Universidad Rey Juan Carlos

ESCUELA SUPERIOR DE INGENIERÍA INFORMÁTICA
GRADO EN DISEÑO Y DESARROLLO DE VIDEOJUEGOS

Curso Académico 2024/2025

Trabajo Fin de Grado

Desarrollo de redes neuronales para su uso en personajes de videojuegos

Autor: Manuel Abarca Crespo
Tutor: Daniel Palacios Alonso

Esta página se ha dejado en blanco a propósito.

Resumen:

Las redes neuronales son un modelo de inteligencia artificial que puede procesar datos y aprender patrones. En el mundo de los videojuegos y su desarrollo, estas redes han ofrecido nuevas formas de crear personajes no jugadores (PNJs), de mejorar la experiencia de juego y de generar entornos dinámicos. Los PNJs se pueden crear permitiéndoles aprender del entorno y del jugador, adaptándose a estos. La experiencia mejora al reducir tiempos de renderizado y al crear efectos visuales con el uso de las redes neuronales. La inteligencia artificial generativa permite la creación automática de contenido tanto dentro del juego como en su desarrollo.

Estas diferentes maneras de abordar las redes neuronales se pueden encontrar en diversos trabajos en el mundo académico, como:

- *How to make npc learn the strategy in fighting games using adaptive AI?* Donde se estudia el uso de *Machine Learning* para generar un juego de peleas donde la dificultad de los enemigos se adapte al jugador.
- *State of the art on neural rendering*. Analiza las técnicas y proyectos existentes sobre el renderizado mediante el uso de redes neuronales.
- *Generative music in video games: State of the art, challenges, and prospects*. Recoge diferentes ejemplos de usos de la inteligencia artificial generativa para su uso en la creación de música y sonido dinámicos dentro de los videojuegos, además de analizar los problemas y desafíos que esto conlleva.

En este trabajo se pretende profundizar en el uso y creación de redes neuronales, centrándose en el desarrollo de inteligencias artificiales dentro de los videojuegos, en específico se busca desarrollar enemigos con comportamientos y estrategias que no dependan de un humano para ser creados, si no que surjan de la propia red neuronal y su entrenamiento. Con tal fin se ha analizado y hecho uso de las redes neuronales genéticas, que imitan el proceso de evolución del mundo real para mejorar su adaptación al entorno a lo largo de diversas generaciones.

Finalmente, se ha logrado usar estas redes para generar un jugador enemigo con diferentes niveles de dificultad capaz de generar estrategias en tiempo real sin un humano que le haya entrenado o le indique como actuar.

Palabras clave: *Redes Neuronales, Aprendizaje Automático, Algoritmos Genéticos, Entrenamiento no Supervisado.*

Abstract:

Neural networks are an artificial intelligence model capable of processing data and learning patterns. In the world of video games and their development, these networks have introduced new ways to create non-player characters (NPCs), enhance gaming experience, and generate dynamic environments. NPCs can be designed to learn from their surroundings and from the player, adapting to them. Gaming experience improves by reducing rendering times and creating visual effects with the use of neural networks. Generative artificial intelligence enables the automatic creation of content both within the game and during its development.

These different approaches to neural networks can be found in multiple projects in the academic world, such as:

- *How to make npc learn the strategy in fighting games using adaptive AI?* Explores the use of machine learning and use it to develop a fighting game where the difficulty adapts to the player.
- *State of the art on neural rendering*. Analyzes current techniques and projects related to rendering using neural networks.
- *Generative music in video games: State of the art, challenges, and prospects*. Includes various examples of generative artificial intelligence applications for creating dynamic music and sound in video games, as well as an analysis of the issues and challenges involved.

This work aims to delve into the use and creation of neural networks, focusing on the development of artificial intelligence within video games. Specifically, the goal is to develop enemies with behaviors and strategies that do not rely on human input but instead emerge from the neural network itself and its training process. To this end, genetic neural networks have been analyzed and utilized, mimicking the real-world process of evolution to improve adaptability to the environment over several generations.

Finally, these networks have been successfully used to generate an enemy player with varying difficulty levels, capable of creating real-time strategies without human training or guidance on how to act.

Keywords: *Neural Networks, Machin Learning, Genetic Algorithms, Unsupervised Training*

ÍNDICE

Universidad Rey Juan Carlos

ÍNDICE DE FIGURAS

1.- INTRODUCCIÓN

Mediante este trabajo se pretende ver, analizar y entender las redes neuronales y su uso en el campo del desarrollo de videojuegos, analizar sus pros y sus contras y compararlas con otras formas de inteligencia artificial más clásicas como las máquinas de estado finito.

Con el apogeo de las herramientas de Inteligencia Artificial (IA) basadas en redes neuronales en los últimos años utilizadas para la generación de contenido como *ChatGPT*, capaz de generar respuestas, preguntas y contenido en general mediante texto tal como lo haría un humano, o *DALL-E 2*, una herramienta que genera imágenes realistas a través de una simple entrada de texto, ponen de manifiesto la gran importancia que la IA tiene y tendrá en la creación de cualquier tipo de contenido en los próximos años.

Dentro del mundo de los videojuegos se han empezado a desarrollar experiencias que hacen uso de dicha IA generativa para ofrecer un contenido adaptado al jugador, como el caso de *Suck Up (https://www.playsuckup.com)*, un juego desarrollado en 2023 que pone al jugador en el lugar de un vampiro que trata de convencer a los residentes de un pueblo de que le dejen entrar a sus casas, para poder chuparles la sangre y robarles la ropa con el fin de disfrazarse. La forma que tiene el jugador de interactuar con los personajes no jugadores es mediante una consola de texto, y basándose en el texto introducido por el usuario y la personalidad del personaje, estos continuaran la conversación, echaran al vampiro de sus casas o le abrirán la puerta.

Otro ejemplo del uso de esta tecnología en videojuegos es *Oasis (https://oasis-model.github.io)*, un juego desarrollado en 2024 que simula una partida del popular juego *Minecraft*, donde cada fotograma se genera mediante inteligencia artificial a partir del anterior fotograma y las acciones del jugador, provocando que el juego no tenga memoria de las acciones o estados pasados y altere el mundo de juego constantemente.

Viendo la capacidad de creación casi ilimitada que ofrecen las herramientas mencionadas anteriormente, suena muy apetecible aprovechar esa capacidad para crear herramientas dentro de proyectos que requieran comportamientos flexibles y adaptados a diferentes usuarios sin necesidad de tener un equipo probando cada caso de uso. Un gran ejemplo, y el que se desarrollará durante el resto del trabajo, será el comportamiento de personajes y enemigos dentro de un videojuego.

El diseño manual de arquitecturas de redes neuronales es un proceso laborioso y a menudo subjetivo, que requiere experiencia y conocimiento técnico. Los enfoques tradicionales, como la búsqueda aleatoria o el ajuste de hiperparámetros por fuerza bruta, pueden ser ineficientes y no garantizar resultados óptimos. En este contexto, los algoritmos genéticos proporcionan un enfoque prometedor al aplicar mecanismos de selección natural, cruce y mutación para explorar y explotar el espacio de configuraciones de manera automática sin la necesidad de intervención humana.

2.- OBJETIVOS

- **Entender y explorar el funcionamiento de los algoritmos genéticos y las redes neuronales**
    - Explorar la historia de las redes neuronales y su funcionamiento.
    - Analizar el uso de algoritmos genéticos en las redes neuronales.
    - Lograr implementar los algoritmos estudiados.
    - Desarrollar una aplicación para usar, entender y demostrar los conocimientos adquiridos.
- **Crear un videojuego en el que se pueda implementar un enemigo controlado por una red neuronal**
    - Diseñar y documentar el juego.
    - Implementar el juego en Unity.
    - Optimizar su funcionamiento para permitir cientos de instancias simultaneas en las sesiones de entrenamiento.
- **Crear una red neuronal mediante algoritmos genéticos para controlar al enemigo.**
    - Hacer uso de los algoritmos vistos para hacer una implementación de la red neuronal.
    - Adaptar la red, sus parámetros, entradas y salidas al contexto del juego.
    - Generar un algoritmo de entrenamiento para la inteligencia artificial haciendo uso de algoritmos genéticos.
    - Entrenar la red hasta lograr enemigos lo suficientemente inteligentes para implementarlos en el juego final.
- **Analizar la inteligencia resultante**
    - Validar la inteligencia enemiga con personas ajenas al proyecto.

- Analizar las estrategias generadas por la red.
- Valorar el resultado obtenido y el trabajo que requiere llegar a el frente a otras implementaciones más tradicionales.

3.- MARCO TEÓRICO: IA Y REDES NEURONALES

3.1.- Que es la Inteligencia Artificial

> *"La IA es la capacidad de las máquinas para usar algoritmos, aprender de los datos y utilizar lo aprendido en la toma de decisiones tal y como lo haría un ser humano" (Rouhiainen, 2018, p. 17).*

> "*La Inteligencia Artificial es la ciencia de construir máquinas para que hagan cosas que, si las hicieran los humanos, requerirían inteligencia" (Marvin Minsky).*

Existen muchas definiciones posibles para lo que se entiende como Inteligencia Artificial. En líneas generales se podría definir como la habilidad de las maquinas, dispositivos electrónicos y medios digitales para exhibir un comportamiento inteligente, es decir, de ser capaces de entender, comprender, resolver problemas, tomar decisiones, razonar o aprender [7], muchas veces usando al propio ser humano como modelo de referencia, aunque no tiene por qué cumplir todas las características mencionadas.

Así, por ejemplo, podemos encontrar una gran multitud de objetos en la vida diaria que sin darnos cuenta, son inteligentes, desde cosas que se pueden pasar fácilmente por alto, como aires acondicionados que regulan su potencia en base a la temperatura de la habitación, pasando por cosas integradas en nuestro día a día como el teclado del teléfono (que autocompleta, sugiere y corrige en tiempo real) o la publicidad personalizada que aparece en cualquier aplicación, hasta lo que se entiende más popularmente como Inteligencia Artificial, superordenadores, redes neuronales, análisis fotográfico…

La necesidad de que las máquinas sean capaces de "pensar" es prácticamente obligatoria hoy en día, facilitando multitud de procesos que de otra manera serian prácticamente imposibles debido a la inversión necesaria de trabajo humano en ellos, como, por ejemplo, las antiguas operadoras de teléfono que necesitaban personal para conectar llamadas en diferentes líneas telefónicas a mano, cosa que hoy en día sería impensable ya que se producen automáticamente en menos de un segundo.

Concluyendo, la Inteligencia Artificial es la capacidad de las máquinas de analizar ciertos datos y responder a ellos, de manera similar a como lo haría un humano, pero sin necesidad de descanso y con la capacidad de hacerlo a muchísima mayor escala y de manera más rápida.

3.2.- Tipos de Inteligencia Artificial

Existen tantos tipos de Inteligencia Artificial como implementaciones hay de ella en todo el mundo, cada diferente problema requiere de un sistema que se adapte a él. Sin embargo, existen varios grandes grupos que definen a grandes rasgos como se estructuran las diferentes implementaciones que se pueden realizar de estas:

- **Búsqueda y optimización:** Este tipo de inteligencia artificial se centra en una búsqueda adecuada dentro de un conjunto de posibles soluciones. Esta búsqueda se puede realizar tanto dentro de un espacio de estados, tratando de encontrar el mejor resultado dentro de un árbol de diferentes estados (por ejemplo, una partida de ajedrez donde cada posible movimiento o respuesta es un estado y se trata de encontrar el movimiento óptimo), como lo que se conoce como búsqueda local, que consiste en el uso de funciones matemáticas de optimización para encontrar la mejor solución al problema. Un ejemplo muy típico de esta última sería el descenso de gradiente, una función iterativa que permite encontrar mínimos locales [11].

- **Lógica:** Mediante el uso del razonamiento y la representación del conocimiento, las IAs pueden deducir conclusiones a partir de otras conclusiones que ya hayan demostrado ser ciertas, lo que se conoce como premisas. Un ejemplo de este tipo de Ias se puede encontrar en el popular juego “Akinator” donde un genio adivinará mediante preguntas de si o no un personaje que el jugador este pensando [12].

- **Métodos probabilísticos:** Este tipo de Ias permiten trabajar con datos desconocidos o incompletos, lo que las hace buenas herramientas para usar al intentar predecir el clima o el cambio de las acciones en bolsa. Usan funciones y métodos tanto de la probabilidad matemática como de la economía para aproximar datos, suavizar listas incompletas y predecir eventos [11], [12].

- **Clasificadores:** Gracias a la búsqueda de patrones, estas Ias se pueden entrenar con ejemplos mediante aprendizaje supervisado, donde cada ejemplo es etiquetado para que la IA pueda aprender a reconocer los patrones de cada etiqueta y así poder clasificar posteriores elementos sin la necesidad de la etiqueta. Uno de los usos más famosos de este tipo de aprendizaje ha sido su uso médico en la detección del cáncer de mama [11], [14].

- **Redes Neuronales Artificiales:** Las redes neuronales artificiales (referidas a partir de ahora como simplemente redes neuronales) son un modelo computacional de análisis estadístico y adaptativo que se forma de manera analógica a la estructura del cerebro [15]. Están formadas por una alta cantidad de neuronas artificiales, organizadas en capas, interconectadas entre ellas compartiendo señales para ofrecer un resultado final en una capa de salida [13]. Son adaptativas en el sentido de que son capaces de adaptar sus parámetros internos en base a varios ejemplos para poder lograr un resultado más acertado, siendo capaces de encontrar patrones en los datos que le son compartidos. Junto con el *Deep Learning* estos serán los sistemas que se investigarán en este proyecto y se profundizara en ellos más adelante.

- ***Deep Learning*:** El *Deep Learning* es un conjunto de métodos empleados en el campo del aprendizaje automático para trabajar con redes neuronales. Dotan a las redes neuronales de funciones que les permiten representar los datos con los que tienen que trabajar de tal manera que les sea mucho más fácil lograr su objetivo, sin necesidad de un entrenamiento tan largo como el que necesitarían si solo trabajasen con los datos sin procesar. Estos métodos tienen múltiples niveles de representación y se suelen organizar en diferentes capas que van abstrayendo los datos originales cada vez más, hasta lograr funciones complejas con los que la red neuronal pueda trabajar [16],[17].

