UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería Informática

Entrenamiento de una Red Neuronal en un entorno interactivo mediante Neuroevolución.

Trabajo Fin de Grado

Grado en Ingeniería Informática

AUTOR/A: Ordoño Saiz, Álvaro

Tutor/a: Sapena Vercher, Oscar

CURSO ACADÉMICO: 2023/2024

Resumen

Este proyecto se centra en el entrenamiento de una red neuronal mediante neuroevolución dentro de una simulación de fútbol sala. El fin de este proyecto es que la red neuronal adquiera la inteligencia espacial suficiente para vencer a un humano en la simulación. Además se desarrollarán mejoras novedosas en el algoritmo de neuroevolución y en el proceso de entrenamiento.

Palabras clave: aprendizaje profundo, neuroevolución, redes neuronales, algoritmos genéticos, aprendizaje por refuerzo, aprendizaje supervisado.

Abstract

This project focuses on training a neural network through neuroevolution within a football simulation. The goal of this project is for the neural network to acquire sufficient spatial intelligence to defeat a human in the simulation. Additionally, novel improvements will be developed in the neuroevolution algorithm and the training process.

Keywords: deep learning, neuroevolution, neural networks, genetic algorithms, reinforcement learning, supervised learning.

Índice de contenidos

Índice de imágenes

Índice de código

1. Introducción

En este proyecto se diseñará una simulación de un partido de fútbol sala empleando las librerías de Pygame. Se desarrollará y vinculará a él un script de entrenamiento de redes neuronales empleando las librerías de neuroevolución de Neat, para que la simulación pueda ser usada como un entorno de selección natural. Y finalmente se entrenará en ella a una población de redes neuronales profundas para obtener al mejor individuo, uno con suficiente inteligencia espacial como para aprender a jugar al deporte y vencer en él a un ser humano. Además, se explorarán técnicas que puedan mejorar el proceso de entrenamiento o añadir nuevas posibilidades a este.

1.1 Motivación

Tal y como se explica en el artículo [20] "Designing neural networks through neuroevolution", con la explosión en popularidad de las redes neuronales profundas, se han rescatado métodos de aprendizaje automático formulados con bastante anterioridad, que no pudieron explotar su potencial debido a la poca complejidad de las redes neuronales del momento.

De entre ellos, recientemente se ha puesto el foco sobre el campo de la Neuroevolución, que busca imitar computacionalmente los mecanismos de reproducción del material genético y la selección natural del mejor individuo, y aplicarlos al diseño de una red neuronal.

Existe una nueva tendencia de explorar estos algoritmos neuroevolutivos para el desarrollo de redes neuronales profundas, especialmente en el campo del aprendizaje por refuerzo.

Por ese motivo, este proyecto busca la introducción del alumno en los campos del aprendizaje por refuerzo y la neuroevolución de una forma práctica, progresando desde cero y sin experiencia previa. Estas áreas supondrán para el alumno no sólo un nuevo campo del conocimiento en inteligencia artificial, también un refuerzo directo y un abanico de nuevas herramientas a aplicar en el aprendizaje profundo tradicional.

Además supone una profundización y ampliación en los conocimientos del alumno en: Redes neuronales, algoritmos genéticos, aprendizaje automático, y desarrollo con Python en general.

Estos conocimientos fueron adquiridos por dos medios:

- Durante sus estudios en la rama de Computación, dentro de la carrera de Ingeniería Informática.
- Durante sus prácticas de empresa como Investigador en Inteligencia Artificial donde diseñó redes neuronales para el diseño inverso de meta superficies mediante aprendizaje profundo.

Estos conocimientos no solo cimentan o amplían sus bases sino que abren la puerta a la hibridación de las técnicas aprendidas en este trabajo con las ya conocidas por la formación del

alumno en su futuro de investigación o desarrollo. En el apartado de “Estado del Arte” se muestran multitud de ejemplos de hibridación de neuroevolución con aprendizaje por refuerzo y supervisado.

Al mismo tiempo, este proyecto supone también una oportunidad para añadir pequeñas mejoras en el algoritmo de Neat y explorar nuevas técnicas de entrenamiento, fomentando la creatividad y el pensamiento crítico del alumno en el área. Si la calidad de las nuevas ideas es suficiente, tal vez hasta puedan aportar al campo, o al menos, servir de base para nuevos proyectos o investigaciones del alumno.

De igual forma este proyecto propone el reto de desarrollar un pequeño videojuego desde cero y apenas sin herramientas de motor, lo que supone una manera de entrenamiento de las habilidades de programación del alumno.

1.2 Objetivos

En esta sección mostramos objetivos específicos u operativos agrupados en objetivos generales:

Diseñar un entorno de simulación de fútbol sala en Pygame que implique físicas relativamente complejas:

- Instalar y evitar errores de compatibilidad entre las librerías de Pygame y Neat
- Adquirir conocimientos de Pygame mediante el desarrollo de ejemplos sencillos
- Desarrollar los elementos del partido (Jugador, portería, pelota, paredes) y vincular su funcionamiento en un script principal.
- Construir un sistema de colisiones desde cero
- Elaborar mecánicas de velocidad, aceleración y deceleración realistas.
- Crear un mecanismo de rebotes realista.
- Establecer la lógica del partido.
- Disponer la información importante de manera accesible para el script de entrenamiento
- Mantener el código ordenado y fácilmente accesible para el script de entrenamiento
- Comprobar que ninguno de los anteriores elementos resulta en errores que puedan afectar al entrenamiento de las futuras redes neuronales, y solucionar dichos errores

Desarrollar un script de entrenamiento de redes neuronales empleando las librerías de neuroevolución de Neat, para que la simulación pueda ser usada como un entorno de selección natural:

- Adquirir conocimientos de Neat mediante el desarrollo de ejemplos sencillos.
- Diseñar un script altamente modificable con facilidad.
- Crear una población inicial de genomas desde cero en base a una configuración personalizada de los aspectos de Neat.
- Poder obtener de forma alternativa una población inicial a partir de un *checkpoint*, almacenando en un entrenamiento previo de forma regular para poder reanudar o revertir los entrenamientos.

- Introducir mecanismos de monitorización del estado de la población, la adquisición del fitness y la evolución del entrenamiento.
- Almacenar al mejor individuo después de un proceso de entrenamiento.
- Establecer una forma de enfrentar a humano y IA de forma simultánea.
- Vincular el script de entrenamiento con el entorno de simulación, pudiendo simular los partidos, controlar a los jugadores, recopilar la información necesaria, etc.

Entrenar una población de redes neuronales profundas para obtener un individuo suficientemente inteligente como para vencer a un humano:

- Observar y entender (La red neuronal) la posición propia y de la pelota en los dos ejes, vertical y horizontal, y deducir la interrelación de estos dos en forma de distancia y posición relativa.
- Aprender (La red neuronal) el concepto de movimiento y controlar a su jugador para aproximarse a la pelota, asociando cada output al movimiento en una dirección.
- Comprender (La red neuronal) las ideas de velocidad y aceleración y relacionar como estas afectan a su movimiento y distancia con la pelota.
- Establecer (La red neuronal) la relación entre su contacto con la pelota y el impulso que su *chut* provoca en ella
- Ser capaz (La red neuronal) de marcar un gol, entendiendo de nuevo el concepto de posición de la portería y la distancia y posición relativa de la pelota a la portería y de ella a la pelota.
- Entender el concepto de equipo y la diferenciación entre porterías
- Apreciar (La red neuronal) los elementos estáticos de su entorno, como las paredes y la portería, y entender cómo estos afectan a su movimiento.
- Descubrir (La red neuronal) los efectos del rebote de la pelota en los elementos estáticos.
- Reconocer el papel del jugador del equipo contrario, su posición relativa a él, la pelota y las porterías y aprender a sortearlo.
- Sortear obstáculos en el terreno y realizar goles con ángulo.

Explorar técnicas que puedan mejorar el proceso de entrenamiento o añadir nuevas posibilidades a este:

- Técnicas de aprendizaje con complejidad progresiva.
- Técnicas que expongan a los individuos a diversas situaciones no alcanzables en un entrenamiento rápido.
- Técnicas que permitan crear redes de diferentes funcionalidades, al estilo de las posiciones en el fútbol.
- Técnicas que permitan entrenar la red contra inteligencias adversas, humanas o artificiales.
- Técnicas que aceleren el proceso de entrenamiento o lo paralelicen.

1.3 Metodología

Para organizar el proyecto hemos aplicado la metodología ágil Scrum. Aunque esta está pensada para el trabajo en equipo, y este proyecto es individual, fue escogida debido a la naturaleza flexible y evolutiva del proyecto, donde es difícil estimar los tiempos o las tareas necesarias.

Esta opción se centra en la flexibilidad, permitiendo cambiar los requisitos del proyecto durante el desarrollo del mismo, función que empleamos en repetidas ocasiones dado que el proyecto necesitó de diversas reestructuraciones de sprints como se muestra en el apartado de "Plan de Trabajo"

Esta metodología se divide en fases de implementación aplicadas en cada uno de los *sprints*:

- Planificación: Cada sprint es un periodo corto de longitud temporal similar donde se agrupan diversas tareas con una misma finalidad. Estos son definidos en esta primera fase. Durante este paso, se presentan los objetivos y se detallan las tareas a desarrollar, se identifican posibles riesgos, se establecen los plazos de entrega y se definen las pruebas de aceptación.
- Desarrollo: En esta fase, se implementan todas las tareas establecidas en la fase de planificación y se realizan las pruebas de aceptación correspondientes, dentro del período designado para el sprint.
- Revisión: Al finalizar el sprint, se lleva a cabo una evaluación personal, dada la naturaleza individual del trabajo, donde se repasan los avances del proyecto y se analiza el proceso de trabajo.
- Retroalimentación: Se aplican las lecciones aprendidas durante las fases de desarrollo y revisión, revisando posibles ajustes en la planificación de futuros sprints para integrar los nuevos conocimientos adquiridos.

Estas fases resultaron de gran ayuda durante el desarrollo del proyecto, especialmente la fase de revisión y retroalimentación. En un trabajo introductorio para el alumno como este, el conocimiento ganado durante los *sprints* supuso la adición de nuevas tareas en futuros *sprints* para implementar funcionalidades extra. En ocasiones estas funcionalidades extra nacieron precisamente de la retroalimentación debida a los problemas en el desarrollo.

Los roles comunes en la metodología Scrum (Scrum Master, Product Owner, Development Team) no han sido necesarios, dado que no existe un cliente para este proyecto y el equipo de desarrollo lo conforma un solo alumno.

1.4 Colaboraciones

El entorno Pygame original fue realizado junto a Adrián Soriano Rodríguez, que en el momento de publicación de esta memoria trabaja en un TFG análogo a este centrado en aprendizaje por refuerzo.

1.5 Estructura de la memoria

- El primer capítulo tratará de una **introducción** al proyecto. En el se expondrá las motivaciones para su realización, los objetivos establecidos con él y la metodología de trabajo aplicada.
- El segundo capítulo lo compone una serie de **fundamentos teóricos** sobre los que se fundamente el proyecto, útiles para el correcto entendimiento de los conceptos expuestos durante esta memoria
- El tercer capítulo es una revisión del **estado del arte** de la neuroevolución y el algoritmo de neuroevolución empleado, NEAT, en términos de evolución, logros recientes e implementaciones a comparar con nuestro trabajo.
- El cuarto capítulo aborda el **análisis del problema**, identificando las oportunidades vinculadas a este proyecto, explicando la solución propuesta, especificando los requisitos funcionales y no funcionales de la propuesta y exponiendo el plan de trabajo seguido por el alumno.
- El quinto capítulo se centra en el **diseño de la solución**, donde se expone el diseño de arquitectura y diseño detallado de los elementos más importantes del proyecto. Además se analizan las herramientas de posible empleo para nuestro proyecto y se argumenta la decisión final.
- El sexto capítulo comprende el **desarrollo de la solución,** explicando la implementación de los componentes mas importantes o complejos del proyecto, las dificultades surgidas y su solución.
- En el séptimo capítulo se resumen las **pruebas** realizadas por el humano y un grupo de ayudantes para medir la inteligencia de la red y el buen funcionamiento del proyecto.
- Finalmente, durante el séptimo, octavo y noveno capítulo se presentarán las **conclusiones** del proyecto, la **relación con los estudios** cursados por el alumno y los posibles **trabajos a futuro** derivados de él. En ellos se mostrarán los conocimientos afianzados gracias al proyecto, los problemas y caminos a evitar, y las futuras ideas o líneas de trabajo a poder realizar.

2. Fundamentos teóricos

La aparentemente irreplicable inteligencia del cerebro humano, objeto de intentos de imitación desde los inicios de la computación, es fruto de millones de años de selección natural: En el hostil ambiente de la naturaleza, sobrevivía el mejor genoma capaz de codificar la estructura cerebral más apta para un entorno salvaje, y este genoma pasaba a la siguiente generación por medio de la reproducción con otro individuo similarmente apto, con quien se producía una mezcla y mutación del material genético. Es solo gracias a esa hostilidad y continua adaptación que se logró el hito de la inteligencia biológica.

Imitando este proceso, pero a mucha menor escala, en este proyecto pretendemos emplear algoritmos de neuroevolución para simular la selección natural, y encontrar el mejor genoma (artificial) capaz de codificar la red neuronal más apta para jugar y ganar un partido de fútbol sala. Idealmente, lograremos obtener una inteligencia que rivalice o supere a la humana en esta prueba.

A continuación presentamos los fundamentos teóricos sobre los que se sustenta la neuroevolución, destacando su importancia en nuestro proyecto, en orden descendente según la especificidad, desde el concepto mismo de inteligencia artificial hasta la subrama neuroevolutiva. Observaremos en dichos fundamentos cómo en múltiples ocasiones mecanismos propios de la naturaleza son empleados para alcanzar nuevas cotas de inteligencia artificial. Cómo reza la frase popular “El arte imita a la vida”:

2.1 Inteligencia artificial

La inteligencia artificial, rama de la computación, se centra en replicar el funcionamiento de la inteligencia humana mediante modelos para resolver tareas complejas. La I.A. se divide en varias subdisciplinas, aunque nosotros profundizaremos en el aprendizaje automático.

2.2 Aprendizaje automático

El aprendizaje automático se enfoca en desarrollar modelos de IA que resuelvan dichas tareas complejas sin ser programados manualmente para resolverlos. Estos modelos aprenden patrones a partir de datos y utilizan esa información para hacer predicciones, clasificar, o tomar decisiones.

2. 3 Redes neuronales

Las redes neuronales son un tipo específico de modelo dentro del aprendizaje automático, inspirado en la estructura y funcionamiento del cerebro humano. Debido a este intento de replicarlo, la red está compuesta por unidades básicas llamadas neuronas (codificadas con las llamadas funciones de activación, que alteran los datos de entrada imitando cómo las neuronas procesan la información). Estas neuronas están organizadas en capas: una capa de entrada, una o más capas ocultas y una capa de salida. Las neuronas de las capas se encuentran conectadas entre sí mediante conexiones equivalentes a las sinapsis entre neuronas biológicas (Conexiones con pesos asociados que determinan la "importancia" que tienen para la neurona los datos de la neurona anterior).

La existencia de las anteriormente mencionadas capas ocultas implica uno de los mayores problemas de las redes neuronales: Estas actúan cómo una caja negra, pues no podemos comprender el significado de los valores de los pesos y funciones de activación de las inconmensurables conexiones y neuronas. Además, el tipo de conectividad entre las neuronas puede dar lugar a diferentes arquitecturas cómo las redes totalmente conectadas, las redes convolucionales o las redes recurrentes. Nuestro proyecto se centrará en redes neuronales de gran profundidad con una arquitectura totalmente conectada, es decir, donde cada neurona de una capa se conecta con las neuronas de la siguiente capa.

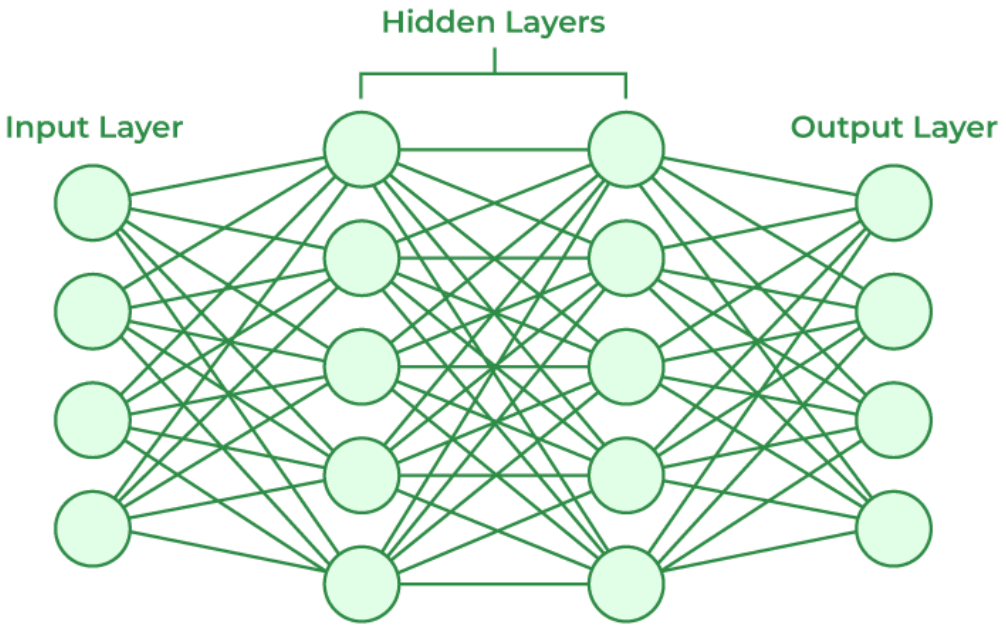

Figura 2.1: Red Neuronal

2. 4 Aprendizaje profundo

A medida que la capacidad computacional ha aumentado, se ha hecho posible entrenar redes neuronales más profundas, es decir, redes con muchas capas ocultas, lo que ha dado lugar al término "aprendizaje profundo". Estas redes pueden aprender representaciones jerárquicas de los datos, es decir, pueden extraer características simples en las primeras capas y combinarlas en características más complejas en las capas superiores. A continuación explicaremos tres ramas del aprendizaje profundo cuyo entendimiento es crucial para este proyecto:

2.5 Aprendizaje supervisado

En el contexto actual, la subrama con más popularidad en el entrenamiento de redes neuronales profundas es el aprendizaje supervisado. Con este enfoque un modelo se entrena utilizando un conjunto de datos etiquetados. Esto significa que para cada entrada en el conjunto de datos, se proporciona también la salida deseada (etiqueta).

2.6 Retropropagación

El modelo aprende a mapear entradas a salidas, repetición tras repetición, de manera que pueda predecir correctamente las salidas para datos nunca vistos. Uno de los métodos más comunes de conseguir dicho aprendizaje es la retropropagación: Con él, se calcula el error de la red en base a cuánto se diferencia la predicción de la red con el dato real (la anteriormente mencionada etiqueta.) Una conexión actualiza su peso en base a cuanto contribuyó dicho peso al error de predicción.

2.7 Aprendizaje por refuerzo profundo

De otro modo, una subrama diferente del aprendizaje profundo es el llamado aprendizaje por refuerzo profundo (Aunque el aprendizaje por refuerzo, a secas, es una subrama del aprendizaje automático, nos centraremos en su variante combinada con redes neuronales). En oposición al aprendizaje supervisado, que como hemos mencionado, entrena directamente redes neuronales en base a datos etiquetados, este enfoque no entrena únicamente las redes neuronales, sino que se centra en enseñar a individuos (llamados agentes) a través de su interacción con el entorno, recibiendo penalizaciones o bonificaciones en base a lo acertado de sus acciones, de ahí el término “por refuerzo”.

Aquí, el papel de las redes neuronales es optimizar las políticas que guían las decisiones del agente en entornos complejos, decidiendo sobre estas penalizaciones y bonificaciones. Este es otro ejemplo de reproducción de mecanismos naturales para mejorar inteligencias

artificiales: Se imita el aprendizaje de los seres vivos con “premios” (Como la liberación de hormonas relacionadas con la felicidad al realizar tareas que favorezcan la supervivencia del material genético.) y con “castigos” (Como el dolor en situaciones peligrosas, o sentimientos cómo el enfado o la vergüenza en situaciones sociales detrimentales.)

2.8 Neuroevolución

Plantearemos una tercera rama del aprendizaje profundo, aquella en la que nos centraremos, la neuroevolución. Explicaremos esta apoyados por la información dispuesta en el artículo [20]. Esta opción es radicalmente diferente al aprendizaje supervisado, en el sentido de que no dispone de información etiquetada; empero es similar al aprendizaje por refuerzo debido a que su interacción con el entorno es quien le transmite la información sobre su adecuación al medio, esta vez con una puntuación denominada *fitness*.

La neuroevolución, asimismo, trabaja no con un solo individuo, sino con una población de ellos, cada uno codificando una red neuronal diferente. En esta población pueden aparecer especies si se presenta mucha diversidad “genética”, al estilo de las especies animales, agrupando a los individuos en ellas. Todos los individuos de la población se enfrentan para ver quien se adapta mejor al entorno, imitando así un ecosistema natural de competitividad.

No se entrenará la red neuronal utilizando métodos tradicionales como el anteriormente mencionado descenso de gradiente , sino que se utilizarán algoritmos genéticos para optimizar tanto los pesos de las conexiones como la topología de red y otros muchos factores de diseño, como las funciones de activación o los hiper parámetros de red (Aquellos parámetros que codifican características de la red neuronal y no son aprendidos a partir de los datos etiquetados).

Antes de adentrarnos en qué son los algoritmos genéticos, debemos destacar tres puntos de la información previamente presentada:

- La neuroevolución permite entrenar en entornos sin datos etiquetados, únicamente gracias a un medidor de adaptabilidad al medio, el “fitness”. Esto la hace independiente de la gran cantidad de datos necesaria para las redes profundas supervisadas modernas.
- Al almacenar una población de soluciones, en lugar de una sola, la neuroevolución alcanza una alta paralelización de soluciones.
- La neuroevolución es capaz de explorar arquitecturas de red con mucha más variedad que la retropropagación, debido a que esta puede alterar aspectos como la topología, función de activación o los hiper parámetros de la red.

Estas propiedades diferencian, y potencialmente aventajan, a la neuroevolución respecto al aprendizaje supervisado y al aprendizaje por refuerzo profundo. Por tanto, fácilmente pueden convertirse en una ayuda interdisciplinar para con estas ramas, como comentamos en el apartado de “Estado del Arte”.

2.9 Algoritmos genéticos (Caso particular de NEAT)

A continuación, presentamos una serie de conceptos básicos para poder entender el algoritmo de neuroevolución (NEAT) que emplearemos en el proyecto, ordenados de forma secuencial en tanto en cuanto intervienen en el algoritmo de nuestro caso particular: (No hemos querido entrar en gran complejidad, pues en los apartados de "Diseño de la solución" y "Desarrollo de la solución" ya profundizaremos en varios aspectos teóricos avanzados). Para nuestra explicación seguimos la información mostrada en el artículo [14].

El algoritmo genético, durante su primera ejecución, comienza inicializando una población de soluciones aleatorias.

Una población es un conjunto de genomas, divididos por especies.

- Un genoma, en nuestro caso, es la codificación de una red neuronal que tomará los inputs del partido de fútbol y devolverá la acción a realizar.
- Los genomas se agrupan en una especie si comparten similitudes genéticas, y se separan en especies diferentes cuando uno resulta genéticamente diverso respecto a otro. Las especies se emplean para conservar variedad genética en la población, ya que estas tienen mecanismos que evitan que se extingan rápidamente antes de mostrar todo su "potencial".

A continuación, la población se simula en un entorno y se evalúa el fitness de cada individuo mediante la función de fitness.

- El *fitness* es un valor que mide cómo de bien la red neuronal codificada por el genoma ha funcionado en nuestro problema.
- La función que evalúa el *fitness* debe ser codificada manualmente por el programador, ya que debe medir cómo de óptima ha sido la red neuronal para con el entorno.

