UNIVERSIDAD AUTONOMA DE MADRID

ESCUELA POLITECNICA SUPERIOR

Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

Evolución de redes neuronales mediante topologías aumentadas

Félix Coto Guardo
Tutor: Jack Mario Mingo Postiglioni
Ponente: Miguel Ángel Mora Rincón

JUNIO 2019

Evolución de redes neuronales mediante topologías aumentadas

AUTOR: Félix Coto Guardo
TUTOR: Jack Mario Mingo Postiglioni
PONENTE: Miguel Ángel Mora Rincón

Dpto. Ingeniería Informática
Escuela Politécnica Superior
Universidad Autónoma de Madrid
Junio de 2019

Resumen

Este Trabajo Fin de Grado consiste en estudiar los sistemas de redes neuronales desde un punto de vista evolutivo. Estos sistemas son extremadamente flexibles, es decir, se adaptan correctamente a distintos problemas. El inconveniente principal ocurre a la hora de elegir la estructura de estos sistemas para cada problema. Por ello, existen diversos enfoques e ideas para tratar con dicho inconveniente, partiendo desde el más básico, que consistiría en "ensayo y error", esto es, ir probando manualmente distintas estructuras y parámetros de la red neuronal hasta dar con un resultado satisfactorio. Puede intuirse que realizar dicha tarea puede llegar a ser tediosa, dependiendo del tamaño del sistema. Es por ello por lo que aparecieron algoritmos para automatizar este proceso, siendo NEAT (NeuroEvolution of Augmenting Topologies) uno de los más conocidos, desarrollado en 2002 por Ken Stanley.

NEAT no es el primer intento de aproximarse a sistemas de redes desde el punto de vista evolutivo. Sin embargo, su popularidad se debe a que ofrece soluciones a problemas que sistemas más antiguos presentaban. Por ejemplo, uno de los objetivos de NEAT es garantizar que la estructura de las redes se mantenga lo más simple posible. Otro objetivo es evitar que una solución predomine, por medio de la especiación. Esto es, separar las redes en especies según su estructura, para permitir que el sistema explore varios tipos de posibles soluciones a la vez.

Además de las redes neuronales, NEAT utiliza otro elemento común del campo del Aprendizaje Automático, como son los algoritmos genéticos. La idea principal consiste en tratar a las redes neuronales como individuos, codificarlas como genes, y evaluarlas a través de dicho algoritmo genético. Las redes neuronales que arrojen mejores resultados tendrán más probabilidad de ir pasando su información genética a sus descendientes, hasta llegar a una red que se considere lo suficientemente buena para el problema planteado.

El contenido de este trabajo gira en torno al estudio e implementación de NEAT, así como la discusión de sus ventajas y desventajas, y, por último, la aplicación de NEAT en diversos problemas, para comprobar cómo se comporta y ver si realmente es un algoritmo efectivo frente a otras estrategias.

Palabras clave

Redes Neuronales, Algoritmo Genético, Fitness, Dataset, Aprendizaje Automático.

Abstract

This Bachelor Thesis consists of studying neural network systems from an evolutionary point of view. These systems are extremely flexible, that is, they adapt correctly to different kind of problems. The main problem that the neural networks present it is associated with having to find the best structure for each problem. Therefore, there are various approaches and ideas to deal with this problem, starting from the most basic idea, which would consist of "trial and error". That is, to manually test different structures and parameters of the neural network to find a satisfactory result for a given problem. It is obvious that performing this task can be tedious, depending on the size of the system. That is why new algorithms appeared to automate this process, being NEAT (NeuroEvolution of Augmenting Topologies) one of the most popular ones, developed in 2002 by Ken Stanley.

NEAT is not the first attempt to approach neural network systems from an evolutionary point of view. However, its popularity is due to the fact that it offers solutions to problems that older algorithms presented. For example, one of the objectives of NEAT is ensuring that the structure of the networks is kept as simple as possible. Another objective is to keep the population split in species, to allow the system to explore several types of possible solutions at the same time.

In addition to neural networks, NEAT uses another common element of the field of Machine Learning, such as genetic algorithms. The main idea is to treat neural networks as organisms, encode them in genes and evaluate them by means of genetic algorithms. Neural networks that offer better results will have more probability of passing on their genetic information to their descendants, until reaching a network that is considered optimum for the given problem.

The content of this work revolves around the study and implementation of NEAT, as well as about the discussion of its advantages and disadvantages, and, finally, the application of NEAT in various problems, to check how it behaves and see if it really is an effective algorithm.

Keywords

Neural networks, Genetic algorithm, Dataset, Machine Learning.

INDICE DE CONTENIDOS

INDICE DE FIGURAS

INDICE DE TABLAS

INDICE DE ECUACIONES

1 Introducción

1.1 Motivación

A lo largo de la carrera han aparecido repetidamente temas que giran en torno al Aprendizaje Automático, tales como sistemas de clasificación, algoritmos genéticos, redes neuronales, etc. En cuanto a las redes neuronales, se ha aprendido cómo funcionan de forma básica. También se ha visto que tanto los algoritmos genéticos como las redes son herramientas que pueden llegar a ser bastante potentes. La motivación de este TFG es aprovechar lo visto en esos temas, combinarlo y potenciarlo, para aprender con algo más de detalle elementos sobre dichas materias. Por otro lado, alejándolo del punto de vista personal, la motivación de querer estudiar a fondo NEAT (a través del artículo original donde se propone **[1]**) surge como respuesta a los problemas de las redes neuronales. Esto es, que pese a que es cierto que son sistemas que se adaptan a muchos tipos de problemas, cada problema requiere de una estructura en concreto, que puede llegar a ser tediosa de encontrar de forma manual. NEAT proporciona un enfoque evolutivo para automatizar ese proceso. Partiendo de redes neuronales simples, y a través de un algoritmo genético, dichas redes van evaluándose, y en función del rendimiento que ofrezcan ante un problema, sus genes tendrán más probabilidades de ser transmitidos a sus descendientes en la siguiente generación. A partir de cierta generación, aparecerán estructuras lo suficientemente buenas como para ser consideradas soluciones. Todo ello evita el paso de tener que buscar manualmente una estructura óptima para las redes.

1.2 Objetivos

Los objetivos del trabajo son los siguientes:

- Estudiar y entender el algoritmo evolutivo NEAT.
- Implementar una versión propia de NEAT. Como quiere probarse en distintos problemas, la implementación debe de ser genérica, para poder aplicarse sin esfuerzo si el problema cambia.
- Evaluar la implementación del algoritmo en problemas de diferentes dominios.
- Analizar, evaluar y discutir los resultados, con especial hincapié en el estudio de las ventajas que ofrece NEAT frente a otros sistemas clásicos, para finalmente obtener conclusiones sobre su potencia y utilidad

1.3 Organización de la memoria

La memoria consta de los siguientes capítulos:

- Introducción: Se comentará la motivación de realizar este proyecto, tanto de forma personal como desde un punto de vista más técnico, así como los objetivos que se esperan lograr.
- Estado del arte: Se repasarán algunos detalles relevantes sobre la hibridación de algoritmos evolutivos y redes neuronales.
- Diseño: En esta parte se mostrará el diseño seguido a la hora de realizar el proyecto, discutiendo y argumentando cada decisión tomada.
- Desarrollo: En esta parte se explicarán los tipos de problema que se tratarán de resolver, así como las características de cada uno, y lo que se espera del sistema a la hora de enfrentarse a ellos.
- Integración, pruebas y resultados: En esta sección se estudiarán los resultados arrojados para los problemas estudiados en la parte de desarrollo, y se razonará

el por qué esos resultados son los que son, así como posibles cambios o mejoras que realizar al sistema.

- Conclusiones y trabajo futuro: En la parte final de la memoria se discutirán las ventajas y desventajas del algoritmo propuesto en función de los resultados obtenidos, así como posibles ampliaciones o enfoques que pueden realizarse en este trabajo para ampliarlo.

2 Estado del arte

NEAT es un algoritmo evolutivo que ya lleva 17 años publicado, y, todavía sigue siendo relevante. Gracias a NEAT, han ido apareciendo otros algoritmos de carácter evolutivo, que, o bien son extensiones de NEAT más completas, o su base es completamente diferente y ofrecen otro tipo de características. En este apartado solo se mencionarán algoritmos del mismo ámbito que NEAT, es decir, que permitan evolucionar redes neuronales para evitar tener que diseñar sus estructuras manualmente.

2.1 Extensiones de NEAT

2.1.1 rtNEAT

rtNEAT, o *real-time NEAT* es una variación de NEAT que consiste en lo que su propio nombre indica. En lugar de evaluar individuos uno por uno, suministrando los datos de forma estática, se evaluán en un entorno en tiempo real, como puede ser un videojuego, y cada individuo deberá adaptarse al entorno en cada instante de tiempo, y actuar en consecuencia.
Fue planteado por el propio Stanley en 2003, sobre el videojuego llamado "NERO" **[4]**.
A lo largo del trabajo, se hablará de permitir evolucionar en tiempo real, idea similar a la propuesta en esta variante.

2.1.2 HyperNEAT

NEAT permite evolucionar tanto la topología de las redes neuronales, como los pesos de las conexiones. Por tanto, ha de buscar un equilibrio que produzca soluciones útiles. Es por ello, que aplicar NEAT sobre redes muy grandes puede llevar mucho tiempo.

HyperNEAT surge como solución a ese problema. Es una variante pensada para aplicarse en sistemas más grandes. Por ejemplo, una de las aplicaciones de HyperNEAT es la evolución de un sistema para controlar las 4 patas de un robot de forma coordinada **[7]**.

HyperNEAT se aprovecha de elementos geométricos para no tener que guardar una lista completa de nodos y conexiones, lo que lo hace más eficiente que NEAT en sistemas de gran tamaño. Entre dichos elementos, se encuentra el reconocimiento de patrones en la red, simetría, etc.

2.1.3 SharpNEAT

Otra variante de NEAT, ideada por Colin Green en 2004 **[5]**, en la que en esencia se estudia la posibilidad de podar las redes neuronales, buscando eliminar partes que no sean esenciales, conservando la simplicidad. Pese a que uno de los principios de NEAT es precisamente que se garantiza simplicidad, no siempre se encuentra la solución más simple de forma absoluta, y es por ello que sharpNEAT existe.

2.1.4 cgNEAT

Extensión orientada principalmente al campo de los videojuegos. En **[2]**, se habla acerca de cómo puede usarse esta variante de NEAT para desarrollar contenido de un videojuego, como gráficos y partículas, en función de las preferencias de los usuarios. Se pone a prueba sobre el juego "Galactic Arms Race". Los usuarios deben ir eliminando enemigos y en función de qué enemigos se eliminen, se recompensa con cierto contenido especial. cgNEAT busca generar ese contenido en tiempo real para lograr satisfacer al mayor

número de usuarios posibles, y evitar así que el contenido del juego sea estático e igual para todos.

2.1.5 Otras extensiones

Puede apreciarse que NEAT es un algoritmo bastante popular, cuyo núcleo se ha extendido en función del campo en el que se quiera aplicar. Aquí se han mencionado algunas de esas extensiones más relevantes, pero hay muchas más, como DeepNEAT, para trabajar en el ámbito del *Deep Learning*, o odNEAT **[3]**, que es una versión de NEAT descentralizada, en el que distintos individuos evolucionan de forma independiente y luego pueden compartir su información con el resto.

2.2 Otros algoritmos evolutivos

2.2.1 EANT

Evolutionary Acquisition of Neural Topologies, (EANT), es un algoritmo presentado en 2005 por Kassahun y Sommer, Está basado en NEAT, y también comparte varias de sus características principales, como partir de topologías simples. Sin embargo, la característica principal de EANT es que la codificación genética se realiza de forma mucho más compacta que NEAT, y no hace falta decodificar dicha información para evaluar una red.
En **[11]** se muestra como este algoritmo puede superar a NEAT en términos de rendimiento para los problemas presentados.

2.2.2 DXNN

Deus Ex Neural Network (DXNN), es un algoritmo publicado por Gene Sher en 2012. Utiliza una variante de los algoritmos genéticos tradicionales, conocida como algoritmos meméticos, optimizados para evitar que el proceso evolutivo no se estanque en máximos locales y explore otras soluciones.

2.2.3 CMA-HAGA

Covariance Matrix Adaptation with Hypervolume Sorted Adaptive Grid Algorithm (CMA-HAGA), es uno de los algoritmos evolutivos más modernos, publicado por Shanin Rostami en 2016. El objetivo de este algoritmo es generar redes neuronales que ofrezcan soluciones para múltiples objetivos que de por sí ofrecen problemas de compatibilidad entre ellos. Entre sus características, el algoritmo introduce la idea del atavismo, esto es, que haya genes de antepasados que para ciertas generaciones desaparecen, y después vuelven a aparecer en generaciones posteriores.

3 Diseño

3.1 Introducción

En esta sección se mostrará el proceso de diseño, de forma detallada, que se ha seguido para la realización del proyecto.

A grandes rasgos, podemos dividir el proyecto en tres principales secciones. Por un lado, tenemos la parte de la construcción de las redes neuronales. Hay una gran cantidad de formas en las que diseñar como van a funcionar dichas redes. Por otro lado, tenemos el algoritmo genético. El funcionamiento básico de dicho algoritmo es siempre el mismo: se instancia una población, se evalúan los individuos, se cruzan y se aplica mutación cuando sea correspondiente. Sin embargo, existen multitud de variantes que se pueden tener en cuenta, tales como, por ejemplo, aplicar elitismo, (los mejores individuos de cada generación se pasan a la siguiente directamente). Dicho algoritmo se usará para ir evolucionando las redes neuronales desde sus formas simples hasta otras más complejas, que encuentren mejores soluciones a cualquier problema planteado. Es aquí donde entra la tercera y última parte, que sería el diseño del problema en sí, es decir, cómo evolucionar estructuras de redes neuronales mediante computación evolutiva.

Por tanto, los requisitos que debe cumplir el proyecto serían los siguientes:

- Construcción de una red neuronal especificando el número de entradas y salidas que debe tener.
- La red evolucionada debe ser capaz de procesar correctamente los datos de entrada y devolver las salidas correspondientes, tal y como una red convencional lo haría.
- Se debe poder crear una población que represente un conjunto de dichas redes (los podemos denominar organismos o individuos).
- Dicha población se debe instanciar en el módulo que contenga el problema a resolver, y debe poner a prueba a cada individuo.
- Para evaluar un individuo, se debe establecer qué entradas y salidas tendrá cada red, y qué función de fitness se va a utilizar para medir lo bien que se comporta el individuo en el problema planteado.
- Una vez evaluada una generación de la población, se deben aplicar los operadores genéticos apropiados que decidan como cruzar cada individuo para generar descendientes en función de su rendimiento, y generar una red nueva a partir de las de sus progenitores.
- Se repiten los requisitos anteriores para cuantas generaciones se desee, hasta obtener individuos cuya red neuronal ofrezca una solución lo suficientemente buena para el problema en curso.

Luego, se puede observar, que hay tres módulos principales, el que contiene la información de una red, el que implementa el algoritmo genético, y otro que describe el problema a resolver e instancia la población. Una vez se tienen construidos los dos primeros, este diseño permite que, si se quiere resolver un problema, solo hay que establecer qué entradas y salidas van a tener las redes, y cómo se va a medir el fitness, y el proyecto funcionaría correctamente.

3.2 Redes neuronales

3.2.1 Resumen básico

Antes de comenzar a explicar cómo se van a construir las redes neuronales, se va a introducir de forma básica y rápida cómo funcionan las redes que se van a usar en este proyecto.

Se va a trabajar con perceptrones multicapa, es decir, son redes formadas por múltiples capas, donde las salidas de una capa solo pueden conectarse a las entradas de las capas más profundas. Un ejemplo de red sencilla se presenta en la figura 1.

