.2. Modo Interactivo

Una de las ventajas que nos proporciona Pygame es la comunicación entre el usuario y programa mediante la utilización de periféricos como el teclado y el ratón.

Por ello, elegiremos unos caracteres del teclado y les asignaremos una función especifica.

- Tecla C. Con esta tecla guardaremos en la ruta previamente establecida, en un archivo de texto, la clasificación en la que se encuentran los vehículos en ese momento la simulación,en orden descendente. El archivo guardará el índice del coche, las vueltas realizadas y la distancia recorrida. Cabe mencionar que existe la posibilidad de que el coche que más distancia ha recorrido no sea el coche que más vueltas ha dado.Esto es debido a diversas causas como, por ejemplo, que el coche se encuentre pegado al extremo opuesto a donde se esté la curva, incrementando así la distancia que tiene que recorrer para tomarla.

- Tecla R. Al pulsar dicha tecla, se imprimirá por consola la recompensa del coche que haya obtenido la mayor cantidad de puntos en la generación. Esto resulta especialmente útil para verificar si, en ese instante, la recompensa es mayor que el umbral de *fitness* configurado en el documento ***config-feedforward***, el cual se encuentra en la sección de configuración correspondiente.

- Tecla P. El objetivo de esta tecla es consultar en ese instante de tiempo todos los datos de los coches que están en pista, como la velocidad actual, la velocidad máxima que han alcanzado en la generación, el número de veces que han modificado la velocidad y los tiempos, tanto el tiempo que han tardado en dar una vuelta como el tiempo total que llevan en la generación.

```
# Obtain data of all the vehicles that are on the track
if event.key == pygame.K_p:
    print("\n")
    for i, car in enumerate(cars):
        if car.is_alive == True:

            new_directory = "current_data_car"
            if not os.path.exists(new_directory):
                os.makedirs(new_directory)

            Path_car_i = os.path.join(Path_txt_files,f'generation_{str(generation)}/{new_directory}/car_{str(i)}.txt')

            f= open(Path_car_i,'w')
            f.write(f"Coche:     {i}\n")
            f.write(f"Estado:  {car.get_alive()}\n")
            f.write(f"Angle:  {car.angle}\n")
            f.write(f"Velocidad:  {car.speed}\n")
            f.write(f"Maxima Velocidad:  {car.max_speed}\n")
            f.write(f"Fitness:  {genomes[i][1].fitness}\n")
            f.write(f"Distancia recorrida:  {car.distance}\n")
            f.write(f"Numero de veces que ha girado a la derecha:   {car.number_times_turned_to_the_right}\n")
            f.write(f"Numero de veces que ha girado a la izquierda: {car.number_times_turned_to_the_left}\n")
            f.write(f"Numero de veces que NO ha girado: {car.number_times_no_turned}\n")
            f.write(f"Numero de veces que ha acelerado: {car.number_times_accelerated}\n")
            f.write(f"Numero de veces que ha frenado:   {car.number_times_braking}\n")
            f.write(f"Tiempo gastado:  {car.time_spent}\n")
            f.write(f"Tiempo ultima vuelta:  {car.last_time_lap}\n")
            f.write(f"Vuelta Habilitada:  {car.lap_enable}\n")
            f.write(f"Vuelta de get_lap:  {car.get_lap()}\n")
            f.write(f"Vuelta de lap:  {car.lap}\n")
            f.write(f"Posicion:  {car.pos}\n")
            f.write(f"Centro:  {car.center}\n")
            f.write(f"Proximo sector:  {car.next_sector}\n")
            f.close()
```

Figura 3.7: Detalles tecla P

- Tecla K. Esta tecla está diseñada para salir del programa utilizando el comando "sys.exit(0".

- Tecla Espacio. Mediante la tecla espacio, podemos dar por finalizada la generación actual. Con esta función, comprobaremos qué coches cumplen con la condición ***car.is_alive==True***, para posteriormente asignarle el valor de *False* y que el programa detecte en la siguiente generación que no quedan coches con vida, forzando así una nueva generación.Es útil si, al usar la tecla R, vemos que el *fitness* ya es superior al umbral, para dar por finalizado el proyecto, puesto que ya se ha cumplido el objetivo.

También debemos mencionar que con otras teclas se puede controlar el tiempo que dura la generación, ya sea para activar o desactivar esta función o para ajustar el tiempo.

```
if active_timestop == True:
    delta = (datetime.datetime.now() - now)
    if delta.total_seconds()>counter_max:
        print("Time Stop")
        for i, car in enumerate(cars):
            if car.is_alive == True:
                car.is_alive = False
        break
```

Figura 3.8: Detalles función Tiempo de Parada

- Tecla T. habilita o deshabilita el Tiempo de Parada. Si está activado, cuando llegue el tiempo, termina la generación. El tiempo por defecto es 20 segundos, aunque puede ajustarse con las

teclas mencionadas en los puntos siguientes.

- Tecla División(/). divide el Tiempo de Parada a la mitad, con un máximo de 2 segundos.
- Tecla Multiplicación(*). se duplica el Tiempo de Parada.
- Tecla Resta(-). reduce el Tiempo de Parada en 1 segundo.
- Tecla Suma(+). añade 5 segundos al Tiempo de Parada.

```python
# Activate/Deactivate Time Stop
if event.key == pygame.K_t:
    global active_timestop
    active_timestop = not active_timestop
    global counter_max

    if active_timestop == True:
        print(f"\nActivado Time Stop con un tiempo de {counter_max}\n")
    else:
        print("\nDesactivado Time Stop\n")

if event.key == pygame.K_KP_DIVIDE:
    if counter_max > 2:
        counter_max /= 2
        counter_max = int(counter_max)
        print(f"New time {counter_max}")
    else:
        print("Counter_max is too small to divide")


if event.key == pygame.K_KP_PLUS:
    counter_max += 5
    print(f"New time {counter_max}")

if event.key == pygame.K_KP_MINUS:
    #global counter_max
    if counter_max > 20:
        counter_max -= 1
        print(f"New time {counter_max}")

if event.key == pygame.K_KP_MULTIPLY:
    counter_max *= 2
    print(f"New time {counter_max}")
```

Figura 3.9: Detalles teclas Tiempo de Parada

Este modo puede usarse tanto con el modo supervisión como en el modo monitor.

3.2.3. Modo Supervisión

Este sería el modo más amplio, puesto que dentro de este, existen 3 formas de supervisar, que dependerán en esta ocasión, del argumento que acompañe a invocar al archivo python, el cual tiene extensión *.py*.

```
if option_Value == 0: # HELP
    print("Selecciones permitidas son: \n")
    print(" 1) -h o --Help para mostrar el menú ayuda")
    print(" 2) -t o --Train para entrenar el algoritmo")
    print(" 3) -l X o --Load X para continuar el entrenamiento del algoritmo en una generación determinada")
    print(" 4) -r X o --Replay X para mostrar una repeticion de una generacion determinada")
    print(" 5) -c o --Check si se desea consultar el valor de la ultima generacion con checkpoint")
    print(" 6) Si no se utiliza ningún argumento se tomara la opción 2 por defecto\n")
    print("En el caso de que se seleccione Load o Replay y no se indique la generacion se cogera por defecto la ultima almacenada")
```

Figura 3.10: Modo Supervisión

Para poder hacer esto, es necesario importar la librería **getopt**, analiza las opciones de la línea de comandos, y la librería **sys**, que proporciona los recursos necesarios para interactuar con el sistema operativo.

```
import getopt, sys
```

Figura 3.11: Importar librerías

El siguiente paso es detectar si las palabras que acompañan a la invocación del programa son las preestablecidas o por el contrario, son palabras no reconocidas. En este último caso, se lanzará un mensaje de error y se activará el modo entrenamiento.

```
Mode = 1
GenerationSelected = None
argumentList = sys.argv[1:]
# Options
options = "htlrc"
# Long options
long_options = ["Help", "Train", "Load","Replay","Check"]
try:
    # Parsing argument
    arguments, values = getopt.getopt(argumentList, options, long_options)
    checking = True
except getopt.GetoptError:
    print("\n Comando no valido, opcion Train se activa por defecto \n")
    Mode = 1
    checking = False
```

Figura 3.12: Comprobar palabras clave

Para asociar la palabra introducida por el usuario al modo que queremos ejecutar, será necesario implementar la lógica correspondiente en el código Python. Podemos utilizar estructuras de control como condicionales o diccionarios para mapear la palabra introducida a la acción correspondiente. Por ejemplo, si el usuario introduce la palabra *"Replay"*, podemos ejecutar una función específica para el **Modo Repetición**; si introduce la palabra *"Load"*, ejecutamos otra función para el **Modo Cargar**, y así sucesivamente.

De esta manera, el programa podrá interpretar la entrada del usuario y realizar las acciones correspondientes según el modo asociado a la palabra introducida. Es importante asegurarse de manejar casos

de entrada no reconocida para evitar errores y proporcionar mensajes claros al usuario en caso de que la palabra no sea válida

```
if checking == True:
# checking each argument
    for currentArgument, currentValue in arguments:
        if currentArgument in ("-h", "--Help"):
            Mode = 0
        elif currentArgument in ("-t", "--Train"):
            Mode = 1
        elif currentArgument in ("-l", "--Load"):
            Mode= 2
            if values == []:
                GenerationSelected = None
            else:
                GenerationSelected = values[0]
        elif currentArgument in ("-r", "--Replay"):
            Mode = 3
            if values == []:
                GenerationSelected = None
            else:
                GenerationSelected = values[0]
        elif currentArgument in ("-c", "--Check"):
            Mode = 4
return Mode,GenerationSelected;
```

Figura 3.13: Relación Palabras-Modo.

.

El objetivo de esta función es poder proporcionar flexibilidad al programa, de manera que podamos reproducir una generación especifica o por el contrario, continuar el entrenamiento a partir de una generación previamente ejecutada y guardada.

3.2.3.1. Modo Entrenamiento

Este es el modo que se ejecuta cuando se introduce como argumento “-t” o cuando se introduce un comando no registrado. Este modo se encarga de realizar el entrenamiento del algoritmo. Se detallará en profundidad en el apartado 3.3.

Este tipo de visualización del proyecto es compatible con el resto de modos.

3.2.3.2. Modo Repetición

Mediante el argumento “-r”, somos capaces de reproducir nuevamente la generación deseada, de manera que podemos comprobar el comportamiento de la generación en otros circuitos.

La manera en que hacemos esto es analizar la carpeta donde se guardan las generaciones y buscar el índice más alto, ya que los archivos se guardan en formato neat-checkpoint-X, donde X representa la generación.

Por último, utilizando la función de *Checkpointer* que posee el algoritmo NEAT (neat.Checkpointer-.restore_checkpoint(checkpoint)), podemos recuperar esa generación.

