TELECOMUNICACIÓN

Campus Sur

POLITÉCNICA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA Y SISTEMAS DE TELECOMUNICACIÓN

PROYECTO FIN DE GRADO

TÍTULO: APRENDIZAJE AUTOMÁTICO CON ALGORITMOS GENÉTICOS Y REDES NEURONALES

AUTOR: ALBERTO MOLINA CUELLO

TITULACIÓN: GRADO EN INGENIERÍA TELEMÁTICA

TUTOR: ENRIQUE RENDÓN ANGULO

DEPARTAMENTO: DEPARTAMENTO DE INGENIERÍA AUDIOVISUAL Y COMUNICACIONES (IAC)

VºBº

Miembros del Tribunal Calificador:

PRESIDENTE: JESÚS RODRÍGUEZ MOLINA

TUTOR: ENRIQUE RENDÓN ANGULO

SECRETARIO: MARTINA ECKERT

Fecha de lectura:

Calificación:

El Secretario,

RESUMEN

1. PALABRAS CLAVE

Inteligencia Artificial, Algoritmos Genéticos, Computación Evolutiva, Redes Neuronales Artificiales, Unity.

2. RESUMEN

La inteligencia artificial ha crecido exponencialmente desde sus orígenes alrededor de los años 30. Dentro de la inteligencia artificial encontramos las redes neuronales artificiales, un modelo computacional que surge como analogía de las redes neuronales biológicas, que son capaces de realizar predicciones precisas mediante su entrenamiento con un gran volumen de datos. En este Proyecto de Fin de Grado, se explora el entrenamiento de estas redes neuronales mediante la aplicación de algoritmos genéticos y computación evolutiva, una metodología basada en la teoría de la evolución de Charles Darwin.

El presente Proyecto de Fin de Grado tiene como misión la unión de Unity y estos aspectos de la inteligencia artificial mencionados, mediante el desarrollo de un módulo prototipo que facilita el entrenamiento e incorporación de redes neuronales artificiales. Para el entrenamiento de estas, se hace uso de algoritmos genéticos y computación evolutiva ofreciendo una amplia gama de parámetros: selección de padres, recombinación, mutación, etc. De esta forma, se permite la dotación de inteligencia a diferentes componentes de un proyecto de Unity.

Con la finalidad de facilitar la lectura y comprensión del proyecto, el documento se estructura en tres grandes bloques. En primer lugar, se dispone de una introducción al mismo, junto con los conceptos del estado del arte necesarios para la correcta comprensión. En segundo lugar, se detalla el diseño e implementación del proyecto. Se finaliza con un conjunto de pruebas que verifican el funcionamiento del prototipo junto con las respectivas conclusiones. Al final del documento se incluyen los anexos y apéndices con información adicional.

ABSTRACT

1. KEYWORDS

Artificial Intelligence, Genetic Algorithms, Evolutionary Computing, Artificial Neural Networks, Unity.

2. ABSTRACT

Artificial intelligence has grown exponentially since its origins around the 1930s. Within artificial intelligence we find artificial neural networks, a computational model that emerges as an analogy of biological neural networks, which can make accurate predictions through training with a large volume of data. In this Final Degree Project, the training of these neural networks is explored through the application of genetic algorithms and evolutionary computing, a methodology based on Charles Darwin's theory of evolution.

The purpose of this Final Degree Project is to unite Unity and these aspects of artificial intelligence mentioned before, through the development of a prototype module that facilitates the training and incorporation of artificial neural networks. For their training, genetic algorithms and evolutionary computing are used, offering a wide range of parameters: parent selection, recombination, mutation, etc. In this way, the provision of intelligence to different components of a Unity project is allowed.

To facilitate the reading and understanding of the project, the document is structured in three main blocks. First, there is an introduction to it, along with the concepts of the state of art necessary for correct understanding. Second, the design and implementation of the project is detailed. The document ends with a set of tests that verify the operation of the prototype together with the respective conclusions. Annexes and appendices with additional information are also included at the end of the document.

Índice

RESUMEN 3

1. **PALABRAS CLAVE** 3
2. **RESUMEN** 3

ABSTRACT 5

1. **KEYWORDS** 5
2. **ABSTRACT** 5

ÍNDICE 7

ÍNDICE DE FIGURAS 11

INTRODUCCIÓN 14

1. **Introducción al proyecto desarrollado** 14
2. **Objetivos** 16
3. **Estructura del documento** 18

ESTADO DEL ARTE 19

1. **Redes Neuronales Artificiales** 19
   1. Inteligencia Artificial 19
   2. Elemento básico del cerebro humano: la neurona 20
   3. Perceptrón 21
   4. Perceptrón de Varias Capas 24
2. **Computación Evolutiva** 33
   1. Introducción 33
   2. Representación de los individuos 34
   3. Función Fitness 35
   4. Recombinación 36
   5. Mutación 40
   6. Gestión de la Población 43
   7. Selección de Padres 43
   8. Selección de Supervivientes 48
   9. Algoritmos Genéticos 49
3. **Aplicación de Algoritmos Genéticos en Redes Neuronales** 51
4. **NEAT** 54
   1. Introducción 54
   2. Red neuronal inicial 54
   3. Representación del genotipo 55

4. Mutación 57
5. Recombinación 58

DISEÑO E IMPLEMENTACIÓN 60

1. Introducción 60

2. Restricciones de diseño 63
1. Restricciones de aplicación 63
2. Otras restricciones 63

3. Implementación 65
1. Funcionamiento del módulo 65
2. Ficheros generados 66
3. Panel principal 69
4. Panel de opciones básicas 73
5. Panel de selección de posición inicial 74
6. Panel de selección de padres 76
7. Panel de opciones de cruzamiento 77
8. Panel de opciones de mutación 78
9. Panel de opciones de tiempo 80
10. Panel de opciones de gestión de la población 81
11. Panel de información de la simulación 82

4. Estructura del proyecto 83
1. Clases principales 83
2. Clases de redes neuronales simples 85
3. Clases de redes neuronales NEAT 87
4. Clases para la gestión de agentes muertos 87
5. Clases para la recombinación de redes 87
6. Clases para la mutación de redes 88
7. Clases para la selección de padres 88
8. Clases para la lectura de triggers de mutación 89
9. Clases para el manejo de números binarios 89
10. Otras clases 90

PRUEBAS Y RESULTADOS I: PRUEBAS DE CONVERGENCIA 91

1. Introducción 91

2. Pruebas de recombinación binaria 95
1. Prueba 1: cruzamiento en un punto 95
2. Prueba 2: cruzamiento en n-puntos 96
3. Prueba 3: cruzamiento uniforme 96
4. Conclusiones parciales 97

3. Pruebas de recombinación flotante 98
1. Prueba 4: recombinación discreta 98
2. Prueba 5: recombinación aritmética simple 99
3. Prueba 6: recombinación aritmética individual 99
4. Prueba 7: recombinación aritmética completa 100
5. Conclusiones parciales 100

4. Pruebas de mutación discreta 102
1. Prueba 8: mutación uniforme 102

2. Prueba 9: mutación no uniforme 102
3. Conclusiones parciales 103

5. Pruebas de selección de padres 104
1. Prueba 10: Selección basada en el fitness 104
2. Prueba 11: Selección basada en el ranking (lineal) 104
3. Prueba 12: Selección basada en el ranking (exponencial) 105
4. Conclusiones parciales 105

PRUEBAS Y RESULTADOS II 108

1. Introducción 108

2. Resultados 112

PRESUPUESTO 113

CONCLUSIONES 115

TRABAJOS FUTUROS 117

1. Diferentes escenarios en una misma simulación 117

2. Elementos interactuables con los individuos 120

3. Incorporación de redes neuronales NEAT 121

REFERENCIAS 123

ANEXO I: EJEMPLO DE FICHERO COMPLETO INPUT.XML 125

ANEXO II: EJEMPLO DE FICHERO COMPLETO COMMANDS.XML 128

ANEXO III: EJEMPLO DE FICHERO COMPLETO MUTATIONTRIGGERS.XML 132

ANEXO IV: EJEMPLO FICHERO COMPLETO SIMULATIONRESUME.CSV 134

ANEXO V: EJEMPLO DE FICHERO COMPLETO DE RED NEURONAL 136

APÉNDICE A: MANUAL DE USUARIO 138

1. Introducción 138

2. Procedimiento 139
1. Aplicación de ejemplo 139
2. Descarga e importación del modulo 141
3. Adaptación de GeneticBehaviour.cs 143
4. Estados del individuo y fitness 146
5. Completar el objeto GeneticManager 148

6. Ajuste de la cámara 149
7. Arrancar la aplicación 149
8. Utilización de una red neuronal entrenada..152

ÍNDICE DE FIGURAS

FIGURA 1: ESQUEMA GENERAL DE LA UTILIZACIÓN DEL MÓDULO 14
FIGURA 2: ESQUEMA DEL FUNCIONAMIENTO OBJETIVO DEL MÓDULO 16
FIGURA 3: VEHÍCULOS DE BRAITENBERG BASADOS EN [1] 19
FIGURA 4: MODELO BIOLÓGICO DE INTERCONEXIÓN NEURONAL [3] 21
FIGURA 5: ESQUEMA BÁSICO DE UN PERCEPTRÓN 22
FIGURA 6: PERCEPTRÓN DE EJEMPLO 23
FIGURA 7: FUNCIÓN AND (LINEALMENTE SEPARABLE) Y FUNCIÓN XOR (NO LINEALMENTE SEPARABLE) [6] 23
FIGURA 8: RED NEURONAL CON DOS CAPAS OCULTAS 24
FIGURA 9: FUNCIÓN SIGMOIDE PARA T = 1, T = 2 Y T = 3 26
FIGURA 10: FUNCIÓN SIGMOIDE PARA Θ = 0, Θ = 1 Y Θ = -1 26
FIGURA 11: RED NEURONAL DE 3 NODOS DE ENTRADA, 1 CAPA OCULTA CON 3 NODOS, Y 2 NODOS DE SALIDA 27
FIGURA 12: ERROR CUADRÁTICO MEDIO PARA UNA RED NEURONAL DE DOS PESOS [8] 30
FIGURA 13: ESQUEMA DE ALGORITMO EVOLUTIVO 33
FIGURA 14: CRUZAMIENTO EN UN PUNTO 37
FIGURA 15: CRUZAMIENTO EN N-PUNTOS 37
FIGURA 16: CRUZAMIENTO UNIFORME 38
FIGURA 17: RECOMBINACIÓN DISCRETA 38
FIGURA 18: RECOMBINACIÓN ARITMÉTICA SIMPLE 39
FIGURA 19: RECOMBINACIÓN ARITMÉTICA INDIVIDUAL 40
FIGURA 20: RECOMBINACIÓN ARITMÉTICA COMPLETA 40
FIGURA 21: EXTRACTO DE LA TABLA DE PROBABILIDADES DE DISTRIBUCIÓN NORMAL ESTÁNDAR [11] 42
FIGURA 22: EJEMPLO SELECCIÓN EN FUNCIÓN DEL FITNESS 44
FIGURA 23: RANKING LINEAL PARA S = 1.01, S = 1.5 Y S = 2 45
FIGURA 24: RANKING EXPONENCIAL PARA Q = 0.75, Q = 0.25 Y Q = 0.1 46
FIGURA 25: EJEMPLO DE PROBABILIDADES EXPONENCIALES CON BAJA POBLACIÓN 47
FIGURA 26: COMPARACIÓN DE PROBABILIDADES CON Y SIN AJUSTAR, PARA Q = 0.1 Y M = 10 48
FIGURA 27: EJEMPLO DE REPRESENTACIÓN FLOTANTE DE UNA RED NEURONAL 52
FIGURA 28: RED NEURONAL INICIAL AL APLICAR NEAT, CON 5 NODOS DE ENTRADA Y 3 DE SALIDA 55
FIGURA 29: EJEMPLO DE GENOTIPO DE RED NEURONAL BASADO EN [15] 56
FIGURA 30: RED NEURONAL EQUIVALENTE AL GENOTIPO DE LA FIGURA ANTERIOR 57
FIGURA 31: CREACIÓN DE UN NUEVO NODO MEDIANTE NEAT 57
FIGURA 32: CREACIÓN DE UNA NUEVA CONEXIÓN MEDIANTE NEAT 58
FIGURA 33: CRUZAMIENTO DE DOS GENOTIPOS DE REDES NEURONALES CON TOPOLOGÍAS DIFERENTES [17] 59
FIGURA 34: ESCENARIO DE EJEMPLO 61
FIGURA 35: DIAGRAMA DE FLUJO DEL FUNCIONAMIENTO DEL MÓDULO 65
FIGURA 36: CARPETA SIMULATION DEL PROYECTO 67
FIGURA 37: CARPETA SIMULATION/SIMULATION_X DEL PROYECTO 68
FIGURA 38: FRAGMENTO XML DE UN FICHERO DE RED NEURONAL 69
FIGURA 39: PANEL PRINCIPAL 70
FIGURA 40: CUADRO DE SELECCIÓN 70
FIGURA 41: ETIQUETA COMMAND 72
FIGURA 42: FRAGMENTO DE ETIQUETA COMMAND 72
FIGURA 43: PANEL DE OPCIONES BÁSICAS 73
FIGURA 44: MENSAJE DE PARÁMETRO INCORRECTO 74
FIGURA 45: PANEL DE SELECCIÓN DE POSICIÓN 74
FIGURA 46: EJEMPLO DE POSICIÓN INICIAL SIMPLE CON 5 SOLUCIONES 75
FIGURA 47: FORMATO DE SELECCIÓN DE RANGO DE POSICIONES 75
FIGURA 48: EJEMPLO DE POSICIÓN INICIAL DE RANGO CON 5 SOLUCIONES 75
FIGURA 49: FORMATO DE SELECCIÓN DE RANGO DE ROTACIÓN 76
FIGURA 50: EJEMPLO DE ROTACIÓN INICIAL DE RANGO CON UN ÁNGULO ENTRE 315 Y 45 GRADOS, CON 5 SOLUCIONES 76
FIGURA 51: PANEL DE OPCIONES DE SELECCIÓN DE PADRES 76
FIGURA 52: PANEL DE OPCIONES DE CRUZAMIENTO 77
FIGURA 53: PANEL DE OPCIONES DE MUTACIÓN 78
FIGURA 54: ETIQUETA MUTATIONTRIGGERS 79

Figura 55: Etiqueta Trigger ... 79
Figura 56: Panel de opciones de tiempo ... 80
Figura 57: Panel de opciones de gestión de la población ... 81
Figura 58: Panel de información de la simulación ... 82
Figura 59: Comunicación entre CanvasManager.cs y GeneticManager.cs ... 84
Figura 60: Comunicación entre GeneticManager.cs y los agentes ... 85
Figura 61: Escenario de pruebas con agentes ... 91
Figura 62: Rayos de los agentes ... 92
Figura 63: Red neuronal utilizada para PRUEBAS Y RESULTADOS I ... 93
Figura 64: Resultados prueba 1: cruzamiento en un punto ... 95
Figura 65: Resultados prueba 2: cruzamiento en n puntos ... 96
Figura 66: Resultados prueba 3: cruzamiento uniforme ... 97
Figura 67: Comparación pruebas 1, 2 y 3 ... 97
Figura 68: Resultados prueba 4: recombinación discreta ... 99
Figura 69: Resultados prueba 5: recombinación aritmética simple ... 99
Figura 70: Resultados prueba 6: recombinación aritmética individual ... 100
Figura 71: Resultados prueba 7: recombinación aritmética completa ... 100
Figura 72: Comparación pruebas 4, 5, 6 y 7 ... 101
Figura 73: Resultados prueba 9: mutación no uniforme ... 103
Figura 74: Comparación pruebas 8 y 9 ... 103
Figura 75: Resultados prueba 11: selección basada en la ranking lineal (s = 2) ... 105
Figura 76: Resultados prueba 12: selección basada en el ranking exponencial (q = 0,5) ... 105
Figura 77: Comparación pruebas 10, 11 y 12 ... 106
Figura 78: Escenario utilizado en PRUEBAS Y RESULTADOS II ... 108
Figura 79: Red neuronal utilizada para PRUEBAS Y RESULTADOS II ... 109
Figura 80: Resultados prueba 13 ... 112
Figura 81: Comunicación entre los elementos del módulo tras incorporar GeneticManagerNEAT ... 122
Figura 82: Fichero input.xml completo ... 126
Figura 83: Notificación si el fichero input.xml es erróneo ... 127
Figura 84: Fichero xml de ejemplo de comandos de simulación ... 130
Figura 85: Fichero xml de ejemplo de triggers de mutación ... 132
Figura 86: Pseudocódigo del ejemplo de triggers de mutación ... 133
Figura 87: Fichero simulationResume.csv ... 135
Figura 88: Fichero xml completo de red neuronal ... 137
Figura 89: Aplicación de ejemplo ... 139
Figura 90: Prefab de las soluciones del algoritmo ... 140
Figura 91: Prefab del objetivo ... 140
Figura 92: Prefab de los obstáculos ... 140
Figura 93: Prefab del delimitador del mapa ... 141
Figura 94: Importar módulo ... 142
Figura 95: Escena GeneticModule ... 142
Figura 96: Scripts importados ... 143
Figura 97: GeneticModule en la jerarquía ... 143
Figura 98: Componente GeneticBehaviour.cs ... 144
Figura 99: Seleccionar la topología de la red neuronal ... 144
Figura 100: Ejemplo de bloques USER CODE ... 145
Figura 101: Método DecideDistance() ... 145
Figura 102: Inicialización de otros componentes en GeneticBehaviour.cs ... 146
Figura 103: Script Detector.cs ... 147
Figura 104: Script Detector.cs editado ... 147
Figura 105: Obtener GeneticBehaviour ... 148
Figura 106: Objeto GeneticManager con el prefab del componente a entrenar ... 149
Figura 107: Panel de selección de parámetros ... 150
Figura 108: Simulación arrancada (generación 1) ... 150
Figura 109: Primeros dos individuos en alcanzar el objetivo (generación 6) ... 151
Figura 110: Convergencia obtenida (generación 19) ... 152

FIGURA 111: INICIALIZACIÓN DE RED NEURONAL ENTRENADA 153

INTRODUCCIÓN

1. Introducción al proyecto desarrollado

El proyecto de fin de grado desarrollado consiste en un módulo prototipo que tiene como objetivo facilitar a los usuarios del motor de videojuegos Unity la incorporación de inteligencia a los componentes de sus proyectos. Esta inteligencia consiste en redes neuronales artificiales entrenadas por algoritmos genéticos que se conectan al comportamiento de los componentes activos de la aplicación Unity. Este proyecto ha diseñado e implementado un módulo importable en Unity que se encarga de todo el proceso de entrenamiento, proporcionando una red neuronal entrenada de acuerdo con las especificaciones del usuario para conseguir el comportamiento deseado.

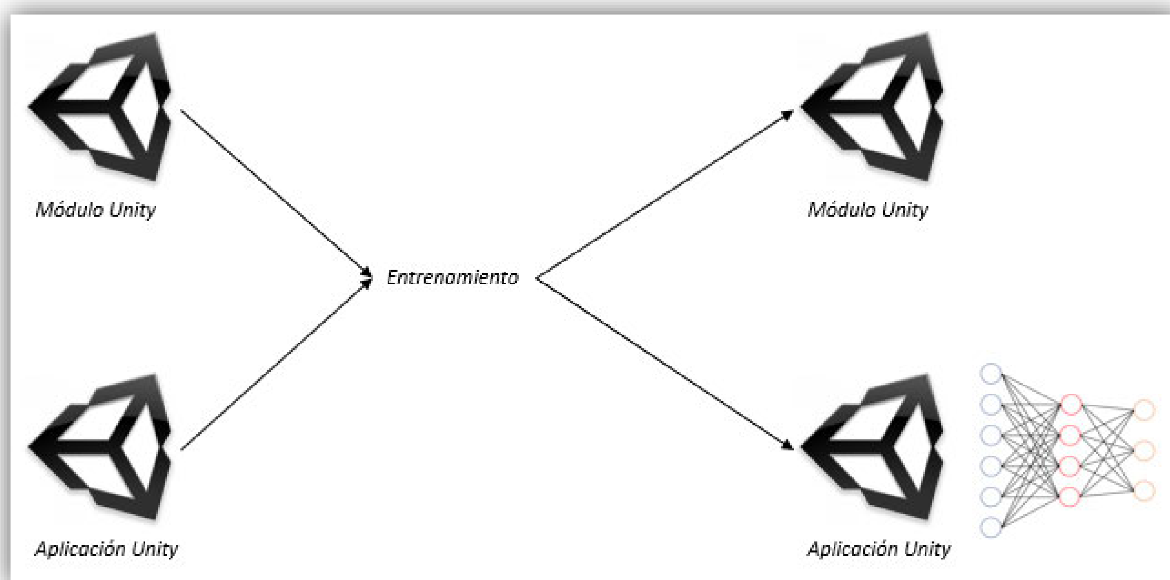

Figura 1: Esquema general de la utilización del módulo

Para realizar esta importación de forma correcta se detalla un manual de usuario en el *APÉNDICE A*. Una vez finalizado el entrenamiento, se puede incorporar la red neuronal entrenada para que se encargue de manejar el componente seleccionado.

Este proyecto es un prototipo que prueba que el concepto funciona y valida un conjunto de herramientas para la aplicación de redes neuronales entrenadas por algoritmos genéticos dentro de Unity. Por tratarse de un prototipo tiene limitaciones que se detallan en el apartado de *DISEÑO E IMPLEMENTACIÓN*. A su vez abre muchas posibilidades de extensión y mejoras futuras, que se detallan en el apartado *TRABAJOS FUTUROS*.

2. Objetivos

A continuación, se detallan los objetivos esperados que debe cumplir el proyecto finalizado. En primer lugar, los pasos generales para la utilización del módulo deben ser los siguientes:

1. El usuario programa su propia aplicación de Unity.
2. Genera una versión de la aplicación adaptada al módulo e incorpora el mismo.
3. Realiza el entrenamiento, obteniendo las redes neuronales entrenadas.
4. Utiliza estas redes neuronales en su aplicación.

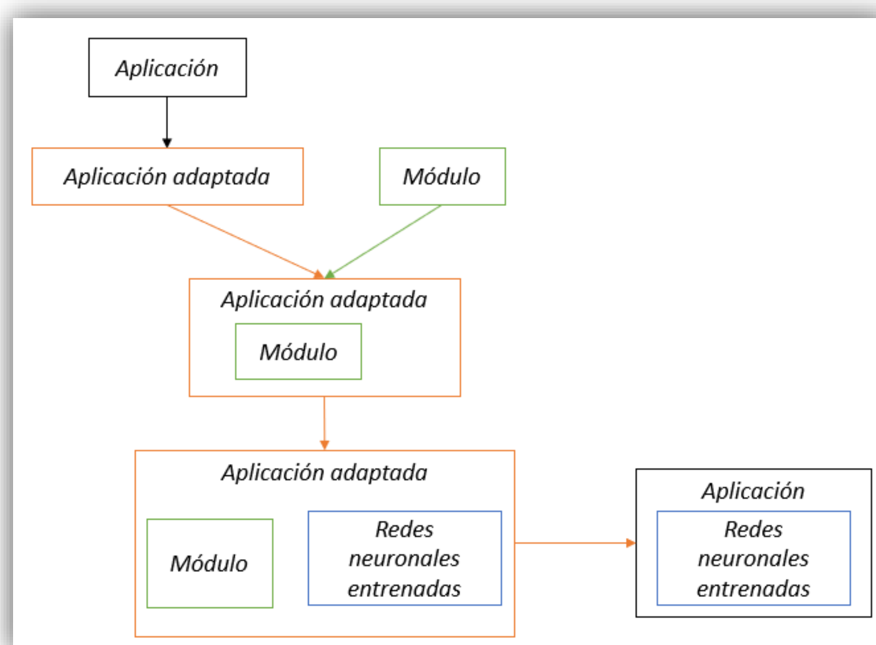

Figura 2: Esquema del funcionamiento objetivo del módulo

Para conseguir el entrenamiento de las redes neuronales se utilizará una combinación de redes neuronales y algoritmos genéticos. Como se detalla en los siguientes apartados, los algoritmos genéticos son una rama específica de la computación evolutiva. Puesto que, en función de la aplicación a entrenar, puede ser conveniente utilizar unos parámetros u otros, el módulo no implementa únicamente algoritmos genéticos, sino que también contempla otras posibilidades de la computación evolutiva.

El módulo por desarrollar y las clases relacionadas deben cumplir también los siguientes objetivos:

- Proporcionar una interfaz simple y clara, que permita su utilización de forma sencilla.
- Disponer de un correcto control de parámetros. Puesto que el usuario puede introducir de forma accidental parámetros incoherentes, debe implementar un exhaustivo control de parámetros junto con un sistema de notificaciones que permitan al usuario percatarse del error.
- Almacenar las redes neuronales, y proporcionar un método que permita cargar las redes almacenadas. De esta forma el usuario podrá incluir dichas redes en sus proyectos finales.
- Crear un fichero de log, csv o con extensión similar que permita obtener información a posteriori del proceso de aprendizaje.
- Permitir al usuario automatizar el proceso de entrenamiento, mediante la posibilidad de ejecutar varias simulaciones seguidas.

Uno de los principales objetivos de este módulo consiste en que la carga de programación que debe experimentar el usuario debe ser la menor posible. Por esta razón, el usuario sólo debe programar la adaptación de la aplicación al módulo. Para facilitar este proceso de unión módulo-aplicación, se incluye un manual de usuario en *APÉNDICE A: MANUAL DE USUARIO*.

3. Estructura del documento

El documento presente se estructura de la siguiente forma:

- En primer lugar, encontramos el capítulo *ESTADO DEL ARTE*. En este capítulo se comienza con la descripción de conceptos relacionados con las redes neuronales artificiales como el concepto de perceptrón o perceptrón multicapa. A continuación, se detalla de forma extendida los aspectos relacionados con la computación evolutiva y los algoritmos genéticos. Por último, dos apartados que introducen brevemente a la relación de redes neuronales con algoritmos genéticos y al método NEAT.

- En segundo lugar, tenemos un capítulo dedicado al proyecto desarrollado llamado *DISEÑO E IMPLEMENTACIÓN*. Se comienza describiendo las restricciones de diseño utilizadas para la realización de este. Después, se describe cómo se ha implementado el proyecto, junto con explicaciones de las clases que forman el mismo.

- Después, se detallan un conjunto de pruebas que permiten observar la funcionalidad del proyecto desarrollado mediante la ejecución de varias simulaciones con diferentes parámetros.

- A continuación, se describen unos posibles trabajos futuros junto con las conclusiones del proyecto.

- Por último, encontramos las referencias junto con los anexos y apéndices.

ESTADO DEL ARTE

1. Redes Neuronales Artificiales

1. Inteligencia Artificial

El concepto de inteligencia artificial es difícil de definir de forma precisa. Uno de los grandes pioneros de la inteligencia artificial, Jhon McCarthy describió el objetivo de la inteligencia artificial como el desarrollo de sistemas que se comporten como si fueran inteligentes. El experimento conocido como los vehículos de Braitenberg, desarrollado por el neurocientífico Valentino Braitenberg, se puede considerar la forma más básica de inteligencia artificial cognitiva, cumpliendo con la descripción de McCarthy.

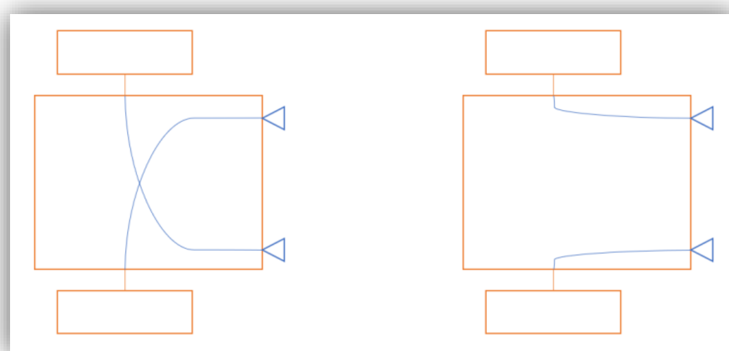

Figura 3: Vehículos de Braitenberg basados en [1]

Los vehículos disponen de dos motores (uno para la rueda derecha y otro para la izquierda) permitiendo recorrer el espacio libremente. Los motores están conectados a sensores de luz, de tal forma que, cuanta más luz detecte, más movimiento proporcionarán. Por lo tanto, ante los dos vehículos de la figura anterior, podemos deducir que ambos mostrarán un comportamiento inteligente, uno de ellos evitando la luz y el otro yendo hacia la misma.

El principal objetivo de la inteligencia artificial consiste en el desarrollo de soluciones que sean capaces de resolver una gran variedad de problemas. Para conseguir esto, existen diferentes tipos de enfoques: minería de datos, redes neuronales artificiales, aprendizaje por refuerzo, etc.

Las redes neuronales artificiales son una rama de la computación inspirada en el modelo biológico del sistema nervioso humano. En los siguientes apartados, detallaremos los aspectos más importantes de las redes neuronales, comenzando por una breve descripción de la unidad fundamental del del cerebro humano, la neurona, continuando con el modelo matemático de una neurona, el perceptrón, y finalizando con el análisis de las redes neuronales. [2]

2. Elemento básico del cerebro humano: la neurona

La gran mayoría de ramas de la inteligencia artificial buscan realizar las funciones propias de la inteligencia natural, como el razonamiento lógico o el razonamiento probabilístico, pero se modelan utilizando herramientas que no tienen ninguna similitud con el cerebro humano. Las redes neuronales artificiales, por el contrario, son una rama de la inteligencia artificial que se modela basándose en el conocimiento de este. A continuación, procederemos a explicar resumidamente la estructura y funcionamiento de la unidad básica cerebro humano, la neurona, para poder comprender correctamente las redes neuronales artificiales.

El cerebro humano está compuesto por aproximadamente 10^{11} neuronas. La neurona es la célula nerviosa que, mediante la interconexión con otras neuronas, permite al ser humano llevar a cabo el proceso de aprendizaje, adaptación, percepción, pensamiento, conciencia, etc.

Una neurona está compuesta principalmente por los siguientes elementos:

1. Cuerpo de la célula.
2. Dendritas.
3. Axón.

Cada neurona está conectada con aproximadamente de 1000 a 10000 otras neuronas. Por lo tanto, en el cerebro humano existen aproximadamente 10^{14} conexiones. Estas conexiones se denominan sinapsis, y es el punto de unión entre un axón y una dendrita. Los axones actúan como una línea terminal, propagando la información a las dendritas de otras neuronas, que actúan como líneas receptoras.

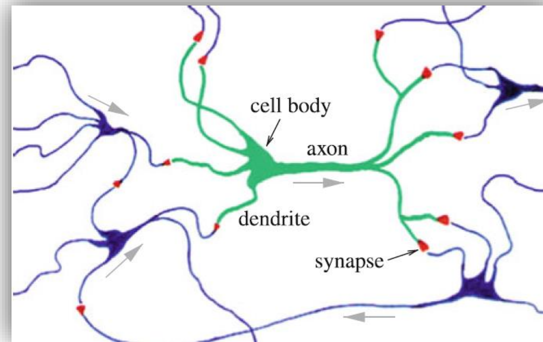

Figura 4: Modelo biológico de interconexión neuronal [3]

La información que manejan las neuronas consiste en pequeños impulsos eléctricos que viajan entre ellas. El cuerpo de la neurona es capaz de almacenar carga eléctrica, actuando como un condensador. Esta carga eléctrica viajará por el axón, estimulando las dendritas que estén conectadas al mismo. Es importante mencionar que no toda carga eléctrica producirá un estímulo, esto dependerá de la sinapsis que une el axón con cada dendrita.

La sinapsis no es una unión completa, en cuanto a conductividad se refiere. Existe una separación que impide el paso directo de electrones entre ambos extremos. Esta separación está compuesta de diferentes sustancias químicas conocidas como neurotransmisores. Para poder transportar carga, estos neurotransmisores deben ser primero ionizados mediante la aplicación de voltaje. Por lo tanto, el impulso eléctrico debe ser en primer lugar capaz de ionizar los neurotransmisores, para seguidamente transmitir la carga a través de la sinapsis.

El proceso de aprendizaje se lleva a cabo gracias a la adaptación de las sinapsis, modificando su conductividad para potenciar una conexión o debilitarla. De tal forma que, una conexión con una alta conectividad será capaz de transmitir pulsos eléctricos más fácilmente que otra con baja conectividad. [2] [4] [5]

3. Perceptrón

El perceptrón es la representación matemática de una neurona biológica. Está formada por un nodo de salida y un número indeterminado de nodos de entrada:

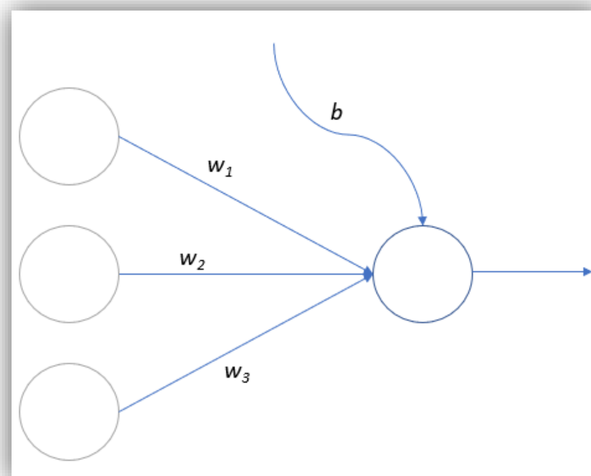

Figura 5: Esquema básico de un perceptrón

Estos nodos de entrada sólo aportan la información de entrada al perceptrón, no realizan ningún cálculo sobre los datos. La salida del perceptrón se calcula de la siguiente forma:

$$y = (\sum_{i=1}^{n} w_i x_i) + b$$

Siendo x_i el valor de entrada del nodo i, y w_i el peso del enlace. El bias (b) es equivalente a disponer de un nodo de entrada, cuyo input siempre es 1. Si el valor de salida es positivo, se iguala a +1. Por el contrario, si es negativo se iguala a -1. Por lo tanto, la salida del perceptrón se decide mediante una función que activa o desactiva la misma (±1).

Por ejemplo, supongamos que disponemos del siguiente perceptrón:

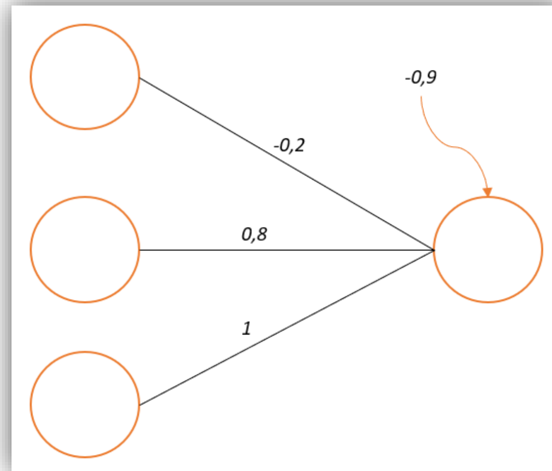

Figura 6: Perceptrón de ejemplo

Ante una entrada *x = (1, 1, 1)* tenemos como salida:

$$y = (-0{,}2) * 1 + 0{,}8 * 1 + 1 * 1 + (-0{,}9) = 0{,}7 \ \rightarrow 1$$

El objetivo de un perceptrón consiste en clasificar un conjunto de entradas en dos clases o grupos: si el valor de salida es +1, pertenecerá al primer grupo, y si es -1 al segundo. Para que un perceptrón pueda realizar la clasificación correctamente, estos deben ser linealmente separables en el espacio. Si observamos una representación de la función AND y la función XOR, podemos observar que solamente la función AND es linealmente separable. Por lo tanto, no podemos implementar un perceptrón que realice la función de XOR.