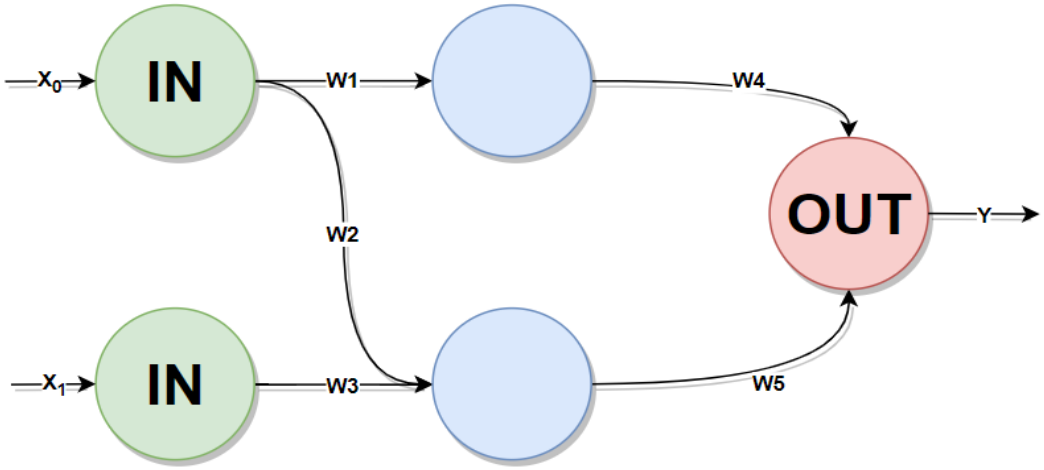

Figura 1. Ejemplo de perceptrón multicapa simple

Dicha red se compone de dos entradas, dos nodos ocultos internos, y una salida. A la hora de alimentar la red con datos de entrada, cada neurona recoge los datos y los pasa a las neuronas de la siguiente capa que estén conectadas a ella, multiplicando cada valor por el peso indicado en la conexión correspondiente. Por ejemplo, el valor que recibe la primera neurona de la capa oculta es $\mathbf{X_0W_1}$, y lo que recibe la segunda es $\mathbf{X_1W_3 + X_0W_2}$. Dicho valor, a su vez, se pasa por una función de activación. En este proyecto usamos la función sigmoide. Se elige esta función porque para cualquier valor de X la salida se encuentra en el intervalo [-1,1], y en la red neuronal, todos los valores, tanto entradas, como salidas, como pesos, funcionan en ese intervalo. Existen otras funciones que se usan en este ámbito, pero la sigmoide es una de las más empleadas. La función sigmoide se representa en la figura 2 para un intervalo específico.

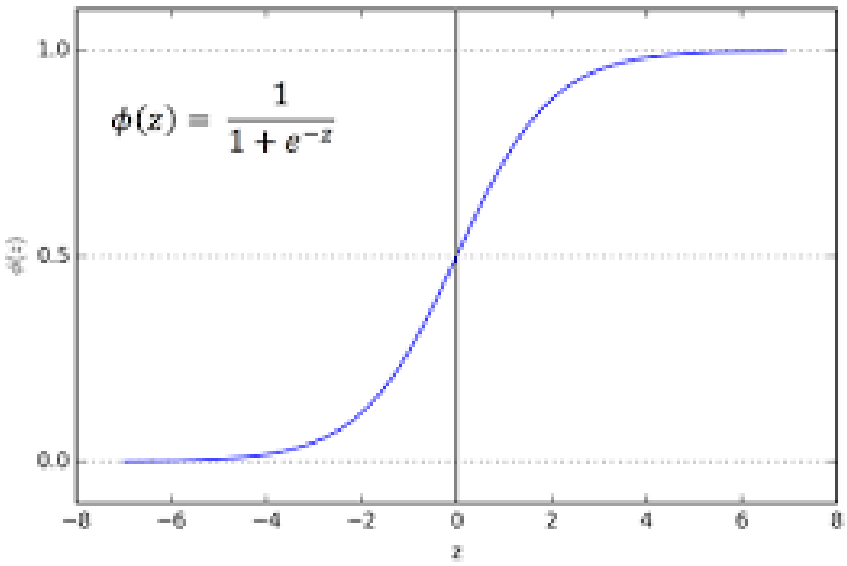

Figura 2. Función sigmoide

Una vez cada neurona de una capa pasa los valores correspondientes a la siguiente capa, y se incluyen en la función de activación, el resultado de dicha función es lo que las neuronas de esa capa pasan a la siguiente. Es decir, la salida de la red se calculará de la siguiente forma (Ecuación 1):

$$\Phi(w_4 * A + w_5 * B)$$

Ecuación 1. Función que representa la salida de la red de la figura 1

Donde A y B representan los valores de la capa anterior calculados según las ecuaciones 2 y 3:

$$A = \Phi(w_1 X_0)$$

Ecuación 2. Función auxiliar 1 de la ecuación 1

$$B = \Phi(w_2 X_0 + w_3 X_1)$$

Ecuación 3. Función auxiliar 2 de la ecuación 1

Con ello ya se tiene el resultado de la red neuronal. Ese resultado estará entre 0 y 1, y la interpretación del valor depende del problema a resolver.

3.2.2 Diseño de las redes neuronales para NEAT

El núcleo del proyecto reside en esta parte, ya que es lo que lo convierte en funcional. Como ya se ha mencionado, uno de los problemas de las redes es que no hay una forma exacta de establecer su tamaño y parámetros, si no que más bien se emplean otras técnicas. Una de las más habituales consiste en ir probando distintas estructuras, para después comparar resultados e ir descartando los modelos que peor se comportan. NEAT se pensó precisamente para evitar eso, de forma que las redes parten de estructuras muy simples y se van construyendo de forma automática a través del algoritmo genético.

Por tanto, el diseño de estas redes no tiene nada especialmente complicado ni que se salga de lo común en una red convencional. Simplemente hay que guardar cada nodo y cada conexión, con toda su información correspondiente (peso de la conexión, si está activa o no, etc.), para poder ser procesada por el algoritmo genético.

En el artículo original en el que se describe NEAT **[1]** se establece que inicialmente cada entrada de las redes está conectada a cada salida. Esto tiene una ventaja. Como la única forma de añadir elementos nuevos a las redes es por mutación, si no se conectan las entradas con las salidas, las primeras generaciones tendrían redes extremadamente simples, sin conexiones, que no harían nada. Si las entradas están conectadas desde el primer momento, el algoritmo se ahorra tener que esperar a que en esas generaciones iniciales aparezcan conexiones por mutación. Sin embargo, puede que no siempre interese esa situación. Dependiendo del problema, puede que haya entradas que inicialmente parecía que serían de utilidad pero que luego resultan ser inútiles. Por tanto, no interesa tenerlas conectadas, y en este proyecto, se propone, como modificación de NEAT, un punto intermedio entre tener todas las entradas conectadas a todas las salidas, y no tener conexiones. Dicha modificación consiste en crear una conexión entre una entrada y una salida aleatorias, para cada red, en la población inicial. Naturalmente, habrá entradas que aporten más información que otras, y los individuos que hagan uso de dichas entradas tendrán cierta ventaja evolutiva frente a los que no, luego así se consigue priorizar solo las entradas que de verdad aportan información, y se deja al algoritmo descubrir qué entradas son las que más importan. El problema es que a cambio, las primeras generaciones son algo más lentas, porque hay que esperar a que aparezcan nuevas conexiones por mutación.
Esta modificación es especialmente útil en dos escenarios. Si la red tiene muchas entradas, puede que no se necesiten todas, y así se pueden ignorar dichas entradas. Por otro lado, si la solución a un problema se puede encontrar con redes muy simples, puede que no se necesite tener cada entrada conectada a las salidas, y así la red se ahorra conexiones, y queda simplificada. Los problemas planteados en este proyecto se pueden agrupar bajo el segundo escenario descrito. Por tanto, se ha decidido seguir este diseño, y no el original de NEAT.

3.2.2.1 Nodos

La información que un nodo contiene es un índice que indica la capa en la que se encuentra (el índice de las entradas tiene un valor 0, los nodos de la siguiente capa un 1, y así sucesivamente), así como las conexiones salientes, para el procesado de los datos de entrada. NEAT no necesita saber la capa en la que se encuentra cada nodo, pero para la implementación realizada en este proyecto se pensó que sería útil, a la hora de hacer funcionar la red para producir una salida.

La información que lleva cada nodo, por tanto, es la siguiente:

- Índice de la capa en la que se encuentra (Por decisión de diseño propia).
- Listado de conexiones salientes
- Identificador del nodo

3.2.2.2 Conexiones

Las conexiones contienen más información. Necesariamente llevan el peso de dicha conexión, así como los nodos que conecta. Interesa decidir qué conexiones son útiles y cuáles no, así que las conexiones adicionalmente llevan un parámetro que indica si están activadas o no. Puede que se tenga un individuo que tiene una estructura casi perfecta para

la resolución del problema, pero hay alguna conexión que interfiere negativamente con los resultados finales. Si ese individuo muta y por azar desactiva la conexión negativa, de repente se convierte en una nueva solución interesante. Adicionalmente, cada red debe llevar un histórico de las conexiones que se han ido creando. Esto es necesario para realizar el cruce de forma efectiva, y es una de las razones por las cuales NEAT tiene gran popularidad y sigue siendo relevante hoy en día.

Vamos a poner un ejemplo de dos redes que contienen la misma información, y, por tanto, representan lo mismo (Figura 3):

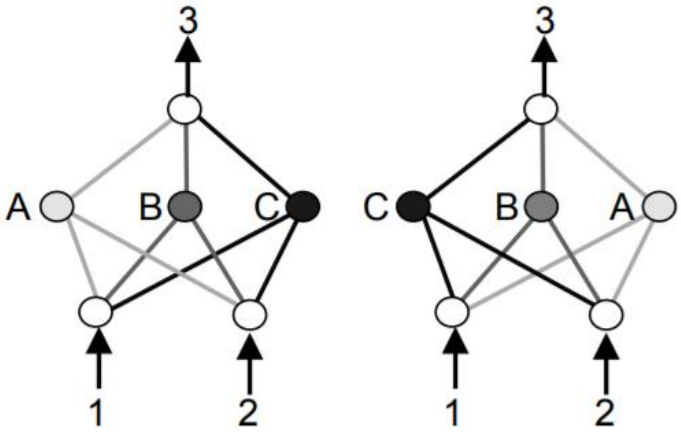

Figura 3. Comparación de la estructura de dos redes similares [1]

Los nodos y conexiones de dichas redes no tienen por qué haberse generado en el mismo orden. Por ejemplo, se puede suponer perfectamente que los nodos de la primera red se han creado en el orden [A, B, C] y los de la segunda en el orden [C, B, A]. Ahora se quiere cruzar dichas redes para generar nuevos individuos. Se podría aplicar cruce en un solo punto, es decir, elegir un punto por el cual se pueden generar dos descendientes nuevos, uno con la información genética de una red hasta antes de dicho punto, junto con la información de la segunda red después del punto, y otro descendiente con los genes restantes (Figura 4).

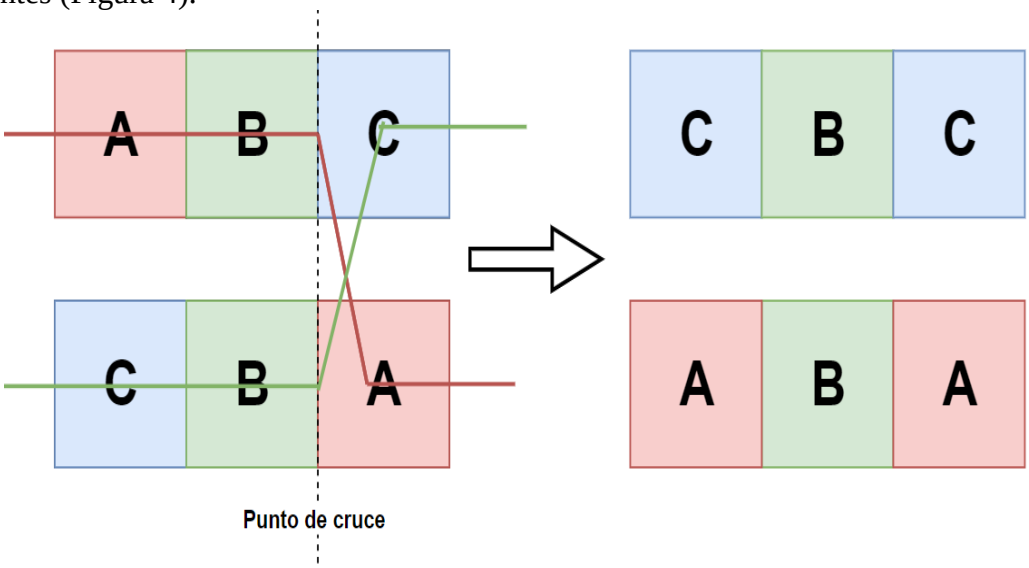

Figura 4. Ejemplo de cruce en un punto

Luego, los dos descendientes nuevos en este ejemplo podrían llevar los genes [C, B, C] y [A, B, A]. Como puede verse, se ha perdido información, una red ha perdido el nodo C, y otra el nodo A.

El problema es que distintas estructuras pueden representar la misma red. Por tanto, se necesita una forma de eliminar esa ambigüedad, y hacer saber al sistema qué genes son equivalentes. Si la población registrara un histórico de los genes añadidos a un genoma, podría asignar a cada gen un identificador, de manera que cuando aparece un nuevo gen, por la mutación de otro genoma, se puede buscar en el histórico si ya ha habido un gen igual o no, y asignar el identificador correspondiente. En NEAT son las conexiones las que llevan el identificador, y se denomina número de innovación. El mecanismo para implementar el histórico es relativamente simple. Se inicializa un contador a 1 en la población, y se genera una lista de conexiones vacía. Cada vez que se añade una conexión a un genoma, se mira en dicha lista si existe una conexión igual, es decir, que conecte los mismos dos nodos (el peso no se tiene en cuenta). Si no existe, se entiende que es la primera vez que aparece una conexión así, se le asigna como identificador el valor del contador, y dicho contador se incrementa. Adicionalmente, la nueva conexión se añade a la lista. Si, por el contrario, en la lista existe una conexión igual, implica que en el pasado ya existió una conexión similar. Por tanto, se le asigna como identificador el de la conexión que era igual en la lista, y el contador no se modifica. De esta forma se puede llevar un histórico de todos los tipos de conexiones que van apareciendo, y cada conexión tiene un identificador único. A la hora de realizar el cruce, se comparan las conexiones cuyo identificador coincida, en lugar de cruzar a ciegas. Los detalles del cruce se explican posteriormente en esta sección de diseño del algoritmo genético.

La información que contienen las conexiones queda resumida de la siguiente forma:

- Identificador de la conexión (número de innovación).
- Nodo de entrada
- Nodo de salida
- Peso
- Estado de activación (indica si la conexión esta activada o no).

3.3 Algoritmo genético

A diferencia de las redes, que no tienen diseño propiamente dicho, precisamente porque el objetivo de NEAT es que se construyan automáticamente, el algoritmo genético sí que tiene importantes decisiones de diseño a tener en cuenta.

3.3.1 Resumen básico

En esencia, un algoritmo genético evalúa la población, selecciona progenitores en función de su rendimiento, los cruza y puede aplicar una mutación que cambie los genes de un hijo resultante. Con dichos hijos se construye la nueva población y se itera durante tantas generaciones como se desee, o hasta que se encuentre una solución lo suficientemente buena.

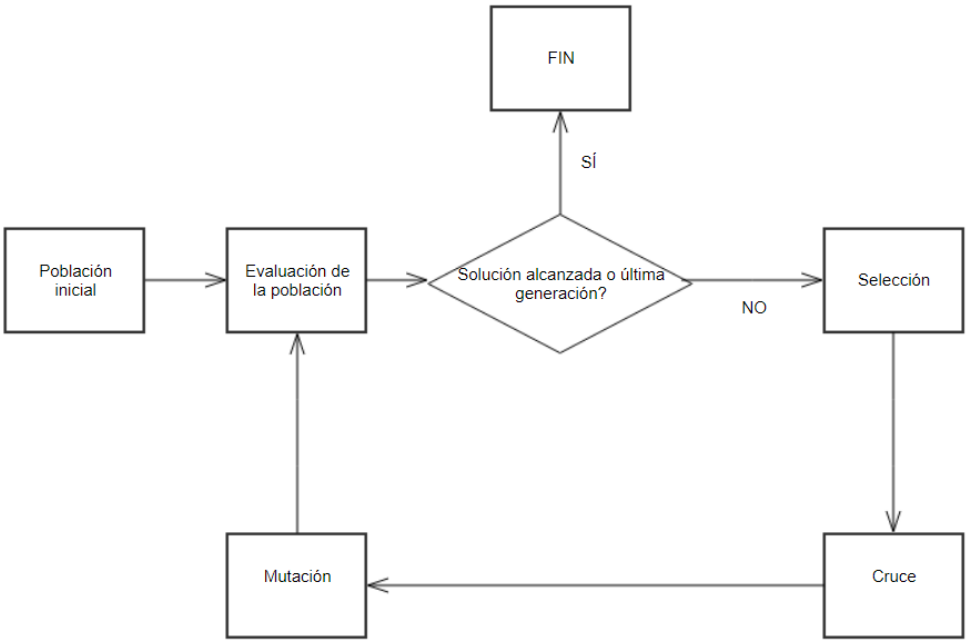

Figura 5. Esquema básico de un algoritmo genético

En la figura 5 se muestra el esquema básico de un algoritmo genético. Sin embargo, hay infinidad de variantes y ajustes que hacen que pueda funcionar mejor o peor en función de la codificación utilizada, los operadores genéticos considerados o del problema al que se enfrente.