Este modo es compatible con el resto de modos.

```python
elif option_Value == 3:

        valido = False
        if GenerationSelected == None:
            for i in range(0,1001):
                txt = f'txt_files/generation_{i}'
                i = int(i)
                if not os.path.exists(txt):
                    if i == 0:
                        print(f"No existe generaciones guardadas")
                    else:
                        if not os.path.exists(txt):
                            txt2 = f'txt_files/generation_{i}/neat-checkpoint-{i}'
                            if not os.path.exists(txt2):
                                generation = int(i-2)
                            else:
                                generation = int(i-1)
                    break
        else:
                txt = f'txt_files/generation_{GenerationSelected}/neat-checkpoint-{GenerationSelected}'
                try:
                    if os.path.exists(txt):
                        print(f"Cargando generacion {GenerationSelected}")
                        generation = int(GenerationSelected)
                except:
                    print(f"No existe la generacion {GenerationSelected}")
```

Figura 3.14: Modo repetición

3.2.3.3. Modo Cargar

Durante todo el proceso de entrenamiento, se han experimentado diversas circunstancias que derivaban en cerrar la aplicación. Esto provocaba que se tuviera que volver a generar todo el programa desde cero. Debido a ello, nació la idea de aprovechar que ya se podía guardar el estado del entrenamiento para poder recuperar este estado posteriormente y entrenar a partir de ahí.

También se ha optado este modo para monitorizar un determinado coche (esto se explicará en el siguiente modo) y para trasladar dicha generación a otro circuito, con el fin de comprobar que las variaciones son realmente favorables o no.

```python
elif option_Value == 2: # LOAD
        valido = False
        if GenerationSelected == None:
            for i in range(0,10000):
                txt = f'txt_files/generation_{i}'
                if not os.path.exists(txt):
                    if i == 0:
                        print(f"No existe generaciones guardadas")
                    else:
                        if not os.path.exists(txt):
                            txt2 = f'txt_files/generation_{i}/neat-checkpoint-{i}'
                            if not os.path.exists(txt2):
                                #print(f"ultima generacion es {i-2}")
                                generation = int(i-2)
                            else:
                                print(f"ultima generacion es {i-1}")
                                #generation = int(i-1)
                    break
        else:
                txt = f'txt_files/generation_{GenerationSelected}/neat-checkpoint-{GenerationSelected}'
                if  os.path.exists(txt):
                    print(f"Cargando generacion {GenerationSelected}")
                    generation = int(GenerationSelected)
```

Figura 3.15: Modo carga

3.2.4. Modo Monitorización

Mediante este modo, podemos analizar el comportamiento de un coche en particular. Registramos en cada iteración cada variación del coche, ya sean relacionadas con la velocidad, como acelerar y frenar, o relacionadas con los giros que realiza el coche, los cuales se exportan en un fichero JSON.

```
if mode_debug == True:
    if index == car_monitored:
        new_directory = "monitirizar_coche"
        if not os.path.exists(os.path.join(Path_txt_files,f'{new_directory}')):
            os.makedirs(os.path.join(Path_txt_files,f'{new_directory}'))


        radars2 = car.radars
        ret2  = [0] * sensors
        for i, r in enumerate(radars2):
            ret2[i] = r[1]
        import json
        monitorizar["Datos"].append({
            "Iteracion" : it,
            "output":  output,
            "Speed":  car.speed,
            "Angle":  car.angle,
            "Numero de veces que ha girado a la derecha":   car.number_times_turned_to_the_right,
            "Numero de veces que ha girado a la izquierda": car.number_times_turned_to_the_left,
            "Numero de veces que NO ha girado" : car.number_times_no_turned,
            "Numero de veces que ha acelerado": car.number_times_accelerated,
            "Numero de veces que ha frenado":   car.number_times_braking,
            "Distancia Radares" : ret2
        })

        it += 1
```

Figura 3.16: Modo de monitorización

La función principal de este modo es ser una herramienta complementaria a la opción de repetir una generación y poder ver por qué se ha producido el éxito de la misma, o por el contrario, por qué ese determinado coche no ha podido dar la vuelta correctamente.

El modo de monitorización ha resultado muy útil para detectar que el coche no aceleraba ni frenaba en las curvas, manteniendo una velocidad constante. Esto indica que en lugar de generalizar el comportamiento del coche para cada circuito, solo funcionará en ese mapa específico.

```
{
    "Iteracion": 19182,
    "output": [
        1.0,
        0.9112303417446528
    ],
    "Speed": 10.000271186959642,
    "Angle": 166.799999999992,
    "Numero de veces que ha girado a la derecha": 11859,
    "Numero de veces que ha girado a la izquierda": 7324,
    "Numero de veces que ha acelerado": 909,
    "Numero de veces que ha frenado": 1052,
    "Distancia Radares": [
        75,
        75,
        95,
        170,
        31
    ]
},
{
    "Iteracion": 19183,
    "output": [
        1.0,
        -0.8990986036581451
    ],
    "Speed": 9.101172583301498,
    "Angle": 172.799999999992,
    "Numero de veces que ha girado a la derecha": 11860,
    "Numero de veces que ha girado a la izquierda": 7324,
    "Numero de veces que ha acelerado": 909,
    "Numero de veces que ha frenado": 1053,
    "Distancia Radares": [
        71,
        71,
        89,
        175,
        42
    ]
},
```


Figura 3.17: Ejemplo de modo monitor

3.3 Entrenamiento

En esta sección se detallarán los archivos del proyecto necesarios para llevar a cabo el entrenamiento, entre los que destacan:

- **Car.py**: En este archivo se define el vehículo, se establecen las recompensas, se obtienen los datos provenientes del radar, se manejan las colisiones, etc.

- **Pycar.py**: En este archivo se realizan los ajustes de velocidad y ángulo de giro, se configura el modo de visualización y se representa visualmente los coches en los mapas.

- **Config-feedforward.txt**: Este archivo de texto contiene los parámetros necesarios para que el algoritmo NEAT funcione correctamente, como las probabilidades de mutación, el número de individuos, etc.

3.3.1. Archivo Car.py

En este archivo de configuración se implementarán las funciones principales relacionadas con el coche. Comenzando con la creación de la clase coche, se inicializarán las variables como la posición inicial, velocidad inicial, etc. Además, se incluirán funciones relacionadas con las colisiones, como calcular la distancia entre nuestro vehículo y el punto de colisión. También se definirán las funciones de recompensas.

3.3.1.1. Establecer posición

En primer lugar, será necesario inicializar los mapas y las posiciones iniciales. Dado que los mapas no tienen las mismas dimensiones y la ubicación de la salida varía, se deben establecer diferentes valores para cada mapa. Esto implica definir las dimensiones de cada mapa específico y establecer las posiciones iniciales y las ubicaciones de salida correspondientes a cada uno. De esta manera, el programa podrá adaptarse correctamente a las características individuales de cada mapa durante el entrenamiento.

```
# Set initial position, angle and sp
type1_map = [1,2,4,5,6,7,8]
type2_map = [3]
type3_map = [9]
type4_map = [10]
type5_map= [11]
if random_map in type1_map:
    self.pos = [680, 650]
    self.last_position = [680, 650]

elif random_map in type2_map:
    self.pos = [675, 635]
    self.last_position = [675, 635]

elif random_map in type3_map:
    self.pos = [1560, 1000]
    self.last_position = [1560,1000]

elif random_map in type4_map:
    self.pos = [730, 915]
    self.last_position = [730,915]

elif random_map in type5_map:
    self.pos = [890, 700]
    self.last_position = [890, 700]
```

Figura 3.18: Definición de posiciones según el mapa

Dependiendo de si el mapa está registrado en una u otra lista, estableceremos la posición.La variable ***pos*** hace referencia a la posición actual del vehículo y la variable ***last_position*** hace referencia a la última posición registrada, necesaria para comprobar periódicamente si el coche se está moviendo a velocidad y dirección correcta o por el contrario se encuentra realizando giros sobre si mismo.El código será detallado más adelante.

3.3.1.2. Establecimiento de variables

Para el caso de la velocidad inicial, se ha optado por empezar con una velocidad de 0. Esta velocidad coincide con la velocidad mínima que puede tener el vehículo. Si la velocidad es negativa, el vehículo será eliminado. En cuanto a la velocidad máxima, se ha establecido en 11.

```
self.angle = 0
self.speed = 0 #1
self.max_speed = 0
self.center = [self.pos[0] + (CAR_SIZE_X / 2) , self.pos[1] + (CAR_SIZE_Y /2)]
self.lap = 0
self.rewards = 0
self.sensores = sensors
self.radars = []
self.radars_for_draw = []
```

Figura 3.19: Configuración de las variables iniciales

Otras de las variables que queremos monitorizar es la de la cantidad de veces que ha frenado y acelerado, para ello nos valdremos de las variables que aparecen en la siguiente figura.

```
self.number_times_accelerated = 0
self.number_times_braking = 0

self.number_times_turned_to_the_right = 0
self.number_times_turned_to_the_left = 0
```

Figura 3.20: Configuración de las variables de las salidas

También queremos registrar los tiempos, debido a que queremos hacer una comparativa entre los tiempos que tarda cada coche en completar una vuelta, ya que esta es una de las condiciones que tendremos para establecer una recompensa.

```
# Config Times
self.initial_time = datetime.datetime.now()
self.stop_time = 0
self.last_time_lap = None
self.time_spent = 0
self.last_time_spent = 0
self.best_personal_time = None
self.new_time = 0
```

Figura 3.21: Configuración de variables del tiempo

3.3.1.3. Definir funciones

Como se ha comentado anteriormente, monitorizaremos la posición del coche cada cierto tiempo, para detectar principalmente si está girando sobre si mismo.A efectos de distancia, se cumpliría porque se encuentra recorriendo una distancia pero estaría falseando los resultados, ya que no avanzaría.

De este modo, cada 150 iteraciones comprobaremos la posición del vehículo. Si se encuentra en una posición de $\pm$ 65 píxeles, este será tomado como no válido y será eliminado.

```
def avoid_turning_on_itself(self,time,i):
    change_position = False

    if time % 150 == 0:
        if ((self.pos[0] -65) <= self.last_position[0]   <= (self.pos[0] +65)):
            if ((self.pos[1] -65) <= self.last_position[1]   <= (self.pos[1] +65)):
                self.check_dies(i, 0)
                self.is_alive = False
                self.rewards = 0
            else:
                change_position = True
        else:
            change_position = True

        if change_position == True:
            self.last_position[0] = self.pos[0]
            self.last_position[1] = self.pos[1]
```

Figura 3.22: Función posición vehículo

Durante todo el entrenamiento, tenemos que tener cuidado con las colisiones, debido a que un error a la hora de establecer los puntos de colisión podría ocasionar un entrenamiento erróneo.