En consecuencia a un mayor (en ocasiones menor) valor de fitness, se seleccionan a los individuos más aptos y se les cruza.

- El cruce, o *crossover*, es un proceso en el que se combinan los cromosomas de dos redes para crear nuevas redes (descendientes de las anteriores). Esto simula la reproducción biológica y tiene como objetivo combinar las mejores características genéticas de los padres.
- Cada genoma está constituido por genes, que representan una neurona o una conexión entre dos neuronas.
  Cada gen de neurona, también llamado gen de nodo, posee un identificador único, una categoría de neurona: entrada, oculta, salida, y una función de activación.
  Cada gen de conexión almacena los identificadores de las neuronas de origen y fin de conexión, un peso y un número de innovación (Usado, de manera simplificada, para facilitar el cruce de redes de diferente topología.)

Tras el cruce, se producen mutaciones de manera aleatoria en la nueva población.

- La mutación introduce cambios aleatorios en los cromosomas de los descendientes para mantener la diversidad genética en la población y evitar que el algoritmo se quede atrapado en óptimos locales.

La población antigua es sustituida por sus descendientes.

- En NEAT es posible seleccionar redes denominadas “élites” que permanecerán en la siguiente generación, al contrario que el resto de su población, que como máximo podrán actuar de “padres” durante el cruce si es que poseen un alto *fitness*

Se repite el proceso desde la evaluación hasta el reemplazo, esta vez sin una inicialización aleatoria, durante varias generaciones, con la esperanza de que la población evolucione hacia soluciones que superen el límite de fitness o se acaben las iteraciones estipuladas

- El límite de *fitnes* establece un fitness aceptable, tras cuya consecución no es necesario seguir iterando.

3. Estado del Arte

Presentamos este apartado subdividido en dos secciones:

En la primera sección abordaremos el avance y los logros previos y recientes del campo de la neuroevolución que demostrarán su relevancia en presente y futuro para el desarrollo de redes neuronales.

En la segunda sección documentamos implementaciones del algoritmo Neat que han sido relevantes para la inspiración de este proyecto.

3.1 Avances y logros en la neuroevolución

Citando a [20], pese a la gran popularidad de la retropropagación, esta técnica posee dos mayores pegas que la neuroevolución es capaz de solventar: El bajo grado de alteración de la red, y la poca paralelización de exploración de soluciones.

La neuroevolución supone un avance significativo en el grado de modificación de la red neuronal, permitiendo aprender otros aspectos fundamentales de las redes más allá de sus pesos. Estos incluyen sus funciones de activación, hiper parámetros, incluso arquitecturas de red, o los mismos algoritmos de aprendizaje.

Al almacenar una población de soluciones, en lugar de una sola, la neuroevolución alcanza un nivel totalmente nuevo de exploración en variedad, permitiendo una alta paralelización de soluciones.

Muchas técnicas y patrones de diseño de aprendizaje profundo fueron descubiertas décadas antes de la reciente explosión en popularidad del campo. Sin embargo, estas no parecían mostrar resultados efectivos en redes neuronales de poca profundidad.

No ha sido sino hasta el reciente avance en capacidad de cómputo y tamaño de datasets, que trabajar con redes neuronales de gran profundidad ha sido posible. Con estas nuevas redes, las técnicas ya conocidas han podido explotarse con mucha mayor eficacia.

Del mismo modo, varias técnicas de neuroevolución han mostrado funcionar mucho mejor al aprovecharse de la mejora en cómputo.

Se ha demostrado que la neuroevolución es una alternativa viable al descenso por gradiente para entrenar redes neuronales en el aprendizaje por refuerzo.

Además se ha demostrado su eficacia para optimizar redes neuronales, especialmente en casos donde el descenso de gradiente no es eficaz.

Comparación con otras técnicas de aprendizaje profundo

En el artículo "Neuroevolution gives rise to more focused information transfer compared to backpropagation in recurrent neural networks" [1], de reciente publicación, se demuestra cómo el entrenamiento de una red neuronal con neuroevolución en oposición a la retropropagación confiere una mayor resistencia a perturbaciones en los pesos, a la par que la información es transmitida de forma mucho más concentrada en un menor número de neuronas.

La robustez frente a los cambios en los pesos confiere a la red una resistencia a "ruido" en los datos, o soporte de técnicas de regularización (técnica empleada para que la red responda mejor a datos nuevos y no se especialice en el set de datos etiquetados) sin bajar su rendimiento significativamente. Asimismo, se aumenta la especialización de la red, con neuronas encargadas de tareas en específico similares a módulos, y por consecuencia se aumentaría el rendimiento computacional y disminuiría el costo temporal al requerir menos operaciones de propagación de información. En igual forma se reduciría la dimensionalidad del espacio de representación de la red, facilitando el aprendizaje y su convergencia (Definimos convergencia como el alcance del objetivo de predicción de la red.)

Por otro lado, la concentración de la información en ciertas neuronas ofrece una mayor interpretabilidad de la red, solventando en cierta manera el problema de "la caja negra" (Que como mencionamos en el apartado "Fundamentos teóricos", supone la poca interpretabilidad de las capas ocultas/intermedias de una red neuronal).

En el campo del aprendizaje por refuerzo profundo, un estudio [17] reveló que una forma de algoritmo evolutivo llamada estrategia evolutiva natural (NES) pudo competir eficazmente con los mejores algoritmos de aprendizaje por refuerzo profundo, como DQN y A3C, en tareas con redes neuronales profundas grandes. La sorpresa radica en que un algoritmo evolutivo logró esto en espacios de parámetros de alta dimensión. No obstante, como el NES se puede interpretar como un método basado en gradientes , algunos investigadores no consideraron que un algoritmo evolutivo completamente libre de gradientes pudiera funcionar a esta escala.

Y así fue hasta la llegada del artículo "Deep Neuroevolution: Genetic Algorithms are a Competitive Alternative for Training Deep Neural Networks for Reinforcement Learning" [19], donde se mencionan dos tipos de neuroevolución capaces de rivalizar con la retropropagación en el campo del aprendizaje por refuerzo profundo:

- Las estrategias de evolución (evolutionary strategies), empleando descenso por gradiente estocástico.
- Los algoritmos genéticos, libres de gradiente, como los empleados en nuestro proyecto.

Ambos casos son similarmente eficaces para entrenar redes neuronales profundas en aprendizaje profundo con más de 4 millones de parámetros libres, y además, superan a otros métodos de aprendizaje por refuerzo cómo el Q-Learning. Este estudio demuestra que las técnicas de neuroevolución libre de gradiente suponen una alternativa viable en tareas de optimización.

Asimismo, en muchos casos se demostró que los algoritmos de neuroevolución superan en velocidad de ejecución a los mejores algoritmos de aprendizaje por refuerzo profundo, DQN y A3C, debido a su alta capacidad de paralelización. Un ejemplo de esos casos es el mostrado en el artículo [13] "*Evolution Strategies as a Scalable Alternative to Reinforcement Learning*." (Realizado por miembros de la famosa empresa de investigación de IA, Open Ai)

Un gran caso de ejemplo de cómo las técnicas de neuroevolución pueden hibridarse con técnicas clásicas de aprendizaje profundo y además de aprendizaje por refuerzo se encuentra en [8], (Demostrando además que el interés en combinar métodos de gradiente y neuroevolución está creciendo.) En su trabajo, introdujeron mutaciones seguras a través de gradientes de salida.

Este enfoque se basa en la idea de que evaluar una política de aprendizaje por refuerzo suele ser costoso computacionalmente, ya que necesita realizar toda una simulación, pero evaluar las salidas de una red neuronal lo es mucho menos. Empleando neuroevolución, se realizan mutaciones aleatorias en la política. Algunas de estas pueden no tener efecto en el comportamiento de la red, mientras que otras podrían tener consecuencias importantes. La idea detrás de las mutaciones seguras es que podemos mantener una biblioteca de referencia y usar la información de gradiente para ajustar la magnitud de las mutaciones por peso, de manera que los cambios en la política en el conjunto de referencia no sean ni demasiado grandes ni demasiado pequeños.

Aplicaciones de la Neuroevolución

Concentrándonos en campos específicos, podemos destacar el framework NEP o *Neuroevolution Potential*, mostrado en el artículo "Neuroevolution machine learning potentials: Combining high accuracy and low cost in atomistic simulations and application to heat transport" [18] donde se entrena una red neuronal mediante neuroevolución para simular dinámicas moleculares de larga escala. En el consecuente artículo "Improving the accuracy of the neuroevolution machine learning potential for multi-component systems" [7] se consigue una mejora radical en la precisión de dicho framework sin aumentar el costo computacional.

En el campo del testeo del software, específicamente videojuegos, se combina el *Search-based software testing* (Una técnica de generación de casos de prueba automáticos) con la neuroevolución para alcanzar y validar de manera sistemática diferentes estados del programa (por ejemplo, todos los posibles caminos de un agente) incluso aleatorizado el mismo. Esto se muestra en el artículo [12] .

En el ya mencionado artículo [13] se demuestra que la neuroevolución, y especial el algoritmo NES que introducimos con anterioridad, resulta eficaz para enseñar a que un robot humanoide controlado por una red neuronal de dos capas camine, y lo hace más rápidamente en tiempo real que los algoritmos de aprendizaje por refuerzo competidores, gracias a que la evolución facilita una mejor paralelización. Estos resultados no son extraños, dado que desde sus inicios, la neuroevolución ha demostrado una gran serie de sucesos en el campo de la robótica evolutiva.

3.2 Implementaciones en Neat

Grandes logros de Neat

Entre los logros mas destacados del algoritmo Neat, se encuentra su uso en el acelerador de partículas Tevatron para descubrir de la manera más precisa la masa del quark [1], o sus diversos usos en robótica evolutiva como el demostrado en [6], destacado por enseñar al robot Sony Aibo a dar sus primeros pasos.

Más recientemente, se usó una ampliación de Neat denominada HyperNeat, destacada por la codificación indirecta (Al contrario que una codificación directa de la red neuronal, donde el genoma de cada individuo describe paso a paso una red neuronal, una codificación indirecta actúa cómo una fórmula matemática para su creación.), para optimizar LTSM's (Un tipo de red neuronal recurrente, es decir, con conexiones cíclicas entre sus neuronas, especializada en tratar con datos secuenciales) en tareas de modelado de lenguajes. Esto se muestra en el artículo [4] (Ha et al., n.d., #)

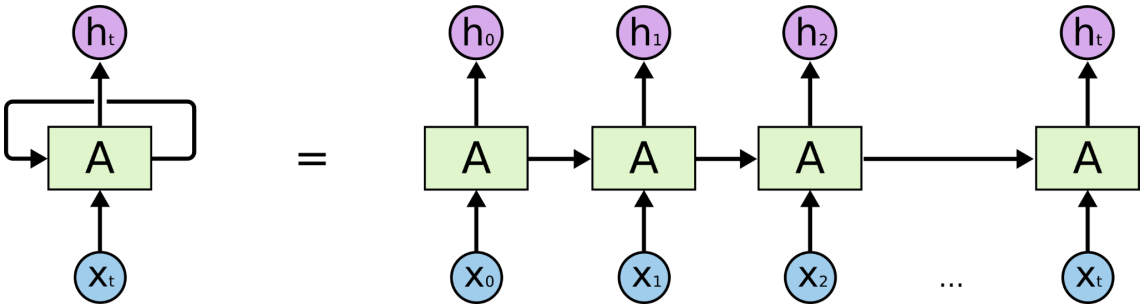

Figura 3.1: Estructura simplificada de una LTSM

Implementaciones ejemplares para nuestro proyecto

En el repositorio de Neat-Python [9], del cual tomamos el código base de Neat para realizar este proyecto, encontramos diversas implementaciones del algoritmo relativamente similares en complejidad a nuestro problema:

El caso de ejemplo del "aterrizador lunar" es el más cercano al nuestro: Intenta hacer aterrizar un cohete en una superficie irregular sin chocar contra el suelo y permaneciendo cerca de la zona de aterrizaje.

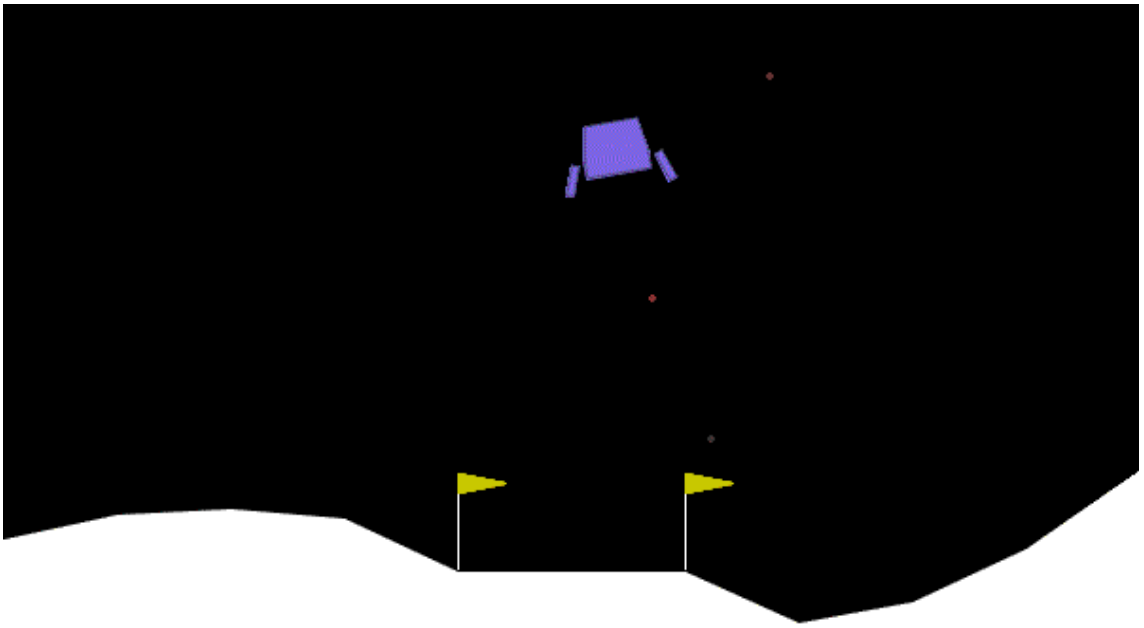

Figura 3.2: Ejemplo “Lunar Lander” de la biblioteca de Neat-Python

Los inputs de la red son 8 dimensiones: las coordenadas del cohete en x e y, sus velocidades lineales en x e y, su ángulo, su velocidad angular, y dos valores booleanos que representan si cada pata está en contacto con el suelo o no. Nuestro proyecto es ligeramente más complejo, ya que recibe 13 inputs en una de sus etapas de mayor complejidad.

Sus outputs son 4: Impulsarse a izquierda, derecha, hacia arriba o no hacer nada. Nuestro proyecto, durante todo el entrenamiento, empleará 5, añadiendo el impulso hacia abajo.

Sobre su evaluación de fitness, destacamos que añade una penalización por cada ocasión en la que se usa el “combustible”: De esta forma se pretende fomentar el ahorro del mismo. Nuestro proyecto, en ciertas etapas del desarrollo, ha añadido penalizaciones al marcar gol en propia puerta o no acercarse suficientemente a la pelota en ningún momento.

Destacamos también el tamaño de su población, de 150. Nuestro proyecto empezó usando un tamaño de población de 10-20, y no fue hasta que nos fijamos en este ejemplo que fuimos conscientes de la importancia de una población grande para un problema de esta complejidad (Sin una población grande, no hay oportunidad de introducir variación genética suficiente como para explorar suficientemente el espacio de soluciones.) Aumentar el tamaño de nuestra población ha resultado un factor clave en el éxito del proyecto.

El caso del balanceo de una vara (Single pole balancing) se usa como ejemplo en el artículo del propio algoritmo Neat [14] , mencionando que este entorno ha sido como un *benchmark* o punto de referencia para la neuroevolución y el aprendizaje por refuerzo en los últimos 50 años. Ser prolífico en este test demuestra que un método podrá funcionar en otros entornos. De hecho, la efectividad de Neat se demostró empleando este mismo ejemplo.

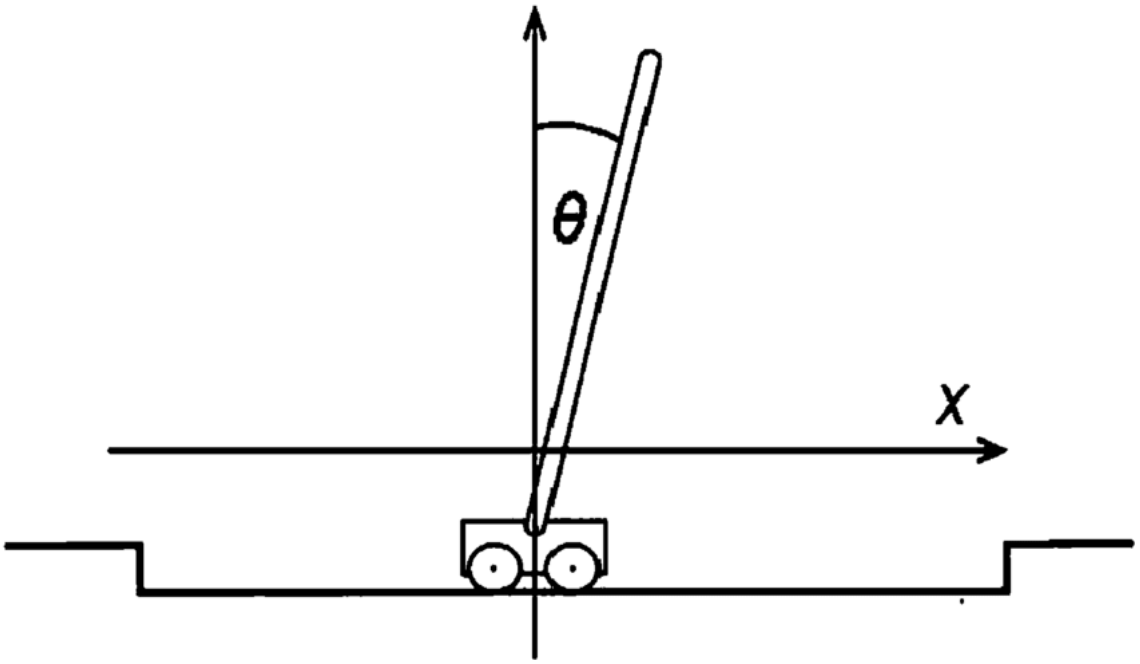

Figura 3.3: Ejemplo "Pole Balancing" de la biblioteca de Neat-Python

El problema trata de mover un pequeño carro para balancear una barra sin que esta caiga.

Sus inputs son únicamente 4, bastantes menos que nuestro caso: La posición en x, la variación de posición, el ángulo de la vara y la variación del ángulo.

Su output es únicamente 1, mover hacia derecha o izquierda (O en valor 0, quedarse quieto)

Este ejemplo destaca especialmente por su población, de 250 miembros. Como mencionamos en la implementación del aterrizador lunar, aumentar el número de individuos fue clave para que nuestro algoritmo tuviera variedad genética suficiente.

Otros ejemplos de este repositorio a mencionar son la simulación de circuitos eléctricos, la lógica xor o la secuencia de outputs binarios.

Queremos mencionar los tutoriales de los siguientes canales de YouTube: Tech With Tim [15] , Cheese Ai [2] , y Neat AI [10], dado que fueron nuestra inspiración para empezar este proyecto.

De entre ellos destacamos sus implementaciones de Neat para el videojuego "Flappy Bird", coches en un circuito, "Snake", y "Pong", que fueron replicadas para comprender las tecnologías usadas.

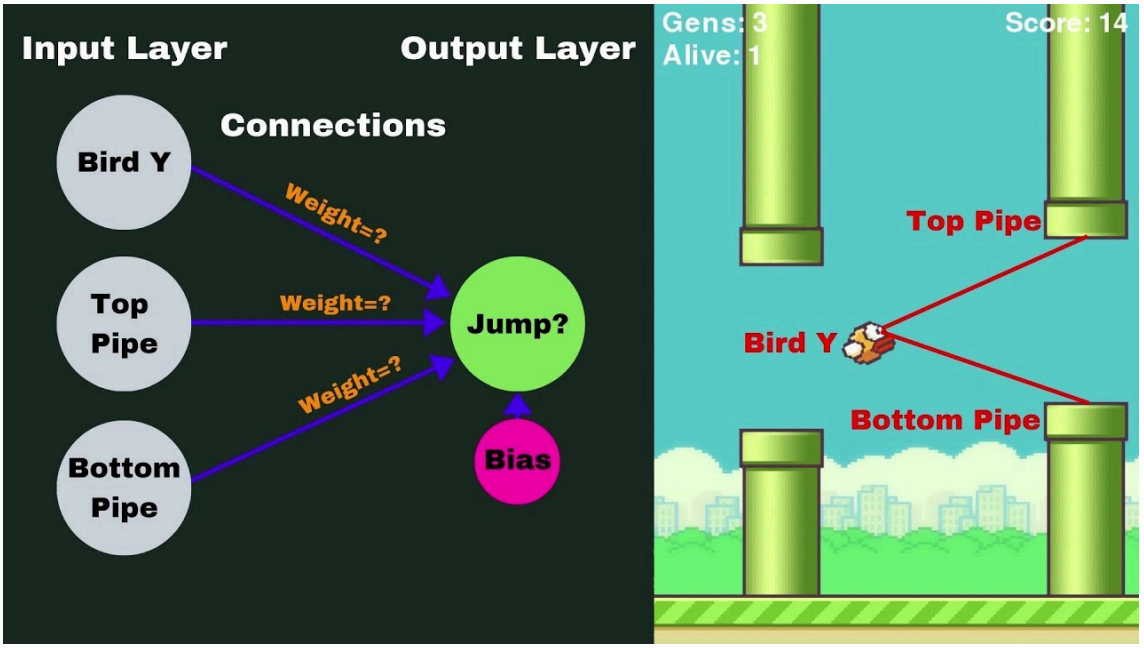

Figura 3.4: Tutorial de "NEAT Flappy Bird" del canal Tech With Tim

Estas implementaciones son más sencillas que la nuestra, debido sobre todo a los espacios de soluciones mucho menores de sus entornos y el número de inputs que sus redes obtienen. Sin embargo, fueron una excelente base tanto para comprender Python como Pygame.

3.3 Propuesta

Con este proyecto experimentaremos el funcionamiento de Neat en un problema altamente relacionado con la inteligencia espacial y especialmente con el enfrentamiento entre dos individuos. Debido a que los ejemplos propuestos de simulaciones 2D en la biblioteca de Neat-Python o los mayormente mencionados en la literatura no exploran estos dos conceptos en profundidad. La eficiencia de nuestras redes neuronales en el uso de trigonometría y cálculo de fuerza para realizar goles con ángulo y rebote, además de la consideración de la posición y velocidad del rival, pueden aportar nuevas perspectivas de la potencia del algoritmo.

Asimismo, las técnicas de entrenamiento y modificaciones en el algoritmo que desarrollaremos en el trabajo no aparecen mencionadas en la literatura ni usando la búsqueda por internet, por lo que creemos que aportarán nuevas ideas en el campo.

4. Análisis del problema

4.1 Identificación de oportunidades

Hibridación de la neuroevolución con otros métodos del aprendizaje profundo

Tal y como se menciona en los apartados de "Estado del Arte" y "Motivaciones", con el auge de las redes neuronales profundas, han resurgido métodos de aprendizaje automático que fueron propuestos hace tiempo, pero cuyo potencial no pudo aprovecharse debido a la limitada complejidad de las redes neuronales de aquella época.

Entre estos métodos, ha cobrado relevancia recientemente el campo de la Neuroevolución, que busca replicar computacionalmente los procesos de reproducción genética y selección natural para aplicarlos al diseño de redes neuronales.

Actualmente, hay un creciente interés en explorar estos algoritmos neuroevolutivos para desarrollar redes neuronales profundas, especialmente en el ámbito del aprendizaje por refuerzo.