3.3.2 Diseño del algoritmo genético para NEAT

Hay que establecer, por tanto, como codificar los genes de un individuo para guardar la información de la red y que pueda ser usada por el algoritmo genético. NEAT indica que la red se puede guardar como un listado de nodos y de conexiones.

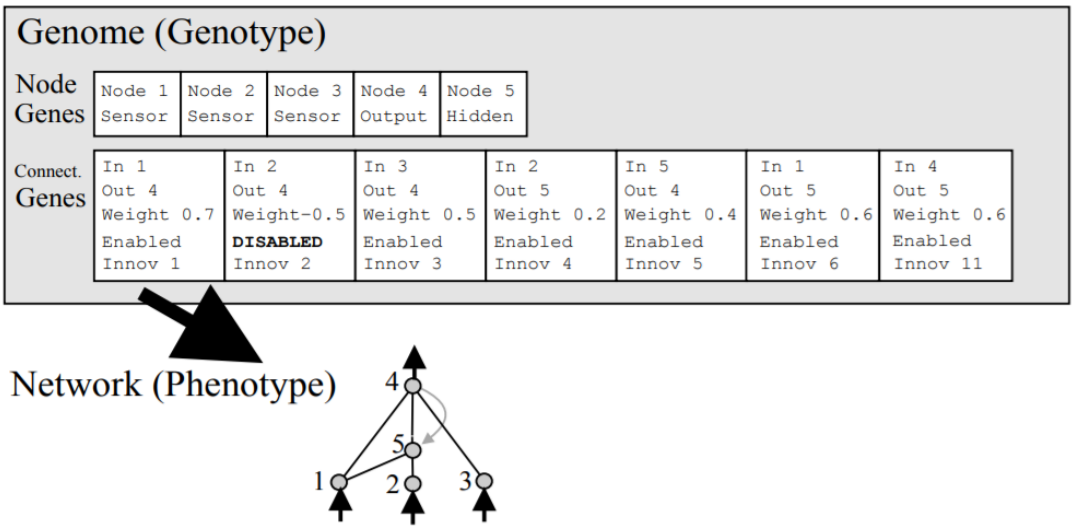

Figura 6. Codificación genética de la red neuronal [1]

Como puede apreciarse en la figura 6, podemos denominar como "genoma" al conjunto del listado de nodos y conexiones de la red, donde cada elemento lleva su información correspondiente. A partir de este esquema podemos construir la red.

Para crear el algoritmo genético de este proyecto hay que establecer como realizar la selección, el cruce y la mutación. Como se ha mencionado en la introducción de este apartado, el método de evaluación está separado del módulo que implementa el algoritmo genético y se hace en el módulo que describe el problema a resolver, para que cada problema exprese por sí mismo su forma de obtener el rendimiento de un individuo y el algoritmo sea flexible.

3.3.2.1 Selección

En un algoritmo genético la selección de dos individuos para el cruce se hace de forma proporcional a su rendimiento, esto es, los individuos más eficientes tienen más probabilidades de ser elegidos. Por ejemplo, si se tuviera una población con los siguientes 4 individuos:

Fitness Individuo 1	Fitness Individuo 2	Fitness Individuo 3	Fitness Individuo 4
20	40	10	30

Tabla 1. Ejemplo simple del fitness de una población

El primer individuo tendría un 20/(20+30+10+40) = 20% de probabilidades de ser seleccionado, el segundo un 40%, etc.

Como la población es de tamaño 4, se tienen que seleccionar individuos para generar 4 nuevos descendientes, de manera que los individuos más eficientes tienen más probabilidad de seleccionarse un mayor número de veces y, por tanto, según pasan las generaciones el fitness medio de la población debería ir creciendo. El problema es que cuando aparece una solución dominante es muy difícil que otras soluciones, que al principio son peores, prosperen. Dichas soluciones podrían ser mejores a la larga, y sin un mecanismo de protección, la población se acabaría estancando, con todos los individuos siendo similares entre ellos.

NEAT tiene esto en cuenta y lo soluciona por medio de la especiación, una técnica que consiste en separar la población en especies antes de hacer la selección de individuos. Los organismos de una especie se parecen entre sí, y ofrecen soluciones similares. El cruce solo se podrá realizar entre organismos de la misma especie. Antes de hacer el cruce, a cada especie se le asigna un número de hijos que puede generar según su fitness medio. De esta forma, la especie dominante será la que predomine y la que más hijos tenga, pero no por ello se eliminan individuos de otras especies. La población, por tanto, cuenta con un conjunto de especies entre las que se distribuyen los individuos.

3.3.2.1.1 Especies

Las especies son un conjunto de individuos que son similares entre sí. Hay que definir, por tanto, que significa "parecerse a", y como se decide si un individuo pertenece a una especie o no.

Intuitivamente, se puede entender que dos redes son similares si los pesos son parecidos y la estructura también lo es. El artículo original de NEAT **[1]** menciona esta circunstancia, y

ofrece una función que produce la similitud entre dos redes en función de su estructura y de sus pesos (Ecuación 4).

$$\delta = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 \cdot \overline{W}.$$

Ecuación 4. Cálculo de la similitud de los individuos

E representa el número de conexiones de exceso, es decir, las conexiones que una red nunca ha tenido con respecto a otra que sí, y ***D*** representa las conexiones diferentes que una red tiene y otra no, pero que las tuvo en algún momento. Diferenciar entre este tipo de conexiones es algo complicado y a efectos prácticos no hay muchas aportaciones al hacerlo, así que en este proyecto se suman ambos valores, y se plantea la similitud de la siguiente forma: cuando se comparan dos redes, se cuentan las conexiones que una red tiene y otra no, y no se considera diferencia entre tipos de conexiones. Por ejemplo, en la figura 7 dicho valor sería 5:

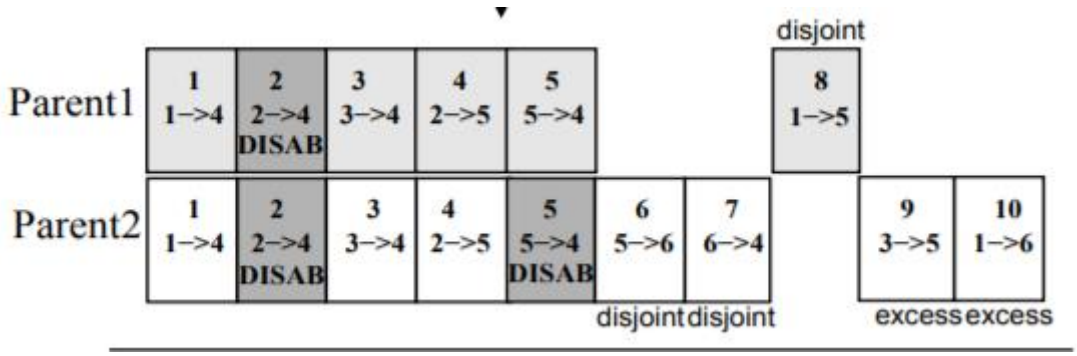

Figura 7. Ejemplo de genes de dos progenitores [1]

Continuando con la ecuación 4**,** ***N*** es el número de conexiones de las redes. En esencia sirve para normalizar redes grandes, de más de 20 conexiones. Para redes más pequeñas, como las tratadas en este proyecto, puede tomarse ***N = 1*** directamente [1].

$\overline{\mathbf{W}}$ es la media de diferencia de pesos entre las conexiones. Para calcularlo, hay que recorrer las conexiones de las dos redes, y para las conexiones que sean iguales, se calcula la diferencia entre los dos pesos, obteniendo entonces la media final. Como puede apreciarse, el hecho de que cada conexión lleve un identificador que indica cuándo se creó, resulta ser muy útil para implementar fácilmente estos pasos.

Los parámetros ***c1, c2, y c3*** son coeficientes que se deben ir alterando según el problema. En esencia indican la importancia del parámetro al que multiplican. Por ejemplo, si interesa considerar que dos redes son distintas, especialmente por sus pesos, entonces el parámetro ***c3*** tiene que tomar un valor mayor que los otros dos.

Como las variables ***E*** y ***D*** se han unido respecto al trabajo original propuesto para NEAT, hay que ajustar la fórmula final, que representa la distancia entre dos organismos en este proyecto (Ecuación 5):

$$\delta = (c1 * Q)/N + c3 * W$$

Ecuación 5. Cálculo de la similitud entre individuos, segunda aproximación

Donde ***Q*** es la suma de conexiones diferentes, es decir, ***E + D***. Puede apreciarse que aquí también se consideran solo los coeficientes ***c1*** y ***c3***.

Con toda esta información, ya se puede separar la población en especies. Antes de hacer la selección, se crea una especie inicial con el primer individuo de la población. Después, se va comparando el resto de los individuos con él. Si son lo suficientemente parecidos, esto es, el valor de la distancia o similitud no excede un umbral especificado, ambos se consideran de la misma especie y el individuo comparado se añade a dicha especie ya existente. Si no, se crea una nueva con dicho individuo como sujeto principal. Por tanto, cada especie tiene un sujeto principal que sirve para ser comparado con el resto.

Ahora ya se puede hacer la selección de progenitores. En el apartado de selección se ha indicado que las especies con un fitness medio mayor tienen derecho a producir más hijos para la siguiente generación, pero no se ha especificado cómo calcular cuántos hijos. Para ello, simplemente se suman todos los fitness medios de todas las especies, y se divide el fitness medio de la especie dada entre esa suma. Así se obtiene un valor proporcional a lo eficientes que son sus individuos, que indica cuántos individuos puede generar dicha especie.

Una opción que NEAT no considera es guardar el mejor individuo de cada especie directamente, y pasarlo a la siguiente generación, lo que se denomina habitualmente elitismo. Esta posibilidad es mejor o peor dependiendo del problema. En poblaciones pequeñas para problemas simples, quizás es mejor no guardar el campeón de cada especie, porque se estaría fomentando que no hubiera demasiada diversidad, ya que ese campeón acabaría pasando sus genes a la larga. Sin embargo, en poblaciones grandes, sí que interesa conservarlo, pues puede que el campeón proponga una solución interesante que si no se conserva se pueda perder.

Por último, en cuanto a especies, se tiene en cuenta un parámetro que indica si una especie se está quedando estancada. La idea de la especiación es precisamente ofrecer variedad en las soluciones, pero si una especie no crece en un número determinado de generaciones, se está quedando estancada y es poco probable que vaya a ser útil, así que se elimina. Se considera que una especie no ha crecido en una generación si el mejor de sus individuos sigue siendo el mismo que el de la generación anterior, es decir, el fitness máximo de la especie no ha cambiado.

3.3.2.2 Cruce

Una vez se tienen los individuos seleccionados para el cruce, hay que determinar una forma de cruzar sus genes, teniendo en cuenta la codificación genética de la red. La forma de cruzar en NEAT es la siguiente:

- El descendiente tiene directamente los mismos nodos que el progenitor con fitness mayor.
- El descendiente tiene directamente las mismas conexiones sobrantes que tiene el progenitor con mayor fitness. Esto es, las conexiones que el segundo progenitor no tiene.

- Para cada conexión que ambos progenitores compartan, se elige de forma aleatoria cuál tendrá el descendiente.

Puede observarse que las conexiones que el segundo progenitor tenga y que el primero no tiene se pierden. A la hora de cruzar, en el algoritmo genético existe un parámetro que indica la probabilidad de que dos individuos seleccionados produzcan nuevos descendientes. Si a la hora del cruce, por probabilidad se decide que no van a cruzarse, se pasa como hijo directamente el progenitor con mayor fitness.

Si se diera la casualidad de que al cruzar dos padres, ambos tienen el mismo *fitness*, no se descarta nada, es decir, se añaden todos los nodos y conexiones que tengan ambos. Un ejemplo de ello puede encontrarse en la figura 8.

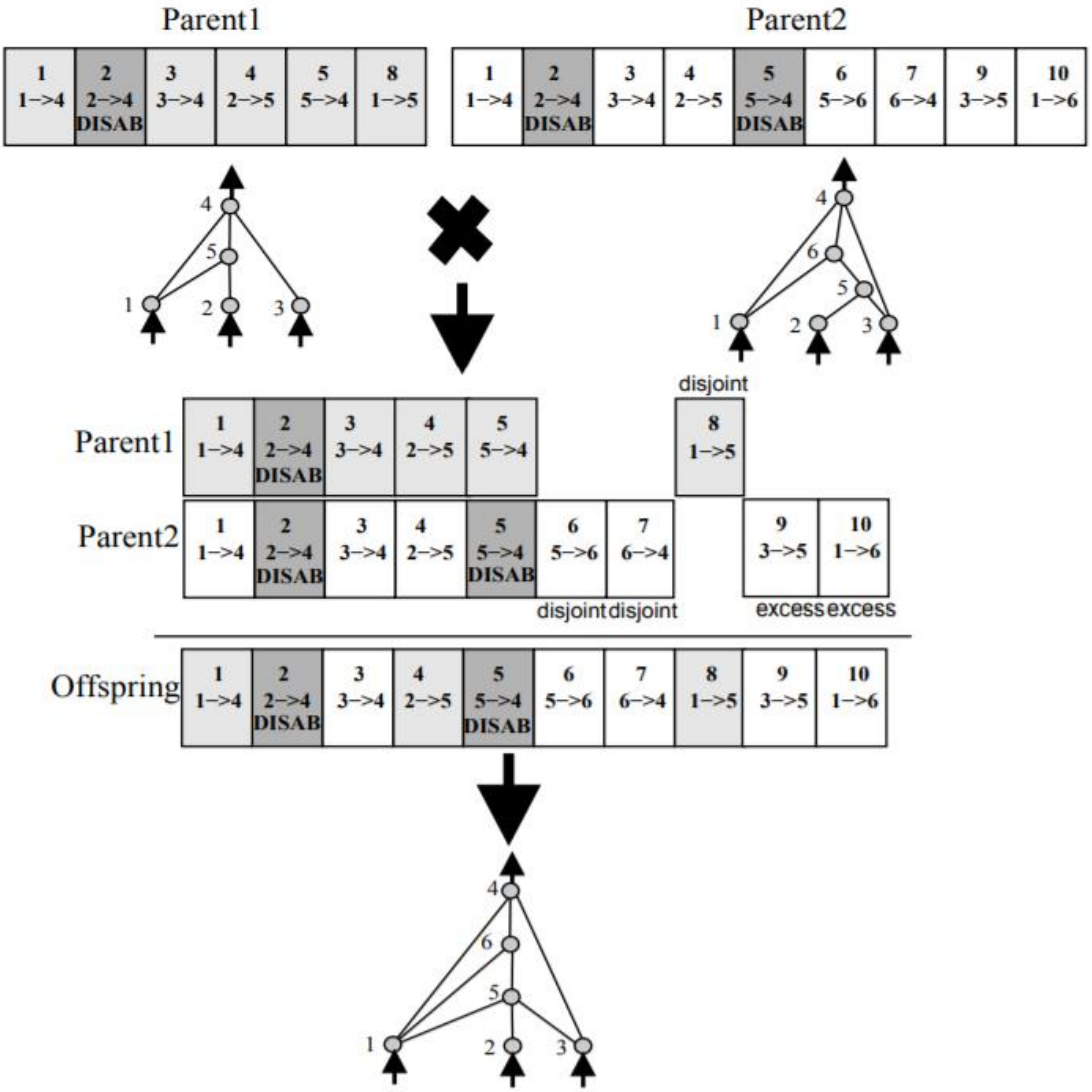

Figura 8. Cruce de dos individuos [1]

3.3.2.3 Mutación

Si el algoritmo genético solo se sirviera del cruce para crear nuevas soluciones, la variedad sería limitada. La mutación previene eso. Después del cruce, a cada individuo se le da una

pequeña probabilidad de mutar y de cambiar sus genes. Teniendo en cuenta la codificación genética de las redes, NEAT tiene en cuenta distintos tipos de mutación.