Se tomará como base el centro del vehículo, para posteriormente calcular los 4 puntos de colisión, estableciendo un margen de la mitad del tamaño que tiene el utilitario.

```
# calculate 4 collision points
self.center = [int(self.pos[0]) + (CAR_SIZE_X/ 2 ), int(self.pos[1]) + (CAR_SIZE_Y / 2 )]
len = CAR_SIZE_X/2
left_top = [self.center[0] + math.cos(math.radians(360 - (self.angle + 30))) * len, self.center[1] + math.sin(math.radians(360 - (self.angle + 30))) * len]
right_top = [self.center[0] + math.cos(math.radians(360 - (self.angle + 150))) * len, self.center[1] + math.sin(math.radians(360 - (self.angle + 150))) * len]
left_bottom = [self.center[0] + math.cos(math.radians(360 - (self.angle + 210))) * len, self.center[1] + math.sin(math.radians(360 - (self.angle + 210))) * len]
right_bottom = [self.center[0] + math.cos(math.radians(360 - (self.angle + 330))) * len, self.center[1] + math.sin(math.radians(360 - (self.angle + 330))) * len]
self.four_points = [left_top, right_top, left_bottom, right_bottom]

self.check_collision(map,i)
```

Figura 3.23: Establecer los puntos de colisiones

Después, solo tendremos que comprobar en cada iteración que no existe la colisión, para lo cual comprobaremos el color del píxel del circuito en el que se encuentra el coche.

```
def check_collision(self, map,i):
    self.is_alive = True
    for p in self.four_points:
        try:
            if map.get_at((int(p[0]), int(p[1]))) == BORDER_COLOR:
                self.is_alive = False
                break
        except:
            print(map.get_at((int(p[0]), int(p[1]))))
            print(self.pos)
```

Figura 3.24: Comprobar posibles colisiones

En función del tipo de evento que provoque que el coche ya no siga con vida, se registrará para que una vez acabada la generación, la información se vuelque en un archivo de texto en el que se detallará cada vehículo y su correspondiente razón de la "muerte". Esta función es bastante útil a la hora de entrenar el algoritmo de 0, ya que, a efectos visuales, no existe diferencia entre una colisión y una velocidad negativa.

```
def check_dies(self,i, reason):

    reasons = ['Avoid turning on itself', 'Collision' , '0 Speed' , 'Few Distance', 'Opossite Direction', 'Key Space', 'Key Q', 'Exit Game']
    record_dies[i] = reasons[reason]
    record_rewards[i] = self.rewards
    self.stop_time = datetime.datetime.now()
```

Figura 3.25: Función para registrar la muerte

Para comprobar si el coche ha dado una vuelta, se ha diseñado una función similar a la de calcular la colisión, donde se comparará el píxel central del coche con una serie de valores correspondientes a los pintados en la meta. Si coincide, se cumplirá la condición y se incrementará el contador de vueltas. Para contar otra vuelta, será necesario que vuelva a detectar un píxel negro, ya que, de otra manera, si el vehículo va despacio, podría ocurrir que se cuenten múltiples vueltas en una sola ocasión.

```python
def check_lap(self,car,i,number_map,map):

        warningTime = (datetime.datetime.now() - self.initial_time)
        if warningTime.total_seconds() > 10 :
            Meta1 = (20 , 77, 0) #
            Meta2 = (163 , 193, 160) #
            Road = (0 , 0, 0) #
            p = self.center
            if self.lap_enable == True:
                if ((map.get_at((int(p[0]), int(p[1]))) == Meta1) or map.get_at((int(p[0]), int(p[1]))) == Meta2):

                    self.lap += 1
                    self.last_time_spent = self.time_spent

                    if self.lap == 1:
                        self.last_time_lap = round((datetime.datetime.now()- self.initial_time).total_seconds())
                        print(f'Car number {i} has done {self.lap} lap with a time of {(datetime.timedelta(seconds=self.last_time_lap))}')
                    else:
                        self.last_time_lap = round((datetime.datetime.now()- (self.new_time)).total_seconds())
                        print(f'Car number {i} has done {self.lap} laps with a time of {(datetime.timedelta(seconds=self.last_time_lap))}')

                    self.new_time = datetime.datetime.now()
                    self.lap_enable = False

                    if self.best_personal_time == None :
                        self.best_personal_time = self.last_time_lap
                    else:
                        if self.last_time_lap < self.best_personal_time:
                            self.best_personal_time = self.last_time_lap

                    global best_lap
                    global author_best_lap

                    if best_lap == None:
                        best_lap = self.last_time_lap
                        print(f"car {i} set faster lap")
                        author_best_lap = f'car {i}'

                    else:
                        if self.last_time_lap < best_lap:
                            print(f"car {i} set faster lap")
                            best_lap = self.last_time_lap
                            author_best_lap = f'car {i}'

            else:
                if map.get_at((int(p[0]), int(p[1]))) == Road:
                    self.lap_enable = True
```

Figura 3.26: Función comprobar vuelta

La función con la que comprobaremos la distancia del radar consistirá en,a partir de un determinado ángulo, ir incrementando en esa dirección la variable hasta llegar a un píxel blanco, el cual está establecido como la variable **BORDER_COLOR**. Una vez calculada la distancia, procederemos a

```python
def check_radar(self, degree, map):
    len = 0
    x = int(self.center[0] + math.cos(math.radians(360 - (self.angle + degree))) * len)
    y = int(self.center[1] + math.sin(math.radians(360 - (self.angle + degree))) * len)

    try:
        while not map.get_at((x, y)) == BORDER_COLOR:
            len = len + 1
            x = int(self.center[0] + math.cos(math.radians(360 - (self.angle + degree))) * len)
            y = int(self.center[1] + math.sin(math.radians(360 - (self.angle + degree))) * len)
    except:
        nada=0

    finally:
        len -= 2
        x = int(self.center[0] + math.cos(math.radians(360 - (self.angle + degree))) * len)
        y = int(self.center[1] + math.sin(math.radians(360 - (self.angle + degree))) * len)

    dist = int(math.sqrt(math.pow(x - self.center[0], 2) + math.pow(y - self.center[1], 2)))

    self.radars.append([(x, y), dist])
```

Figura 3.27: Función para comprobar radar

pintarla de verde para que sea más sencillo visualizar la generación. Se pintará con una línea verde el trayecto hasta la pared más cercana y con un círculo verde de radio 5 el último píxel. A su vez, para llevar un cierto control, se ha optado por la opción de imprimir el identificador del número en el centro del vehículo si quedan 5 coches o menos. No se hace en todos, puesto que ralentizaría mucho el programa.

```python
def draw_radar(self, screen,remain_cars):
    self.get_data()
    for r in self.radars:
        pos, _ = r
        pygame.draw.line(screen, (0, 255, 0), self.center, pos, 1)
        pygame.draw.circle(screen, (0, 255, 0), pos, 5)
        font = pygame.font.SysFont('Arial', 25)
        if remain_cars <= 5:
            media = [0,0]
            media[0] = (self.pos[0] + self.center[0])/2
            media[1] = (self.pos[1] + self.center[1])/2
            screen.blit(font.render(str(self.number), True, (255,255,0)), media)
```

Figura 3.28: Función para dibujar el radar

Una vez configurada la función de detectar los límites de pista, el siguiente paso será obtener los datos proporcionados por cada sensor. Estos se escalaran a una escala de 1:30 para manejar los datos de los radares. El número de sensores es elegido por el usuario.

```python
def get_data(self):
    radars = self.radars
    ret  = [0] * sensors
    for i, r in enumerate(radars):
        ret[i] = int(r[1] / 30) # escale
    return ret
```

Figura 3.29: Función para obtener datos sensores

Dependiendo de la cantidad de sensores que elijamos, tendremos que segmentar el rango en diferentes valores para posteriormente invocar la función que comprueba el radar y, finalmente, pintar los resultados.

```
if sensors == 2:
    for d in range(-30, 31, 60):
        self.check_radar(d, map)
if sensors == 3:
    for d in range(-30, 31, 30):
        self.check_radar(d, map)
if sensors == 4:
    for d in range(-60, 61, 40):
        self.check_radar(d, map)
if sensors == 5:
    for d in range(-90, 120, 45):
        self.check_radar(d, map)
if sensors == 7:
    for d in range(-60, 60, 20):
        self.check_radar(d, map)
if sensors == 9:
    for d in range(-60, 61, 15):
        self.check_radar(d, map)
if sensors == 11:
    for d in range(-90, 62, 12):
        self.check_radar(d, map)
```

Figura 3.30: Función para elegir tipo de sensorización

Otra cosa a tener en cuenta es que la forma en que representamos un juego en Pygame es repintando el mapa y adaptándolo a las nuevas circunstancias. Por lo tanto, debemos tener en cuenta que hay que rotar la imagen original un determinado ángulo. Este ángulo se deriva de la salida del algoritmo al activar la red neuronal.

```
def rot_center(self, image, angle):
    orig_rect = image.get_rect()
    rot_image = pygame.transform.rotate(image, angle)
    rot_rect = orig_rect.copy()
    rot_rect.center = rot_image.get_rect().center
    rot_image = rot_image.subsurface(rot_rect).copy()
    return rot_image
```

Figura 3.31: Función para rotar la imagen

Finalmente, en este archivo, quedaría explicar la parte más relevante del código, la función de recompensa. Esta función se explicará en el siguiente apartado.

3.3.1.4. Fitness

Al ser un aprendizaje por refuerzo, la clave del éxito o fracaso del algoritmo reside en la recompensa, para ello, en este caso, se ha optado por realizar dos tipos de ”*fitness*”distintos, según el coche esté en una etapa u otra.

Para el caso de que el coche aún no haya conseguido dar una vuelta, la recompensa obtenida será el producto de un valor constante, en nuestro caso 0.000001 (cuyo objetivo es empequeñecer el valor numérico de la recompensa) y la distancia recorrida por el coche ***self.distance***. Posteriormente, tiene lugar la fase de la penalización en la recompensa.

El objetivo de esta penalización es evitar que un coche pueda girar sobre sí mismo e ir en dirección contraria a la esperada, contaminando así la muestra. Así que, al realizar esta división, perjudicaremos

a aquellos coches que tengan una distancia entre los sensores laterales mayor de lo habitual.

Cuando el coche haya conseguido dar una vuelta completa, el algoritmo para calcular el *fitness* se modificaría para tener en cuenta la cantidad de vueltas dadas por el coche, así como el tiempo en darlas, ya que buscamos el coche que tarde menos tiempo en realizar una vuelta.

Debido a que el algoritmo de este proyecto es del tipo por refuerzo se han tenido en cuenta diversas formas de penalizar a los coches.