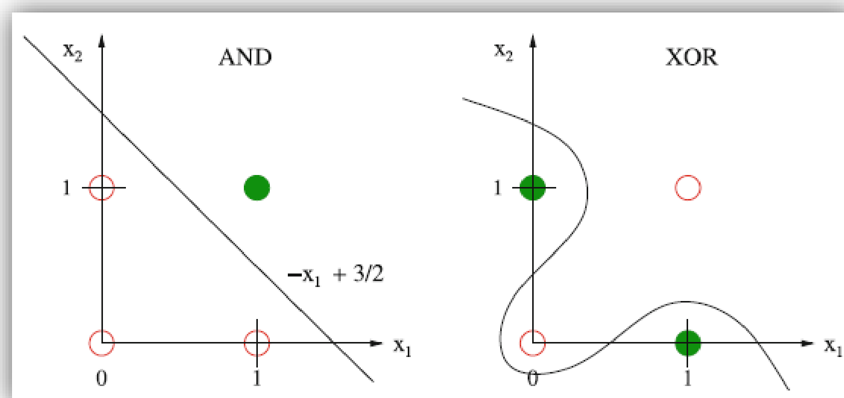

Figura 7: Función AND (linealmente separable) y función XOR (no linealmente separable) [6]

Para poder realizar predicciones más complejas, podemos agrupar varios perceptrones formando redes neuronales como se detalla en el siguiente apartado. [2] [4] [5] [7]

4. Perceptrón de Varias Capas

En el apartado anterior detallamos los aspectos de un perceptrón. Un perceptrón de varias capas es uno de los tipos más comunes de redes neuronales artificiales y concretamente la que se utiliza en este proyecto. Su estructura es la siguiente:

- Una capa de entrada. Al igual que un perceptrón, una red neuronal dispone de un conjunto de nodos de entrada que no realizan ningún proceso sobre los datos, simplemente es el punto de entrada de estos a la red.
- Una o más capas ocultas. Cada capa está compuesta por un número determinado de nodos, en el que cada uno está conectado con cada nodo de la capa anterior. Los nodos de una misma capa no tienen ningún tipo de conexión entre ellos. Habitualmente, el número de capas ocultas suele ser 1, 2 o 3.
- Una capa de salida. Esta capa consiste en un conjunto de nodos en el que cada uno de ellos está conectado con cada nodo de la última capa oculta.

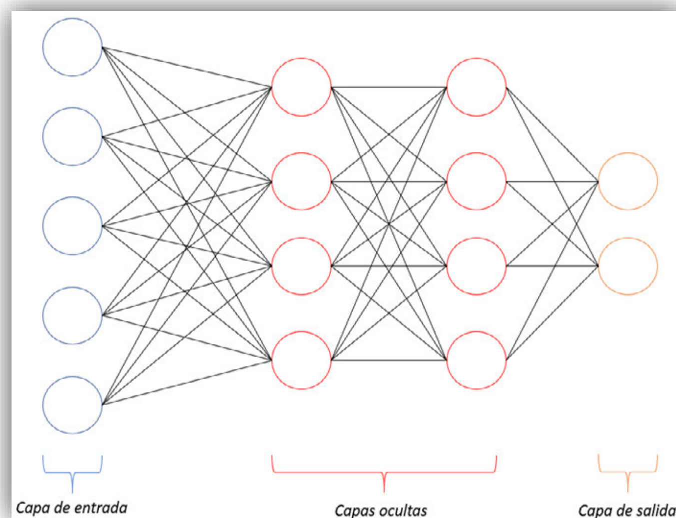

Figura 8: Red neuronal con dos capas ocultas

Uno de los procedimientos para realizar el entrenamiento de un perceptrón de varias capas consiste en la presentación de un gran conjunto de valores de entrada de los cuales ya se conoce la salida. En función de la respuesta de la red ante estos valores, se aplican ciertas correcciones en los pesos. A continuación, se describen brevemente los aspectos más fundamentales de las redes neuronales, utilizados en este tipo de entrenamiento.

Propagación hacia adelante

Sean unos valores de entrada $x = (x_1, x_2, \ldots, x_n)$ de una red neuronal de n nodos de entrada. Para calcular los valores de salida, se aplica el proceso conocido como propagación hacia adelante.

En primer lugar, se calcula el valor de salida de cada nodo de la primera capa intermedia ante los valores de entrada x. Para cada uno de estos valores obtenidos, se aplica una función de transferencia. Esta función de transferencia permite que los valores de salida de cada nodo estén acotados. La función de transferencia más utilizada es la función sigmoide:

$$f(x) = \frac{1}{1 + e^{-\frac{x-\theta}{T}}}$$

Siendo:

- x: salida del nodo (entrada a la función sigmoide).
- θ: valor central de la función.
- T: parámetro de curvatura de la función.

Habitualmente se utilizan los valores $(\theta = 0)$ y $(T = 1)$.

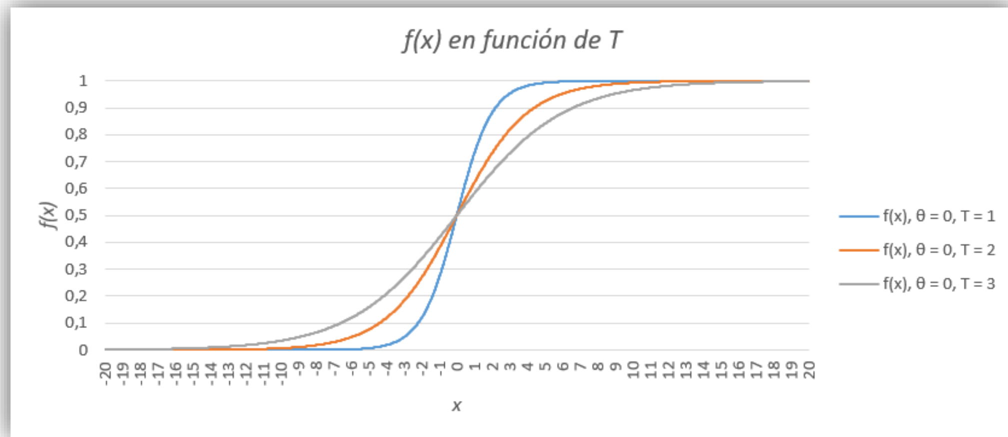

Figura 9: Función sigmoide para T = 1, T = 2 y T = 3

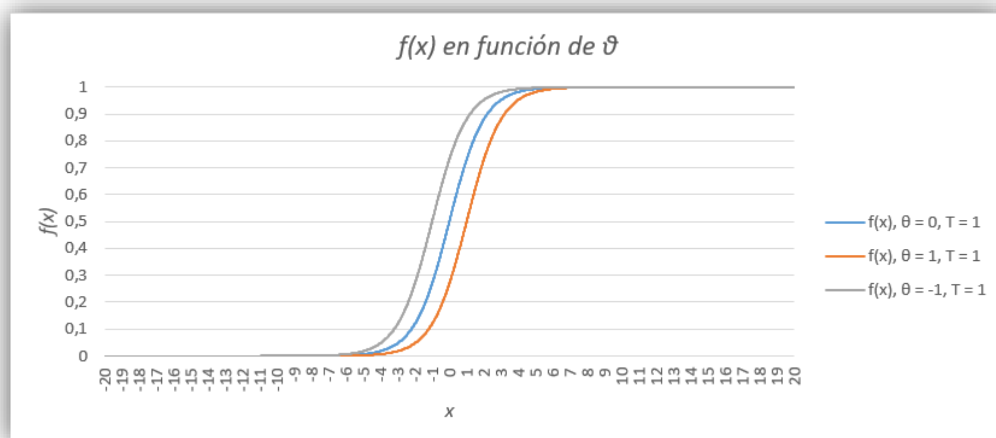

Figura 10: Función sigmoide para θ = 0, θ = 1 y θ = -1

Podemos observar que la salida de los nodos de la primera capa intermedia serán valores comprendidos entre 0 y 1. Estos valores serán utilizados como valores de entrada para la siguiente capa. Así sucesivamente hasta alcanzar la capa de salida. Los valores obtenidos en la última capa oculta son introducidos en la capa de salida, obteniendo los valores salida de la red neuronal.

Al aplicar la función de transferencia en la capa de salida, los valores de salida también estarán comprendidos entre [0, 1]. Finalmente, se activará el nodo de salida cuyo valor sea el mayor de todos.

Por ejemplo, supongamos que disponemos de la siguiente red neuronal:

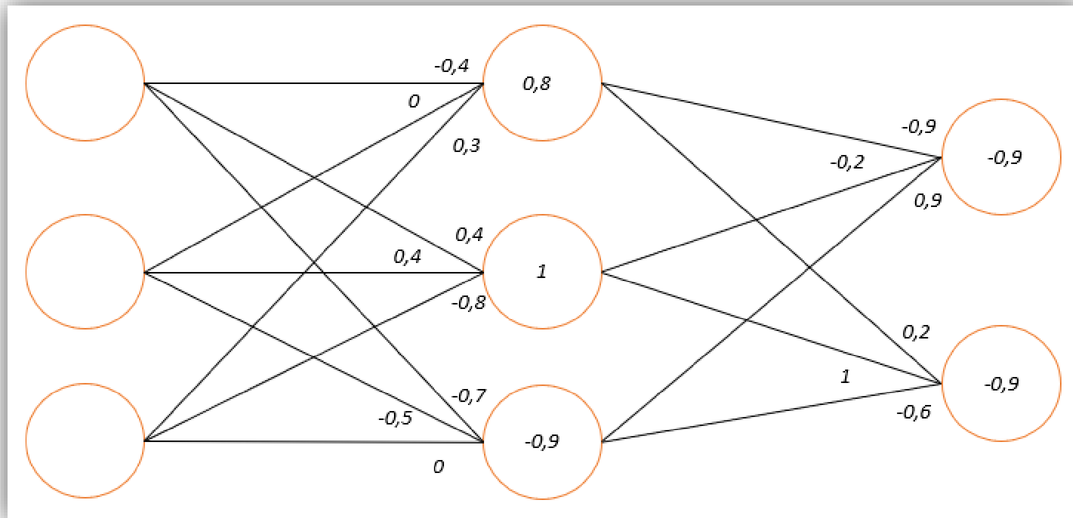

Figura 11: Red neuronal de 3 nodos de entrada, 1 capa oculta con 3 nodos, y 2 nodos de salida

En la imagen anterior, los valores de bias de cada nodo se encuentran en el interior de estos y los pesos se encuentran en las conexiones. Ante una entrada *x = (-1, 0, 1)*, para los nodos de la capa intermedia (con *T = 1* y *θ = 0*) tenemos (redondeando a una cifra decimal):

$$nodo_1 = sigmoid\big((-0{,}4) * (-1) \ + \ 0 * 0 \ + \ 1 * 0{,}3 + 0{,}8\big) = sigmoid(1{,}5) = \ 0{,}8$$

$$nodo_2 = sigmoid(0{,}4 * (-1) + 0{,}4 * 0 + (-0{,}8) * 1 + 1) = \ sigmoid(-0{,}2) = 0{,}5$$

$$nodo_3 = sigmoid\big((-0{,}7) * (-1) + (-0{,}5) * 0 + 0 * 1 + (-0{,}9)\big) = sigmoid(-0{,}2) = \ 0{,}5$$

Una vez calculada la salida de la capa oculta, se calcula la salida de la red neuronal, introduciendo en la capa de salida los valores *x = (0.8, 0.5, 0.5)*:

$$s_1 = sigmoid\big((-0{,}9) * 0{,}8 + (-0{,}2) * 0{,}5 \ + \ 0{,}9 * 0{,}5 \ + (-0{,}9)\big) = sigmoid(-1{,}3) = 0{,}2$$

$$s_2 = sigmoid\big(0{,}2 * 0{,}8 \ + 1 * 0{,}5 \ + (-0{,}6) * 0{,}5 \ + (-0{,}9)\big) = sigmoid(-0{,}5) = \ 0{,}4$$

Ante estas salidas, se consideraría como activado el segundo nodo de salida de la red neuronal.

Tasa de error

Cuando entrenamos una red neuronal mediante un conjunto de datos, de los cuales ya se conoce el resultado, es posible que la red neuronal no acierte todos los valores de este. Ante esta situación, se define el concepto de tasa de error.

La tasa de error es la relación entre el número de errores cometidos por la red neuronal y el número total de valores. Por ejemplo, si se utiliza un data set de 1000 valores, y la red devuelve un resultado erróneo de 30 de ellos, la tasa de error es:

$$tasa_{error} = \frac{fallos}{total} = \frac{30}{1000} = 0{,}03$$

El objetivo de un entrenamiento de una red neuronal consiste en obtener una tasa de error nula (o prácticamente nula).

Vector objetivo

Como se pudo observar en el ejemplo del apartado *Propagación hacia adelante*, la salida de la red neuronal es un conjunto de valores que activan el nodo que obtenga el mayor de ellos. Se considera vector objetivo al conjunto de valores que debe aportar la red neuronal como respuesta para que esta se considere perfecta. Esta respuesta debe estar formada únicamente por 0 o 1.

Volviendo al ejemplo mencionado, ante la entrada $x = (-1, 0, 1)$ hemos obtenido la salida $y = (0.2, 0.4)$, activándose el segundo nodo de salida. El vector objetivo (*target vector*) de esta salida sería $t = (0, 1)$.

Error cuadrático medio

El error cuadrático medio (*Mean Square Error*) consiste en la diferencia entre los valores de salida reales de la red y el vector objetivo de esa salida:

$$MSE = \frac{1}{m} \sum_{i=1}^{m} (t_i - y_i)^2$$

Siendo *m* el número de valores de salida. El error cuadrático medio del ejemplo sería:

$$MSE = \frac{1}{2}((0-0{,}2)^2 + (1-0{,}4)^2) = 0{,}2$$

Propagación hacia atrás

En un entrenamiento, por cada elemento que se introduce en la red neuronal y se comprueba su salida, es necesario aplicar un ajuste en los pesos y los bias de la red para que mejore su funcionamiento. Este ajuste se denomina propagación hacia atrás. En este apartado se detalla el procedimiento utilizado para las redes neuronales que disponen de una capa de entrada, una oculta y una de salida.

Al crear la red neuronal, se inicializa con valores aleatorios, generalmente pequeños, entre 0.1 y -0.1. Durante el entrenamiento estos valores se irán adaptando a los valores correctos que permiten a la red realizar predicciones. Este entrenamiento se basa en el concepto del gradiente descendiente (*gradient descent*). Desde un punto de vista matemático, el entrenamiento consiste en la búsqueda del mínimo local de una función cuyas variables son los pesos de la red neuronal y el resultado es el error cuadrático medio. Esta función es gráficamente irrepresentable, por eso, para facilitar la comprensión de este concepto, es habitual mostrar la gráfica correspondiente a una red neuronal con tan sólo dos pesos (que sí es gráficamente representable):

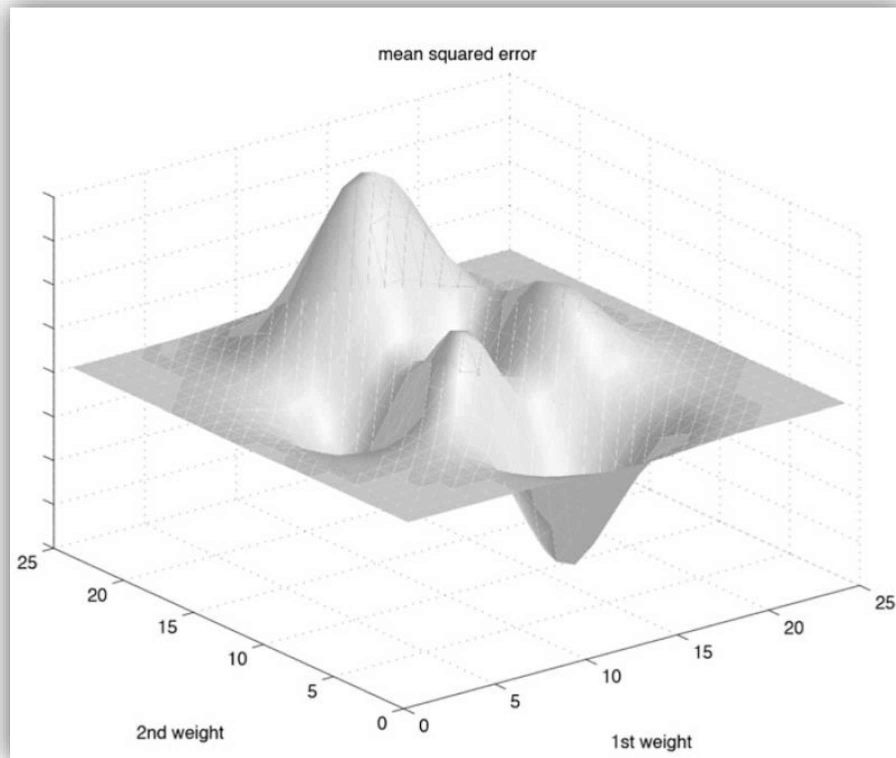

Figura 12: Error cuadrático medio para una red neuronal de dos pesos [8]

Como podemos ver en la imagen, el conjunto de valores de los pesos da como resultado diferentes valores del error cuadrático medio. Al encontrar el mínimo de la función, estamos encontrando los valores de la red neuronal que minimizan el error de predicción, por lo tanto, los valores óptimos.

Cuando una red neuronal comete un error al realizar una predicción, se debe aplicar una corrección de los pesos, pero no todos los pesos han tenido la misma presencia en ese error. Por lo tanto, se debe aplicar una corrección en función de la contribución de cada uno, δ_i. Las fórmulas de dicha contribución varían en base a la función de transferencia utilizada. En el caso de la función sigmoide, son:

$$nodos\ de\ la\ capa\ de\ salida \rightarrow \delta_i = y_i(1 - y_i)(t_i - y_i)$$

Si analizamos la fórmula, el término $(t_i - y_i)$ nos indica la diferencia entre el valor de salida y el valor del vector objetivo. Por otro lado $y_i(1 - y_i)$ permite variar el resultado final de δ_i en función de si la salida tiene un valor extremo o no. Por lo tanto, si el valor de salida de la red

neuronal es un valor extremo (casi 0 o casi 1) se multiplicará por un valor próximo a 0, obteniendo una contribución menor. Si la salida es un valor medio como 0.5, se multiplicará por un valor mayor, obteniendo un δ_i mayor. La fórmula de contribución de los nodos de la capa intermedia es:

$$nodos\ de\ la\ capa\ oculta\ \rightarrow \delta_j = h_j(1 - h_j)\sum_i \delta_i w_{ji}$$

Siendo h_j la salida del nodo (equivalente a y_j en la formula anterior), δ_i la contribución del nodo de la capa siguiente y w_{ji} el peso de la conexión que los une a ambos. De esta forma, propagamos las contribuciones obteniendo las de cada nodo (comenzando por la capa de salida).

Una vez calculadas las contribuciones de cada nodo en el error cometido, se procede a actualizar los valores de los pesos:

$$nodos\ de\ la\ capa\ de\ salida\ \rightarrow w_{ji} = w_{ji} + \eta\delta_i h_j$$

$$nodos\ de\ la\ capa\ oculta\ \rightarrow w_{kj} = w_{kj} + \eta\delta_j x_k$$

Por lo tanto, la corrección que se aplica a los pesos es de $\eta\delta_i h_j$ para los nodos de la capa de salida, y de $\eta\delta_j x_k$ para los nodos de la capa oculta. La tasa de aprendizaje (η) es un parámetro que permite al usuario especificar el tamaño de las correcciones que se aplicarán a los pesos.

Una vez conocido el procedimiento de aprendizaje de la red neuronal se pueden detallar los pasos de entrenamiento de una red neuronal. Supongamos que tenemos un data set de 1000 ejemplos con el que deseamos entrenar la red neuronal:

1. Se inicializa la red neuronal con valores aleatorios.
2. Se introduce el primer valor en la red neuronal, obteniéndose un conjunto de valores de salida.
3. Se aplica propagación hacia atrás, adaptando los pesos de la red neuronal.
4. Se repite el proceso desde el paso 2 con el siguiente valor del data set.

Una vez se recorren los 1000 datos del data set, se considera que ha transcurrido una época. Normalmente las redes neuronales se entrenan con un mismo conjunto de datos durante más de una época, y se finaliza el entreno cuando se cumple una determinada condición. [2] [4] [5] [7] [9]

En estos apartados se ha detallado un posible procedimiento para el entrenamiento de las redes neuronales. En este proyecto se aplica un método alternativo, que busca los pesos más adecuados de una red neuronal mediante la aplicación de algoritmos evolutivos. Para su correcta comprensión, en los siguientes apartados se detallan los conceptos necesarios relacionados con este tipo de algoritmos.

2. Computación Evolutiva

1. Introducción

La computación evolutiva es un área de la ciencia de la computación utilizada para la resolución de problemas u optimización de soluciones basada en la teoría de la evolución de Charles Darwin. Ante un determinado problema, las posibles soluciones actúan como seres vivos de la teoría de la evolución. Durante la ejecución del algoritmo, se mantienen activas un conjunto de soluciones conocidas como población. Cuanto mejor sea cada solución, mayor probabilidad tendrá de cruzarse con otras soluciones y, por lo tanto, mayor probabilidad de preservar su información genética en futuras soluciones de la población. Por otro lado, las soluciones que no sean válidas no tendrán prácticamente probabilidad de continuar en el algoritmo evolutivo.

El siguiente esquema resume el procedimiento de un algoritmo de computación evolutiva:

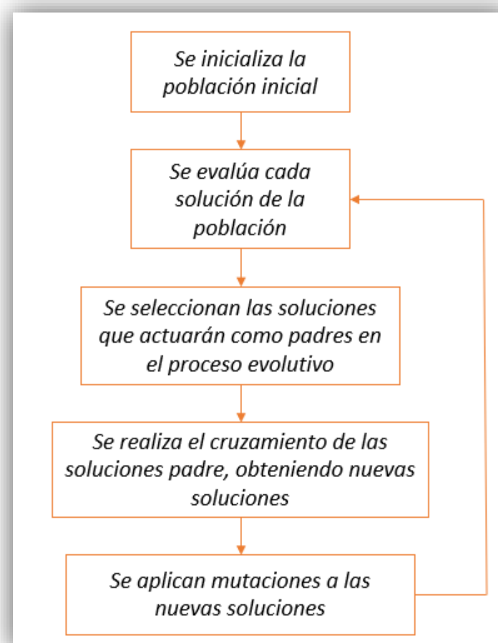

Figura 13: Esquema de algoritmo evolutivo

1. Se inicializa la población inicial, generalmente con soluciones aleatorias.

2. Cada una de estas soluciones es evaluada y se le asigna una puntuación conocida como *fitness* (grado de adecuación).

3. Se seleccionan los individuos que actuarán como padres de las futuras soluciones. Cuanto mejor sean estas soluciones, es decir, cuanto mejor sea su fitness, mayor probabilidad tendrán de ser seleccionados.

4. Se realiza el cruzamiento de los padres seleccionados. Este cruzamiento puede ser llevado a cabo utilizando 2 o más padres. En caso de utilizar un solo padre, no se realizaría cruzamiento, y sólo se aplicaría la mutación (siguiente paso).

5. Se producen ligeras variaciones en las soluciones, conocidas como mutaciones.

6. Se repite el proceso, desde el paso 2, con la población actual.

Un algoritmo evolutivo consta de los siguientes elementos:

- Representación de los individuos
- Función fitness
- Recombinación
- Mutación
- Gestión de la población
- Selección de padres
- Selección de supervivientes

En los siguientes apartados analizaremos cada uno de estos componentes en mayor detalle, para comprender completamente los algoritmos evolutivos. [10]

2. Representación de los individuos

Se consideran individuos al conjunto de posibles soluciones ante un problema que se desea resolver mediante computación evolutiva. Cada individuo está constituido por su fenotipo (*phenotype*) y su genotipo (*genotype*). El genotipo es la representación codificada de una

solución, mientras que el fenotipo es el objeto que forma el genotipo en el marco de la computación evolutiva. El genotipo se divide en unidades conocidas como genes (*genes*), los cuales representan una característica del individuo. Las diferentes posibles características que puede representar un gen se denomina alelo (*allele*). La posición de los genes, en el genotipo, que representan cierta característica se denomina locus (*loci*).

En la computación evolutiva existe un conjunto de sinónimos que se utilizan indistintamente:

- El término individuo (*individual*) también es conocido como posible solución (*posible solution*).
- El genotipo también es conocido como cromosoma (*chromosome*), estructura (*structure*), cadena (*string*, principalmente genotipos binarios) o individuo. [10] [13]

El primer paso para resolver un problema mediante computación evolutiva consiste en la representación (*representation*). La representación es la definición de la codificación del fenotipo en su genotipo equivalente. Por ejemplo, ante un problema cuyas soluciones son redes neuronales, el fenotipo estaría formado por la lista de pesos y bias de dichas redes neuronales, mientras que el genotipo es la codificación de dicha lista (binario, complemento A2, flotante, etc). [10]

3. Función Fitness

La función fitness (también conocida como función de evaluación) nos permite conocer la calidad de cada una de las soluciones que van surgiendo en la ejecución de un algoritmo evolutivo. La gran mayoría de métodos de selección de padres y de supervivencia se apoyan en la función fitness para la toma de decisiones, permitiendo que los individuos con mayor fitness tengan mayores probabilidades de crear nuevos individuos o de sobrevivir. Debe definirse de forma específica para cada problema, de tal forma que cuanto mejor sea la solución, más alta debe valer. También nos permite conocer cómo evolucionan las soluciones a lo largo del tiempo. De tal forma que, si un algoritmo de computación evolutiva funciona correctamente, es de esperar que las soluciones mejoren su valor fitness según avanzan las generaciones de la ejecución. [10]

Por ejemplo, supongamos que disponemos de una topología en malla de redes de comunicaciones formada por routers, donde cada salto tiene un determinado coste. El objetivo es encontrar el camino que une dos determinados extremos con menor coste posible. Si aplicamos computación evolutiva para hallar dicho camino, se podría utilizar una función inversa al coste del camino de la solución, de tal forma que cuanto mayor sea el coste de la solución, menor será el valor fitness asignado. Sea una solución x, la función fitness:

$$f(x) = \frac{1}{coste_total}$$

4. Recombinación

La recombinación (*recombination*) es el proceso por el cual se generan una o más posibles soluciones nuevas a partir de dos o más soluciones anteriores, mediante la combinación del genotipo de estas. Este proceso también es conocido como cruzamiento (*crossover*) debido a su analogía con el cruzamiento genético observable en la naturaleza. A continuación, se exponen algunas de las formas de recombinación en función de la representación de los individuos:

Recombinación de individuos con representación binaria

Los individuos con representación binaria son aquellos cuya representación del genotipo consiste en una cadena de bits. Ante este genotipo, se realizan comúnmente los siguientes cruzamientos:

1. *Cruzamiento en un punto (one-point crossover)*: para este tipo de cruzamiento se utilizan dos padres, obteniendo dos individuos nuevos. Primero se debe obtener un número aleatorio $r \in [1 , d-1]$ siendo d la longitud del genotipo. A continuación, se realiza un corte en el genotipo de ambos padres por la posición r. El primer hijo estará formado por la primera parte del genotipo de un padre y la segunda parte del otro. El segundo hijo estará formado por las otras partes del genotipo que no ha heredado el primero.

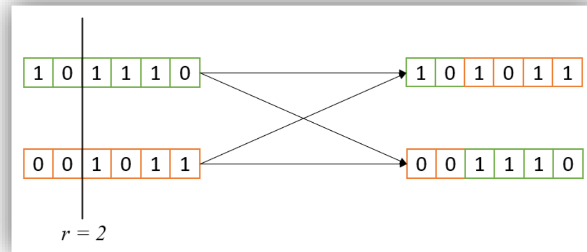

Figura 14: Cruzamiento en un punto

2. *Cruzamiento en n-puntos (n-point crossover)*: para realizar el cruzamiento en n-puntos se obtienen n números aleatorios $r \in [1 , d\text{-}1]$ diferentes. Se reparten los segmentos de los cromosomas padres entre los dos hijos resultantes.

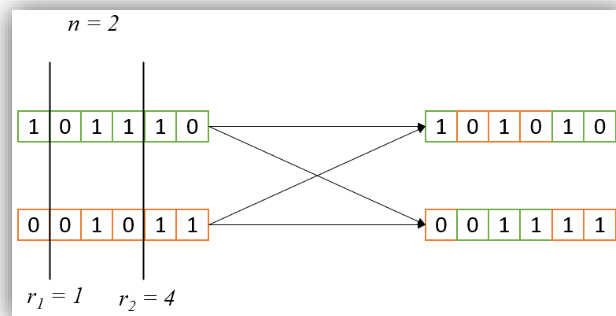

Figura 15: Cruzamiento en n-puntos

3. *Cruzamiento uniforme (uniform crossover)*: cada bit del genotipo del primer hijo tiene una probabilidad p de proceder del primer padre y una probabilidad $(1 - p)$ de proceder del segundo padre. Lo más común es utilizar una probabilidad de $(p = 0.5)$ para que el individuo descendiente tenga una proporción equivalente de información genética de ambos padres. El segundo hijo es generado utilizando el mapeo de bits inverso.

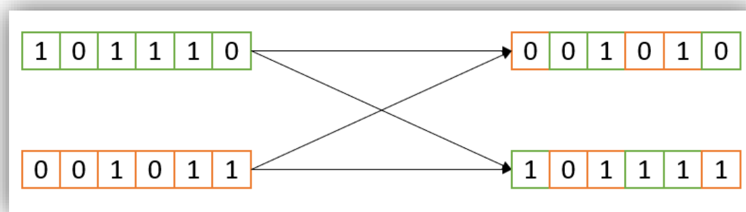

Figura 16: Cruzamiento uniforme

Recombinación de individuos con representación coma flotante

Los individuos con representación coma flotante son aquellos cuyo genotipo está formado por una secuencia de números de coma flotante. Ante este genotipo se realizan habitualmente los siguientes cruzamientos:

1. *Recombinación discreta (discrete recombination)*: equivalente al cruzamiento uniforme de representación binaria. Cada valor tiene una probabilidad *p* y *(1 - p)* de proceder de un padre o de otro. La principal desventaja de este tipo de cruzamiento es que no se introduce material genético nuevo, simplemente se realiza una combinación de los valores ya existentes. Por lo tanto, solo se introducen modificaciones en estos valores mediante la mutación.

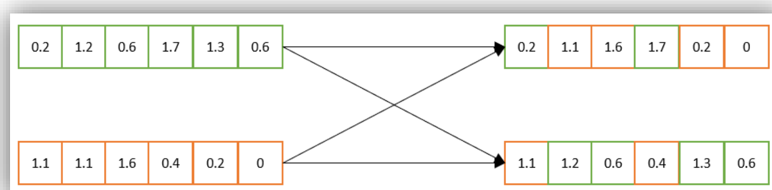

Figura 17: Recombinación discreta

2. *Recombinación aritmética o intermedia (arithmetic or intermediate recombination)*: se introduce una operación para el cálculo del nuevo genotipo. Sea un valor del genotipo del primer padre x_i y otro del segundo padre y_i, el valor perteneciente al genotipo hijo z_i se calcula:

$$z_i = \alpha x_i + (1 - \alpha) y_i$$

Este tipo de cruzamiento sí genera valores nuevos en el genotipo, pero al realizar una media entre los valores de los cromosomas, se ven reducidos los valores máximos a lo largo de los cruzamientos. El valor de α se puede obtener como un número aleatorio entre 0 y 1 pero habitualmente se utiliza *(α = 0.5)*, en cuyo caso se denomina recombinación aritmética uniforme *(uniform arithmetic recombination*). Podemos distinguir tres tipos de recombinación aritmética:

a. *Recombinación aritmética simple (simple arithmetic recombination)*: se selecciona un valor $k \in [1\ ,\ d\text{-}1]$. El primer hijo tendrá los genes del primer padre hasta *k*. El segundo hijo tendrá los genes del segundo padre hasta *k*. A partir de *k*, el genotipo de ambos hijos estará formado por la media de ambos padres (con el correspondiente α seleccionado).

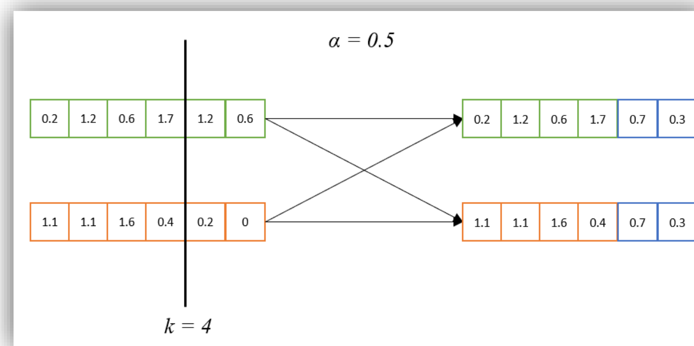

Figura 18: Recombinación aritmética simple

b. Recombinación aritmética individual (single arithmetic recombination): se elige un valor $k \in [1\ ,\ d]$. El primer hijo hereda todo el genotipo del primer padre y el segundo hijo del segundo padre. El gen situado en la posición k de ambos hijos es calculado como la media de ambos. La principal desventaja de este tipo de recombinación consiste en que introduce muy poco cruzamiento entre los genomas de los padres.

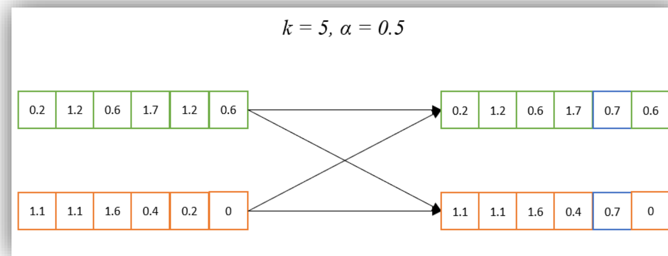

Figura 19: Recombinación aritmética individual

c. *Recombinación aritmética completa (whole arithmetic recombination)*: cada gen del genotipo se calcula como la media de ambos padres. Para cada hijo tenemos:

$$hijo_1 \rightarrow z_i = \alpha x_i + (1 - \alpha) y_i$$

$$hijo_2 \rightarrow z_i = \alpha y_i + (1 - \alpha) x_i$$

Si se utiliza *(α = 0.5)* ambos hijos serán idénticos.

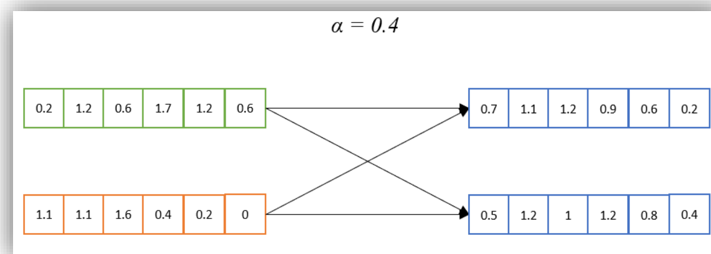

Figura 20: Recombinación aritmética completa

De los anteriores, la recombinación aritmética completa es la más utilizada para representaciones de tipo flotante. [10]

5. Mutación

La mutación permite obtener un nuevo genotipo a partir de otro, introduciendo pequeñas modificaciones de forma aleatoria. Las mutaciones permiten obtener una mayor variedad de genotipos, facilitando la exploración de posibles nuevas soluciones. Al igual que la

recombinación, podemos clasificar los diferentes métodos de mutación en función de la representación de los individuos.