El desarrollo de este proyecto puede suponer, por tanto, una oportunidad de obtener técnicas aplicables al aprendizaje profundo. Para observar ejemplos de hibridación exitosos y de gran novedad, se puede consultar el apartado de "Estado del arte"

Innovación en Neat

Las más destacadas mejoras en el algoritmo de Neat son:

- El uso de patrones de conexión neuronales complejos y regulares gracias a la codificación indirecta, como los presentes en HyperNeat y ES-HyperNeat.
- El aprovechamiento de la estructura espacial de la entrada, de nuevo, gracias a la codificación indirecta en HyperNeat y ES-HyperNeat
- La selección automática de características en FS-NEAT
- La evolución de la red neuronal en tiempo real empleando rtNEAT
- La combinación de subredes especializadas presente en CoDeepNEAT

De entre las anteriormente mostradas, no se mencionan algunos aspectos en los que nuestro proyecto puede aportar, bien alterando el algoritmo o presentando tácticas de entrenamiento:

- Simplificación del entrenamiento sin exponer a la red a una complejidad inicial demasiado elevada, con una progresiva cantidad de inputs que aumenta conforme lo hace la complejidad del entorno.
- Exposición de los individuos a diversas situaciones no alcanzables en un entrenamiento rápido.
- Crear redes de diferentes funcionalidades bajo un mismo proceso de entrenamiento.

- Tácticas de entrenamiento contra inteligencias adversas, humanas o artificiales.
- Aceleración del proceso de entrenamiento o mayor paralelización del mismo.

4.2 Identificación de soluciones

Para cumplir con los objetivos de nuestro proyecto dos elementos indispensables son: Un entorno de simulación y un script que gestione la neuroevolución.

Las implementaciones mostradas en el apartado de "Estado del arte" muestran que este patrón se repite en todos los casos.

A la hora de identificar diferentes soluciones, el abanico se centra en las tecnologías empleadas para la solución y no en la estructura de la solución. Por ello, en la sección de "Diseño de la solución", en el apartado de "Análisis de herramientas" encontraremos un amplio análisis de las mismas.

4.3 Solución propuesta

Nuestro proyecto consistirá en la simulación de un partido de fútbol-sala en un entorno similar a un videojuego en el que entrenar redes neuronales mediante neuroevolución hasta conseguir un jugador capaz de rivalizar con un humano en inteligencia especial, y además descubrir y aplicar nuevas técnicas que mejoren el algoritmo de neuroevolución o el entrenamiento.

El entorno/videojuego será realizado en Pygame, representando a dos (en ocasiones más) jugadores en una cancha de fútbol sala. Cada jugador será controlado por una red neuronal o un jugador humano. Se incluirán físicas de rebotes y colisiones, lógica de partido y se organizará el código de manera que el acceso desde el algoritmo de entrenamiento sea sencillo.

En ese entorno se empleará la biblioteca Neat, junto al algoritmo de entrenamiento y las modificaciones añadidas por el alumno, para entrenar una población de redes neuronales en búsqueda de un individuo igual de inteligente espacialmente que un ser humano, realizando un entrenamiento progresivamente complejo en el que se apliquen diferentes técnicas. Para ello el algoritmo deberá generar instancias de la simulación correctamente y transmitir la información desde las redes neuronales a los jugadores, almacenar las poblaciones y mejores individuos progresivamente, permitir una fácil configuración, ofrecer varias técnicas de entrenamiento (como uno solitario y otro en enfrentamiento) y establecer funciones de fitness adecuadas.

Finalmente se mejorarán los procesos de entrenamiento y el propio algoritmo Neat con las ideas del alumno para aumentar el desempeño o añadir nuevas funcionalidades.

4.4 Especificación de requisitos

Requisitos funcionales

Requisitos funcionales del entorno de juego:

RF1

Mover jugador

El humano o la I.A. podrá mover al jugador de fútbol a lo largo de la cancha sin salir de ella. Los jugadores se moverán con una cierta aceleración y decelerarán con el tiempo.

RF2

Chutar pelota

El jugador de fútbol trasnmitirá a la pelota su velocidad y dirección o actuará como un muro en el que la pelota rebotará

RF3

Registrar chuts

Se registrarán los toques que los jugadores realicen en la pelota.

RF4

Driblar pelota

Si dos jugadores empujan la pelota uno contra otro, uno de ellos debe vencer

RF5

Rebotar pelota

La pelota rebotará contra los muros de la cancha y los jugadores

RF6

Marcar gol

Si la pelota entra en contacto con la portería, se registrará el gol y se reiniciarán las posiciones de los elementos del escenario.

Requisitos funcionales de la inteligencia artificial:

RF7

Percibir posición

La IA debe entender la relación de los inputs de coordenadas del jugador con su posición en el mapa

RF8

Aproximarse a pelota

La IA debe ser capaz de mover a su jugador en dirección a la pelota

RF9

Comprender velocidad

La IA debe percibir su velocidad , relacionarla con su posición y entender como compensarla a la hora de moverse

RF10

Chutar con ángulo

La IA debe relacionar su ángulo y velocidad con el efecto que provoca en la pelota para poder tirar en la dirección deseada

RF11

Marcar gol

La IA debe entender como rebota la pelota y marcar gol incluso en situaciones donde la pelota no pueda entrar en la portería directamente

RF12

Marcar con rebote

La IA debe entender como rebota la pelota y marcar gol incluso en situaciones donde la pelota no pueda entrar en la portería directamente

RF13

Identificar equipo

La IA debe identificar el equipo al que pertenece y marcar únicamente en la portería contraria

RF14

Percibir rival

La IA deberá identificar la posición y velocidad del jugador rival

RF15

Sortear rival

La IA deberá realizar chutes evitando el bloqueo del rival

RF16

Defender portería

La IA deberá realizar chutes evitando que se marque en su portería

Requisitos no funcionales

Los requisitos que se presentan a continuación se basan en el modelo ISO/IEC 25010:

RNF1

Corrección funcional

El entorno de simulación no debe poseer errores o *bugs* que interfieran con el entrenamiento

RNF2

Utilización de recursos

El entorno de simulación no debe ser computacionalmente exigente para permitir un entrenamiento rápido

RNF3

Capacidad de ser modificado

Los scripts deberán permitir una fácil modificación para probar diferentes técnicas de entrenamiento

4.5 Plan de trabajo

Como se menciona en el apartado de "Metodología", hemos empleado una metodología ágil en nuestro proyecto, basada en sprints.

Debemos destacar que la organización inicial de nuestro proyecto asumió 4 sprints:

- Sprint 0: Documentación y Organización
- Sprint 1: Desarrollo de entorno Pygame
- Sprint 2: Desarrollo de script de entrenamiento Neat
- Sprint 3: Entrenamiento
- Sprint 4: Mejoras originales del alumno

Sin embargo, tal y como explicamos dentro de cada sprint, diversos problemas necesitaron de aplicar la flexibilidad de la metodología ágil, concentrando los sprints en la resolución del conflicto y añadiendo un nuevo sprint para paliar el retraso en las tareas del anterior.

Estos son los sprints reales llevados a cabo durante el desarrollo de nuestro proyecto:

Sprint 0

La finalidad de este sprint era la de documentación y organización al proyecto:

- Como actividad inicial el alumno realizó la documentación sobre el campo de la neuroevolución que puede encontrarse en el apartado de "Estado del arte" y "Fundamentos teóricos" con el fin de guiar el proyecto.
- A continuación se procedió con la documentación de las diversas herramientas disponibles para realizar el proyecto, presentadas en el apartado de "Diseño de la solución", y la elección de una de ellas.
- La siguiente actividad fue una "lluvia de ideas" sobre el tipo de entorno/videojuego deseado para entrenar neuroevolutivamente, estableciendo objetivos de forma muy general.
- Para finalizar, una vez tomadas las decisiones más importantes, se organizó el conjunto de diferentes tareas previstas y se las repartió en cada Sprint.

Sprint 1

Este sprint inicia el trabajo directo en el proyecto, con la finalidad de desarrollar el entorno del videojuego de fútbol-sala en Pygame:

- Inicialmente desarrollamos una serie de entornos de prueba en Pygame siguiendo los tutoriales mencionados en la sección de "Estado del arte", tanto con el fin de aprender a usar la herramienta como para aproximar la dificultad del proyecto.
- Con los fundamentos de la herramienta claros, se diseñó la representación visual de los elementos en pantalla, los inputs y el movimiento.
- Se desarrolló el sistema de colisiones y el sistema de físicas. Fue durante este desarrollo en el que nos encontramos nuestro gran problema, al no prever que la diferencia entre las físicas de nuestro proyecto y los tutoriales fueran a ser tan difíciles de solventar.
- En vista a lo anterior, y siguiendo la metodología ágil, se decidió centrar los esfuerzos de este sprint en obtener unas físicas sin errores. Originalmente se disponía de las tareas de construcción de lógica de partido y reestructuración y formateo de código dentro de este sprint, pero se decidieron trasladar al siguiente.

Sprint 2

Este sprint nació durante el desarrollo del proyecto, y no en su planificación, como mencionamos en el apartado anterior. Resolvió las dificultades de desarrollo del entorno:

- Se construyó la lógica de partido y los mecanismos de obtener la información de partido.
- A continuación se reestructuró y reformateó el código en diversas clases alrededor de un script principal.
- Adicionalmente se facilitó la disposición de la información de las clases para que fuera mas accesible a la sección de Neat
- Para finalizar se testearon los errores y se solucionaron, hasta obtener un entorno sin fallo alguno apreciable, y por lo tanto suficientemente robusto como para aguantar miles de simulaciones en el entrenamiento de la IA.

Sprint 3

A lo largo de este sprint se desarrolló el script de neuroevolución:

- Como actividad inicial se aplicó Neat en los entornos de prueba de Pygame mencionados en el Sprint 1.
- Posteriormente se crearon los mecanismos de importar configuración, salvado de checkpoint de población, de monitorización y de guardado de mejor individuo.
- A continuación se desarrolló el algoritmo para tomar el genoma de cada elemento de una población, asociar dicho genoma a una nueva red y vincularla con los inputs y outputs de cada jugador. Al mismo tiempo se escribió el código asociado a crear una nueva partida de fútbol por cada genoma entrenado y tomar los datos de ella.

- Esta etapa supuso también la reescritura de ciertas partes del código asociados al entorno de Pygame que provocaban errores al combinarse con Neat. Estos errores supusieron un problema considerable, obligando a realizar decenas de pruebas y retrasando el desarrollo hasta descubrir el origen de los errores.

Sprint 4

Con este sprint inician los múltiples ciclos de simulación, observación y corrección que fueron necesarios para el correcto entrenamiento de la IA. Teniendo en cuenta la cantidad de ciclos necesarios para solventar todos los problemas nacidos durante este proceso, fue necesario, de nuevo, crear varios sprints no previstos durante la organización del proyecto:

- En este primer sprint se sucedieron numerosas sesiones de entrenamiento en las que se observó el nulo aprendizaje del algoritmo.
- En un intento de acelerar el proceso de entrenamiento, se decidió simplificar los objetivos y intentar que el jugador fuera capaz de entrar en contacto con la pelota, no exigiendo que jugare completamente al partido.
- Vista la poca eficacia de este intento simplificado, se reescribió tanto el código de Neat como el de Pygame para que en cada instancia de entrenamiento se emplearan decenas de individuos en lugar de uno solo. Así, idealmente, se paralelizaría el entrenamiento y se consiguiría el objetivo.
- Ante el no funcionamiento de la solución previa, se decidió dedicar el resto del sprint a leer documentación en busca de una solución y reorganizar el resto del trabajo.

Sprint 5

En este sprint finalmente se dió con la solución al problema mencionado en el script anterior. Para ello, la tarea de creación de técnicas novedosas tuvo que adelantarse a este sprint, pese a que ese tipo de tarea estaba prevista una vez la IA funcionara suficientemente bien:

- Para explicar la técnica desarrollada, debemos mencionar que en este sprint se consiguió entrenar una red neuronal capaz de moverse hacia la pelota mediante la simplificación del input a dos elementos. Sin embargo, dado que esta solución no le permitiría jugar el partido, se realizó la tarea siguiente.
- La técnica novedosa que tuvo que desarrollarse en este script, y que satisfactoriamente resolvió el problema de la ausencia de aprendizaje con un gran número de inputs fue permitir un aumento progresivo de los inputs asociados a la red insertando manualmente genes. Desarrollarla fue increíblemente costoso, dado que no existía documentación alguna en internet sobre ningúna implementación de este tipo de, así que tuvo que estudiarse a fondo la documentación de Neat para poder desarrollar el script de forma totalmente original.

Sprint 6

Este sprint supone la evolución en complejidad de los objetivos de la red, a la par que se aumentaba el número de inputs en ella para asegurarse de un aprendizaje progresivo:

- Debemos mencionar que durante este proceso tuvieron que solucionarse los varios errores asociados a nuestra implementación de los inputs progresivos.
- A lo largo del sprint se implementaron también nuevas técnicas de enfrentamiento en parejas, con una red enfrentándose a otra en tareas progresivamente difíciles. Aunque esta tarea también se había previsto para un sprint posterior a conseguir una red capaz de marcar gol por si misma, se demostró que la competición aceleraba el proceso de entrenamiento incluso en tareas simplificadas.
- Del mismo modo a lo largo del sprint se tuvieron que desarrollar los técnicas de relocalización de la red en ambos equipos y con la pelota en posiciones diversas para ayudarle a explorar mayor variedad del espacio. El uso de estas tecnicas ralentizó notablemente el entrenamiento pero permitió una mucha mejor comprensión de las bases combinado con los inputs progresivos.
- Una última tarea realizada a lo largo de este sprint fueron las adaptaciones y pruebas de diferentes tipos de fitness para cada complejidad de entrenamiento. El uso de estas técnicas también supuso una mejora radical en el entrenamiento.
- Finalizando el sprint pudo conseguirse una red capaz de marcar gol por si misma, incluso empleando rebotes al enfrentarse a diferentes situaciones de la pelota.

Sprint 7

Finalmente, este sprint supuso la consecución de una red capaz de enfrentarse a una inteligencia humana y vencer.

- Para ello se ampliaron las técnicas ya desarrolladas en el anterior sprint, relacionadas con la exploración del espacio, la asignación de fitness y la competitividad con otras redes.
- Conforme se producía el aumento de complejidad de inputs (Y consecuentemente de la densidad de las redes) una nueva tarea no prevista fue añadida, que comprendía la exploración de las implementaciones de la biblioteca de Neat [9] en busca de una forma de aumentar la variedad de aprendizaje de nuestra red.
- Fue gracias a esa investigación que se realizó la tarea de exploración de diferentes tamaños de población y funciones de activación en la red, que finalmente tuvo un buen resultado, combinada con las anteriores técnicas, para obtener la variedad genética suficiente como encontrar una solución cercana a la óptima.

Sprint 8

Durante este sprint final se desarrollaron nuevas técnicas de mejora del proceso de entrenamiento y del algoritmo Neat. Algunas de estas mejoras, programadas para este sprint

final, tuvieron que ser desarrolladas en sprints anteriores para hacer avanzar el entrenamiento de la red base:

- Se implementó la nueva función de redes especialistas para imitar las diferentes posiciones de un jugador de fútbol. Esta nueva función, al igual que la función de inputs progresivos, requirió una gran documentación en el algoritmo neat y consumió un largo tiempo del script.
- Posteriormente se realizó una fase de comparación de la capacidad de la red final con un grupo de humanos.
- Para concluir con el trabajo se documentaron las partes restantes de la memoria del proyecto.

5. Diseño de la solución

5.1 Análisis de herramientas

Lenguajes de programación

Python

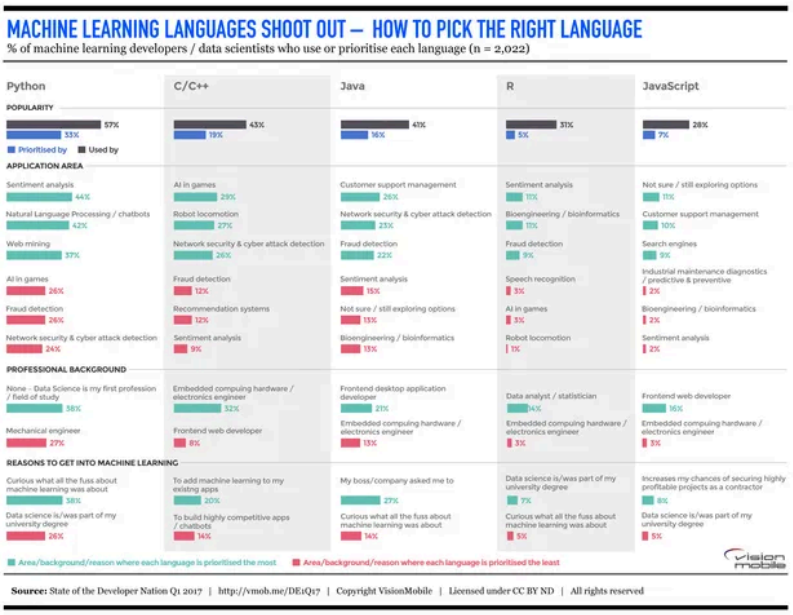

Figura 5.1: Lenguajes de programación usados en aprendizaje automático

En primer lugar, la experiencia del alumno en aprendizaje automático, tanto durante la carrera universitaria como en sus prácticas de empresa, ha sido con el lenguaje Python. Este lenguaje destaca por la calidad de sus librerías de aprendizaje profundo como PyTorch y Tensorflow, con las que el alumno ya posee experiencia.

En segundo lugar, consultando la página web [16] confirmamos que Python es el lenguaje más popular y prioritario en ciencia de datos y aprendizaje automático, con un 57% de uso y un 33% de priorización. Su dominio se debe a la evolución de marcos de aprendizaje profundo como TensorFlow y una amplia gama de bibliotecas. Python tiene la mejor relación entre prioridad y uso (58%). C/C++ es el segundo en uso (44%) y priorización (19%), seguido de cerca por Java.

Por lo tanto, pensamos que Python es la opción más útil de cara a la futura investigación o desarrollo del alumno en esta área. Además siente un gran interés en áreas donde el lenguaje

destaca, cómo el procesamiento natural de lenguajes y el análisis de sentimientos. Ambos motivos han sido el factor decisivo para elegir el resto de bibliotecas.

C/C++

Continuando con la anterior cita, C/C++suele ser empleado en implementaciones de la inteligencia artificial sobre videojuegos y robótica, debido a su ventaja en control, eficiencia y desempeño.

Debido a que estos son campos de interés para el alumno, futuros proyectos podrían desarrollarse empleando este lenguaje, pero parece más apropiado mantenerse con Python.

Algoritmos de neuroevolución

Neat

Neat (NeuroEvolution of Augmenting Topologies) es nuestra alternativa elegida como algoritmo de neuroevolución. Específicamente elegimos su biblioteca en el lenguaje Python para funcionar en combinación con el entorno de Pygame, y además, permitirnos profundizar en el aprendizaje de Python.

Entre los algoritmos de Neuroevolución, NEAT es uno de los grandes destacados por las diversas cualidades que ahora mencionaremos, destacadas en el artículo [14].

Entre los puntos positivos que nos hicieron decantarnos por Neat, y que además lo hacen sobresalir como uno de los más usados algoritmos de Neuroevolución, encontramos:

- En Neat, las redes neuronales evolucionan no sólo en términos de pesos, sino también en términos de estructura. Esto permite que la red neuronal desarrolle y adapte su topología durante el proceso evolutivo, lo cual puede ser beneficioso para problemas complejos (como el nuestro). Se inicia el entrenamiento con redes neuronales simples y va añadiendo complejidad (nuevas neuronas y conexiones) a medida que la evolución avanza.
- Emula el proceso de selección natural de forma más precisa que otros algoritmos genéticos, lo que puede llevar a descubrimientos de soluciones innovadoras y eficientes que no serían fáciles de encontrar mediante técnicas tradicionales de entrenamiento de redes neuronales.
- Ha sido aplicado a problemas similares al nuestro, como mencionamos más extensamente en el apartado de “Implementaciones en Neat”.
- Utiliza mecanismos para mantener la diversidad en la población de redes neuronales, lo que ayuda a evitar el estancamiento en soluciones subóptimas y fomenta la exploración de soluciones. Esto nos ha resultado extremadamente útil para nuestro problema dada su complejidad y la cantidad de comportamientos que se deben aprender antes de ser capaz de marcar un gol.

- Neat-Python es de código abierto, lo que nos facilitado enormemente a comprender el funcionamiento del algoritmo en detalle, y en la personalización y extensión de la biblioteca según nuestras necesidades
- Cuenta con una gran cantidad de documentación y tutoriales, así como combinaciones previas con PyGame.
- La representación directa de redes neuronales en comparación con Hyper-Neat y la gran variedad de documentación facilitarán el desarrollo de mejoras y nuevas técnicas sobre las que intentar innovar o mejorar el proceso de aprendizaje de nuestro problema.

Sin embargo, estos son algunos de los puntos negativos del algoritmo, de los cuales hemos sido testigos de primera mano en el desarrollo del proyecto y nos han obligado a desarrollar soluciones para paliarlos:

- La evolución de redes neuronales es computacionalmente costosa empleando Neat. La necesidad de evaluar muchas redes en cada generación y de mantener un historial de topologías puede llevar a tiempos de computación significativos, especialmente para problemas complejos cómo el nuestro. Durante los entrenamientos, especialmente en aquellos que nos han requerido enfrentar a cada red contra cada otra de la población, y con un tiempo suficiente para marcar varios goles, hemos experimentado tiempos de alrededor de 5 minutos por miembro de la población, con poblaciones de 100-250 miembros.
- Aunque esta tecnología es eficaz para problemas pequeños y medianos, puede enfrentar dificultades al escalar a problemas más grandes o más complejos debido al aumento en la complejidad del genoma y la necesidad de evaluar muchas redes. Este ha sido nuestro caso, ya que cómo comentamos en el apartado anterior, debíamos evaluar redes con gran profundidad, en poblaciones de al menos 100 miembros, una red contra otra.
- El algoritmo posee varios parámetros que deben ajustarse, como las tasas de mutación, los parámetros de cruzamiento y el tamaño de la población. Aparte de la complejidad para comprender estos y su papel en la evolución, encontrar una configuración óptima nos ha requerido experimentar y ajustar, y sobre todo, basarnos en ejemplos similares.
- Como ha sucedido en nuestro caso, NEAT puede converger lentamente a una solución óptima, especialmente si la función de fitness es difícil de evaluar o si el problema tiene una alta dimensionalidad. Las nuevas técnicas de entrenamiento que planteamos más adelante han sido necesarias para que el entrenamiento diera frutos, encontrándonos con periodos de varios días de simulación donde no se realizaban avances.

Hyper-Neat

Hyper-Neat, un algoritmo que amplía sobre los fundamentos de Neat, fué una de las posibles alternativas exploradas. Su principal ventaja respecto a Neat es la representación indirecta de la red neuronal mediante una función de producción de patrones composicionales (CPNN).

La codificación indirecta que la caracteriza aporta ventaja para aprovechar la estructura espacial de la entrada, es decir, que es útil en problema de visión artificial o control robótico, además de para representar grandes redes neuronales.

Con esto en cuenta, explicaremos los motivos por los que descartamos este algoritmo:

La profundidad estimada de la red neuronal que resolviera el problema no parecía requerir de un encoding indirecto para reducir su tamaño. Esto se dedujo a vista de los otros ejemplos ligeramente más sencillos, pero similares, como los expuestos en la sección "Implementaciones de Neat", realizados con Neat básico.

La mayoría de documentación y tutoriales disponibles en internet se centran en Neat. Debido a que este proyecto busca explorar los algoritmos neuroevolutivos como una introducción a ellos y de forma progresiva, la complejidad que esta codificación puede añadir a la comprensión de estos nos hizo optar por un enfoque más sencillo.

Aunque es cierto que este problema se podría haber abordado tomando los píxeles de la pantalla como un input para aprovechar la regularidad especial, tornándose un caso de visión por ordenador, creímos preferible abordarlo como un caso de inputs genéricos como los que comúnmente suelen encontrarse en otras tareas de aprendizaje profundo. Esto fue con el objetivo de poder extrapolar los conocimientos de este trabajo a futuros estudios en el campo, donde no necesariamente encontraremos regularidades espaciales.