Las penalizaciones son las siguientes:

- Marcha atrás: Se establece que el coche no puede tener una velocidad inferior a 0. Si esto ocurre, el coche se eliminará y se le dará un valor de 0 al *fitness* total para evitar que el programa genere nuevos coches utilizando a este coche como padre.

- Coche no avanza o gira sobre sí mismo: Dado que uno de los objetivos del código es que el vehículo realice las vueltas al circuito en el menor tiempo posible, se monitorea su posición. En el caso de que no exista una distancia considerable en un determinado período de tiempo, se considera que el coche no es válido y se le asigna un valor de 0, como en el caso anterior.

- Coche no utiliza el freno: Durante el entrenamiento, se ha observado que existe el riesgo de que el coche utilice una velocidad constante para dar la vuelta al circuito. Esto implica que, con alta probabilidad, el coche adaptará su velocidad a ese circuito en particular. Sin embargo, cuando esa generación se pruebe en otro circuito, existe la posibilidad de que no se adapte correctamente. Esto puede suceder porque el coche va a una velocidad muy baja, lo que hace que tarde mucho tiempo en dar una vuelta, o porque al tener una velocidad constante no frene al llegar a una curva y se salga del circuito, lo que provoca su eliminación.

- Coche se prepara para girar sobre sí mismo: A la hora de obtener la recompensa, ésta es inversamente proporcional a la distancia entre los sensores laterales. Es decir, cuanto mayor sea la distancia, menor será la recompensa.

```python
def get_reward(self):
    delta = round((datetime.datetime.now()- self.initial_time).total_seconds())
    radars2 = self.radars
    ret2  = [0] * sensors
    for i, r in enumerate(radars2):
            ret2[i] = r[1]
    if self.last_time_lap == None:
        reward_total = self.distance * 0.000001
        if ret2[0] != ret2[-1]:
            reward_total = reward_total/(abs(ret2[0] - ret2[-1]))
        else:
            reward_total = 0
    else:
        if ret2[0] != ret2[-1]:
            reward_total = (self.distance * 0.000001)*delta + 0.5*self.lap*(best_lap /self.last_time_lap)*2
            reward_total = reward_total/(abs(ret2[0] - ret2[-1]))
        else:
            reward_total = 0
    return reward_total
```

Figura 3.32: Función recompensa

3.3.2. Archivo PyCar.py

En este apartado se detallarán las partes principales del código correspondiente al fichero PyCar.py.

3.3.2.1. Iniciar NEAT

En este apartado se declararán los vectores iniciales de las redes y los coches, para posteriormente ir añadiendo los elementos a su vector correspondiente.

```
# Init NEAT
nets = []
cars = []


for id, g in genomes:
    net = neat.nn.FeedForwardNetwork.create(g, config)
    nets.append(net)
    g.fitness = 0.0

    # Init my cars
    cars.append(Car(random_map))
```

Figura 3.33: Iniciar algoritmo NEAT

Para el vector de los coches, este será rellenado con el objeto Coche. Se incluirán tantos vehículos como tamaño de la población esté definido en el archivo de configuración ***config-forward***.

Por otro lado, para el caso de las redes, crearemos una red neuronal recurrente con la configuración preestablecida en el mismo archivo de configuración que para el caso de los coches, a partir de los genomas.

3.3.2.2. Iniciar juego

El siguiente paso será declarar las variables necesarias para poder iniciar el juego. Entre las variables declaradas destaca la configuración de una pantalla mediante la función *display.set_mode*, en la que pasaremos como parámetros el tamaño de la pantalla.

También se destaca la carga del circuito en el que entrenará el coche y la configuración de una imagen como icono del programa.

```
# Init my game
pygame.init()
screen = pygame.display.set_mode((screen_width, screen_height))
clock = pygame.time.Clock()
generation_font = pygame.font.SysFont("Arial", 70)
font = pygame.font.SysFont("Arial", 30)
map = pygame.image.load(f'maps/map{random_map}.png')
Icon = pygame.image.load('icon.png')
pygame.display.set_icon(Icon)
```

Figura 3.34: Iniciar el juego

3.3.2.3. Controles generados por la red neuronal

A continuación, se explicará el funcionamiento de la red y los valores que se asignarán según el resultado de la salida de la red neuronal.

En primer lugar, estableceremos la condición de que solo se aplique esta parte del código a aquellos vehículos que aún siguen en pista. Invocaremos la función *get_data*, en la que recibiremos los datos procedentes de los sensores que dispone el coche en ese instante de tiempo. Los datos que maneja esta función son los relativos a la distancia en los sensores para los ángulos dados.

Después, activaremos la red neuronal y obtendremos una salida vectorial con dos valores. El primer índice corresponde al ángulo de giro y el segundo índice corresponde a la velocidad del coche.

El siguiente paso será asignar un valor al ángulo y a la velocidad del coche.

Mientras que la velocidad la obtendremos directamente como el segundo valor del vector de salida, el ángulo, por el contrario, lo trataremos en distintos tramos.

En cada iteración, el valor del ángulo oscilará entre los valores $\pm$ 1.2º y $\pm$ 6º. Es de vital importancia que el ángulo de giro sea un número mayor que 0 debido a que el coche irá recto.

```
# Input my data and get result from network
for index, car in enumerate(cars):
    if car.is_alive == True:

        output = nets[index].activate(car.get_data())

        if output[0] > 0:

                if output[0]> 0.0 and output[0] <= 0.3:
                    car.angle += (3/5)*2
                elif output[0]> 0.3 and output[0] <= 0.4:
                    car.angle += (6/5)*2
                elif output[0]> 0.4 and output[0] <= 0.5:
                    car.angle += (7.5/5)*2
                elif output[0]> 0.5 and output[0] <= 0.7:
                    car.angle += (9/5)*2
                elif output[0]> 0.7 and output[0] <= 0.9:
                    car.angle += (12/5)*2
                else:
                    car.angle += (15/5)*2
                if car.angle > 360:
                    car.angle -=360
                car.number_times_turned_to_the_right +=1
        elif output[0] < -0:

                if output[0] < -0 and output[0] >= -0.3:
                    car.angle -= (3/5)*2
                elif output[0] < -0.3 and output[0] >= -0.4:
                    car.angle -= (6/5)*2
                elif output[0] < -0.4 and output[0] >= -0.5:
                    car.angle -= (7.5/5)*2
                elif output[0] < -0.5 and output[0] >= -0.7:
                    car.angle -= (9/5)*2
                elif output[0] < -0.7 and output[0] >= -0.9:
                    car.angle -= (12/5)*2
                else:
                    car.angle -= (15/5)*2
                if car.angle  <  -360:
                    car.angle +=360
                car.number_times_turned_to_the_left += 1
        else:
                car.angle += 0
```

Figura 3.35: Determinar ángulo de giro

3.3.2.4. Marcha atrás

Para la marcha atrás, seguiremos la misma filosofía que para determinar si ha dado o no una vuelta, es decir, la de comprobar el píxel. Se ha detectado que un coche tiene mayor probabilidad de ir marcha atrás en el inicio de la generación que una vez lleve cierto tiempo. Por ello, se ha establecido un margen temporal de 10 segundos. Si en un tiempo menor a 10 segundos el coche detecta la línea de meta, la única razón plausible es que esté yendo en dirección contraria.

```
for i, car in enumerate(cars):

    if car.get_alive():

        remain_cars += 1
        car.update(map,i,random_map)

        genomes[i][1].fitness += car.get_reward()

        warningTime = (datetime.datetime.now() - now)
        if warningTime.total_seconds() < 10 :
            Meta1 = (20 , 77, 0) #
            Meta2 = (163 , 193, 160) #
            Road = (0 , 0, 0) #
            p = car.center
            if ((map.get_at((int(p[0]), int(p[1]))) == Meta1) or map.get_at((int(p[0]), int(p[1]))) == Meta2):

                    car.is_alive = False
                    print(f'car {i} marcha atras')
                    genomes[i][1].fitness = 0       # penalizar si va marcha atras
                    car.rewards = genomes[i][1].fitness
        #            car.check_dies(i, 4)
```

Figura 3.36: Detectar marcha atrás

Una vez detectado este comportamiento del vehículo, será inmediatamente eliminado de la generación y se le dará una recompensa con un valor de 0 para que no pueda propagarse en futuras generaciones.

3.3.2.5. Actualización de las vueltas

Durante todo este entrenamiento, una parte fundamental es contar las vueltas que esta realizando la simulación.

```
for i, car in enumerate(cars):
    num_lap[i] = car.get_lap()
    if car.get_lap() > lap:
            lap = car.get_lap()
```

Figura 3.37: Obtener vueltas

Una vez obtenidas las vueltas que ha dado el coche, buscaremos el valor máximo para que sea este el que aparezca en la pantalla de Pygame.

3.3.2.6. Recompensas, penalizaciones velocidad y giros

Una vez establecido que velocidad y que ángulo de giro tiene el vehículo, el siguiente paso sera actualizar la recompensa, que esta depende del *fitness* acumulado, por ello, para obligar a que el coche no sea propenso a ir a una velocidad constante, o una velocidad muy baja, se ha añadido la condición de que los coches tienen, por obligación, que haber frenado en algún momento.

```
# Update fitness value
for i, car in enumerate(cars):
    car.rewards = genomes[i][1].fitness

for i, car in enumerate(cars):
    if car.number_times_braking == 0:
        genomes[i][1].fitness = 0       # penalizar si no ha frenado
        car.rewards = genomes[i][1].fitness

for i, car in enumerate(cars):
    if ((car.number_times_turned_to_the_right == 0) or (car.number_times_turned_to_the_left)) == 0:
        genomes[i][1].fitness = 0       # penalizar si no ha girado
        car.rewards = genomes[i][1].fitness
```

Figura 3.38: Penalizar recompensas

3.3.2.7. Precisión algoritmo

En esta sección se detallará otra de las partes fundamentales de este proyecto, la búsqueda de una mejor precisión. Para ello distinguiremos 3 etapas, cada etapa tiene un objetivo diferente y se diferenciarán en el número de vueltas y en la cantidad de coches que quedan en el circuito en ese momento.

- **Acumulación de coches:** En esta etapa buscaremos acumular coches en el circuito, ya que una mayor cantidad de coches equivale a una demostración práctica de que el algoritmo funciona para este circuito, por el contrario, podría darse el caso de que un coche fuera bien en un circuito por casualidad.

```
# check
if option_Value != 3 :
    if remain_cars <= 11 and max(num_lap) >= 5 and max(num_lap)<9:
        total_Car = []
        for i, car in enumerate(cars):
            if car.is_alive == True:
                total_Car.append(i)
                car.check_dies(i, 8)
                car.is_alive = False
        print(f"Un total de {len(total_Car)} coches mueren por condicion 1 estos son : {total_Car}")
        remain_cars = 0
```

Figura 3.39: Condición 1 - Acumulación de coches

En este caso, el mínimo de coches que estarán simulando son 12, si hay 11 o menos, la generación se da por terminada. Este mínimo de coches se mantendrá mientras el coche que haya dado más vueltas esté en un valor entre 5 y 9.