Mutación de individuos con representación binaria

La forma más habitual de aplicar mutaciones en *strings* binarios consiste en invertir algunos de los bits. Cada bit se trata de forma individual, con una probabilidad p_m de invertirse. Esto implica que, si estos bits representan, por ejemplo, números enteros, existe la posibilidad de que un mismo número entero sea mutado dos veces, mientras que otros no sean mutados ninguna. Además, debido a que algunos bits pueden ser más significativos que otros, existe la misma probabilidad de que se produzca una mutación leve que una mutación grande. Por ejemplo, ante la secuencia de bits *00000000* que representa al número entero 0, existe la misma probabilidad de que una mutación cambie el valor a 1 (*00000001*) a que lo cambie a 128 (*10000000*). [10]

Mutación de individuos con representación coma flotante

Para genotipos de tipo coma flotante existen los siguientes tipos de mutaciones:

1. *Mutación uniforme (uniform mutation)*: sea x_i el valor original y x'_i el valor mutado del genotipo, se debe definir un rango $[L_i, U_i]$ de mutación que cumpla que:

$$x_i,\ x_i' \ \in [L_i,\ U_i]$$

   El valor mutado será un número aleatorio contenido en este rango. Normalmente, cada número del genotipo tiene la misma probabilidad de mutación p_m. Al igual que la mutación para valores binarios, este tipo de mutación tiene la misma probabilidad de realizar cambios grandes o leves sobre un número.

2. *Mutación no uniforme (nonuniform mutation)*: Se define igualmente un rango $[L_i, U_i]$ entre los cuales se comprenderán los valores del genotipo. Al valor que se desea mutar se le suma un valor obtenido de una distribución normal. Una distribución normal o distribución Gaussiana, es una distribución de probabilidad de variable continua cuya función de densidad tiene forma de campana. La función de densidad se define como:

$$\phi_{\mu,\sigma^2}(z) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(z-\mu)^2}{2\sigma^2}} \quad , \quad (-\infty < z < \infty)$$

Siendo μ la media y σ la desviación típica. Si se utiliza una media ($\mu = 0$) y una desviación típica ($\sigma = 1$), se denomina distribución normal estándar:

$$\phi(z) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}z^2} \quad , \quad (-\infty < z < \infty)$$

La probabilidad de un valor menor o igual a Z de una distribución normal estándar se calcula utilizando la función de distribución, *Φ(z)*:

$$P(Z < z) = \Phi(z) = \int_{-\infty}^{z} \phi(x)dx$$

No existe ninguna fórmula más simple, por lo que se suele utilizar una tabla con las probabilidades calculadas.

	.0	.01	.02	.03	.04	.05	.06	.07	.08	.09
0.0	.5000	.5040	.5080	.5120	.5160	.5199	.5239	.5279	.5319	.5359
0.1	.5398	.5438	.5478	.5517	.5557	.5596	.5636	.5675	.5714	.5753
0.2	.5793	.5832	.5871	.5910	.5948	.5987	.6026	.6064	.6103	.6141
0.3	.6179	.6217	.6255	.6293	.6331	.6368	.6406	.6443	.6480	.6517
0.4	.6554	.6591	.6628	.6664	.6700	.6736	.6772	.6808	.6844	.6879
0.5	.6915	.6950	.6985	.7019	.7054	.7088	.7123	.7157	.7190	.7224
0.6	.7257	.7291	.7324	.7357	.7389	.7422	.7454	.7486	.7517	.7549
0.7	.7580	.7611	.7642	.7673	.7703	.7734	.7764	.7794	.7823	.7852

Figura 21: Extracto de la tabla de probabilidades de distribución normal estándar [11]

Por lo tanto, para utilizar esta distribución en la mutación uniforme, en primer lugar obtenemos un valor aleatorio $r \in [0, 1]$. Utilizamos este valor obtenido como probabilidad de la distribución, y mediante la tabla obtenemos el valor Z correspondiente. Finalmente, se ajusta para que no sobrepase el rango definido. Este método de mutación permite que las modificaciones sobre los números sean más pequeñas, pero no existe una probabilidad nula de que también se produzca algún cambio brusco.[10] [12]

6. Gestión de la Población

Se denomina población al conjunto de individuos o soluciones presentes en un algoritmo evolutivo en un momento dado. En lo relativo a la gestión de la población, existen dos posibilidades:

1. *Modelo generacional (generational model)*: a partir de las soluciones existentes, se genera una *"piscina de apareamiento"* (*mating pool*). Esta *mating pool* estará formada, generalmente, por más copias de los individuos con mejor fitness, y menos copias o ninguna de los que tienen peor fitness. A partir de los individuos que la forman, se generará una población nueva que sustituirá a la existente. Una vez realizado esto, se considerará una nueva generación en el algoritmo evolutivo.

2. *Modelo estado-estacionario (steady-state model)*: a diferencia del modelo generacional, no se sustituye toda la población por cada generación. Se selecciona un número λ tal que ($\lambda < \mu$), siendo μ el número de individuos de la población. Se generan λ nuevas soluciones, manteniendo (μ - λ) soluciones antiguas. La proporción de la población que es sustituida entre generaciones se denomina brecha generacional (*generational gap*). [10]

7. Selección de Padres

Para la obtención de una nueva solución a partir de dos o más soluciones previas, es necesario realizar previamente una selección de dichas soluciones. Este proceso se denomina la selección de padres (*parent selection*) y tiene como finalidad transmitir los mejores genes a futuros individuos. Existen diversas formas de realizar esta selección:

1. *Selección proporcional al fitness (fitness proportional selection)*: la probabilidad de que un individuo sea seleccionado para ser padre depende del fitness que ha obtenido en comparación con el resto de la población. Para esto, se le asigna una probabilidad a cada individuo proporcional al fitness que ha obtenido, de tal forma que la suma de todas las probabilidades de todos los individuos es 1. Por lo tanto, la probabilidad de un individuo cualquiera es:

$$p_i = \frac{f_i}{f_t}$$

Siendo f_i el fitness del individuo y f_t la suma del fitness de toda la población. Este método también es conocido como selección de ruleta (*roulette wheel selection*) debido a su analogía a obtener individuos de una ruleta con las probabilidades indicadas anteriormente. Por ejemplo, supongamos una población de tres individuos con fitness 3, 6 y 9 respectivamente. La probabilidad de cada individuo queda:

Individuo 1	$p_1 = \frac{3}{(3+6+9)} = \frac{1}{6}$
Individuo 2	$p_2 = \frac{6}{(3+6+9)} = \frac{1}{3}$
Individuo 3	$p_3 = \frac{9}{(3+6+9)} = \frac{1}{2}$

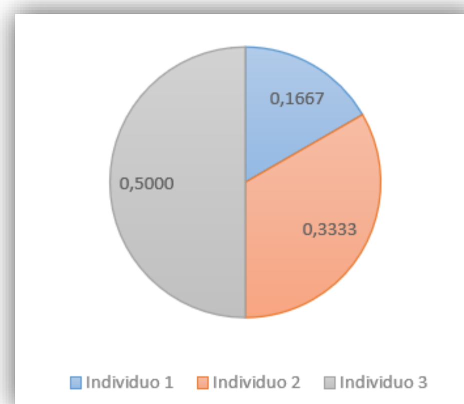

Figura 22: Ejemplo selección en función del fitness

La desventaja de este método de selección es que los individuos con un fitness muy superior al resto abarcan gran parte de la probabilidad de selección, teniendo lugar una convergencia prematura sin la posibilidad de explorar el espacio de posibles soluciones. También hay que mencionar que si, por el contrario, el valor fitness de

todos los individuos es muy próximo, se reduce la selección de las mejores soluciones, ralentizando el proceso de convergencia.

2. *Selección por clasificación (ranking selection)*: en este método, una vez obtenido el fitness de todos los individuos, se genera una clasificación. La probabilidad de selección es repartida en función de esta. Esta repartición puede realizarse de forma lineal o exponencial:

   - *Lineal*: se distribuyen de forma lineal todas las probabilidades, siendo la posición ($i = 0$) la del individuo con el peor fitness. La probabilidad de cada individuo:

$$p_i = \frac{(2-s)}{\mu} + \frac{2i(s-1)}{\mu(\mu-1)}$$

Siendo μ el número de individuos por población y s un parámetro tal que:

$$s \in (1, 2]$$

Cuanto mayor sea s, mayor será la diferencia entre las probabilidades de los extremos de la clasificación, es decir, $(i = 0)$ e $(i = \mu - 1)$. Por ejemplo, para una población de 100 soluciones, tendríamos las siguientes probabilidades para ($s = 1{,}01$), ($s = 1{,}5$) y ($s = 2$):

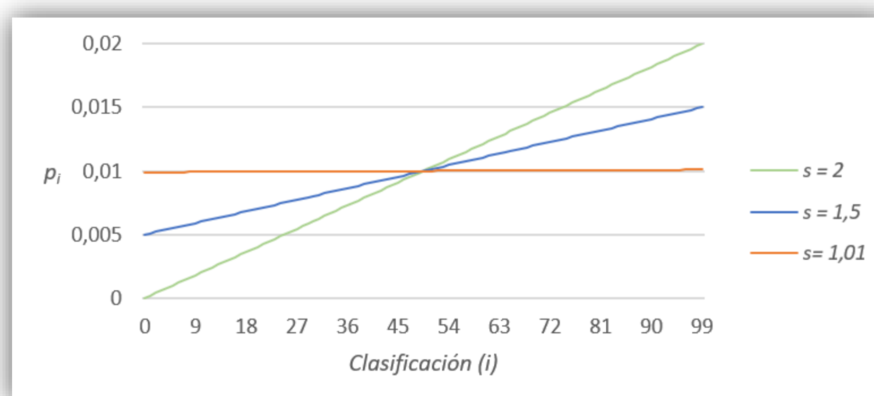

Figura 23: Ranking lineal para s = 1.01, s = 1.5 y s = 2

- *Exponencial*: se distribuyen de forma exponencial, teniendo una probabilidad muy inferior los individuos con menor fitness. Siendo j la clasificación de la solución, con $j = 1$ para el mejor individuo y $j = \mu$ la del peor:

$$p_j = q(1 - q)^{j-1}$$

Siendo q un parámetro variable que indica la probabilidad del individuo con mejor fitness.

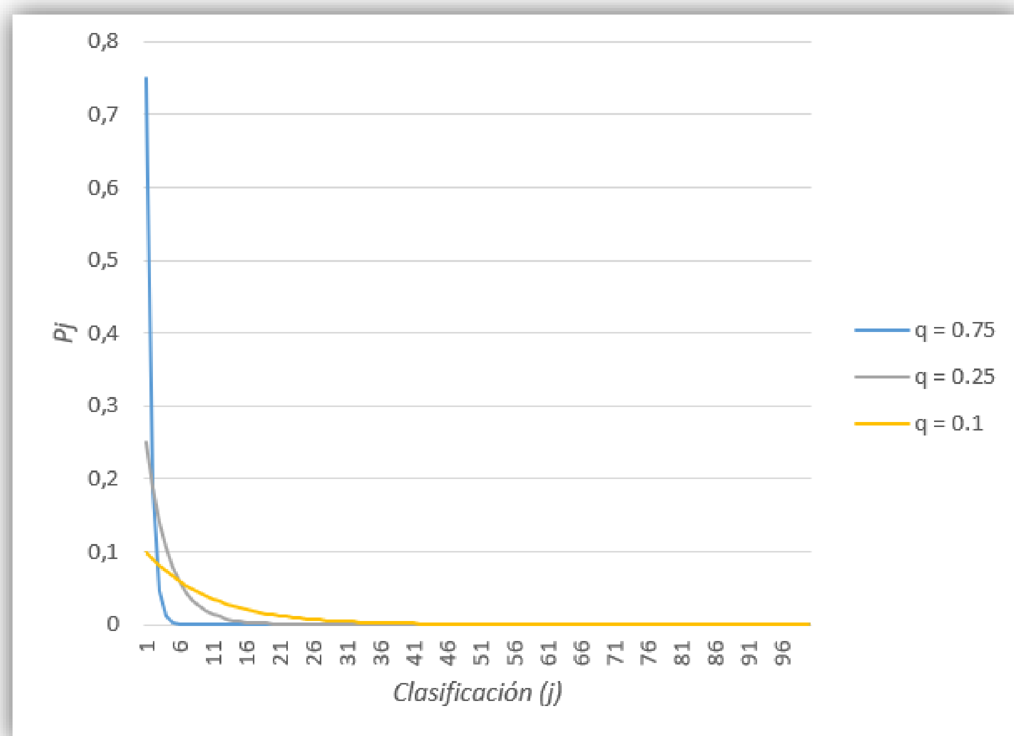

Figura 24: Ranking exponencial para q = 0.75, q = 0.25 y q = 0.1

Si el valor q o el número de individuos es muy pequeño, es necesario ajustar los valores para que las suma de todas las probabilidades sea 1. Por ejemplo, si utilizamos ($q = 0.1$) para una población de tan sólo 10 soluciones, las probabilidades, sin ajustar y ajustadas, son:

j	*p_j (sin ajuste)*	*p'_j (con ajuste)*
1	0,1	0,153533993
2	0,09	0,138180594
3	0,081	0,124362535
4	0,0729	0,111926281
5	0,06561	0,100733653
6	0,059049	0,090660288
7	0,053144	0,081594259
8	0,04783	0,073434833
9	0,043047	0,06609135
10	0,038742	0,059482215
Suma total	0,65132156	1

Figura 25: Ejemplo de probabilidades exponenciales con baja población

Para realizar el ajuste basta con dividir cada probabilidad con la suma total de probabilidades:

$$p'_j = \frac{q(1-q)^{j-1}}{(\sum_{j=1}^{\mu} p_j)}$$

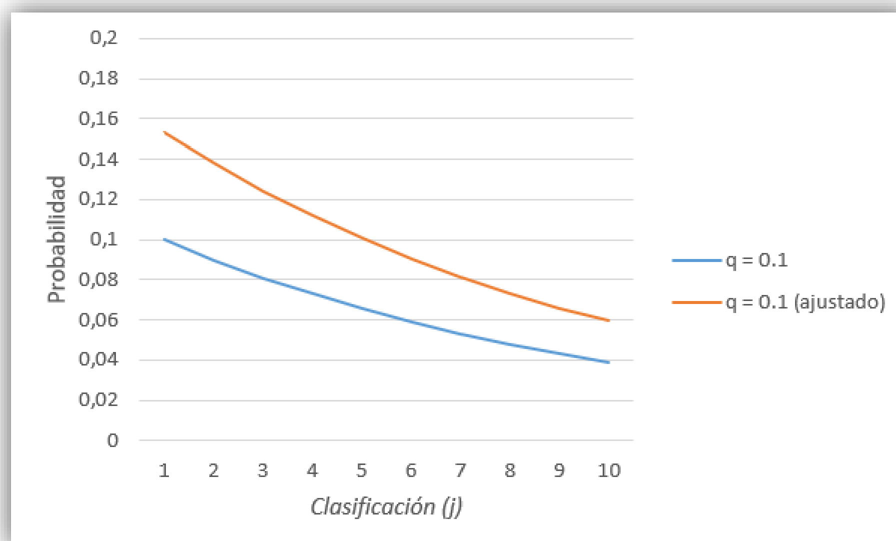

Figura 26: Comparación de probabilidades con y sin ajustar, para q = 0.1 y μ = 10

3. *Selección de torneo (tournament selection)*: este tipo de selección se utiliza cuando se tiene un conjunto de población muy grande. Se seleccionan k individuos de forma aleatoria. De estos k individuos, se seleccionan los mejores. Este proceso se repite hasta que se hayan seleccionado todos los padres.

4. *Selección uniforme de padres (uniform parent selection)*: en este método, cada solución tiene las mismas probabilidades de ser seleccionadas. Para ello, se seleccionan números aleatorios tal que $r \in [1 , \mu]$. Para que este método sea efectivo, debe imponerse un método de selección de supervivientes basado en el fitness. En caso contrario, no se obtendría convergencia. [10] [13]

8. Selección de Supervivientes

A lo largo de la ejecución de un algoritmo evolutivo, se debe decidir qué soluciones deben seguir compitiendo y cuales deben eliminarse. Este proceso se denomina selección de supervivientes (*survivor selection*). En un principio, cualquier técnica de las mencionadas anteriormente para la selección de padres también sería válida para la selección de supervivientes. No obstante, suelen utilizarse otras técnicas para este tipo de selección:

1. *Reemplazo basado en la edad (age-based replacement)*: la decisión se toma en función del tiempo (o iteraciones del algoritmo) que lleva el individuo en la población. Por lo tanto, no utiliza en ningún momento el fitness para evaluar la supervivencia.

2. *Reemplazo basado en fitness (fitness-based replacement)*: la decisión de supervivencia se toma en función del fitness obtenido por cada individuo. Existen diferentes métodos:

    a. *Reemplazo del peor (replacement of the worst)*: los individuos con el menor fitness son reemplazados por nuevos individuos.

    b. *Elitismo (elitism)*: se mantiene la solución con mayor fitness de toda la población. El resto son reemplazados por nuevas soluciones. [10]

9. Algoritmos Genéticos

Existen diferentes tipos de algoritmos evolutivos en función de las características mencionadas anteriormente. Algunas de estas variantes son: programación genética (*genetic programming*), estrategias evolutivas (*evolution strategies*), programación evolutiva (*evolutionary programming*) y algoritmos genéticos (*genetic algorithms*).

Los algoritmos genéticos son la rama de la computación evolutiva más conocida, y está formada por las siguientes características:

1. Representación: binaria
2. Recombinación: cruzamiento en un punto
3. Mutación: a nivel de bit
4. Selección de padres: selección proporcional al fitness
5. Gestión de la población: generacional.
6. Selección de supervivientes: reemplazo basado en la edad.

Por lo tanto, los algoritmos genéticos son algoritmos evolutivos con un conjunto de parámetros específicos. [10] [13]

Como se mencionó anteriormente, en función de la aplicación puede ser conveniente la utilización de unos parámetros u otros. En este Proyecto de Fin de Grado se han implementado los siguientes posibles parámetros:

- Representación de los individuos: tanto binaria como flotante
- Recombinación: en función del tipo de representación elegida:
    - Representación binaria: cruzamiento en un punto, cruzamiento en n-puntos y cruzamiento uniforme.
    - Representación flotante: recombinación discreta, recombinación aritmética simple, recombinación aritmética individual y recombinación aritmética completa.
- Mutación: en función del tipo de representación seleccionada:
    - Representación binaria: mutación binaria.
    - Representación flotante: mutación uniforme y mutación no uniforme.
- Gestión de la población: modelo generacional y modelo estado estacionario.
- Selección de padres: método basado en el fitness, método basado en el ranking lineal y método basado en el ranking exponencial.

Esta implementación se explica con mayor detalle en el apartado *DISEÑO E IMPLEMENTACIÓN*.

Una vez detallado el estado del arte de las redes neuronales artificiales y la computación evolutiva, el siguiente paso consiste en explicar la unión de ambas. En los siguientes apartados se describe esto, indicando primero las diferencias de utilizar propagación hacia atrás y computación evolutiva para redes neuronales, y continuando con posibles representaciones del genotipo de estas en computación evolutiva.

3. Aplicación de Algoritmos Genéticos en Redes Neuronales

En el apartado *Perceptrón de varias capas* se detalló como entrenar una red neuronal mediante el método conocido como propagación hacia atrás. Esta no es la única forma de entrenar redes neuronales. Aplicando lo detallado en el apartado anterior, podemos obtener los pesos más adecuados para una red neuronal aplicando algoritmos evolutivos.

En este proyecto se aplicarán los conocimientos de computación evolutiva para entrenar las redes neuronales. Las principales diferencias entre el entrenamiento aplicado en el proyecto y el entrenamiento que hace uso de la propagación hacia atrás son las siguientes:

Entrenamiento mediante propagación hacia atrás	***Entrenamiento en este proyecto***
Se utiliza la misma red neuronal durante todo el entrenamiento.	Se utiliza un gran número de redes neuronales durante el entrenamiento, manejando más de una red simultáneamente.
Al final del entrenamiento se obtiene una única posible solución al problema.	Al final del entrenamiento se obtienen varias posibles soluciones al problema.
El entrenamiento se realiza mediante la presentación de un conjunto de datos de los cuales se conoce de antemano la respuesta correcta que debe aportar la red neuronal.	El entrenamiento se realiza mediante la presentación de entradas de forma continua a lo largo del tiempo para cada red neuronal, evaluando la calidad de las respuestas cuando finaliza la generación.
La búsqueda de los pesos correctos de la red neuronal se realiza mediante la propagación hacia atrás.	La búsqueda de los pesos correctos de la red se realiza mediante el cruzamiento y mutación de las redes que muestran mejor adaptación al problema a lo largo del tiempo.

Si se desea aplicar algoritmos genéticos u evolutivos para obtener la red neuronal artificial que mejor respuesta de ante un problema, el primer paso consiste en la definición del genotipo. La información de una red neuronal se encuentra en los pesos y bias de la misma, por lo tanto, el genotipo debe representar todo este conjunto de valores.

Ante este escenario tenemos dos posibilidades:

- Representación flotante: los valores de los pesos y los bias son valores flotantes. Por lo tanto, se puede representar el genotipo de una red neuronal mediante una lista ordenada de pesos y bias:

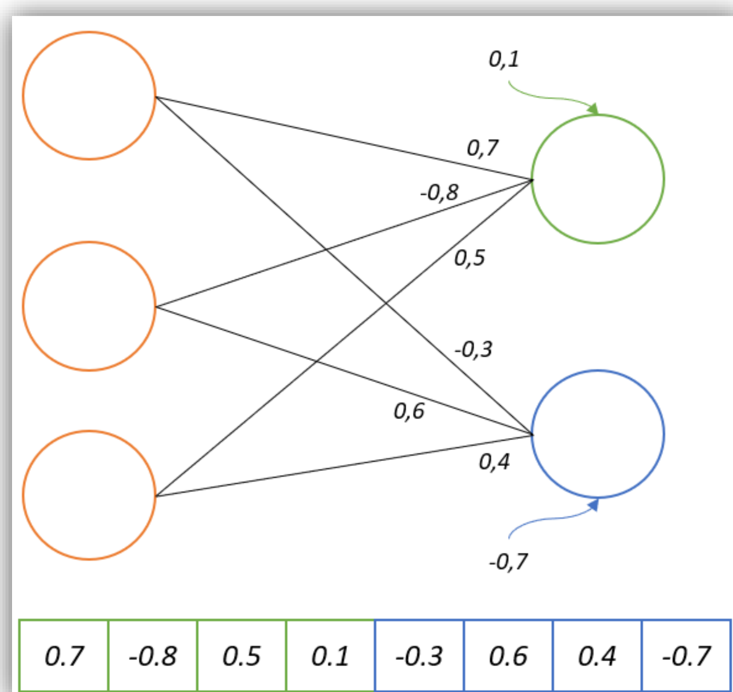

Figura 27: Ejemplo de representación flotante de una red neuronal

Para tratar con redes neuronales, lo más habitual es utilizar una representación flotante.

- Representación binaria: la lista de valores obtenida con la representación flotante se convierte en un *string* de bits, convirtiendo cada valor de la lista en binario.

Una vez representado el genotipo, el procedimiento del algoritmo genético (o algoritmo evolutivo) es el mismo al descrito anteriormente. Si se desea que la topología de la red

evolucione, es necesario que esta información también esté incluida en el genotipo. Para ello, podemos utilizar NEAT, como se describe en el apartado siguiente. [10] [14]

4. NEAT

1. Introducción

Previamente se comentó que una red neuronal artificial está compuesta por una capa de entrada, una capa de salida y un conjunto de capas intermedias. En un principio, esta estructura se define antes de comenzar el entrenamiento y se mantiene durante el mismo.

NEAT (*NeuroEvolution of Augmented Topologies*) es una técnica que permite el entrenamiento de una red neuronal mediante la aplicación de algoritmos evolutivos, modificando los pesos de las conexiones y la topología de esta. En los siguientes apartados se describe brevemente los conceptos más importantes del método NEAT

2. Red neuronal inicial

El objetivo del método NEAT consiste en encontrar los valores y la topología más adecuada de la red que permita solucionar un cierto problema. Por lo tanto, para poder explorar todo el espacio de topologías posibles, cuando se aplica NEAT se comienza con una red neuronal que sólo dispone de nodos de entrada y nodos de salida. El número de nodos de entrada y salida dependerán de las entradas y las salidas necesarias para resolver dicho problema.

Por ejemplo, si deseamos resolver un problema con una red neuronal artificial con cinco nodos de entrada y tres nodos de salida, la red inicial al aplicar NEAT será:

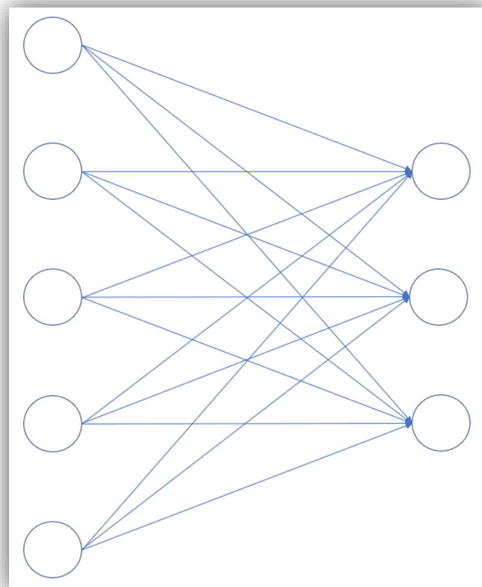

Figura 28: Red neuronal inicial al aplicar NEAT, con 5 nodos de entrada y 3 de salida

El resto de los nodos y conexiones surgirán con el paso de las generaciones, al aplicar mutaciones que favorezcan las soluciones del algoritmo.

3. Representación del genotipo

El genotipo de las redes neuronales, cuando se aplica NEAT, está diseñado para facilitar la recombinación entre dos de ellas. Para conseguir esto, se alinean los genes equivalentes de ambas redes como veremos en el apartado de recombinación.

El genotipo de estas redes consiste en una lista de todas las conexiones de las que dispone y una lista de los nodos. De cada conexión se almacena:

- Nodo entrante en la conexión.
- Nodo saliente en la conexión.
- Peso de la conexión.
- Bit de activación.

- Valor de innovación de la conexión.

El bit de activación permite habilitar o deshabilitar dicha conexión de la red. El valor de innovación de la conexión permite conocer si dos genes de redes diferentes se pueden cruzar o no. De cada nodo, en la lista de nodos se almacena:

- Número del nodo.

- Tipo del nodo.

El número del nodo es un número que identifica unívocamente al mismo. El tipo del nodo puede ser: *Sensor* (nodo de entrada), *Hidden* (nodo oculto) u *Output* (nodo de salida).

Genotipo

Nodo 1 Sensor	*Nodo 2: Sensor*	*Nodo 3: Output*	*Nodo 4: Hidden*

Entrada: 1	*Entrada: 2*	*Entrada: 2*	*Entrada: 4*
Salida: 3	*Salida: 3*	*Salida: 4*	*Salida: 3*
Peso: -0,7	*Peso: 0,8*	*Peso: 0,1*	*Peso: -0,5*
Habilitado: Sí	*Habilitado: No*	*Habilitado: Sí*	*Habilitado: Sí*
Innovación: 1	*Innovación: 2*	*Innovación: 3*	*Innovación: 4*

Figura 29: Ejemplo de genotipo de red neuronal basado en [15]

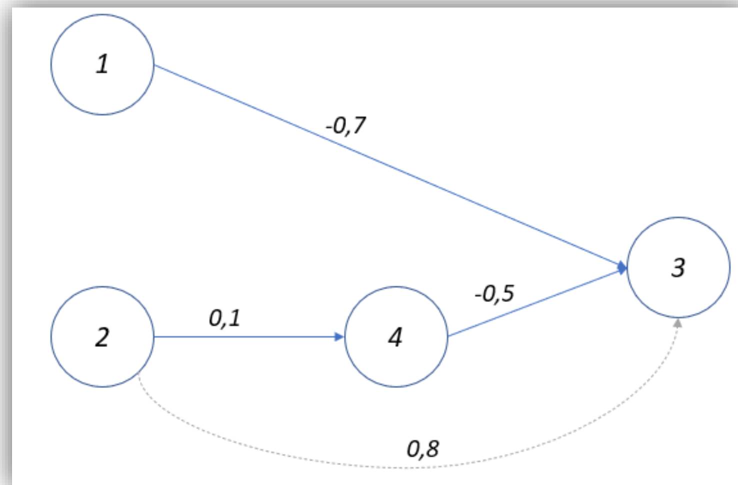

Figura 30: Red neuronal equivalente al genotipo de la figura anterior

4. Mutación

Existen dos tipos de mutaciones aplicables en el método NEAT:

- Mutación de valores: estas mutaciones modifican los pesos y los bias de las conexiones y los nodos respectivamente. Estas mutaciones son equivalentes a las descritas en los apartados anteriores.
- Mutación de la topología: la topología de la red puede mutar de dos formas diferentes:
    - Adición de un nuevo nodo. Este nodo se crea entre dos nodos ya existentes y que estaban conectados previamente, dividiendo la conexión en dos:

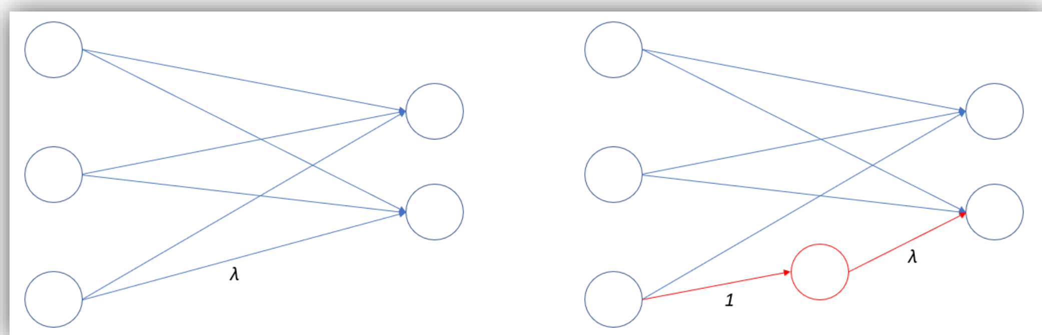

Figura 31: Creación de un nuevo nodo mediante NEAT

Para minimizar el impacto de la creación de un nuevo nodo, en una de las dos nuevas conexiones se mantiene el valor del peso antiguo, mientras que en la otra se establece a uno. De esta forma, se permite a la topología evolucionar sin crear cambios drásticos, permitiendo que la mutación de pesos/bias y la recombinación busquen los valores más apropiados para estas nuevas conexiones y para el nuevo nodo. El bias del nuevo nodo puede ser un valor aleatorio.

- Creación de una nueva conexión. Entre dos nodos existentes, si no existe una conexión entre ellos, se puede mutar una nueva conexión que los una a ambos, estableciendo un peso aleatorio.

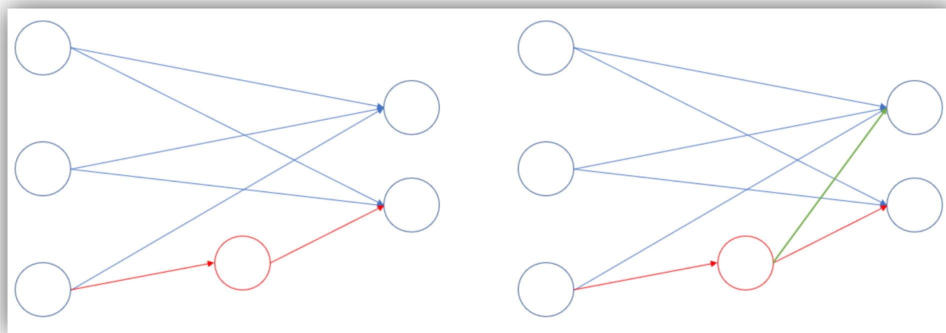

Figura 32: Creación de una nueva conexión mediante NEAT

5. Recombinación

El primer problema que surge al recombinar dos genotipos de redes neuronales obtenidas mediante NEAT consiste en que ambas redes pueden no presentar la misma topología. Es necesario establecer un mecanismo que nos permita conocer que genes de ambos genotipos se pueden cruzar y cuáles no.

Para solucionar esto, se incluye el valor de innovación de una conexión. Cuando se genera una nueva conexión, se le asigna un valor de innovación. Este valor es una variable que se incrementa con cada asignación en esa red.

En el momento de realizar el cruzamiento, las conexiones con el mismo valor de innovación serán cruzadas. Estos genes se denominan genes coincidentes (*matching genes*). Los genes de ambos genotipos cuyos valores de innovación no se encuentran en el contrario, se denominan genes disjuntos (*disjoint genes*) o genes excesivos (*excess genes*) en función de si los valores de innovación de estos se encuentran dentro del rango del otro genotipo. [16]

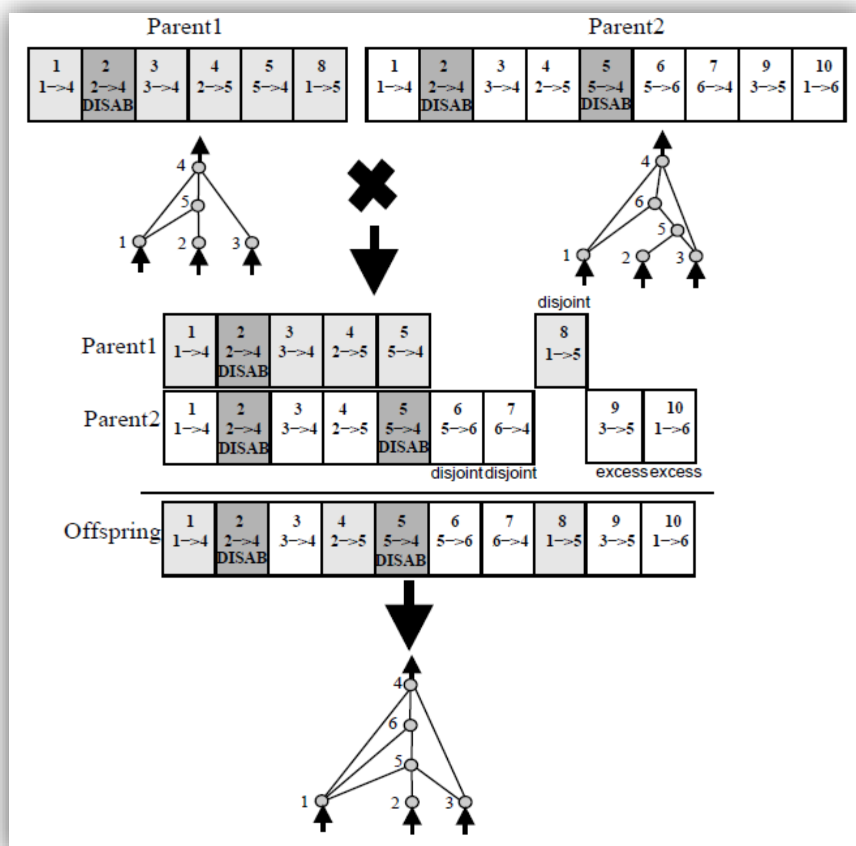

Figura 33: Cruzamiento de dos genotipos de redes neuronales con topologías diferentes [17]

En este proyecto no se utiliza el método NEAT para el entrenamiento de las redes neuronales. Sin embargo, se proporciona un código que permite instanciar dichas redes neuronales, permitiendo su incorporación en el proyecto como un trabajo futuro. Esto se explica con más detalle en el capítulo TRABAJOS FUTUROS.