3.3.2.3.1 Mutación de nodos

Añade un nodo nuevo a la red. Debería ser la mutación más extraña, pues es la que más afecta a la estructura de la red. Para añadir un nodo nuevo se elige una conexión existente, se desactiva y se coloca el nodo en ese lugar. Para conectarlo a la red, se crea una conexión que sale del nodo del cual salía la conexión desactivada hacia el nuevo nodo, y otra que salga de dicho nodo al nodo al cual estaba conectada la conexión antigua (Figura 9). Como se han añadido 2 conexiones nuevas y se ha deshabilitado otra, para no afectar a los valores de la red, la conexión entrante al nodo nuevo tiene un peso de 1, y la saliente tiene el peso que tenía la conexión antigua.

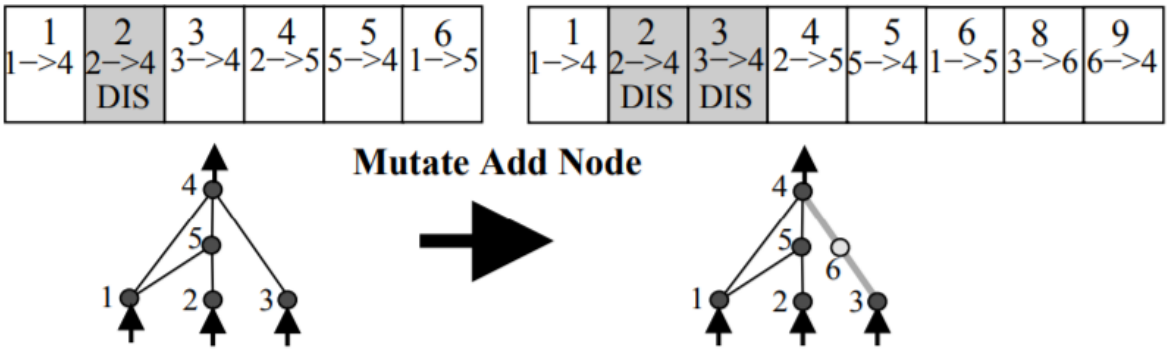

Figura 9. Ejemplo de mutación de un nodo [1]

3.3.2.3.2 Mutación de conexiones

Se trata de un tipo más simple de mutación, ya que simplemente se escogen dos nodos y se crea una conexión con un peso aleatorio entre ellos (Figura 10). El problema obvio que hay que tener en cuenta es que esos nodos no sean del mismo nivel, esto es, por ejemplo, dos nodos de entrada.

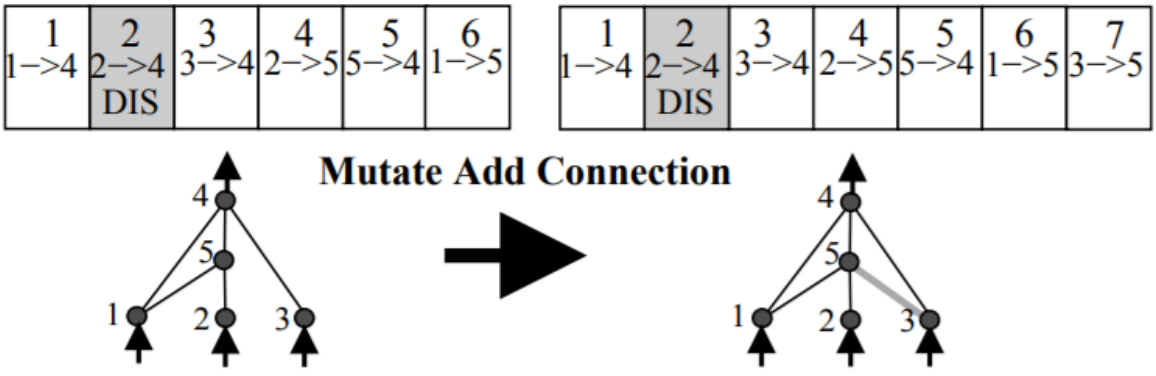

Figura 10. Ejemplo de mutación de una conexión [1]

3.3.2.3.3 Mutación de pesos

La mutación más simple. Dentro de este tipo existen dos subtipos de mutación. La más común es una mutación suave, en la que se suma o resta un número pequeño a una conexión. Con menos probabilidad, se puede realizar una mutación brusca, que vuelve a generar un valor aleatorio para la conexión. Si se quiere hacer una mutación de pesos, se

recorre la red y para cada conexión se realiza uno de los dos subtipos, según la probabilidad asignada.

3.4 Diseño del problema

La red neuronal y el algoritmo genético son elementos que se diseñan y codifican una sola vez, y simplemente se varían distintos parámetros para ajustar los resultados. Sin embargo, interesa que dichos elementos sean útiles independientemente del problema al que se enfrenten. Es por ello por lo que se necesita un módulo que contenga la información del problema a resolver. Dicho módulo es quien instancia la población, decide que entradas y salidas tendrán las redes y evalúa el rendimiento de cada una calculando su fitness. Aquí es donde brilla la utilidad de envolver el módulo que contiene la información de las redes en una clase contenedora. Lo único que hay que hacer es, dentro del problema, crear una nueva clase que herede de "Genoma", la clase contenedora. Así puede tener toda la información necesaria de NEAT, y a la vez, otros métodos o clases propios. Queda más claro con un ejemplo. Se quiere obtener una red que sea capaz de realizar correctamente la función XOR con dos entradas. Primero, se crea un módulo nuevo, "*XORProblem*", y dentro una clase que herede de "Genoma", que será "*XORSolver*". Hay que establecer el número de entradas y salidas de la red. Como solo se quiere resolver la función XOR con dos bits de entrada, tendrá 2 entradas. Como el valor solo puede ser 0 o 1, tendrá una sola salida. Si el valor de la salida se acerca a 0, se tomará que la red indica que el resultado es 0, y si es 1 viceversa. Lo único que hay que hacer es instanciar una población del tamaño deseado de XORSolvers, iterar sobre cada individuo, ponerlo a prueba con las cuatro combinaciones posibles de entradas y ver que resultados arroja. Se necesitará una forma de evaluar lo bien que funciona la red. Una opción sería, por ejemplo, medir el rendimiento en función de los aciertos. Una vez evaluada la población, se genera la siguiente población, y así hasta que se llegue al límite de generaciones dado o se obtenga una solución lo suficientemente buena.
Se aprecia, por tanto, que el diseño es flexible a cualquier problema, y solo hay que seguir una serie de pasos:

- Crear un módulo que contenga la información del problema, y una clase dentro de él que haga las veces de individuo.
- Establecer el número de entradas y salidas del problema.
- Decidir qué valores va a recibir cada individuo en las entradas, así como la forma de interpretar los valores de salida. (Tanto los valores de las entradas, como los de las salidas están comprendidos en el intervalo [0,1]).
- Establecer de qué forma se va a calcular el fitness de cada individuo.

3.5 Diagrama de clases

Recopilando toda la información de este capítulo se puede ya plantear qué clases serán necesarias y como interactúan entre ellas. Omitiendo los atributos de cada clase e indicando las constantes que representan las probabilidades de los operadores genéticos, el diseño final sería el mostrado en la figura 11.

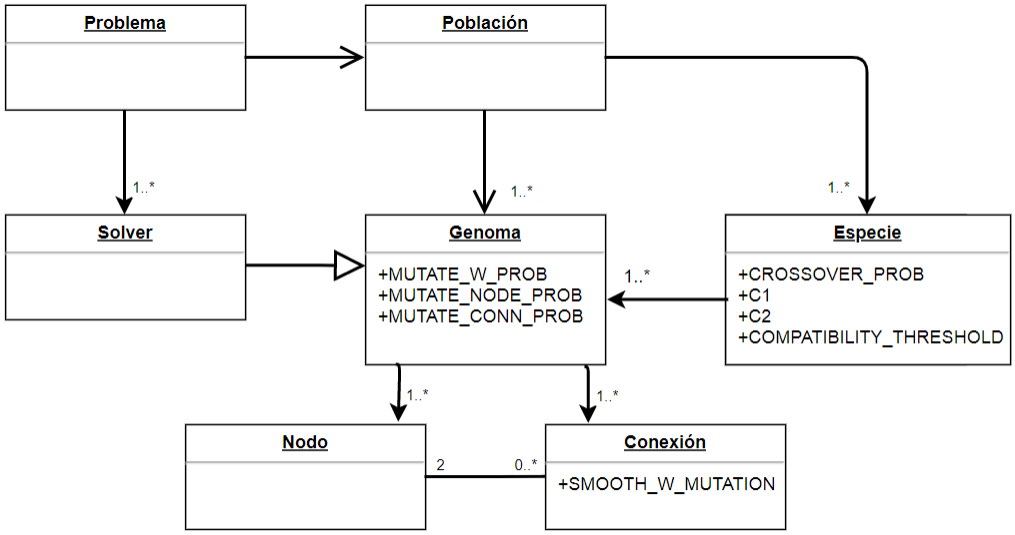

Figura 11. Diagrama de clases

Se puede observar que la clase principal es "Población", que tiene el listado de genomas, que representan los individuos que la componen, así como el listado de especies en las que se van a separar dichos individuos. A su vez, cada red se compone de nodos y conexiones. Cada nodo tiene un listado de conexiones salientes, y cada conexión tiene los dos nodos que conecta. Por otro lado, tenemos la clase con el problema, que tiene la población, y la instancia con individuos del tipo "Solver", que heredan de la clase "Genoma". El diagrama permite ver cómo se relacionan los componentes más estáticos, correspondientes a los elementos que implementan el algoritmo genético, con los dinámicos, relativos a la descripción de cada problema.

Por otro lado, se han anotado los parámetros que o bien reflejan probabilidades, o bien son coeficientes o constantes que afectan a la forma en que evoluciona la población. Además del parámetro obvio, que es el tamaño de la población, estos son los valores que hay que ajustar para un problema en particular. A continuación, se hará un resumen de lo que representa cada parámetro, así como su valor por defecto.

- **CROSSOVER_PROB**: Probabilidad de que, a la hora de cruzar dos individuos, en vez de obtener un hijo y pasarlo a la siguiente generación, pasemos directamente uno de los padres (valor por defecto: 25%).
- **C1**: Coeficiente usado para calcular si dos individuos pertenecen a la misma especie, según la ecuación 5 (valor por defecto: 1).
- **C2**: Misma utilidad que C1. Equivale a C3 en la ecuación 5 (valor por defecto: 0.4).
- **COMPATIBILITY_THRESHOLD**: Aplicando la fórmula descrita en la ecuación 5, se obtiene un valor que representa lo parecidos que son dos individuos. Si ese valor sobrepasa un umbral, se consideran distintos. Este valor representa dicho umbral.
- **MUTATE_W_PROB**: Probabilidad de que la red mute sus pesos (80%).
- **MUTATE_NODE_PROB**: Probabilidad de que la red añada un nodo (3%).
- **MUTATE_CONN_PROB**: Probabilidad de que la red añada una conexión (5%).
- **SMOOTH_W_MUTATION**: Probabilidad de que cuando una red quiere mutar sus conexiones, la mutación de los pesos sea suave (90%). Por tanto, la

probabilidad de que la mutación de pesos sea brusca es del 10%. Esta probabilidad se calcula conexión por conexión, y no se aplica al conjunto en sí.

4 Desarrollo de los problemas

4.1 Codificación

Una vez se tiene el diseño bien definido, la codificación es relativamente sencilla. Lo más simple y eficiente es codificar los módulos más compactos, sin dependencias, someterlos a pruebas unitarias e ir posteriormente escalando, con el propósito de evitar encontrar problemas.

El objetivo es codificar todo sin librerías auxiliares, por tanto, utilizar un lenguaje como Python simplifica mucho el manejo de los datos. Es por ello por lo que es el elegido para este proyecto. Además, cuenta con una librería gráfica, Pygame, que va a ser útil para representar visualmente algunos problemas.

En cuanto a la parte del algoritmo genético y de las redes, si se sigue el diseño correctamente, la implementación es sencilla. Como problema se puede mencionar que como se están manejando poblaciones con varios individuos, que a su vez tienen varios nodos y conexiones, y muchas de las situaciones dependen de probabilidades, depurar el código puede llegar a ser bastante tedioso.

La parte del problema a resolver, por otro lado, presenta un carácter más abierto. Como ya se ha explicado, el diseño propuesto permite flexibilidad ante el tipo de problema a solucionar. Mientras se especifique el número de entradas y salidas de las redes, los datos de entrada, y la función que calcula el rendimiento de los individuos, cualquier problema es, en principio, tratable. Luego, la codificación genética de los individuos de cada problema es la misma, los genes no son más que un listado de conexiones y nodos.

Por tanto, el proyecto se evaluará con 3 tipos de problemas y centraremos la discusión sobre el desarrollo en la forma en que debe describirse cada uno de esos problemas abordados. El desarrollo del módulo de redes neuronales y el del algoritmo genético, como se ha indicado, se consideran auto-contenidos a partir de su diseño.

4.1.1 Primer tipo. XOR

En primer lugar, se intenta resolver un problema sencillo. Un ejemplo clásico sería resolver correctamente la puerta lógica XOR con 2 bits de entrada (Figura 12).

Entradas		Salida
0	0	0
0	1	1
1	0	1
1	1	0

Figura 12. Comportamiento de la puerta lógica XOR

El flujo del programa será, por tanto:

- Inicializar la población con redes de 2 entradas y una salida.
- Para cada individuo, introducir las 4 posibles combinaciones de valores de entrada, y después comprobar y evaluar qué valor se devuelve como salida para cada una de esas cuatro entradas. Comparar la salida propuesta por la red evolucionada con la salida deseada.
- Si no se ha encontrado una solución y no se ha llegado al límite de generaciones, crear la siguiente generación aplicando operaciones genéticas.

Hay que tener en cuenta dos cosas. La primera, cómo interpretar el valor de salida. Todas las entradas de la red deben tener valores comprendidos entre 0 y 1. En este caso las entradas ya tienen esos valores, pero en otros problemas habría que tenerlo en cuenta. Las salidas, por tanto, también son valores comprendidos en ese intervalo. Una aproximación inicial sería interpretar que, si el valor devuelto es menor que cierto umbral, la red "cree" que el resultado es un 0, y si es mayor que otro umbral dado, un 1. La robustez de la predicción dependerá de lo cerca que esté el valor devuelto de 0 y de 1. Un valor de salida de 0.95 se puede interpretar como que la red cree que el valor a predecir es un 1 con un 95% de seguridad, y un 0 con un 5%. Por tanto, los umbrales sirven para establecer a partir de qué porcentaje se considera una predicción robusta. Puede asumirse que a partir de un 90% el resultado es fiable. Dichos umbrales, por tanto, son 0.1 para el 0, y 0.9 para el 1. La segunda idea a tener en cuenta es cómo medir el rendimiento de un individuo. En este caso, una forma simple de hacerlo consiste en contar cuántos aciertos obtiene sobre los 4 casos.
Para correr una red, la clase "Genoma" tiene un método dedicado a obtener el valor de salida a partir de los valores de entrada. Esencialmente, se recorre cada nodo, en orden, según la capa a la que pertenezcan, y se calcula de forma manual el valor de salida de cada uno de dichos nodos en función de sus entradas. Aquí se puede apreciar la utilidad de guardar en cada nodo la capa a la que pertenece.