- **Mejorar la precisión:** Para el caso siguiente, sólo querremos que el mínimo de coches sean 10, entre las vueltas 9 y 15, está es fase es una mezcla entre buscar precisión y buscar acumulación de coches.

```
if remain_cars <= 9 and max(num_lap) >= 9 and max(num_lap)< 15:
    total_Car = []
    for i, car in enumerate(cars):
        if car.is_alive == True:
            total_Car.append(i)
            car.check_dies(i, 8)
            car.is_alive = False
    print(f"Un total de {len(total_Car)} coches muere por condicion 2 estos son : {total_Car}")
    remain_cars = 0
```

Figura 3.40: Condición 2 - Mejorar precisión

- **Buscar resultados:** Y por último, llegaremos a la última fase, búsqueda de resultados finales. Para ello, en esta ocasión, tendremos que tener un mínimo de 7 coches (elitismo definido en el archivo de configuración más 2), la condición respecto a vueltas es sencilla, dar más de 20 vueltas, teniendo en cuenta que partimos de la anterior etapa, 15, es fácil de conseguir.

```
if remain_cars <= 6 and max(num_lap) >= 20:
    total_Car = []
    for i, car in enumerate(cars):
        if car.is_alive == True:
            total_Car.append(i)
            car.check_dies(i, 8)
            car.is_alive = False
    print(f"Un total de {len(total_Car)} coches muere por condicion 3 estos son : {total_Car}")
    remain_cars = 0
```

Figura 3.41: Condición 3 - Buscar resultados

3.3.2.8. Representar datos en la generación

Para terminar, la última parte a mencionar es la relativa a mostrar gráficamente los datos relativos a los coches, para poder controlar la simulación.

Mediante el código que aparece en la siguiente imagen podemos pintar en el mapa, y actualizar, el número de generación en ese instante, el tiempo que llevamos ejecutando dicha generación, la cantidad de coches que aún se encuentran en el circuito y el número máximo de vueltas que ha realizado un vehículo en la generación.

```
# Drawing
screen.blit(map, (0, 0))
for car in cars:
    if car.get_alive():
        car.draw(screen,remain_cars)

text = generation_font.render("Generation : " + str(generation), True, (255, 128, 0))
text_rect = text.get_rect()
text_rect.center = (screen_width/2, 100)
screen.blit(text, text_rect)

text = font.render("Time : " + str(time), True, (255, 128, 0))
text_rect = text.get_rect()
text_rect.center = (screen_width/2, 170)
screen.blit(text, text_rect)



text = font.render("Remain cars : " + str(remain_cars), True, (255, 128, 0))
text_rect = text.get_rect()
text_rect.center = (screen_width/2, 200)
screen.blit(text, text_rect)


lap = max(num_lap)


text = font.render("Lap : " + str(lap), True, (255, 128, 0))
text_rect = text.get_rect()
text_rect.center = (screen_width/2, 250)
screen.blit(text, text_rect)
pygame.display.flip()
clock.tick(0)
```

Figura 3.42: Actualización de los datos en la pantalla

3.3.3. Archivo de configuración

Se ha optado por una población en cada generación de 100 coches, en lugar de los 30 que aparecían en la anterior versión, puesto que, al realizar el proceso de mutación, es más probable que se lleve a cabo si el número de individuos es mayor. Se ha elegido un valor muy alto de *fitness_threshold* (un millón) para asegurarnos de que realmente aprenda bien el algoritmo. Hemos elegido el criterio de *fitness* Max, de manera que aquel individuo con el mayor valor de *fitness* sirva como base para crear nuevos individuos. También se ha establecido que, si se produce un estancamiento, todas las especies se extingan al mismo tiempo.

```
[NEAT]
fitness_criterion     = max
fitness_threshold     = 1000000
pop_size              = 100
reset_on_extinction   = True
```

Figura 3.43: Configuración de población y umbral

El siguiente apartado a detallar es el correspondiente al genoma. Estableceremos una activación del tipo tangente hiperbólica. También podríamos haber usado la función sigmoidea, pero hemos optado por la primera debido a que el rango de números es mayor, [-1:1] en lugar de [0:1]. Con estos valores, podremos decir, por ejemplo, que si la velocidad es menor que 0, consideraremos que el coche está frenando, y si es mayor que 0, consideraremos que está acelerando.

```
[DefaultGenome]
# node activation options
activation_default      = tanh
activation_mutate_rate  = 0.05
activation_options      = tanh
```

Figura 3.44: Configuración de activación

Para el apartado de parámetros de red, se ha optado por 5 entradas y 2 salidas. Las entradas serán los valores obtenidos por los sensores, por lo tanto, cada entrada representará un sensor. En cuanto a las salidas, como se mencionó en el apartado de Adaptación del algoritmo, también se establece que no se desean nodos ocultos.

```
# network parameters
num_hidden              = 0
num_inputs              = 5
num_outputs             = 2
```

Figura 3.45: Parámetros de red

Para la configuración del umbral de compatibilidad, se ha establecido un valor de 2. En otras palabras, se considerará que los individuos pertenecen a la misma especie siempre y cuando la distancia genómica sea menor a 2.

```
[DefaultSpeciesSet]
compatibility_threshold = 2.0
```

Figura 3.46: Configuración del umbral misma especie

En el caso del estancamiento, se ha decidido eliminar aquellas especies que no muestren mejoras en 10 generaciones, protegiendo a las 2 mejores especies. El criterio para determinar si una especie es mejor o peor está configurado según el valor de *fitness* más alto en la generación.

```
[DefaultStagnation]
species_fitness_func = max
max_stagnation       = 10
species_elitism      = 2
```

Figura 3.47: Configuración estancamiento

Por último, se explicará la configuración realizada en el apartado de reproducción predeterminada. Con esta configuración que se muestra en la imagen, se conservarán las 5 mejores muestras de cada especie sin realizar ninguna modificación para la siguiente generación. También se ha ajustado el umbral de supervivencia a 0.1, en lugar del valor predeterminado de 0.2. Esto significa que esta fracción de cada especie se reproducirá en cada generación para crear nuevos individuos.

```
[DefaultReproduction]
elitism            = 5
survival_threshold = 0.1
```

Figura 3.48: Configuración reproducción

Capítulo 4

Pruebas

Durante la realización del proceso de entrenamiento del algoritmo, se ha experimentado con diferentes configuraciones para poder comparar entre los resultados obtenidos y, a partir de estos, elegir cuál es la configuración óptima. En el presente apartado se irán detallando las distintas configuraciones tomadas así como las conclusiones sacadas en el proceso.

4.1 Sensores

Durante el proceso de entrenamiento se ha probado con diferentes cantidad de sensores para ver cuál es la configuración que mejor se adapta a este problema. Se han utilizado 2, 3, 4, 5, 7, 9 y 11 sensores.

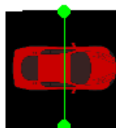


Figura 4.1: Dos sensores laterales

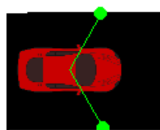


Figura 4.2: Dos sensores frontales.

Para el caso de dos sensores se ha probado de dos formas distintas, dos sensores laterales (uno a cada lado del vehículo) y dos en posición frontal.

Para estas dos ubicaciones se han encontrado ciertas carencias y ciertas ventajas entre si, con lo cual, a partir de estos experimentos, se ha llegado a la conclusión de que ambos sistemas son necesarios y, para casos posteriores, se va a implementar un sistema mixto.

En el caso de tener los sensores laterales, calculaba correctamente la distancia a las paredes, pero no podía calcular correctamente cuánto quedaba hasta el obstáculo frontal y colisionaba. Por otro lado, para el caso de los sensores frontales, calculaba correctamente la toma de curvas, pero al no calcular la distancia a las paredes laterales, era propenso a un colisiones lateral.

El siguiente número a comprobar ha sido con 3 sensores. Esta ha sido una combinación de las ubicaciones de los sensores anteriores, en la cual se mantienen los dos sensores laterales y se agrega uno central. Para este caso, se han obtenido mejores resultados en aquellas curvas que no eran muy pronunciadas. El principal problema de esta distribución era que, al tener el sensor central, no era capaz de distinguir si se producía una curva a la derecha o a la izquierda.

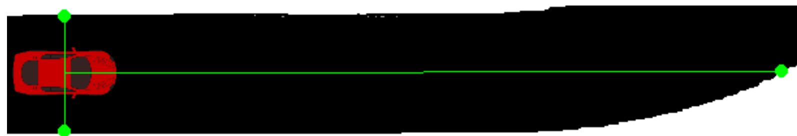


Figura 4.3: Tres sensores.

En vista de los resultados obtenidos con ese número de sensores, se ha optado por incrementar en una unidad la cantidad de radares que se instalarán en el vehículo.

En este caso, se ha mejorado el rendimiento del algoritmo, logrando alcanzar mayores distancias. Con los radares laterales controlamos las paredes del circuito, y con los radares centrales vamos monitoreando la ubicación de las curvas y las distancias.

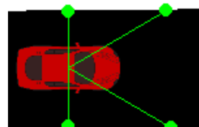


Figura 4.4: Cuatro sensores.

Otra de las configuraciones probadas es con 5 sensores, en los que se repartirían de la siguiente forma: dos laterales, comprobando las distancias de las paredes; uno central, que se encargará de determinar que distancia hay respecto al final de pista; y los dos restantes centro-laterales, que se encargarián de detectar si la curva siguiente es hacia un lado o hacia el otro.

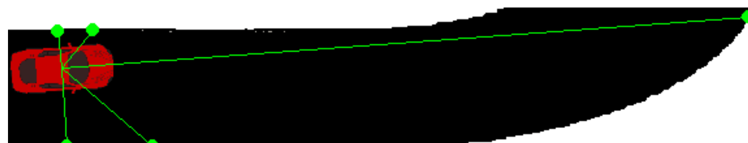
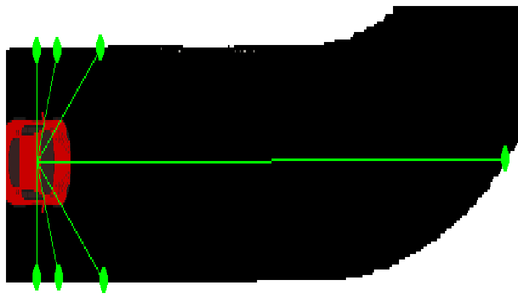
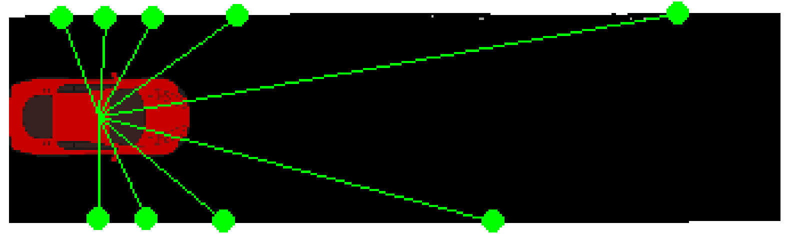


Figura 4.5: Cinco sensores.