- **IA Generativa:** La inteligencia artificial generativa es un conjunto de técnicas computacionales capaz de crear contenido original como imágenes, video o texto, generalmente en base a unos comandos o datos de entrada. Estos modelos de IA son entrenados con datos, de los cuales aprenden sus patrones y estructuras para poder crear contenido similar posteriormente [18].

3.3.- Redes Neuronales Artificiales

Como se ha comentado anteriormente en el apartado 0, las redes neuronales son un modelo computacional basado en la estructura del cerebro. Están formadas por nodos llamados neuronas artificiales, que se conectan mediante aristas que representan la sinapsis de estas. Cada neurona recoge la señal de las neuronas conectadas a ella, la procesa y la envía a otras neuronas posteriores, organizándose en capas, donde la señal se transmite desde la capa de entrada (la primera capa) hasta la entrada de salida (la última capa) pasando por las capas ocultas (las capas intermedias). El objetivo de estas redes es resolver problemas de manera similar al cerebro humano, usándose para resolver problemas donde su resolución mediante programación tradicional seria extremadamente compleja, como la visión artificial o el reconocimiento de voz.

Si las neuronas solo transmiten su salida a las neuronas en capas posteriores la red neuronal es una red neuronal prealimentada, el tipo más común de red neuronal y el que se empleara en el resto de este trabajo de investigación. Sin embargo, también existen otro tipo de redes neuronales, más complejas, pero potencialmente más potentes que las prealimentas, las redes neuronales recurrentes. Estás se distinguen porque sus neuronas son bidireccionales y permiten que la salida de ciertas neuronas sirva como entrada posteriormente para las mismas neuronas [1].

3.3.1. Perceptrones

Los perceptrones son el tipo más simple de neurona artificial y fue uno de los primeros modelos de redes neuronales artificiales desarrollados en la historia. Fueron planteados por primera vez en 1943 por Warren McCulloch y Walter Pitts e implementados por primera vez en la "*Mark 1 Perceptron Machine*", construida en 1957 por Frank Rosenblatt, diseñada para el reconocimiento de imágenes [19].

Estos perceptrones en su modelo más básico funcionan aceptando múltiples entradas binarias y produciendo una única salida también binaria. Para calcular la salida, se le asigna a cada entrada x un peso w. Si el sumatorio de todas las entradas multiplicadas por su peso es superior a un límite, el resultado será un valor verdadero (1) y en caso contrario uno falso (0).

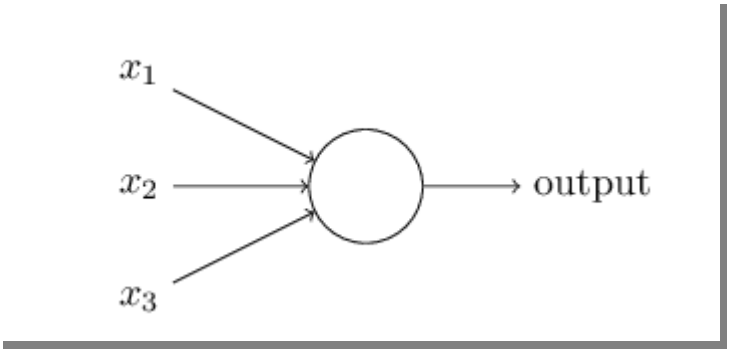

Figura 1: Representación visual de un perceptrón

3.3.2. Redes neuronales monocapa de perceptrones

Con el fin de ejemplificar el funcionamiento de los perceptrones, se han desarrollado un conjunto de redes neuronales interactivos basados en ellos, mostrando sus capacidades y sus límites.

Para esta ejemplificación se usarán datos de prueba, los cuales están adaptados a las necesidades del ejercicio y no corresponden totalmente con la realidad con el fin de poder facilitar el desarrollo del ejemplo.

El objetivo de la red neuronal será hacer de clasificador binario, indicando si una zona es válida para el cultivo de granos de café (valor equivalente a 1), teniendo en cuenta su altitud y temperatura media a lo largo del año. Los ejemplos han sido desarrollados junto con este trabajo de investigación y se puede acceder a ellos desde el siguiente enlace: *Aplicación interactiva, redes neuronales de perceptrones.* (https://dakexd.itch.io/redesneuronales)

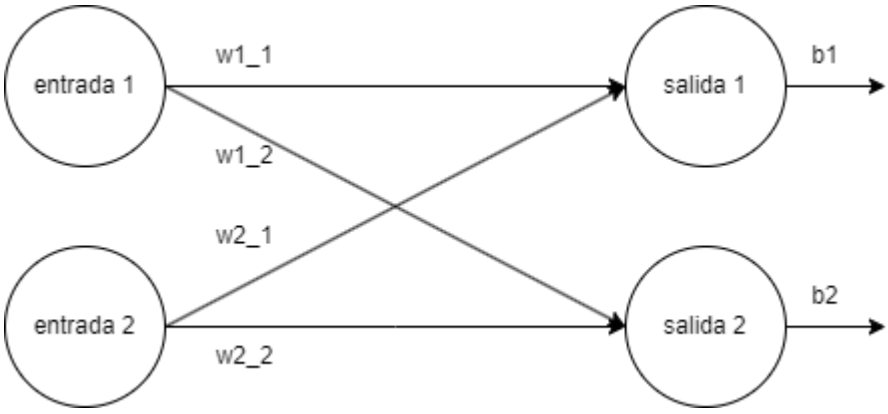

Figura 2: representación visual de una red neuronal de perceptrones

La red neuronal con la que se empezara, representada en la *Figura 2-2* es una de las más básicas posibles, consta con 2 entradas (*entrada 1* y *entrada 2*) y dos salidas (*salida 1* y *salida 2*). Cada entrada

se conecta con cada neurona de salida, y a esa conexión se le aplicara un peso *(w1_1, w1_2, w2_1* y *w2_2*). Además, cada neurona de salida tiene una inclinación, conocida como *bias* (*b1* y *b2*). El valor final, es decir, si el terreno es válido para el cultivo para el café, se obtendrá calculando el valor de las neuronas de salida (multiplicando el valor de cada entrada por su peso hacia la neurona de salida que le corresponde y sumándole la inclinación de esa misma salida). Aunque este ejemplo se podría hacer únicamente con una neurona de salida, para facilitar las explicaciones consecuentes se han establecido dos. El resultado de la red neuronal será válido (terreno cultivable) cuando el valor de la salida 1 sea mayor que el valor de la salida 2.

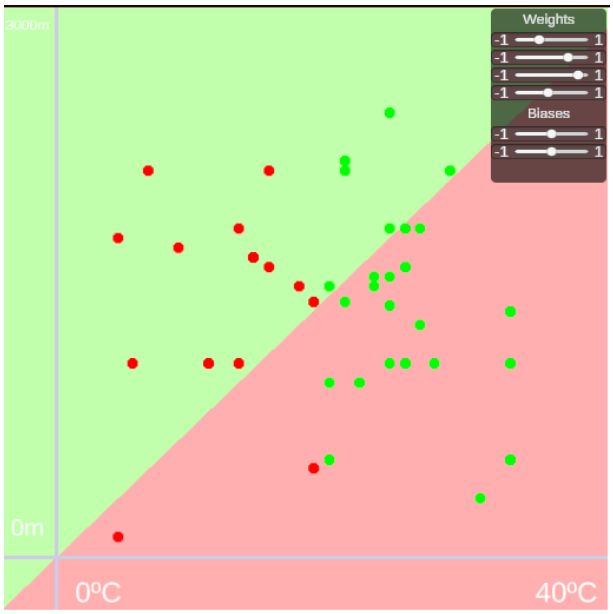

Figura 3: Gráfica correspondiente a una red neuronal básica de perceptrones

En la primera escena de la aplicación interactiva, el primer ejemplo que se encuentra es el correspondiente a la Figura 2-3. Este sistema es un perfecto ejemplo de la funcionalidad más básica de una red neuronal de perceptrones. Se pude modificar manualmente cada uno de los pesos e inclinaciones de la red y observar cómo esta afecta a los datos. En la imagen, los puntos rojos representan zonas donde el café no es cultivable, mientras que los verdes son zonas donde si lo es. La zona verde es el área que la red neuronal cree que es cultivable para cualquier punto que se encuentra en su interior, y correspondientemente, la zona roja es la que piensa que no lo es.

Si se van moviendo los controles deslizantes se podrá observar que los perceptrones funcionan como un clasificador lineal, un tipo de clasificación estadística que mediante una combinación linear

de las características de un objeto es capaz de identificar a que grupo pertenece [20]. En este caso, la separación es realizada por una recta, que separa en dos el conjunto de datos indicando cuales son válidos y cuáles no. Se puede observar que los pesos modifican la pendiente de la recta, mientras que la inclinación altera su ordenada u origen.

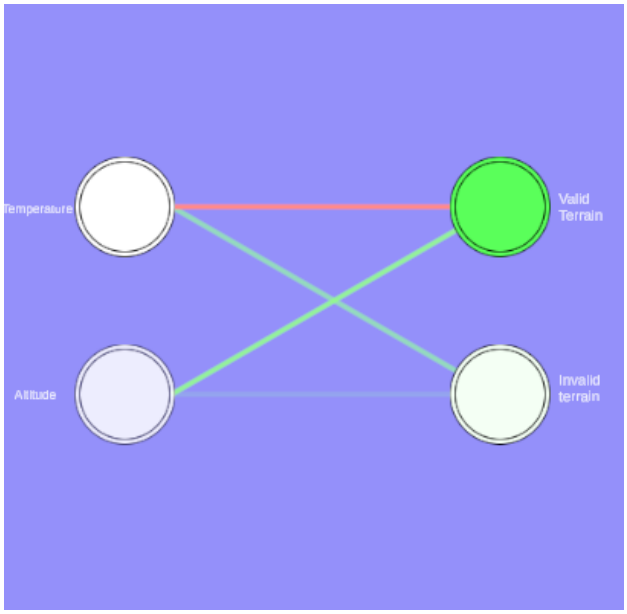

Figura 4: Representación visual de los pesos en la red neuronal

En la aplicación también se puede observar debajo de la gráfica de la red una representación gráfica del estado actual de la red neuronal, donde los pesos son las líneas que pueden ser verdes si su influencia es positiva o rojos si es negativa, además de que la transparencia del color indica la fuerza del peso, donde los pesos transparentes no ejercen influencia alguna y los pesos de color intenso ejercen una gran influencia. Del mismo modo, el color de las neuronas indica si su inclinación es positiva (verde), neutra (blanca) o negativa (roja).

Este set de datos es ideal para una red tan sencilla, cualquier conjunto de datos que sea capaz de ser separado por una línea recta puede ser procesado por un único perceptrón. En este caso, la condición de validez es que la temperatura del terreno sea superior a 18 grados, por lo que una línea sencilla con pendiente 0 y origen en 18 es suficiente. Sin embargo, los datos no siempre son tan sencillos. Si se tiene en cuenta también una altitud mínima de 800 metros, será imposible resolverlo con este método. El siguiente paso de complejidad se encuentra en el uso de la función de activación de las neuronas. Aunque existen múltiples formas de realizar esta activación, la más básica consiste en una función que activa la neurona (la salida de la neurona tomara valor 1) si la suma de sus predecesoras multiplicadas

por sus pesos es mayor que la inclinación de la neurona. En caso contrario la neurona no se activa (la salida de la neurona tomara valor 0). Todo esto viene expresado en la siguiente función matemática:

$$salida = \begin{cases} 0\ si\ \sum_j w_j x_j \leq inclinación \\ 1\ si\ \sum_j w_j x_j > inclinación \end{cases}$$

Donde *j* es cada neurona que conecta con la salida, *w* es el peso de esa neurona conectada con la salida y *x* es el valor de salida de esa neurona.,

Se puede reordenar este sistema para facilitar en gran medida su interpretación e implementación. El primer cambio seria cambiar $\sum_j w_j x_j$ por un producto escalar, w·x, donde tanto w como x son vectores cuyos componentes son los pesos y los valores de salida de las neuronas de entrada. Además, podemos cambiar de lado la inclinación, y le daremos una equivalencia nueva para simplificar, obteniendo *b* = -inclinación. Tras estos pasos se obtendrá una función mucho más manejable:

$$salida = \begin{cases} 0\ si\ \ w \cdot x + b \leq 0 \\ 1\ si\ w \cdot x + b > 0 \end{cases}$$

Tras implementar estas modificaciones, se podrán ver en funcionamiento en la segunda escena de la aplicación, además, el set de datos se ha modificado a uno que no se podía resolver con la primera implementación de la red neuronal como se puede ver en la siguiente figura.

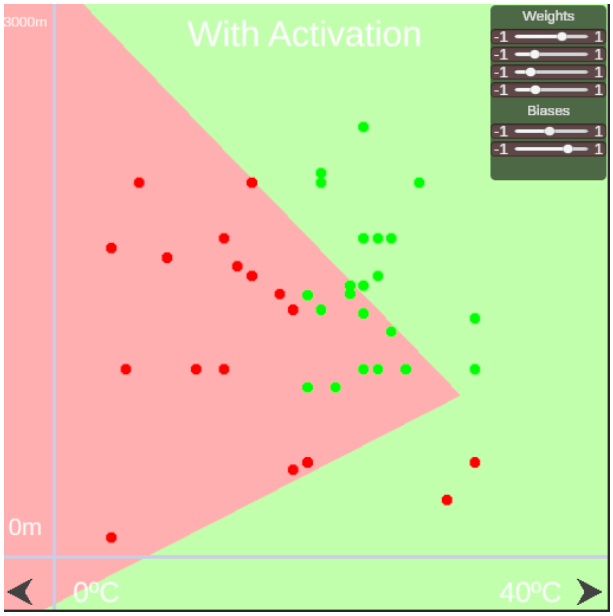

Figura 5: Red neuronal de perceptrones con función de activación

Tal y como se puede observar al ejecutar la aplicación, la red neuronal sigue siendo un clasificador lineal. En este caso, en vez de separar los datos en base a una línea, al estar usando una función de activación esta permite que los pesos tengan un efecto no linear en el resultado de la red, la cual se comportara como un sector circular. Visualmente los datos se separan en torno a un punto central y un ángulo que indica la amplitud del área dentro del cual los puntos son válidos. Esto permite que ahora los puntos puedan cumplir dos condiciones, que la temperatura sea mayor de 18 grados y la altitud mayor de 800 metros.