DISEÑO E IMPLEMENTACIÓN

1. Introducción

Como se indicó al principio de este documento, el objetivo de este proyecto consiste en el desarrollo de un prototipo de un módulo de Unity que permita el entrenamiento de redes neuronales mediante la utilización de algoritmos evolutivos.

Unity es un motor de videojuegos multiplataforma, apto para sistemas operativos Windows y Mac. Permite el desarrollo de aplicaciones interactivas bidimensionales y tridimensionales en tiempo real para una gran variedad de plataformas. Junto con Unreal Engine, son los motores de videojuegos más utilizados. Pero no es únicamente utilizado en el sector de videojuegos, también tiene aplicaciones en el sector automotriz, arquitectura, ingeniería y construcción entre otros. Por esta razón se ha optado utilizar Unity como entorno de desarrollo para este Proyecto de Fin de Grado. [18] [19]

Para realizar el entrenamiento de las redes neuronales se aplicarán algoritmos evolutivos, como se mencionó en capítulo anterior, de tal forma que:

- Los individuos del algoritmo son las redes neuronales que se desea entrenar.
- Cada red neuronal controla una réplica de un componente del proyecto. Este componente también es conocido como agente.
- Los agentes son entrenados en un escenario por el módulo, culminando con las redes neuronales que mejor se hayan adaptado.
- Cada entrenamiento, formado por un conjunto determinado de generaciones, se conoce como simulación.

Para que este entrenamiento sea posible, es necesario que exista una unión entre las redes neuronales y algoritmos genéticos y entre los componentes a entrenar y el proyecto:

- El componente por entrenar debe introducir en la red neuronal información que percibe del escenario o entorno del proyecto.

- Una vez se procesa esta información y la red devuelve la salida, el agente debe interpretar esta salida para controlar su comportamiento.
- Cada agente es evaluado en base a una función fitness que permite conocer la calidad de la solución.
- En un principio, cada agente dispone de una misión que deben cumplir antes de que finalice la generación, existiendo dos posibilidades: éxito o fracaso.

Para ilustrar mejor estos aspectos, explicamos brevemente el ejemplo utilizado en el capítulo *PRUEBAS Y RESULTADOS I*:

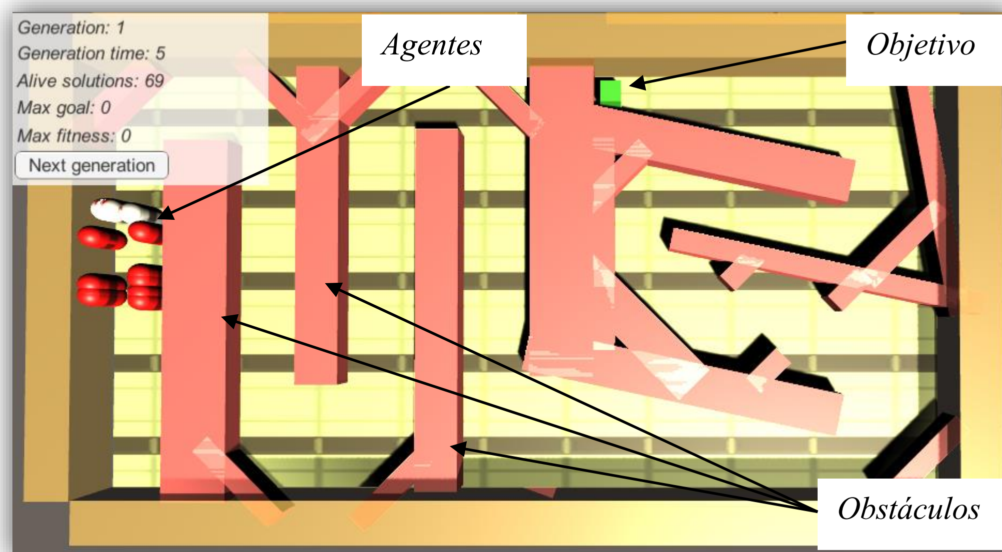

Figura 34: Escenario de ejemplo

La aplicación ilustrada consiste en un conjunto de agentes que navegan por un escenario tridimensional con el fin de alcanzar el componente objetivo. Este agente dispone de una presencia física en el escenario y, por lo tanto, de una posición y una rotación en el mismo. El agente puede moverse, interactuar y recibir información del escenario, pero no con otras réplicas de este. Es decir, las diferentes soluciones del algoritmo genético no interactúan entre sí.

Cada agente dispone de una red neuronal. En esta red neuronal introduce la información que perciben del escenario, en este caso, la distancia respecto a los obstáculos. La red neuronal

procesa esta información, obteniendo la información de salida. El agente interpreta esta información, controlando su comportamiento. El objetivo de la red neuronal en este ejemplo consiste en esquivar los obstáculos para finalmente alcanzar la meta. De esta forma, se considera que el agente que controla ha tenido éxito o fracaso. Tanto si ha tenido éxito alcanzando el objetivo como si no, se debe evaluar el comportamiento del agente en el escenario para continuar con la siguiente generación.

Una vez explicado el estado del arte relacionado con redes neuronales artificiales y algoritmos genéticos, procedemos en los siguientes apartados a describir el proyecto desarrollado. En primer lugar, se realiza una breve descripción de las restricciones de diseño del módulo. En segundo lugar, se detalla la implementación de este, explicando detalladamente cada una de las opciones de las que dispone. Por último, se realiza una breve descripción de las clases que lo forman.

2. Restricciones de diseño

Existen ciertas restricciones en el diseño y desarrollo del módulo. Estas están principalmente relacionadas con la aplicación en la cual se desee implementar. A continuación, se describen las restricciones de diseño de este Proyecto de Fin de Grado:

1. Restricciones de aplicación

Para que el módulo sea aplicable a una determinada aplicación, esta debe cumplir ciertas condiciones:

- Los individuos deben ser capaces de estar presentes en la simulación de forma simultánea. Esto implica que no pueden existir elementos temporales que desaparecen cuando el individuo interactúa con ellos o con los que no se puede interactuar más de una vez. Si existen este tipo de elementos, tan sólo una solución de toda la generación interactuaría con estos, y estarían inaccesible para el resto.

- Preferiblemente, las aplicaciones deben estar diseñadas para alcanzar una determinada meta u objetivo. El módulo considera finalizada una generación cuando todos los individuos mueren o alcanzan la meta, por lo tanto, para que funcione correctamente es necesario que los individuos tengan esos dos posibles finales.

- En el presente documento se utiliza una aplicación concreta para la explicación del funcionamiento y utilización del módulo. En *APÉNDICE A: MANUAL DE USUARIO* se detalla cómo unir el módulo con una aplicación, utilizando la misma aplicación mencionada anteriormente. Debido a la gran diversidad de aplicaciones, no es posible detallar una unión módulo-aplicación que cubra todas ellas. Por ello, el usuario debe realizar una comparación entre su aplicación y la aplicación de ejemplo expuesta, para realizar la configuración módulo-aplicación que se adapte a su caso particular.

2. Otras restricciones

Además de las restricciones de aplicación, es importante mencionar otras restricciones que pueden afectar al desarrollo y ejecución del módulo:

- Restricción computacional. Para realizar el entrenamiento, múltiples redes neuronales estarán en ejecución de forma simultánea. Cuando se aplica un algoritmo genético habitualmente se utiliza un número muy elevado de soluciones, desde varios miles de individuos incluso hasta llegar a superar el millón [10]. En este escenario, el número de individuos dependerá de la capacidad computacional del equipo en el que se ejecute, siendo, probablemente, mucho menor de lo debido. Por otro lado, cuanto mayor sea la red neuronal que se desee entrenar, mayor coste computacional tendrá. Por lo tanto, es otro factor que tener en consideración como restricción computacional.

- Restricción temporal. Para poder crear una nueva generación, es necesario que finalice la generación previa. En función de la aplicación y del número de generaciones establecidas, el tiempo de ejecución del entrenamiento puede variar de algunas horas a incluso días. Al igual que con la restricción computacional, el tamaño de la red neuronal elegida afecta a la restricción temporal. Entrenar una red neuronal de mayores dimensiones requiere un número mayor de generaciones, afectando al tiempo requerido de simulación.

- Restricción de incertidumbre. Este proyecto es una primera aproximación a los algoritmos evolutivos aplicados a redes neuronales en Unity. Por ello, se trata de una prueba de concepto con las dificultades que conlleva la aplicación de algoritmos complejos.

3. Implementación

En esta sección se desarrolla la implementación final del módulo, detallando el funcionamiento de este, cada uno de los paneles que lo forman y las diferentes opciones que presenta.

1. Funcionamiento del módulo

A continuación, se muestra un diagrama de flujo del funcionamiento general del módulo:

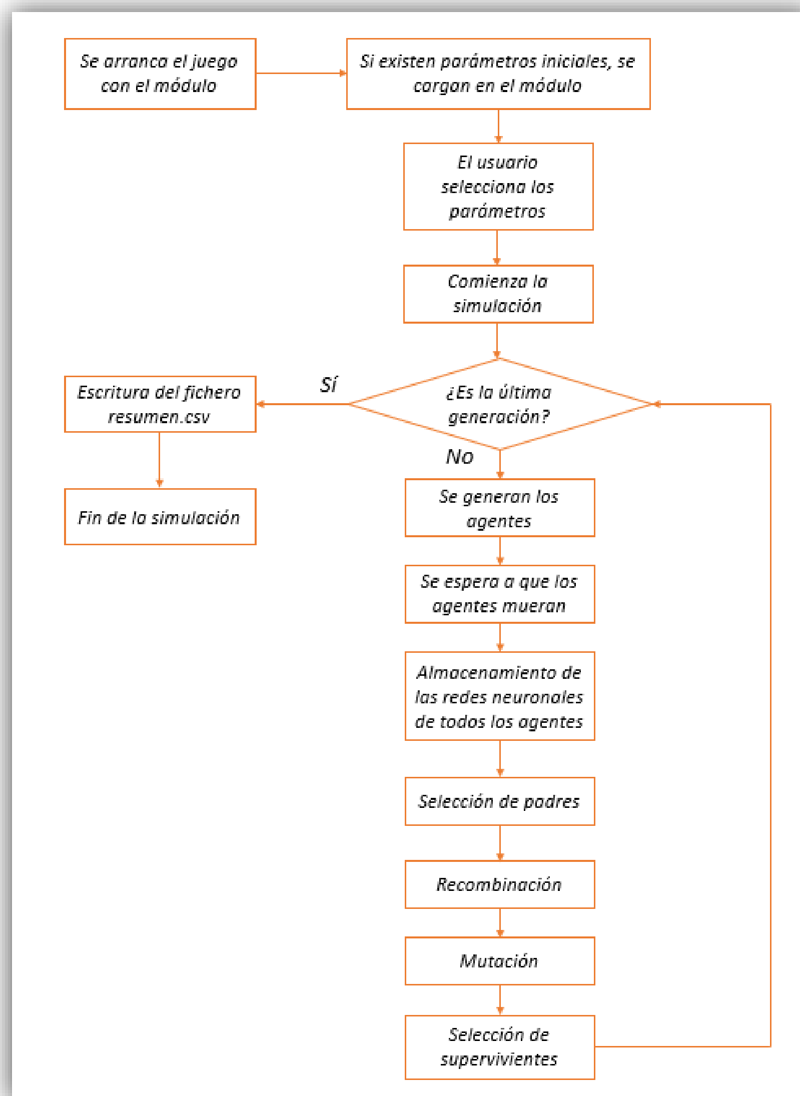

Figura 35: Diagrama de flujo del funcionamiento del módulo

- Cuando el usuario arranca la aplicación con el módulo incorporado, en primer lugar, se intenta cargar en los paneles la configuración de la simulación anterior. De esta forma, se puede evitar que el usuario tenga que indicar todos los parámetros de nuevo.

- Una vez se completa la carga de estos parámetros, se presenta al usuario un panel principal desde el cual puede acceder a paneles secundarios para introducir los parámetros correspondientes. Estos paneles se describen en los apartados siguientes.

- Una vez seleccionados todos los parámetros se inicia la simulación. El proceso de la simulación es equivalente al de un algoritmo evolutivo, ejecutando las generaciones con los individuos indicados.

- Una vez finaliza, el módulo escribe un fichero con información relacionada de la simulación.

2. Ficheros generados

Existen un conjunto de ficheros que se generan durante la ejecución de la simulación. Estos ficheros se describen seguidamente:

Input.xml

Cuando el usuario ha introducido los parámetros que desea ejecutar en la simulación y la arranca, el módulo genera un fichero llamado *input.xml* en el directorio del proyecto que contiene todos los parámetros de la simulación.

De la misma forma, cuando se arranca el juego, busca este fichero y, si lo encuentra, carga los parámetros de la simulación anterior en los campos del módulo. De esta forma se evita que el usuario tenga que introducir todos los parámetros cada vez que realice una simulación nueva con parámetros similares.

simulationResume.csv

Una vez se finaliza la simulación, se genera un fichero *csv* que contiene un resumen con la información más importante de la simulación. Gracias a este fichero podemos observar cómo han evolucionado los agentes a lo largo de las generaciones.

La información reflejada (por cada generación) es la siguiente:

- Valor fitness medio.
- Valor fitness máximo.
- Número de agentes que han alcanzado la meta.
- Duración de la generación.

Ficheros de redes neuronales

A la vez que se ejecuta la simulación, el módulo guarda cada red neuronal que utiliza. De esta forma, si el usuario desea utilizar esta red neuronal en otro proyecto puede hacerlo sin problemas.

En el proyecto, si no existe, se creará una carpeta llamada *simulation*. Dentro de esta carpeta se encontrará las diferentes carpetas de cada simulación. Cada simulación genera una carpeta llamada *simulation_X*, siendo *X* un número entero comenzando por 0:

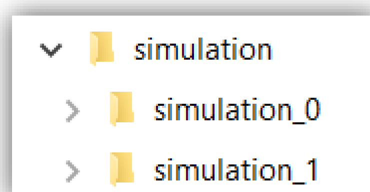

Figura 36: Carpeta simulation del proyecto

Dentro de cada carpeta *simulation_X* existe una carpeta por cada generación que se ha ejecutado en esa simulación llamada *genX*:

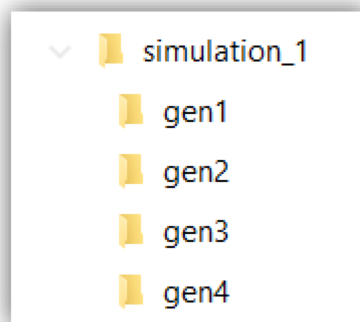

Figura 37: Carpeta simulation/simulation_X del proyecto

Por último, cada carpeta *genX* contiene un fichero xml por cada red neuronal que se ha utilizado en esa generación. Estos ficheros están ordenados en función del fitness de menor a mayor, facilitando la búsqueda de las mejores redes neuronales de cada generación. Como se muestra a continuación, cada fichero xml contiene toda la información necesaria para poder cargar los pesos y los bias en una red neuronal:

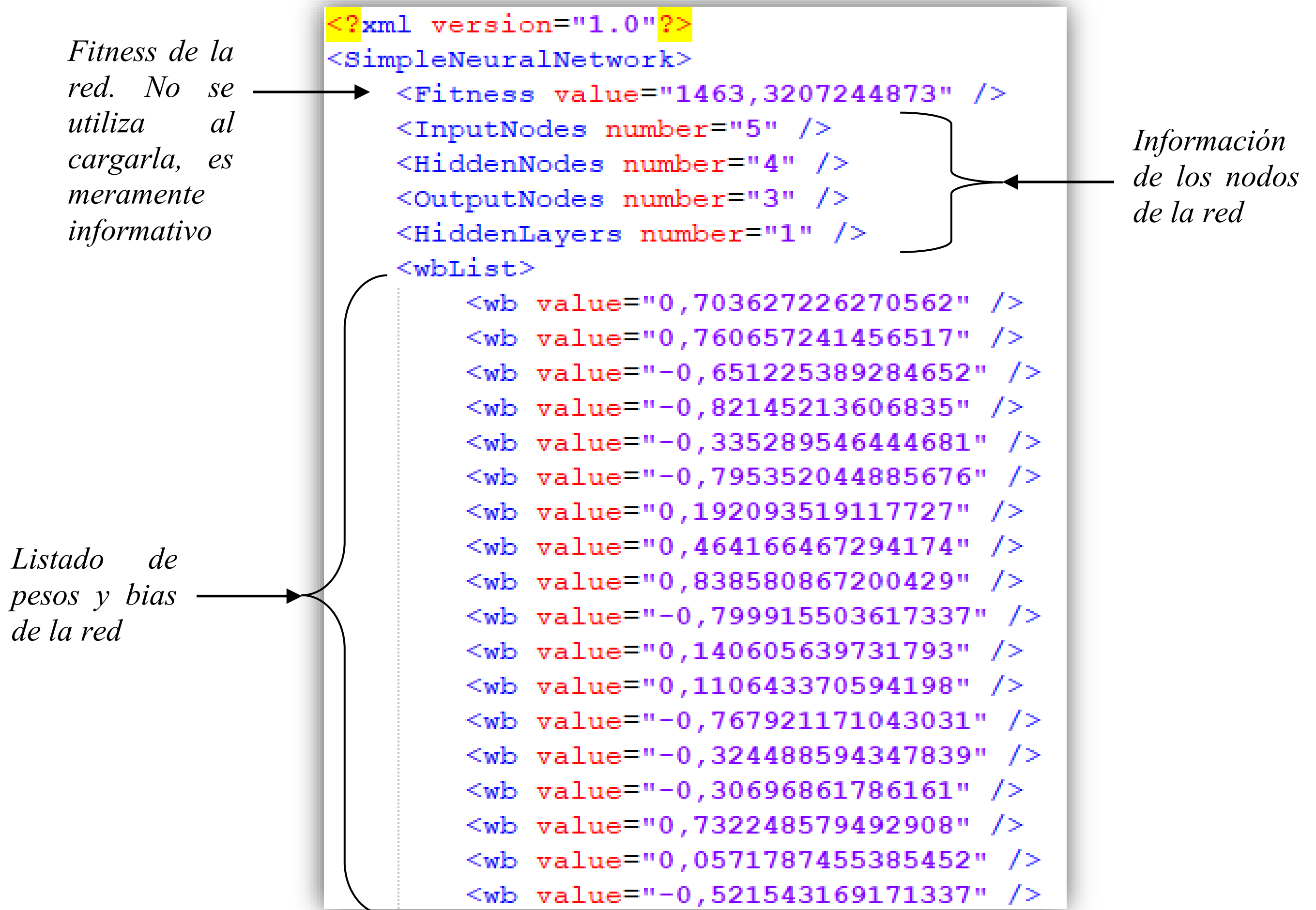

Figura 38: Fragmento xml de un fichero de red neuronal

3. Panel principal

Al arrancar un juego con el módulo incorporado, en la pantalla del juego se presenta la interfaz principal del módulo:

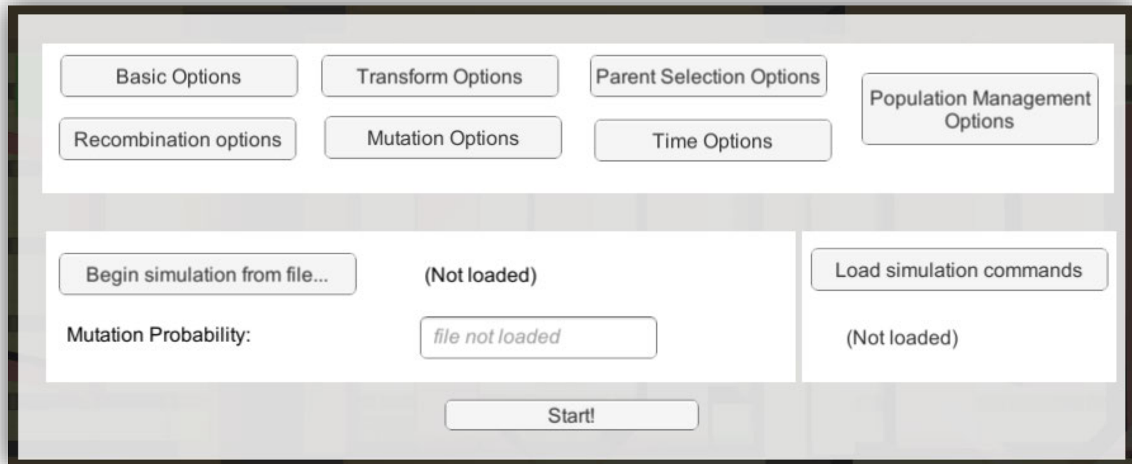

Figura 39: Panel principal

En la parte superior, encontramos un conjunto de botones que al pulsarlos permiten acceder a los paneles que están relacionados con la configuración de los parámetros del algoritmo evolutivo que se va a aplicar. Estos paneles se explican en las siguientes secciones. En la parte inferior tenemos unas de opciones especiales:

Cargar redes neuronales iniciales

Al pulsar el botón *Begin simulation from file…* se abre un cuadro de selección que permite al usuario seleccionar un directorio:

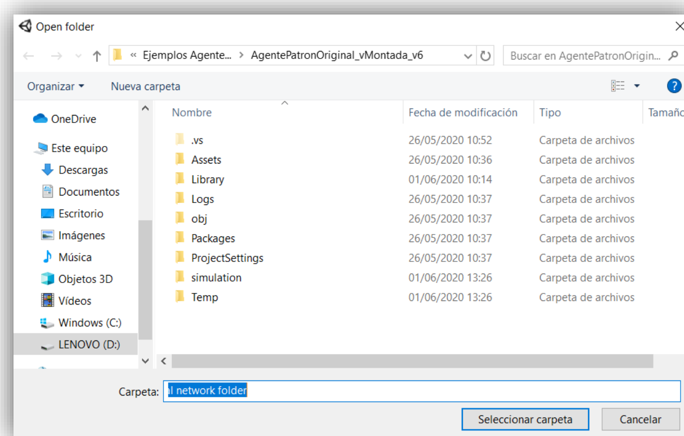

Figura 40: Cuadro de selección

Si el directorio seleccionado es válido, se indicará el mensaje *(Loaded)* en el panel. Cuando se arranque la simulación, en lugar de comenzar con redes neuronales aleatorizadas, comenzará con los ficheros xml de las redes neuronales que se encuentren en el directorio seleccionado. Si el número de individuos por generación es mayor al número de redes neuronales leídas, se duplicarán hasta completar la población de la primera generación. Por último, el usuario puede introducir una probabilidad de mutación a aplicar a las redes neuronales cargadas. Esta mutación se aplica una sola vez, antes de comenzar la simulación. Una vez iniciada, se utilizarán los parámetros establecidos en la parte superior del panel como una simulación normal.

Esta opción permite:

- Continuar una simulación incompleta. Puesto que el módulo almacena todas las redes neuronales de cada generación, si se necesita continuar una simulación, basta con seleccionar la carpeta de la última generación ejecutada, e indicar una probabilidad de mutación de $p_m = 0$.

- Búsqueda de mejor solución de un máximo local. Seleccionando un conjunto de redes neuronales que han dado buenos resultados en simulaciones anteriores, se puede favorecer la búsqueda de la mejor solución de ese máximo local.

Cargar comandos de simulaciones

Si el usuario desea ejecutar varias simulaciones de forma seguida, puede ser incómodo que tenga que manualmente iniciar cada una de ellas. Por ejemplo, si quisiera ejecutar más de una simulación mientras no está presente (por ejemplo, en el trabajo), no podría, puesto que una vez finalice la primera simulación, tendría que introducir los parámetros de la siguiente.

Esto se soluciona mediante un fichero de comandos. Podemos indicar al módulo un número indefinido de simulaciones con sus respectivos parámetros mediante un fichero xml con la siguiente sintaxis:

- La raíz del documento es la etiqueta *Commands*. Esta contiene cualquier cantidad de etiquetas *Command*:

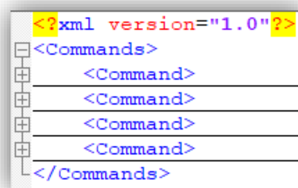

Figura 41: Etiqueta Command

- Dentro de cada etiqueta *Command* se encuentran todos los parámetros pertenecientes a una simulación:

```xml
<?xml version="1.0"?>
<Commands>
    <Command>
        <BasicPopulation>100</BasicPopulation>
        <BasicGenerations>100</BasicGenerations>
        <BasicNumParents1>False</BasicNumParents1>
        <BasicNumParents2>True</BasicNumParents2>
```

Figura 42: Fragmento de etiqueta Command

Estos parámetros tienen la misma estructura que el fichero xml que genera el módulo al arrancar una simulación, *input.xml*. Por lo tanto, el usuario puede copiar los parámetros que desea de ese fichero. También se aporta una plantilla llamada *commands.xml* que puede rellenar. Existe un parámetro adicional (que no contiene el fichero *input.xml*) que permite indicar el nombre de la carpeta de salida, tal y como se indica en *ANEXO II: EJEMPLO DE FICHERO COMPLETO COMMANDS.XML*.

Pulsando el botón *Load simulation commands* se abrirá un cuadro de diálogo que permite seleccionar el fichero xml que contiene estos comandos. Si se selecciona un fichero correcto, se mostrará el texto *(Loaded)* en el panel principal. El módulo recorrerá cada una de las etiquetas *command*, y ejecutará secuencialmente todas las simulaciones. Es importante indicar que, si se utiliza esta opción, se ignorarán todos los parámetros introducidos en la interfaz y sólo ejecutará los comandos del fichero.

4. Panel de opciones básicas

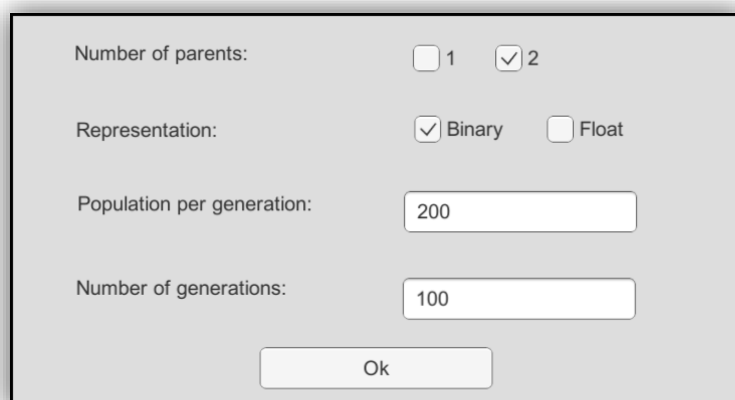

Figura 43: Panel de opciones básicas

En este panel se incluyen las opciones básicas de cualquier simulación. Estos parámetros deben ser los primeros que se elijan al arrancar el módulo:

- Número de padres: número de soluciones que se utilizarán para generar nuevos individuos. Si se selecciona 1, se ignorará el panel de opciones de recombinación.
- Tipo de representación a utilizar: binaria (*Binary*) o flotante (*Float*).
- Número de individuos por generación (*Population per generation*).
- Número de generaciones (*Number of generations*).

En los paneles de configuración, si se introduce un valor erróneo, mostrará una notificación que indique al usuario dicho error para que lo pueda corregir. Por ejemplo, si se introduce un número de individuos negativo, nos aparecerá el siguiente mensaje:

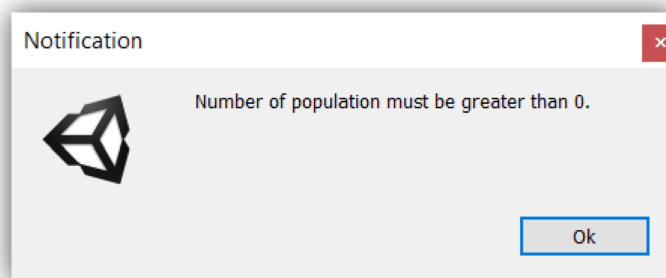

Figura 44: Mensaje de parámetro incorrecto

5. Panel de selección de posición inicial

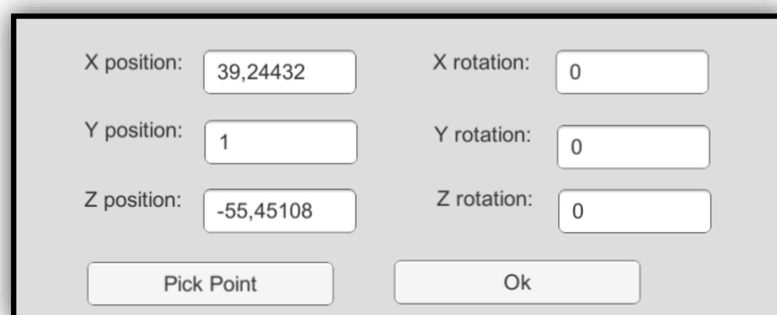

Figura 45: Panel de selección de posición

Mediante el panel de selección inicial indicamos la posición y rotación en la que los individuos comienzan cada generación. En lo referido a la posición, el usuario puede introducir manualmente las coordenadas, o seleccionar una posición determinada mediante el uso del ratón. Al hacer clic en el botón *Pick Point*, la interfaz del módulo desaparece, permitiendo al usuario hacer clic en la posición deseada. Para ello, es necesario que la cámara esté situada encima del escenario en el que se va a realizar la simulación.

Existen dos tipos de posiciones iniciales:

- Simple: todos los individuos de cada generación comienzan en la misma posición. Para seleccionar una opción simple con la función *Pick Point* basta con hacer un clic izquierdo en la posición deseada.

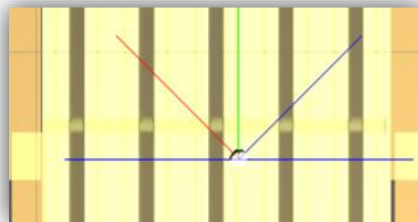

Figura 46: Ejemplo de posición inicial simple con 5 soluciones (los rayos son los elementos con los que perciben el entorno).

- Rango: los individuos comenzarán en una posición aleatoria contenida en un rango de posiciones. Para seleccionar un rango de posiciones con la función *Pick Point*, hay que hacer clic izquierdo en una posición y, sin soltar, arrastrar hasta la posición final. En el panel de opciones de posición podremos ver valores similares a los siguientes.

X position:	38,5->39,76	X rotation:	0
Y position:	1->1	Y rotation:	0
Z position:	-56,3->-54,8	Z rotation:	0
Pick Point		Ok	

Figura 47: Formato de selección de rango de posiciones

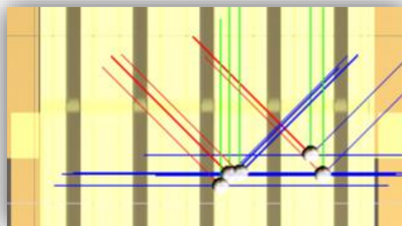

Figura 48: Ejemplo de posición inicial de rango con 5 soluciones

Los campos de rotación tienen la misma sintaxis que los de la posición, pero deben ser introducidos de forma manual. Por ejemplo, si deseamos que en el eje Y aparezcan con rotación aleatoria entre 315 y 45 grados, debemos indicar:

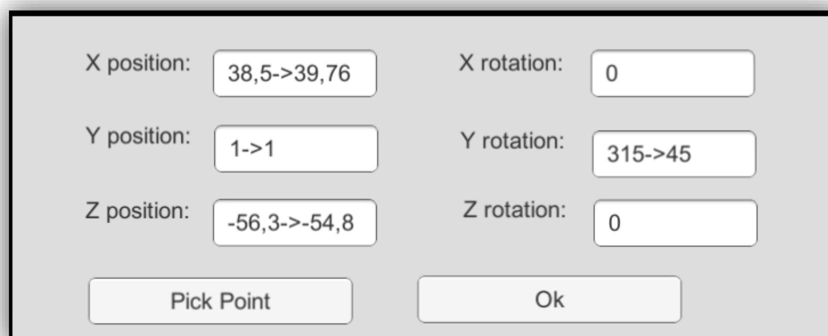

Figura 49: Formato de selección de rango de rotación

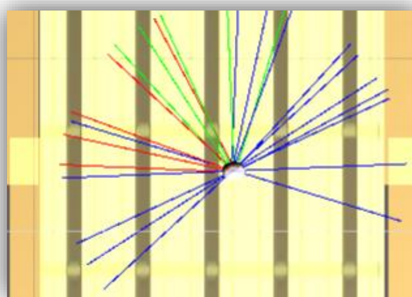

Figura 50: Ejemplo de rotación inicial de rango con un ángulo entre 315 y 45 grados, con 5 soluciones

6. Panel de selección de padres

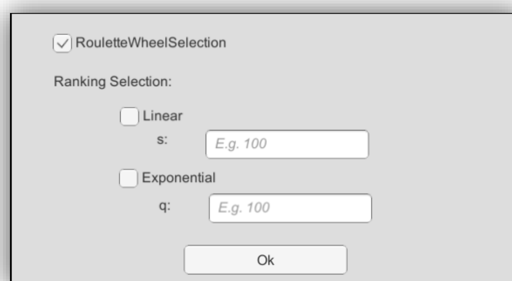

Figura 51: Panel de opciones de selección de padres

Este panel permite la selección del método de selección de padres. Las posibilidades son:

- Método basado en el fitness (*Roulette Wheel Selection)*.
- Método basado en la clasificación lineal (*linear ranking selection*).
- Método basado en la clasificación exponencial (*exponential ranking selection*).

Si se selecciona algún método basado en la clasificación, se debe indicar el parámetro s o q (lineal o exponencial respectivamente).

7. Panel de opciones de cruzamiento

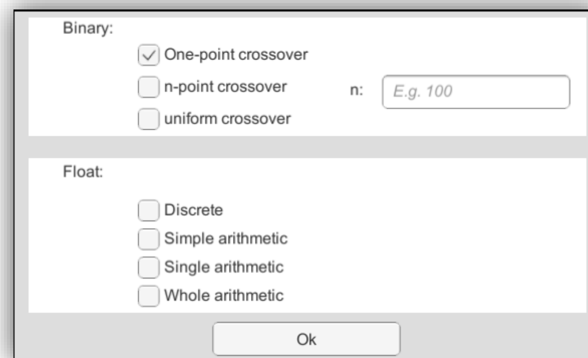

Figura 52: Panel de opciones de cruzamiento

Este panel nos permite seleccionar las distintas posibilidades de cruzamiento entre dos soluciones. Si en el panel de opciones básicas se indicó 1 como número de padres, este panel será ignorado. En función de la representación elegida en el panel básico, los posibles cruzamientos implementados son:

- Representación binaria: cruzamiento en un punto (*one-point crossover*), cruzamiento en n-puntos (*n-point crossover*) o cruzamiento uniforme (*uniform crossover*). Si se selecciona el cruzamiento en n-puntos, se debe indicar el número de cortes en los que se fragmentará el genotipo. Si este valor es mayor al número máximo de cortes posibles, no se indicará ningún error, simplemente se ajustará al valor máximo posible.

- Representación flotante: recombinación discreta (*discrete recombination*), recombinación aritmética simple (*simple arithmetic recombination*), recombinación aritmética individual (*single arithmetic recombination*) o recombinación aritmética completa (*whole arithmetic recombination*). Las recombinaciones aritméticas están implementadas con ($\alpha = 0.5$).