Se escoge el problema de la XOR porque no tiene una solución lineal. Eso es equivalente a decir que, si la red no tiene al menos una capa interna de nodos ocultos, el problema no se puede resolver. Por tanto, la población inicial no será capaz de clasificar correctamente los 4 casos y habrá que esperar a que un individuo mute para conseguir un nodo intermedio nuevo en la red (como mínimo). A partir de ese momento, esos genes nuevos se irán transmitiendo hasta desarrollarse en una solución viable. Este comportamiento se puede ver en el apartado de pruebas y resultados.

4.1.2 Segundo tipo. Clasificación

El problema de la XOR es extremadamente simple. Una vez se comprueba que el sistema funciona para dicho problema, interesa ver cómo se comporta con conjuntos de datos más grandes.

Para ello, se ha decidido utilizar el conjunto de datos típico conocido como "Iris" **[8]**. Este conjunto tiene datos sobre la longitud y anchura de los pétalos y sépalos de 3 especies distintas (Figura 13). La idea es que, con estos datos, el sistema sea capaz de distinguir entre los tipos de especies.

El dataset contiene 150 entradas, con 4 atributos por entrada. Hay 3 especies, y por ello hay 50 entradas de cada especie. Cada entrada se puede visualizar de la forma en la que se recoge en las figuras 14 y 15.

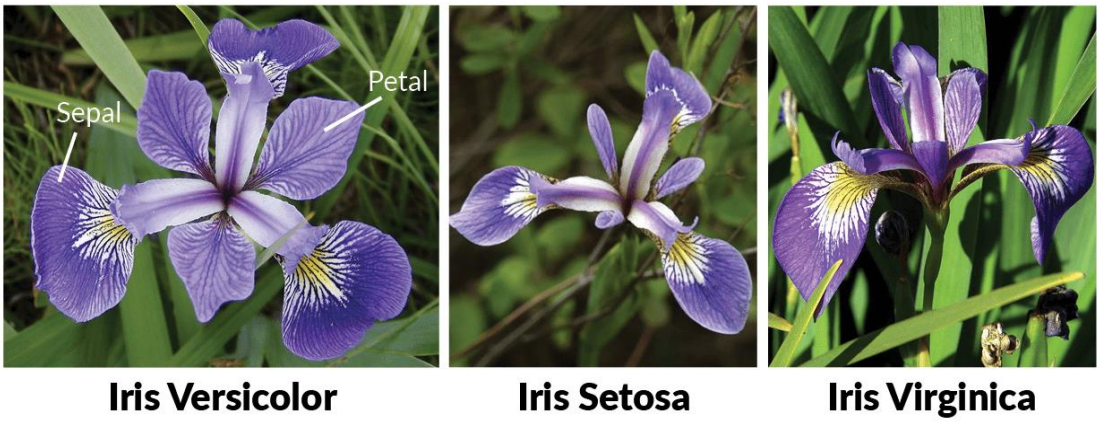

Figura 13. Tipos de especies Iris [8]

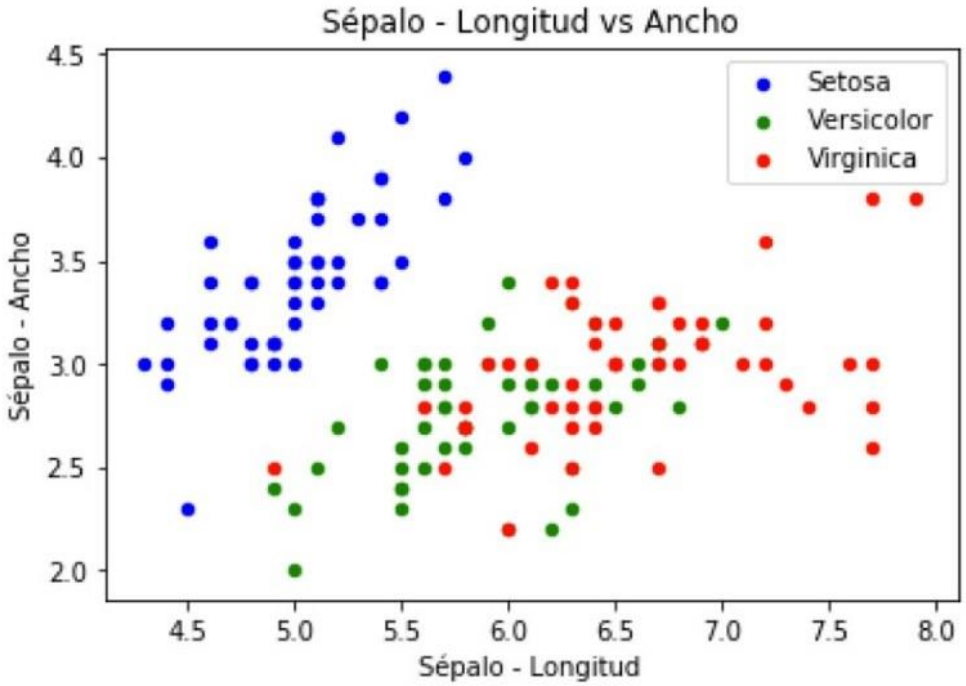

Figura 14. Representación gráfica de cada especie en función de la longitud y anchura del sépalo

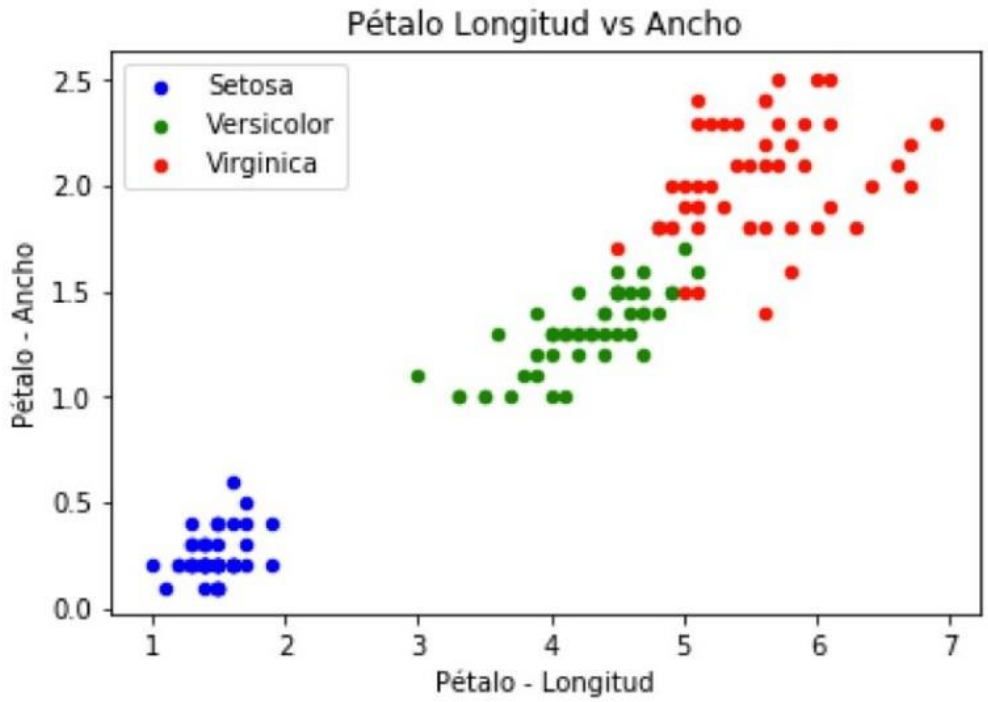

Figura 15. Representación gráfica de cada especie en función de la longitud y anchura del pétalo

Como puede apreciarse, hay un tipo de especie que es más fácil de distinguir que los otros dos, pues se puede clasificar de forma lineal. Sin embargo, distinguir entre las especies Versicolor y Virginica requiere de mayor precisión, y, por tanto, de mayor complejidad del sistema.

De la misma forma que en el problema de la XOR, simplemente se construyen redes con 4 entradas, una por atributo, y esta vez 3 salidas. Para cada dato, el sistema se quedará con la salida de mayor valor, y en función de ese valor, representará la predicción de una clase u otra. Para obtener el rendimiento de un individuo se calcula el número de aciertos sobre los 150 datos.

4.1.3 Tercer tipo. Problema en tiempo real.

Los dos tipos de problemas anteriores tienen un carácter más teórico. Al fin y al cabo, se tratan de problemas de clasificación supervisada clásicos. Es por ello por lo que es interesante aplicar NEAT a otro tipo de problemas. En lugar de suministrar un conjunto de datos a cada individuo y estudiar cómo se comporta, podemos enfrentar al sistema a una situación que requiera una respuesta en tiempo real.

Para ello, la idea es codificar algún tipo de minijuego y ver de forma gráfica cómo se comporta cada individuo. Un ejemplo sencillo es imitar el juego conocido como “Snake”, donde simplemente el jugador es una serpiente que tiene que ir recogiendo comida para aumentar la puntuación y su longitud (Figura 16).

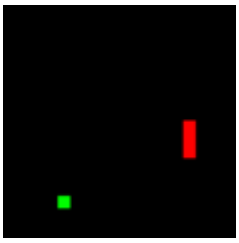

Figura 16. Estado inicial del juego (Snake)

Este tipo de problemas es bastante más interesante, porque los resultados van a depender en gran medida de cómo se calcula el rendimiento, (o se castiga el comportamiento que no interesa), y qué entradas tiene cada individuo. Las salidas, sin embargo, son claras, el individuo puede avanzar en línea recta, o girar a la izquierda o derecha. Luego, de la misma forma que en el problema del dataset Iris, habrá 3 salidas, el sistema se quedará con la de mayor valor, y según qué salida sea esa, el individuo realizará un movimiento u otro.

Como se quiere hacer hincapié en que se pueda apreciar la evolución de los individuos, la idea es simularlos todos a la vez en su propio tablero o casilla (Figura 17).

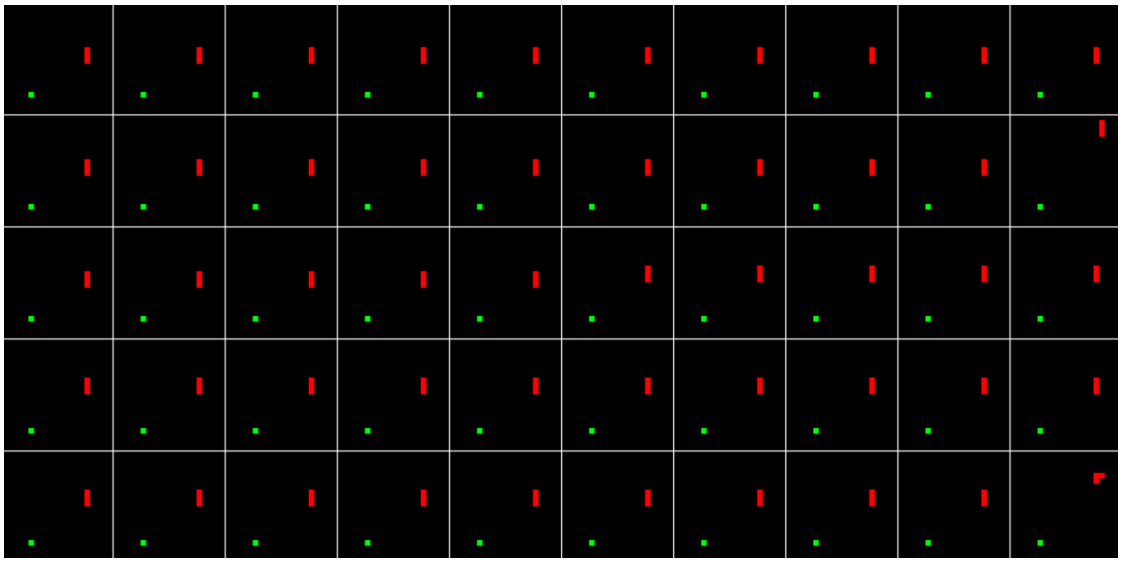

Figura 17. Estado inicial de la población (Snake)

En este caso la población sería de 50 individuos, para que todos quepan en pantalla, y así se pueda ver cómo se van desarrollando.

5 Integración, pruebas y resultados

5.1 Problema XOR

Como ya se mencionó en el apartado de desarrollo, la XOR no tiene solución lineal. Esto implica que NEAT no va a ser capaz de encontrar una solución hasta que algún individuo mute, de forma que añada un nodo nuevo intermedio.

En el artículo original de NEAT **[1],** también se evalúa la XOR. En particular, se dice lo siguiente:

"*The standard deviation for number of nodes used in a solution was 1.11, meaning NEAT very consistently used 1 or 2 hidden nodes to build an XOR network. In conclusion, NEAT solves the XOR problem without trouble and in doing so keeps the topology small."*

Es decir, que con 1 o 2 nodos ocultos el sistema es capaz de encontrar una solución rápidamente. También se muestran dos diagramas (Figura 18), donde se aprecia la red en su estado inicial, y la estructura más simple y compacta que puede dar una solución.

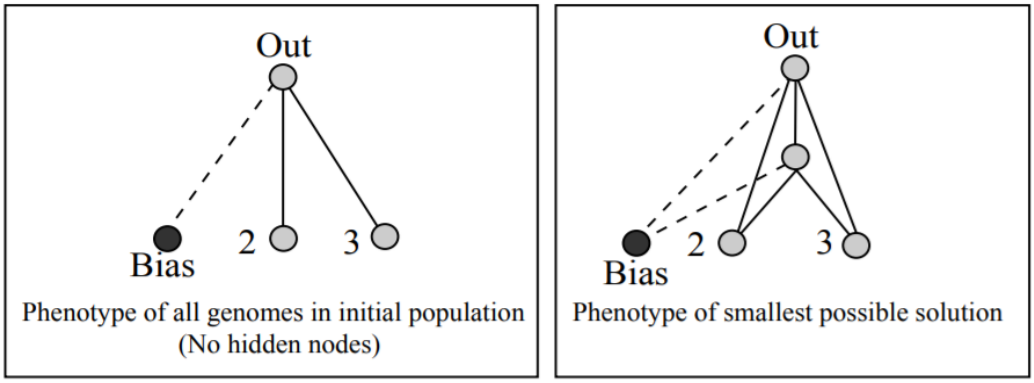

Figura 18. Red neuronal mínima para resolver el problema XOR [1]

Nótese como en la situación inicial todas las entradas están conectadas a la salida. En el apartado de diseño se discutió sobre ello, explicando que en este proyecto no se va a seguir esta idea, sino que solo va a haber una conexión inicial entre una entrada y una salida elegidas ambas de forma aleatoria. La argumentación que se propone es que puede que haya entradas en problemas más complejos que no sean necesarias, lo cual simplifica la red. En este problema, en cambio, todas las entradas se necesitan, así que es una desventaja en esta situación, pues hay que esperar a que aparezcan nuevas conexiones por mutación que el modelo estándar de NEAT trae inicialmente.

Entre otros parámetros, en **[1]** se toma como probabilidad de mutación de añadir un nodo un 3%, y de añadir una conexión un 5%. Si en la implementación propia se usase una probabilidad de añadir una conexión tan baja, al no tener las conexiones iniciales, lo que pasaría es que para cuando se encuentra una solución aceptable, la red tiene varios nodos internos, y no uno o dos. Una de las ventajas principales de NEAT es precisamente encontrar soluciones compactas, evitando añadir elementos adicionales. Para comprobar lo que se está exponiendo, ejecutamos la simulación con esa probabilidad de conexión tan baja. En particular, lanzamos la simulación con los siguientes valores, que son los mismos que en **[1]**:

CROSSOVER_PROB	**0.25**
C1	**1**
C2	**0.4**
CompatibiltyThreshold	**3**
MUTATE_W_PROB	**0.8**
MUTATE_NODE_PROB	**0.03**
MUTATE_CONN_PROB	**0.05**
SMOOTH_W_MUTATION	**0.9**

Tabla 2. Parámetros empleados para resolver la XOR

El resultado obtenido es bastante pobre. Para encontrar una solución, el sistema crea redes demasiado complejas. Un resultado de ejemplo es el mostrado en la figura 19.