Por desgracia, este es el nivel máximo al que pueden llegar las redes neuronales de perceptrones de una sola capa. Los granos de café no solo tienen una mínima altitud y temperatura, también tienen una máxima que pueden soportar, pero para poder implementar eso se necesitara implementar una red neuronal multicapa.

3.3.3. Redes neuronales multicapa de perceptrones

Las redes neuronales de una sola capa no tienen una gran utilidad más allá de su función didáctica. Los problemas que puede resolver, como se ha visto en el apartado anterior, requieren que el conjunto de datos que analiza este separado de una forma en concreto para que la red pueda separarlos, de tener un conjunto separado tan claramente se podrían usar funciones directamente para hallar la recta que los diferencia. Para problemas de mayor complejidad, donde la solución no se puede encontrar con

rectas, los datos estén acotados en los ejes o exista un sistema con más de dos dimensiones, la complejidad de nuestra red neuronal tiene que aumentar.

El realizar esto teniendo en cuenta lo visto en el apartado interior es sencillo, se diseña una nueva red neuronal, que, en vez de conectar las entradas con las salidas, agregue una capa de neuronas internas, conocidas como capa oculta. Las capas se organizan en orden y todas las neuronas de una capa tienen que conectarse con cada una de las neuronas de la capa siguiente, formando una red que crece en complejidad muy fácilmente. El tamaño y cantidad de capas ocultas necesarias varia completamente dependiendo del problema y su complejidad, en este caso se usará la siguiente red:

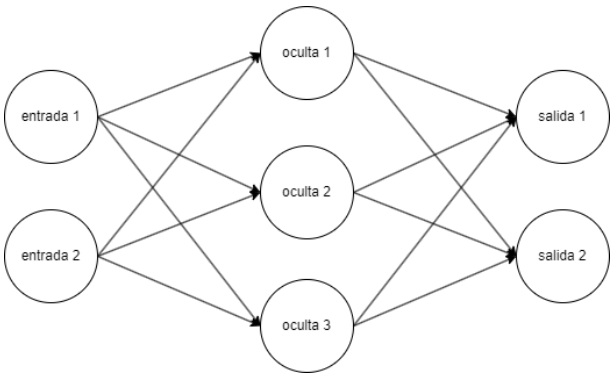

Figura 6: Red neuronal multicapa

Como se puede observar, la red es similar a la del apartado anterior, con una capa oculta de 3 neuronas separando la entrada y la salida. Todas las entradas conectan con cada una de las neuronas de la capa oculta y estas a su vez conectan con cada neurona de salida. Sigue siendo una red muy sencilla, pero la cantidad de pesos ha aumentado exponencialmente, pasando de 4 pesos y 2 inclinaciones a 12 pesos y 5 inclinaciones. Esta red se ha implementado en la tercera escena: Redes neuronales multicapa.

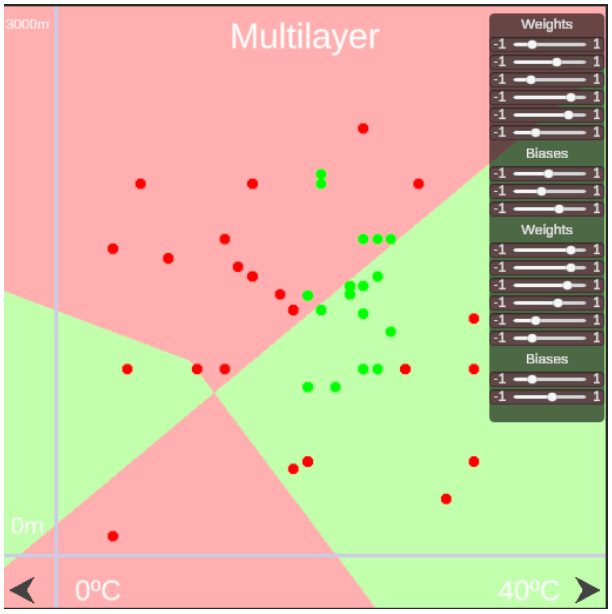

Figura 7: Gráfica correspondiente a una red neuronal multicapa

Al usar los controladores de la red neuronal se puede observar que la complejidad de uso ha aumentado a la par que la complejidad de la red, y manejar tantos parámetros hace que sea extremadamente difícil para un humano ajustarla para que el área verde coincida con los puntos válidos. Por otro lado, las áreas se vuelven más complejas y es posible separar conjuntos de datos que estén agrupados de maneras mucho más complejas. Aumentando la cantidad de capas ocultas permitiría mejorar la complejidad de las áreas, y añadir múltiples capas de entrada y salida convertirían esta separación bidimensional en una de múltiples dimensiones, difícil de representar y entender para el ser humano pero sencillo para la máquina. Por ejemplo, si en vez de 2 entradas, hubiese 3, el espacio del problema en vez de un plano sería un área tridimensional, donde el tercer dato detonaría la profundidad de los puntos y la red calcularía volúmenes en vez de áreas.

Un punto negativo de esta red neuronal multicapa es su función de activación. Cumple su cometido, sin embargo, al usar una función de paso, un pequeño cambio en un peso puede cambiar totalmente el resultado de la red. Lo ideal para tanto el manejo de la red como para que la maquina sea más adelante capaz de aprender con facilidad, es que pequeños cambios en los pesos produzcan pequeños cambios en los resultados. Una forma de obtener esto es con un nuevo tipo de neurona, la neurona sigmoide.

3.3.4. Red neuronal sigmoide

Las redes vistas hasta ese punto usan los perceptrones, en los cuales se puede observar que ligeros cambios en los pesos o las inclinaciones alteran el resultado abruptamente. Esto se debe a que su función de activación es extremadamente sencilla y el valor que devuelve cambia de 0 a 1 en un único punto. Esto no solo afecta a una persona que este probando a modificar los pesos a mano, si no que, al implementar algoritmos de aprendizaje, dificulta mucho el llegar a un resultado optimo. Mientras se realiza el aprendizaje automático se requiere que pequeños cambios en los datos se traduzcan en pequeños cambios en el resultado. De esta manera todo el sistema será más eficiente.

Para solucionar esto, se propone el uso de las neuronas sigmoides en lugar de los perceptrones. Su principal diferenciación es la función de activación de cada uno. La del perceptrón cambia de 0 a 1 en un punto fijo del intervalo numérico. La sigmoide va cambiando lentamente a lo largo de él, facilitando visualizar el efecto de pequeños cambios. Además, esto hace que la salida de cada neurona sea diferente a únicamente 0 o 1 como pasa en los perceptrones, sino que puede estar en el intervalo entre esos dos números, lo que resulta en áreas más complejas en la representación gráfica de esta red. La función sigmoide es la siguiente (donde *w* es el vector de pesos, *x* el vector de entradas y *b* el vector de inclinaciones):

$$salida = \sigma(w \cdot x + b)$$

Donde σ equivale a:

$$\sigma(z) \equiv \frac{1}{1 + e^{-z}}$$

Por lo tanto, podemos expresar la salida de forma completa:

$$salida = \frac{1}{1 + e^{(-w \cdot x - b)}}$$

Podemos observar mejor la diferencia entre ambas funciones mirando sus funciones de activación en una gráfica:

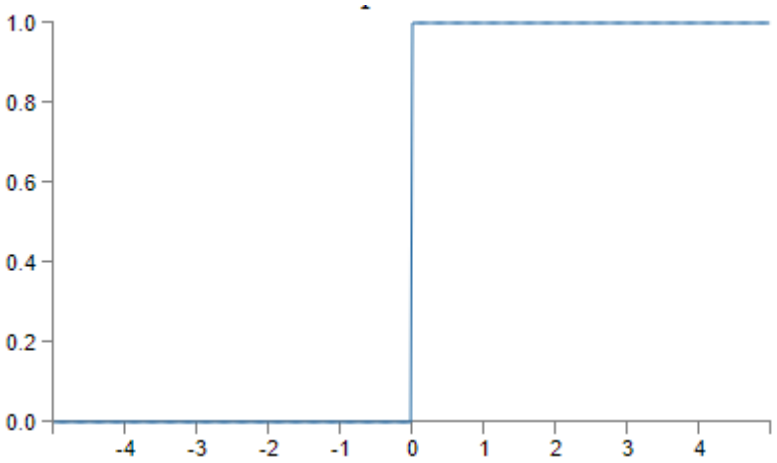

Figura 8: Función de paso

La función de activación de los perceptrones, también conocida como función de paso, pasa de 0 a 1 en un único punto.

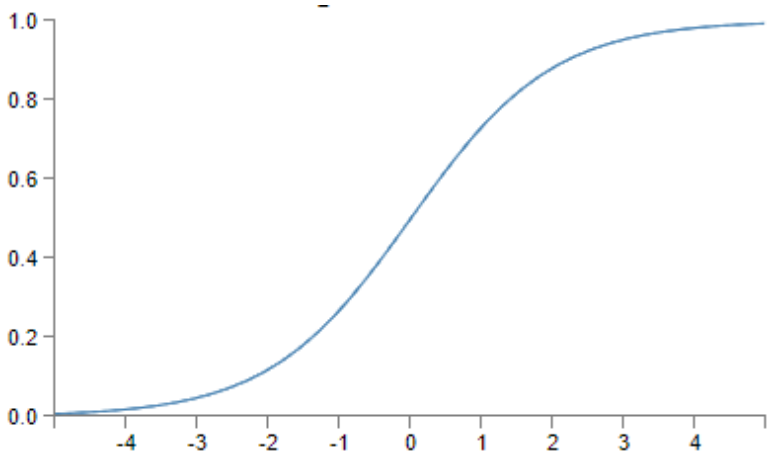

Figura 9:: Función sigmoide

La función sigmoide obtiene su valor intermedio en el punto donde la función de paso cambiaria su valor por completo se puede observar que durante el tramo inicial se aleja lentamente del 0, en el centro cambia rápidamente de valor, y al final vuelve a acercarse lentamente al 1. Esto permite mucha más flexibilidad al manejar las redes y una mayor complejidad en la diferenciación de conjuntos de datos al no limitar su valor a 0 o 1.

En la aplicación interactiva, se ha desarrollado la cuarta escena, redes neuronales sigmoide, que visualiza una red neuronal sigmoide con el mismo diseño que la vista en el apartado anterior, 2 neuronas de entrada, 3 neuronas ocultas y 2 neuronas de salida

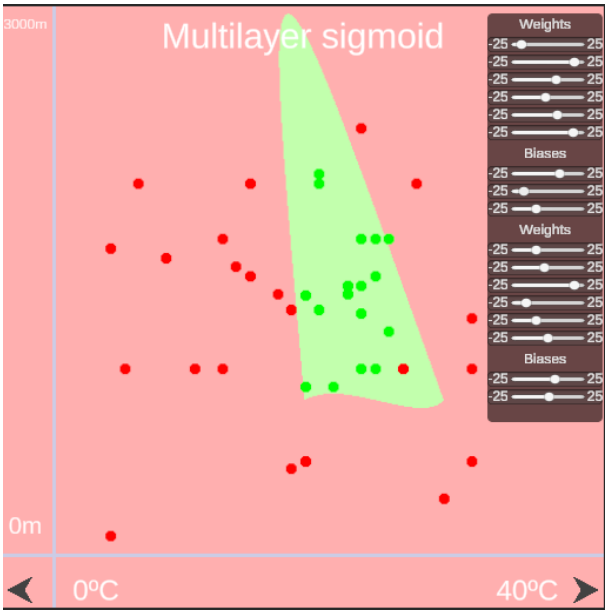

Figura 10: Gráfica correspondiente a una red neuronal multicapa sigmoide

Si se compara esta red con la red neuronal anterior se puede notar que la complejidad del área que es capaz de procesar ha aumentado muchísimo, permitiendo definir zonas curvas y mejor acotadas. Esto es debido a que la función sigmoide no se limita a dos únicos posibles números, el 0 o el 1, si no que puede otorgar cualquier valor decimal entre esos dos. Una alteración respecto a la gráfica anterior es el aumento del rango de los pesos, ya que al producirse los cambios más lentamente respecto a una función de paso el rango anterior de valores entre -1 a 1 quedaba corto para poder mostrar graficas más avanzadas, por lo que se hace uso de un rango de valores entre -25 y 25. (Esto solo afecta al valor de los pesos *w y* las inclinaciones *b*, el valor de salida de cada neurona seguirá en el intervalo entre 0 y 1).

3.3.5. Función de coste

Ya sea que la red se esté entrenando a mano como si se está entrenando automáticamente (como se verá más adelante) se necesita definir de alguna manera lo bien o mal que la red se está desempeñando. La manera más intuitiva de hacer esto sería contar el número de puntos conocidos que se están clasificando correctamente, pero esto impide que pequeños cambios en la red alteren el resultado si no significa que un nuevo punto está clasificado correctamente, lo que resulta en que no hay manera de saber si ese cambio ha sido beneficioso o perjudicial.

La manera de solucionar esto es definir lo que se conoce como función de coste, función de pérdida o función de error. Esta función asocia un valor numérico a un conjunto de una o más variables,

representando el "coste" asociado a ellas [21]. En este contexto, el coste será una penalización por una clasificación incorrecta de datos. El objetivo de la red neuronal y los algoritmos de aprendizaje que se verán más adelante será reducir en la medida máxima de lo posible este coste. Existen una gran variedad de diferentes funciones de coste, pero las dos más sencillas y prácticas para redes neuronales son:

Función de coste 0-1

Esta función valora únicamente la cantidad de nodos clasificados incorrectamente. Evalúa todos los puntos y por cada uno agrega un valor al coste de la red. El coste añadido será 0 en el caso de que el punto evaluado sea correcto y 1 en caso contrario. La manera de expresar la función de perdida *L* para un conjunto de valores objetivos $\hat{y}$ y un conjunto de valores clasificados *y* es la siguiente:

$$L(\hat{y}, y) = [\hat{y} \neq y]$$

Función de coste Cuadrática

Esta función es usada comúnmente por su varianza y su simetría, lo que la hace matemáticamente más manejable. En esta función un error por encima del objetivo causa una pérdida de la misma magnitud que un error equivalente por debajo del objetivo. Si el objetivo es *t*, la ecuación de la función de coste Cuadrática es:

$$\lambda(x) = C(t - x)^2$$

Donde C es una constante, conocida como Perdida de Error Cuadrático (*SEL* por sus siglas en inglés) [21]. El valor de esta constante no causa diferencias en las decisiones, ya que todos los valores de perdida estarán multiplicados por esta constante, por lo que se puede ignorar asignándole un valor de 1. Como se ha mencionado anteriormente, *t* es el valor, por lo que *x* es el valor definido por el sistema.