8. Panel de opciones de mutación

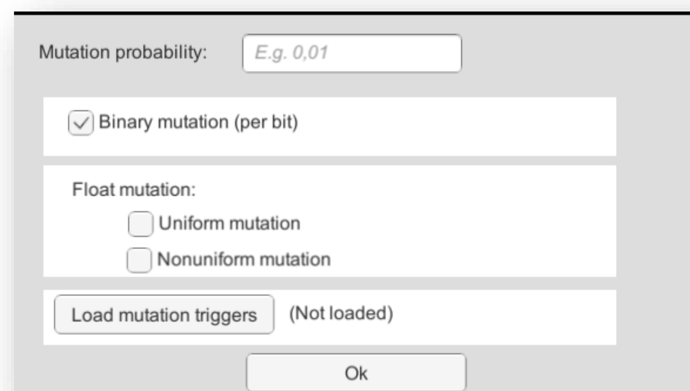

Figura 53: Panel de opciones de mutación

Este panel permite la selección de las opciones de mutación. En primer lugar, se debe seleccionar la probabilidad de mutación. A continuación, en función de la representación seleccionada, se puede elegir:

- Representación binaria: mutación a nivel de bit (*Binary mutation*).

- Representación flotante: mutación uniforme (*Uniform mutation*) o mutación no uniforme (*Nonuniform mutation*). Para la mutación no uniforme se utiliza una distribución normal estándar, cuyos valores se encuentran almacenados en un documento csv externo llamado *normal_distribution.csv*. Este documento forma parte del programa y NO debe ser eliminado o manipulado, puesto que produciría errores en la ejecución si se utiliza este tipo de mutación.

Por último, existe un botón que permite cargar *triggers de mutación*. Un trigger de mutación consiste en una condición que, de cumplirse, modificaría la probabilidad de mutación a otro valor. Esto permite modificar la probabilidad de mutación en función de cómo vaya progresando la simulación.

El usuario puede definir un número indefinido de triggers de mutación en un fichero con extensión xml. Este fichero tiene la siguiente sintaxis:

- La raíz del documento es una etiqueta llamada *MutationTriggers*. Esta puede contener cualquier número de etiquetas *Trigger*:

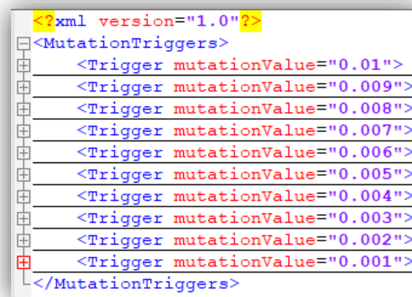

Figura 54: Etiqueta MutationTriggers

- Cada etiqueta *Trigger* contiene toda la información necesaria para inicializar un trigger. La sintaxis es la siguiente:

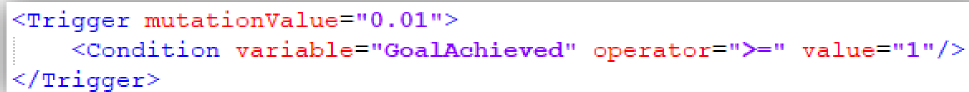

Figura 55: Etiqueta Trigger

El atributo *mutationValue* indica la probabilidad de mutación resultante si se activa el trigger. La etiqueta *Condition* indica la condición que debe se debe cumplir para que se produzca la activación. Pueden existir cualquier número de etiquetas *Condition* dentro de un mismo trigger. En ese caso, para que se active deben cumplirse las condiciones de todas ellas (AND condicional). Si se desea que cualquiera de las condiciones active el

trigger (OR condicional) basta con crear varias etiquetas trigger, cada una con una de las condiciones.

- Cada etiqueta *Condition* está formada por tres atributos:
    - *variable*: variable a utilizar en la condición. Los posibles valores son: *“MaxFitness”* (valor fitness máximo de la generación), *“AvgFitness”* (valor fitness medio de toda la generación), *“GoalAchieved”* (número de individuos que han alcanzado la meta en esta generación) o *“Generation”* (número de la generación).
    - *operator*: operador a utilizar en la condición. Los posibles valores son: “>” (*variable* mayor que *value*), “>=” (*variable* mayor o igual que *value*), “==” (*variable* igual que *value*), “<” (*variable* menor que *value*) o “<=” (*variable* menor o igual que *value*).
    - *value*: valor a utilizar en la condición. Debe ser siempre un valor numérico.

Por ejemplo, en la figura anterior, se define un trigger que establecerá la probabilidad de mutación a $p_m = 0.01$ si el número de individuos que alcanzan la meta es superior o igual a 1.

Los triggers serán evaluados al final de cada generación, antes de crear los individuos de la siguiente. Se evalúan en orden descendente, estableciendo el valor del último trigger que cumpla su condición. De esta forma, podemos ordenar los triggers de nuestra simulación en función de la prioridad de activación.

9. Panel de opciones de tiempo

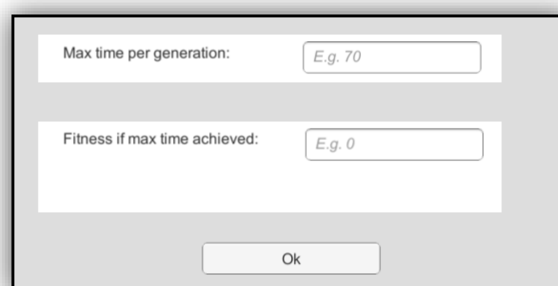

Figura 56: Panel de opciones de tiempo

El módulo considera que una generación finaliza cuando todos los agentes que la componen mueren o alcanzan la meta. Ante esta situación, en función del escenario, existe la posibilidad de que algunos agentes no mueran ni alcancen la meta nunca, puesto que, accidentalmente, su red neuronal está configurada para esto. Si esto ocurre, el módulo permanecería en esa generación incapaz de continuar.

Para solucionar esto, con el panel de opciones de tiempo, se establece el tiempo máximo que puede durar cada generación. Una vez transcurrido este tiempo, todos los agentes que sigan vivos son eliminados, recibiendo como valor fitness el especificado en este panel.

10. Panel de opciones de gestión de la población

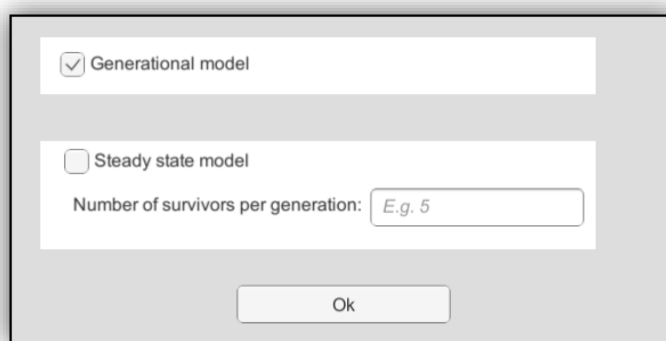

Figura 57: Panel de opciones de gestión de la población

Se han implementado dos métodos de gestión de la población:

- Modelo generacional (*Generational model*): se reemplazan todos los individuos al comenzar una nueva generación.
- Modelo estado-estacionario (*Steady-state model*): Se conserva un fragmento de la población. El método de selección de los supervivientes está basado en el fitness, conservándose los individuos que mejor valor hayan alcanzado. Por ejemplo, si se indican 5 supervivientes de una población de 100 soluciones, en el proceso de calcular

la siguiente generación, los 5 mejores individuos se copian en la nueva generación y los 95 restantes se calculan siguiendo los parámetros indicados.

11. Panel de información de la simulación

Una vez arrancada la simulación, se muestra un panel con la siguiente información:

- Número de la generación.
- Tiempo transcurrido de la generación.
- Número de soluciones vivas.
- Máximo número de individuos que han alcanzado la meta (en toda la simulación).
- Máximo fitness alcanzado (en toda la simulación).

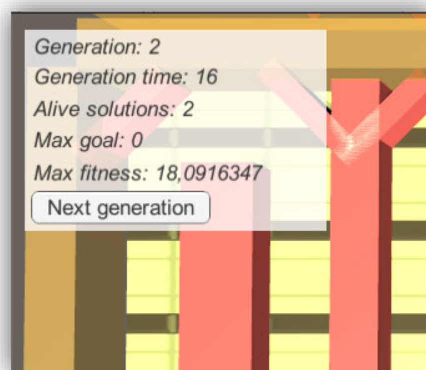

Figura 58: Panel de información de la simulación

Además, el panel dispone de un botón llamado *Next generation*. Si el usuario observa que las soluciones restantes de la generación no son útiles, puede pulsar este botón para pasar directamente a la siguiente generación. El fitness asignado a estos individuos es el indicado en el panel de opciones del tiempo. [20]

4. Estructura del proyecto

El proyecto está compuesto por un conjunto de clases desarrolladas en C#. Existen dos tipos:

- Editables por el usuario: clases que el usuario que desea implementar el módulo debe editar para completar la unión aplicación/módulo. Estas clases se proporcionan como plantilla, especificando fácilmente el código que debe incluir. Actualmente existen dos clases editables: *GeneticBehaviour.cs* y *GeneticManager.cs*. En el siguiente apartado se describen los aspectos más fundamentales de estas clases. En *APÉNDICE A: MANUAL DE USUARIO* se detalla cómo debe editarse para una correcta unión aplicación-módulo.

- No editables por el usuario: conjunto de clases que componen el proyecto que no deben ser editadas. El usuario las añade al proyecto al importar el módulo, y queda transparente el funcionamiento de estas. Sólo se deben modificar si se desea realizar alguna expansión del módulo, como se describe en el apartado de *TRABAJOS FUTUROS*.

Seguidamente, se describe brevemente las clases que componen el proyecto.

1. Clases principales

A continuación, se detallan las clases más importantes del proyecto. Estas clases hacen uso del resto de clases descritas en los siguientes apartados:

- *CanvasManager.cs*: script encargado de gestionar los paneles iniciales de la simulación. Se encarga de la lectura y escritura del fichero *input.xml*, al igual que de la comprobación de los parámetros que introduce el usuario. Una vez el usuario los ha introducido y decide comenzar la simulación, esta clase pasa todos los parámetros a la clase *GeneticManager.cs* y le indica que comienza la simulación.

- *GeneticManager.cs*: script encargado de todo el proceso de simulación. Por cada generación, inicializa los nuevos agentes, espera a que la generación finalice y crea nuevas soluciones a partir de los resultados obtenidos. Para que pueda instanciar los

agentes, es necesario que disponga del prefab del mismo. Esta es la única modificación que debe realizar el usuario en esta clase.

- *GeneticBehaviour.cs*: define el comportamiento del agente. Esta clase dispone de una red neuronal y debe ser completada por el usuario, definiendo los valores de entrada de la red e interpretando los valores de salida. De esta forma, el individuo será dirigido por esta red neuronal. También dispone de una instanciación de la clase *GeneticEventMessage.cs* que utilizará para comunicarse con el *GeneticManager.cs*.

- *GeneticEventMessage.cs*: clase que permite la comunicación entre *GeneticManager* y *GeneticBehaviour*. Dispone de los siguientes métodos, que debe utilizar el usuario en su implementación:

    - *getGenerationTime()*: permite obtener el tiempo actual de la generación, en segundos. El uso de este método no es obligatorio.

    - *goalAchieved()*: indica al *GeneticManager* que la solución correspondiente ha alcanzado la meta. El uso de este método sí es obligatorio.

    - *kill()*: indica al *GeneticManager* que la solución debe retirarse de la generación. Esto no significa que haya fracasado. El fracaso o éxito de la solución sólo se indica en función del objetivo. El uso de este método también es obligatorio.

Las siguientes figuras muestran el funcionamiento de estas clases:

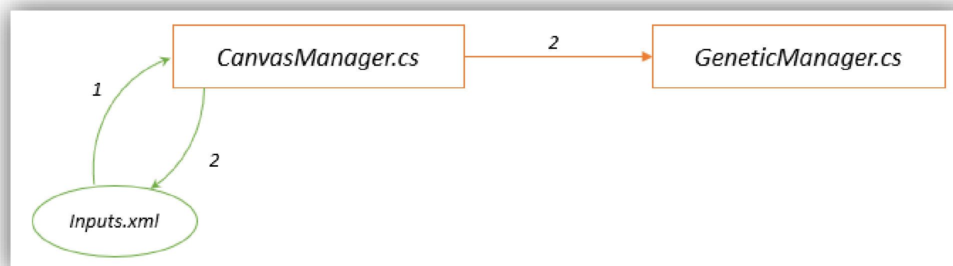

Figura 59: Comunicación entre CanvasManager.cs y GeneticManager.cs

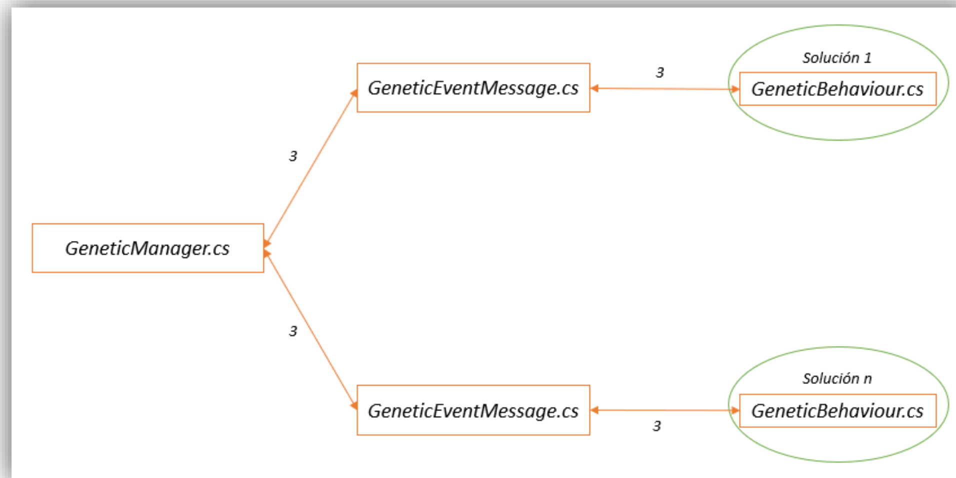

Figura 60: Comunicación entre GeneticManager.cs y los agentes

1. *CanvasManager.cs* carga los parámetros de la simulación anterior en los paneles.
2. Comienza la simulación. Escribe los parámetros de la actual simulación en *input.xml* y pasa estos parámetros a *GeneticManager.cs*.
3. Durante la simulación, *GeneticManager.cs* mantiene la comunicación con los agentes a través de *GeneticEventMessage.cs*.

2. Clases de redes neuronales simples

En el proyecto se diferencian dos tipos de redes neuronales: redes neuronales simples y redes neuronales NEAT. Las redes neuronales conocidas en este proyecto como simples son aquellas cuya topología no varía durante el algoritmo evolutivo. Las redes neuronales NEAT son aquellas que están implementadas para poder modificar su topología a lo largo del algoritmo genético. Pese a que en el proyecto actual no están implementadas las redes neuronales NEAT, se proporciona un código de estas, posibilitando la incorporación de estas redes al módulo como trabajo futuro.

Las clases que gestionan las redes neuronales simples son las siguientes:

- *SimpleNode.cs*: almacena y procesa la información de un nodo de una red neuronal.

- *SimpleLayer.cs*: gestiona una capa de la red neuronal.

- *SimpleNeuralNetwork.cs*: gestiona una red neuronal completa. El usuario debe hacer uso de la instanciación de esta clase en *GeneticBehaviour.cs* para definir las entradas e interpretar las salidas. Para obtener la salida de la red neuronal se proporciona el método *processInput(List<double>)*. Este método recibe como parámetro una lista de valores *double* como entrada a la red y devuelve la salida de esta.

  Por otro lado, una vez haya finalizado la simulación, si el usuario quiere incorporar la red neuronal al componente de su proyecto, debe instanciar un objeto de esta clase e inicializarlo con los pesos deseados. Para ello debe:

  - Instanciar el objeto, pasando como parámetros del constructor la topología de la red neuronal:

    SimpleNeuralNetwork(int inputNodes, int hiddenNodes, int outputNodes, int hiddenLayers)

  - Llamar al método *setWeightsAndBiasList(List<double>)* pasando como parámetro la lista de pesos y bias. Para obtener esta lista de un fichero de red neuronal, se utiliza el método estático *readSimpleNeuralNetworkWeightsAndBias(string path)* de la clase *SimpleNeuralNetworkReaderManager.cs*.

- *SimpleNeuralNetworkWriterManager.cs*: se encarga de la escritura de las redes neuronales de cada generación. También es responsable de crear los directorios *simulation_XX* y *genXX*.

- *SimpleNeuralNetworkReaderManager.cs*: permite la lectura de un fichero xml (que contenga la información de la red neuronal) y devuelve una lista que contiene todos los pesos y bias. También puede leer todos los ficheros de un determinado directorio, devolviendo una lista por cada red leída. El usuario debe utilizar esta clase si desea inicializar una red neuronal con los pesos de un fichero xml.

3. Clases de redes neuronales NEAT

Como se ha mencionado en el apartado anterior, las redes NEAT no son usadas en el proyecto, pero se proporciona un código que permite implementarlas como trabajo futuro. Las clases proporcionadas son las siguientes:

- *NEATNode.cs*: gestiona la información de un nodo de la red.
- *NEATNodeType.cs*: enum de los diferentes tipos de nodos de la red (*Input, Hidden* u *Output*).
- *NEATConnection.cs*: almacena y gestiona la información de una conexión de la red.
- *NEATNeuralNetwork.cs*: Almacena y gestiona las conexiones y los nodos de la red neuronal. Dispone de métodos que permiten la incorporación de nuevos nodos y la creación de nuevas conexiones, permitiendo modificar su topología.

4. Clases para la gestión de agentes muertos

Es importante recordar que existen dos posibles finales para cada individuo en cada generación: éxito o fracaso. Durante la ejecución de una generación se almacena la información de los agentes que se retiran de la generación (cuando se llama al método *kill()* de *GeneticEventMessage()*), tanto si han alcanzado la meta como si han fracasado. Una vez finalizada la generación, se utiliza para la creación de la siguiente. Esta información se almacena y se gestiona en las siguientes clases:

- *DeadPlayerInformation.cs*: almacena la red neuronal y el fitness de un agente.
- *DeadPlayerInformationManager.cs*: gestiona una lista de *DeadPlayerInformation* que contiene la información de toda una generación. La lista se ordena en función del fitness.

5. Clases para la recombinación de redes

Las siguientes clases son utilizadas para realizar el cruzamiento entre dos soluciones:

- *RecombinationBinary.cs*: dispone de varios métodos estáticos que permiten obtener las siguientes recombinaciones binarias entre dos soluciones: cruzamiento en un punto, cruzamiento en n-puntos y cruzamiento uniforme.

- *RecombinationFloat.cs*: dispone de varios métodos estáticos que permiten obtener las siguientes recombinaciones flotantes entre dos soluciones: recombinación discreta, recombinación aritmética simple, recombinación aritmética individual y recombinación aritmética completa.

6. Clases para la mutación de redes

Para realizar la mutación de redes neuronales se utilizan las siguientes clases:

- *MutationBinary.cs*: realiza la mutación binaria de una red neuronal.

- *MutationFloat.cs*: permite aplicar un conjunto de mutaciones flotantes a una red neuronal: mutación uniforme y mutación no uniforme.

- *MutationFloatNormalDistributionReader.cs*: los valores de la distribución normal utilizada en la mutación no uniforme se encuentran en un fichero externo con extensión *.csv*. Esta clase carga estos valores, proporcionando métodos para la obtención de los correspondientes valores de la distribución en función de la probabilidad.

- *NormalDistributionValue.cs*: almacena la información de un valor de la distribución normal (valor y probabilidad).

Todos los métodos de estas clases reciben una solución como parámetro y devuelven la misma red tras aplicar la mutación.

7. Clases para la selección de padres

Para la selección de padres, se han codificado las siguientes clases:

- *ParentSelectorRouletteWheel.cs*: selección de padres basados en el fitness.

- *ParentSelectorRankingLinear.cs*: selección de padres mediante clasificación lineal.

- *ParentSelectorRankingExponential.cs*: selección de padres mediante clasificación exponencial.

Estas clases son inicializadas aportando la lista de toda la generación junto con el fitness de cada individuo (y parámetros adicionales como s o q para los métodos basados en la clasificación). Una vez inicializadas, estas clases devuelven padres aplicando el método correspondiente.

8. Clases para la lectura de triggers de mutación

Las clases encargadas de la lectura y almacenamiento de los triggers de mutación son las siguientes:

- *MutationCondition.cs*: almacena la información de una condición de trigger.

- *MutationConditionOperator.cs*: enum de los diferentes tipos de operadores posibles de una condición.

- *MutationConditionVariable.cs*: enum de las diferentes variables posibles en una condición.

- *MutationTrigger.cs*: almacena la información de un trigger de mutación. Dispone un método que realiza la comprobación de las condiciones del trigger.

- *MutationTriggerReader.cs*: clase encargada de la lectura del fichero de triggers, devolviendo una lista de *MutationTrigger*.

9. Clases para el manejo de números binarios

Para gestionar los números binarios se aportan las siguientes clases:

- *DoubleBinaryManager.cs*: gestiona un número flotante como binario. Permite aplicar mutaciones basados en una probabilidad pasada por parámetro.

- *DoubleBinaryDNAManager.cs*: realiza la conversión del genotipo de una red neuronal de lista de números flotantes a *string* de bits y viceversa.

Estas clases son utilizadas por las clases de mutación y recombinación de representación binaria (*RecombinationBinary.cs* y *MutationBinary.cs*).

10. Otras clases

- *GenerationInformation.cs*: almacena la información relacionada con una generación.
- *CSVWriter.cs*: clase encargada de la escritura del fichero *simulationResume.csv*. La información la obtiene de una lista de *GenerationInformation*.
- *Command.cs*: almacena los parámetros de un comando para una simulación.
- *SimulationCommandManager.cs*: lee los comandos de un fichero xml y los almacena en una lista de *Command*.
- *InputSaver.cs*: almacena los parámetros de una simulación en el fichero *input.xml*.
- *InputReader.cs*: lee los parámetros de la simulación anterior del fichero *input.xml*. [21]

PRUEBAS Y RESULTADOS I: PRUEBAS DE CONVERGENCIA

1. Introducción

En los siguientes apartados se desarrollan las diferentes pruebas que se han realizado para comprobar el funcionamiento del módulo desarrollado. Debido al factor aleatorio, es posible que dos simulaciones con los mismos parámetros den lugar a resultados diferentes. Para poder tomar una mejor perspectiva de los resultados, se realiza cada simulación tres veces, mostrando la media de los resultados.

La aplicación utilizada para realizar las pruebas consiste en unos agentes (soluciones de la aplicación) que deben esquivar los muros para finalmente alcanzar la meta. Si un agente toca un muro, se considera que ha muerto, y por lo tanto desaparece del mapa:

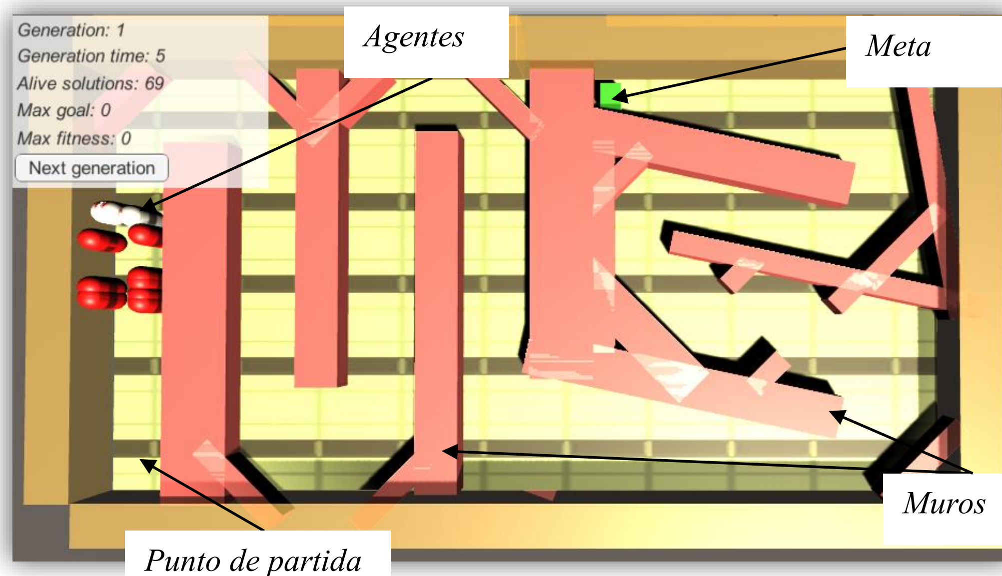

Figura 61: Escenario de pruebas con agentes

Los agentes siempre están avanzando y pueden girar en dos direcciones (izquierda y derecha) o no girar. Por otro lado, se han establecido cinco rayos por los cuales perciben su entorno. La información que devuelven estos rayos es la siguiente:

- Si detecta muro, devuelve la distancia a ese muro,
- Si no detecta nada, devuelve la longitud del rayo (10), y
- Si detecta la meta, devuelve también la longitud del rayo (10).

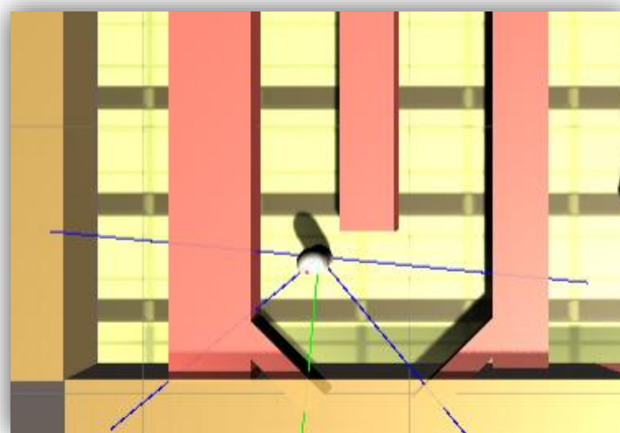

Figura 62: Rayos de los agentes

De esta forma, la máxima distancia posible la detectará cuando no haya ningún muro o cuando esté detectando la meta. Por lo tanto, el objetivo del entrenamiento consiste en obtener redes neuronales que aprendan a interpretar las distancias proporcionadas, esquivando los obstáculos si los agentes se aproximan más de lo debido.

Ante esta situación se opta por utilizar una red neuronal con cinco nodos de entrada (uno para cada rayo) y tres nodos de salida. En función de que nodo de salida se active, se tomará la decisión del giro. Para las capas intermedias se ha decidido utilizar una capa formada por cuatro nodos.

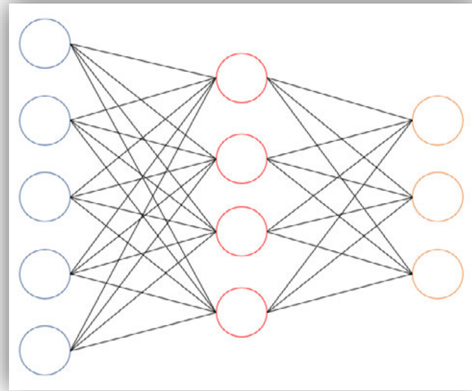

Figura 63: Red neuronal utilizada para PRUEBAS Y RESULTADOS I

Para favorecer el avance de los agentes a lo largo de las generaciones, se establece un fitness proporcional al tiempo que los agentes se mantienen vivos. Por otro lado, el principal objetivo de la aplicación consiste en que los agentes alcancen la meta en el menor tiempo posible. Para conseguir esto, se establece un fitness diferente en función de si el agente ha alcanzado la meta o no:

- Si el agente no ha alcanzado la meta, todo el tiempo posible que se haya conseguido mantener con vida es positivo. Por lo tanto, el fitness se calcula como:

  $$fitness = tiempo_{vida}$$

  Siendo $tiempo_{vida}$ el tiempo en segundos que el agente se mantiene vivo.

- Si el agente ha alcanzado la meta, cuanto más tiempo haya tardado peor fitness debe alcanzar. Por lo tanto, se otorga una puntuación extra por alcanzar la meta, pero se penaliza el tiempo de vida:

  $$fitness = bonus_{meta} - tiempo_{vida}$$

  Para que este método sea efectivo, el bonus por alcanzar la meta debe ser muy superior al del tiempo de vida. De no ser así, los agentes que mueren antes de alcanzar la meta obtendrían un mejor fitness que los que sí la alcanzan, convirtiéndose la meta en algo negativo que los agentes aprenderían a esquivar. En nuestro escenario, el bonus

otorgado a los agentes que alcanzan la meta es de 1500 y el tiempo que tardan en alcanzarla, aproximadamente 50 segundos.

Todas las pruebas se realizan en el mismo escenario para facilitar la comparación de resultados.

El objetivo de estas pruebas consiste en la comparación de las diferentes combinaciones de parámetros ante una misma aplicación. Esto no implica que otros parámetros puedan demostrar mayor efectividad ante otro tipo de proyectos.

Es de notable importancia indicar cuánto generaliza la red neuronal. En este escenario, ante los valores de entrada que recibe, la red aprende únicamente a esquivar obstáculos. Por lo tanto, si se utilizara un escenario más abierto, donde el agente tuviera más libertad de movimiento, no se observaría convergencia, puesto que no dispone de información relativa a la posición del objetivo. En el siguiente capítulo se explora la solución a este problema con un escenario y red neuronal diferente.

2. Pruebas de recombinación binaria

A continuación, se muestran las pruebas realizadas con la representación binaria y diferentes métodos de recombinación. Para estas las pruebas se mantienen los siguientes parámetros fijos, modificando únicamente el tipo de recombinación

- Triggers de mutación: no se utilizan triggers de mutación.
- Agentes por generación: 200.
- Número de generaciones: 100.
- Probabilidad de mutación: 0.01.
- Numero de padres: 2.
- Gestión de la población: estado-estacionario con 1 superviviente.
- Selección de padres: basado en el fitness.

1. Prueba 1: cruzamiento en un punto

La media del número de agentes que alcanzaron la meta de las tres simulaciones con cruzamiento en un punto es la siguiente:

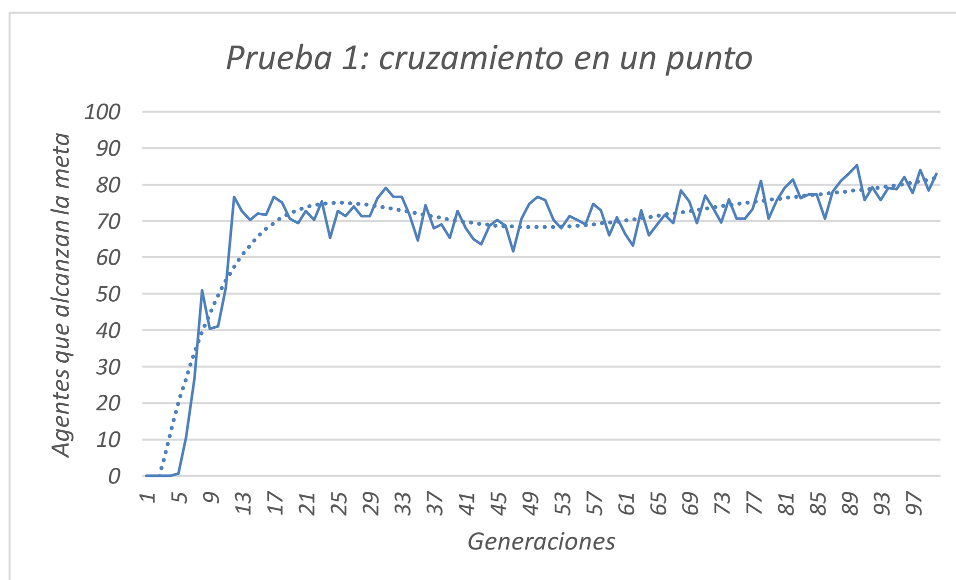

Figura 64: Resultados prueba 1: cruzamiento en un punto

2. Prueba 2: cruzamiento en n-puntos

La media del número de agentes que alcanzaron la meta de las tres simulaciones con cruzamiento en n-puntos (con $n = 100$) es la siguiente:

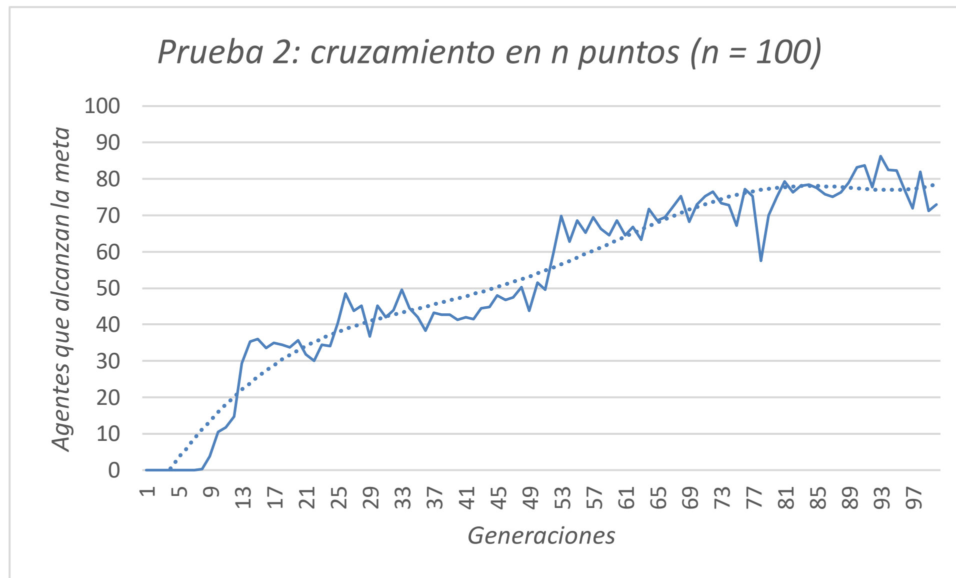

Figura 65: Resultados prueba 2: cruzamiento en n puntos

3. Prueba 3: cruzamiento uniforme

La media del número de agentes que alcanzaron la meta de las tres simulaciones con cruzamiento uniforme es la siguiente:

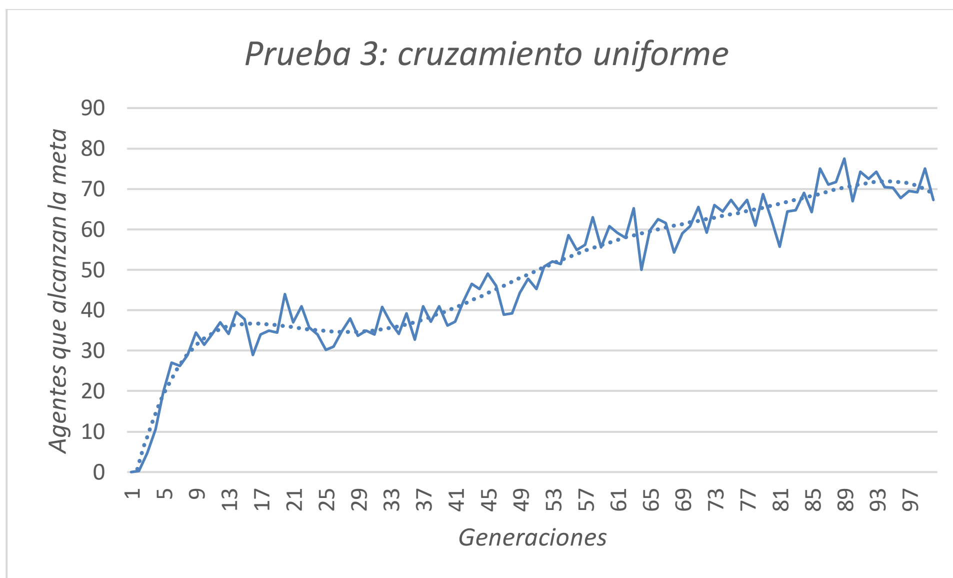

Figura 66: Resultados prueba 3: cruzamiento uniforme

4. Conclusiones parciales

Comparando las líneas de tendencias de las tres pruebas realizadas previamente obtenemos:

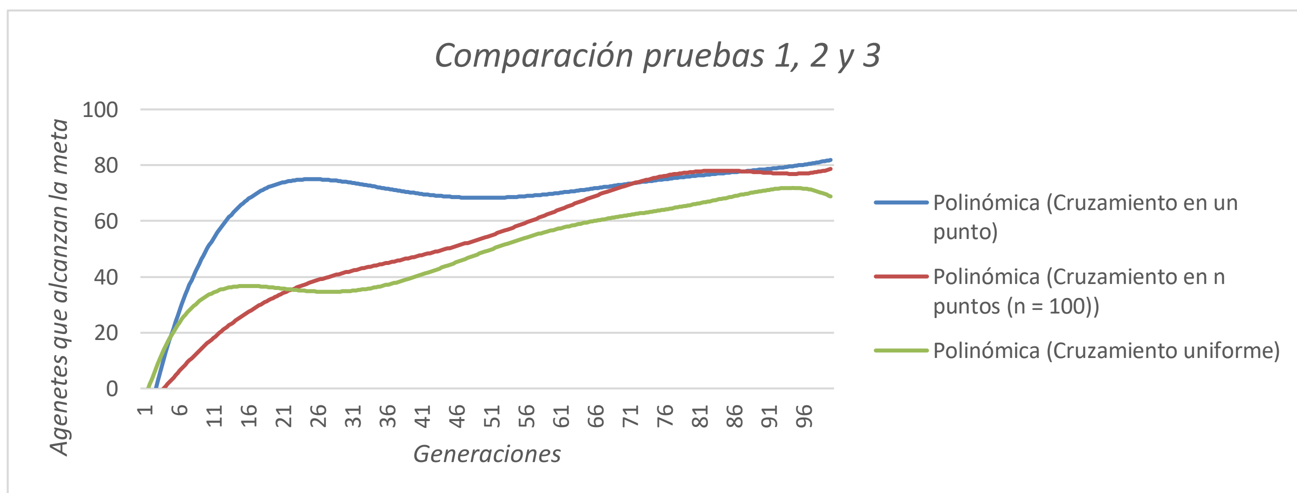

Figura 67: Comparación pruebas 1, 2 y 3

De la gráfica anterior no podemos extraer conclusiones relativas a cuál cruzamiento es más apropiado, aunque es necesario indicar que el cruzamiento en un punto destaca en las primeras generaciones. Los tres tipos de recombinación muestran convergencia al final de las 100 generaciones.