ES-HyperNeat

Por los mismos motivos que en el caso anterior se decidió no optar por ES-HyperNEAT. Pese a ello, creemos conveniente reconocer la gran utilidad que podría aportar al introducir la capacidad de alterar la posición de las neuronas en el espacio geométrico, no sólo alterando las conexiones entre ellas. Esta nueva capacidad podría haber facilitado la búsqueda de soluciones óptimas, dada la complejidad del problema y el alto espacio de soluciones.

PGPE

(Policy Gradients with Parameter-based Exploration) Se encuentra disponible dentro de la librería EvoTorch. Es un método de optimización que utiliza gradientes de políticas (Es decir, se emplea mayoritariamente en aprendizaje por refuerzo) para actualizar los parámetros de una política directamente, basado en la exploración de los parámetros en lugar de las acciones. Este enfoque mejora la eficiencia al reducir la varianza en la estimación de gradientes.

Debido a que trabaja con gradientes, Neat supone una opción más variada que permite al alumno explorar métodos de optimización diferentes. Además, dispone de mucha menos documentación y ejemplos de uso.

CoSyNE

(Cooperative Synapse Neuroevolution). Está disponible también en la librería de EvoTorch. Un enfoque de neuroevolución en el que los pesos de las sinapsis de una red neuronal se optimizan de manera cooperativa mediante evolución. Las sinapsis se consideran individuos que evolucionan en paralelo, permitiendo una optimización eficaz. De nuevo, suele emplearse en problemas de aprendizaje por refuerzo.

Debido a que la evolución se produce sobre los pesos de las uniones red, creemos que Neat, con su variación de topología y otros hiper parámetros supone un mayor interés a la hora de explorar técnicas exóticas de neuroevolución no presentes en la retropropagación.

Motores de desarrollo de videojuegos

Pygame

Pygame es una biblioteca de Python diseñada para el desarrollo de videojuegos. Proporciona herramientas para manejar gráficos, entrada del usuario, manejo de eventos y otros elementos necesarios para nuestra simulación.

Es innegable que, en comparación a otras alternativas, Pygame queda atrás en temas de herramientas prediseñadas, facilidad de uso, soporte gráfico... Sin embargo, su punto más fuerte es que es el pseudo-motor de videojuegos de mayor calidad disponible en el lenguaje Python, y usar este lenguaje era un requisito indispensable para poder interconectar el entorno con Neat, y además, mejorar nuestros conocimientos del mismo.

Como otros puntos positivos que nos han hecho escoger esta biblioteca, podemos recalcar:

- Pygame es la biblioteca de creación de videojuegos con mejor documentación de Python y mayor base de usuarios, por tanto, con mayor cantidad de tutoriales y soporte en foros. Además, su uso con Neat ya ha sido demostrado en varios tutoriales, como los expuestos en la sección de “Implementaciones de Neat”
- Ha sido escogido por ser relativamente fácil de aprender y usar, especialmente por el alumno, que ya poseía conocimientos básicos de Python pero ninguno de Pygame.
- Al ser un pseudo-motor a nivel de código directo, a diferencia de las interfaces de otros motores de juego más avanzados, ha sido escogido por ofrecer una mayor libertad y control sobre los detalles de la simulación, además de suponer un mayor reto de programación.
- Ofrece funciones para dibujar figuras, manejar imágenes y realizar operaciones de transformación sobre gráficos. No era necesaria una mayor complejidad visual que esa. Estas han sido usadas para representar por pantalla el juego y las puntuaciones.
- En temas de rendimiento, aunque no está diseñado para ser el motor más rápido o avanzado, es suficientemente eficiente para proyectos de juegos 2D de pequeña o mediana escala. Necesitamos un rendimiento mínimamente aceptable para poder realizar las miles de simulaciones requeridas en un entrenamiento de red neuronal sin consumir un tiempo excesivo. Al desbloquear la limitación de frames por segundo, observamos entrenamientos donde se podían encajar de dos a tres goles en 0.3 segundos con redes neuronales enfrentadas.

Estas son algunas de las limitaciones o partes negativas que también tuvimos que tener en cuenta:

- A diferencia de motores de juego más avanzados (como Unity o Unreal Engine), Pygame carece de características avanzadas como física, motores de colisiones complejos, o herramientas integradas de desarrollo. Este punto débil nos ha obligado a programar manualmente las colisiones entre "objetos", las físicas de la aceleración y del rebote. Sin embargo, la mayoría de mejoras que estos otros motores aportaban apenas eran necesarias, pues en este proyecto se buscaba desarrollar un videojuego relativamente simple.
- Pygame es una biblioteca, no un motor de juego completo. Esto significa que, aunque ofrece muchas herramientas útiles, el desarrollador necesita implementar manualmente muchas funcionalidades. Esto ha estado presente durante todo el proceso de programación, y se ha hecho notar especialmente a la hora de trabajar con coordenadas y movimiento, representación gráfica, gestión de eventos… Que han tenido que ser programados prácticamente desde cero.

Panda3D

Este motor sí dispone de un sistema de físicas, y es ampliamente utilizado en simulaciones. Más importante, está disponible en lenguaje Python. Esta alternativa era la segunda mejor candidata de no haber empleado Pygame, sin embargo, el factor decisivo fue la cantidad de documentación de uso de Neat con Pygame, como mostramos en el apartado de "Implementaciones de Neat".

Dado que inicialmente no pensamos que se requiriera un gran trabajo en las físicas (Aunque durante el desarrollo fuimos conscientes de que sí sería necesario invertir tiempo en ellas), decidimos decantarnos por Pygame. En el momento de finalización de este proyecto consideramos que, de haber necesitado una ligera mayor complejidad en las físicas, esta habría sido la mejor opción.

Unity

Unity destaca sobre los demás motores por su facilidad para principiantes, multitud de herramientas, documentación y extensa variedad de tutoriales. Sin embargo, Pygame supone una mejor opción no únicamente debido a estar disponible en lenguaje Python, también por no contar con interfaces que ahorren la programación como si lo hace Unity. Aunque este último punto parezca contraintuitivo, nuestro deseo era programar la mayoría de lógica a mano y en una estructura de clases para poder acceder a ella con mayor facilidad desde Neat, y así mejorar nuestras habilidades de programación.

Cabe mencionar que Unity dispone de implementaciones para combinarse con Neat, y puede ser perfectamente considerado si el foco del proyecto buscase un videojuego mas complejo.

5.2 Arquitectura

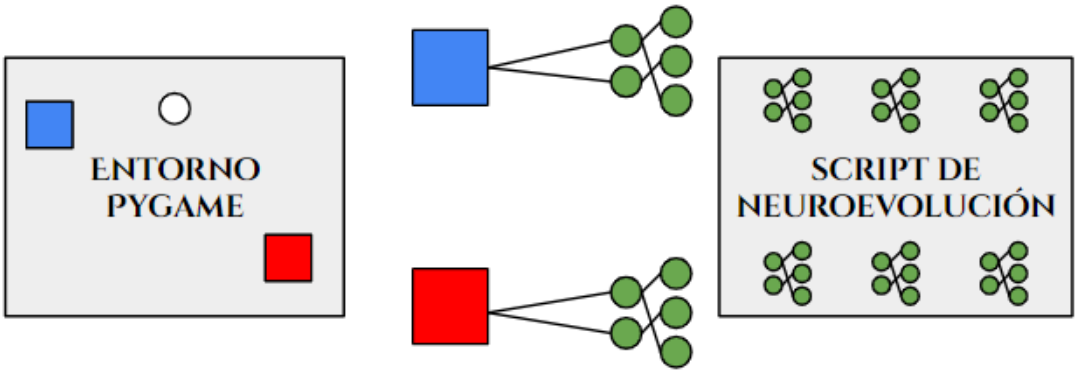

Figura 5.2: Relación simplificada del entorno Pygame y el script de neuroevolución

En la imagen superior se muestra la estructura básica y simplificada del proyecto. En esencia, los jugadores de colores azul y rojo que interactúan en el entorno pygame son controlados por las redes neuronales codificadas genéticamente en el script de neuroevolución. Estas redes son parte de una población mas amplia, cuyos miembros se alternan el control de cada jugador (En aquellos entrenamientos donde ambas redes son adversarias.)

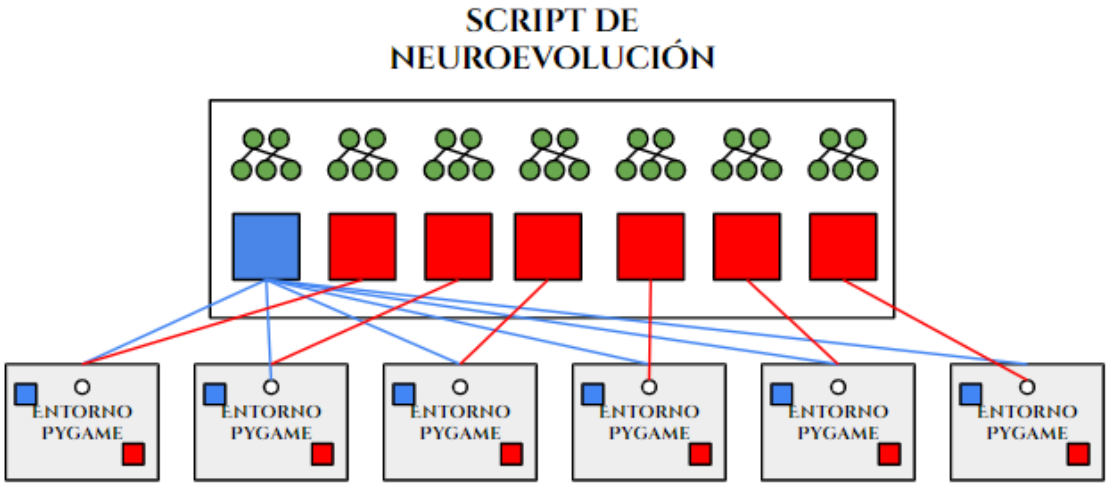

Figura 5.3: Bucle de evaluación enfrentado simplificado

La imagen superior muestra una perspectiva distinta y ligeramente mas compleja del mismo proceso. El script de neuroevolución actúa como un "organizador" que enfrenta consecutivamente a una red individual contra todas las otras redes de la población. (No todas las redes concurrentemente, sino en diferentes instancias del entorno de pygame) (Este caso de ejemplo, de nuevo, se refiere sólo a aquellos entrenamientos donde ambas redes se enfrentan.)

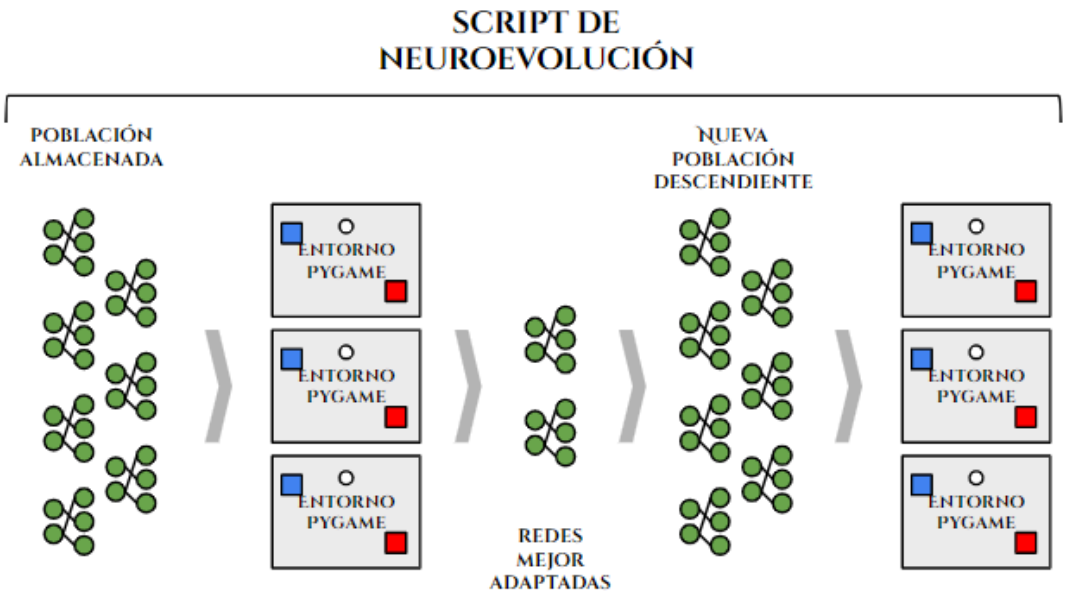

Figura 5.4: Bucle de evolución simplificado

Sin entrar en detalles, esta tercera imagen es otra réplica del mismo proceso, pero desde una perspectiva temporal y añadiendo los procesos mas básicos de evolución. El ciclo principal de entrenamiento es gestionado por el script de neuroevolución, que tomará una población previa y empleará los entornos pygame para medir la adaptación de las redes. (Bien enfrentándolas entre ellas, como se mostraba en las imágenes anteriores, o con entrenamientos en solitario.) Aquellas redes mas adaptadas al entorno se reproducirán entre ellas creando una nueva población, proceso que también gestiona el script de neuroevolución. El ciclo continua hasta alcanzar una puntuación de adaptación establecida o llegar a un máximo de generaciones establecido.

5.3 Diseño detallado

Diseño principal

Entorno de simulación

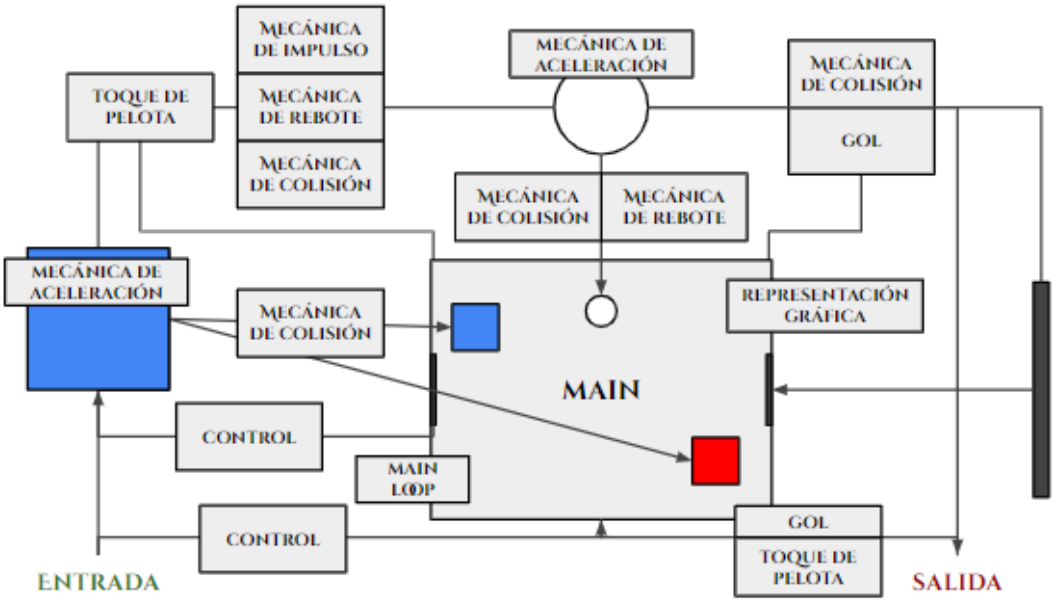

Figura 5.5: Diseño del entorno Pygame

Esta imagen muestra la estructura de nuestro entorno en Pygame. Cada elemento del partido (El jugador, la pelota, las porterías, los muros) se codifica en su propia clase.
La clase principal, indicada en la imagen como "main", contiene:

- Los muros del estadio con su figura de colisión y figura gráfica (La figura gráfica se dibuja en la representación gráfica del bucle principal, mientras que la figura de colisión se emplea en las mecánicas de colisión)
- Las configuraciones e instanciaciones de los elementos de las otras clases
- La lógica de partido y la representación de goles y toques de pelota
- El bucle principal de dibujado gráfico, medición de tiempo y actualización de todos los elementos del partido, asi como su reseteo

Esta clase principal actúa como un nexo para el resto de clases, y supone la conexión entre el algoritmo de neuroevolución y los inputs y outputs necesarios.

La clase del jugador, representada por el cuadrado azulado, puede ser instanciada como un jugador de un equipo u otro, o como varios jugadores de ambos equipos. Contiene:

- La figura de colisión y la figura gráfica
- La mecánica de aceleración y deceleración que permite un movimiento realista
- La mecánica de colisión para no superponerse a la pelota o a los muros del estadio

La clase de la portería es relativamente simple, conteniendo únicamente:

- La figura de colisión y la figura gráfica

La clase de la pelota, representada por el círculo, presenta estos elementos:

- La figura de colisión y la figura gráfica
- La mecánica de aceleración y deceleración que permite un movimiento realista

- La mecánica de colisión para evitar la superposición
- La mecánica de impulso o rebote con el jugador en base a la velocidad del impulso producido (Si la pelota impacta con velocidad superior a la del jugador, el jugador actuará como un muro de rebote y no como un impulsor)
- La llamada a la función de "pelota tocada" en main cada vez que entre en contacto con el jugador
- La llamada a la función de "gol marcado" en main cada vez que entre en contacto con una portería

Script de neuroevolución

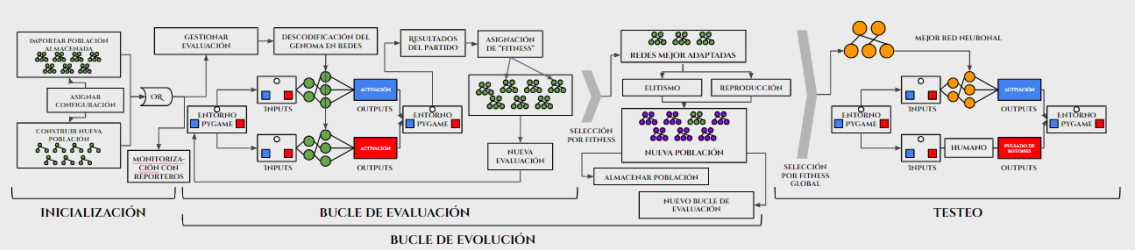

Figura 5.6: Diseño del script de neuroevolución

En la imagen superior observamos el script de neuroevolución. Por la naturaleza de esta sección de, se ha elegido una programación lineal, y en la imagen se muestran los elementos del script según el orden de su empleo, de izquierda a derecha.

Para simplificar su comprensión, explicaremos el mismo en sus diferentes fases:

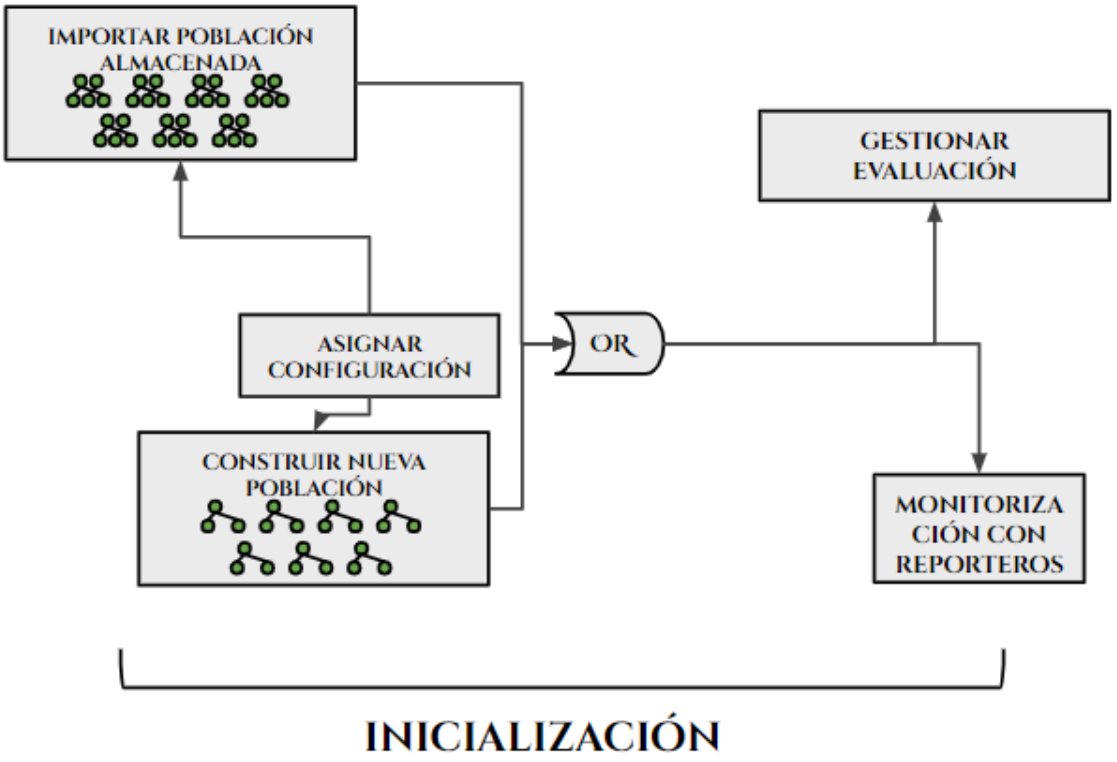

Figura 5.7: Fase de inicialización del script de neuroevolución

Como se muestra en la imagen superior, durante el periodo de inicialización deberemos tomar una población de genomas, cada uno de ellos codificador de una red neuronal.

Tenemos dos opciones, representadas por la puerta lógica or para entender mejor la imagen:

- Si ya hemos realizado alguna neuroevolución previa, habremos almacenado la población resultante en el directorio del proyecto, por lo que la importaremos.
- De no existir una población, esta se generará de manera aleatoria y con una profundidad relativamente baja gracias al archivo de configuración.

Una vez importada la población, se le vinculará la configuración Neat que importamos de nuestro archivo de configuración. Esta será necesaria para dirigir varios pasos del bucle de neuroevolución, cómo la reproducción. Además, al aplicar la configuración realizaremos un control que asegure que las redes neuronales codificadas en los genomas se corresponden con los parámetros de configuración, como el número de inputs u outputs.

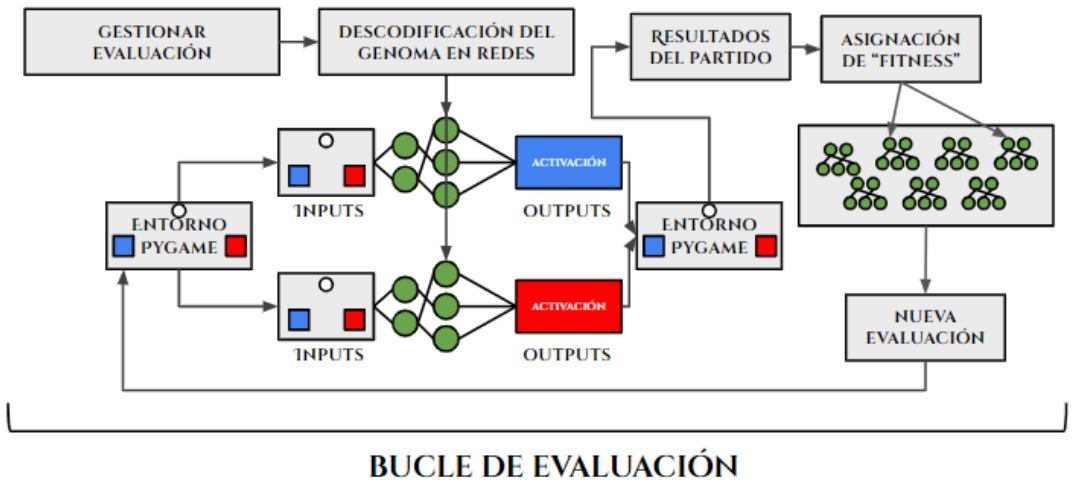

Figura 5.8: Fase de evaluación del script de neuroevolución

Guiándonos por la imagen superior, nos centraremos en el bucle de evaluación, que se emplea para medir el desempeño de los individuos en nuestro partido de fútbol:

Veremos que a lo largo del bucle de evaluación usaremos un gestor de evaluación para realizar diferentes tipos de evaluaciones que se explican en la imagen y texto posterior. El gestor elige qué genomas miembros de la población se deben enfrentar en el partido.