La implementación de esta función de coste se puede observar en la quinta escena, función sigmoide, donde se usa la red neuronal sigmoide definida en el apartado anterior y observar el cambio del coste de la red neuronal. Si se usase la red neuronal de perceptrones no tendría sentido usar la función de coste cuadrática, ya que el resultado de las neuronas de salida solo puede ser 0 o 1. Como las neuronas sigmoides tienen un valor intermedio de salida, se puede usar ese valor como la seguridad que tiene la red de que la respuesta es correcta. Es decir, en el caso de la red de la aplicación, la cual clasifica terrenos dependiendo si son hábiles para el cultivo de café, tiene dos neuronas de salida. Una indica su seguridad respecto a que el terreno es cultivable y la otra respecto a que no lo es. La respuesta de la

red es que el terreno es válido si su seguridad de que es cultivable supera a la de que no lo sea, por lo tanto, aunque la red tenga un punto donde correctamente ha indicado que es cultivable, puede seguir teniendo margen de mejora, aumentando su seguridad en la respuesta. De la misma manera, el tener la respuesta correcta no elimina todo el peso de coste de la red, si no que cuanto más segura este la red de la respuesta más se reducirá el coste, dando margen a la optimización del sistema. Para evitar que diferentes sets de datos alteren el coste total de la función, el valor de la función de coste será la media de todos los puntos, los que permite una forma más intuitiva de determinar si la red está mejorando o empeorando.

Al emplear una red que cuenta con dos salidas, la función de coste podrá otorgar un valor comprendido entre 0 y 2. Cada neurona de salida otorga un coste entre 0 y 1 utilizando la función de coste cuadrática y el coste de un punto de la red es la suma del coste de todas sus salidas.

Por ejemplo, un nodo valido para el cultivo debería tener una salida de 0 en su neurona de salida etiquetada como terreno no cultivable y de 1 en la etiquetada como si cultivable. Si el resultado que otorga la red es el opuesto (1, 0), el coste de este nodo tendrá valor 2. Si el resultado que otorga es el correcto (0, 1) el coste será 0. Para valores intermedios, como (0, 0.5) el coste se encontrará entre esos dos valores, siendo en este caso 0.25 ($(0-0)^2 + (1-0.5)^2$). Cuanto más cerca de 0 este el coste de la red, más correcta será la respuesta que esta otorga.

3.4.- Aprendizaje Automático

El aprendizaje automático es un campo de la inteligencia artificial que se encarga del estudio de algoritmos estadísticos capaces de desarrollarse a partir de un conjunto dado de datos y aplicarse con posterioridad a otros conjuntos de datos nunca vistos, logrando resolver problemas para los cuales no se han creado instrucciones específicas. Esto es de especial utilidad en tareas cuya implementación es especialmente compleja, como la visión artificial o el reconocimiento de voz [22].

Generalmente el aprendizaje automático se divide en tres categorías principales, correspondientes a diferentes paradigmas de aprendizaje y basados en la naturaleza del manejo de los datos de aprendizaje.

- **Aprendizaje supervisado:** Se presentan a la maquina un conjunto de datos de entrada emparejados con su respuesta esperada por el sistema (también conocido como etiqueta). La

máquina debe desarrollar una regla que consiga asociar correctamente las entradas y las respuestas.

- **Aprendizaje no supervisado:** La premisa es la misma que la del aprendizaje supervisado, desarrollar una regla que asocie entradas con sus respuestas correctas. En este caso, sin embargo, los datos no vienen con etiqueta y la maquina tiene que buscar patrones y similitudes en los datos por ella misma.

- **Aprendizaje por refuerzo:** La máquina interactúa en un entorno dinámico que otorga respuestas a sus acciones, las cuales funcionan como recompensas que la maquina debe intentar maximizar para llegar a cumplir un objetivo. Este es el tipo de aprendizaje que se usara más adelante a la hora de crear redes neuronales para videojuegos.

En el apartado 3.3.5. Función de coste se definió la función de coste. Esta función leía todos los valores de las neuronas de salida y devolvía un valor numérico indicando lo bien o mal que la red neuronal estaba funcionando, cuanto más pequeño el valor mejor lo hacía la red, siendo 0 el valor devuelto por una red perfectamente entrenada y funcional.

Si se encuentra un algoritmo capaz de ajustar los pesos e inclinaciones de la red de manera que reduzca el resultado de la función de coste, ese algoritmo estará haciendo que la red aprenda automáticamente. Esta es la base del aprendizaje automático, donde uno de los algoritmos más usados para lograr este objetivo es el descenso de gradiente.

3.4.1.- Descenso de Gradiente

Para resolver un problema grande, muchas veces lo mejor es simplificar el problema, resolverlo y aplicar la solución al original. Antes de definir como minimizar una función con miles de capos de entrada y decenas de salidas, lo mejor es aprender a minimizar una función con una entrada y una salida.

Poniendo por ejemplo la función cuadrática $0.2x^4 - 0.2x^3 - 2x + 0.5$ se obtendrá una función con la siguiente figura:

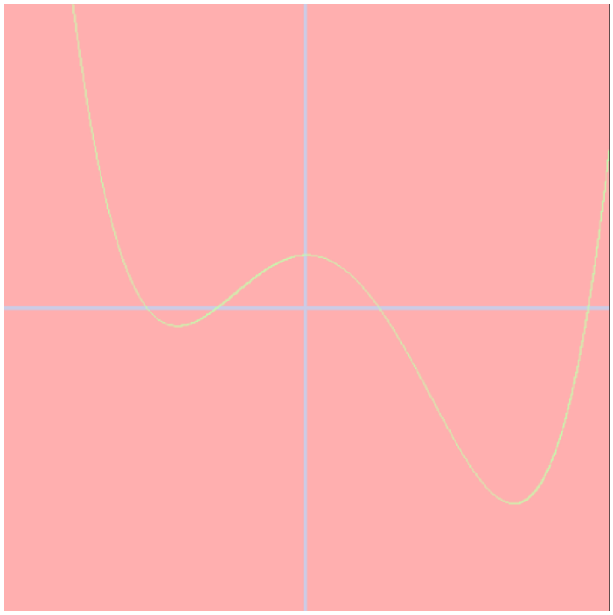

Figura 11: Función a minimizar

El objetivo del descenso de gradiente es hallar el mínimo absoluto de la función. Conociendo la fórmula de la función sería posible hacer cálculos matemáticos para hallarlo, sin embargo, estos cálculos serian prácticamente imposibles con funciones más complejas como las que se usan en las redes neuronales.

Otra forma de hallar este valor es comprobar la pendiente de la función en un punto aleatorio, que será referido como punto de control, desplazarlo una cantidad de espacio determinada en la dirección de dicha pendiente y repetir el proceso hasta encontrar un valor donde no se pueda seguir descendiendo en ninguna dirección, es decir, un mínimo. Este mínimo encontrado seria relativo, no existe forma de saber si es el mínimo absoluto solo con una comprobación. Por lo que para hallar el mínimo absoluto se puede repetir este proceso varias veces y comparar los resultados, de manera que se pueda asegurar con un gran grado de confianza de haber encontrado el valor deseado.

Para calcular la pendiente de la función para el punto de control, se define un factor de aprendizaje con un valor numérico muy pequeño, llamado *h*. La pendiente *p* en un punto *x* se puede calcular fácilmente restando el valor de la función en $x + h$ *y* en *x*. Si *p* es positivo, la función en ese punto descenderá hacia el lado izquierdo de la gráfica, y si es negativo la función tendera hacia la derecha.

Una vez conocida la dirección en la que se va a avanzar, se puede definir un valor de aprendizaje que controle lo mucho que afecta la inclinación al desplazamiento del punto de control. El resultado de multiplicar el valor de la pendiente por este valor de aprendizaje será lo mucho que se desplazará el

punto de control. De esta manera, lugares de la función donde la pendiente es muy inclinada harán que el punto de control avance más rápido, mientras que aquellas con una inclinación más baja alteraran menos el desplazamiento de la comprobación. Este comportamiento se puede comprobar en la sexta escena de la aplicación, descenso de gradiente.

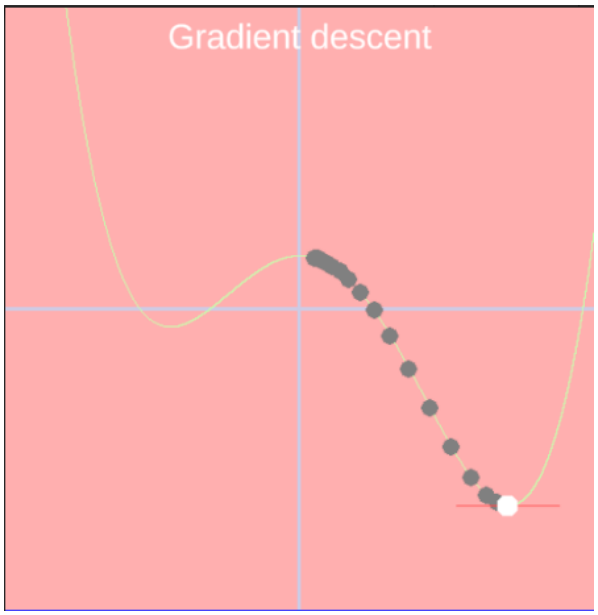

Figura 12: Función minimizada

Se pueden usar los diferentes controladores de la aplicación para observar el aprendizaje por descenso de gradiente y como el punto de control se va desplazando a lo largo de la función hasta encontrar su mínimo, sin necesidad de complejas funciones matemáticas.

La función que se ha visto consta de una única variable, por lo cual se puede representar en un sistema de dos dimensiones. Todo esto se puede aplicar a una red neuronal. Si hubiese, por ejemplo, dos pesos que afectan al resultado de la red, la función de coste de la red se podría representar en un sistema de tres dimensiones, formando una malla como la siguiente, donde *v1 y v2* son los dos pesos que afectan a la red y *C* es el valor de la función de coste:

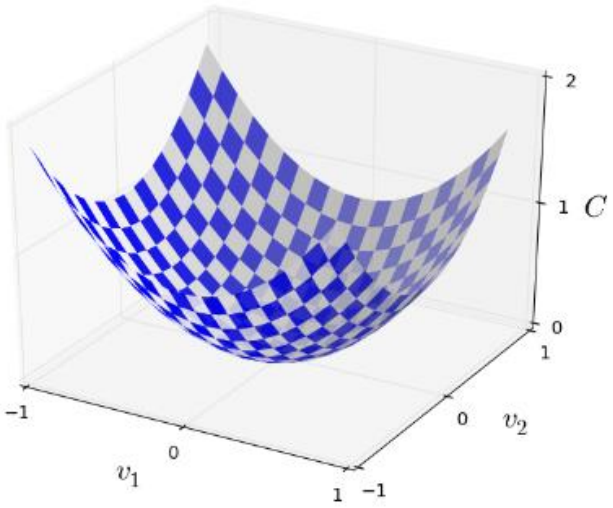

Figura 13 Función a minimizar en 3D

Al igual que en la función en dos dimensiones, dado un punto de control podemos buscar un mínimo avanzado en dirección de la pendiente. Para calcular la pendiente anteriormente se ha dividido el cambio en el resultado de la función entre el cambio en la posición, lo que se puede representar con la siguiente formula:

$$\frac{\Delta f}{\Delta x}$$

Para calcular el descenso de gradiente para dos variables se seguirá exactamente la misma lógica. Hay que tener en cuenta que en este caso cada variable se verá modificada por separado, es decir, se calcula la pendiente para cada una de las variables y se desplaza el punto de control por ambas a la vez, lo que se puede representar con la siguiente formula:

$$\left[\frac{\Delta C}{\Delta v_1}, \frac{\Delta C}{\Delta v_2}\right]$$

Cuantos más variables afecten el resultado de la red más dimensiones se necesitan para representar el sistema, pero la lógica para encontrar los mínimos y optimizar la función permanecerá exactamente igual, por lo que para una red con *n* variables que afectan a la función de coste podemos aplicar el descenso de gradiente con la formula:

$$\left[\frac{\Delta C}{\Delta v_1}, \ldots, \frac{\Delta C}{\Delta v_n}\right]$$

Toda esta lógica se puede aplicar a la red neuronal vista hasta ahora, lo que da como resultado la séptima escena de la aplicación, aprendizaje con descenso de gradiente.

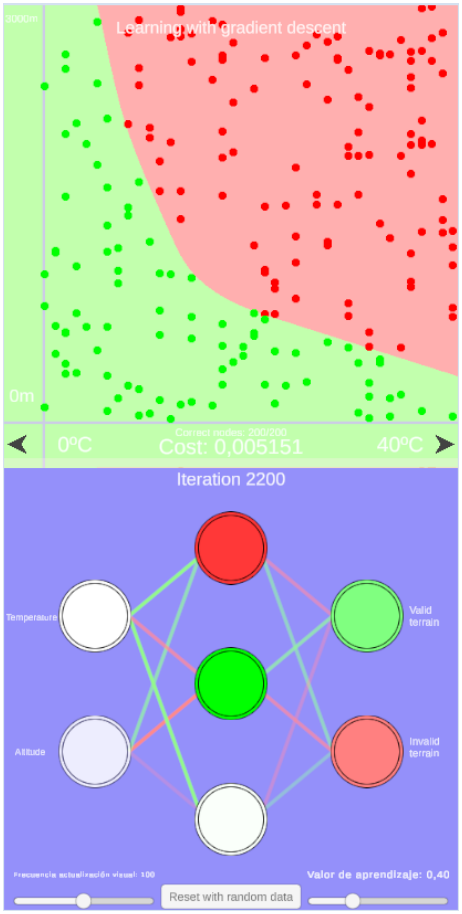

Figura 14: Red neuronal con aprendizaje automático

En esta escena la data se genera de manera pseudoaleatoria para que cada vez la red tenga que aprender un patrón distinto. El algoritmo de descenso de gradiente funciona tal y como se ha explicado anteriormente. Primero se calcula el coste de la red. Después por cada peso e inclinación se les añade un valor muy pequeño, *H*, y se calcula el coste de la red con ese único cambio. La diferencia entre el valor de coste original y el nuevo se dividen otra vez entre *H*, cuyo resultado es el gradiente, se guarda para aplicar posteriormente y se devuelve el peso o inclinación a su valor original (a nivel de implementación esta es la principal diferencia). Tras realizar esto para cada uno de los pesos e inclinaciones, se le resta su correspondiente gradiente multiplicado por el valor de aprendizaje.