3. Pruebas de recombinación flotante

A continuación, se muestran un conjunto de pruebas que permiten comparar los diferentes métodos de recombinación flotante. Para ello, utilizaremos los siguientes parámetros:

- Representación: flotante.
- Número de padres: 2
- Individuos por generación: 200
- Número de generaciones: 100
- Selección de padres: método basado en el fitness.
- Mutación: flotante uniforme.
- Gestión de la población: estado estacionario con 1 superviviente.
- Probabilidad de mutación inicial: 0.05.
- Triggers de mutación: ninguno.

1. Prueba 4: recombinación discreta

La media del número de agentes que alcanzaron la meta de las tres simulaciones con recombinación discreta es la siguiente:

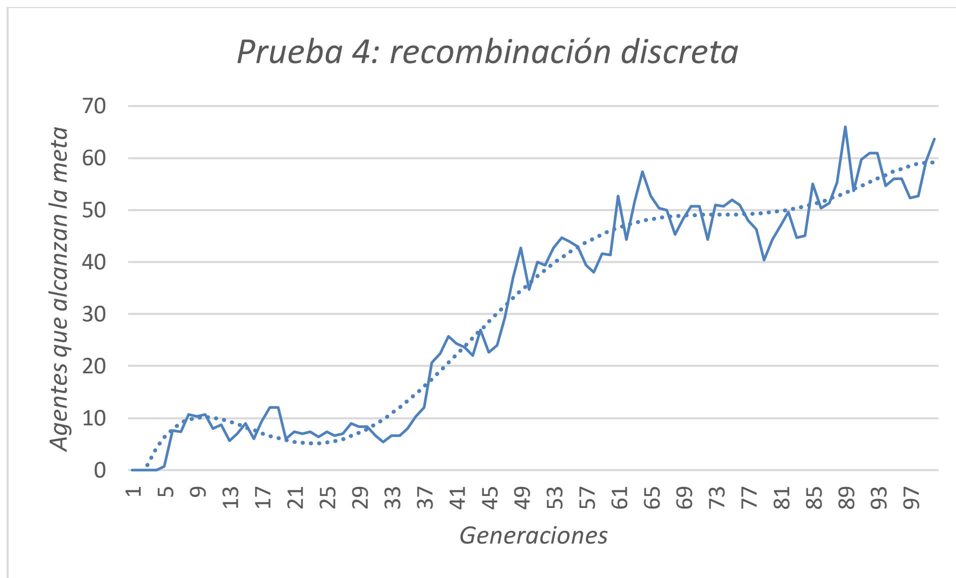

Figura 68: Resultados prueba 4: recombinación discreta

2. Prueba 5: recombinación aritmética simple

La media del número de agentes que alcanzaron la meta de las tres simulaciones con recombinación aritmética simple es la siguiente:

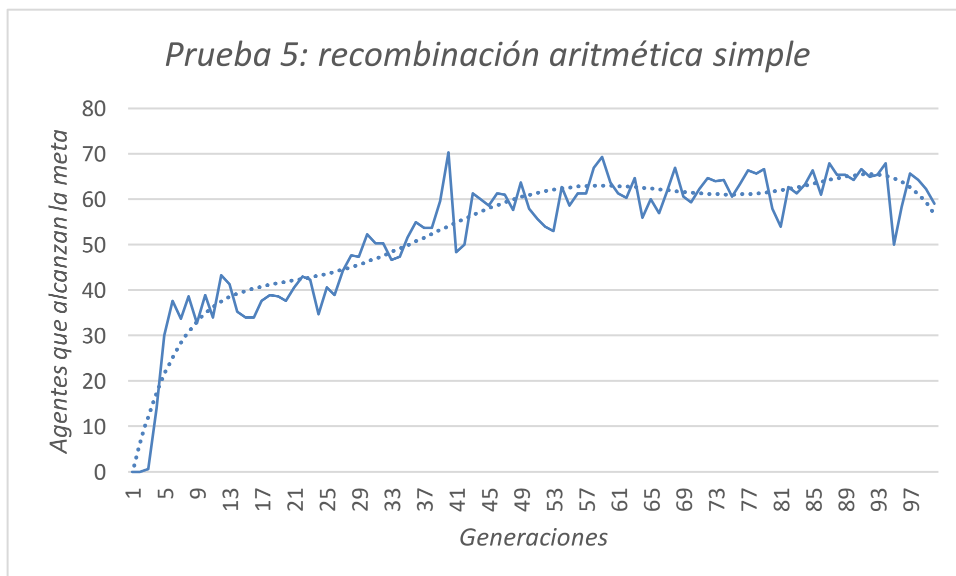

Figura 69: Resultados prueba 5: recombinación aritmética simple

3. Prueba 6: recombinación aritmética individual

La media del número de agentes que alcanzaron la meta de las tres simulaciones con recombinación aritmética individual es la siguiente:

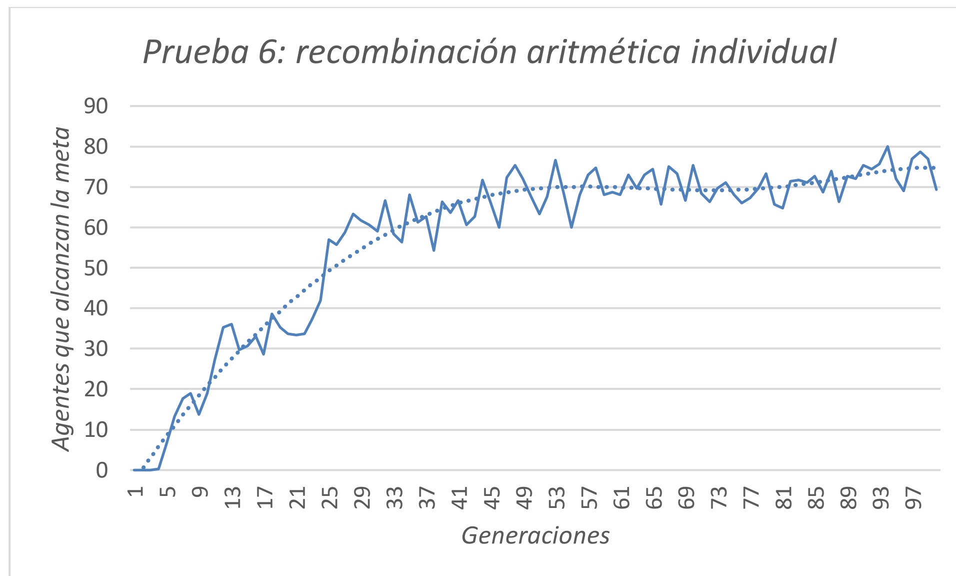

Figura 70: Resultados prueba 6: recombinación aritmética individual

4. Prueba 7: recombinación aritmética completa

La media del número de agentes que alcanzaron la meta de las tres simulaciones con recombinación aritmética completa es la siguiente:

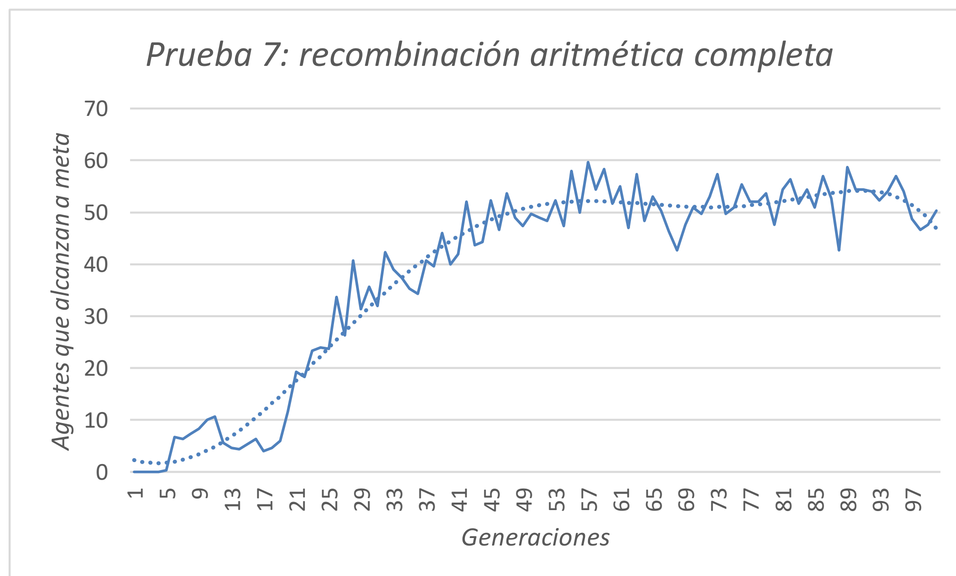

Figura 71: Resultados prueba 7: recombinación aritmética completa

5. Conclusiones parciales

Comparando las líneas de tendencia de las cuatro pruebas realizadas previamente tenemos:

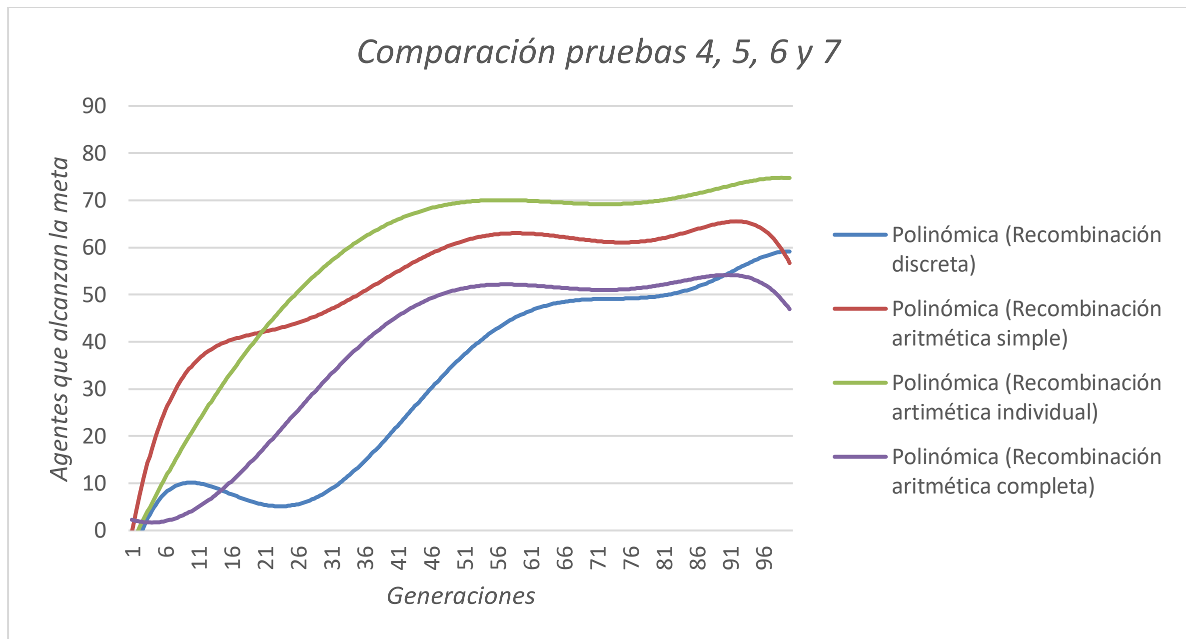

Figura 72: Comparación pruebas 4, 5, 6 y 7

Observando la gráfica, se puede notar que las pruebas con recombinación aritmética simple y aritmética individual muestran ligeramente mejores resultados que los otros dos métodos de recombinación. De cualquier forma, los cuatro tipos de cruzamiento muestran un buen resultado tras las 100 generaciones.

4. Pruebas de mutación discreta

Para comparar los dos tipos de mutación flotante existente se utilizarán los siguientes parámetros en las pruebas 8 y 9:

- Representación: flotante.
- Número de padres: 2
- Individuos por generación: 200
- Número de generaciones: 100
- Selección de padres: método basado en el fitness.
- Recombinación: aritmética simple.
- Gestión de la población: estado estacionario con 1 superviviente.
- Probabilidad de mutación inicial: 0.05.
- Triggers de mutación: ninguno.

1. Prueba 8: mutación uniforme

La prueba 8, al utilizarse mutación flotante uniforme, es idéntica a la prueba 5. Por lo tanto, se utilizarán los datos obtenidos de la prueba 5 como prueba 8.

2. Prueba 9: mutación no uniforme

La media del número de agentes que alcanzaron la meta de las tres simulaciones con mutación no uniforme es la siguiente:

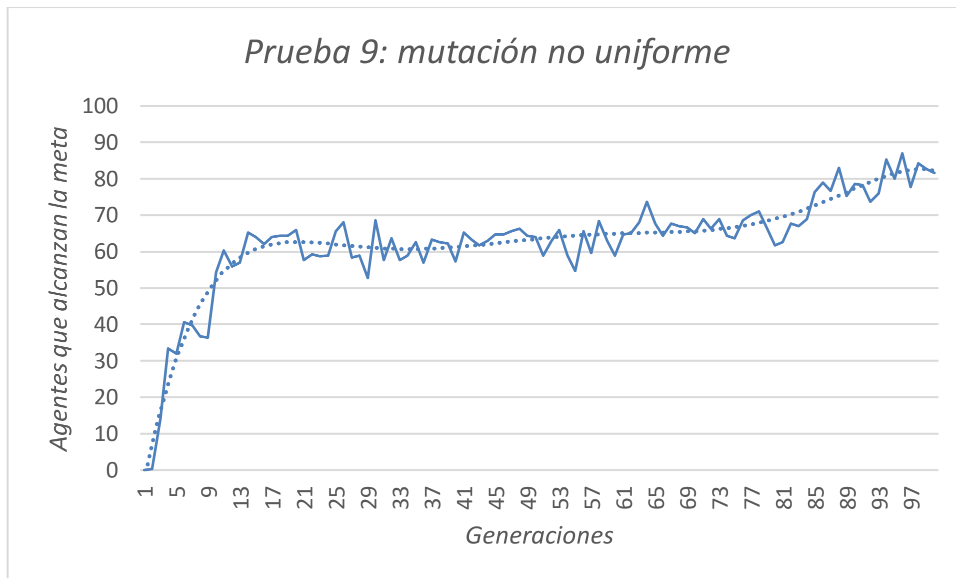

Figura 73: Resultados prueba 9: mutación no uniforme

3. Conclusiones parciales

Comparando las líneas de tendencia tenemos:

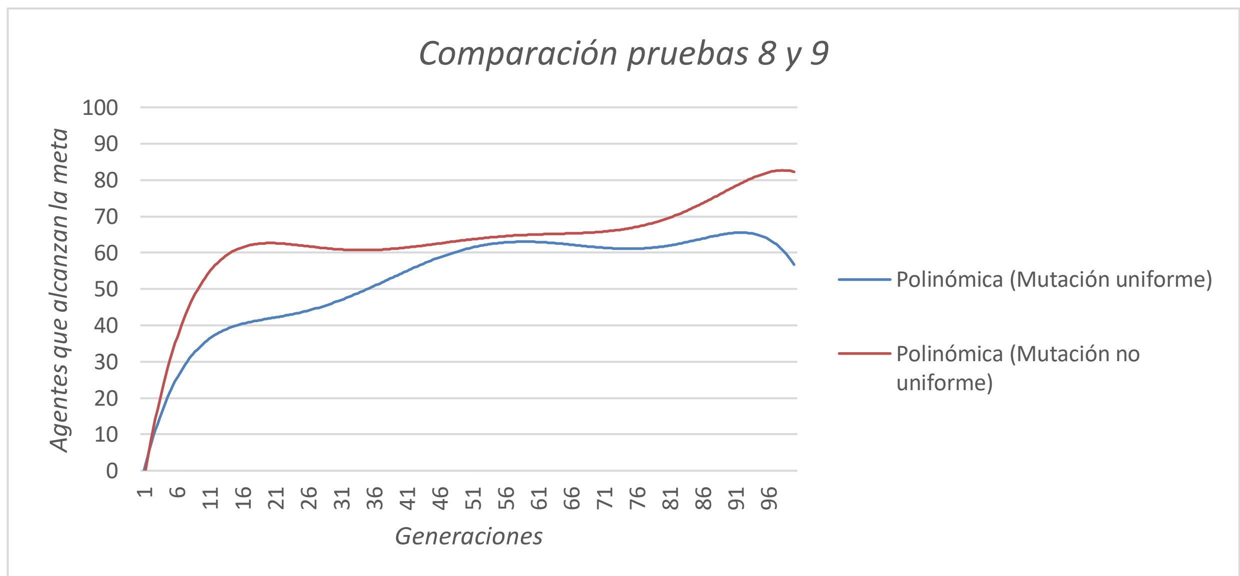

Figura 74: Comparación pruebas 8 y 9

No existen grandes diferencias en los resultados obtenidos, aunque la mutación no uniforme muestra una ligera mejora respecto a la mutación uniforme. Ambos tipos de mutación muestran buenos resultados tras 100 generaciones de simulación.

5. Pruebas de selección de padres

Para comparar los diferentes tipos de selección de padres existentes se utilizarán los siguientes parámetros en las pruebas 10, 11 y 12:

- Representación: flotante.
- Número de padres: 2
- Individuos por generación: 200
- Número de generaciones: 100
- Recombinación: aritmética simple.
- Mutación: uniforme
- Gestión de la población: estado estacionario con 1 superviviente.
- Probabilidad de mutación inicial: 0.05.
- Triggers de mutación: ninguno.

1. Prueba 10: Selección basada en el fitness

Al utilizar selección basada en el fitness, la prueba 10 es idéntica a la prueba 5. Por lo tanto, se utilizarán estos resultados como los de la prueba 10.

2. Prueba 11: Selección basada en el ranking (lineal)

La media del número de agentes que alcanzaron la meta de las tres simulaciones con selección basada en el ranking lineal ($s = 2$) es la siguiente:

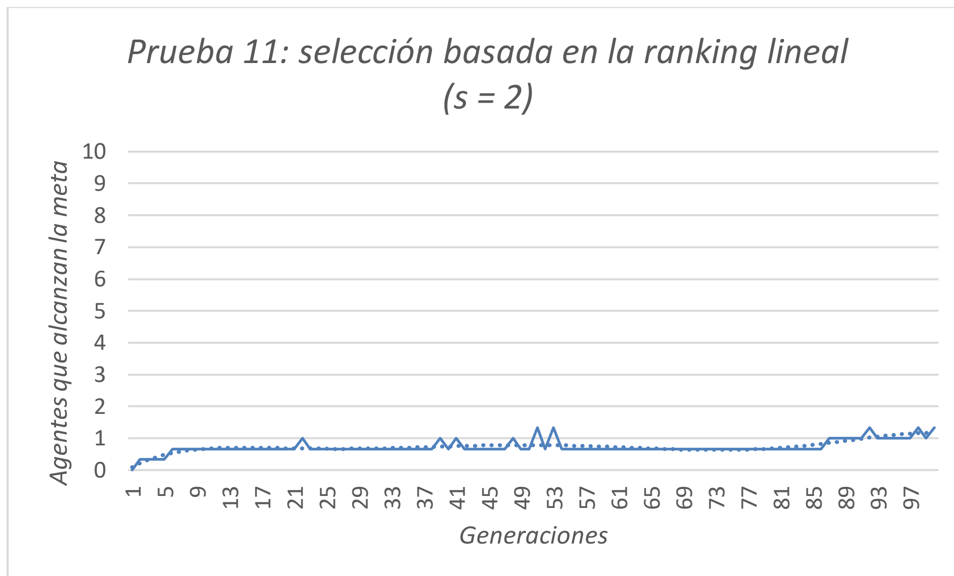

Figura 75: Resultados prueba 11: selección basada en la ranking lineal (s = 2)

3. Prueba 12: Selección basada en el ranking (exponencial)

La media del número de agentes que alcanzaron la meta de las tres simulaciones con selección basada en el ranking exponencial ($q = 0.5$) es la siguiente:

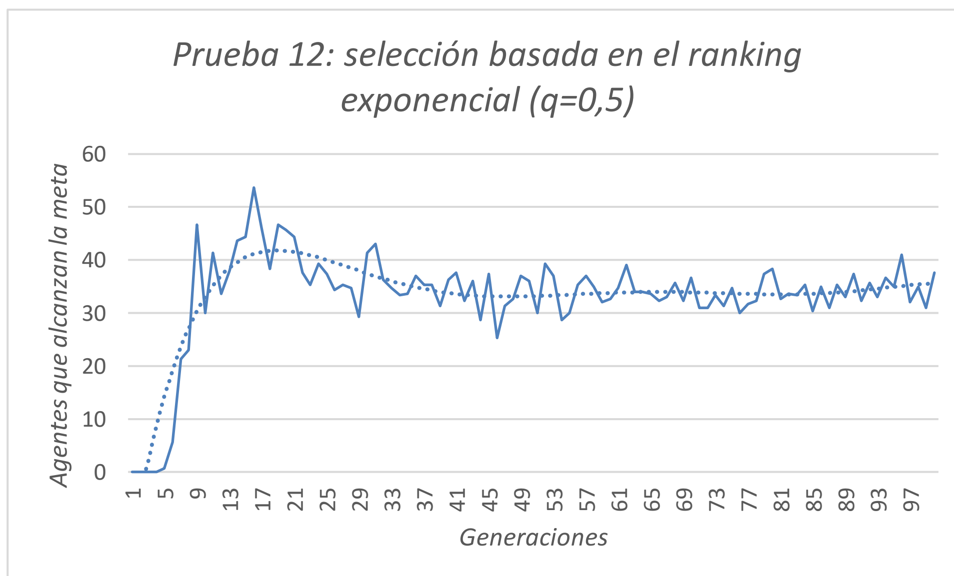

Figura 76: Resultados prueba 12: selección basada en el ranking exponencial (q = 0,5)

4. Conclusiones parciales

Comparando las líneas de tendencia:

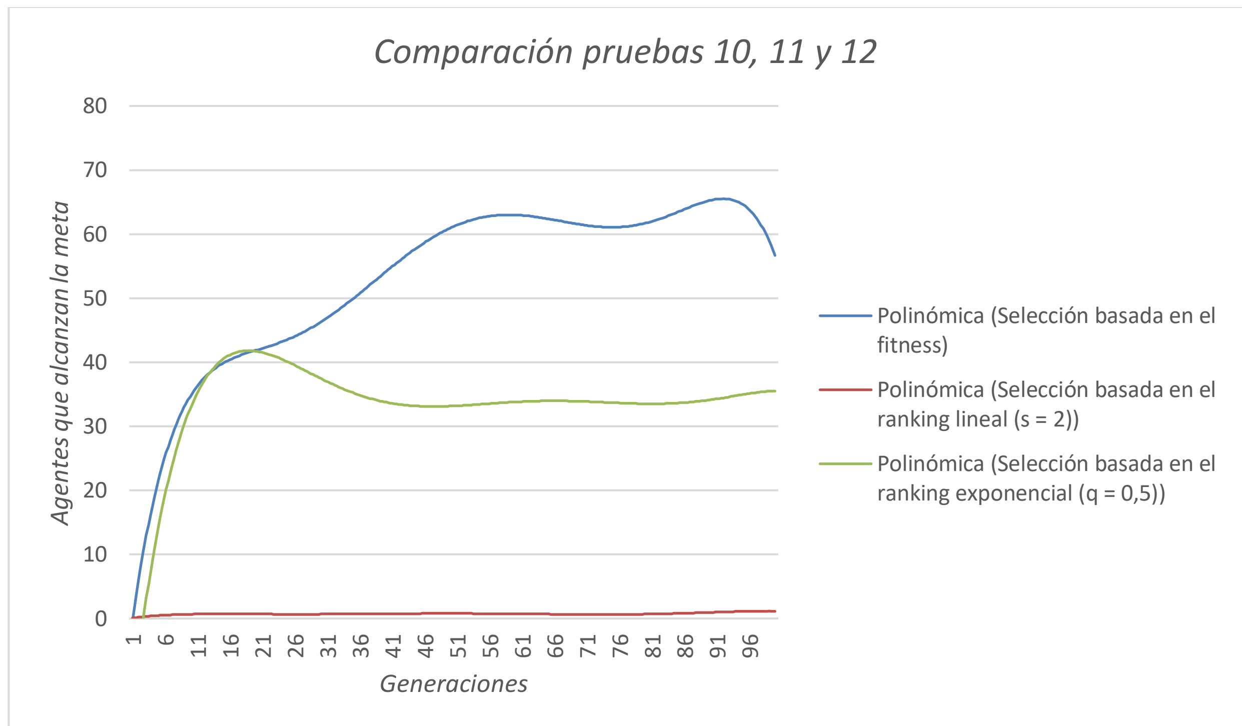

Figura 77: Comparación pruebas 10, 11 y 12

Podemos extraer las siguientes conclusiones de la gráfica anterior:

- En el ejemplo utilizado el método basado en el fitness muestra mejores resultados que el método basado en el ranking exponencial. Esto se debe a la forma no lineal de asignar el fitness (los agentes que alcanzan la meta reciben una puntuación muy superior a los que no). Ante un escenario en el cual el fitness se reparte de una forma más lineal, el método basado en el ranking exponencial probablemente muestre mejores resultados.

- El método basado en el ranking lineal no muestra correctos resultados. Observando el reparto de probabilidades del ranking lineal, se puede concluir que, al ser lineal, no existe una diferencia notable entre las mejores soluciones y el resto, dificultando la supervivencia de estas. Por otro lado, permite una mayor exploración del espacio de soluciones. Por lo tanto, una posible utilidad de este tipo de selección de padres puede ser la exploración de nuevas soluciones una vez ya se ha obtenido convergencia. Por ejemplo:

1. Se realiza el entrenamiento utilizando el método de selección basado en el fitness o el método de selección basado en el ranking exponencial.

2. Una vez obtenidas las redes neuronales entrenadas, se puede realizar otro entrenamiento (partiendo de estas soluciones) utilizando el método basado en el ranking lineal. De esta forma se realiza la exploración del espacio de soluciones en busca de alguna solución mejor.

PRUEBAS Y RESULTADOS II

1. Introducción

En el apartado anterior se detallaron un conjunto de pruebas cuyo objetivo era comparar los diferentes tipos de combinaciones de parámetros ante una aplicación determinada. A continuación, se detalla una prueba con un escenario relativamente más complejo.

En el apartado anterior, el individuo recibía únicamente información relativa a los rayos, permitiendo a la red aprender a esquivar obstáculos. Esto era perfecto para el escenario utilizado, puesto que, al estar guiado por las paredes, siempre que consiguieran esquivar los obstáculos acabarían hallando el objetivo. No obstante, si se utiliza esa red neuronal entrenada en un escenario más abierto, no se observaría convergencia. Esto se debe a que la red neuronal no dispone de ninguna información de entrada relativa a la ubicación del objetivo. En esta prueba se resuelve un ejemplo que utiliza tanto la información de los obstáculos como del objetivo final.

El escenario utilizado es el siguiente:

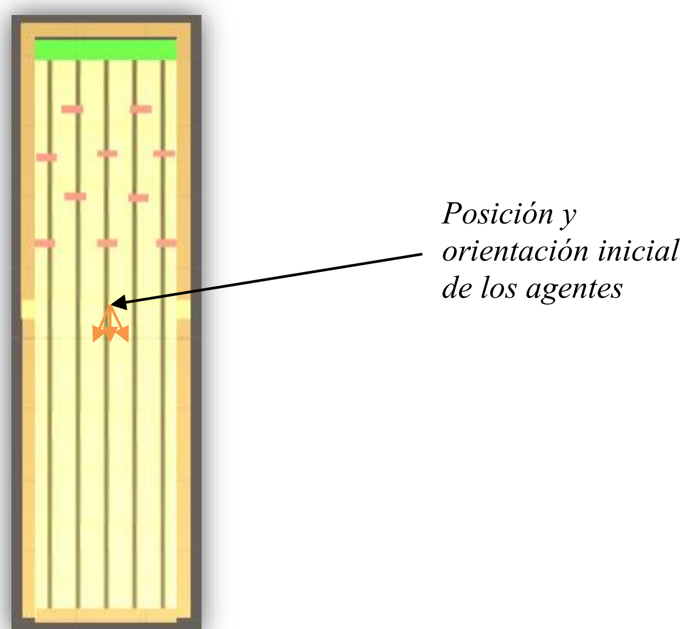

Figura 78: Escenario utilizado en PRUEBAS Y RESULTADOS II

Como se puede observar, los agentes tienen libertad para recorrer en todas las direcciones, y necesitan información de la meta para poder alcanzar la misma. Debido a su posición y orientación iniciales, los agentes deben percatarse que están yendo en la dirección opuesta a la meta y corregir su rumbo. De no ser así, no la alcanzarán a tiempo, debido al tiempo máximo por generación establecido. Existe la posibilidad de que algún individuo la encuentre por accidente, pero, de ser así, en las siguientes generaciones probablemente no la alcance, perdiendo esta información genética que no es la correcta. La red neuronal utilizada es la siguiente:

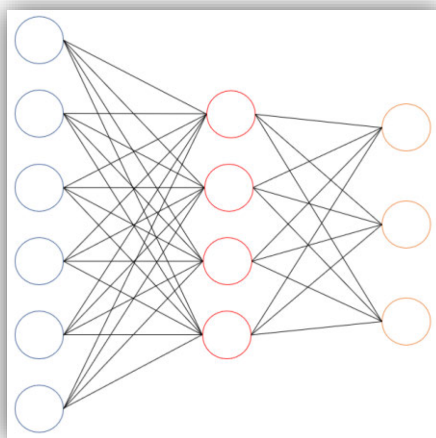

Figura 79: Red neuronal utilizada para PRUEBAS Y RESULTADOS II

Disponemos de una capa de entrada formada por seis nodos, una capa oculta de cuatro nodos y finalmente una capa de salida de tres nodos. La salida de la red neuronal se interpreta de forma idéntica al apartado anterior. Las entradas a la red neuronal son las siguientes:

- Nodos 1-5: información relativa a los rayos, idéntica a la proporcionada en el apartado anterior.
- Nodo 6: información relativa al objetivo.

La información entrante en los nodos 1-5 (información de los rayos) se encuentra en el rango [0,10] (próxima a 0 si está a punto de colisionar con el obstáculo, 10 si no detecta nada). Para facilitar la convergencia, codificamos la información del sexto nodo de forma equivalente al

resto de nodos. El objetivo de esta entrada es indicar a la red neuronal si, a lo largo del tiempo, se está acercando a la meta o no.

Para conseguir esto se utiliza una variable nueva llamada *progress* que será el resultado de las siguientes condiciones. En cada frame de la ejecución:

- Si el agente se encuentra más próximo del objetivo que en el frame anterior:

$$progress += 0.05$$

- Si el agente se encuentra más alejado del objetivo que en el frame anterior:

$$progress -= 0.05$$

El valor de esta variable se sitúa en el rango [0,10]. Por lo tanto, si el agente se está alejando de la meta, el valor de entrada del sexto nodo disminuirá hasta 0 y viceversa.

La red neuronal no es el único elemento que debemos modificar para poder implementar este ejemplo correctamente. También debemos modificar el fitness para que tenga en consideración la distancia agente-objetivo en lugar del tiempo de vida, debido a que merece un mayor valor fitness un individuo que muere cerca de la meta que uno que muere al comienzo del escenario (aunque ambos hayan permanecido vivos el mismo tiempo). El fitness definido para este escenario es el siguiente:

- Si no alcanza la meta:

$$fitness = (factor_{distancia})^2$$

- Si alcanza la meta:

$$fitness = bonus_{meta} + (factor_{distancia})^2$$

Siendo el $factor_{distancia}$:

$$factor_{distancia} = \frac{100}{(40 - posicion_z)}$$

A diferencia de la función fitness utilizada en el apartado *PRUEBAS Y RESULTADOS I*, existe un factor relacionado con la distancia del individuo. El objetivo por alcanzar se encuentra en la

posición (x, y, 40). Si un agente alcanza la meta, por las dimensiones de la meta y del agente, se encontrará en la posición (x, y, 38). La posición más lejana de la meta en la que los individuos pueden morir es el muro inferior del escenario, acabando en la posición (x, y, -38). Por lo tanto:

- Para un individuo que alcance la meta:

$$factor_{distancia} = \frac{100}{(40\ -\ 38)} = 50$$

- Para un individuo que muera en la posición más lejana del objetivo:

$$factor_{distancia} = \frac{100}{(40 - (-38))} \cong 1{,}28$$

Como se puede observar, cuanto mayor sea la posición del eje z (más cerca del objetivo), mejor será el fitness. El $bonus_{meta}$ utilizado es el mismo al utilizado en el apartado anterior.