Los genomas elegidos por el gestor deberán decodificarse para construir las redes neuronales que emplearemos en el partido. Es decir, se estructurarán sus genes de neurona y sus genes de conexión, contenidos en el genoma.

Iniciaremos el entorno de partido en pygame. Del entorno tomaremos la información de partido necesaria para que las redes puedan tomar decisiones. Como mostramos en el apartado de "Desarrollo de la solución", la cantidad y tipo de *input* suministrado a la red variará durante nuestra técnica de entrenamiento progresivo. Esa información se suministra a las neuronas de entrada de la red.

La red se activa, y procesa la información de los inputs a través de sus capas ocultas, hasta devolver por sus neuronas de salida el movimiento a realizar. Las neuronas de salida de la red son siempre 5 a lo largo de todo el proyecto: Representan moverse a izquierda, derecha, abajo, arriba, o no hacerlo. Cada una de las neuronas exporta una probabilidad, que sumada con las probabilidades del resto de neuronas de salida resulta en un 100%. Es decir, que las redes neuronales de este problema actúan como si se tratara de un problema de clasificación multiclase, donde se pretende predecir cuál es el movimiento más adecuado para la información recibida.

De entre las probabilidades exportadas de la red, se toma aquella con mayor valor, asociado a un movimiento, y este se envía al entorno de Pygame para que el jugador asociado con la red se mueva de forma acorde. Este proceso sucede por cada *tic* del reloj de Pygame, es decir, por cada *frame* de simulación. Para entenderlo más simplemente, cada vez que la clase main de

nuestra simulación reevalúa la situación de todos sus elementos, en su *main loop*, nuestra red toma una decisión en base a la nueva información que le transmite sus inputs.

Cuando la partida finaliza, o se le fuerza a finalizar, (En la sección "Desarrollo de la solución" mostramos que el criterio de finalización del partido es cambiante según la fase del entrenamiento.) se obtienen los resultados del partido (De nuevo, en diferentes etapas del entrenamiento será necesario recopilar diferentes datos.) y se llama a la función de asignación de *fitness*.

La función de asignación del *fitness* emplea los datos recopilados del partido para otorgar una puntuación de adaptabilidad a los dos genomas involucrados en el partido que acaba de concluir. Los criterios de asignación *fitness* variará según la fase del entrenamiento en la que nos encontremos.

Finalizado el partido, se cerrará el entorno y se iniciará una nueva evaluación, con el mismo procedimiento, hasta que no queden genomas por ser puestos a prueba.

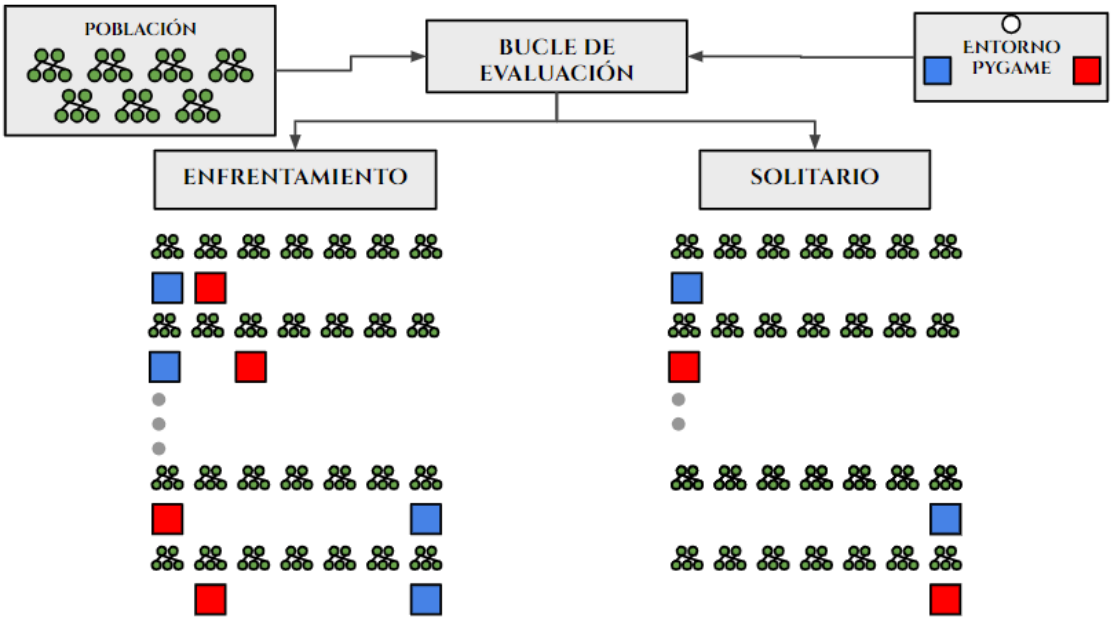

Figura 5.9: Técnicas de evaluación del script de neuroevolución

Para la gestión de la evaluación se han diseñado dos tipos generales de evaluación: En solitario o enfrentada. La imagen superior muestra, en cada una de ellas, que redes/genomas intervienen:

- En un enfrentamiento, la primera red de la población toma el papel del jugador azul, y se enfrenta consecutivamente a todas las otras redes que tomarán el papel del jugador rojo. Una vez acaba esta tanda de evaluaciones, el papel del jugador azul le corresponde a la siguiente red en la población, enfrentándose de nuevo a todas las otras redes, incluida la primera. Este tipo de entrenamiento expone a todas las redes contra todas las redes, imitando la competitividad de un ambiente natural. Además, cada red se habrá

situado tantas veces en el lado azul como en el lado rojo, fomentando la exploración de diferentes situaciones

- En solitario, la primera red se expone al problema desde el lado derecho y luego desde el lado izquierdo, sin ninguna otra red en juego. Esta aproximación requiere muchas menos iteraciones sobre el bucle de evaluación que su alternativa, pero es sólo útil para asentar las bases del aprendizaje de la red, ya que no está experimentando una situación de partido real. Como veremos en el apartado de "Desarrollo de la solución", cada opción presente sus ventajas y desventajas, y es labor del ingeniero juzgar cual es la mejor alternativa en determinado momento del entrenamiento.

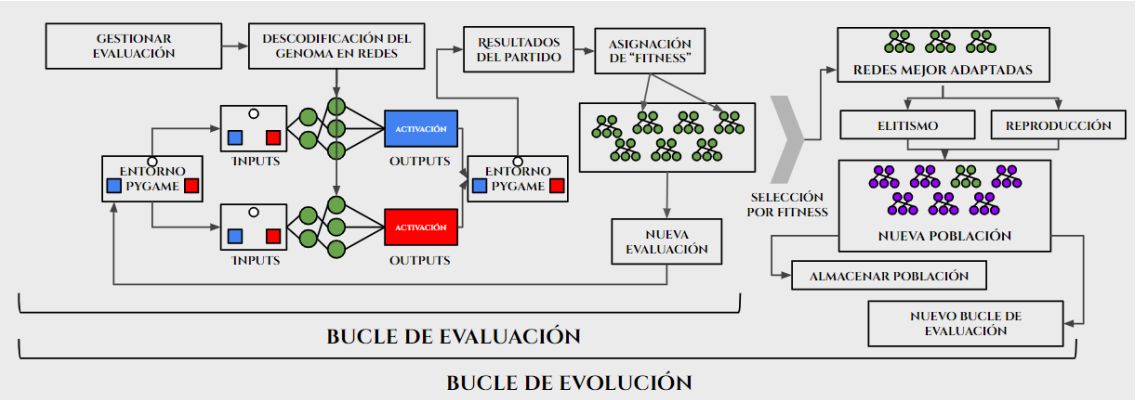

Figura 5.10: Fase de evolución del script de neuroevolución

El bucle de evolución es una anidación del bucle de evaluación, iterando sobre el primero cada vez que se crea una nueva población para evaluar. En la imagen superior vemos su proceso:

Cuando un bucle de evaluación termina, todos sus genomas/redes han sido evaluados y se les ha sido asignada una puntuación de *fitness*. El primer paso del bucle de evolución es seleccionar a un porcentaje (estipulado en nuestra configuración) de los individuos mejor adaptados al partido en base a esa puntuación. En nuestro caso, priorizaremos los que tengan un mayor fitness, aunque podríamos haber configurado priorizar el mínimo si usábamos el fitness para representar los errores en el partido, por ejemplo.

Los genomas con mayor adaptabilidad dispondrán de dos mecanismos para transmitir su información genética a la siguiente generación:

- El elitismo, un parámetro activado en nuestro proyecto, conservará íntegramente a una cierta cantidad de los individuos mejor adaptados, cantidad estipulada en nuestra configuración.
- La reproducción cruzará a dos individuos aleatorios de entre los mejores adaptados para crear a un nuevo individuo de la futura población, repitiendo este proceso hasta que la población tenga el tamaño que se estipula en nuestra configuración.
  En el proceso de cruce genético, aquellos genes que comparten el mismo identificador, llamados *matching genes* se conservarán de forma aleatoria, pero aquellos que solo

están presentes en un padre, los llamados *excedent* y *disjoint genes,* serán conservados solo si el padre es el que posea una mayor puntuación de fitness.

La nueva población se almacenará en el directorio para poder ser importada tal y cómo se indica en la fase de inicialización. Esto se empleará para poder realizar entrenamientos progresivos.

Acabado este proceso, se iniciará un nuevo bucle de evaluación para la población de descendientes. Una vez se haya producido la cantidad de generaciones estipulada en el código, o se haya llegado a un límite de *fitness* establecido en nuestra configuración, el bucle superior, de evolución, también finalizará.

Cabe destacar que a lo largo del bucle de evolución se almacena el mejor genoma de todas las generaciones, pues la población de descendientes podría resultar menos adaptada que sus ancestros.

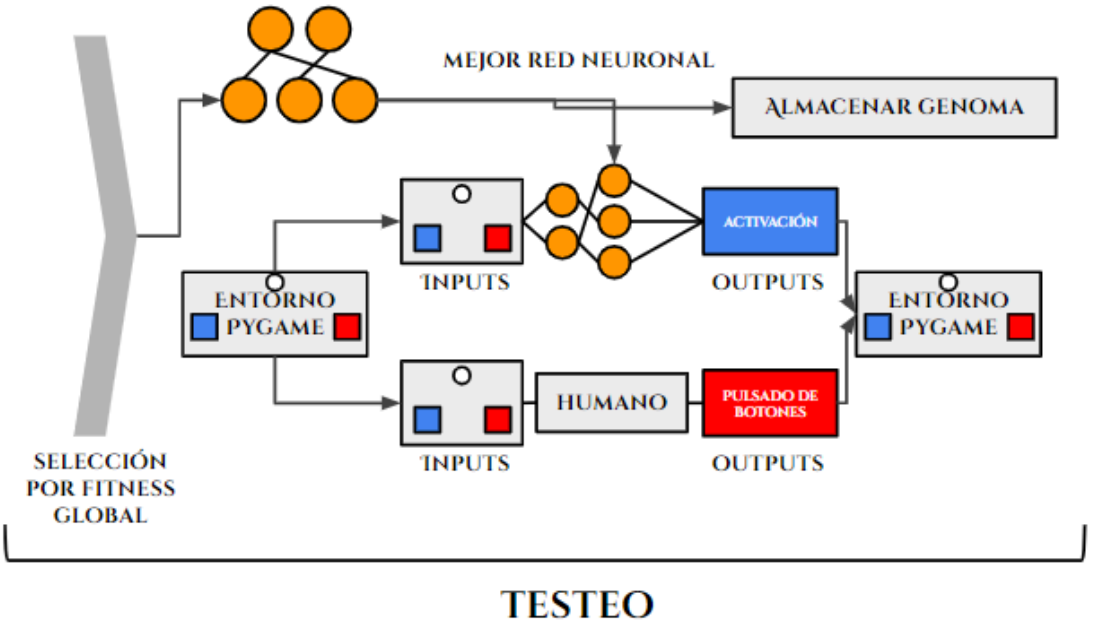

Figura 5.11: Fase de testeo del script de neuroevolución

La última fase se trata del testeo, mostrada en la imagen superior:

En esta fase se obtiene el mejor de todos los genomas del proceso de evolución, y se almacena para que esté disponible en el directorio a la hora de hacer un testeo rápido sin pasar por el resto de fases.

El genoma de nuevo se decodifica para construir una red, y se realiza el mismo proceso que en el bucle de evaluación, a excepción de que en este caso su oposición no será otra red sino el propio humano controlando al otro jugador por teclado.

Diseño de funcionalidades extra

Incremento de inputs progresivo

En el apartado de "Desarrollo de la solución" se menciona como en el inicio del proceso de entrenamiento de la red se hizo evidente la necesidad de simplificar el aprendizaje de una forma u otra, ya que la red contaba con una canta de inputs demasiado grande en comparación con los ejemplos que pueden verse en "Implementaciones de Neat"

Para entender el problema decidimos fijarnos en cómo funciona el aprendizaje en los seres vivos, y especialmente en los humanos. Por ejemplo, al aprender la asignatura de matemáticas, no es asumible enseñar a un niño a resolver problemas de trigonometría si aún no ha comprendido el concepto de posición o de distancia. Incluso si nuestro algoritmo es capaz de similar a cientos de miles de individuos, pretender que se establezcan conexiones "mentales" tan complejas en un espacio de soluciones tan amplio requeriría de largas simulaciones. Es posible, por supuesto, pero requeriría una computación que no podemos afrontar con nuestro dispositivo ni con un tiempo limitado.

En cambio, optamos por seguir una estrategia menos "natural" y mas "humana." Nuestro diseño se basa en la misma estrategia que utilizaría un profesor: Si el alumno no es capaz de aprender trigonometría de un vistazo, si puede ser enseñado los conceptos mas simples, asegurándose de que los comprende completamente, e ir progresando hasta llegar a esa complejidad deseada.

En esencia, reduciremos los inputs de la red a un mínimo, e iremos aumentando estos conforme se vayan cimentando el aprendizaje de los conceptos deseados. Destacamos que esta funcionalidad no está presente en el algoritmo Neat, no ha sido encontrada en la documentación para este proyecto y tampoco durante la búsqueda en internet del alumno para resolver el problema, así que se presupone con cierta seguridad que es novedosa.

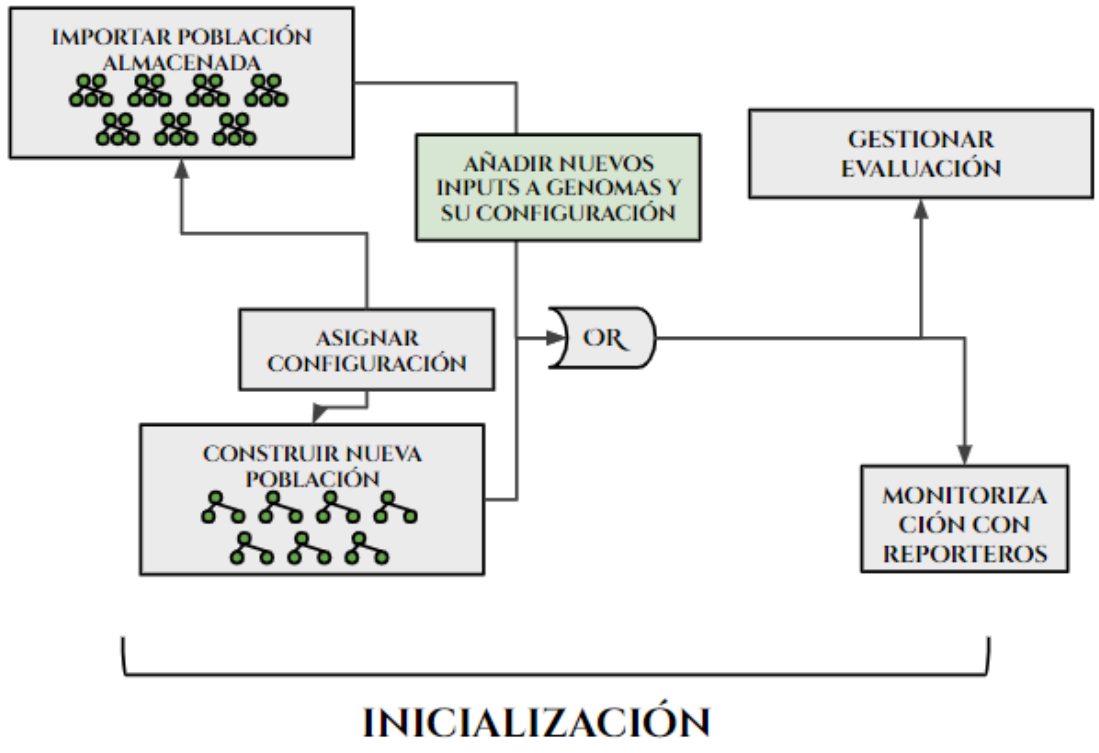

Figura 5.12: Adición de inputs en la fase de inicialización del script de neuroevolución

Esta funcionalidad se ubica en la fase de inicialización. (Marcada en color verde en la imagen superior). Se indicará en el archivo de configuración la nueva cantidad de inputs deseada. Cuando se importe una población almacenada, se aumentará el número de inputs de cada uno de sus genomas para ajustarse al número de inputs especificados en la configuración

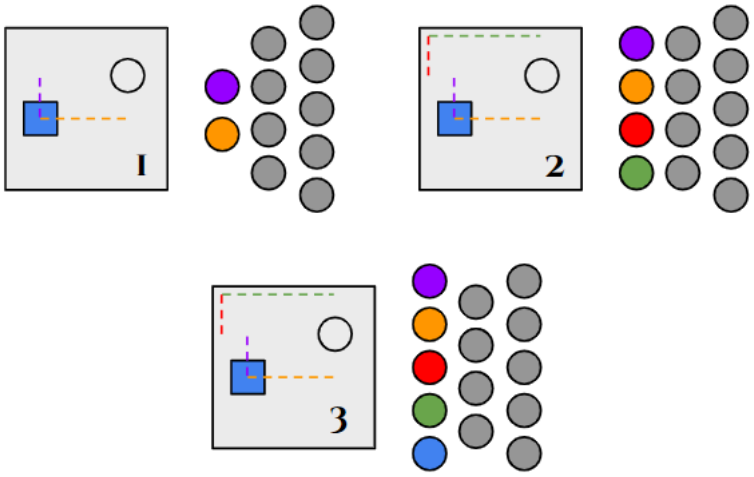

Figura 5.13: Evolución de una red con el incremento de inputs

Para explicar cómo transcurre un entrenamiento empleando esta funcionalidad, tomaremos como ejemplo el mismo proceso que tomamos al entrenar nuestra red.

Inicialmente querremos entrenar una red capaz de comprender su distancia a la pelota, representada en altura y anchura por los colores morado y naranja en el caso 1 de la imagen superior. Para ello, estableceremos en nuestra configuración que nuestra red solo posee dos inputs, y vincularemos estos con los datos de distancia obtenidos desde el entorno Pygame.

Entrenaremos las redes con los bucles de evaluación y evolución hasta que esta sea capaz de abalanzarse a por la pelota y chutar desde varias situaciones espaciales, momento en el que podremos concluir que la red ha aprendido correctamente estos conceptos.

En segundo lugar querremos que la red aprenda también la posición vertical y horizontal de la pelota, representada con los colores verde y rojo en el caso 2 de la imagen superior. Para ello querremos importar la población de redes ya entrenadas para comprender la distancia del jugador a la pelota. Sin embargo, al importar esta, nuestro programa devolverá error porque hemos establecido cuatro inputs en nuestra configuración, mientras que nuestra red cuenta con solo dos inputs.

En este momento la nueva funcionalidad añadirá dos *inputs* más a la red, representados con los colores rojo y verde porque están asociados a las coordenadas de la pelota mentadas anteriormente. La red ahora podrá ahora entrenar para aprender la posición relativa de la pelota.

Al comprobar que la pelota aprende a disparar hacia una portería, por ejemplo, podemos concluir que está teniendo en cuenta la posición de la pelota en el espacio y posicionándose para tirar.

Los inputs no solo pueden representar distancias, por supuesto. En el caso 3 se le añade un quinto input a la pelota para representar su equipo, el azul, y que esta aprenda a marcar en la portería del equipo rival y nunca en la suya.

Creación de especialistas

Aunque nuestro algoritmo de aprendizaje nos permite obtener al mejor "individuo generalista", tal vez queremos entrenar a individuos especializados en un cierto tipo de juego. En esencia, deportes cómo el fútbol necesitan de agentes que destaquen en su cometido, y no de forma general. Un buen portero, un buen centrocampista, un buen delantero…

Desde el punto de vista educativo, como el planteado en el apartado anterior, podríamos ver esta funcionalidad como una aplicación de la teoría de las inteligencias múltiples [3] donde se estipula que cada ser humano posee una combinación de diferentes inteligencias en distintos grados, y por tanto evaluar a todos los alumnos bajo un mismo espectro no es efectivo a la hora de encontrar a individuos con capacidades únicas.

Queremos representar diferentes tipos de puntuación de "fitness" para poder conservar durante la reproducción a los individuos que más destaquen en alguna de ellas. Recordamos que el fitness es un atributo asociado a cada genoma. Este se vé modificado durante la fase de evaluación. En nuestro caso, se actualiza cada vez que un genoma participa en un partido, lo cual sucede varias veces por bucle de evolución.

Esta es una mejora sobre el algoritmo Neat base, dado que en él, el "fitness" de un genoma mide su desempeño en general y no existe mecanismo para diferenciar en él si la puntuación se vincula a goles, atajadas o control del balón. Por ejemplo, en una de nuestras últimas configuraciones, el "fitness" aumentaba con la cantidad de goles marcados y disminuía con los encajados, pero finalmente se almacenaba como un simple entero. Hemos decidido buscar tres clases de jugadores, en base a tres tipos diferentes de "fitness":

- Porteros/Defensas: Aquellos con el mejor "fitness" de defensa, que es mayor cuantos menos goles se hayan encajado en un partido.
- De ataque, que es mayor cuantos más goles se marquen en un partido.
- De control, que es mayor cuantas más veces se toque el balón y menos veces lo toque el rival a lo largo del partido.

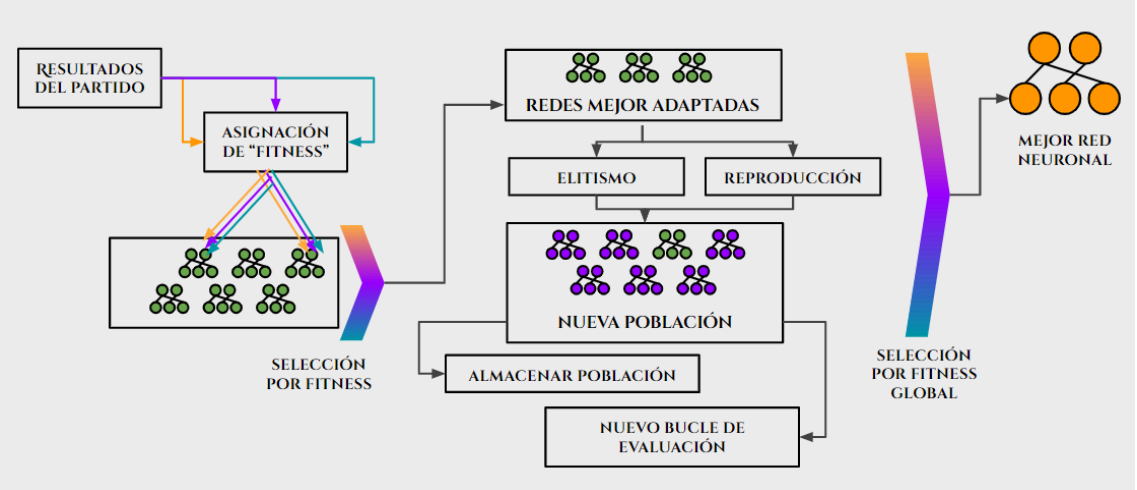

Figura 5.14: Creación de especialistas dentro de las fases de evaluación y evolución

En la imagen superior mostramos aquellas fases del script de neuroevolución afectadas por esta mejora:

En primer lugar, los resultados obtenidos del partido serán diferentes para los tres *fitness,* cómo se explica en la clasificación de los tres jugadores del texto superior. Esos datos servirán para realizar tres asignaciones de fitness diferentes a cada uno de los genomas involucrados en el partido.