En la implementación que se ha realizado de esta técnica, existen varios factores que limitan la velocidad de aprendizaje de la red.

El primero es el actualizar la vista de la red. Para actualizar la imagen se tiene que calcular el resultado para cada uno de sus pixeles. Al ser una imagen de 512 x 512 esto es un total de 262144 iteraciones. Para solventar esto, mientras se está aprendiendo, en vez de calcular el valor de cada píxel de la red, se aproxima, calculando el valor para un píxel y aplicando ese valor en un área determinada a su alrededor, disminuyendo enormemente la cantidad de iteraciones necesarias para dibujar la red. Otra manera de disminuir el efecto de esto ha sido establecer una frecuencia de actualización, de manera que la imagen solo se actualiza cada vez que se realizan tatas iteraciones de aprendizaje como determina ese valor.

El segundo factor que limita la velocidad es que cada iteración de aprendizaje necesita recorrer cada uno de los datos de aprendizaje establecidos, lo cual a una escala pequeña como la de esta red no supone un gran problema, pero si aumenta la cantidad de ejemplos puede resultar en un aprendizaje muy lento (y hay que recordar que cuantos más ejemplos de entrenamiento tenga la red más eficiente será). Para resolver esto se pueden seleccionar una cantidad reducida de ejemplos en cada iteración de aprendizaje en vez de usar todo el set, de manera que reduce drásticamente la cantidad de iteraciones necesaria, además de añadir ruido al aprendizaje (ya que cada iteración de aprendizaje se encontrara con puntos diferentes y el resultado que busca cada uno no es exactamente el mismo). Este ruido, al contrario de ser malo, puede ayudar a acelerar el aprendizaje evitando que este se estanque en zonas donde el gradiente avance muy lentamente. Esta técnica se conoce como *mini-batching*, o mini procesamiento por lotes en español.

Por último, al tener que calcular la función de coste para cada peso en cada iteración de aprendizaje supone un gran gasto computacional, sobre todo en redes grandes con muchos pesos donde calcular la función de coste para cada uno de ellos significa una ralentización muy pronunciada. Debido a ello, se necesitan algoritmos nuevos que aceleren el cálculo del gradiente y eviten la necesidad de calcular la función de coste para cada elemento. El algoritmo que se va a explicar se conoce como retropropagación (o *backtracking* en inglés).

3.4.2.- La regla de la cadena

Antes de empezar con la retropropagación es importante entender la regla de la cadena. En calculo, la regla de la cadena es una fórmula que expresa la derivada de la composición de dos funciones diferenciales en términos de las derivadas de estas mismas funciones.

Si una variable *z* depende de una variable *y*, la cual a su vez depende de una variable *x*, entonces *z* depende de *x* también a través de la variable intermedia *y*. Dado este caso se puede expresar la regla de la cadena en notación de Leibniz (es decir, donde *dx* representa un aumento infinitesimal de *x* y Δx representa una variación finita de *x*, donde $\frac{dz}{dx}$ representa como z cambia cuando *x* altera su valor) como:

$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx}$$

La explicación intuitiva de esta regla es que, conocida la varianza instantánea de *z* relativa a la de *y*, y la de *y* relativa a la de *x*, se puede calcular la varianza instantánea de *z* relativa a *x* como el producto de dos varianzas [23].

Una analogía expresada por un famoso matemático americano explica la regla de la cadena como:

> *Si un coche viaja el doble de rápido que una bicicleta, y la bicicleta viaja 4 veces más rápida que una persona caminando, entonces el coche viaja 2 x 4 = 8 veces más rápido que la persona. (George F. Simmons)*

Avanzando con el mismo razonamiento, dadas *n* funciones, se puede aplicar repetidamente la regla de la cadena, donde la derivada será:

$$\frac{df_1}{dx} = \frac{df_1}{df_2} \cdot \frac{df_2}{df_3} \cdots \frac{df_n}{dx}$$

Esta regla es útil para el cálculo del gradiente en la red neuronal, ya que el aumento infinitesimal de una variable es lo que se usa para el cálculo del gradiente. Hay que recordar que la derivada de una función para un punto dado, $\frac{df}{dx}$, es la pendiente exacta en ese mismo punto. Si se pudiese calcular la derivada de la función que forma la red neuronal, el gradiente de la función se obtendría automáticamente en vez de tener que emplear las técnicas computacionalmente costosas que se han visto hasta ahora. Sin embargo, el cálculo de la derivada resulta extremadamente complejo en estos casos, por lo que hay que recurrir a técnicas como la regla de la cadena para obtener el mismo resultado.

3.4.3.- Retropropagación

Una vez entendida la regla de la cadena, podemos aplicarla en las redes neuronales para el cálculo del gradiente. Para establecer un marco de partida, se va a ejemplificar una red neuronal extremadamente simple, compuesta por cuatro capas de una neurona cada una, con un peso entre cada una de ellas:

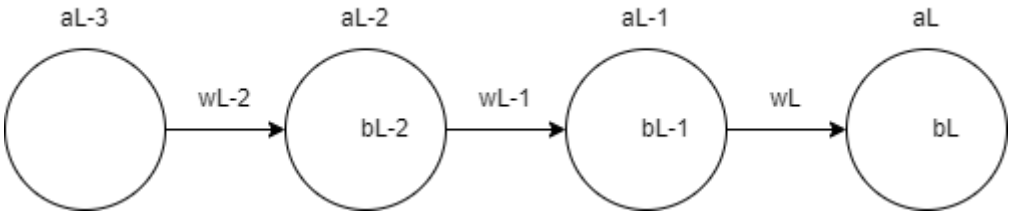

Figura 15: Red neuronal de una sola fila

Esta red está formada por 4 neuronas cada una con su función de activación (*a*), su inclinación (*b*) y un peso que une cada neurona (*w*). Se define una salida esperada para la red que se denominara *y*. La función de coste de la red como:

$$C = (a_L - y)^2$$

La función de activación de una neurona viene definida por su inclinación, el resultado de la función de activación de la neurona anterior y el peso que la une con ella de la siguiente manera:

$$a_L = \sigma(w_L . a_{L-1} + b_L)$$

Donde σ es la función de activación de la neurona. Para simplifica los cálculos se puede agrupar los componentes que se usan en la función de activación como una única variable llamada *z*

$$z_L = w_L . a_{L-1} + b_L$$

$$a_L = \sigma(z_L)$$

Si se quiere calcular la variación en el coste producido de un peso de la red, se requiere calcular la pendiente para ese peso, que se representaba como $\frac{df}{dx}$. Como el resultado de la red es la función de coste *C*, esta derivada se puede definir como $\frac{dC}{dw_L}$. Se puede observar en las definiciones anteriores que C depende de a_L, que a su vez depende de z_L, que a su vez depende de w_L. Aplicando la red de la cadena vista anteriormente se puede inducir la siguiente ecuación:

$$\frac{dC}{dw_L} = \frac{dC}{da_L} \cdot \frac{da_L}{dz_L} \cdot \frac{dz_L}{dw_L}$$

La manera intuitiva de explicar esto es que lo que se busca es calcular cuánto varia la función C con pequeños cambios en *w*. El cambio en *w* causa un cambio en *z* ($\frac{dz_L}{dw_L}$). A su vez un cambio en z produce un camio en *a*. Este cambio en *a* es el que finalmente realiza el cambio de valor en *C*.

El resultado de la multiplicación de esos tres ratios indica la variación de C con respecto a un cambio en *w*, por lo que se puede usar esta técnica para evitar tener que calcular la función de coste para cada peso e inclinación de la red al realizar el descenso de gradiente.

Para poder calcular las ratios, hay que calcular sus derivadas:

$$C = (a_L - y)^2$$

$$\frac{dC}{da_L} = 2(a_L - y)$$

$$a_L = \sigma(z_L)$$

$$\frac{da_L}{dz_L} = \sigma'(z_L)$$

$$z_L = w_L . a_{L-1} + b_L$$

$$\frac{dz_L}{dw_L} = a_{L-1}$$

$$\frac{dC}{dw_L} = 2(a_L - y) \cdot \sigma'(z_L) \cdot a_{L-1}$$

Este es el caso de que se busque la varianza causada por un peso, en el caso de buscar el cambio causado por una inclinación la función general será:

$$\frac{dC}{db_L} = \frac{dC}{da_L} \cdot \frac{da_L}{dz_L} \cdot \frac{dz_L}{db_L}$$

Hay que calcular una derivada diferente para $\frac{dz_L}{db_L}$:

$$z_L = w_L . a_{L-1} + b_L$$

$$\frac{dz_L}{db_L} = 1$$

$$\frac{dC}{db_L} = 2(a_L - y) \cdot \sigma'(z_L)$$

Estos cálculos sirven solo para la neurona en la capa de salida, ya que para las neuronas en capas inferiores se ha de calcular como ese cambio se propaga por las siguientes capas. Un cambio en el peso w_{L-1} causara un cambio en a_{L-1} que a su vez afectara a z_L, agregando cada vez mas elementos a la regla de la cadena. En vista de esto, es importante conocer la derivada de a_{L-1}:

$$z_L = w_L . a_{L-1} + b_L$$

$$\frac{dz_L}{da_{L-1}} = w_L$$

$$\frac{dC}{da_{L-1}} = 2(a_L - y) \cdot \sigma'(z_L) \cdot w_L$$

Con todas las derivadas preparadas, se puede calcular la varianza de la función de coste para cualquier peso o inclinación encadenando la regla de la cadena para obtener el resultado. Como ejemplo, para calcular la varianza causada por el peso w_{L-2}:

$$\frac{dC}{dw_{L-2}} = \frac{dz_{L-2}}{dw_{L-2}} \cdot \frac{da_{L-2}}{dz_{L-2}} \cdot \frac{dz_{L-1}}{da_{L-2}} \cdot \frac{da_{L-1}}{dz_{L-1}} \cdot \frac{dz_L}{da_{L-1}} \cdot \frac{da_L}{dz_L} \cdot \frac{dC}{da_L}$$

$$\frac{dC}{dw_{L-2}} = a_{L-3} \cdot \sigma'(z_{L-2}) \cdot w_{L-1} \cdot \sigma'(z_{L-1}) \cdot w_L \cdot \sigma'(z_L) \cdot 2(a_L - y)$$

El nombre de retropropagación viene dado por la forma de implementar este algoritmo. Si se quiere calcular el cambio de gradiente para los pesos de esta red, se calcularía primero el de la capa de salida y las derivadas parciales obtenidas en esta ($\frac{da_L}{dz_L} \cdot \frac{dC}{da_L}$) se emplean en el calculo del peso en la capa anterior ($\frac{dz_{L-1}}{dw_{L-1}} \cdot \frac{da_{L-1}}{dz_{L-1}} \cdot \frac{dz_L}{da_{L-1}} \cdot \frac{da_L}{dz_L} \cdot \frac{dC}{da_L}$) que a su vez se emplean en el primer peso de la red. De esta manera el calculo del gradiente se va propagando hacia atrás desde la capa de salida.

Todo este proceso es útil cuando la red neuronal solo tiene una neurona por capa, pero ese nunca va a ser el caso en una red de verdad. Hace falta extender la regla de la cadena a una red más compleja, como la siguiente:

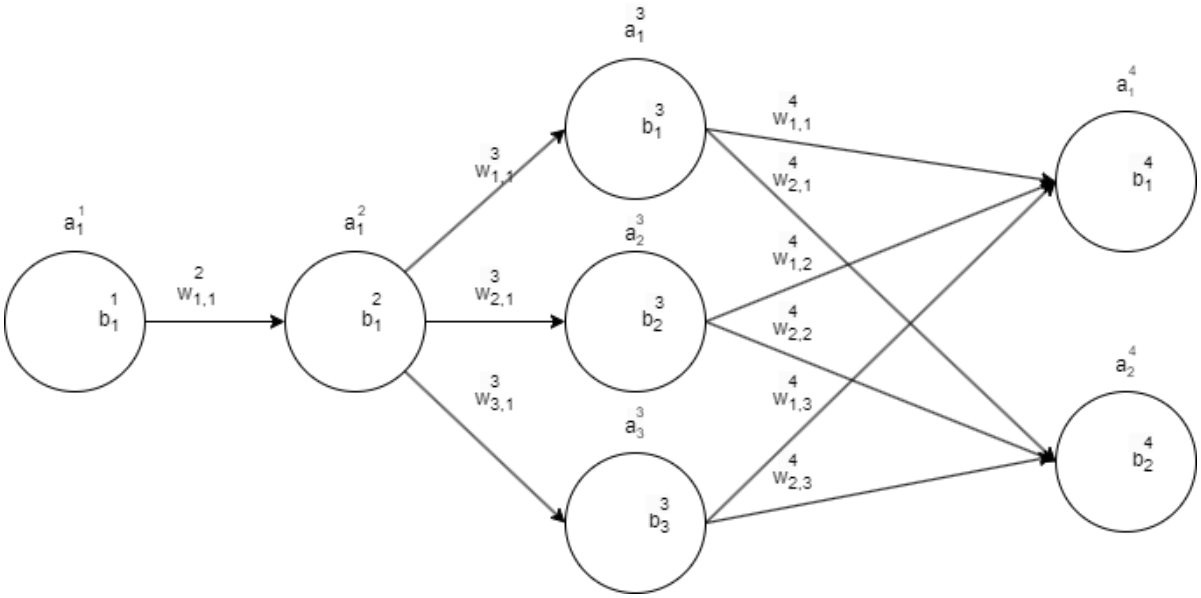

Figura 16: Red neuronal compleja

La nomenclatura de los pesos viene expresada para los pesos de la forma $w_{j,k}^{L}$ donde L es la capa a la que se dirige el peso, j es la posición de la neurona a la que se dirige el peso y k es la posición de la neurona de donde proviene el peso. De esta forma, $w_{2,1}^{4}$ es el peso que conecta la neurona 1 de la capa 3 con la neurona 2 de la capa 4,

De manera similar, para la función de activación a y las inclinaciones b, la nomenclatura se expresa de la forma b_j^L donde L es la capa de la que forman parte y j la neurona a la que hacen referencia.