2. Resultados

Para esta prueba se han realizado tres simulaciones con los mismos parámetros que los utilizados en la prueba 5 del capítulo anterior, excepto por la posición y orientación inicial. En la siguiente gráfica podemos observar la media del número de agentes que alcanzaron la meta en estas simulaciones:

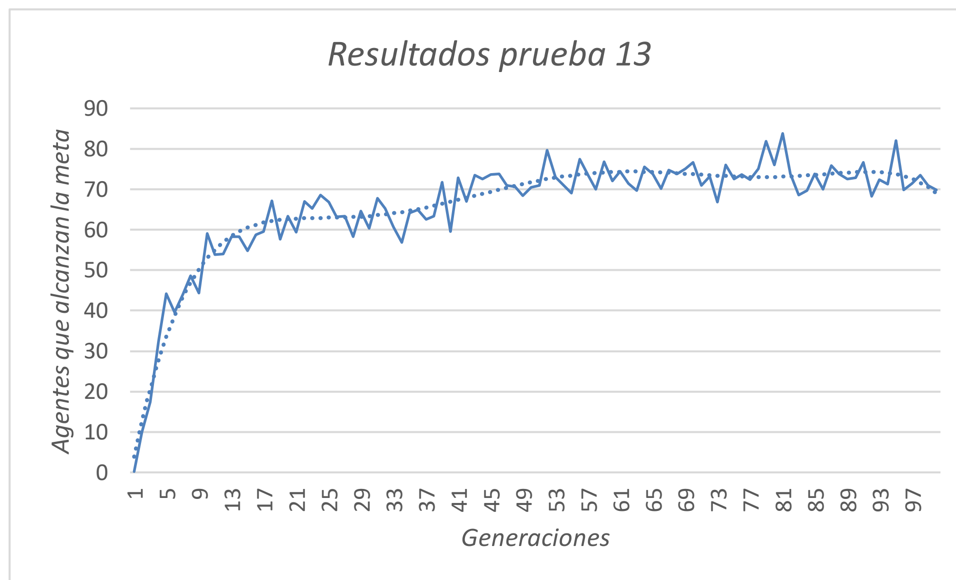

Figura 80: Resultados prueba 13

Se puede observar una clara convergencia desde generaciones tempranas, con un elevado número de agentes que alcanzan la meta, aproximadamente a partir de la generación 20.

PRESUPUESTO

En este capítulo se detalla el presupuesto necesario para la realización del Proyecto de Fin de Grado. El presupuesto de los recursos materiales, incluyendo las licencias del software necesario, es el siguiente:

- Equipo Lenovo ideapad 700:
    - Coste por unidad: 1000 €
    - Cantidad: 1
- Microsoft 365 Personal:
    - Coste por unidad: 69 € (anuales)
    - Coste final: 0 € (licencia obtenida por la Universidad Politécnica de Madrid)
    - Cantidad: 1.
- Unity LTS Personal:
    - Coste por unidad: 0 €
    - Cantidad: 1

Por lo tanto:

Total Recursos Materiales	1000 €

El presupuesto del coste de trabajo es el siguiente:

- Desarrollador ingeniero junior:
    - Coste: 12 € (hora)

- Cantidad: 1

Suponiendo un tiempo de trabajo de 6 meses, con una dedicación de 4 horas diarias, tenemos un total de 480 horas. Teniendo esto en cuenta, el presupuesto del coste de trabajo es:

Total Coste de Trabajo	5760 €

Finalmente, el presupuesto total del proyecto es el siguiente:

Total Recursos Materiales	1000 €
Total Coste de Trabajo	5760 €
Total	**6760 €**

CONCLUSIONES

En el presente proyecto se ha realizado un módulo de Unity prototipo que facilita el entrenamiento de redes neuronales artificiales, y su unión con los correspondientes componentes del proyecto. Respecto a los objetivos iniciales del proyecto, se puede concluir:

- Utilizando escenarios simples, demuestra que existe la posibilidad de entrenamiento de redes neuronales artificiales con algoritmos genéticos, aplicando el resultado en un motor de videojuegos.
- Cubre una amplia gama de parámetros relacionados con algoritmos genéticos y computación evolutiva, permitiendo al usuario de este elegir los que crea que son más apropiados para el entrenamiento de su aplicación.
- Dispone de una interfaz amigable y clara, fácil de manejar para usuarios que conocen los diferentes parámetros de los algoritmos evolutivos. También dispone de un sistema de control de errores, para notificar al usuario en caso de seleccionar parámetros conflictivos.
- Proporciona información a posteriori de las simulaciones ejecutadas, permitiendo extraer información de estas.

Por otro lado, para comprobar la validez del prototipo se han realizado un conjunto de pruebas detalladas en apartados anteriores. Tras el desarrollo de estas podemos extraer las siguientes conclusiones:

- En el capítulo *PRUEBAS Y RESULTADOS I* se han realizado un conjunto de pruebas con el objetivo de comparar los diferentes tipos de parámetros. Observando las diferentes gráficas se puede afirmar que se obtiene convergencia en todas las pruebas realizadas, culminando con un elevado número de soluciones que alcanzan el objetivo. Debido a la amplia variedad de posibilidades de parámetros no se puede extraer una conclusión certera respecto a cuáles son los más apropiados. Basándonos en los

resultados, algunos parámetros muestran mejor respuesta que otros, pero esto puede variar en función de la aplicación.

- La prueba realizada en el capítulo *PRUEBAS Y RESULTADOS II* tiene como objetivo mostrar el resultado de aplicar el módulo para llevar a cabo el entrenamiento de un escenario más complejo. Al igual que en el capítulo *PRUEBAS Y RESULTADOS I*, las soluciones alcanzan el objetivo deseado.

- De las pruebas anteriores podemos concluir que el módulo permite obtener un conjunto de soluciones que cumplen con el objetivo deseado, pero es importante recordar la restricción computacional mencionada en el apartado *RESTRICCIONES DE DISEÑO*. Los algoritmos genéticos suelen aplicarse con un número muy elevado de soluciones. Ante aplicaciones muy complejas, es posible que sea necesario la utilización de un sistema con mayor capacidad computacional que permita la utilización del módulo con dicha cantidad de soluciones por generación.

Por último, es de notable importancia mencionar que se proporciona una interfaz de programación de aplicaciones (*API*) que permite al usuario unir su aplicación específica con este módulo. Existe una cierta limitación respecto a las aplicaciones que se pueden utilizar, por esto, en el siguiente capítulo se expone un listado de posibles mejoras del proyecto que permitirían solucionar dicha limitación.

TRABAJOS FUTUROS

El proyecto expuesto en este documento tiene amplias posibilidades de mejora y expansión. En los capítulos *PRUEBAS Y RESULTADOS I* y *PRUEBAS Y RESULTADOS II*, se detallaron un conjunto de pruebas que mostraban el funcionamiento del módulo. Como trabajo futuro es importante realizar más pruebas que permitan:

- Observar la convergencia de las redes neuronales ante aplicaciones diferentes a la expuesta.
- Estudiar el tipo de parámetros de entrada de la red neuronal que facilitan la correcta convergencia ante diferentes escenarios.
- Búsqueda de posibles errores ocultos.
- Observar la convergencia de las redes con diferentes tamaños ante una misma aplicación.
- Observar la convergencia de las redes ante la misma aplicación y escenario, pero realizadas en diferentes sistemas informáticos (con diferentes capacidades), pudiendo comparar, por ejemplo, una simulación con 100 agentes por generación y otra con 1000.

A continuación, también se proponen posibles mejoras futuras que se pueden incorporar al módulo desarrollado.

1. Diferentes escenarios en una misma simulación

Ante un escenario en el cual las soluciones comienzan siempre en la misma posición y orientación, existe la posibilidad de que no generalicen, adaptándose únicamente a esta situación. Para solucionar esto, en el panel de la posición se permite la introducción de rangos de posiciones. De la misma forma, existe la posibilidad de que los individuos no generalicen completamente, aprendiendo sólo en el escenario en el que se realiza el entrenamiento.

Permitiendo al usuario seleccionar varios conjuntos de coordenadas de inicio, se pueden iniciar las generaciones en diferentes escenarios, permitiendo a las redes neuronales adaptarse a un mayor número de estímulos. También deberá permitir al usuario elegir cuando realizar la transición de unas coordenadas a otras. Así, podemos dar solución al problema mencionado.

Por ejemplo, supongamos que el usuario selecciona tres conjuntos de coordenadas *C1*, *C2* y *C3* (cada conjunto formado por sus respectivas coordenadas x, y, z). Se pueden ofrecer las siguientes posibilidades:

- Una coordenada por generación: se almacenan el conjunto de coordenadas de forma ordenada en una lista. Al comienzo de la generación, el módulo elige de la lista las coordenadas correspondientes. Para cada generación todos los individuos comenzarán en las coordenadas elegidas. Por lo tanto, tendríamos:
    - Generación 1 utiliza las coordenadas *C1*.
    - Generación 2 utiliza las coordenadas *C2*.
    - Generación 3 utiliza las coordenadas *C3*.
    - Generación 4 utiliza las coordenadas *C1*…

  Y así sucesivamente.

- Una coordenada aleatoria por generación: de forma similar al escenario anterior, pero ahora las coordenadas se eligen de forma aleatoria.

- En función del fitness: se establece una relación coordenada-fitness máximo, y se almacenan las coordenadas en una lista ordenada por el fitness máximo, de menor a mayor. Cuando una solución alcance el fitness máximo de una determinada coordenada, se realizará la transición a la siguiente coordenada. Por ejemplo, supongamos las siguientes relaciones coordenadas-fitness máximo:
    - C1 → 100

- C2 → 200
- C3 → ∞

De esta forma, se comenzará utilizando las coordenadas *C1*. Si un individuo obtiene un fitness superior a 100, se utilizarán las coordenadas *C2*. Así sucesivamente hasta llegar a *C3*. El valor del fitness máximo de *C3*, al ser las últimas coordenadas disponibles, debe ser inalcanzable por cualquier solución para no realizar la transición a unas coordenadas que no existen (*C4*).

- Varias coordenadas por generación: en lugar de utilizar únicamente un conjunto de coordenadas (*C1*, *C2* o *C3*) en cada generación, utilizar todos los conjuntos de coordenadas, inicializando un número equivalente (o casi equivalente, si la división no es entera) de soluciones en cada uno. Por ejemplo, si disponemos de 100 individuos por generación:
  - 33 individuos comienzan en las coordenadas *C1*.
  - 33 individuos comienzan en las coordenadas *C2*.
  - 34 individuos comienzan en las coordenadas *C3*.

2. Elementos interactuables con los individuos

En el apartado de restricciones de diseño se mencionó que la aplicación que se fusione con el módulo no puede incorporar elementos que interactúen de forma individual con las soluciones. Se pueden incorporar estos elementos con diferentes opciones y posibilidades como trabajo futuro. Por ejemplo, supongamos una aplicación en la que se desea entrenar una red neuronal que controla un dron siendo el escenario un parque, teniendo que esquivar diferentes obstáculos reales. Un ejemplo de obstáculo sería un pájaro. Si este obstáculo está programado para desaparecer al colisionar con un dron, el resto de los drones que se estén entrenando simultáneamente no podrán aprender a esquivarlo. Ante esta situación, se pueden plantear diferentes posibilidades:

- Replicar el obstáculo: en el momento en el que desaparece del escenario por una situación como la descrita, se puede replicar un obstáculo idéntico (en la misma posición), de tal forma, que el resto de los individuos de la generación seguirán interactuando con el mismo.

- Crear un nuevo obstáculo: en lugar de replicar el mismo obstáculo, se puede crear otro con las mismas propiedades, pero otra posición diferente.

- Inicializar copias de cada obstáculo: si un determinado obstáculo cumple las condiciones mencionadas en el ejemplo, se pueden inicializar tantos como individuos tenga la generación de tal forma que sean idénticos (incluyendo la misma posición y movimiento). De tal forma que, si uno desaparece por la interacción con alguna solución, seguirán existiendo todas las demás copias. Para que esto sea posible, los obstáculos idénticos no pueden colisionar ni interactuar entre sí.

- Replicar el escenario completo: creando el escenario (y todos los elementos que lo componen) como un prefab, existe la posibilidad de replicar el escenario tantas veces como soluciones por generación, de tal forma que en cada réplica exista únicamente un solo individuo. Al haber un sólo individuo por escenario, no existe ningún problema con este tipo de elementos interactuables.

3. Incorporación de redes neuronales NEAT

Como se mencionó en el apartado de estructura del proyecto, el módulo sólo implementa el entrenamiento genético de las redes neuronales artificiales con topología fija. Se proporciona un código que permite utilizar las redes neuronales con topología variante, aunque no esté implementado su entrenamiento. Este código permite:

- Instanciar redes neuronales. Su topología inicial consta sólo de nodos de entrada y nodos de salida.
- Obtener la salida ante una lista de valores de entrada.
- Crear una nueva conexión (aleatoria) en caso de ser posible.
- Crear un nuevo nodo oculto (aleatorio).

La implementación se puede realizar permitiendo al usuario elegir entre ambos tipos de redes neuronales en los parámetros del módulo. Una posible implementación sería la siguiente:

1. Se incluye en el módulo la opción de redes neuronales NEAT y simple.
2. Incluir en el panel de mutación, la probabilidad de añadir un nuevo nodo y la probabilidad de añadir una nueva conexión. Estas opciones sólo deben ser válidas si se elige este tipo de redes.
3. Debido a la gran diferencia entre la gestión de las redes neuronales simples y las redes neuronales NEAT, una posibilidad sería disponer de dos scripts de *GeneticManager*: *GeneticManagerSimple* y *GeneticManagerNEAT*. En función del tipo de red seleccionado, el *CanvasManager* inicializará uno u otro.
4. El script *GeneticBehaviour* también deberá editarse, incluyendo una instanciación de una red neuronal NEAT (además de la instanciación de la red neuronal simple que ya dispone).

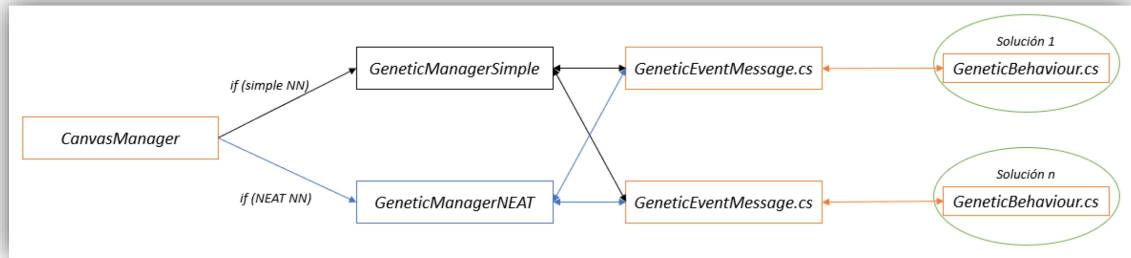

Figura 81: Comunicación entre los elementos del módulo tras incorporar GeneticManagerNEAT

REFERENCIAS

[1] W. Ertel, “What Is Artificial Intelligence?”, en *Introduction to Artificial Intelligence*. Weingarten, Germany: Springer, 2018, pp 1-3.

[2] W. Ertel, *Introduction to Artificial Intelligence*. Weingarten, Germany: Springer, 2018.

[3] W. Ertel, “Neural Networks”, en *Introduction to Artificial Intelligence*. Weingarten, Germany: Springer, 2018, pp 245-246.

[4] S. Haykin, *Neural Networks and Learning Machines*. Ontario, Canada: Pearson International Edition, 2009.

[5] E. Alpaydin, *Introduction to Machine Learning*. Cambridge, Massachusetts; London, England: The MIT Press, 2014.

[6] W. Ertel, “The Perceptron, a Linear Classifier”, *Introduction to Artificial Intelligence*. Weingarten, Germany: Springer, 2018, pp 183-185.

[7] C. C. Aggarwal, *Neural Networks and Deep Learning*. NY, USA: Springer, 2018.

[8] M. Kubat, “Backpropagation of Error”, *An Introduction to Machine Learning*. FL, USA: Springer, 2017, pp 97-99.

[9] M. Kubat, *An Introduction to Machine Learning*. FL, USA: Springer, 2017.

[10] A. E. Eiben, J. E. Smith, *Introduction to Evolutionary Computing*. Amsterdam, The Netherlands; Bristol, UK: Springer, 2015.

[11] J. Pitman, “Appendix 5: Normal Table”, *Probability*. NY, USA; Ontario, Canada: Springer, 1993, pp 531.

[12] J. Pitman, *Probability*. NY, USA; Ontario, Canada: Springer, 1993.

[13] Z. Michalewicz, *Genetic Algorithms + Data Structures = Evolution Programs*, Charlotte, USA: Springer, 1998.

[14] R. L. Haupt, S. E. Haupt, *Practical Genetic Algorithms*. USA: Wiley-Interscience, 2004.

[15] K. O. Stanley, R. Miikkulainen, “Initial Populations and Topological Innovation”, *Evolving Neural Networks through Augmenting Topologies*. Austin, USA: The MIT Press Journals, pp 105-106.

[16] K. O. Stanley, R. Miikkulainen, *Evolving Neural Networks through Augmenting Topologies*. Austin, USA: The MIT Press Journals.

[17] K. O. Stanley, R. Miikkulainen, “Tracking Genes through Historical Markings”, *Evolving Neural Networks through Augmenting Topologies*. Austin, USA: The MIT Press Journals, pp 108-109.

[18] Unity. [En línea]. Disponible en: https://unity.com/es. [Accedido: 29-jun-2020]

[19] Lightbox Academy: Diferencias entre Unity y Unreal. [En línea]. Disponible en: https://lboxacademy.es/blog/diferencias-entre-unity-y-unreal/. [Accedido: 29-jun-2020]

[20] Unity Documentation. [En línea]. Disponible en: https://docs.unity3d.com/Manual/index.html. [Accedido: 24-jun-2020]

[21] Documentación de C#. [En línea]. Disponible en: https://docs.microsoft.com/es-es/dotnet/csharp/. [Accedido: 24-jun-2020]

ANEXO I: EJEMPLO DE FICHERO COMPLETO INPUT.XML

A continuación, se muestra el contenido completo de un fichero *input.xml*:

```
<?xml version="1.0"?>
<Inputs>
    <BasicNumberOfParents1>False</BasicNumberOfParents1>
    <BasicNumberOfParents2>True</BasicNumberOfParents2>
    <BasicPopulationPerGeneration>100</BasicPopulationPerGeneration>
    <BasicNumberOfGenerations>100</BasicNumberOfGenerations>
    <TransformXPosition>39,37569</TransformXPosition>
    <TransformYPosition>1</TransformYPosition>
    <TransformZPosition>-55,68975</TransformZPosition>
    <TransformXRotation>0</TransformXRotation>
    <TransformYRotation>0</TransformYRotation>
    <TransformZRotation>0</TransformZRotation>
<ParentSelectionRouletteWheelSelection>True</ParentSelectionRouletteWheelSel
ection>

<ParentSelectionLinearRankingSelection>False</ParentSelectionLinearRankingSel
ection>
    <ParentSelectionLinearRankingSelectionS>0</ParentSelectionLinearRankingSel
ectionS>

<ParentSelectionExponentialRankingSelection>False</ParentSelectionExponential
RankingSelection>
    <ParentSelectionExponentialRankingSelectionQ>0</ParentSelectionExponential
RankingSelectionQ>
    <RecombinationBinaryOnePointCrossover>False</RecombinationBinaryOnePoint
Crossover>
    <RecombinationBinaryNPointCrossover>False</RecombinationBinaryNPoint
Crossover>
    <RecombinationBinaryNPointCrossoverN>0</RecombinationBinaryNPoint
CrossoverN>
    <RecombinationBinaryUniformCrossover>True</RecombinationBinaryUniform
Crossover>
    <RecombinationFloatDiscreteRecombination>False</RecombinationFloatDiscrete
Recombination>
    <RecombinationFloatSimpleArithmeticRecombination>False</RecombinationFloat
SimpleArithmeticRecombination>
    <RecombinationFloatSingleArithmeticRecombination>False</RecombinationFloat
SingleArithmeticRecombination>
    <RecombinationFloatWholeArithmeticRecombination>False</RecombinationFloat
WholeArithmeticRecombination>
    <MutationProbability>0,1</MutationProbability>
    <MutationBinaryMutation>True</MutationBinaryMutation>
    <MutationFloatUniformMutation>False</MutationFloatUniformMutation>
    <MutationFloatNonuniformMutation>False</MutationFloatNonuniformMutation>
    <MutationTriggersLoaded>True</MutationTriggersLoaded>
    <MutationTriggersPath>D:/Disco duro/Disco duro/universidad y
estudios/PFG/mutationTriggers.xml</MutationTriggersPath>
    <TimeMaxTimePerGeneration>70</TimeMaxTimePerGeneration>
```

```
        <TimeFitnessIfMax>0</TimeFitnessIfMax>
        <PopulationManagementGenerational>False</PopulationManagementGenerational>
        <PopulationManagementSteadyState>True</PopulationManagementSteadyState>
        <PopulationManagementSteadyStateSurvivors>5</PopulationManagementSteadyState
Survivors>
</Inputs>
```

Figura 82: Fichero input.xml completo

Es importante recordar que este fichero se genera automáticamente con los parámetros introducidos para la simulación. El fichero *input.xml* contiene todos los parámetros, incluyendo los que no se utilizan. En el de la figura anterior, se han configurado los siguientes parámetros:

- Número de padres: 2.
- Individuos por generación: 100.
- Número de generaciones: 100.
- Posición x: 39,37569.
- Posición y: 1.
- Posición z: -55,68975.
- Rotación x: 0.
- Rotación y: 0.
- Rotación z: 0.
- Selección de padres: método basado en el fitness (*Roulette wheel selection*).
- Recombinación: binaria uniforme.
- Probabilidad de mutación: 0.1.
- Mutación: binaria.
- Triggers de mutación: sí, en:

  D:/Disco duro/Disco duro/universidad y estudios/PFG/mutationTriggers.xml

- Tiempo máximo por generación: 70 segundos.

- Fitness si se alcanza el tiempo máximo: 0.

- Gestión de la población: modelo estado estacionario con 5 supervivientes.

Si se manipula este fichero, es posible que algunos parámetros entren en conflicto. Si se produce esto, se cancelará la operación y se informará al usuario con el siguiente mensaje:

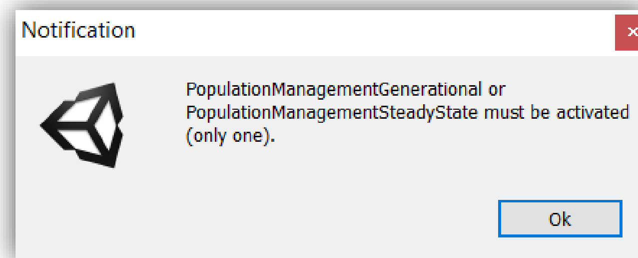

Figura 83: Notificación si el fichero input.xml es erróneo

ANEXO II: EJEMPLO DE FICHERO COMPLETO COMMANDS.XML

A continuación, se muestra el contenido completo de un fichero de comandos de simulaciones de ejemplo:

```
<?xml version="1.0"?>
<Commands>
	<Command>
		<SimulationName>Sim1</SimulationName>
		<BasicNumberOfParents1>False</BasicNumberOfParents1>
		<BasicNumberOfParents2>True</BasicNumberOfParents2>
		<BasicPopulationPerGeneration>100</BasicPopulationPerGeneration>
		<BasicNumberOfGenerations>200</BasicNumberOfGenerations>
		<TransformXPosition>39,37569</TransformXPosition>
		<TransformYPosition>1</TransformYPosition>
		<TransformZPosition>-55,68975</TransformZPosition>
		<TransformXRotation>0</TransformXRotation>
		<TransformYRotation>0</TransformYRotation>
		<TransformZRotation>0</TransformZRotation>
	<ParentSelectionRouletteWheelSelection>False</ParentSelectionRoulette
WheelSelection>
		<ParentSelectionLinearRankingSelection>False</ParentSelectionLinear
RankingSelection>
		<ParentSelectionLinearRankingSelectionS>0</ParentSelectionLinear
RankingSelectionS>
		<ParentSelectionExponentialRankingSelection>True</ParentSelection
ExponentialRankingSelection>
		<ParentSelectionExponentialRankingSelectionQ>0.5</ParentSelection
ExponentialRankingSelectionQ>
		<RecombinationBinaryOnePointCrossover>False</RecombinationBinaryOne
PointCrossover>

	<RecombinationBinaryNPointCrossover>False</RecombinationBinaryNPoint
Crossover>
		<RecombinationBinaryNPointCrossoverN>0</RecombinationBinaryNPoint
CrossoverN>

	<RecombinationBinaryUniformCrossover>False</RecombinationBinaryUniform
Crossover>
		<RecombinationFloatDiscreteRecombination>False</RecombinationFloat
DiscreteRecombination>

	<RecombinationFloatSimpleArithmeticRecombination>True</Recombination
FloatSimpleArithmeticRecombination>

	<RecombinationFloatSingleArithmeticRecombination>False</Recombination
FloatSingleArithmeticRecombination>

	<RecombinationFloatWholeArithmeticRecombination>False</Recombination
FloatWholeArithmeticRecombination>
		<MutationProbability>0,1</MutationProbability>
		<MutationBinaryMutation>False</MutationBinaryMutation>
```

```
            <MutationFloatUniformMutation>True</MutationFloatUniformMutation>
            <MutationFloatNonuniformMutation>False</MutationFloatNonuniform
    Mutation>
            <MutationTriggersLoaded>False</MutationTriggersLoaded>
            <MutationTriggersPath>null</MutationTriggersPath>
            <TimeMaxTimePerGeneration>70</TimeMaxTimePerGeneration>
            <TimeFitnessIfMax>0</TimeFitnessIfMax>
            <PopulationManagementGenerational>False</PopulationManagement
    Generational>
            <PopulationManagementSteadyState>True</PopulationManagement
    SteadyState>
            <PopulationManagementSteadyStateSurvivors>5</PopulationManagement
    SteadyStateSurvivors>
        </Command>
        <Command>
            <SimulationName>Sim2</SimulationName>
            <BasicNumberOfParents1>False</BasicNumberOfParents1>
            <BasicNumberOfParents2>True</BasicNumberOfParents2>
            <BasicPopulationPerGeneration>100</BasicPopulationPerGeneration>
            <BasicNumberOfGenerations>200</BasicNumberOfGenerations>
            <TransformXPosition>39,37569</TransformXPosition>
            <TransformYPosition>1</TransformYPosition>
            <TransformZPosition>-55,68975</TransformZPosition>
            <TransformXRotation>0</TransformXRotation>
            <TransformYRotation>0</TransformYRotation>
            <TransformZRotation>0</TransformZRotation>
        <ParentSelectionRouletteWheelSelection>False</ParentSelectionRoulette
    WheelSelection>
            <ParentSelectionLinearRankingSelection>False</ParentSelectionLinear
    RankingSelection>
            <ParentSelectionLinearRankingSelectionS>0</ParentSelectionLinear
    RankingSelectionS>
            <ParentSelectionExponentialRankingSelection>True</ParentSelection
    ExponentialRankingSelection>
            <ParentSelectionExponentialRankingSelectionQ>0.5</ParentSelection
    ExponentialRankingSelectionQ>
            <RecombinationBinaryOnePointCrossover>False</RecombinationBinaryOne
    PointCrossover>

        <RecombinationBinaryNPointCrossover>False</RecombinationBinaryNPoint
    Crossover>
            <RecombinationBinaryNPointCrossoverN>0</RecombinationBinaryNPoint
    CrossoverN>

        <RecombinationBinaryUniformCrossover>False</RecombinationBinaryUniform
    Crossover>
            <RecombinationFloatDiscreteRecombination>False</RecombinationFloat
    DiscreteRecombination>

        <RecombinationFloatSimpleArithmeticRecombination>False</Recombination
    FloatSimpleArithmeticRecombination>

        <RecombinationFloatSingleArithmeticRecombination>True</Recombination
    FloatSingleArithmeticRecombination>

        <RecombinationFloatWholeArithmeticRecombination>False</Recombination
    FloatWholeArithmeticRecombination>
            <MutationProbability>0,1</MutationProbability>
```

```
                <MutationBinaryMutation>False</MutationBinaryMutation>
                <MutationFloatUniformMutation>True</MutationFloatUniformMutation>
                <MutationFloatNonuniformMutation>False</MutationFloatNonuniform
Mutation>
                <MutationTriggersLoaded>False</MutationTriggersLoaded>
                <MutationTriggersPath>null</MutationTriggersPath>
                <TimeMaxTimePerGeneration>70</TimeMaxTimePerGeneration>
                <TimeFitnessIfMax>0</TimeFitnessIfMax>
                <PopulationManagementGenerational>False</PopulationManagement
Generational>
                <PopulationManagementSteadyState>True</PopulationManagement
SteadyState>
                <PopulationManagementSteadyStateSurvivors>5</PopulationManagement
SteadyStateSurvivors>
        </Command>
</Commands>
```


Figura 84: Fichero xml de ejemplo de comandos de simulación

Podemos observar que el fichero contiene dos bloques *Command*, por lo tanto, el fichero define dos simulaciones, que se ejecutarán de forma secuencial. En el primer bloque *Command* se definen los siguientes parámetros:

- Número de padres: 2.
- Individuos por generación: 100.
- Número de generaciones: 200.
- Selección de padres: método basado en la clasificación exponencial, con $q = 0,5$.
- Recombinación: aritmética simple.
- Probabilidad de mutación: 0,1.
- Mutación: uniforme.
- Triggers de mutación: ninguno.
- Tiempo máximo por generación: 70 segundos.
- Fitness si se alcanza el tiempo máximo: 0.
- Gestión de la población: modelo estado estacionario con 5 supervivientes.

Una vez finalice la primera simulación, con los parámetros descritos anteriormente, comenzará la simulación correspondiente al segundo bloque *Command*. El segundo bloque tiene los mismos parámetros que el primer bloque excepto por el parámetro de recombinación, cuyo valor es recombinación aritmética individual.

Existe un parámetro opcional que permite indicar el nombre de la carpeta de salida de los ficheros de la simulación, llamado *SimulationName*. En el ejemplo anterior, la primera simulación se guardará en una carpeta llamada *Sim1*, mientras que la segunda se guardará en *Sim2*.

ANEXO III: EJEMPLO DE FICHERO COMPLETO MUTATIONTRIGGERS.XML

A continuación, se muestra un fichero xml de ejemplo con triggers de mutación:

```
<?xml version="1.0"?>
<MutationTriggers>
    <Trigger mutationValue="0.01">
        <Condition variable="GoalAchieved" operator=">=" value="1"/>
    </Trigger>
    <Trigger mutationValue="0.009">
        <Condition variable="GoalAchieved" operator=">=" value="2"/>
    </Trigger>
    <Trigger mutationValue="0.008">
        <Condition variable="GoalAchieved" operator=">=" value="3"/>
    </Trigger>
    <Trigger mutationValue="0.007">
        <Condition variable="GoalAchieved" operator=">=" value="4"/>
    </Trigger>
    <Trigger mutationValue="0.006">
        <Condition variable="GoalAchieved" operator=">=" value="5"/>
    </Trigger>
    <Trigger mutationValue="0.005">
        <Condition variable="GoalAchieved" operator=">=" value="6"/>
    </Trigger>
    <Trigger mutationValue="0.004">
        <Condition variable="GoalAchieved" operator=">=" value="7"/>
    </Trigger>
    <Trigger mutationValue="0.003">
        <Condition variable="GoalAchieved" operator=">=" value="8"/>
    </Trigger>
    <Trigger mutationValue="0.002">
        <Condition variable="GoalAchieved" operator=">=" value="9"/>
    </Trigger>
    <Trigger mutationValue="0.001">
        <Condition variable="GoalAchieved" operator=">=" value="10"/>
    </Trigger>
</MutationTriggers>
```

Figura 85: Fichero xml de ejemplo de triggers de mutación

Se definen 10 triggers que van disminuyendo progresivamente la probabilidad de mutación cuantos más agentes vayan alcanzando la meta. Este fichero se interpreta de la siguiente forma:

```
INICIO

SI GoalAchieved >= 1 Entonces pm = 0.01
SI GoalAchieved >= 2 Entonces pm = 0.009
SI GoalAchieved >= 3 Entonces pm = 0.008
SI GoalAchieved >= 4 Entonces pm = 0.007
SI GoalAchieved >= 5 Entonces pm = 0.006
```

```
SI GoalAchieved >= 6 Entonces pm = 0.005
SI GoalAchieved >= 7 Entonces pm = 0.004
SI GoalAchieved >= 8 Entonces pm = 0.003
SI GoalAchieved >= 9 Entonces pm = 0.002
SI GoalAchieved >= 10 Entonces pm = 0.001

FIN
```

Figura 86: Pseudocódigo del ejemplo de triggers de mutación

Es importante recordar que los diferentes triggers se evalúan mediante un *SI (if)* no un *SI NO (if else)*. Por lo tanto, la última condición que se cumple es la que establece el valor final de la probabilidad de mutación.

ANEXO IV: EJEMPLO FICHERO COMPLETO SIMULATIONRESUME.CSV

Como se indicó en el apartado de *Implementación*, al final de cada simulación se genera un fichero llamado *simulationResume.csv* que muestra información relacionada con cada generación simulada. A continuación, se muestra un fichero *simulationResume.csv* completo de una simulación de 50 generaciones:

Numero gene	Avg fitness	Max fitness	Goal	Time
1	3,78055923	25,714798	0	25,71686
2	3,9871767	25,7586594	0	90,01389
3	3,71736792	28,5427246	0	28,54713
4	4,02173206	55,1111603	0	55,12035
5	4,14794495	55,612915	0	55,62082
6	5,36671973	55,8303223	0	55,84604
7	4,63587659	55,3867493	0	55,39493
8	9,75398218	1441,30542	1	58,69913
9	15,8980313	1441,10168	2	63,41919
10	10,3202478	1441,15808	1	58,84277
11	21,7026182	1441,79816	3	64,10785
12	15,6834314	1441,22394	2	64,32593
13	27,2115056	1441,74744	4	64,09503
14	27,346032	1441,56213	4	64,3335
15	26,8969775	1441,6601	4	64,06848
16	27,1428118	1441,86609	4	64,31805
17	26,9844414	1441,70154	4	64,08765
18	26,90875	1441,69397	4	64,06763
19	44,6765725	1441,89978	7	90,01062
20	49,3178213	1441,59351	8	59,94031
21	32,1625432	1441,75171	5	90,00488
22	37,0870234	1441,83765	6	90,01538
23	43,6928486	1441,63745	7	59,60486
24	49,5566592	1441,60266	8	60,35791
25	38,3558672	1441,74524	6	59,20642

26	50,5328809	1441,4978	8	60,9165
27	44,777439	1441,6925	7	59,51416
28	33,268144	1441,91235	5	61,13708
29	58,072624	1441,56738	9	65,43982
30	76,3228584	1442,31201	12	58,90247
31	81,8910298	1442,23303	13	59,79053
32	59,5360874	1442,46729	9	59,448
33	106,674076	1442,55322	17	59,66577
34	94,4887012	1442,56323	15	59,63281
35	100,550937	1442,55933	16	58,9873
36	84,4454795	1442,46484	13	58,81641
37	118,696844	1442,45898	19	59,44678
38	129,988701	1442,46411	21	90,00928
39	124,453596	1442,49512	20	61,93335
40	114,228446	1442,30615	18	90,00537
41	92,1914297	1442,59839	14	60,03027
42	137,255422	1442,41113	22	60,43726
43	212,691373	1442,31299	35	61,83496
44	178,977938	1442,22852	29	61,25879
45	315,867482	1442,52881	53	61,38159
46	278,037051	1442,53467	46	61,70703
47	286,152886	1442,57959	47	61,32935
48	331,619927	1442,41309	55	60,33179
49	322,041583	1442,52368	53	63,46094
50	338,613477	1442,48926	56	62,59204

Figura 87: Fichero simulationResume.csv

Como se puede observar, existen 5 columnas: número de generaciones, fitness medio de la generación, fitness máximo de la generación, número de individuos que alcanzaron el objetivo y tiempo de la generación. Gracias a esta información podemos presentar la información de la simulación de una forma más visual, como en los capítulos *PRUEBAS Y RESULTADOS I* y *PRUEBAS Y RESULTADOS II*.