Consecuentemente, la selección de los mejores individuos de la población en base a su *fitness* ahora escogerá a una cierta cantidad de mejores redes adaptadas para cada categoría de fitness. El elitismo y la reproducción se realizará teniendo cada una de las categorías de fitness en cuenta de forma individual, de modo que un tercio de la población será fruto del elitismo y cruce en base al fitness de ataque, otro tercio en base al de defensa y otro en base al de control.

Finalmente, a la hora de seleccionar una mejor red neuronal global, será necesario también tener en cuenta los tres tipos de *fitness*, y por tanto, almacenar a los tres mejores individuos de todo el proceso de evolución.

Exploración de posiciones de pelota

Esta mejora es bastante simple de implementar, pero como se explica en el apartado de "Desarrollo de la Solución", resultó extremadamente útil para solucionar los problemas derivadas de un entrenamiento rápido. Consiste en colocar la pelota en diferentes posiciones verticales y horizontales para que la red aprenda a enfrentarse a situaciones reales que no se puedan explorar en un entrenamiento rápido.

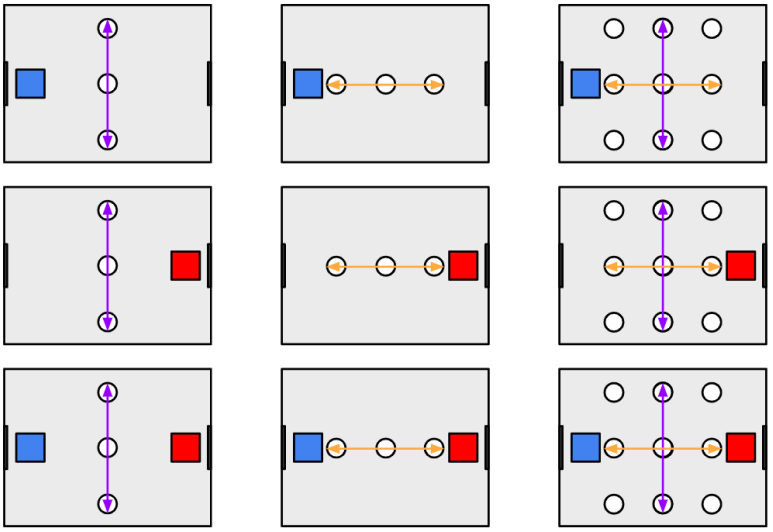

Figura 5.15: Disposición de la pelota en el entorno con la modificación de exploración de posiciones de pelota

En esencia la pelota se colocará en tres posiciones verticales, tres horizontales, y la combinación de ambas. Esto se realizará tanto en los entrenamientos individuales como en los enfrentados, de forma que el individuo se enfrente a todo tipo de posiciones relativas respecto a la pelota, hecho del que no nos podemos asegurar en el poco tiempo que dura cada una de las simulaciones,

Fitness de aproximación

Durante el apartado de "Desarrollo de la solución" mencionamos que uno de los problemas de entrenamiento está estrechamente relacionado con el "problema de la asignación de crédito", común en el aprendizaje por refuerzo. En esencia, una política no recibe una recompensa adecuada debido a la dificultad de medir cómo de importante es para la resolución del problema. En nuestro caso, durante una etapa del entrenamiento la red era incapaz de impactar con la pelota debido a la gran cantidad de espacio del estadio y lo poco que ocupaba esta, teniendo que probar movimientos en las cuatro direcciones hasta acertar, lo que sucedía en muy pocos momentos.

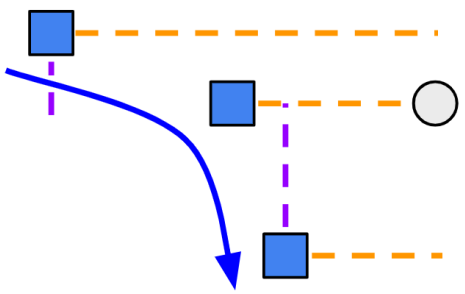

Figura 5.16: Medición de la máxima aproximación a la pelota

La solución adoptada se muestra en la imagen de arriba. El nuevo fitness asociado al genoma era mayor según como de cerca había estado de tocar la pelota en el momento más próximo a ella de su recorrido. Para ello se registraban todas las distancias (Sus componentes vertical y horizontal) a lo largo de la simulación, almacenando la menor.

6. Desarrollo de la solución

El contenido expuesto en este apartado indaga en los aspectos más importantes , complejos o novedosos de nuestra implementación, ofreciendo junto a ellos los problemas surgidos, como se han solucionado, y las alternativas de implementación contempladas. Los aspectos de menor interés se resumirán todo lo posible para mantener una comprensión mínima del conjunto.

Durante nuestra implementación seguimos la estructura de *sprints* mencionada en la sección de "Plan de Trabajo". Los problemas surgidos fueron evaluados en la fase de Revisión mencionada en el apartado de "Metodología" y en base a ellos, durante la fase de Retroalimentación se reestructuraron los sprints adecuadamente a la estimación del tiempo necesario para resolver dichos problemas.

6.1 Entorno de simulación

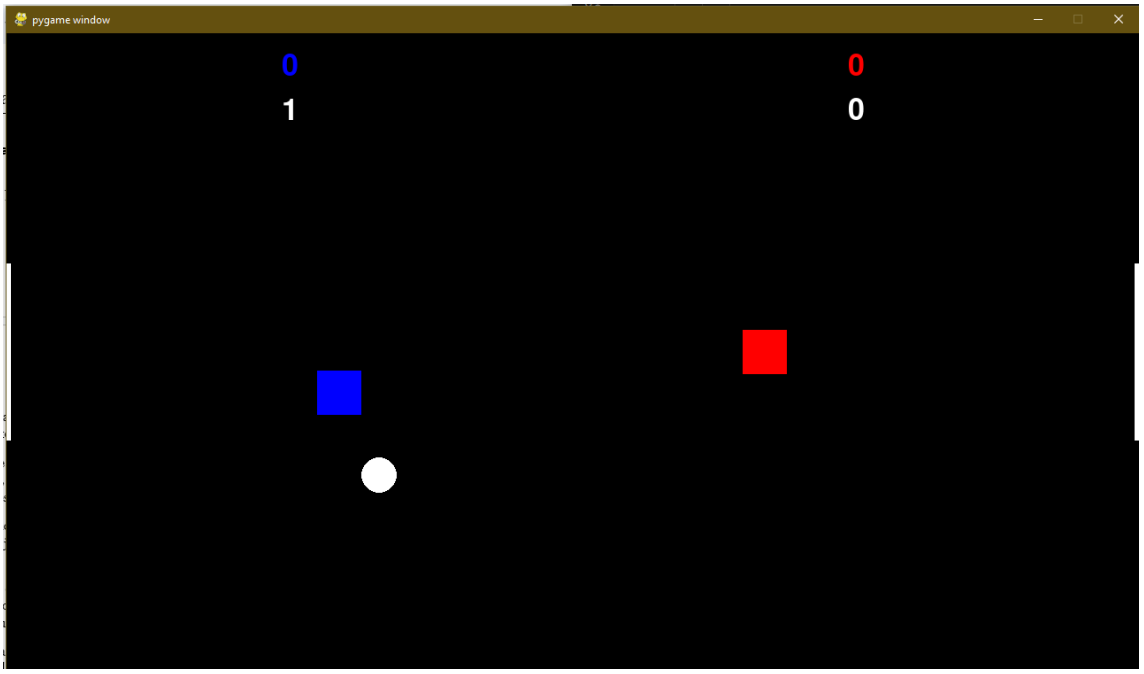

Figura 6.1:Entorno Pygame

Durante la creación del entorno de simulación nos expusimos a varios problemas que bloquearon nuestro desarrollo. Estos se mencionarán, junto a su resolución, en las clases en las que se produjeron.

Para solventar la ralentización que estos problemas supusieron, reestructuramos el sprint de desarrollo del entorno y añadimos un nuevo sprint para solventarlos, de forma exitosa.

A pesar de los problemas, fuimos capaces de crear el entorno que planteamos en el diseño, y este resultó consistente a lo largo de los entrenamientos gracias a las pruebas realizadas.

Una vez acabado un mismo script que contenía todo el código de esta implementación, decidimos reestructurarlo en diferentes clases para facilitar la posterior interacción con el algoritmo de neuroevolución. A continuación exploraremos la implementación de cada una de las clases que finalmente conforman el entorno:

Clase principal

La clase principal, llamada ***Game*** o ***Main*** en los diseños originales, actúa como un nexo entre el resto de clases del entorno.

En ella se declarara una serie de constantes que contienen los datos de configuración que posteriormente se emplearán para instanciar el entorno y el resto de clases:

- Tamaño de pantalla, fuentes y tamaño de fuentes
- Posiciones, tamaños y colores iniciales de los elementos
- Velocidades, aceleraciones y deceleraciones máximas

El método constructor de la clase se encarga de inicializar todos los elementos del escenario: Muros, jugadores, equipos de los jugadores en forma de listas, porterías y pelota.

Cabe destacar que en este método se crearán también dos *groups* de la biblioteca pygame, *all_sprites* y *collision_sprites* contenedores en los que almacenaremos *sprites* con el fin de utilizarlos para las mecánicas futuras. *Sprite* es un término de desarrollo de videojuegos referido a una imagen bidimensional para representar un objeto.

La clase contiene además dos métodos, *goalScored()* y *ballHit()* que serán llamados desde la clase **Ball** para almacenar los chutes y goles.

Otro elemento fundamental es el *main loop* o bucle principal: En él, se realizan las acciones asociadas a cada *frame* de la simulación. Un *frame* es la unidad mínima de tiempo, en la que se varía el estado de la simulación. Dentro de este bucle se llama a:

- La función *ide dibujado* para mover visualmente a los elementos del entorno en pantalla y posteriormente la aplicación de este cambio en pantalla.
- La función de actualización de todos los elementos del entorno cuyos sprites fueron almacenados en el *group all_sprites*. En esencia es una llamada a los métodos *update()* localizados en jugadores y pelota para que su posición cambie en base a sus velocidades.

Un error en el que la pantalla permanecía negra supuso un parón en el desarrollo, pero finalmente, y tras investigar otras implementaciones de Pygame, pudimos inferir que el problema lo causaba el orden de dibujo de los elementos representados en la función de dibujado, al dibujar en ultimo orden el fondo negro y eclipsar al resto de elementos.

Un último error, el que más tiempo consumió de todos, se produjo también en relación a esta clase. Dentro del bucle principal se añadió un objeto *pygame.time.Clock()* cuya función *clock.tick(60)* permitía que el videojuego no funcionara a la máxima velocidad de *frames* que el ordenador permitiese, sino a sesenta *frames* por segundo. Al llamar al método *loop()* de la clase ***Game*** para ejecutar la simulación, de la misma forma que posteriormente se hizo en el

algoritmo de neuroevolución, se llamaba de nuevo a un reloj de Pygame con ralentización a 60 segundos, lo que provocaba una lentitud anormal que no podíamos entender. Este último error se resolvió con prueba y error a lo largo de toda la secuencia de código.

Clase portería

La clase ***Goal*** es relativamente simple. Únicamente se utiliza para crear la imagen de las porterías y su rectángulo de colisión. Además permite identificar que sus instancias pertenecen a esta clase cuando la pelota colisione con ellas, para permitirle detectar el gol.

Clase jugador

La clase ***Player*** realiza la misma función que portería, pero además, añade dos nuevas mecánicas:

- El movimiento y la aceleración: El jugador cuenta con dos tipos de velocidades, vertical y horizontal, que pueden tomar valores negativos y positivos para representar arriba y abajo e izquierda y derecha.

  Cuando un input de movimiento llega al jugador, se aumenta o disminuye una de las dos velocidades en tanta cantidad como valor tenga la constante de aceleración.

  Cuando se llama a la función *update()* del jugador, se asegura que la velocidad en las cuatro direcciones no supere a la velocidad máxima establecida, y si lo hace se establece esta como la velocidad actual. La posición del rectángulo del jugador y del propio jugador se actualiza entonces, sumándole la velocidad a sus coordenadas x e y.

  A lo largo del desarrollo se probaron diferentes formas de realizar este movimiento. El enfoque original comprendía una valor de velocidad único combinado con un ángulo para formar un vector de velocidad, pero este era incapaz de representar correctamente una deceleración en una dirección mientras se producía una aceleración en otra. Por ello se optó por separar las velocidades, y representar el movimiento como una suma de ellas.

- Colisiones: Se llama a la función *collision()* cada vez que se llama a *update()*. La finalidad de esta función es comprobar si el *sprite* del jugador colisiona con el de otro objeto. Además, dentro de la llamada a *update(),* se guarda en la variable *old_rect* la posición del rectángulo en el *frame* anterior.

```
def collision(self):
    collision_sprites = pygame.sprite.spritecollide(self,self.obstacles,False)
    for player in self.otherplayers:
        if self.rect.colliderect(player.rect):
```

```
            collision_sprites.append(player)
        if collision_sprites:
            if self.x_speed != 0:
                for sprite in collision_sprites:
                    if self.rect.right >= sprite.rect.left and self.old_rect.right <= sprite.old_rect.left:
                        self.rect.right = sprite.rect.left
                        self.pos.x = self.rect.x
```

Código 6.1: Mecánica de colisión

Como observamos en la imagen superior, se llama a la función *spritecollide()* para recolectar en una lista todos aquellos sprites de los elementos considerados obstáculos. Estos son pasados al jugador dentro de una lista cuando es instanciado en la clase ***Main.*** Si además el rectángulo perteneciente a otro jugador está en contacto con el actual, se añadirá de la misma forma a los *sprites* de colisiones.

En el caso de la colisión en el lado derecho del rectángulo del jugador, si existe algún *sprite* en contacto con nuestro jugador, y nuestro jugador lleva algún tipo de velocidad hacia la derecha, se comprobará que el lateral derecho de nuestro rectángulo no se encuentre más avanzado que la parte izquierda del elemento con el que ha chocado. para ello, además, se comprueba que la variable *old_rect,* que pertenece a la posición del rectángulo en el *frame* anterior, no tuviera esta superposición. De ser así, el rectángulo y el jugador se moverán hacia la izquierda, en una posición donde no se superponen.

Solo hemos mostrado el código de una colisión horizontal y hacia la derecha para no ser redundantes. El mismo proceso, cambiando las variables como podría esperarse, se produce para los cuatro lados.

Una lógica muy similar se produce si se colisiona con los muros del estadio, simplemente comprobando que los bordes del rectángulo no excedan el tamaño de la pantalla y aplicando las mismas modificaciones a su posición.

Existe un problema entre las colisiones del jugador y la pelota, ya que a veces esta atraviesa al jugador si se le empuja contra una pared y no tiene manera de rebotar. Tras intentar rehacer la mecánica de colisiones de otra manera, fuimos incapaces de solucionar este error, pero creímos que no afectaría se forma significativa al entrenamiento.

Clase pelota

La clase pelota construye sobre las mecánicas ya añadidas en la clase ***Player***. Únicamente mencionaremos cómo funciona su mecánica de rebotes y gol, que se integran dentro de la colisión. Para entenderla será necesario haber leído la explicación de la mecánica de colisiones en la clase ***Player.***

```
                if isinstance(sprite, Player):
                    self.rect.right = sprite.rect.left
                    self.pos.x = self.rect.x
                    self.x_speed = min(sprite.x_speed*1.5, self.x_speed *-1)
                    self.y_speed += sprite.y_speed*0.5
                    self.game.ball_hit(sprite.team)
                elif self.goal_L in collision_sprites:
                    self.game.goalScored("left")
                elif self.goal_R in collision_sprites:
                    self.game.goalScored("right")
                else:
                    self.rect.right = sprite.rect.left
                    self.pos.x = self.rect.x
                    self.x_speed = self.x_speed * -1
```

Código 6.2: Mecánicas de chute y rebote

En la clase pelota se tomarán los mismos procedimientos de colisión. Pero tal y como vemos en la sección de código superior, que pertenece a una parte de la función *collision()* de ***Ball***, si la pelota ha entrado en contacto con un sprite perteneciente a un jugador, en el caso de un choque en su lado derecho, se le aplicará el mínimo de dos posibles velocidades, o dicho de otra forma, la velocidad horizontal más negativa de entre:

- La velocidad horizontal del jugador con un cierto impulso para representar un chute
- La velocidad de la propia pelota, invertida, para representar un rebote.

Es comprensible que se compare que velocidad es más negativa, pues si el choque es en el lado derecho, querremos que la pelota se mueva hacia la izquierda.

Esta comparación entre velocidades nace fruto de un error en el desarrollo de esta mecánica, pues originalmente la pelota tomaba únicamente la velocidad presente en el jugador.

Este era un problema de lógica, ya que cuando la pelota entraba en contacto con el jugador, si este se movía en la misma dirección de la pelota, la pelota le seguía pegada a el.

Ademas, si la colisión se realizó con un jugador o con alguna de las dos porterías, se llamará a las funciones correspondientes en ***Game*** para notificar un chut o un gol. Si la colisión se realiza con algún otro elemento, la pelota simplemente rebotará.

Como podemos intuir, este mismo procedimiento se repite para las cuatro direcciones con los cambios pertinentes.

6.2 Script de neuroevolución

Hablaremos de la implementación del script de neuroevolución en el orden en que sus elementos son usados, para facilitar la comprensión de los pasos. Este script no se ha estructurado en clases porque tiene una estructura bastante secuencial.

No experimentamos problemas graves durante el desarrollo de este script, hubo dudas sobre ciertos aspectos del proceso, pero por suerte la documentación de Neat expuesta en [11], y especialmente sus ejemplos encontrados en el repositorio de Neat-Python fueron una gran fuente de ayuda para solventarlas.

Inicialización

En primer lugar, importaremos del directorio de nuestro proyecto el archivo de configuración de Neat, a través del cual podemos introducir alteraciones en la estructura de las redes o en los procesos de su evolución de forma rápida.

Tomaremos una población de genomas inicial para la neuroevolución:

- Si no disponemos de una población almacenada, crearemos una aleatoria con muy poca profundidad en base a la configuración importada
- Si disponemos de una, importaremos su *checkpoint* de nuestro directorio.

A dicha población se le vincularán los reporteros, que son mecanismos de monitorización y de salvado:

- El primero será un *neat.StatisticsReporter* con objetivo de mostrar por pantalla la evolución del entrenamiento.
- El segundo será un *neat.Checkpointer*, cuya finalidad será almacenar las poblaciones generadas por cada cierto número de bucles de evolución. Debido a la larga longitud temporal de los bucles de evolución, siempre guardaremos cad auna de las poblaciones generadas.

Evaluación y evolución

El siguiente paso finaliza con la fase inicialización, llamando al bucle de evolución de las redes: llamaremos al método *run* presente en la clase *population*, el cual toma como argumentos la función de evaluación y el número de bucles de evolución a realizar. Este método *run,* además, devolverá al genoma con un mejor fitness global después de todos los bucles de evolución.

```
winner_nn = population.run(one_vs_all_training, 50)
```

Código 6.3: Función run de Neat

Con la línea de código de encima de este texto entrenaremos nuestra población de genomas durante 50 bucles de evolución (Es decir creando 50 nuevas generaciones), y empleando como función de evaluación *all_vs_all_training* (Que explicamos en el siguiente párrafo). El genoma ganador resultante se almacenará en la variable *winner_nn* , mientras que las poblaciones se guardarán gracias al *neat.Checkpointer* mencionado previamente.

Existen dos funciones de evaluación principales (Que han ido variando ligeramente a lo largo del entrenamiento):

- *one_vs_all_training()*: Corresponde a la estrategia de enfrentamiento de cada genoma contra todos los demás como mencionamos en el apartado de "Diseño de la Solución".

  Consiste en dos bucles for anidados, el primero iterando sobre los genomas de la población que se colocarán en el equipo azul / lado izquierdo, mientras que el bucle interior lo hace sobre los genomas que se colocarán en el lado derecho. Con la comprobación "*if genome_right is not genome_left"* nos aseguramos que un genoma no se enfrente contra él mismo.

  Dentro del bucle interno se llama a la función *one_vs_all_training_ingame(),* que continua la evaluación dentro del entorno Pygame.

  Uno de los problemas derivados de este tipo de evaluación, en contraposición con *switching_sides_training()*, es el coste temporal mucho mayor, que además escala de forma exponencial con el tamaño de la población. Por supuesto, la ventaja es que permite simular un enfrentamiento real.

  Una de las alternativas originales a esta evaluación fue enfrentar a cada genoma con otro genoma aleatorio. El problema de esta implementación es que creaba situaciones potencialmente injustas, donde un genoma codificaba una red con una adaptación buena, pero al enfrentarse con la mejor red de la población, este apenas obtenía el *fitness* necesario para ser seleccionado. Además, podía darse el caso de que algunos genomas fueran seleccionados aleatoriamente más de una vez, recibiendo mucho más *fitness* que el resto (Ya que este se acumula por cada evaluación en la que la red codificada esté presente), y que algunos no obtuvieran puntuación de *fitness* apenas por no ser elegidos aleatoriamente.

- *switching_sides_training():* Corresponde a la estrategia de evaluación solitaria de cada genoma cambiando de lado, como mencionamos en el apartado de "Diseño de la Solución".

  Presenta varias ventajas respecto al tipo de entrenamiento anterior, especialmente la velocidad de entrenamiento, con un coste temporal que escala de forma lineal con el tamaño de la población, en lugar de escalar.

En los inicios del desarrollo se pensó en enfrentar a un humano contra las redes para entrenarlas. Rápidamente se descartó la idea, debido a la cantidad de bucles de evaluación y evolución por los se que debería pasar jugando al videojuego. Además, al usar un entorno totalmente compuesto por IAs, podemos acelerar la simulación eliminando el *pygame.time.Clock* como se comenta en la explicación de la función de *training_ingame()*.

Ambas funciones de evaluación, una vez pasan a su tramo dentro del entorno Pygame son muy similares, asi que los explicaremos de forma conjunta refiriéndonos a ellos como *training_ingame()*. Dentro de esta función se comienza descodificando los genomas que van a enfrentarse para obtener las redes neuronales asociadas a ellos.

Se instancia un entorno de Pygame pasándole el tamaño de ventana deseado. El tamaño de ventana no debe alterarse nunca durante el entrenamiento, pues las redes habrán aprendido con un tamaño diferente y pueden no funcionar correctamente. Este conocimiento proviene de un error durante nuestro entrenamiento, aunque por suerte fue resuelto rápidamente. Asimismo, no añadiremos a esta simulación *pygame.time.Clock* ni *clock.tick(60)* para que la simulación corra a tanta velocidad como permita nuestro equipo.

Con un bucle while que funciona eternamente realizaremos las activaciones de ambas redes enfrentadas, llamaremos a la función *loop()* de la clase ***Game*** para ejecutar el *main loop*, y finalmente, estableceremos unas condiciones para cerrar el bucle while cuando se cumplan nuestros requerimientos (Por ejemplo, al haberse marcado dos goles.) y además llamar a la función de evaluación de *fitness.*

La activación de una red neuronal toma como entrada las variables obtenidas del entorno Pygame y las imprime en sus neuronas de entrada, y devuelve como salida una lista de probabilidades asociada a las neuronas de salida. De la lista de salida tomamos el índice de aquella con un mayor valor, y en base a ese índice enviamos una orden diferente al jugador asociado a la red mediante la función *move()*

La evaluación de *fitness* varía su asignación dependiendo del aspecto a premiar durante el entrenamiento, incluso incluyendo aspectos a castigar. Esta asignación se aplica a cada uno de los genomas que ha participado en el partido, por lo que durante una evaluación en enfrentamiento uno contra todos un mismo genoma recibe su puntuación P*2-2 veces, y durante una evaluación en solitario 2 veces, siendo P el tamaño de la población. Se debe tener en cuenta este caso a la hora de establecer el *fitness threshold* en nuestros archivos de configuración.