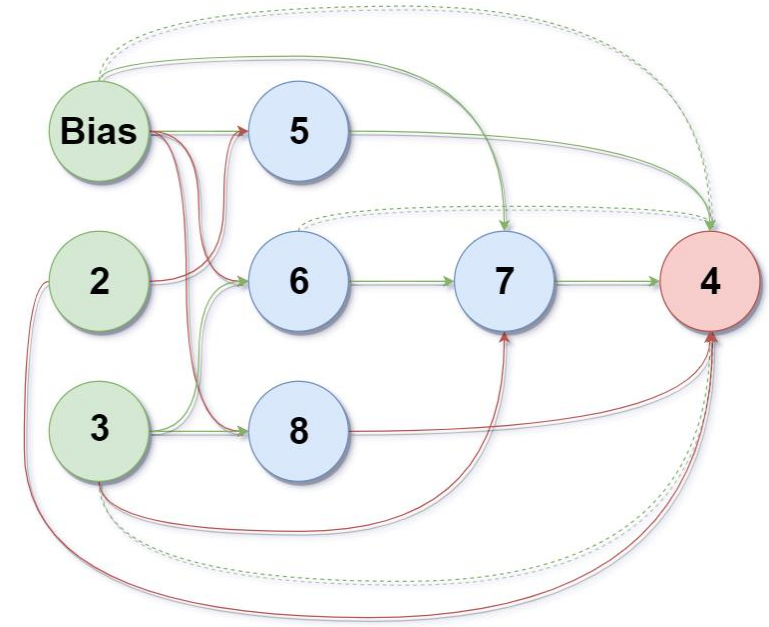

Figura 19. Red neuronal que resuelve XOR, primera aproximación

Los pesos de las conexiones se han omitido por claridad en el diagrama. En su lugar, se muestran de color rojo las conexiones con pesos con valores negativos. Las verdes representan valores positivos. Las conexiones representadas con líneas discontinuas representan conexiones desactivadas.

Si se observa la figura 19, se puede ver que, activadas o no, existen conexiones entre todos los nodos. Por tanto, si volvemos a correr la simulación con una probabilidad de mutación alta, del 90% por ejemplo, en vez del 5% original, obtenemos resultados mucho mejores (Figuras 20, 21, 22).

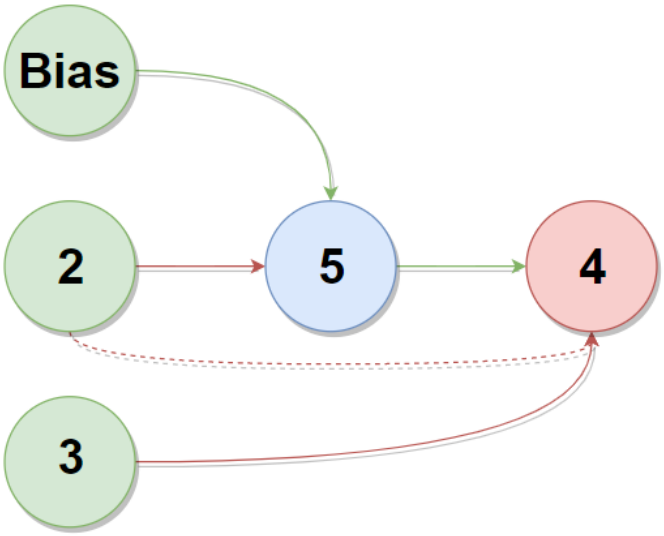

Figura 20. Red neuronal 1 que resuelve XOR, segunda aproximación

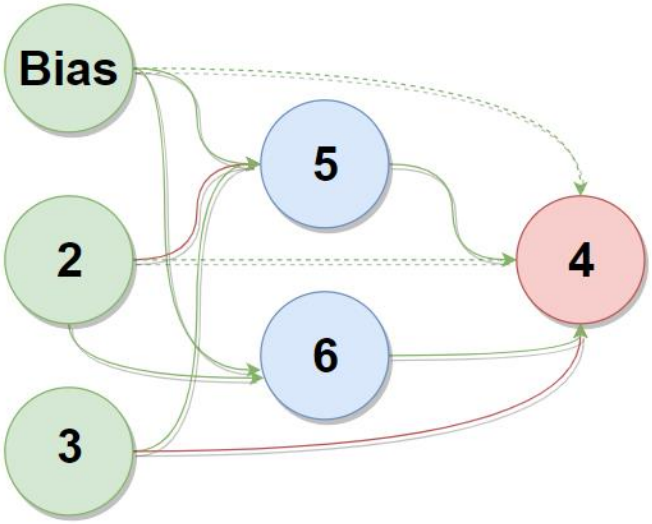

Figura 21. Red neuronal 2 que resuelve XOR, segunda aproximación

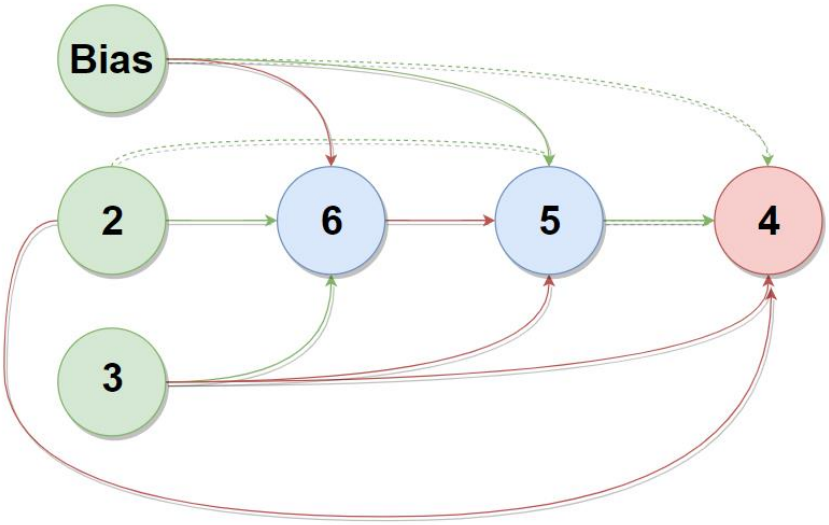

Figura 22. Red neuronal 3 que resuelve XOR, segunda aproximación

Para medir el rendimiento de cada red, inicialmente se puede pensar en dar una puntuación proporcional al número de aciertos. Sin embargo, esto no incita a mejorar. Es decir, es cierto que las redes que acierten más tendrán más posibilidades de pasar sus genes, pero contar los aciertos no ofrece una idea general a la red acerca de cómo ajustar su estructura. Simplemente se generarán estructuras de forma aleatoria hasta que aparezca una que funcione. En cambio, si para cada resultado calculamos el error con respecto al valor esperado, podemos dar una intuición al sistema sobre cuán equivocado está, y no solo indicarle si se ha equivocado o no, con lo cual el sistema se ajusta mucho más rápido **[9].** La fórmula actualizada que calcula el fitness de esta forma es la siguiente (Ecuación 6):

$$4 - \sum (Expected_i - Result_i)^2$$

Ecuación 6. Cálculo del fitness optimizado para XOR [9]

Así, cada individuo empieza con un fitness de 4. Por cada dato de entrada, se calcula el error, que no puede ser mayor que 1, y se le resta al fitness inicial. Hay que establecer un umbral por el cual se considera una solución aceptable. Un umbral preciso sería establecer que la suma de los 4 errores no supere 0.1. Esto es, que si el fitness del individuo tras procesar los datos es mayor a 3.9, se considera que dicha solución es aceptable.

Aplicando la nueva función de fitness, los resultados del crecimiento del fitness para las 3 redes neuronales mostradas anteriormente en las figuras 20, 21 y 22, se pueden visualizar en las figuras 23, 24, y 25, respectivamente.

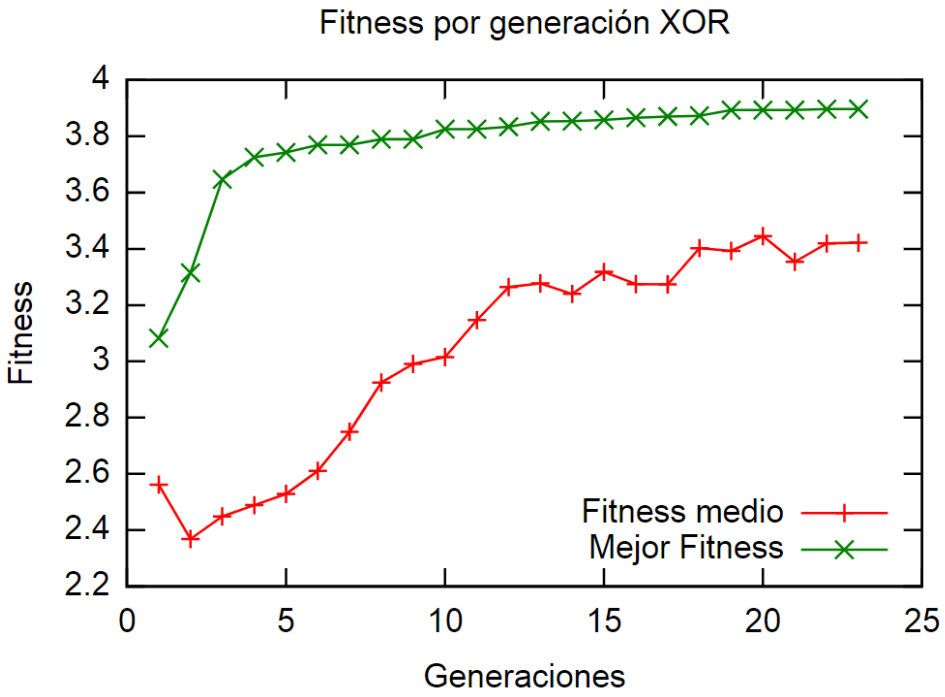

Figura 23. Evolución del fitness para la red mostrada en la figura 20

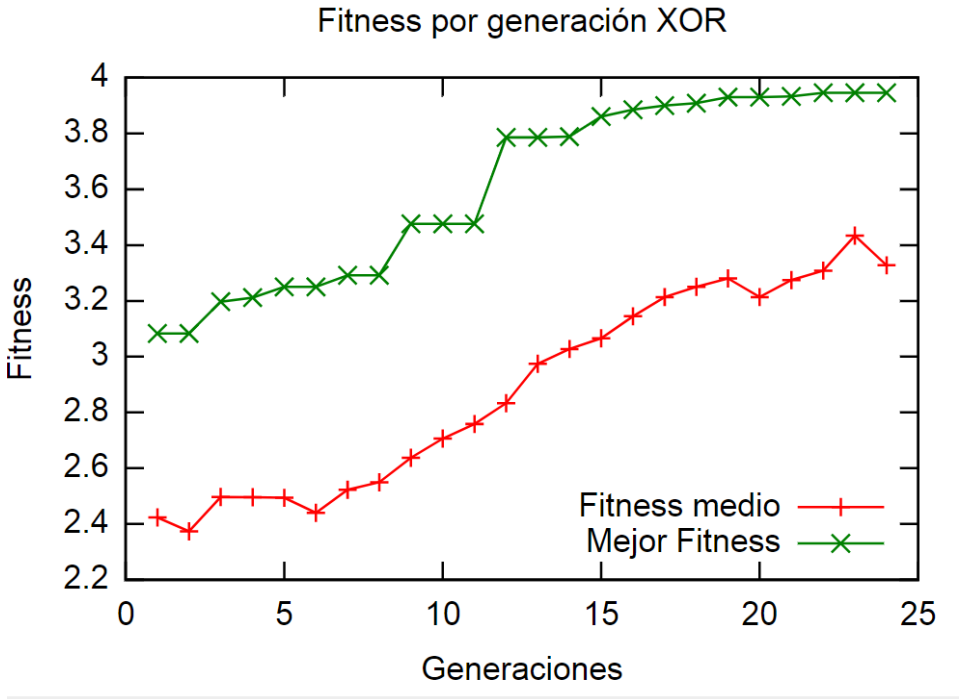

Figura 24. Evolución del fitness para la red mostrada en la figura 21

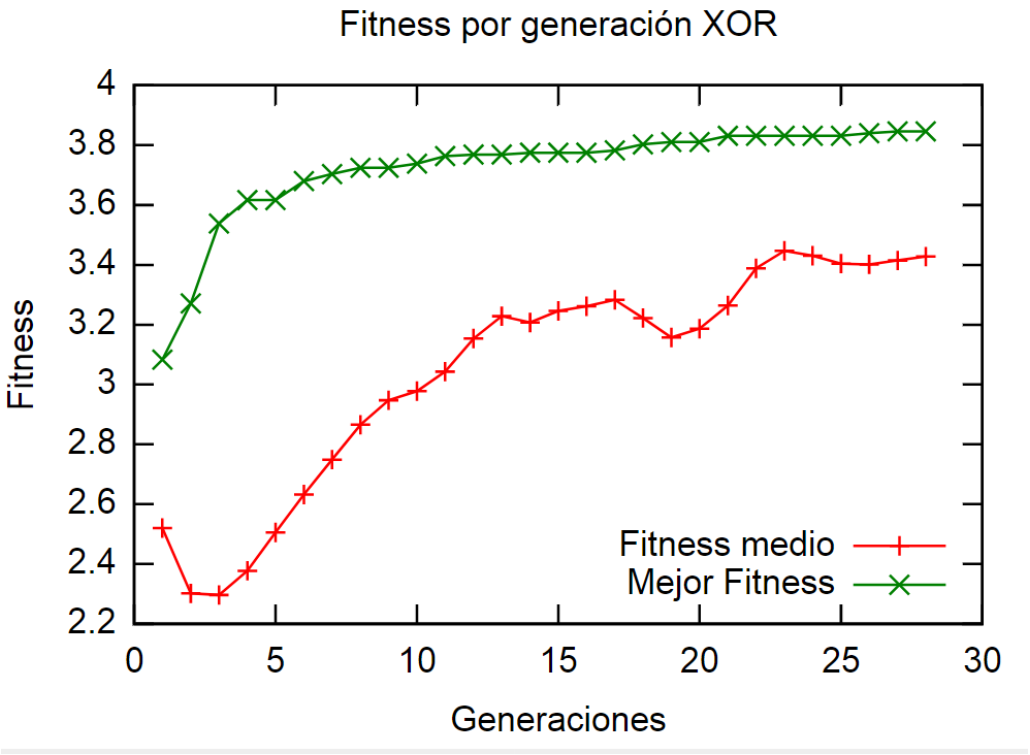

Figura 25. Evolución del fitness para la red mostrada en la figura 22

Se puede observar el mismo patrón. Aparece un crecimiento inicial rápido en las primeras generaciones, para continuar con un crecimiento más lento posteriormente. Probablemente esto se deba a que, al aparecer el primer nodo oculto en las primeras generaciones, el error se reduce drásticamente, y luego es cuestión de ajustar los pesos de las conexiones.

Para asegurarse de que el sistema funciona correctamente, se ejecuta 100 veces. A todas las simulaciones se le da un límite de 100 generaciones, y si no se consigue encontrar una solución en ese intervalo, se considera una simulación fallida. La población inicial es de 150 individuos.

De las 100 simulaciones, ninguna falla. En la figura 26 se puede observar el número de generaciones necesarias para encontrar una solución en cada simulación.

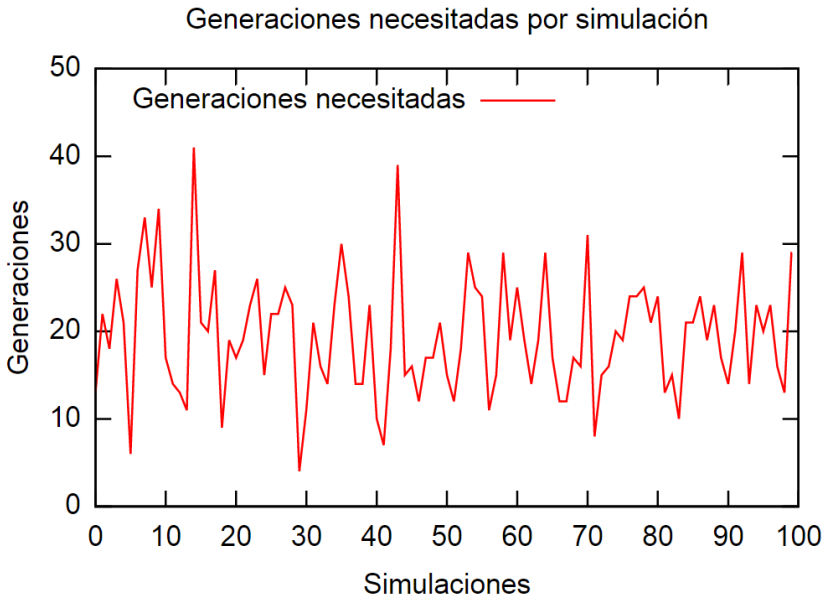

Figura 26. Generaciones necesitadas en 100 simulaciones para resolver XOR

Como puede apreciarse, en el peor de los casos NEAT necesita de alrededor de 40 generaciones para encontrar una solución, concretamente 41. En el mejor de los casos, en 6 generaciones ya se encuentra una solución aceptable. La media de nodos internos necesitados es de 2.2 nodos, valor bastante aceptable. La misma medida en **[1]** es 1.1, un valor que es inferior al obtenido en la implementación propia de NEAT que se refleja aquí. El problema de ese valor es que no se especifica cómo se calcula el fitness, y a partir de qué umbral se considera una solución válida. Probablemente, si se conociera esa información, se podría simplificar aún más las redes. Esto sirve de ejemplo para mostrar la importancia de elegir una buena función de fitness.