El cálculo de la variación de la función de coste para un peso como $w_{2,1}^3$ es análogo al visto anteriormente, se ha de observar como se encadenan los cambios en la función de coste que se ven afectados por el cambio de este peso.

$w_{2,1}^3$ afecta a z_2^3 que afecta a a_2^3. En este punto el calculo se diversifica, ya que a_2^3 afecta tanto a z_1^4 como a z_2^4. Primero hay que observar como se realiza el cálculo de la función de coste, ya que al tener múltiples neuronas de salida el cálculo es diferente. Como se ha visto en el apartado Función de coste Cuadrática, esta se puede expresar como el sumatorio de la diferencia cuadrática entre el resultado esperado y el obtenido de cada neurona en la capa de salida, lo cual expresado matemáticamente se puede indicar de la siguiente manera:

$$C = \sum_{j=0}^{n_L-1} \left(a_j^{(L)} - y_j\right)^2$$

De esto se puede extraer que para un cambio infinitesimal en a_k^{L-1} el cambio que se produce en la función de coste es

$$\frac{dC}{a_k^{L-1}} = \sum_{j=0}^{n_L-1} \left(\frac{dz_j^L}{da_k^{L-1}} \cdot \frac{da_j^L}{dz_j^L} \cdot \frac{dC}{da_j^L}\right)^2$$

Por lo tanto, para el cambio que se estaba explicando en a_2^3, se produce el sumatorio de los cambios en z_1^4 y z_2^4, los cuales producen cambios a_1^4 y a_2^4 que al final producen el cambio en C. Siguiendo la regla de la cadena toda esta sucesión de efectos se puede expresar como

$$\frac{dC}{dw_{2,1}^3} = \frac{dz_2^3}{dw_{2,1}^3} \cdot \frac{da_2^3}{dz_2^3} \cdot \sum_{j=0}^{1} \left(\frac{dz_j^4}{da_2^3} \cdot \frac{da_j^4}{dz_j^4} \cdot \frac{dC}{da_j^4}\right)^2$$

$$\frac{dC}{dw_{2,1}^3} = a_1^2 \cdot \sigma'(z_2^3) \cdot \sum_{j=0}^{1} \left(w_j^4 \cdot \sigma'\left(dz_j^4\right) \cdot 2(a_j^4 - y_j)^2\right)$$

El único elemento faltante para poder utilizar la retropropagación en una red neuronal es calcular la derivada de la función de activación. Si la red usa una función de activación sigmoide, se obtendrá la derivada de esta función:

$$\sigma(x) = \frac{1}{1 + e^{x}}$$

$$\sigma'(x) = \frac{e^{-x}}{(1 + e^{-x})^2}$$

$$\sigma'(x) = \sigma\big(1 - \sigma(x)\big)$$

Estando ya definidas todas las operaciones matemáticas necesarias, se puede realizar la implementación de la retropropagación, tal y como se muestra en la octava escena de la aplicación interactiva.

Aunque la escena sea igual a la anterior, se puede apreciar una mejora importante de tiempo en el procesamiento de la red. En las pruebas realizadas, incluso con una red tan pequeña en la que el efecto no debería ser muy pronunciado, el tiempo para procesar 1000 iteraciones de aprendizaje se ha reducido de 4 segundos a 1,5. Al realizar pruebas con una red de tamaño [2, 30, 30, 2] para el mismo escenario, el tiempo con retropropagación para 1000 iteraciones es de 9 segundos, mientras que sin la mejora el tiempo de procesamiento ha superado los 20 minutos.

Con la base explicada en el desarrollo de redes neuronales, se puede implementar redes complejas que permiten solventar problemas complejos, como el reconocimiento de escritura a mano. Sin embargo, todavía existen múltiples mejoras que se pueden realizar al algoritmo de aprendizaje para mejorar el desempeño de la red, las cuales pueden verse en el anexo 8.1.- Mejoras en el aprendizaje de una red neuronal.

3.4.4.- Aprendizaje por refuerzo

El aprendizaje por refuerzo es el problema al que se enfrenta un agente que debe aprender un comportamiento mediante prueba y error en un entorno dinámico. Esta técnica no necesita ejemplos de entrada ni salida para poder funcionar, si no que se centra en encontrar un balance entre la exploración de un entorno desconocido y la explotación del conocimiento adquirido con el objetivo de maximizar una recompensa en el largo plazo, sin una retroalimentación instantánea [24].

Existen dos estrategias principales para la resolución de problemas basados en aprendizaje por refuerzo:

- Buscar dentro de un espacio de comportamientos uno que se adecuen con corrección al entorno, generalmente mediante el uso de algoritmos y programación genéticos.

- Usar técnicas estadísticas y programación dinámica para estimar el valor de la utilidad que tomar una acción determinada dentro del entorno tendrá en el futuro.

En el modelo estándar del aprendizaje por refuerzo, un agente realiza interactúa con un entorno en intervalo de tiempo discretos. En cada instante de tiempo el agente recibe información sobre el estado del entorno, entonces elige una acción que provoca un cambio en este. El valor de este cambio se comunica al agente mediante una señal de refuerzo. El trabajo del agente es maximizar la señal de refuerzo que obtiene.

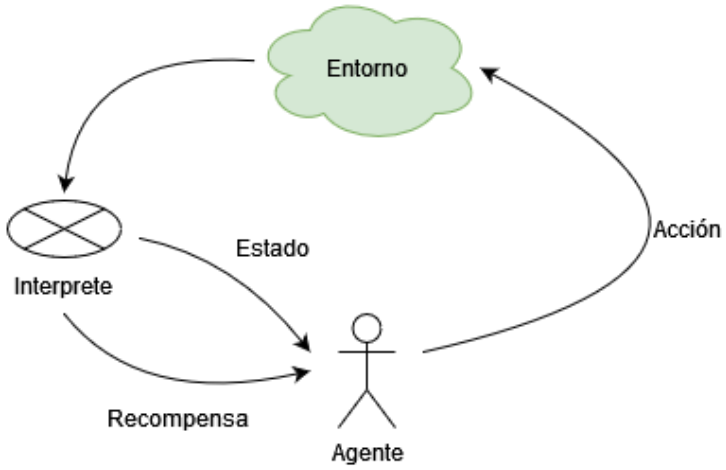

Figura 17: Modelo estándar de aprendizaje por refuerzo

Con el objetivo de poder desarrollar personajes inteligentes en un entorno dinámico como son los videojuegos, donde la recompensa no siempre es inmediata, se explicarán, desarrollarán e implementara el primer tipo de estrategias visto para el aprendizaje por refuerzo, los algoritmos genéticos.

3.5.- Algoritmos genéticos

Los algoritmos genéticos son un modelo computacional de la evolución biológica, inspirados en las ideas de la genética de poblaciones. En ellos una población de individuos se crea aleatoriamente. Cada individuo está formado por un conjunto de valores (su genotipo) que es candidato a ser la solución de un problema. Las diferencias entre cada individuo resultan en individuos más capacitados que otros en

la resolución del problema, por lo que esas diferencias son usadas para elegir un nuevo conjunto de individuos para la siguiente interacción del algoritmo, creándose a partir de los individuos más exitosos de la iteración anterior, pero pudiendo agregar ligeras alteraciones como mutaciones, recombinaciones u otras alteraciones durante la copia de los valores [25].

La nueva generación de individuos, al descender de los individuos más aptos de la generación anterior, tiende a ser de media más apta para la resolución del problema. Mediante este ciclo de evaluación, selección y operaciones genéticas durante muchas generaciones resultan en una mejora de todos los individuos de la población, siendo candidatos más aptos para la resolución del problema.

Se necesitan cinco componentes clave para el desarrollo de estos algoritmos:

- Una forma de codificar soluciones al problema como genotipos
- Una función de evaluación que devuelve una puntuación a cada genotipo que la emplee.
- Una forma de inicializar los genotipos.
- Operadores que se apliquen en los individuos cuando producen la siguiente generación para alterar sus genotipos.
- Ajustes de parámetros para los operadores y el algoritmo.

El algoritmo se ejecuta de la siguiente manera:

1. Se inicializa la población como un nuevo conjunto de genotipos.
2. Se evalúa cada miembro de la población.
3. La población se reproduce hasta cumplir un criterio de parada. Esta reproducción puede consistir en varias iteraciones de los siguientes pasos:
    a. Uno o más padres son escogidos para la reproducción de manera estocástica priorizando a los padres con mejores evaluaciones.
    b. Se aplican operaciones genéticas siguiendo ciertos parámetros.

c. Los hijos se evalúan y son insertados en la población. Dependiendo del algoritmo empleado, la población es remplazada total o parcialmente por los nuevos hijos.

Si se emplea una representación que codifique el genotipo correctamente para adecuarse al problema, contiene una buena función de evaluación y usa operadores que generan mejores hijos a partir de sus padres, el algoritmo genere rara constantemente mejores y mejores individuos, llegando a acercarse al máximo global al maximizar la función de recompensa. El uso de mutaciones y operaciones genéticas permiten esquivar con gran facilidad los máximos locales de la función.

3.5.1.- Operadores genéticos

Los operadores genéticos son operaciones usadas para guiar a un algoritmo hasta la solución de un problema. Existen tres tipos principales, mutación, recombinación y selección, que se usan en conjunto para lograr el éxito. Estos operadores genéticos se usan para crear y mantener diversidad genética, mezclando soluciones tanto existentes como nuevas y seleccionando las mejores.

Algunos de los operadores genéticos existentes que se usaran en el resto del proyecto son:

Mutación imparcial de pesos

Cada variable de el genotipo será intercambiada con una probabilidad fija *p* por un valor de la distribución inicial de valores aleatorios.

Mutación parcial de pesos

A cada variable de el genotipo le será añadida con una probabilidad fija *p* un valor de la distribución inicial de valores aleatorios. Suele ser mejor que la mutación imparcial debido a que las variables de un genotipo entrenado suelen ser mejores que las de uno no entrenado, por lo que alterarlas ligeramente en vez de reiniciar su valor debería ofrecer mejores resultados.

Mutar nodos

La operación selecciona una cantidad determinada de nodos (que no sean de entrada). Para cada entrada de cada uno de esos nodos se añade un valor de la distribución inicial de valores aleatorios. Esta nueva red se codifica en el genotipo hijo.

Mutar nodos débiles

A diferencia del coste o el error de un nodo en la retropropagación, la fuerza de un nodo identifica lo relevante que es el nodo para la red. Esta se define como la diferencia en el valor de la red al evaluarla de manera normal con respecto a evaluarla con el nodo "lobomotizado" (es decir, con todos sus valores de salida siendo 0).

Al utilizar esta operación, se calcula la fuerza de todos los nodos de una red padre y se mutan una cantidad *m* de los nodos más débiles, de manera parcial si la fuerza del nodo es positiva y de manera imparcial si es negativa, de manera que los nodos que contribuyen más sufren mutaciones pequeñas y los que perjudican a la red son reseteados.

Recombinar pesos

Al generar un genotipo hijo de dos genotipos padres, cada una de las variables de este se escoge aleatoriamente del genotipo de uno de los dos padres, creando una mezcla de ambos.

Recombinar nodos

Al generar un genotipo hijo de dos genotipos padres, cada uno de los nodos de este se escoge aleatoriamente de uno de los dos padres, copiando cada una de las variables asociadas a él y manteniendo la lógica de cada nodo.

Recombinar características

Diferentes nodos de una red cumplen diferentes funciones, dependiendo de la capa en la que se encuentren y no de su posición en la capa (se pueden intercambiar todas las variables entre dos nodos de la misma capa y el resultado de la red será el mismo antes y después del intercambio).

Los individuos generados a partir de las recombinaciones descritas anteriormente son afectados en gran medida por la estructura interna de sus padres. La recombinación de características disminuye este efecto buscando por cada nodo en la red del primer padre, un nodo que cumpla una función similar en el segundo padre, mostrando una cantidad de entradas a ambas redes y comparando las respuestas de los diferentes nodos. Tras eso, se reorganiza la red del segundo padre de tal manera que los nodos que cumplen funciones similares estén en la misma posición, formando un hijo de forma similar a la recombinación de nodos.

Escalada simple

Se calcula el gradiente de cada miembro del set de datos y se suman para obtener un gradiente total. Después, se normaliza el gradiente dividiéndolo entre su magnitud. El genotipo hijo se obtiene del padre al dar un paso en la dirección determinada por el gradiente normalizado. El valor del paso es un parámetro adaptativo, que se multiplica por un valor fijo inferior a 1 si la evaluación del hijo es mejor que la del padre y por un valor fijo superior a 1 si es peor.

Este operador se diferencia de la retropropagación en que los pesos se ajustan solo tras haber calculado el gradiente de todos los individuos del conjunto y en que el gradiente esta normalizado de forma que el paso no sea proporcional a su tamaño.

3.5.2.- Parámetros de un algoritmo genético

Existen una diversidad de parámetros cuyos valores alteran en gran medida el funcionamiento de un algoritmo genético, que se deberían mantener fijos a lo largo de todo el algoritmo:

- **Escala del padre (*p*):** Determina con que probabilidad un individuo es escogió como padre. El mejor individuo tiene una probabilidad *p* de ser escogido. El segundo mejor tiene una probabilidad *p*p* de serlo (ya que es la probabilidad de no haber escogido al primero en conjunto a la de ser escogido el). El tercero tendrá una probabilidad *p*p*p* y así sucesivamente.

- **Probabilidad de operadores:** Esta lista de parámetros determina con que probabilidad cada uno de los operadores es escogido. Se puede empezar con una probabilidad igual para todos los operadores y usar mecanismos adaptativos para cambiar sus probabilidades a lo largo del algoritmo en función del resultado que tiene cada uno en la red.