Testeo

Para finalizar, cuando el bucle de neuroevolución finaliza, se realiza el testeo. Para ello se importa la mejor red guardada, y se inicia un entorno de Pygame y un bucle eterno tal y como se ha explicado previamente en este apartado, con la única diferencia de que el segundo jugador será controlado por el ser humano, y no se involucrarán asignaciones de fitness.

6.3 Incremento de inputs progresivo

El incremento de inputs progresivo ha sido la modificación más útil para conseguir nuestro objetivo de entrenamiento. Fué desarrollada una vez iniciado el entrenamiento, tras descubrir que al reducir los inputs de la red el rendimiento del entrenamiento de esta era mucho mayor en problemas simplificados cómo entrar en contacto con la pelota.

Requirió de mucho tiempo para su desarrollo debido a que:

- No existía implementación alguna con esta modificación, así que se tuvo que consultar el código fuente de Neat y ir realizando avances poco a poco, con sus errores asociados.
- El algoritmo de Neat no está pensado para esta modificación, así que son muchos los elementos que interactúan con la clase genome y provocan errores ante este cambio.

Sin embargo, el resultado final es tremendamente satisfactorio, y sobre todo útil para una evolución progresiva.

Esta modificación es llamada justo al obtener la población inicial. Si la población se genera en ese momento, no hay problema ninguno, pués esta se generará según nuestro archivo de configuración con los inputs deseados. Pero si queremos conservar una población ya existente, como será en la mayoría de casos, deberemos aplicar la modificación para añadir los nuevos inputs.

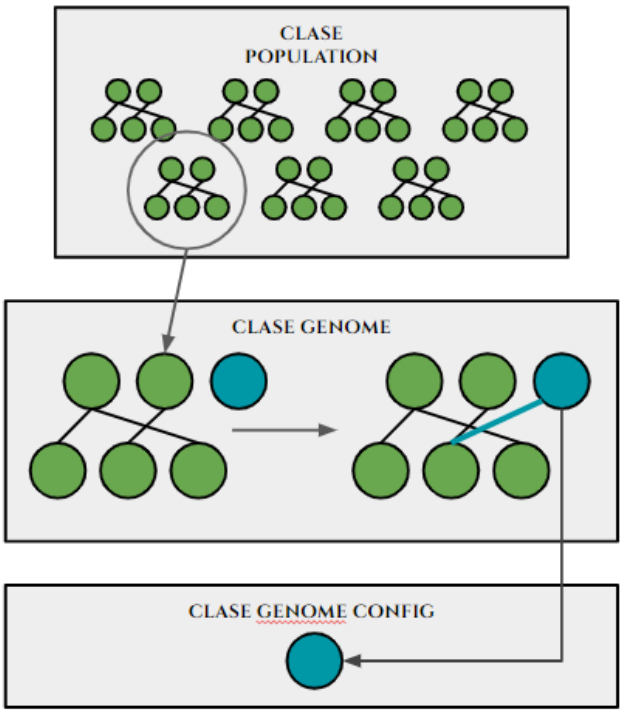

Figura 6.2: Funcionamiento simplificado de la inserción de nuevos inputs

En la imagen superior se explica de manera resumida cómo se añaden los nuevos inputs a una población: Se tomará cada genoma de la población. En ellos se añadirá un nuevo nodo/neurona con categoría *input node*. Se agregará una conexión entre el nuevo nodo y cualquiera de los otros nodos de las capas ocultas o de salida, evitando hacerlo con un nodo de entrada. Realizado este paso, en la clase genome.config se deberá almacenar la información del nuevo input añadido.

```
def add_new_inputs_to_config_and_genome(config, population, num_new_inputs):
    for i in range(num_new_inputs):
        max_node_id = 0
        for genome_id, genome in population.population.items():
            actual_node_id = config.genome_config.get_new_node_key(genome.nodes)
            config.genome_config.node_indexer = None
            if max_node_id < actual_node_id:
                max_node_id = actual_node_id
        config.genome_config.input_keys.append(max_node_id)
        config.genome_config.num_inputs = config.genome_config.num_inputs + 1
```

```
            print(f"Config has {config.genome_config.num_inputs} inputs now")
            for genome_id, genome in population.population.items():
                node_gene = genome.create_node(config.genome_config, max_node_id)
                others = [i for i in iterkeys(genome.nodes) if i not in config.input_keys]
                genome.create_connection(config, max_node_id, choice(others))
                genome.nodes[max_node_id] = node_gene
                print(f"Genome {genome_id} has a new input added")
        return
```

Código 6.4: Adición de nuevos inputs a la población

En el código superior se explica con más detalle la modificación. Para cada uno de los nuevos genes de neurona que queramos añadir a la capa de input, deberemos asignarle un identificador que no se corresponda con un gen ya existente en la población. Para ello recorreremos la población tomando un *id* no existente en cada uno de los genomas (Debido a como funciona el método *get_new_node_key(),* se devolverá un identificador no existente y mayor que cualquier otro en el genoma) y guardaremos el mayor de todos ellos: De esa forma nos aseguraremos de que no exista conflicto con ningún miembro de la población.

En el objeto de configuración añadiremos el identificador de esta nueva neurona, y aumentaremos el contador de número de *inputs.* De no realizarse este cambio se devolverá un error al no coincidir la estructura de la red y la configuración de la misma. Con el mismo objetivo de evitar errores deberemos eliminar el *node indexer* para obligar a Neat a recalcularlo cuando se pida un nuevo identificador de nodo, teniendo en cuenta este nuevo nodo.

De nuevo recorreremos los genomas de la población (tal y como se busca en el tercer bucle for), pero esta vez crearemos el nuevo nodo y una nueva conexión entre este y cualquier y cualquier otro nodo que no forme parte de los nodos de input. Se guardará el nodo dentro del diccionario de nodos, con el identificador que calculamos en el bucle previo.

6.4 Creación de especialistas

Recordamos que nuestro objetivo con esta modificación es representar distintos tipos de puntuación de "fitness" para poder preservar durante la reproducción a los individuos que se destaquen en alguna de estas áreas.

Esta propuesta mejora el algoritmo base de NEAT, ya que en el original el "fitness" de un genoma solo mide su rendimiento general, sin diferenciar si la puntuación proviene de goles, atajadas o control del balón. Por ejemplo, en una de nuestras configuraciones más recientes, el

"fitness" aumentaba según la cantidad de goles marcados y disminuía con los recibidos, pero finalmente se almacenaba como un valor entero simple.

Hemos decidido enfocar nuestra búsqueda en tres clases de jugadores, basándonos en tres tipos diferentes de "fitness":

- Defensores: Aquellos con el mejor "fitness" defensivo que les ayude a no encajar goles.
- Atacantes: Con el "fitness" de ataque más alto, que les permita marcar goles.
- Controladores: Jugadores con el mejor "fitness" de control, relacionado con tocar la pelota y evitar que el rival lo haga.

Personalizando la clase del Genoma

Deberemos crear una clase de genoma personalizada.

Se deberán inicializar los tres tipos de fitness como atributos de clase.

A la hora de realizar un “crossover” genético, es decir, cruzar el material genético de dos genomas, delimitaremos que tipo de “fitness” se debe tener en cuenta para priorizar. (Dado que, ante casos como un gen presente en solo un padre, se toma el código genético del padre con mejor “fitness”). Para ello, añadiremos un parametro a la función de crossover que indique el tipo de “especialista” del que se trata este genoma.

Se ha valorado que a la hora de categorizar el genoma en una especialidad, solo se tenga en cuenta el mayor fitness de los tres: Este es un enfoque erróneo. Primero, porque los valores no se miden bajo la misma escala (Siempre se marcarán menos goles que cuantas veces se toque el balón). Y segundo, porque aunque el genoma tuviera más puntuación como goleador, es posible que sea el mejor en el campo de la defensa comparado con el resto de sus compañeros. Nuestro objetivo, al final, es conservar al mejor de cada especialidad.

Personalizando la clase de la Reproducción

Deberemos crear una clase de reproducción personalizada.

Querremos que en cada generación haya al menos un miembro de cada especialidad. Además, deberemos hacer esto para cada una de las especies.

Se categoriza a los genomas de una población como una especie u otra para promover la diversidad genética, y además, proteger a los “futuros aspirantes” de un juicio precipitado. Es decir, cuando un genoma con suficiente diversidad respecto al resto “nace”, este se categoriza como una especie nueva. Esta especie no compite con los elementos de otra especie, para que así estos, que ya han disfrutado de varias iteraciones de testeo para optimizarse, le venzan y hagan desaparecer. (Aunque, sí una especie no ha mostrado mejora en su “fitness” después de varias generaciones, puede ser considerada para eliminación. Este estado, llamado “stagnation” o estancamiento, ayuda a enfocar los recursos evolutivos en especies que sí están progresando.)

Nos inspiraremos en cómo Neat conserva a las especies para modificar el script de reproducción base de Neat y conservar un individuo de cada especialidad por cada especie.

Nos hemos encontrado con que dos de nuestras ideas en realidad no tenían sentido para con la naturaleza de Neat, y creemos interesante explicarlas:

- Creímos que debíamos cerciorarnos de que hubiera individuos de las tres especialidades en cada especie, una vez se hubieran reproducido. Esto es un error de concepto, dado que una especie puede ser mas apta para crear individuos de una de las tres especialidades. Y de hecho, que eso sucediese sería una buena noticia, ya que se estarían adaptando estructuras genéticamente diversas para optimizar funciones diferentes dentro del equipo de fútbol.
- Inicialmente no comprobamos que un individuo ya hubiera sido elegido por su fitness en una especialidad distinta a la actualmente evaluada. Este caso es más que probable, ya que ser especialista en control no es detrimental para serlo en ataque o defensa, tal vez todo lo contrario. O citando al saber popular: “No hay mejor defensa que un buen ataque”.

De cada una de las especialidades, almacenaremos el fitness máximo y mínimo, además del medio. Esto es necesario para calcular el “adjusted fitness” y el “adjusted fitness” medio. Se calcula cómo el fitness bruto de la especie entre el número de individuos de la especie, y se emplea para penalizar a las especies con muchos individuos y evitar que dominen la población.

Se añadirán, además, *reporters* con todos estos valores, que nos servirán para monitorizar estos “fitness” en tiempo real.

Modificaremos cómo se calcula el cómputo del número de miembros de una especie que se deberán generar en la próxima población: De norma general, se calcula el tamaño como el máximo entre el tamaño mínimo estipulado en el archivo de configuración, o el fitness ajustado de cada especie respecto a la suma de los fitness multiplicado por el tamaño deseado de la población. Es decir, a mayor fitness respecto al resto de especies, mayor parte de la población se ocupará por esa especie. Nos aseguraremos de que el fitness comparado sea el del tipo deseado, además de que en total se generen un tercio de miembros para ataque, defensa y control respectivamente, por lo que cada cantidad deberá calcularse por separado y el resto del algoritmo deberá trabajar individualmente con estas cantidades.

Para cada grupo de miembros de una especialidad que “dar a luz” por especie, tendremos que realizar también modificaciones sustanciales sobre el algoritmo:

En primer lugar, se traspasan los “elites” de cada especie si el elitismo está activado. Para nuestro algoritmo, vamos a darle un nuevo significado al número indicado en la configuración bajo elitismo: De forma general, ese número se refiere al número de miembros de la especie a conservar inalterados durante el proceso de evolución. En nuestro nuevo algoritmo, el elitismo mostrará cuántos miembros de cada especie por cada especialista mantener. Es decir, con un elitismo de 3, 9 miembros serán conservados, 3 por cada especialidad. Estos miembros conservados tomarán la posición que otros descendientes habrían tomado.

Se comprobará que la especie se queda, para cada especialidad, con el mínimo entre su elitismo y el número de descendientes computados que hemos mencionado anteriormente.

Los miembros de cada especie se ordenarán en tres listas según su fitness de especialidad, iniciando por el menor fitness y acabando por el mayor. Si existe elitismo, se guardarán los mejores en la nueva población.

Al añadir descendencia, deberemos comprobar que el número de descendientes de la especie para una especialidad no se haya superado.

Para nuestro siguiente paso, debemos recordar la diferencia entre el elitismo y el "reproduction cutoff"/"survival threshold". El elitismo conserva íntegramente a ciertos individuos. El "reproduction cutoff" es el número de individuos que actuarán cómo padres y se cruzarán entre ellos para obtener descendencia. Los élites serán parte de esos padres.

Mientras no se haya llenado el cupo de nuevo miembros solo con los élites, se cruzarán a los mejores padres para generan nuevos hijos que conformen la nueva población. Para ello se almacenarán los X mejores padres de cada especialidad, siendo X el "reproduction cutoff". A la hora de cruzarlos, se escogerán los padres aleatoriamente de entre ese conjunto. Se realizará el crossover, que ya configuramos en la sección anterior, y además se añadirán las posibles mutaciones.

En la reproducción no propagaremos ninguna clase de etiqueta a los hijos referido a su especialidad. Por mucho que sus padres fueran los mejores delanteros, debemos dejar la posibilidad de que su cruce de lugar a un excelente centrocampista.

Con estos pasos, habremos acabado las modificaciones en la reproducción.

Personalizando la clase de la Población

Solo nos queda mantener a un jugador de cada especialidad, el mejor de ellos, al estilo de cómo en el algoritmo base se guarda el mejor individuo al acabar los bucles de evolución. Para ello, deberemos modificar cómo funciona la "run" de una población, es decir, el proceso mismo de entrenamiento:

En primer lugar, deberemos establecer tres diferentes límites de fitness a tomar desde el archivo de configuración, uno para cada fitness, ya que el valor deseado varía debido a la naturaleza de lo que representa cada fitness. Cien toques de diferencia con el rival no se miden igual que tres goles marcados. Estos límites de fitness, o *thresholds*, sirven para no seguir simulando partidos una vez se hayan alcanzado nuestras especificaciones deseadas. Esto implica, por tanto, que la simulación solo se detendrá en este caso si se alcanzan los tres límites de fitness.

A lo largo de cada generación simulada, almacenaremos los tres mejores individuos en cada una de las especialidades. Además, guardaremos a los tres mejores individuos de todas las generaciones simuladas.

Además , realizaremos la reproducción personalizada que mencionamos con anterioridad, eliminaremos las especies estancadas, y enviaremos la información necesaria a los *reporters* o reporteros, una para cada especialidad.

Modificaciones en el script de neuroevolución

Crearemos tres tipos de funciones de asignación de *fitness* para cada enfrentamiento de jugadores. Estas necesitan que la condición de finalización del partido sea un límite de tiempo y no un número de goles:

- De defensa, que aumenta por cada gol que no se haya recibido durante la duración del partido. Para calcularla se establece un valor de 100, extremadamente alto, que va disminuyendo con cada gol encajado por el equipo del individuo.
- De ataque, que es mayor cuantos más goles se marquen en un partido. Para calcularla, aumenta a 0 y se incrementa con cada gol marcado por el individuo.
- De control, que es mayor cuantas más veces se toque el balón y menos veces lo toque el rival a lo largo del partido. Para calcularla, se toman los toques a la pelota realizados por el individuo y se restan los realizados por el rival.

Se contempló establecer un *fitness* más concreto, pero al observar el comportamiento de los individuos descubrimos que esta complejidad añadida era detrimental, aquí unos casos:

- De defensa, aumentar el fitnes cuando el individuo tocara la pelota con una distancia inferior a 200 píxeles de la portería de su equipo, en lugar de disminuir el fitness al encajar goles. Esto favorecía que el individuo tocara la pelota cerca de su portería pero no que estas paradas no concluyeran en un gol en propia. Además, no tenía en cuenta los casos donde se defendía desde medio campo.
- De ataque, aumentar el fitnes cuando el individuo tocara la pelota con una distancia inferior a 400 píxeles de la portería rival y marcara. Buscando especializar al delantero, diferenciándolo de los goles que podía anotar el centrocampista o defensa, acabamos no recompensando la capacidad de marcar goles a gran distancia.
- De control, teniendo solo en cuenta los toques propios. (En este caso tuvimos que aumentar la complejidad.) Esta recompensa favorecía que el individuo se pusiera a rebotar la pelota contra la pared hasta que le robaran la pelota.

Al tomar el ganador del entrenamiento, tomaremos tres individuos en lugar de uno, los devueltos en la clase de Población, y los guardaremos como archivos .pickle.

6.4 Otras mejoras

Exploración de posiciones de pelota

Tal y como mencionamos en la sección de “Desarrollo de la evolución”, el diseño es sencillo y la implementación lo es también. Básicamente consiste en añadir una nueva anidación de bucle en los entrenamientos en solitario y de todos contra uno, dentro de la cual colocar la pelota en diferentes posiciones.

La utilidad de esta propuesta es exponer a la red a situaciones diferentes cuando la simulación no implica un movimiento continuo de la pelota: Por ejemplo, en las etapas tempranas del entrenamiento, la evaluación finalizaba nada más tocar el balón, así que no se

podía explorar si el jugador era capaz de moverse hacia la pelota si esta se situaba encima suyo o a su izquierda, en lugar de en el centro del escenario.

Evaluación con multitudes

Durante las fases tempranas del entrenamiento, una manera de agilizar la simulación ha sido incluir a varios individuos en cada equipo, dentro de un mismo entorno. La utilidad de este método se pierde una vez hay que interactuar con la pelota, ya que a ojos de una red neuronal la pelota se moverá de forma aleatoria por no disponer de inputs para percibir a los demás individuos.

Almacenaremos a los jugadores de cada equipo en las listas pertenecientes a su equipo una vez se inicializan desde el constructor de la clase ***Game*** del entorno Pygame. Todos los procesos explicados en el *script* de neuroevolución iterar sobre esta lista: Enviar los genomas al entrenamiento training_*ingame()* y asegurarnos de que los genomas testeados no repiten su entrenamiento innecesariamente, crear las redes neuronales correspondientes al número de miembros de cada equipo y asignar cada red en término de inputs y outputs a cada jugador.

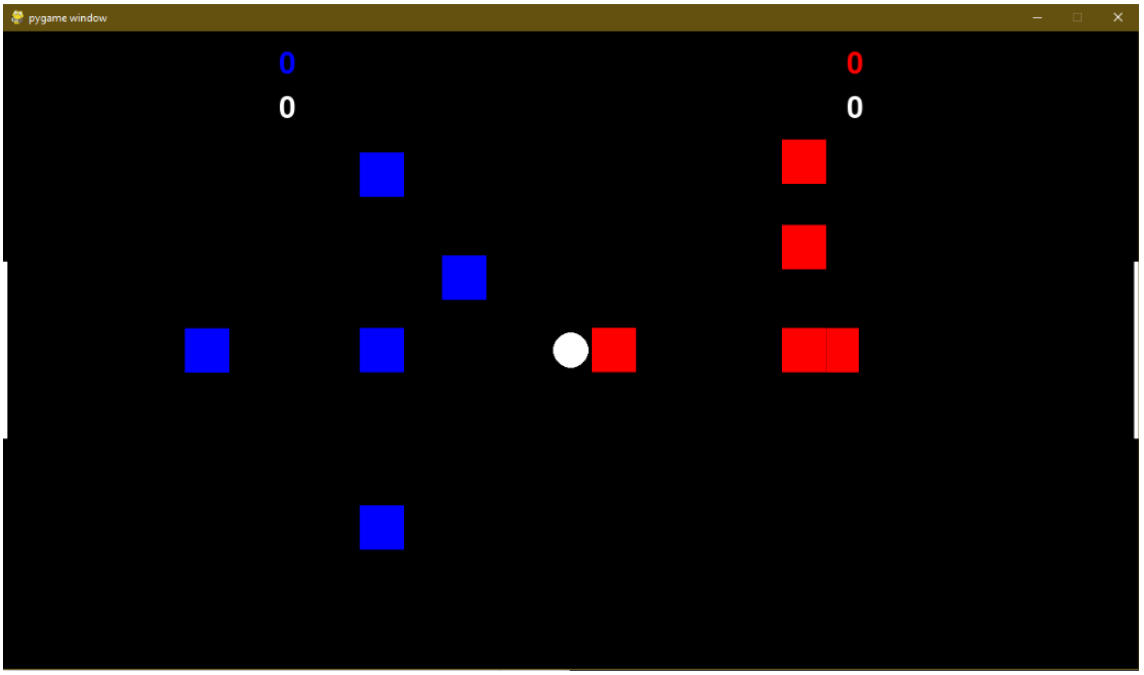

Figura 6.3: Evaluación dentro del entorno con varios genomas/jugadores

Esta modificación fue muy útil durante el entrenamiento de la red para aproximarse a chutar la pelota, ya que aceleró la exploración de solucione sin tener que iniciar y cerrar varios entornos de pygame. Se anularon las colisiones entre el jugador y la pelota, y entre los jugadores entre ellos en las clases ***Ball*** y ***Player***, de forma que cada uno de los jugadores que llegaba hasta la posición de la pelota se mantenía sobre la pelota y no evitaba que el resto de jugadores llegara a ella, sin hacerla rebotar. Estos jugadores que se superponían con la pelota sí llamaban a la función *ball_hit()* de nuestro entorno Pygame, de forma que obtenían un *fitness* favorable.

Fitness de aproximación

La implementación de esta mejora fue simple, únicamente se necesitó calcular la distancia en base al Teorema de Pitágoras y realizar una comparación entre la nueva distancia a pelota y la mejor hasta el momento sobre el bucle *while* eterno dentro de *ingame_training()*. En la función de asignación de *fitness* se otorgaba una mayor puntuación cuanto menor fuera esta distancia.

La utilidad de esta propuesta era solucionar el problema similar al "problema de la asignación de crédito" que experimentaba nuestra red, y al aplicarle permitió conseguir que el jugador se moviera correctamente hacia la pelota fuera cual fuera su posición.

6.6 Desarrollo del entrenamiento

En este subapartado narrararemos nuestro proceso de entrenamiento de forma resumida, centrándonos en la variedad de problemas, alternativas y soluciones que tomamos. Para entender ciertas modificaciones en las que no se profundizará, recomendamos leer los subapartados de desarrollo anteriores, dado que actúan como una base para este.

Tras incorporar cada modificación en el entrenamiento, se medía su eficacia empleando la fase de testeo del *script* de neuroevolución mencionada en el apartado de "Diseño de la solución". Enfrentar la IA contra un humano fué útil para determinar ciertos fallos, como su falta de exploración de posiciones de pelota en el entrenamiento individual.

Podemos afirmar sin dudas que esta parte del desarrollo fue la más problemática de todo el proyecto. Entre otros motivos, queremos destacar un problema asociado al entrenamiento de redes neuronales en general que menciona Andrej Karpathy, catedrático y miembro de OpenAI, en su blog [12]: "Las redes neuronales fallan de forma silenciosa". Esto quiere decir que, aunque no se cometan errores de redacción de código, la red puede no funcionar cómo esperamos debido a los miles de aspectos que influyen en su funcionamiento. Nuestro entrenamiento inició, y seguidamente fracasó, por ese mismo motivo.

Primeros entrenamientos fallidos

Durante el inicio del entrenamiento, aún adaptándonos a la utilización de nuestro script y retocando detalles, tomamos un acercamiento en el que las redes se enfrentaban entre ellas de manera aleatoria hasta que alguna de ellas marcara un gol o pasaran diez segundos. Como era de esperar, las primeras generaciones de neuronas apenas se movían o lo hacían en direcciones aleatorias. Con alrededor de un coste temporal de 3 minutos por generación, llegamos a la generación número cien sin apenas mejoras en nuestros resultados, tras cerca de cinco horas de entrenamiento. Con la generación 200 sin cambios fructuosos, fuimos conscientes de que había un problema que debíamos solucionar.