5.2 Problema de clasificación

Una vez se sabe que la versión de NEAT implementada funciona bien con la puerta lógica XOR, se trabaja con conjuntos de datos más grandes y complejos. El conjunto "Iris" tiene 3 clases entre las que seleccionar, con lo cual no puede aplicarse la misma estrategia que para la XOR, donde si el valor de salida se acerca a 1, se considera un 1, y viceversa con el 0. En su lugar, nos quedamos con el valor máximo de las 3 salidas. Además, este conjunto de datos tiene la característica de que una clase es muy sencilla de separar de las otras dos, pero estas dos son más difíciles de clasificar entre sí. Por tanto, se debería apreciar que NEAT es capaz de acertar como mínimo, en los datos de la clase que se clasifica fácilmente de forma rápida. Después, costará algo más que acierte con el resto de los datos, y el crecimiento será más lento, ajustando los pesos para aumentar la precisión de la frontera de decisión.

Para la función de fitness, inicialmente se prueba simplemente contando el número de aciertos. Hay 150 entradas en el conjunto de datos, y cada acierto suma 1 punto al fitness de un individuo, así que el fitness varía entre 0 y 150.

En las primeras simulaciones de prueba, los resultados son bastante buenos (Figura 27). Los aciertos del mejor individuo en poblaciones de 200 individuos en 100 generaciones rondan entre 144 y 145. Esto representa en torno al 96.6% de aciertos.

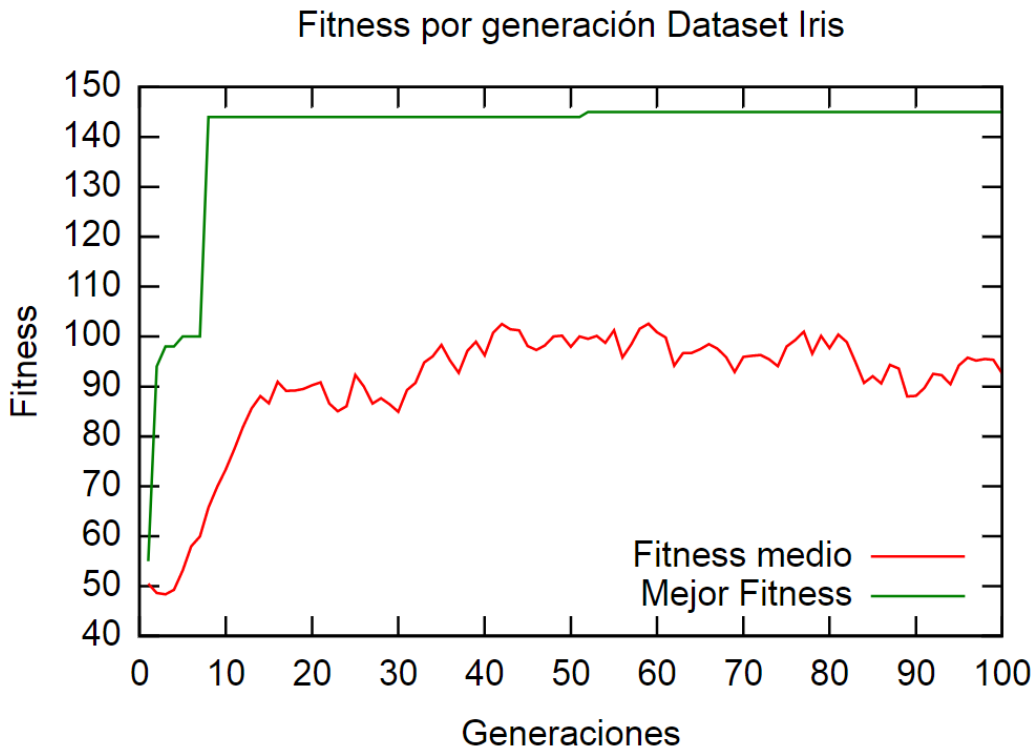

Figura 27. Crecimiento del fitness para el dataset Iris

Estudiando la figura 27 se pueden deducir algunas ideas. Puede verse como, en general, el fitness inicial es 50. Esto implica que desde el primer momento predice correctamente una clase, pues de las 150 entradas hay 50 de cada clase. El valor tiene sentido, pues las redes iniciales son tan simples, que lo que hacen es predecir para todos los datos la misma clase.

También se ve un crecimiento muy rápido hacia un fitness de 100. Lo que ocurre en ese momento es probablemente que el sistema aprende a distinguir entre una clase respecto de las otras dos, pero todavía no sabe separar esas dos clases. Como ya se ha mencionado, esto ocurre porque una clase es fácilmente separable de las otras, pero la frontera entre las dos restantes es más compleja. A partir de ese momento, se produce un crecimiento del fitness lento, en el cual el sistema aprende a separar los valores de las dos clases restantes. Se puede apreciar que no se necesitan 100 generaciones ya que un resultado aceptable se obtiene para los mejores individuos en torno a la generación 10.

También se puede apreciar que el fitness medio de la población es bastante bajo en función del fitness de los mejores de cada generación. Esto se debe a la especiación. Hay especies peores que inicialmente se protegen, por si contienen una solución que puede ser útil. Por defecto, si esas especies no crecen en 10 generaciones, se eliminan completamente. Puede verse que eso ocurre a partir de la generación 10, donde el crecimiento medio crece drásticamente.

La red neuronal que produce esta clasificación es la siguiente (figura 28):

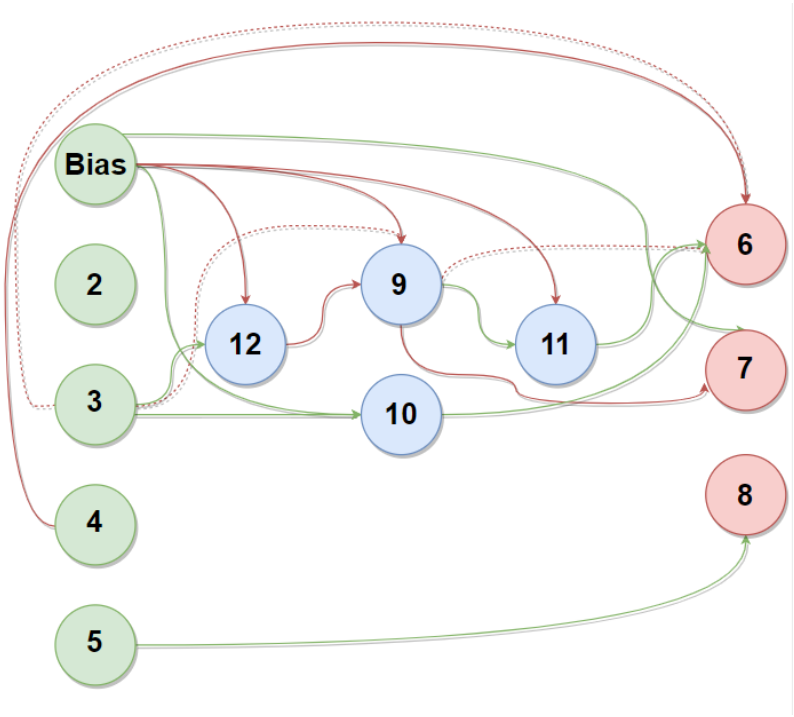

Figura 28. Red que obtiene los resultados de la figura 27

Eliminando del diagrama las conexiones desactivadas, que no tienen utilidad (figura 29):

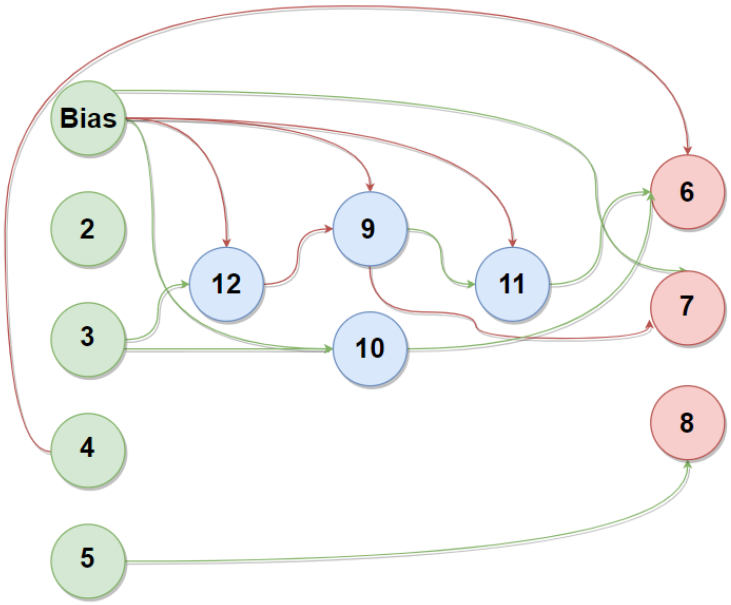

Figura 29. Red de la figura 28 simplificada

Puede verse lo simple que es la red, en comparación con el conjunto de datos proporcionado. Sólo se necesitan 4 nodos internos, y varias de las entradas apenas se usan. De hecho, el nodo número 2 no está conectado a nada. Ese nodo equivale al atributo que representa la longitud del sépalo de las flores. Aquí se aprecia la ventaja de no partir con todas las entradas conectadas a todas las salidas. Eso permite potenciar los atributos que el sistema considera importantes. Otra ventaja es que no solo no hay que construir la red de forma manual, probando combinaciones de estructuras, si no que a primera vista parece

absurdo ignorar uno de los atributos, y, sin embargo, el sistema funciona relativamente bien. Es decir, se pueden encontrar soluciones y estructuras nuevas que inicialmente no se habían pensado.

Otro aspecto interesante es que NEAT puede encontrar múltiples tipos de estructuras que son soluciones igual de válidas. Por ejemplo, otra estructura diferente a la anterior sería la mostrada en la figura 30. En dicha figura, (omitiendo las conexiones desactivadas por claridad) se puede ver que ahora todas las entradas son utilizadas, y hay un nodo interno menos. La estructura es bastante diferente. Si se lanzan múltiples simulaciones y se estudia la estructura del mejor individuo resultante, puede verse como dicha estructura va cambiando. En una red convencional con todos los nodos conectados entre sí, habría infinidad de conexiones que realmente son inútiles, y aquí puede verse que no se necesita conectar la red de forma completa para obtener soluciones aceptables.

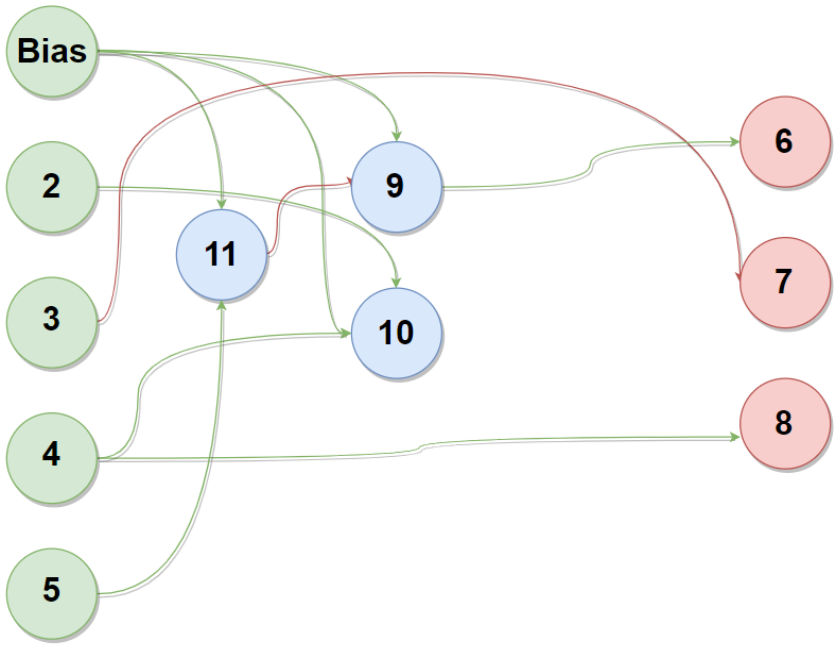

Figura 30. Segundo tipo de red utilizada en el dataset Iris

Una vez se hacen unas simulaciones de prueba y se ve que el sistema suele alcanzar el 96% de aciertos como mínimo, se comprueba de forma más exhaustiva que NEAT funciona correctamente, de la misma forma que en el problema de la XOR. Es decir, se lanzan 100 simulaciones, y se comprueba si en menos de 100 generaciones se alcanza un resultado satisfactorio. En este caso dicho resultado supone acertar como mínimo un 96% de los casos. Los resultados pueden observarse en la figura 31.

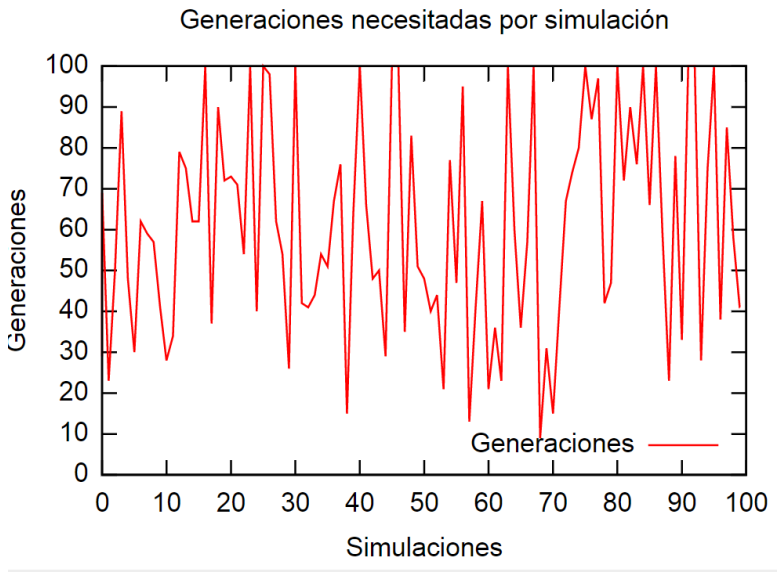

Figura 31. Generaciones necesitadas en 100 simulaciones para el problema de clasificación

Si se establece el límite para 100 generaciones, puede observarse que no siempre es suficiente para clasificar el problema con esa precisión. Como soluciones para ello, simplemente se puede aumentar el número de generaciones límite, o aumentar el tamaño de la población. Sin embargo, el sistema es capaz de encontrar una solución aceptable de forma general, lo que demuestra que NEAT es útil para problemas de clasificación algo más complejos que resolver la XOR.

5.3 Problema en tiempo real

Los problemas con conjuntos de datos son interesantes para ver la eficiencia de NEAT. Sin embargo, la potencia de NEAT no se limita a esa clase de problemas. Se puede aprovechar el carácter evolutivo del algoritmo para darle un enfoque más "biológico" a los problemas.

Una de las aplicaciones que tiene NEAT es crear redes que sean capaces de superar juegos simples. Para esa clase de problemas, no basta con suministrar al sistema el conjunto de datos y estudiar la salida una sola vez, si no que se necesita que el individuo "observe" el estado del juego en cada iteración, y actúe en consecuencia.