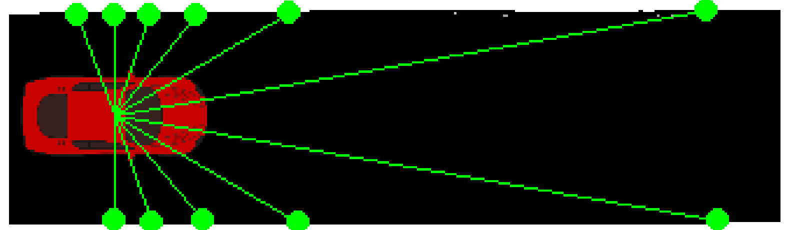
Con esta configuración se consigue una mejora importante con respecto al resto, ya que consigue una mayor probabilidad de realizar una vuelta completa y necesitando menos intentos, tal y como se



(a) Siete sensores.



(b) Nueve sensores.



(c) Once sensores.

Figura 4.6: Diferentes ubicaciones de los dos sensores

menciona en el estudio de sensores en los coches autónomos realizado por la Universidad del Valle de Guatemala [34]

Finalmente se ha ido probando con 7,9 y 11 sensores, estos nuevos sensores se han ido asignando en la parte central del vehículo para mejorar la precisión del coche con respecto a las curvas, a pesar de tener un mayor número de sensores no se ha producido una mejora significativa con respecto a 5 sensores.

4.2 Ajustes en archivo de configuración

En este apartado, predominan 3 diferentes configuraciones que se han realizado. La primera de todas se centra en la población que aparece en la pantalla de entrenamiento. Originalmente, nos encontrábamos con una población de 30, pero se ha aumentado hasta llegar a 100 y se han obtenido mejores resultados. Con esta cantidad, era necesario un menor número de generaciones para tener éxito. Esto se debe a que al tener una población más grande, existe una mayor probabilidad de que aparezca una mutación que resuelva los problemas de la conducción autónoma.

Otro de los parámetros a configurar es el correspondiente a las entradas y salidas de la red neuronal, lo cual se explica en el apartado 4.1 mediante el uso de sensores.

Finalmente, se han realizado pruebas permitiendo un número mayor de generaciones que se permite al algoritmo no mejorar, de 10 a 20. También se ha aumentado el elitismo de 3 a 5 y se ha duplicado el umbral de supervivencia, de 0.1 a 0.2. Con estas condiciones, permitimos una mayor flexibilidad al algoritmo para no eliminar especies funcionales erróneamente, puesto que siguen siendo válidas.

4.3 Mapas

A pesar de que en el programa del cual partimos se poseía un mapa, se ha optado por seguir trabajando en este tema y seguir ampliando el repertorio. Se ha observado que dicho mapa era muy sencillo, ya que las curvas que disponía eran muy suaves y si intentábamos trasladar lo aprendido a otro circuito no se podía garantizar el éxito de la prueba.

Por esta razón, se ha ido modificando ese circuito e implementando otros nuevos. Cada uno dispone de una zona crítica, que puede ser curvas muy cerradas o bien una curva pronunciada después de una recta larga.

Un ejemplo de los diseños que se han realizado ha sido el circuito 2.

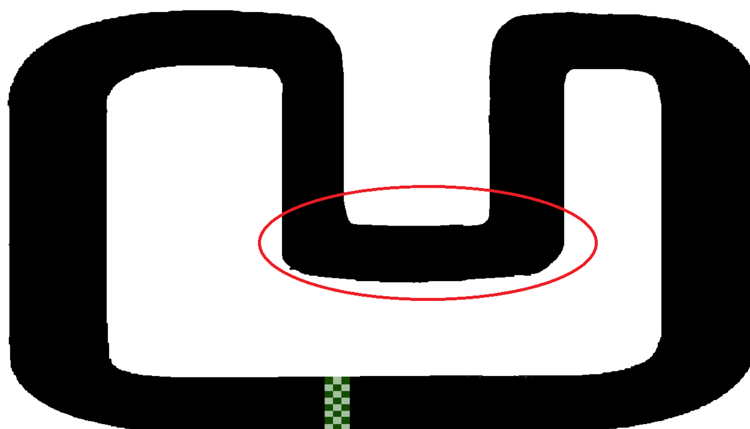


Figura 4.7: Zonas criticas mapa 2.

En él se puede apreciar cómo se ha diseñado una zona del circuito en la cuál tendremos que realizar dos curvas a derechas de 90 grados. En este circuito se ha desechado la idea de tener 3 o menos

sensores, ya que, al tener solo uno central, no hacia distinción entre si la curva era a izquierda o a derecha y seguía recto hasta que chocaba con el borde.

El siguiente circuito a mencionar es el 6. En él, podemos observar cómo hay 3 zonas críticas para los vehículos. Gracias a la primera zona, el algoritmo empieza a hacer distinción entre girar a la derecha y a la izquierda.

La segunda zona esta diseñada para que el coche use el freno, ya que viene de una zona recta en el que alcanzan la máxima velocidad posible. De otro modo, se saldrían del circuito y acabaría la generación.

Para terminar, quedaría detallar la zona critica número 3, en ella se produciría tanto el giro como el controlar la velocidad mínima permitida. Recordemos que si el vehículo alcanza una velocidad negativa, este sería eliminado de la generación en curso.

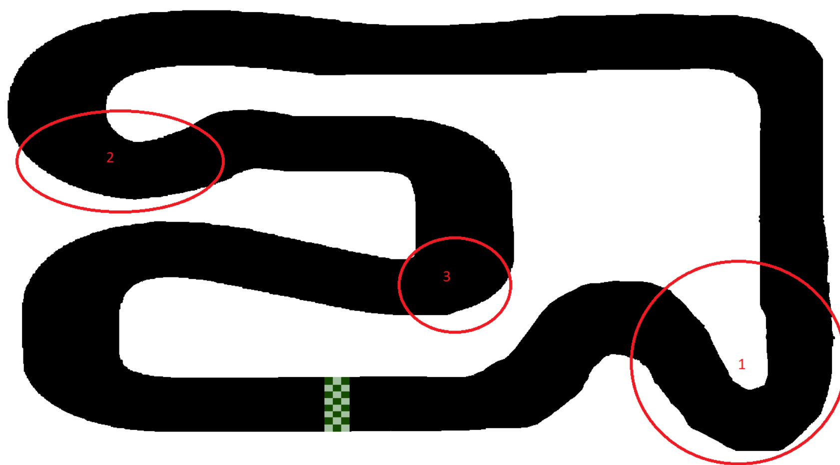


Figura 4.8: Zonas criticas mapa 6.

Por último mencionar el mapa número 10.

Este es el mapa utilizado para el entrenamiento. Este circuito se ha diseñado para englobar todas las casuísticas observadas en todos los mapas anteriores. Podemos observar que, partiendo de la línea de meta, la primera y la segunda zona crítica se asemejan a la zona crítica del mapa 2 mencionado anteriormente, y la tercera y la cuarta zona corresponderían a la zona de frenado y al control de la velocidad mínima que se explicó en el mapa 6.

Al optar por la creación de un circuito genérico que disponga de todas las zonas criticas reportadas en los otros circuitos, conseguimos que, una vez obtengamos una generación que consiga un valor de *fitness* superior al del umbral establecido, podemos ejecutar esa generación en cualquiera de los otros mapas obteniendo una probabilidad de éxito y mejores resultados superior al que obtendríamos si solo ejecutáramos el código en ese circuito en concreto.

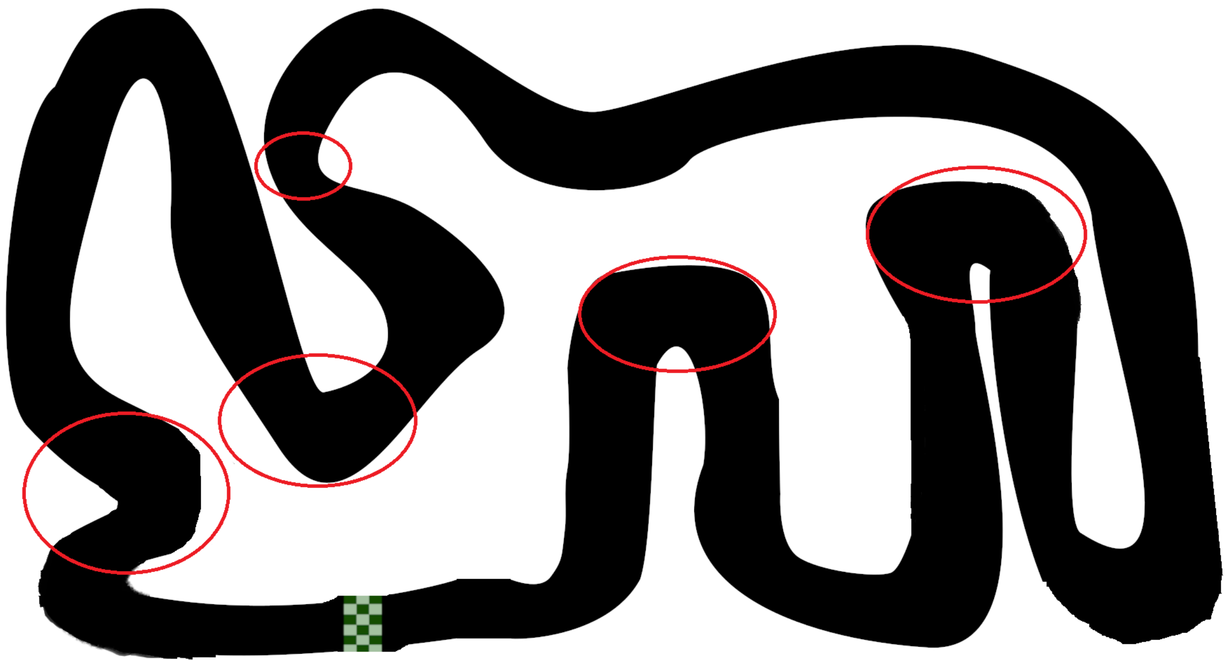


Figura 4.9: Zonas criticas mapa 10.

4.4 Velocidad

Para el aspecto de la velocidad, se ha optado por limitar la velocidad máxima que pueda alcanzar un coche. Esto ha sido por dos razones fundamentales.