Incremento del tamaño de población

Buscamos inspiración en las implementaciones de Neat-Python presentes en su repositorio [0]. Observamos que en los ejemplos del "aterrizador lunar" y el "balanceo de barra", los más similares al nuestro debido a que requieren en la red una inteligencia espacial bidimensional, las poblaciones rondan entre los 100 y 250 elementos. Por ese motivo decidimos aumentar el tamaño de nuestra población a 250 miembros.Esto supuso un aumento del tiempo de entrenamiento a 42 minutos aproximadamente.

Sin embargo, tras varios días de simulación, apenas se observaba una mejora en el aprendizaje de la red. Futuramente se observaría que esta medida fue muy útil para aumentar la variedad genética de la población, ya que permitía realizar cruces genéticos en el proceso y generar nuevas especies.

Simplificación del problema

Ante la poca efectividad de este método, observamos que el número de inputs era inusualmente grande en comparación con las implementaciones mostradas en la sección previa. Se optó entonces por intentar resolver el problema con un menor número de inputs.

Originalmente se disponían de 10 inputs:

- Posición vertical y horizontal del jugador
- Posición vertical y horizontal de la pelota
- Posición vertical y horizontal del jugador enemigo
- Posición vertical y horizontal de la portería enemiga
- Posición vertical y horizontal de la portería aliada

Estos inputs poseían otros problemas a parte de su cantidad que luego fueron solucionados, como no incluir las velocidades de pelota y jugadores.

Conscientes de que no sería capaz de solucionar el problema completo, optamos por simplificar los inputs a los cuatro primeros para la menos conseguir que la IA buscara alcanzara la pelota.Para ello, la función de *fitness* pasó a bonificar a los jugadores que entraran en contacto con la pelota, y a terminar la simulación cuando esto sucediese.

Esta limitación de la duración del partido a un toque supuso a su vez una aceleración considerable del proceso de entrenamiento cuando los individuos conseguían alcanzar la pelota.

Ante una falta de eficacia, se optó por simplificar aun más los inputs a dos, que serían únicamente la distancia a la pelota desde el jugador, medida en los dos ejes. Este enfoque si dió resultados, y se pudo observar cómo la vigésima población ya mostraba signos de alcanzar la pelota en aproximadamente un 25% de sus miembros.

Solución del problema de la asignación de crédito

En el campo del aprendizaje por refuerzo es muy común experimentar un problema similar al nuestro, el llamado "problema de la asignación de crédito", en el que una política no recibe una recompensa adecuada debido a la dificultad de medir cómo de importante es para la resolución del problema. En nuestro caso, durante esta etapa del entrenamiento la red era incapaz de impactar con la pelota debido a la gran cantidad de espacio del estadio y lo poco que ocupaba esta, teniendo que probar movimientos en las cuatro direcciones hasta acertar, lo que sucedía en muy pocos momentos.

Para solucionar este problema se llevó a cabo la mejora de *fitness* de aproximación explicada en el apartado de "Diseño de la solución", con el que se premió la aproximación, y no solo el toque, del jugador a ala pelota.

Mejora de entrenamiento individual

Una complicación observada tras aplicar la solución anterior era que muchos individuos tocaban la pelota antes que otros que sí parecían tener una trayectoria que les llevaría a tocarla. Al hacerlo, provocaban que este individuo fuera descartado inmediatamente sin probar su validez.

Para paliar este problema se diseñó el entrenamiento en solitario mostrado en el apartado de "Diseño de la solución". Su resultado incrementó el número de individuos de la población capaces de aproximarse a la pelota a aproximadamente un 50%.

Como efecto secundario de parar los entrenamientos cuando la red alcanzaba la pelota, en la fase de testeo nos dimos cuenta de que esta solo podía alcanzar la pelota si estaba en el centro del escenario. Como solución a esto se desarrolló el método de exploración de posiciones de pelota expuesto en el "Diseño de la solución".

Además, tras acceder a la fase de testeo nos dimos cuenta de que la red estaba siendo entrenada desde solo el lado izquierdo. Para ello se modificó el script hasta tomar la forma expuesta en el apartado de diseño.

Mejora de entrenamiento enfrentado

Cuando la red pudo alcanzar aproximadamente un 75% de miembros exitosos en alcanzar la pelota con las anteriores modificaciones, el uso del entrenamiento enfrentado volvió a ser una buena aproximación para añadir competitividad: Muchos de esos individuos alcanzaban la pelota realizando movimientos poco eficaces, como espirales alrededor de ella hasta alcanzarla. La competitividad permitió que este defecto fuera superado, pues solo las redes más rápidas en tocar la pelota alcanzaban un buen *fitness*.

Sin embargo se observó como un enfrentamiento aleatorio resultaba injusto. Se daban situaciones donde un genoma codificaba una red con una adaptación buena, pero al enfrentarse

con la mejor red de la población, este apenas obtenía el *fitness* necesario para ser seleccionado. Además del caso de de que algunos genomas fueran seleccionados aleatoriamente más de una vez, recibiendo mucho más *fitness* que el resto.

Para solucionar esto se adaptó el método de entrenamiento enfrentado que se expone en el apartado de "Diseño de la Solución", observando cómo las redes resultantes se lanzaban a la pelota con decisión y sin realizar maniobras innecesarias. Uno de los problemas derivados de este tipo de evaluación es el coste temporal mucho mayor, que además escala de forma exponencial con el tamaño de la población. Por supuesto, la ventaja es que permitió simular un enfrentamiento real y mejorar radicalmente el desempeño de la red.

Entrenamiento con inputs progresivos

Disponíamos de una red capaz de perseguir la pelota se encontrara donde se encontrara, pero esta solución no le permitía meter goles de forma no aleatoria, dado que con sus dos inputs era incapaz de saber siquiera su posición en el mapa.

Para solucionar este problema se desarrolló la solución de los inputs progresivos. Dentro de esta misma sección de "Desarrollo de la Solución", en el apartado de "Incremento de inputs progresivos" explicamos los problemas asociados con su desarrollo. En este apartado nos centraremos únicamente en los grandes beneficios que supuso para el desarrollo.

Buscando conseguir el marcaje de gol, comenzamos añadiendo la posición de la pelota en los inputs, y a alterar la función de *fitness* para no sólo premiar un golpe en la pelota sino también el marcaje de un gol. El entrenamiento usado fue uno individual, ya que la red no podía tener en cuenta aún la posición del rival. Para hacerlo, la condición de cierre de la simulación ya no debía ser un solo golpe a la pelota, sino marcar un gol. Este último cambio supuso un aumento considerable en el tiempo de entrenamiento, pero fué exitoso y demostró la eficacia de esta modificación.

Tras 22 generaciones de simulación, viendo que los goles marcados no eran consistentes, se añadió como input la posición de la portería rival (en un inicio se creyó que la red sería capaz de inferir esta). Además se añadió como input el equipo propio para evitar los goles marcados en la propia portería. Tras esta mejora, finalmente la red alcanzó el marcaje de gol tras aproximadamente 40 simulaciones.

Aprendido el gol, se expuso a la red a un entrenamiento enfrentado. Dos nuevos inputs con la posición del rival fueron añadidos, pero no se consiguió un resultado aceptable tras 30 generaciones. Preveyendo que la red tal vez no estaba realizando las conexiones de pensamiento entre la posición del rival y la de la pelota para prevenir la trayectoria de su chute, se cambiaron los inputs de posición por unos *inputs* de distancia entre el rival y la pelota. Para realizar este cambio se tuvo que recuperar el *checkpoint* de treinta generaciones atrás donde la población de redes aún no contaba con los *inputs* de posición.

Para mejorar aún más los resultados, y consumando una red finalmente inteligente al nivel de un humano, se añadieron como inputs las velocidades de los jugadores y la pelota, una tras de otra en tres entrenamientos consecutivos. tras aproximadamente 100 generaciones, el resultado se pudo considerar como suficiente para los objetivos establecidos.

Entrenamiento de especialidades

Este fue el último entrenamiento realizado, aplicando los cambios de la funcionalidad de creación de especialistas. Aparte de la dificultad de desarrollar los cambios en el algoritmo, el entrenamiento no supuso mucha más dificultad.

Tras 30 generaciones de evolución se obtuvieron resultados aceptables como para observar algunas diferencias en las estrategias de los especialistas.

7. Pruebas

Durante el desarrollo del proyecto se realizaron pruebas durante y al final de cada *sprint* para comprobar que las tareas establecidas en la planificación se cumplían.

Al finalizar el desarrollo del entorno Pygame se pidió a un grupo de personas conocidas que compitieran entre ellos en varios partidos para así poder detectar fallos. Los fallos mecánicos mencionados en el capítulo de "Desarrollo de la solución" fueron solventados gracias a este testeo.

Específicamente se realizó una mayor cantidad de pruebas en la fase de entrenamiento, empleando la función de testeo del script de neuroevolución para enfrentar al propio alumno con la IA y ver sus resultados. La cantidad de partidos ganada contra el humano, desde que se consiguió entrenar una red que marcara goles hasta el modelo final, es de un 25% aproximadamente a un 75%.

En las últimas fases del modelo final se pidió ayuda de nuevo a un grupo de conocidos para testear su fidelidad. De los 20 partidos que se jugaron, la inteligencia artificial ganó 14. Con los modelos especialistas se realizó el mismo testeo, siendo el modelo centrado en el ataque el que obtuvo mejores resultados con 7 de 10 partidos ganados.

8. Conclusiones

Habilidades adquiridas

Haber cumplido este proyecto exitosamente ha supuesto una ampliación de los conocimientos del alumno en el campo del aprendizaje automático. Le ha permitido aumentar su experiencia práctica en las técnicas de neuroevolución y algunas técnicas de aprendizaje por refuerzo. En específico, realizar las mejoras sobre el algoritmo Neat le han permitido incrementar su entendimiento del funcionamiento interno del mismo. Por otro lado, el alumno ha dominado los aspectos básicos y medios de Pygame, lo cual le facilitará realizar simulaciones para otras tareas de este estilo. Además ha mejorado en su dominio de Python.

En otro aspecto, este proyecto ha afianzado su capacidad de organización de un proyecto empleando metodología ágil, su gestión del estrés, y ha supuesto un reto a la hora de realizar explicaciones entendibles del proyecto, afianzando sus habilidades blandas de cara al futuro laboral.

Objetivos cumplidos y no cumplidos

En referente a los objetivos propuestos en las etapas tempranas del desarrollo, se han cumplido todos los objetivos generales y la mayoría de objetivos específicos:

- Hemos programado un entorno de simulación de fútbol-sala con mecánicas realistas y sin errores graves. Sin embargo, si hemos sufrido un error que no hemos podido solventar rehaciendo la mecánica de colisiones, relacionado con la pelota atravesando al jugador cuando esta se lleva a una esquina y no tiene espacio de movimiento. (Pero creemos que este error no ha afectado de forma determinante al entrenamiento.)
- Hemos desarrollado un entorno de neuroevolución en Neat que ha permitido emplear diferentes técnicas de entrenamiento para nuestra red.
- Hemos entrenado una red con suficiente inteligencia como para competir con el humano. Sin embargo, el tiempo no nos ha permitido entrenar a la red a sortear obstáculos con sus tiros.
- Hemos diseñado técnicas que han mejorado el proceso de entrenamiento y han añadido nuevas funcionalidades. Sin embargo, querríamos haber dispuesto de un poco más de tiempo para entrenar durante más generaciones la red con las mejoras de especialistas, pese a que los resultados han sido suficientes.

Dificultades en el desarrollo

Durante la sección de “Desarrollo de la Solución” se presentan varios problemas acaecidos durante la elaboración de las implementaciones. Los más destacables de ellos han sido las

dificultades de bloqueos y retrasos, que han supuesto la reestructuración de la organización de los *sprints* y el retraso de este proyecto. Sin embargo, el resultado final ha conseguido estar a la altura de lo que el alumno esperó.

9. Relación con los estudios

Asignaturas relacionadas

Consideramos de notable mención las asignaturas de: **Percepción, Aprendizaje automático** y **Técnicas, entornos y aplicaciones de la inteligencia artificial,** estudiadas en la rama de Computación. En estas se trataron los conocimientos sobre aprendizaje máquina, redes neuronales y algoritmos genéticos que han resultado vitales para el desarrollo del proyecto. Obviamente, han resultado también fundamentales para los conocimientos básicos de programación las asignaturas de **Programación** e **Introducción a la programación.**

Además, para la gestión del tiempo y organización del proyecto han sido importantes las asignaturas de **Ingeniería del Software** y **Gestión de Proyectos.**

Competencias transversales

Las competencias transversales que más se han requerido y puesto en práctica en este trabajo han sido:

- Análisis y resolución de problemas: Sería la competencia más destacada. El proyecto ha sido una constante secuencia de problemas, observación, replanteamientos de las ideas y solución de los mismos.
- Creatividad, innovación y emprendimiento: Las mejoras y soluciones a problemas propuestas suponen, hasta lo que ha llegado a observar el alumno en la literatura actual, enfoques novedosos.
- Aprendizaje permanente: El trabajo ha supuesto una constante búsqueda de soluciones de los problemas surgidos, y si estos n o se podían resolver observando implementaciones ajenas, se han resuelto por uno mismo.
- Comprensión e integración: Ha sido relevante a la hora de organizar mentalmente los objetivos a lo largo del trabajo y replantearlos. Además a la hora de estructurar lógicamente la exposición de los elementos del proyecto.

10. Trabajos futuros

La primera línea de desarrollo de este proyecto es la aplicación directa de los conocimientos adquiridos a la investigación y desarrollo de proyectos de aprendizaje profundo, dada la reciente tendencia de emplear técnicas de neuroevolución en ellos.

También propondremos ciertas mejoras ciertas mejoras o añadidos que podrían haberse implementado si se contara con más tiempo de desarrollo, o en un trabajo :

- La continuación de un entrenamiento similar a este empleando técnicas de agentes inteligentes para entrenar equipos completos de individuos.
- Haber dedicado más tiempo al entrenamiento de las redes especialistas, y este objetivo puede combinarse muy adecuadamente con el anterior para crear un equipo entero de inteligencias artificiales especializadas.
- Otro objetivo sería probar el algoritmo de Hyper-Neat y su codificación indirecta para superar el problema asociado al gran número de *inputs* que resolvimos con el entrenamiento con *inputs* progresivos, y compararlo con los resultados actuales

Como caminos a no seguir, exponemos los siguientes:

- Si quisiéramos aumentar el realismo de las simulaciones, otros motores con herramientas más avanzadas serían una mejor opción que Pygame, donde cada nueva física o mecánica debe codificarse a mano.
- Si quisiéramos aumentar la cantidad de inputs dedicados a la red neuronal, Neat no resultaría tan buena solución como una solución que emplee visión por ordenador, ya que esta última sería mucho más eficaz para observar el entorno que añadir inputs por cada elemento nuevo.

Bibliografía

[1] Aaltonen, T. (2009). Measurement of the top quark mass with dilepton events selected using neuroevolution at CDF. *Phys. Rev. Lett.*

[2] Cheesy AI. (n.d.). https://youtube.com/@cheesyai?si=a6WJier_8PZhecZP

[3] Gardner, H. (2024, July 5). *La Teoría de las Inteligencias Múltiples de Gardner*. Psicología y Mente. Retrieved July, 2024, from https://psicologiaymente.com/inteligencia/teoria-inteligencias-multiples-gardner

[4] Ha, D., Dai, A., & Le, Q. (n.d.). Hypernetworks. *International Conference on Learning Representations*.

[5] Hitze, A., & Adami, C. (2021, Abril 13). Neuroevolution gives rise to more focused information transfer compared to backpropagation in recurrent neural networks. *2020 INDIA INTL. CONGRESS ON COMPUTATIONAL INTELLIGENCE*.

[6] Hornby, G. (n.d.). Evolving robust gaits with AIBO. *IEEE Conference on Robotics and Automation*.

[7] Improving the accuracy of the neuroevolution machine learning potential for multi-component systems. (2022, Enero 6). *Journal of Physics: Condensed Matter*.

[8] Lehman, J., Chen, J., Clune, J., & Stanley, K. (2018). Safe mutations for deep and recurrent neural networks through output gradients. *Proc. Genetic and Evolutionary Computation Conference*.

[9] McIntyre, A., Miguel, C. G., de Silva, F., & Lobo, M. (n.d.). *CodeReclaimers/neat-python: Python implementation of the NEAT neuroevolution algorithm*. GitHub. Retrieved 2024, from https://github.com/CodeReclaimers/neat-python

[10] Neat AI. (n.d.). https://youtube.com/@neatai6702?si=UwT7VKzSRggfuB-n

[11] *Neat-Python documentation*. (n.d.). NEAT-Python 0.92 documentation. Retrieved July, 2024, from https://neat-python.readthedocs.io/en/latest/

[12] Patric Feldmeier. (2023). Fully Automated Game Testing via Neuroevolution. *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*.

[12] *A Recipe for Training Neural Networks*. (2019, April 25). Andrej Karpathy blog. Retrieved August, 2024, from https://karpathy.github.io/2019/04/25/recipe/

[13] Salimans, T., Ho, J., Chen, X., Sidor, S., & Sutskever, I. (n.d.). *Evolution Strategies as a Scalable Alternative to Reinforcement Learning*.

[14] Stanley, K. O., & Miikkulainen, R. (n.d.). Efficient Evolution of Neural Network Topologies.

[15] Tech With Tim. (n.d.). https://youtube.com/@techwithtim?si=icdqdMGSCA-oMELF

[16] Voskoglou, C., Oentoro, A., & Guide, S. (2017, May 5). *What is the best programming language for Machine Learning?* Towards Data Science. Retrieved September 2, 2024, from https://towardsdatascience.com/what-is-the-best-programming-language-for-machine-learning-a745c156d6b7

[17] Wierstra, D. (2014). Natural Evolution Strategies. *Journal of Machine Learning Research*.

[18] Zheyong Fan. (2021, Septiembre 20). Neuroevolution machine learning potentials: Combining high accuracy and low cost in atomistic simulations and application to heat transport. *PHYSICAL REVIEW B*.

[19] Such, F., Madhavan, V., Conti, E., Lehman, J., Stanley, K. O., & Uber AI Labs. (n.d.). Deep Neuroevolution: Genetic Algorithms Are a Competitive Alternative for Training Deep Neural Networks for Reinforcement Learning.

[20] Stanley, K. O., Clune, J., Lehman, J., & Miikkulainen, R. (2019, January). Designing neural networks through neuroevolution. *Nature machine intelligence*.

12. Glosario

- **Algoritmos genéticos:** Métodos de optimización basados en la teoría de la evolución de Darwin, que utilizan mecanismos como la selección, cruzamiento y mutación para evolucionar soluciones a problemas.
- **Cruzamiento (*crossover*):** Técnica en la cual se combinan características de dos redes neuronales "padres" para generar una nueva red "hija", en un intento de heredar las mejores características de ambos.
- **Élites:** Individuos o redes neuronales con el mejor rendimiento en una población dentro de un algoritmo evolutivo. Las élites se preservan sin cambios en la siguiente generación para garantizar que las mejores soluciones no se pierdan por azar durante el proceso de selección, mutación o cruzamiento.
- **Entrenamiento:** Proceso por el que un modelo de aprendizaje automático evoluciona y mejora su resolución del problema asociado a él.
- **Especiación:** Proceso mediante el cual las redes neuronales evolucionan en subgrupos o especies dentro de un algoritmo evolutivo, promoviendo la diversidad y evitando la convergencia prematura a una única solución.
- **Fitness:** Medida que evalúa qué tan bien se desempeña una red neuronal en una tarea dada, utilizada para guiar la evolución en algoritmos evolutivos.
- **Gen:** En el contexto de la neuroevolución, un gen representa una unidad de información dentro de una red neuronal, como un peso sináptico, una conexión entre neuronas o una característica estructural. Se utiliza para definir los componentes que se pueden heredar o mutar en los algoritmos evolutivos.
- **Genoma:** Es el conjunto completo de genes que define a una red neuronal en un algoritmo evolutivo. El genoma contiene toda la información necesaria para construir y entrenar la red, como los pesos de las conexiones y la topología. Durante el proceso evolutivo, los genomas se recombinan y mutan para generar nuevas redes.
- **NEAT (*NeuroEvolution of Augmenting Topologies*):** Algoritmo de neuroevolución que optimiza tanto los pesos como la topología de las redes neuronales, permitiendo una evolución más eficiente.
- **Neuroevolución:** Proceso de evolución de redes neuronales mediante técnicas inspiradas en la evolución biológica, como la selección natural y la mutación, para optimizar su estructura y parámetros.
- **Mutación:** Modificación aleatoria de los parámetros o estructura de una red neuronal en el contexto de un algoritmo evolutivo para explorar nuevas soluciones.
- **Red / red neuronal:** Modelo computacional inspirado en el cerebro humano, compuesto por nodos (neuronas) organizados en capas. Estas redes aprenden a realizar tareas complejas ajustando conexiones (pesos) entre las neuronas mediante un proceso de entrenamiento.
- **Selección natural:** Proceso en el cual los mejores individuos (redes neuronales, en este caso) son seleccionados para reproducirse y pasar sus características a la siguiente generación.
- **Topología de red:** Estructura o disposición de las conexiones entre neuronas en una red neuronal, incluyendo el número de capas y la cantidad de neuronas en cada capa.

Apéndice A: Objetivos de desarrollo sostenible

En este apéndice realizaremos una reflexión sobre la relación del TFG con los ODS mas relacionados.

El 25 de septiembre de 2015, los líderes mundiales adoptaron un conjunto de objetivos globales para erradicar la pobreza, proteger el planeta y asegurar la prosperidad para todos como parte de una nueva agenda de desarrollo sostenible. Cada objetivo tiene metas específicas que deben alcanzarse en los próximos 15 años. En esta tabla se muestra el grado de relación del trabajo con los Objetivos de Desarrollo Sostenible (ODS):

Objetivos de Desarrollo Sostenibles	Alto	Medio	Bajo	No Procede
ODS 1. **Fin de la pobreza.**				X
ODS 2. **Hambre cero.**				X
ODS 3. **Salud y bienestar.**		X		
ODS 4. **Educación de calidad.**				X
ODS 5. **Igualdad de género.**				X
ODS 6. **Agua limpia y saneamiento.**				X
ODS 7. **Energía asequible y no contaminante.**				X
ODS 8. **Trabajo decente y crecimiento económico.**				X
ODS 9. **Industria, innovación e infraestructuras.**	X			
ODS 10. **Reducción de las desigualdades**.				X
ODS 11. **Ciudades y comunidades sostenibles**.				X
ODS 12. **Producción y consumo responsables.**				X
ODS 13. **Acción por el clima.**				X
ODS 14. **Vida submarina.**				X
ODS 15. **Vida de ecosistemas terrestres.**				X

ODS 16. **Paz, justicia e instituciones sólidas.**				**X**
ODS 17. **Alianzas para lograr objetivos.**				**X**

De los anteriores objetivos de desarrollo sostenibles mencionados, el proyecto relacionado está relacionado con:

- ODS 3. Salud y bienestar: El aprendizaje profundo ha demostrado en numerosas ocasiones sus aplicaciones en el campo de la medicina, la salud y el cuidado. Sus modelos se emplean, por ejemplo , en el diagnóstico de ciertas enfermedades por imágenes como el cáncer de piel, o la prevención de otras patologías antes de que ocurran con modelos de predicción, además de, por ejemplo, la ayuda a personas mayores en el uso de la tecnología con los modelos de lenguaje. Por ello consideramos que este proyecto de investigación puede contribuir al desarrollo de técnicas en el campo del aprendizaje automático que indirectamente mejoren la calidad de vida de los individuos.

- ODS 9. Industria, innovación e infraestructuras: En temas de innovación, el aprendizaje profundo supone una revolución tecnológica. Este proyecto está estrechamente vinculado a este campo, y por tanto, contribuye a dicha evolución. Por supuesto, las innovaciones presentadas en el proyecto pueden ser empleadas en la industria, cada vez más adoptada al uso de inteligencia artificial. Especialmente puede resultar útil en el campo de la IA aplicada a la robótica, donde el algoritmo NEAT usado ha demostrado grandes resultados.