ANEXO V: EJEMPLO DE FICHERO COMPLETO DE RED NEURONAL

Cada red neuronal de cada generación se almacena por si el usuario necesita utilizarlas en otro momento. A continuación, se muestra el contenido completo de un fichero *xml* que contiene la información completa de una red neuronal:


```xml
<?xml version="1.0"?>
<SimpleNeuralNetwork>
    <Fitness value="1442,6044921875" />
    <InputNodes number="5" />
    <HiddenNodes number="4" />
    <OutputNodes number="3" />
    <HiddenLayers number="1" />
    <wbList>
        <wb value="0,75606376899223" />
        <wb value="0,638024308550183" />
        <wb value="-0,88204079440403" />
        <wb value="0,220881621941286" />
        <wb value="0,0656540220003815" />
        <wb value="0,17058823940614" />
        <wb value="-0,287063906069437" />
        <wb value="-0,195769162778467" />
        <wb value="-0,23511898485834" />
        <wb value="0,202619962235093" />
        <wb value="0,220032477527795" />
        <wb value="0,290856878693552" />
        <wb value="0,452324821875818" />
        <wb value="0,123980279222955" />
        <wb value="-0,123137753735106" />
        <wb value="-0,184616905962014" />
        <wb value="0,114305601722567" />
        <wb value="0,10797837231926" />
        <wb value="0,123337821063444" />
        <wb value="0,180591124440317" />
        <wb value="-0,185754841490441" />
        <wb value="-0,147528383512827" />
        <wb value="0,0677373109541666" />
        <wb value="-0,0276115712411422" />
        <wb value="-0,0603583291032653" />
        <wb value="-0,289789298487056" />
        <wb value="0,172322770154415" />
        <wb value="-0,0267858713794249" />
        <wb value="-0,0147336535304182" />
        <wb value="-0,0709257791055762" />
        <wb value="-0,0489245897493349" />
        <wb value="-0,0228610924240528" />
        <wb value="-0,0332590521758833" />
        <wb value="-0,016138906697689" />
        <wb value="-0,0627476990988294" />
        <wb value="-0,0630583602561716" />
        <wb value="-0,0864736604886319" />
```

```
            <wb value="-0,0326358532549 48" />
            <wb value="0,0558203954042725" />
        </wbList>
    </SimpleNeuralNetwork>
```

Figura 88: Fichero xml completo de red neuronal

Se trata de una red neuronal de cinco nodos de entrada, una capa oculta con cuatro nodos y una capa de salida de tres nodos. La red neuronal obtuvo un fitness de 1442,6044921875 en la generación de la simulación correspondiente. Cada peso o bias de la red neuronal se almacena en una etiqueta *wb*. Los valores están añadidos de forma ordenada de tal forma que:

- El primer valor corresponde con el primer peso del primer nodo de la primera capa oculta.
- El segundo valor corresponde con el segundo peso del primer nodo de la primera capa oculta. Así sucesivamente con todos los pesos. Tras los pesos de un nodo, el siguiente valor es el bias del mismo.
- Por lo tanto, el sexto valor corresponde con el bias del primer nodo de la primera capa oculta.
- El último valor corresponde con el bias del último nodo de la capa de salida.

APÉNDICE A: MANUAL DE USUARIO

1. Introducción

El objetivo de este manual consiste en facilitar a cualquier usuario la incorporación del módulo a su aplicación. Para explicar la unión módulo-aplicación se utiliza una aplicación de ejemplo, detallando todos los pasos necesarios. De forma resumida tenemos:

1. Descargar e importar el módulo.
2. Editar el script *GeneticBehaviour.cs* para que se encargue del comportamiento del componente.
3. Indicar cuando una solución alcanza el objetivo o cuando muere, y el fitness en ambos casos.
4. Indicar al objeto *GeneticManager* el prefab del componente.
5. Ajustar la cámara si es necesario.

2. Procedimiento

A continuación, se muestran los pasos a seguir para poder utilizar el prototipo del módulo de Unity en una aplicación.

1. Aplicación de ejemplo

Como aplicación de ejemplo se utilizará la misma que se utilizó en el apartado de *PRUEBAS Y RESULTADOS I*. Esta aplicación consiste en un conjunto de individuos que avanzan y pueden girar tanto en sentido horario como antihorario. Su movimiento está controlado por una máquina de estados y se desea sustituirlo por una red neuronal. El objetivo de la aplicación consiste en recorrer un mapa formado por obstáculos hasta alcanzar la meta:

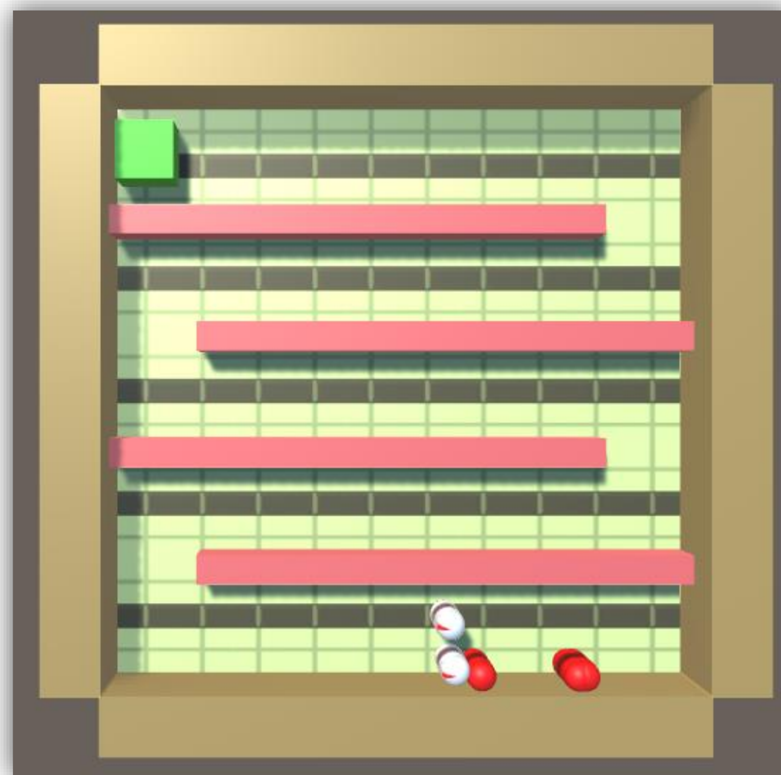

Figura 89: Aplicación de ejemplo

Los prefabs que forman el proyecto son los siguientes:

- Agente: soluciones del algoritmo:

Figura 90: Prefab de las soluciones del algoritmo

- Meta: objetivo que deben alcanzar las soluciones.

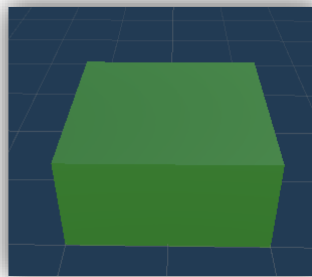

Figura 91: Prefab del objetivo

- Obstáculo: elementos que deben evitar los individuos.

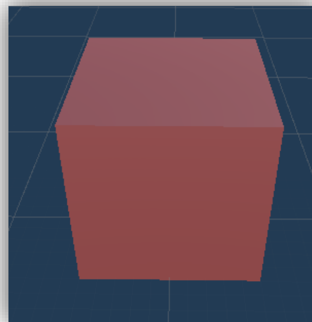

Figura 92: Prefab de los obstáculos

- Pared: elemento que delimita el mapa.

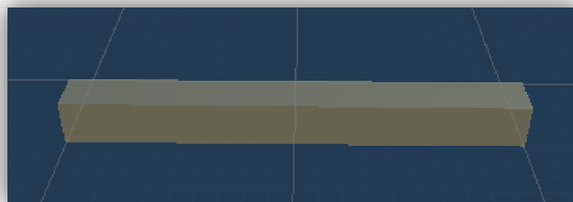

Figura 93: Prefab del delimitador del mapa

Los scripts que componen el proyecto son:

- *Base.cs*: script encargado del movimiento del agente en función del script *Decision.cs*.
- *Decision.cs*: script que contiene la máquina de estados que decide las acciones de la solución.
- *Detector.cs*: script encargado de detectar las colisiones y realizar las acciones correspondientes (considerar éxito en el caso de la meta o fracaso en el caso de un obstáculo o pared).

Los tres scripts son componentes del prefab Agente.

2. Descarga e importación del modulo

Una vez descargado el fichero zip (*GeneticModule.zip*), podemos observar que contiene tanto el módulo (*PFG_GeneticModule.unitypackage*), como una carpeta llamada *GeneticModuleFiles*. En esta carpeta encontramos unos ficheros que se deben copiar en la carpeta del proyecto de Unity:

- *normal_distribution.csv*: fichero que contiene los valores de la distribución normal estándar. Se utiliza para la mutación no uniforme (representación flotante).
- Plantillas/ejemplos: los dos ficheros restantes son ejemplos o plantillas que se pueden utilizar para crear los ficheros de comandos o de triggers de mutación propios (*commands_template.xml* y *mutation_triggers_template.xml*).

A continuación, procedemos a realizar la importación. Para ello, hacemos clic derecho en la carpeta *scenes* y seleccionamos *Import Package* > *Custom Package…*:

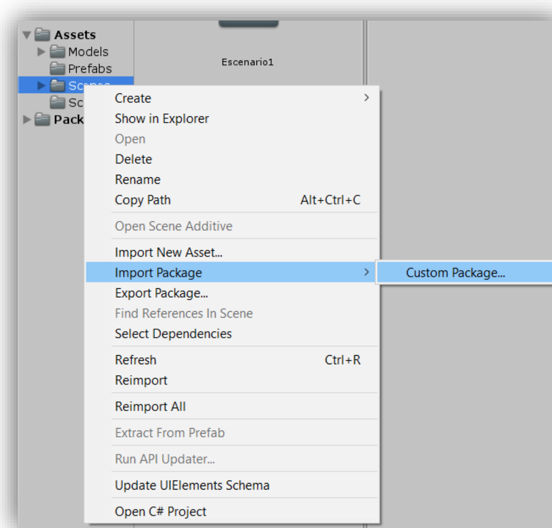

Figura 94: Importar módulo

Aparecerá un cuadro de diálogo para seleccionar el módulo. Lo seleccionamos para finalizar la importación. Podremos observar que aparece la escena *GeneticModule*:

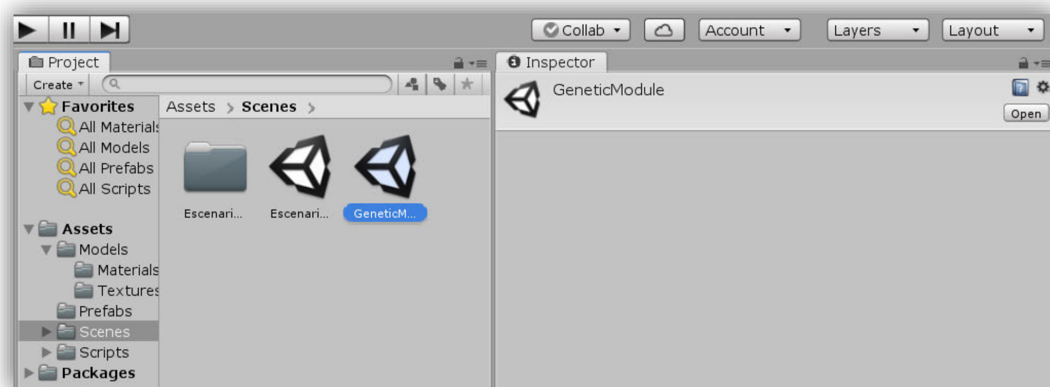

Figura 95: Escena GeneticModule

También se han cargado los scripts del módulo en la carpeta scripts:

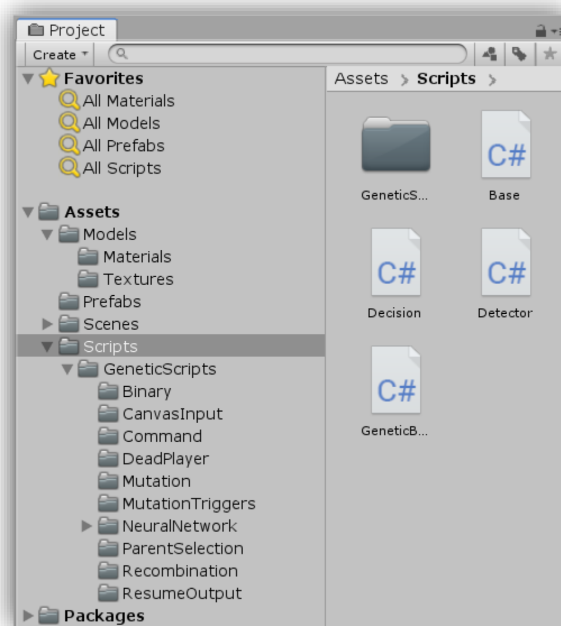

Figura 96: Scripts importados

Por último, añadimos esta escena en la jerarquía y desactivamos la *Directional Light* si es necesario:

Figura 97: GeneticModule en la jerarquía

3. Adaptación de GeneticBehaviour.cs

Una vez importado el módulo, debemos adaptar la aplicación para que pueda ser utilizada por el mismo.

En primer lugar, debemos adaptar el comportamiento del componente que se desea entrenar para que utilice la red neuronal artificial en lugar de la máquina de estados. Al prefab del agente hay que añadir como componente *GeneticBehaviour.cs*:

Figura 98: Componente GeneticBehaviour.cs

El script GeneticBehaviour.cs está comentado de forma adecuada para indicar al usuario que partes se deben o pueden modificar. En primer lugar, debemos elegir la topología de la red neuronal:

```
////////// SET THE NN STRUCTURE //////////
private static int num_input_nodes = 5;
private static int num_hidden_nodes = 4;
private static int num_output_nodes = 3;
private static int num_hidden_layers = 1;
```

Figura 99: Seleccionar la topología de la red neuronal

En segundo lugar, debemos codificar el método que hace uso de esta red neuronal para decidir el comportamiento del componente. Este código debe ser codificado en los bloques *USER CODE* que se encuentran en el script. El resto del código no debe modificarse:

```
// Update is called once per frame
0 referencias
void Update()
{
    ////////// USER CODE //////////

    ///////////////////////////////
}


////////// USER METHODS //////////



///////////////////////////////
```

Figura 100: Ejemplo de bloques USER CODE

En el script *Base.cs* de nuestra aplicación se llama a *Decide.Decide()* para controlar el componente. Por lo tanto, en *GeneticBehaviour.cs* creamos este método (o similar):

```
////////// USER METHODS //////////

0 referencias
public void DecideDistance(float izq_2, float izq, float frontal, float der, float der_2)
{

    List<double> input = new List<double>();
    input.Add(izq_2);
    input.Add(izq);
    input.Add(frontal);
    input.Add(der);
    input.Add(der_2);
    List<double> output = simple_neural_network.processInput(input);

    //output traduction
    if (output[0] > output[1] && output[0] > output[2])
    {
        base_component.FixRotation(-1);
    }
    else if (output[1] > output[0] && output[1] > output[2])
    {
        base_component.FixRotation(0);
    }
    else if (output[2] > output[0] && output[2] > output[1])
    {

        base_component.FixRotation(1);
    }
}

///////////////////////////////
```

Figura 101: Método DecideDistance()

Hemos creado el método similar llamado *DecideDistance()*. Este método recibe las distancias de los rayos como parámetro de entrada. Seguidamente, introduce estos valores en una lista de tipo *double*. Para introducir los valores en la red neuronal, utilizamos el método *processInput*, obteniendo una lista con los valores de salida de esta. Finalmente, interpretamos estos valores obtenidos. En la figura anterior, si se activa el primer nodo de salida, se gira en sentido antihorario, si se activa el segundo no se gira, y si se activa el tercero se gira en sentido horario. En este método ha sido necesario utilizar el método *FixRotation* del script *Base*. Por ello, si se necesita cualquier componente de la aplicación lo podemos inicializar en los bloques *USER CODE*:

```
////////// USER CODE //////////
Base base_component;
0 referencias
private void Awake()
{
    base_component = GetComponent<Base>();
}
//////////////////////////////
```

Figura 102: Inicialización de otros componentes en GeneticBehaviour.cs

Por último, debemos editar los scripts de la aplicación para que en lugar de utilizar *Decide.Decide()* utilicen *GeneticBehaviour.DecideDistance()*. De esta forma, el componente llamará al método que decide en función de la salida de la red neuronal.

4. Estados del individuo y fitness

El siguiente paso consiste en indicar al módulo cuándo un agente alcanza su objetivo o cuando muere. En nuestra aplicación, el script *Detector.cs* se encarga de detectar las colisiones:

```
private void OnTriggerEnter(Collider other)
{
    if ((other.tag == "Wall") || (other.tag == "Obstacle"))
    {
        renderer_agent.material.color = Color.red;
        base_component.active = false;
        Invoke("Disappear", 2);
    }
    else if (other.tag == "Goal")
    {
        renderer_agent.material.color = Color.green;
        base_component.active = false;
        Invoke("Disappear", 2);
    }
}
```

Figura 103: Script Detector.cs

(El método *Disappear* destruye el componente). Por lo tanto, debemos editar este script para indicar al módulo esta información cuando el componente colisiona. Además, también debemos calcular el fitness del individuo:

```
private void OnTriggerEnter(Collider other)
{
    if ((other.tag == "Wall") || (other.tag == "Obstacle"))
    {
        renderer_agent.material.color = Color.red;
        base_component.active = false;
        //1 - Get GeneticEventMessage
        GeneticEventMessage genetic_event_message = genetic_behaviour.getGeneticEventMessage();
        //2 - Set fitness, just time
        genetic_event_message.setFitness(genetic_event_message.getGenerationTime());
        //4 - kill the player
        genetic_event_message.kill();
        Invoke("Disappear", 2);
    }
    else if (other.tag == "Goal")
    {
        renderer_agent.material.color = Color.green;
        base_component.active = false;
        //1 - Get GeneticEventMessage
        GeneticEventMessage genetic_event_message = genetic_behaviour.getGeneticEventMessage();
        //2 - Set fitness, time + extra score for getting to the end
        genetic_event_message.setFitness(1500 - genetic_event_message.getGenerationTime());
        //3 - Set goal achieved
        genetic_event_message.goalAchieved();
        //4 - kill the player
        genetic_event_message.kill();
        Invoke("Disappear", 2);
    }
}
```

Figura 104: Script Detector.cs editado

La comunicación entre *GeneticManager.cs* y *GeneticBehaviour.cs* se realiza mediante *GeneticEventMessage.cs*. Como se observa en la figura anterior, los pasos para indicar al módulo el fitness y si un agente ha muerto o alcanzado la meta son los siguientes:

1. Obtenemos el objeto *GeneticEventMessage* de *GeneticBehaviour*.
2. Establecemos el fitness mediante el método *GeneticEventMessage.setFitness()*. El cálculo del fitness debe ser implementado por el usuario en función de la aplicación y el componente a entrenar. Para esta aplicación establecemos el fitness indicado en *PRUEBAS Y RESULTADOS I*. (Podemos obtener el tiempo de la actual generación mediante el método *GeneticEventMessage.getGenerationTime()*).
3. Si la solución ha alcanzado el objetivo, lo indicamos llamando al método *GeneticEventMessage.goalAchieved()*.
4. Haya alcanzado el objetivo o no, debemos indicar que la solución ha finalizado. Para ello llamamos al método *GeneticEventMessage.kill()*. Este método no hace desaparecer el objeto del mapa.

Como el objeto *GeneticBehaviour* forma parte del componente, podemos obtenerlo antes de acceder a *GeneticEventMessage*:

```
private void Awake()
{
    renderer_agent = GetComponent<MeshRenderer>();
    base_component = GetComponent<Base>();
    //Obtain refference to genetic behaviour
    genetic_behaviour = GetComponent<GeneticBehaviour>();
}
```

Figura 105: Obtener GeneticBehaviour

5. Completar el objeto GeneticManager

El penúltimo paso consiste en indicar al objeto *GeneticManager* el prefab del componente que deseamos entrenar. Para ello, simplemente seleccionar el prefab desde el editor de Unity en el campo *Player* de *GeneticManager*:

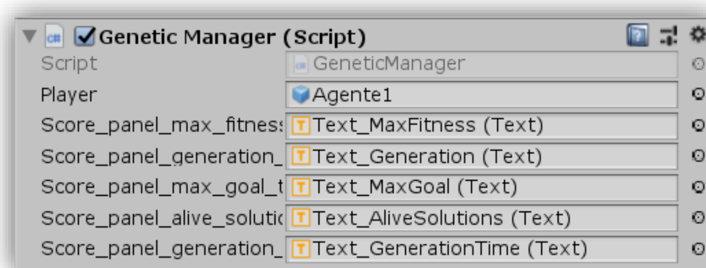

Figura 106: Objeto GeneticManager con el prefab del componente a entrenar

Es importante destacar que el prefab a utilizar debería cumplir ciertas condiciones:

- Debe disponer de elementos que permitan percibir el medio y traducir esta información en la lista de valores punto flotante que se introducirá en la red neuronal.
- De la misma forma, debe interpretar la lista de valores punto flotante de salida de la red y ser capaz de interpretar esta salida y actuar en función de esta.
- Debe ser posible ejecutar estas acciones de forma repetida a lo largo del tiempo.

6. Ajuste de la cámara

Por último, si deseamos utilizar la función *Pick Point* del módulo para seleccionar la posición inicial, o si deseamos ver la simulación en la pestaña *Game*, debemos editar la posición y rotación de la cámara del módulo para que apunte al escenario donde se realizará el entrenamiento.

7. Arrancar la aplicación

Una vez realizados todos los pasos anteriores, el módulo y la aplicación están completamente conectados y se puede comenzar la simulación:

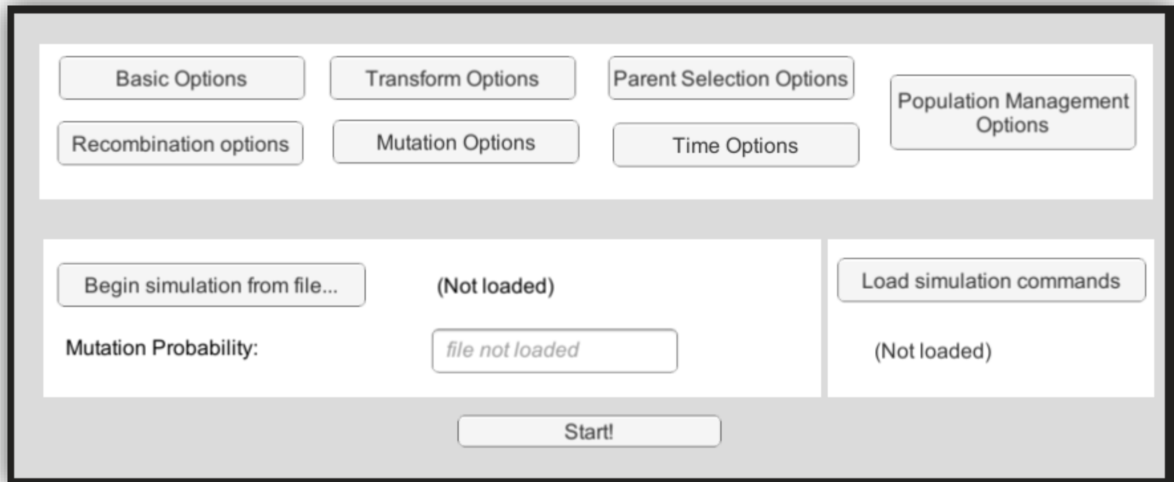

Figura 107: Panel de selección de parámetros

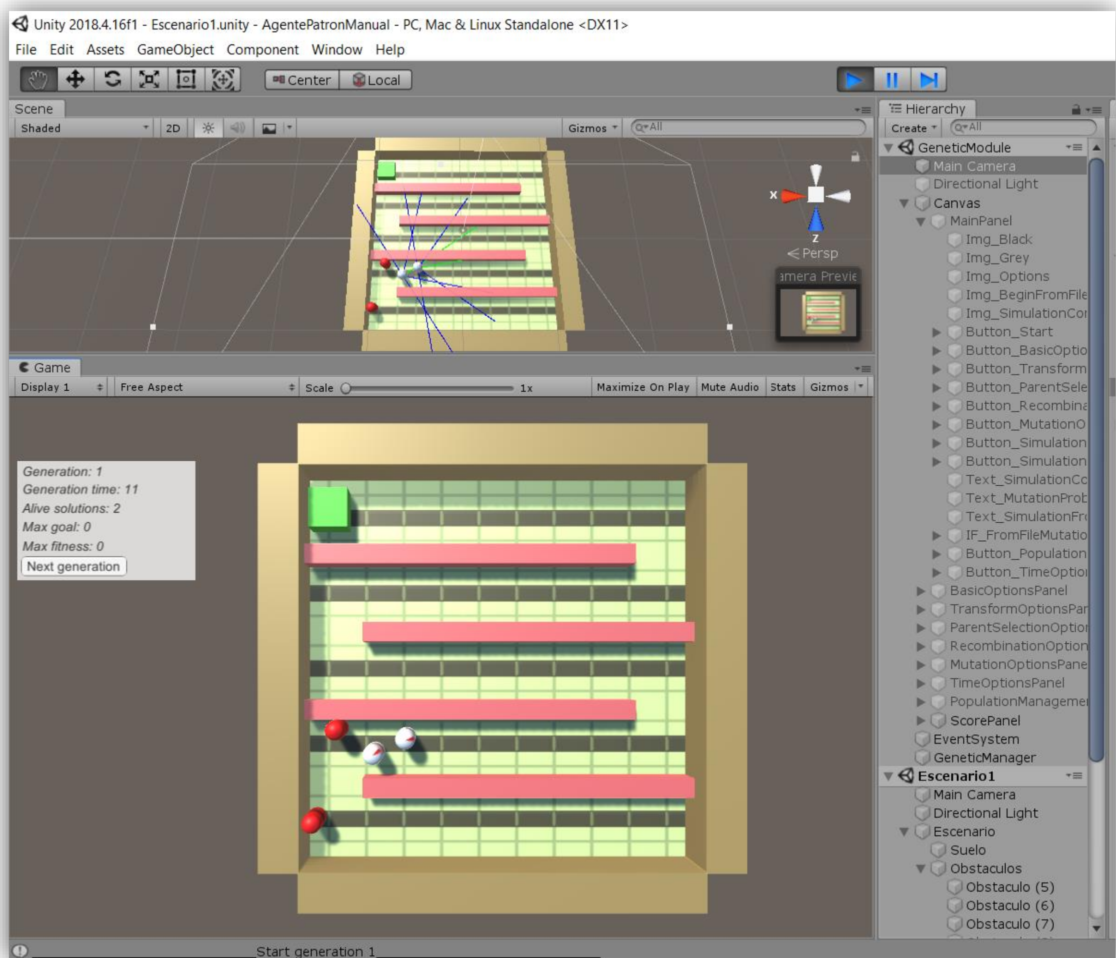

Figura 108: Simulación arrancada (generación 1)

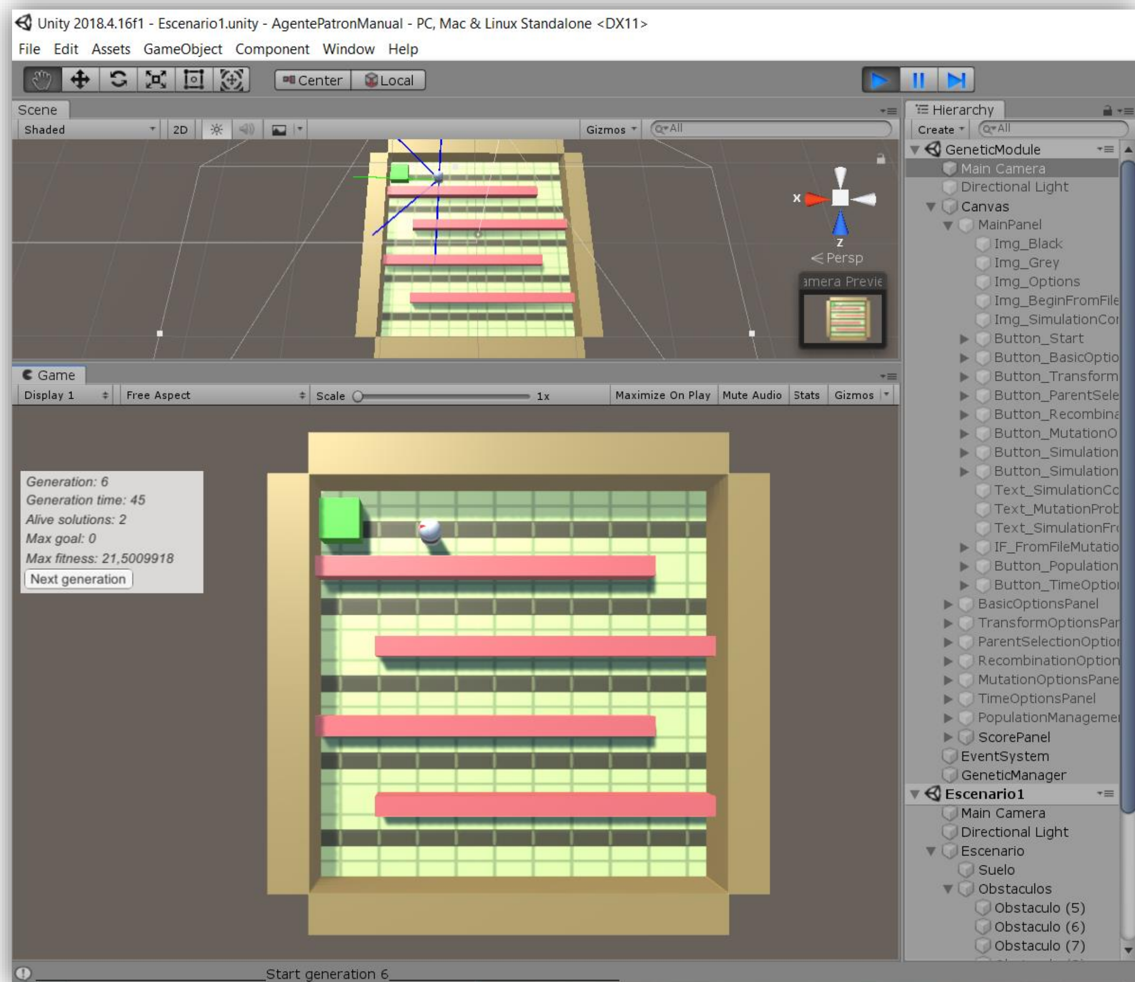

Figura 109: Primeros dos individuos en alcanzar el objetivo (generación 6)

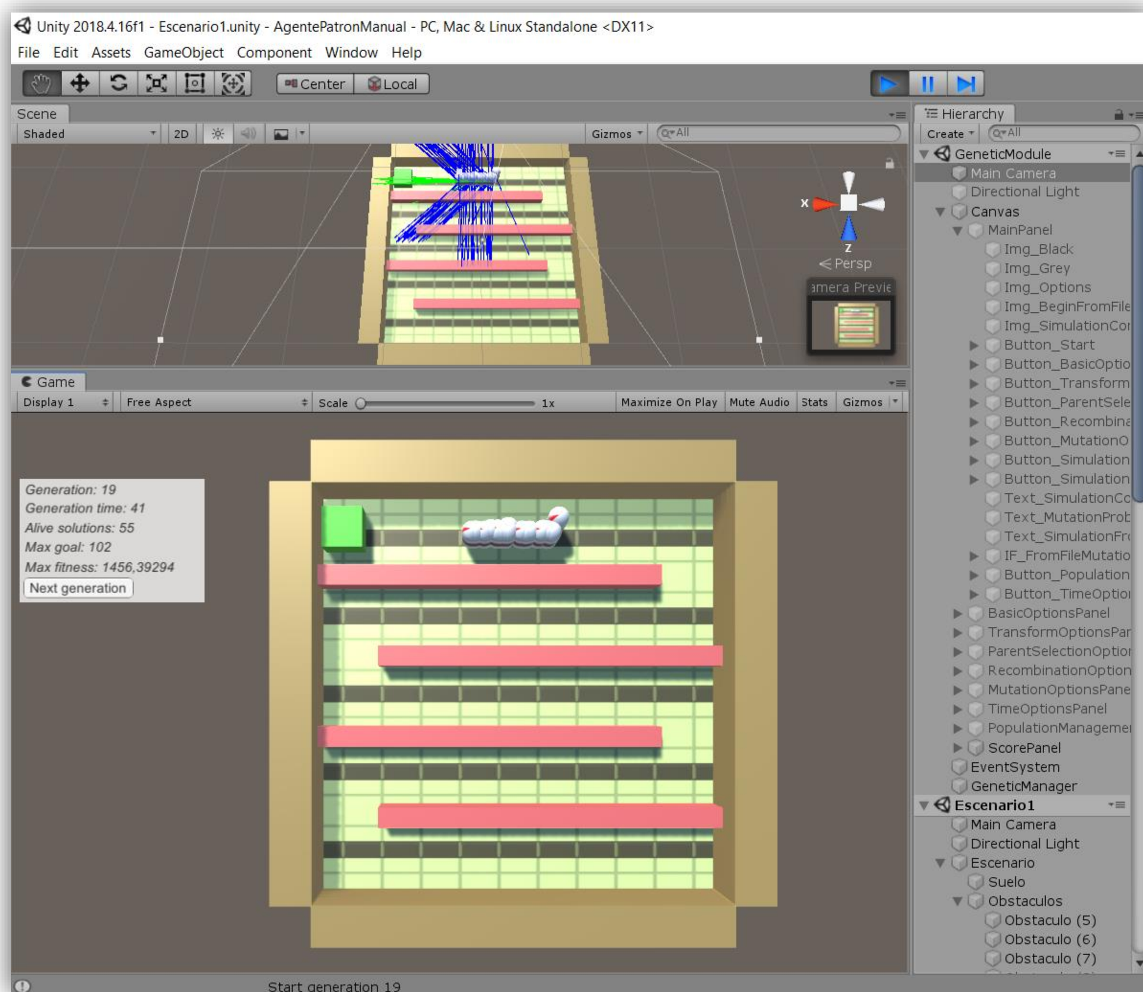

Figura 110: Convergencia obtenida (generación 19)

8. Utilización de una red neuronal entrenada

Tras finalizar el entrenamiento, podemos finalmente incorporar la red neuronal entrenada que se desee en la aplicación. Dentro de la carpeta del proyecto utilizado en el entrenamiento, encontraremos una carpeta llamada *simulation*. En esta carpeta se encuentran los ficheros correspondientes a todas las simulaciones. Dentro de esta, por cada simulación encontraremos una carpeta llamada *simulation_XX* (siendo *XX* el número de la simulación). Los ficheros de las redes neuronales de cada generación ejecutada en una simulación se encuentran en su correspondiente carpeta.

Una vez seleccionado el fichero de la red neuronal que deseamos utilizar, basta con indicar el correspondiente path en el código de la aplicación:

```java
SimpleNeuralNetwork snn = new SimpleNeuralNetwork(5, 4, 3, 1);
snn.setWeightsAndBiasList(SimpleNeuralNetworkReaderManager.readSimpleNeuralNetworkWeightsAndBias("D:/PFG/nn.xml"));
```

Figura 111: Inicialización de red neuronal entrenada

Como podemos observar, para inicializar una red neuronal con unos pesos procedentes de un fichero xml debemos:

1. Inicializar un objeto de la clase *SimpleNeuralNetwork*, indicando la topología de la red.
2. Utilizar el método *setWeightsAndBiasList* para inicializar los pesos con una lista de valores double. Por otro lado, utilizamos el método estático *readSimpleNeuralNetworkWeightsAndBias()*, de la clase *SimpleNeuralNetworkReaderManager*, para obtener los pesos del fichero que se pase por parámetro.