La primera reside en la problemática de la tecnología Pygame, el funcionamiento en este juego es tener un mapa e ir repintándolo, esto se traduce en que existe la posibilidad de que en la siguiente iteración no haya existido un evento que si debería de haber existido,como puede ser cruzar la linea de meta. Si la velocidad del coche es mayor que el área designada de la meta, existe una probabilidad de que las iteraciones ocurran antes y después de la meta, y por lo tanto, no incremente la vuelta, ya que no ha detectado el color del píxel correspondiente a la meta.

Mientras que la segunda se centra más en poder controlar el circuito, con el fin de poder reducir las generaciones necesarias para encontrar una solución óptima se ha valorado positivamente establecer una velocidad máxima. Esto surge como respuesta al comportamiento del vehículo en el circuito: era propenso a alcanzar altas velocidades, pero, en consecuencia, se quedaba sin margen de tiempo y espacio para poder frenar y tomar la curva correctamente (ya que el freno está delimitado entre 0 y 1 por iteración). Una vez finalizada esa generación, se producía un bucle en el cual todos los coches se estrellaban en el mismo punto del circuito y nunca se generaba un coche diferente que resolviera esta problemática.

Una de las modificaciones con respecto al código original es la implementación de una velocidad variable. Para ello, se han propuesto una serie de opciones distintas: acelerar/frenar la velocidad con un valor fijo o con un valor variable. Dentro de esta última opción, se puede establecer una velocidad por tramos (según el valor de salida) o directamente un valor entre 0 y 1 (por ser salida tangente

hiperbólica).

En primer lugar, la velocidad con variaciones constantes provocaba que no siempre se realizaran las vueltas con la mayor velocidad posible, puesto que al no tener la velocidad idónea en esa parte del circuito, provocaba que o bien entrara a una curva con una velocidad mayor de la que debería y se saliera del circuito, o bien realizaba la vuelta pero no conseguía establecer un tiempo de vuelta óptimo.

Debido a esta problemática, se ha optado por la realización de un valor que se incremente o decremente según la salida de la red neuronal. En este proceso se ha considerado la opción de dividir por rangos las velocidades, de manera que una cifra mayor ponderaría de mayor forma el valor de la velocidad. Esta opción fue escogida para una activación de tipo sigmoidea, debido a que se partía de un rango de 0.5 (0-0.5 frenar y 0.5-1 acelerar).

Salida	Velocidad actual	Nueva velocidad
$x < 0$	No influye	Velocidad $-X$
$x = 0$	No influye	No varía
$x > 0$	$0 < X \leq 10$	Velocidad $+X$
$x > 0$	$X > 10$	No varía

Cuadro 4.1: Configuración de la velocidad

Como finalmente se decidió una salida de tipo tangente hiperbólica, ya se disponía de un rango mayor, por lo que ya no era necesario dividir la velocidad por tramos. En su lugar, el atributo velocidad vendrá incrementado o decrementado directamente por la salida de la red neuronal, es decir, si tenemos una salida de 0.7, la nueva velocidad será igual a la suma de la velocidad que teníamos almacenada más el nuevo valor.

Nº Coche	Nº vuelta	Tiempo
20	1	0:00:49
20	2	0:00:44
20	3	0:00:43
20	5	0:00:43
20	10	0:00:44
20	15	0:00:44
20	20	0:00:39
20	30	0:00:37
20	40	0:00:31
20	50	0:00:33

Cuadro 4.2: Evolución de tiempos por vuelta usando velocidad variable

4.5 Ángulo de giro

Paradójicamente, se han obtenido mejores resultados al implementar el ángulo de giro en forma de tramos en lugar de hacerlo directamente, como ocurría con la velocidad.

Se realizaron las mismas tres implementaciones y a continuación se describirán en detalle cómo se llevaron a cabo, tal como se mencionó en la sección anterior.

En el caso de una implementación directa, no se obtuvieron buenos resultados. La razón fue similar a cuando no había límite de velocidad: no había suficiente tiempo ni espacio para realizar el giro. En un coche, un giro de un máximo de 1 grado en una iteración es prácticamente moverse en línea recta, por lo que nunca se llegaba a realizar una curva.

En este caso, se partió de la base de que cada iteración equivalía a un giro de $\pm$ 15º, lo que provocaba que el vehículo tuviera una posición poco estable, ya que en cada iteración oscilaba 15 grados de derecha a izquierda y viceversa.

Se probaron diferentes ángulos, y los dos que obtuvieron mejores resultados fueron 5º y 6º. Aunque el coche seguía oscilando en cada iteración, esta oscilación se producía de manera mucho más suave que antes.

Finalmente, tras realizar pruebas en distintos mapas, se eligió una configuración de 6º como la más adecuada. Esto permitía un mayor porcentaje de éxito en aquellos mapas en los que había curvas muy cerradas y estaban separadas por una corta distancia entre ellas.

Salida	**Modificación giro**
$-1 < X \leq -0{,}9$	-6
$-0{,}9 < X \leq -0{,}7$	$-4{,}8$
$-0{,}7 < X \leq -0{,}5$	$-3{,}6$
$-0{,}5 < X \leq -0{,}4$	-3
$-0{,}4 < X \leq -0{,}3$	$-2{,}4$
$-0{,}3 < X \leq -0{,}0$	$-1{,}2$
0	0
$0{,}0 > X \leq 0{,}3$	$1{,}2$
$0{,}3 > X \leq 0{,}4$	$2{,}4$
$0{,}4 > X \leq 0{,}5$	3
$0{,}5 > X \leq 0{,}7$	$3{,}6$
$0{,}7 > X \leq 0{,}9$	$4{,}8$
$0{,}9 > X \leq 1$	6

Cuadro 4.3: Configuración del ángulo de giro

4.6 Distintas funciones

Para una mayor precisión, se han añadido una serie de funciones y se han comparado los rendimientos obtenidos antes y después de las modificaciones.

En primer lugar, se ha añadido una función para detectar si el coche se encuentra circulando avanzando por el circuito o, por el contrario, se encuentra girando sobre sí mismo. La segunda posibilidad provocaba una contaminación de las muestras, ya que, a efectos de distancia, esta seguía incrementando pero no era distancia útil debido a que no avanzaba. Como consecuencia, podría darse el caso de que el susodicho tuviera un mayor *fitness* que aquel que hubiera avanzado más en el circuito, y en futuras generaciones, se tomara este como referencia. Por ello, se ha optado por monitorizar el tiempo y, si no avanza, eliminarlo.

Respecto al número de vueltas, se ha optado por incluirlo en el proyecto (y en el cálculo de la recompensa). Con esta implementación, se ha conseguido filtrar por el número de vueltas realizadas en el circuito y no por la distancia total recorrida, ya que no existía una relación tan directa como podía parecer en un principio. Esto se debe a que existía un caso en el que tener una mayor distancia no implicaba haber realizado un mayor número de vueltas, y este caso es el de realizar el recorrido tomando las curvas muy abiertas.

Por último, se han realizado diferentes experimentos con relación a las condiciones que se mencionaron en el apartado 3.2.

En la tabla de abajo podemos apreciar cómo varía el algoritmo en función de si estas condiciones son o no aplicadas. Para los datos de la izquierda no se aplica, y para los de la derecha sí se aplica. También, la comparativa es según los mapas.

A través de esta tabla, podemos apreciar que existe una mejora generalizada en cada mapa del circuito, ya que se realizan más vueltas y en mayor número de individuos.

La consecuencia directa al emplear estas condiciones es que se necesita un mayor tiempo para alcanzar los objetivos. Mientras que sin condiciones tardaba unas 2 o 3 horas, con las condiciones se alcanzaban unas 30 horas. Sin embargo, mejora el número de individuos en un promedio de un 30 % a excepción de los dos primeros mapas, que empeora. Pero aún así, se dispone de una muestra de éxito bastante elevada, ya que partíamos de una población de 100 individuos.

SIN CONDICIONES		CON CONDICIONES	
Mapa	**Resultado**	**Mapa**	**Resultado**
1	15 vueltas y 39 aún en pista	1	20 vueltas y 35 aún en pista
2	20 vueltas y 39 aún en pista	2	25 vueltas y 35 aún en pista
3	26 vueltas y 20 aún en pista	3	30 vueltas y 31 aún en pista
4	21 vueltas y 21 aún en pista	4	25 vueltas y 30 aún en pista
5	25 vueltas y 18 aún en pista	5	30 vueltas y 31 aún en pista
6	27 vueltas y 21 aún en pista	6	30 vueltas y 31 aún en pista
7	27 vueltas y 11 aún en pista	7	35 vueltas y 22 aún en pista
8	12 vueltas y 6 aún en pista	8	32 vueltas y 14 aún en pista
9	31 vueltas y 11 aún en pista	9	30 vueltas y 31 aún en pista
10	28 vueltas y 7 aún en pista	10	30 vueltas y 12 aún en pista

Cuadro 4.4: Comparativa entre resultados según la disponibilidad o no de las condiciones de precisión

Capítulo 5

Conclusiones y líneas futuras

5.1 Conclusiones

Durante el desarrollo de este Trabajo Fin de Máster, además de las correspondientes pruebas en este proyecto, se pueden obtener las siguientes conclusiones:

- Para abordar un proyecto técnico, primero es necesario estudiarlo detenidamente y buscar trabajos relacionados, buscar pautas, puntos fuertes y debilidades.

- El uso de algoritmos que utilizan aprendizaje por refuerzo está en auge debido a su efectividad, ya que permiten descubrir enfoques en los que los humanos no han podido implementarlos o no han podido pensarlos. Estos algoritmos han tenido casos de éxito ampliamente reconocidos, como el de AlphaDev, que produjo un código más eficiente en el algoritmo de ordenamiento (*Sort*) [35], o el caso de la IA creada por la compañía *DeepMind* denominado ***AlphaStar*** en la que la propia IA derrotó en una partida a profesionales de *StarCraft* por 10-1[36].

- Por la naturaleza de los algoritmos genéticos, se pueden resolver de manera más efectiva los problemas complejos gracias a su adaptabilidad y a la paralelización.

- Es necesario utilizar como base un mapa de calidad, es decir, lo más completo posible, en el que se alberguen todos los casos conocidos de fallo, de manera que una vez entrenado correctamente en este mapa, se pueda exportar a otro circuito y tener una probabilidad alta de éxito.

5.2 Conocimientos adquiridos

En el proceso de desarrollo del proyecto, el alumno ha adquirido los siguientes conocimientos:

- Estudio de las bases teóricas del aprendizaje por refuerzo, así como las arquitecturas más utilizadas en ese campo.

- Conocimiento y utilización del algoritmo de NEAT (Algoritmo de Neuroevolución de Topologías Aumentadas) y su implementación en los videojuegos.