- **Tamaño de la población:** La cantidad de individuos que conforman la población. La población se puede dividir en diferentes conjuntos que se evalúan individualmente para facilitar el procesamiento de la simulación, y los padres de la nueva generación se escogerán entre los mejores individuos de todos los paquetes de una misma generación.

3.5.3.- Implementación de un algoritmo genético

En la novena escena de la aplicación interactiva, carrera genética se puede observar un circuito de carreras donde se pondrán a prueba los algoritmos genéticos. El objetivo es que, sin escribir ninguna

instrucción sobre el control de un vehículo, un coche controlado por IA sea capaz de superar el circuito lo más rápido posible.

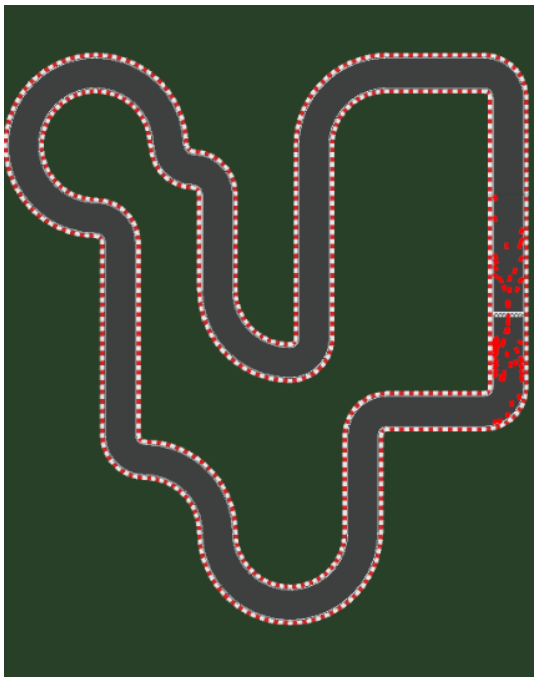

Figura 18: Circuito usado para el aprendizaje mediante algoritmos genéticos

Esto es posible con los algoritmos genéticos que se han explicado con anterioridad, la ejecución se desarrollara mediante generaciones de una cantidad determinadas de vehículos que hacen uso de una red neuronal. Esta red neuronal está formada por 6 neuronas de entrada, cuyos valores vienen dados por la velocidad del vehículo y la distancia hasta los límites del circuito en ángulos determinados, 5 neuronas en una capa oculta y 2 neuronas en la capa de salida, que definirán la aceleración y el giro del vehículo, haciendo uso estas dos últimas neuronas de una función de activación tangente hiperbólica, la cual tiene la siguiente forma:

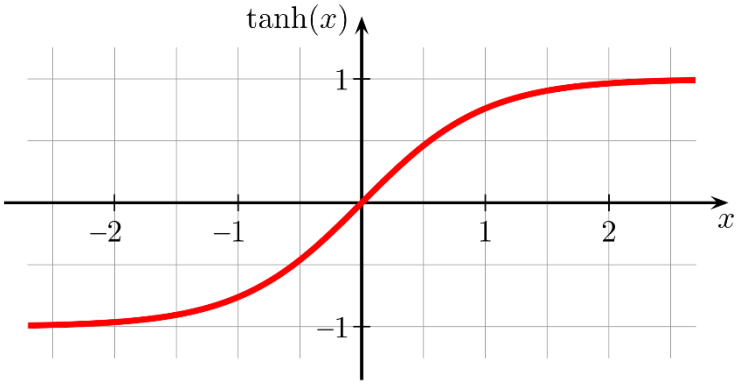

Figura 19: Función tangente hiperbólica

Esta función es muy similar a la función sigmoide, pero en vez de estar comprendida entre el rango [0, 1], lo está en el rango [-1, 1]. Esto facilita mucho el control del vehículo, permitiendo definir aceleración y giro negativos (es decir, si el valor de la función es positivo el vehículo acelerara hacia adelante/girara a la derecha y si es negativo acelerara hacia atrás / girara a la izquierda).

Tras evaluar todos los vehículos de una generación, estos se ordenan según su puntuación y para cada dos vehículos de la nueva generación se escogen dos padres de la original (priorizando los más aptos), se recombinan los genomas de los dos padres aleatoriamente entre los dos hijos y se muta cada uno de los hijos tras ello.

Al terminar de reproducir la nueva generación, a cada vehículo se le asigna un genoma de la nueva generación y se repite el proceso. Con los parámetros establecidos en la simulación, la IA aprende a girar a la derecha sobre la quinta generación, y aprende a girar a la izquierda entre la generación 30 y la 50. Tras aprender a girar en ambas direcciones suele aprender a completar el circuito entero en un par de generaciones. Este proceso es aleatorio y el resultado no es seguro, pudiendo llegar a mínimos locales que frenan o incluso paran el aprendizaje de la población. Los algoritmos genéticos constan de técnicas para afrontar estos problemas que no serán discutidas en esta sección.

Para saber la puntuación de cada vehículo en todo el circuito hay puntos de control que le asignan un punto a los vehículos que pasan a través de ellos. Adicionalmente, se añade un pequeño bonificador en cuanto más cerca se encuentre el vehículo del siguiente punto de control.

4.- ANALISIS Y DISEÑO

<table>
<tr><th colspan="2">GENETIC TOWER
Documento de diseño</th></tr>
<tr><td>Género</td><td rowspan="2">Unidades
Esbirro: Barato, resistente y rápido pero cuenta con poco daño.</td></tr>
<tr><td>Defensa de torre</td></tr>
<tr><td>Plataforma</td><td rowspan="2">Arquero: Barato, cuenta con un ataque a distancia.</td></tr>
<tr><td>Web/HTML5</td></tr>
<tr><td>Descripción</td><td>Farolero: Coste medio, ataque devastador pero lento, muy poca vida.</td></tr>
<tr><td rowspan="3">Trata de defender tu torre y destruir la de tu rival invocando poderosas tropas, mejorando edificios y desarrollando una gran estrategia. Tu rival será una red neuronal genética, que ha aprendido a jugar sin la intervención ni ayuda de ningún humano. ¿Serás capaz de aprender el juego mejor que ella?</td><td>Dinamitero: Daño extra a torres, ataque en área a distancia.</td></tr>
<tr><td>Caballero: Alta resistencia y daño, pero coste muy elevado</td></tr>
<tr><td>Arte</td><td>Edificios</td></tr>
<tr><td>PixelArt (TinySwords)</td><td rowspan="2">Mina: Genera oro cada cierto tiempo, puede ser mejorada</td></tr>
<tr><td>Características</td></tr>
<tr><td>Cuatro niveles de dificultad</td><td>Torre: Cuenta con mucha vida, al ser destruida acaba el juego</td></tr>
<tr><td rowspan="2">IA enemiga generada por una red neuronal genética</td><td>Tecnologías</td></tr>
<tr><td>Motor de juego: Unity</td></tr>
<tr><td rowspan="2">5 tipos diferentes de unidades con ataques variados</td><td>Lenguaje del código: C#</td></tr>
<tr><td rowspan="2">Código fuente redes neuronales: Desarrollo propio.</td></tr>
<tr><td>Interfaz</td></tr>
<tr><td>Botones de dificultad: Seleccionar la dificultad del juego.</td><td rowspan="2">Estructura redes neuronales: {15, 40, 30, 7}.</td></tr>
<tr><td>Botones de unidades: Invoca la unidad correspondiente si consta del oro suficiente.</td><td>Tipo de neuronas: Sigmoides</td></tr>
<tr><td>Botón de mina: Aumenta el oro obtenido por segundo si se consta del oro suficiente.</td><td>Tipo de red: Genética</td></tr>
<tr><td>Botón de reinicio: Una vez destruida una torre, pulsa para volver a seleccionar dificultad</td><td>Tipo de entrenamiento: No supervisado, selección natural al enfrentarse a otras inteligencias artificiales.</td></tr>
</table>

Figura 20: Documento de diseño del juego

4.1.- Diseño del juego.

La primera parte para poder desarrollar una inteligencia artificial para un videojuego es tener el videojuego. Para facilitar el aprendizaje y poder estudiar mejor las capacidades de la IA el juego a diseñar será un juego de uno contra uno, donde el jugador se enfrente a la red para ganar, en un juego simétrico, donde los dos jugadores tienen las mismas opciones y capacidades, y que sea sencillo para facilitar su implementación.

Teniendo en cuenta estos factores se ha decidido implementar un juego sencillo de defensa de torres, estilo *Clash Royale*, donde cada jugador toma el control de un bando, el lado azul o el lado rojo, y tendrá una torre que defender, debiendo destruir la del enemigo mediante la invocación de tropas para ganar.

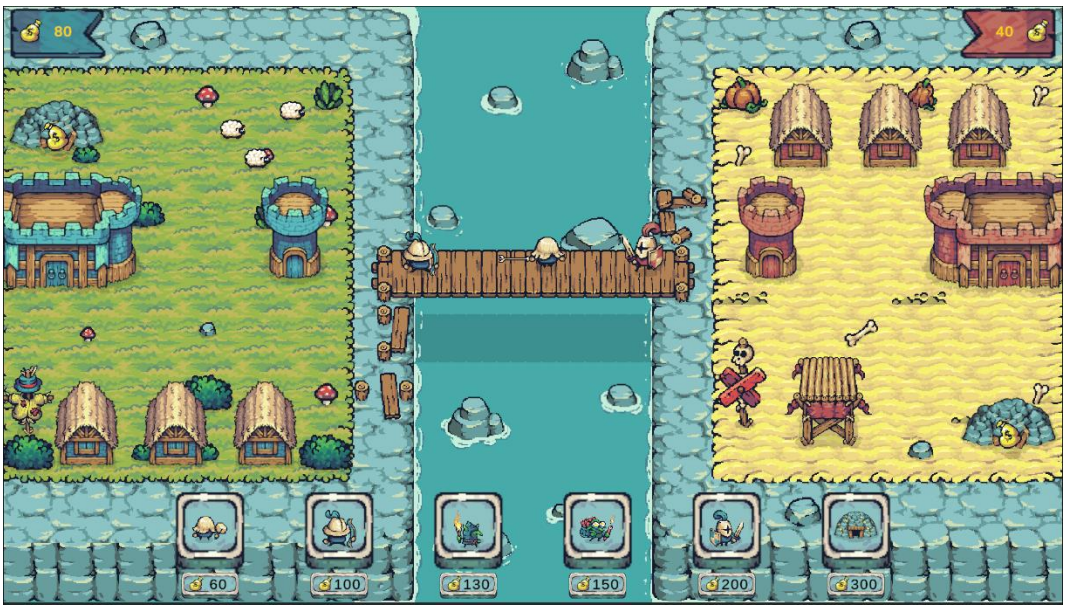

Figura 21: Captura del juego

En el juego se ira generando oro cada cierto tiempo y los jugadores pueden comprar 5 tropas distintas con ese oro o mejorar la mina. Cuando una de las torres sea destruida el juego terminara.

El arte del juego se ha obtenido del paquete de gráficos gratuito "*Tiny Swords*"

4.1.1.- Unidades

En el juego se pueden encontrar 5 unidades diferentes que el jugador puede emplear para hacer frente a su enemigo:

Esbirro

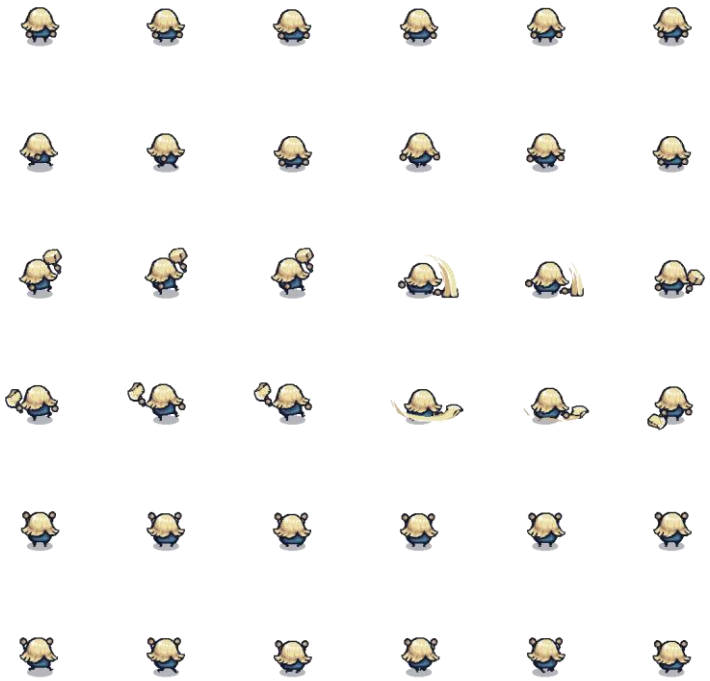

Figura 22: Animaciones del esbirro

Los esbirros son la unidad más barata del juego, siendo fácil generar muchos de ellos para desbordar al enemigo. Tienen un ataque débil pero rápido y una cantidad de vida básica decente, lo que los convierte en una buena opción en general.

Arquero

Figura 23: Animaciones del arquero

Los arqueros son la segunda unidad más barata del juego, tienen una vida reducida y un ataque débil, pero cuentan con la habilidad de disparar flechas a distancia, sin exponerse a un contraataque. Por el lado negativo, los arqueros no se mueven mientras tengan enemigos en rango, lo que provoca que aliados que se encuentren detrás del arquero no puedan avanzar hasta que el arquero sea derrotado.