El juego elegido será una imitación de "Snake". Por tanto, la función de fitness aquí es bastante más complicada de establecer que simplemente el número de aciertos sobre un conjunto de datos. Hay que orientar al individuo para premiar el ir a por la comida, y mantenerse vivo.

En esta clase de problemas se puede apreciar mejor las respuestas que puede generar el sistema según la función de fitness se le proporcione. Si, por ejemplo, se premia simplemente el mantenerse vivo, el sistema se da cuenta de que no hace falta ir hacia la comida para aumentar el fitness, así que se queda dando vueltas en círculos. Por tanto, no vale con mantenerse vivo, hay que premiar moverse hacia la comida y castigar alejarse de ella. Esta idea se recoge en **[10].** Es más, para eliminar a las especies que aun así se queden en un bucle, cada individuo llevará un contador de hambre. A cada paso que el individuo realice, el contador disminuye. Si llega a 0, muere, y si llega a la comida, se reinicia. Por defecto el valor del contador es inicializado a 30, para un tablero de 20 celdas de ancho y alto.

El fitness, por tanto, se calcula de la siguiente manera:

- Si el individuo se acerca a la comida, aumenta en 1.
- Si se aleja con respecto de la posición anterior, disminuye en 0.5.
- Si se llega a la comida, aumenta en 15.

Aplicando esta función de fitness puede verse como los individuos evolucionan, y "comprenden" que el objetivo es buscar la comida. Lo interesante es pensar que realmente no ven el tablero en sí, ni siquiera se ven a sí mismos, solo las casillas adyacentes a las que se pueden mover, y hacia qué dirección está la comida.

Los resultados en función del fitness de las 100 primeras generaciones, para una población de 50, son los siguientes (figura 32):

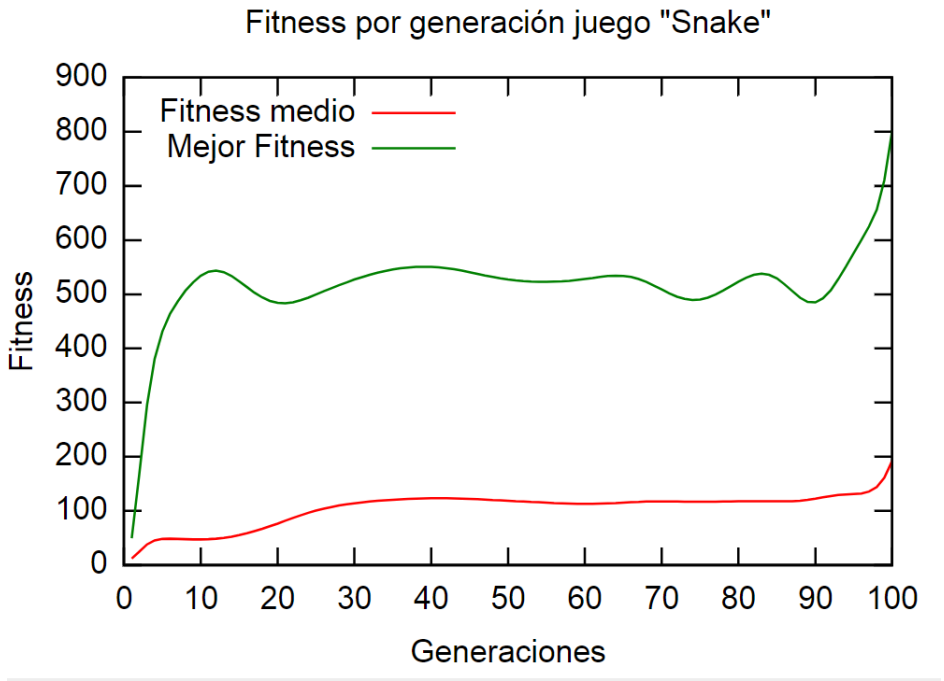

Figura 32. Evolución del fitness para el problema a tiempo real

Puede verse como el fitness de los mejores individuos crece rápidamente al principio. Esto es debido a que se aprende rápidamente a ir a por la comida para sobrevivir. Sin embargo, el fitness medio no cambia demasiado. Al igual que ocurría con el problema de clasificación, la población está dividida en varias especies, y hay especies que no aportan demasiado, y disminuyen el fitness medio.

La red neuronal del mejor individuo viene representada en la figura 33.

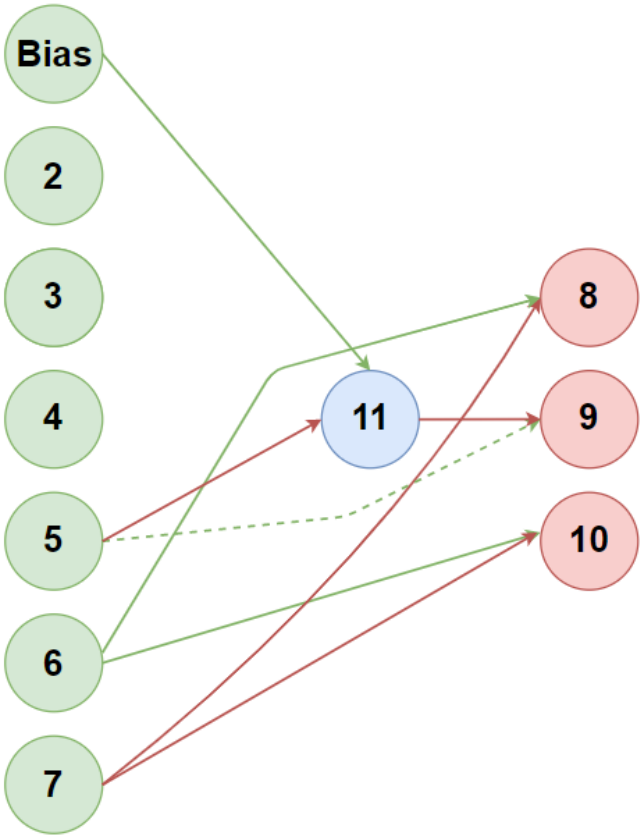

Figura 33. Red neuronal que obtiene el resultado de la figura 32

Puede apreciarse que las entradas 2,3 y 4 son completamente ignoradas, y, sin embargo, este es el individuo que mejores resultados obtiene. Esto es así porque esas entradas representan si hay peligro a la izquierda, derecha, o delante del individuo. Como el único peligro que hay son las paredes, y la comida está dentro de las paredes, no hace falta ni saber donde están dichos límites del tablero.

Para complicar el problema y hacerlo mas interesante, se añade un muro en el medio del tablero, para que el individuo no tenga que aprender simplemente a ir a por la comida, si no que ahora también debe aprender a evitar obstáculos (Figura 34).

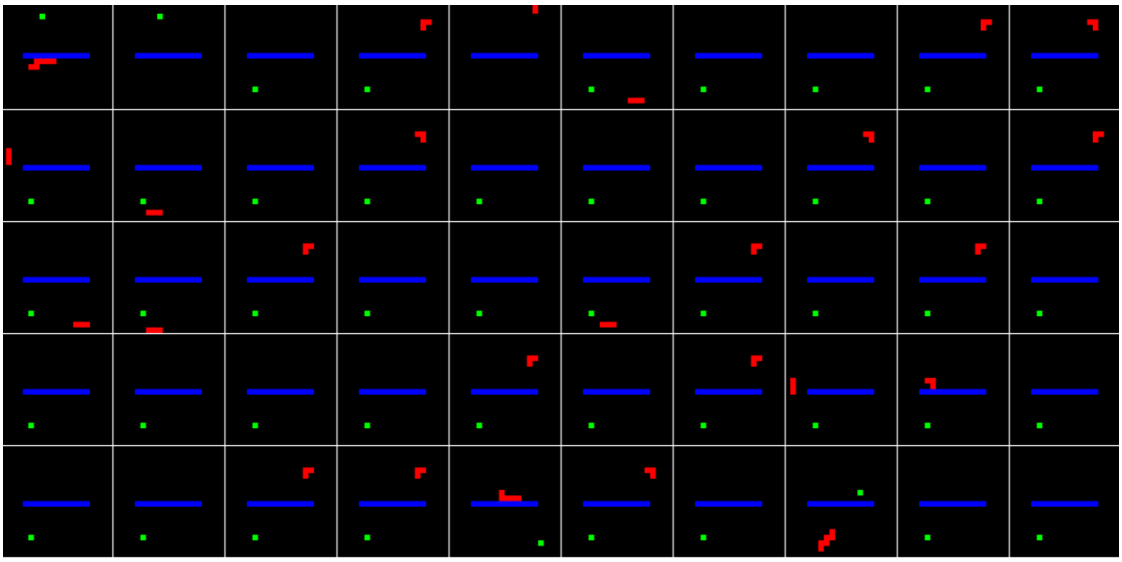

Figura 34. Juego Snake actualizado con muros

En la figura 35 se puede apreciar la nueva red que tiene en cuenta obstáculos.

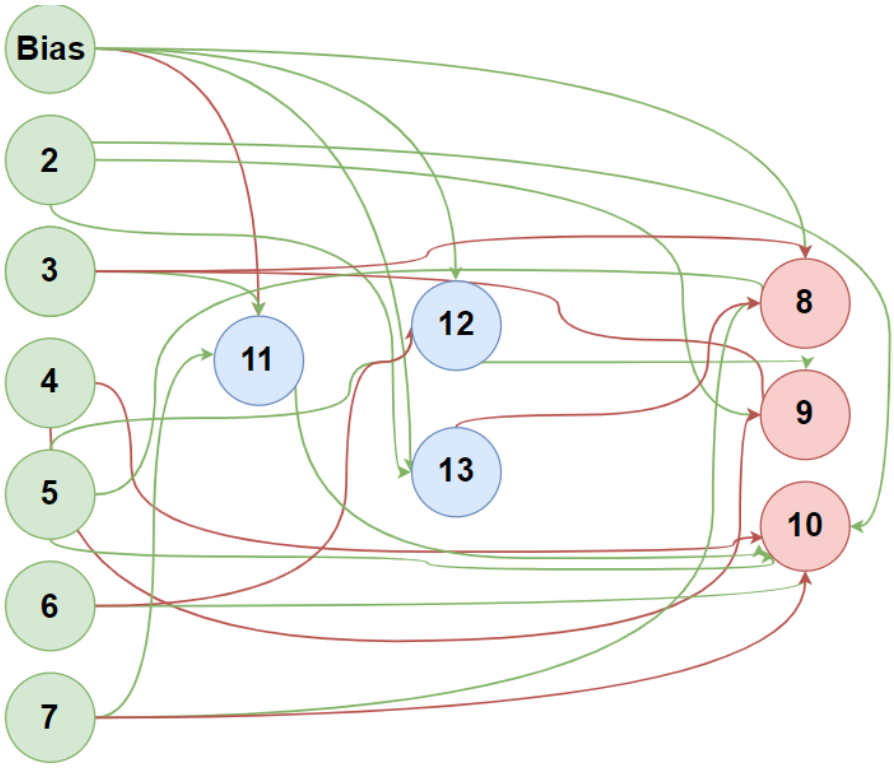

Figura 35. Red para Snake adaptada a obstáculos

Puede observarse que la red resultante es bastante más compleja, en términos de conexiones. Ahora, los nodos 2,3 y 4 cobran importancia, y NEAT potencia dichos nodos con conexiones salientes de dichas entradas. La conclusión que se puede sacar es lo útil que es no partir de redes con todas las entradas y salidas conectadas, porque puede haber entradas que, aunque se piense que son importantes, en realidad no lo son. En cuanto se fuerza una situación en la que dichas entradas necesitan tenerse en cuenta, NEAT se adapta para crear estructuras nuevas que incorporen dicha información.

6 Conclusiones y trabajo futuro

6.1 Conclusiones

NEAT es un algoritmo increíblemente potente y flexible. Sobre todo, es especialmente interesante cuando se enfrenta a problemas "en tiempo real". En problemas clásicos tampoco funciona mal. Quizás no ofrezca la enorme precisión que se puede obtener con otros tipos de sistemas construidos manualmente, pero si el tiempo para generar las poblaciones no es un problema, y dicho problema no requiere de una precisión extremadamente alta, NEAT es, sin duda, una herramienta a tener en cuenta, pues automatiza el proceso de buscar estructuras de redes que funcionen bien, y garantiza que dichas estructuras no tienen apenas elementos inútiles, dado que dichas redes parten de estructuras simples. Otra de las ventajas que ofrece es el hecho de que proporciona varias posibles soluciones al problema, y no se limita a buscar solo una. Además, en esas soluciones, se pueden ver patrones o ideas que inicialmente no se habían pensado, pero que deberían tenerse en cuenta. Por ejemplo, atributos de entrada que a primera vista pueden parecer indispensables, pero luego resultan innecesarios. En definitiva, incrementa el radio de soluciones a tener en cuenta.

Dejando a un lado la utilidad de NEAT, sin duda es innegable que es un algoritmo que ofrece un amplio abanico de posibilidades interesantes, sobre todo en campos relacionados con robótica o juegos, además de que es un algoritmo que ha influenciado en gran medida a otros sistemas más modernos, hasta el punto en que NEAT sigue siendo relevante hoy en día, después de 17 años.

Desde el punto de vista de los objetivos propuestos en este proyecto se puede concluir que se ha implementado un algoritmo a partir de los datos facilitados en el artículo que lo proponía, al tiempo que se han analizado y evaluado sus posibilidades en campos diversos, pertenecientes a diferentes dominios. En todos los casos, sus aportaciones resultan relevantes por lo que ofrecen respecto a la automatización del diseño de estructuras complejas como las redes neuronales.

6.2 Trabajo futuro

En el trabajo se ha mostrado como NEAT se defiende frente a problemas de varios tipos relativamente simples. Sería especialmente interesante utilizar lo aprendido para poner a NEAT a prueba con problemas mucho más grandes y complejos. De forma personal, mi intención es usar NEAT para resolver juegos más grandes, donde el individuo pueda elegir entre varias acciones y se pueda visualizar las estrategias que aprende.

En este trabajo se ha implementado la versión estándar de NEAT, con algunas modificaciones, pero también puede considerarse extenderlo, ya sea imitando algunas variantes ya existentes, como HyperNEAT, o modificándolo para conseguir optimizarlo frente a problemas específicos, creando de esta forma una aportación propia.

Referencias

[1] Kenneth O. Stanley & Risto Miikkulainen (2002). "*Evolving neural networks though aumenting topologies*". **10** (2): 99–127

[2] Erin J. Hastings, Ratan K. Guha, and Kenneth O. Stanley (2009). "Automatic Content Generation in the Galactic Arms Race Video Game ". IEEE Transactions on Computational Intelligence and AI in Games, volume 4, number 1, pages 245-263, New York: IEEE Press, 2009.

[3] Silva, Fernando; Urbano, Paulo; Correia, Luís; Christensen, Anders Lyhne (2015-09-15). "odNEAT: An Algorithm for Decentralised Online Evolution of Robotic Controllers". *Evolutionary Computation*. **23** (3): 421–449

[4] Kenneth O. Stanley; Ryan Cornelius; Risto Miikkulainen; Thomas D'Silva & Aliza Gold (2005). "Real-Time Learning in the NERO Video Game"

[5] Green, C. 2004. SharpNEAT software package. http://sharpneat.sourceforge.net

[6] From: http://www.lac.inpe.br/~rafael.santos/Docs/R/CAP394/WholeStory-Iris.html

[7] Stanley, Kenneth O.; D'Ambrosio, David B.; Gauci, Jason (2009-01-14). "A Hypercube-Based Encoding for Evolving Large-Scale Neural Networks". *Artificial Life*. **15** (2): 185–212

[8] Iris Dataset: https://archive.ics.uci.edu/ml/datasets/iris

[9] NEAT-Python software package. https://neat-python.readthedocs.io/en/latest/

[10] Snake Fitness Design: https://becominghuman.ai/designing-ai-solving-snake-with-evolution-f3dd6a9da867

[11] Y. Kassahun and G. Sommer. Efficient reinforcement learning through evolutionary acquisition of neuraltopologies.

Glosario

NEAT	NeuroEvolution of Augmenting Topologies
EANT	Evolutionary Acquisition of neural topologies
DXNN	Deus Ex Neural Network
CMA-HAGA	Covariance Matrix Adaptation with Hypervolume Sorted Adaptive Grid Algorithm
AG	Algoritmo genético