- Desarrollo de videojuegos usando la biblioteca Pygame y el concepto de detección de colisiones en los videojuegos.

- Conocimiento de la conducción autónoma, incluyendo sus fortalezas y debilidades.

5.3 Líneas futuras

Una vez realizado este proyecto, se abren varias líneas futuras para su ampliación.

- Continuar ajustando los valores del ángulo de giro para evitar las oscilaciones que realiza el vehículo. Encontrar el valor óptimo que permita al vehículo girar en todos los circuitos sin comprometer la viabilidad del algoritmo, evitando oscilaciones indeseadas. Si el ángulo de giro no es suficiente, el coche acabará chocando contra el obstáculo o la curva.

- Integrar obstáculos fijos y móviles en el circuito, ya que en un entorno con tráfico realista se presentan estos desafíos. Los obstáculos fijos pueden interpretarse como baches, conos de tráfico, resaltos o señales de tráfico, mientras que los obstáculos móviles pueden ser otros vehículos, peatones o semáforos.

- Exportar este modelo a simuladores de entrenamiento más complejos que Pygame que utilicen 3D como por ejemplo Unity. Con esto podremos incrementar la complejidad del algoritmo simulando la conducción autónoma de una forma más realista, permitiendo simular situaciones más complejas y realistas de conducción autónoma, incluyendo la interacción con el tráfico, variables ambientales y sensores.

- Exportar la configuración del proyecto para integrarlo en un modelo que permita su puesta en práctica. Para ello nos serviremos de la configuración guardada a través de la clase "Checkpointer" (**save_checkpoint**). Con ese guardado se serializa y se almacena en un archivo el estado que presenta la red en ese momento, así como la población y las especies. El siguiente paso es cargar este archivo en el dispositivo en el que se quiere realizar las pruebas de campo utilizando la función **load_checkpoint**.

Este tema permite una multitud de posibilidades, simplemente se han expuesto algunas ideas que han surgido a partir de los resultados obtenidos en las simulaciones y se han tratado en los trabajos relacionados. Finalmente, no se han incluido en este proyecto debido a su magnitud y a la tecnología disponible. También es probable que existan distintas posibilidades que no se han planteado en este proyecto y que se irán descubriendo a medida que se avance en el trabajo.

Capítulo 6

Bibliografía

[1] Anya Reiser. «History of Autonomous Cars». En: *TOMORROW'S WORLD TODAY®* (4 de abr. de 2023). URL: https://www.tomorrowsworldtoday.com/2021/08/09/history-of-autonomous-cars/.

[2] Pablo G. Bejerano. «Futurama o cómo se veía el futuro en 1939». En: (7 de mayo de 2014). URL: https://www.eldiario.es/tecnologia/diario-turing/futurama-1939_1_4894686.html.

[3] *Essay*. URL: http://selfdrivingcars.weebly.com/essay.html.

[4] FELIPE JIMÉNEZ ALONSO. *Pasado, presente y futuro del coche autónomo*. 2017. URL: https://riuma.uma.es/xmlui/bitstream/handle/10630/13659/Conferencia.pdf?sequence=1.

[5] RoboticsBiz. «History of autonomous vehicles; Timeline». En: *RoboticsBiz* (1 de jul. de 2021). URL: https://roboticsbiz.com/history-of-autonomous-vehicles-timeline/.

[6] *Automated Vehicles for Safety — NHTSA*. URL: https://www.nhtsa.gov/technology-innovation/automated-vehicles-safety.

[7] *Conducción autónoma — Niveles y tecnología*. URL: https://www.km77.com/reportajes/varios/conduccion-autonoma-niveles.

[8] Ibáñez. «De 0 a 5: cuáles son los diferentes niveles de conducción autónoma, a fondo». En: *Xataka* (2 de oct. de 2018). URL: https://www.xataka.com/automovil/de-0-a-5-cuales-son-los-diferentes-niveles-de-conduccion-autonoma.

[9] Amazon Web Services. *¿Qué es la inteligencia artificial (IA)?* URL: https://aws.amazon.com/es/machine-learning/what-is-ai/.

[10] F. Rosenblatt. «The perceptron: A probabilistic model for information storage and organization in the brain.» En: *Psychological Review* 65.6 (1958), págs. 386-408. ISSN: 0033-295X. DOI: 10.1037/h0042519. URL: http://dx.doi.org/10.1037/h0042519.

[11] Cybercaronte. *Historia de la IA: Frank Rosenblatt y el Mark I Perceptrón, el primer ordenador fabricado específicamente para crear redes neuronales en 1957*. URL: https://web.archive.org/web/20180722124753/https://data-speaks.luca-d3.com/2018/07/historia-de-la-ia-frank-rosenblatt-y-el.html#.

[12] David E. Rumelhart, Geoffrey E. Hinton y Ronald J. Williams. «Learning representations by back-propagating errors». En: *Nature* 323.6088 (oct. de 1986), págs. 533-536. ISSN: 1476-4687. DOI: 10.1038/323533a0. URL: https://doi.org/10.1038/323533a0.

[13] Pedro Antonio Sánchez Romero. *Aplicación de algoritmos Machine Learning para un vehículo autónomo*. Abr. de 2021. URL: https://repositorio.upct.es/xmlui/bitstream/handle/10317/9357/tfg-san-apl.pdf.

[14] B. F. Skinner. «Superstition in the pigeon.» En: *PubMed* 38.2 (1 de abr. de 1948), págs. 168-72. DOI: 10.1037/h0055873. URL: https://pubmed.ncbi.nlm.nih.gov/18913665.

[15] Richard S. Sutton y Andrew G. Barto. *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: MIT Press, 1998. ISBN: 0-262-19398-1. URL: http://www.cs.ualberta.ca/%7Esutton/book/ebook/the-book.html.

[16] *Classics in the History of Psychology – Pavlov (1927) Lecture 6*. URL: http://psychclassics.yorku.ca/Pavlov/lecture6.htm.

[17] Yu-Fen Zhang, Qun-Feng Zhang y Rui-Hua Yu. «Markov property of Markov chains and its test». En: *2010 International Conference on Machine Learning and Cybernetics*. Vol. 4. 2010, págs. 1864-1867. DOI: 10.1109/ICMLC.2010.5580952.

[18] Arhum Ishtiaq et al. «Comparative Study of Q-Learning and NeuroEvolution of Augmenting Topologies for Self Driving Agents». En: *arXiv (Cornell University)* (19 de sep. de 2022). DOI: 10.48550/arxiv.2209.09007. URL: http://arxiv.org/abs/2209.09007.

[19] Paul Pauls. «A Primer on the Fundamental Concepts of Neuroevolution». En: (13 de dic. de 2021). URL: https://towardsdatascience.com/a-primer-on-the-fundamental-concepts-of-neuroevolution-9068f532f7f7.

[20] Vijini Mallawaarachchi. «Introduction to Genetic Algorithms — Including Example Code». En: (1 de mar. de 2020). URL: https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3.

[21] Rodrigo Gamba Lavín. *Implementación del algoritmo NEAT (neuroevolución por topologías aumentadas) para el control de un sistema caótico*. 2019. URL: https://repositorio.unam.mx/contenidos/3483112.

[22] Dhimas Dwi Cahyo. «Simulasi Self-Driving Car dengan Reinforcement Learning dan NeuroEvolution of Augmenting Topologies (NEAT)». En: *JATISI: Jurnal Teknik Informatika dan Sistem Informasi* 9.3 (13 de sep. de 2022), págs. 1752-1761. DOI: 10.35957/jatisi.v9i3.2154. URL: https://doi.org/10.35957/jatisi.v9i3.2154.

[23] Sainath G et al. «Application of Neuroevolution in Autonomous Cars». En: *arXiv (Cornell University)* (26 de jun. de 2020). DOI: 10.48550/arxiv.2006.15175. URL: http://arxiv.org/abs/2006.15175.

[24] Mihir M Parmar et al. «Self-Driving Car». En: *International Journal for Research in Applied Science and Engineering Technology* 10.4 (30 de abr. de 2022), págs. 2305-2309. DOI: 10.22214/ijraset.2022.41786. URL: https://doi.org/10.22214/ijraset.2022.41786.

[25] Chuttumanil Manu Samuel. «Self-Driving Cars using Genetic Algorithm». En: *International Journal for Research in Applied Science and Engineering Technology* (30 de nov. de 2020). DOI: 10.22214/ijraset.2020.32200. URL: https://doi.org/10.22214/ijraset.2020.32200.

[26] NeuralNine. *GitHub - NeuralNine/ai-car-simulation: A simple self-driving AI car game, which uses NEAT.* URL: https://github.com/NeuralNine/ai-car-simulation.

[27] Jackiebarman. *GitHub - Jackiebarman/Remote Control Car: Self driving and remote control car : AI based project using NEAT-Python*. URL: https://github.com/Jackiebarman/Semote_Control_Car.

[28] Anton Von. *Building a self-driving Trackmania car with Deep Learning and computer vision*. 2020. URL: https://antonin.cool/trackmania-ia-deeplearning-python-opencv-self-driving/.

[29] Gniziemazity. *GitHub - gniziemazity/Self-driving-car*. URL: https://github.com/gniziemazity/Self-driving-car.

[30] Red. *GitHub - red42/HTML5-Genetic-Cars: A genetic algorithm car evolver in HTML5 canvas.* URL: https://github.com/red42/HTML5_Genetic_Cars.

[31] Diego Berrocal Gutiérrez. «Inteligencia computacional para el guiado de vehículos autónomos». No Publicado. Boadilla del Monte, nov. de 2022. URL: https://oa.upm.es/72877/.

[32] dDevTech. *Autonomous Driving Tutorial*. 24 de mar. de 2019. URL: https://tutorials.retopall.com/index.php/2019/03/01/autonomous-driving-simulation/.

[33] Kenneth O. Stanley y Risto Miikkulainen. «Evolving Neural Networks through Augmenting Topologies». En: *Evolutionary Computation* 10.2 (2002), págs. 99-127. DOI: 10.1162/106365602320169811

[34] Oscar Juárez y José Cifuentes. *Oscar Juárez y José Cifuentes*. 25 de nov. de 2020. URL: https://github.com/OJP98/self-driving-car.

[35] Daniel J. Mankowitz et al. «Faster sorting algorithms discovered using deep reinforcement learning». En: *Nature* 618.7964 (7 de jun. de 2023), págs. 257-263. DOI: 10.1038/s41586-023-06004-9. URL: https://doi.org/10.1038/s41586-023-06004-9.

[36] Oriol Vinyals et al. «Grandmaster level in StarCraft II using multi-agent reinforcement learning». En: *Nature* 575.7782 (30 de oct. de 2019), págs. 350-354. DOI: 10.1038/s41586-019-1724-z. URL: https://doi.org/10.1038/s41586-019-1724-z.