Farolero

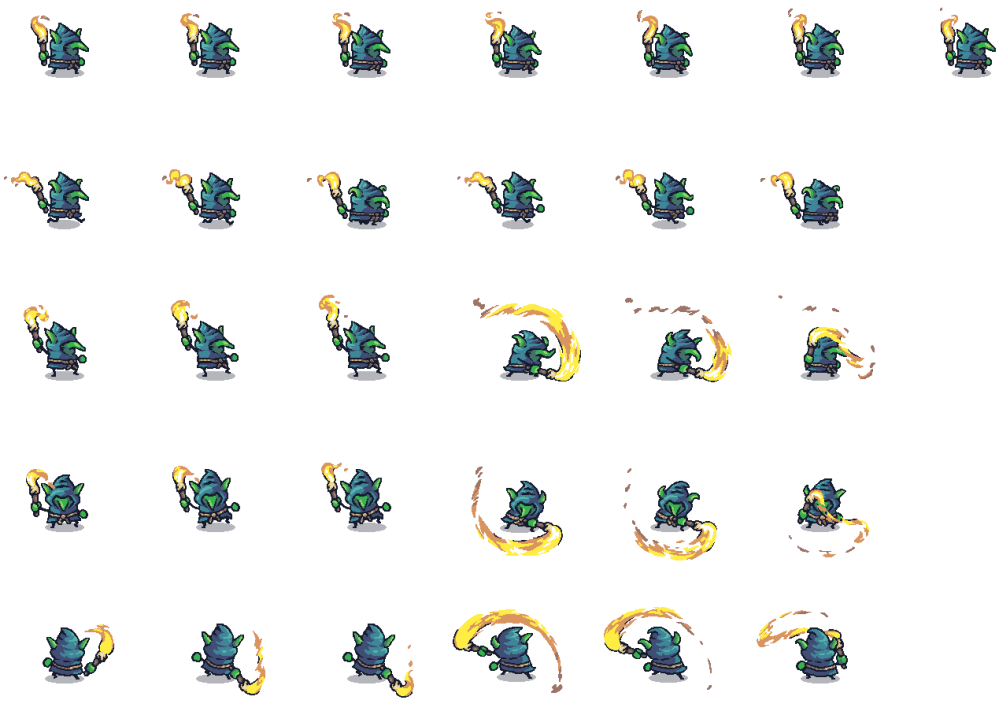

Figura 24: Animaciones del farolero

Los faroleros son la unidad con menos vida del juego, siendo derrotados por un único ataque de cualquier unidad menos los esbirros. Sin embargo, también tienen un ataque extremadamente potente, pudiendo derrotar a casi todas las unidades del juego de un solo golpe, además de ser capaces de destruir las torres en dos ataques.

Los faroleros son un arma de doble filo, siendo capaces de acabar rápidamente con la partida si llegan a la torre rival, pero pudiendo ser derrotados con gran facilidad.

Dinamitero

Figura 25: Animaciones del dinamitero

El dinamitero es una unidad a distancia que tiene la habilidad única de realizar un ataque a distancia que provoca daño en área, pudiendo atacar a varias unidades a la vez, su ataque no es muy fuerte y es lento, pero además de poder dañar múltiples enemigos simultáneamente, estos causan un daño doble a las torretas, siendo muy útiles para dañar las torres contra una defensa cerrada.

Su rango para atacar es inferior para el arquero y es la segunda tropa más cara del juego, por lo que usarlos, al igual que los faroleros, es complicado, pero puede conllevar una gran ventaja en el momento adecuado.

Caballero

Figura 26: Animaciones del caballero

La última unidad y la más cara del juego es el caballero. Tienen una vida extremadamente alta y un daño elevado, permitiendo derrotar fácilmente a todas las unidades del juego. Sin embargo, los faroleros pueden derrotar fácilmente a los caballeros, por lo que hay que saber emplearlos bien para asegurar su eficacia y no malgastar la inversión en oro que supone.

4.1.2.- Edificios

El juego cuenta con dos edificios diferentes que servirán para condicionar la victoria de los jugadores.

Mina

Figura 27: Mina en su máximo nivel

La mina es un edificio que genera oro cada cierto tiempo para cada jugador. Por un alto coste de oro, la mina se puede mejorar para aumentar la ganancia de oro. Esta mejora se puede realizar dos veces, aumento el coste en cada una.

Torre

Figura 28: Torre azul

La torre es la base de cada jugador, es el sitio desde el cual generan las unidades además de que su destrucción supone la derrota del jugador.

4.1.3.- Dificultad

Al empezar el juego se puede seleccionar 4 dificultades diferentes contra las que el jugador puede enfrentarse. Estas dificultades son 4 redes neuronales diferentes que han sido entrenadas las unas contra las otras para lograr que aprendan a jugar.

Tras el entrenamiento se ha analizado la efectividad de cada red manualmente y asociado respectivamente el nivel de dificultad correspondiente.

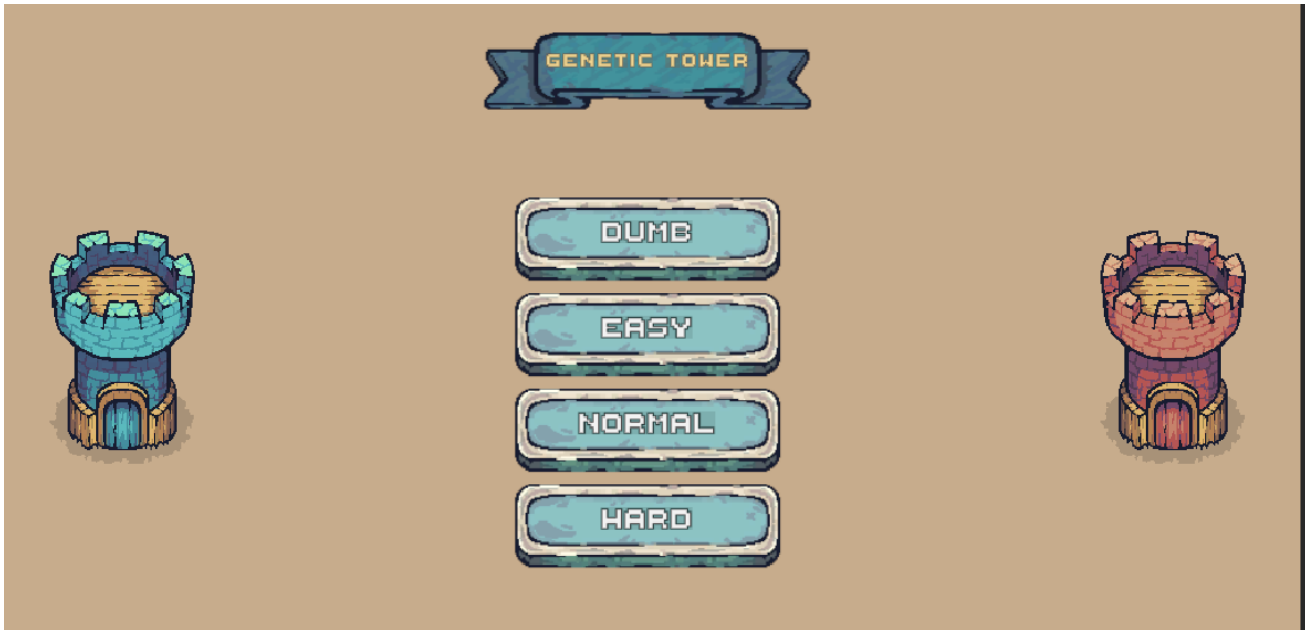

Figura 29: Selección de dificultad

Al seleccionar una dificultad empezará una partida contra la IA correspondiente, y tras la destrucción de una torre se volverá a la pantalla de selección de dificultad.

4.2.- Análisis de la implementación de una red neuronal.

Existen multitud de posibles formas de afrontar el diseño de una Inteligencia Artificial que controle uno de los bandos en el juego. Esta IA debe ser capaz de:

- Analizar el estado del juego
- Realizar acciones en base al estado para favorecer el estado a su favor.
- Ganar la partida gracias a las acciones realizadas.

Estas directrices son bastante generales y se pueden realizar con diversos métodos, como máquinas de estados, arboles de decisiones, *Q-Learning* o, el sistema que se va a implementar, redes neuronales.

Las ventajas que nos ofrece esta elección es una adecuación a las necesidades de la red, ya que, de manera general, las redes neuronales cuentan con una capa de entrada que se puede usar para comunicarle el estado de la partida y una capa de salida que ofrece una respuesta la situación del juego, por lo que, en su diseño más básico, cumple con las capacidades que se buscan en la inteligencia artificial a desarrollar.

El punto interesante del uso de este tipo de sistemas es el cómo se constituyen, ya que en vez de diseñar el cómo se enfrenta la IA al jugador, se diseña como la IA aprende a enfrentarse al jugador. Esta diferenciación es muy importante, ya que de esta manera se pueden lograr comportamientos óptimos que constituyan un alto nivel de dificultad sin necesidad de estar limitado al conocimiento o ideas de las personas que se encarguen del diseño de la inteligencia artificial. Para este caso, las capas intermedias de la red neuronal son una caja negra para el programador. No se sabe el cómo está constituida internamente, pero el juego será capaz de usarla a través de la entrada y salida de datos de ella.

El punto negativo de este sistema es que diseñar un entrenamiento correcto capaz de hacer que la IA aprenda a jugar con buen nivel es en muchos casos más complicados que diseñar directamente la inteligencia, además de requerir adaptaciones en el código para ser capaces de realizar el entrenamiento (siendo necesario en muchos casos ser capaces de ejecutar múltiples instancias simultáneamente), además del tiempo de ejecución que necesita el propio entrenamiento, que puede ir desde minutos hasta días dependiendo de la complejidad. No solo eso, es complicado saber si el modelo de

entrenamiento es el adecuado hasta el fin de dicho entrenamiento, por lo que si el diseño no es bueno desde el principio requiere mucho tiempo adaptarlo hasta encontrar uno que funcione correctamente.

Por lo que se ha visto, el uso de una red neuronal es un arma de doble filo, permite el diseño de estrategias y complejidades que no requieren de que un humano piense en ellas, pero conseguir que llegue hasta ellas puede ser terriblemente costoso.

4.2.1.- Selección del modelo de red neuronal

Existen una gran variedad de modelos de redes neuronales que pueden ser usadas para implementar el comportamiento de una inteligencia rival. Uno de los motivos para la elección de una red neuronal es disminuir la intervención de personas en el diseño de la inteligencia, por lo que se busca implementar un modelo de aprendizaje no supervisado.

Un tipo de modelo a considerar es el aprendizaje por copia, en la que un jugador realizar varias partidas de ejemplo para que la IA tome estas partidas de referencia y forme una base desde la que partir para generar nuevas estrategias. Sin embargo, en este trabajo se busca ver los límites de la IA y lo que puede lograr por su propia cuenta, por lo que el modelo que más se ajusta a esta necesidad es una red genética, que sea capaz de aprender desde 0 todo lo necesario para generar estrategias complejas sin necesidad de ningún tipo de intervención humana.

Para maximizar la variedad genética y permitir que realmente el aprendizaje no necesite intervención humana, para implementar este aprendizaje se crearan cuatro redes totalmente independientes que se enfrentaran entre ellas para lograr generar mejores estrategias. Además, de esta manera, se pueden clasificar las diferentes redes una vez finalizado el entrenamiento para establecer diferentes niveles de dificultad.

4.2.2.- Diseño de las entradas de la red neuronal

La red neuronal debe ser capaz de conocer el estado del juego, pero en un contexto dinámico donde la combinación de todas las variables del juego resulta en un conjunto casi infinito de posibilidades, dicha tarea es compleja. Para conocer el estado real del juego la red debe conocer:

- La cantidad de oro que tiene.
- La cantidad de oro que genera.

- La vida de cada torre
- Todas las tropas existentes en el campo, su tipo y su posición.

Los 3 primeros datos son sencillos de obtener y modelan 6 nodos en la capa de entrada (oro aliado, oro enemigo, nivel de la mina aliada, nivel de la mina enemigo, vida aliada, vida enemiga). El ultimo campo se torna difícil de modelar al tener en cuenta que cada nodo de entrada se modeliza con un único valor numérico y la cantidad de nodos de entrada debe estar definido desde el comienzo.

Se han planteado dos modelos diferentes para poder mostrar la información de las tropas a la red perdiendo la mínima información posible mientras se mantienen limitadas la cantidad de neuronas que forman la capa de entrada.

Codificación de enemigos

Este concepto consiste en codificar únicamente un número limitado de unidades *x* de cada jugador, siendo estas las más avanzadas en el terreno de cada lado.

Existe una neurona que indica la posición de la tropa más avanzada, además de *x* neuronas que indican en orden el tipo de cada tropa. De esta manera la red poseerá información de cuáles son las primeras *x* unidades de cada jugador en orden y sabrá donde está posicionado el frente de batalla de cada jugador.

Para codificar cada tipo de tropa en la neurona de entrada, se le asigna un valor numérico dependiendo de su tipo, de manera que, por ejemplo, si es un caballero el valor de la neurona sea 1, si es un esbirro el valor será 0.2 y si no hay unidades el valor será 0.

Se pueden hacer pruebas para establecer el valor *x* de tropas que la red tiene en cuenta, por lo cual la red tiene acceso a casi toda la información del juego, perdiendo solo datos sobre la posición especifica de cada unidad después de la primera y no conociendo tropas más allá de cierto limite, las cuales al estar más alejadas del frente de la batalla no tendrían un efecto significativo.

Una neurona para cada tipo de enemigo

Para facilitar que la red ponga una mayor prioridad en el tipo de tropas a las que se enfrenta, lo cual es el factor más importante a la hora de tomar decisiones, se plantea que cada tipo de tropa tenga una neurona asociada, la cual indica la posición de la primera unidad de ese tipo en el campo de batalla.

Esto facilita mucho el trabajo para la red, que no tiene que aprender a descodificar valores y asociarlos a cada tipo de tropa. Un campo adicional requerido para que la red funcione adecuadamente es la cantidad de tropas que tiene cada jugador en juego.

Este modelo es el que menos información otorga a la red, solo conoce la posición y tipo de tantas unidades como tropas diferentes haya, por lo que no puede diferenciar entre, por ejemplo, un pelotón enemigo formado por dos caballeros y un esbirro y otro pelotón formado por un caballero y dos esbirros.

Pese a ofrecer una menor cantidad de información, este modelo está planteado teniendo en cuenta que la red funciona mediante la intensidad de cada neurona, por lo que le es mucho más fácil reaccionar a cada tipo de unidad dependiendo de su distancia a la torre, ya que es la propia intensidad de la neurona la que le indica dicha información, sin necesidad de decodificar información lo que necesitaría un trabajo adicional.

4.2.3.- Diseño de las salidas de la red neuronal

La capa de salida de la red neuronal es bastante directa de diseñar, contará con una neurona por cada tipo de unidad, una neurona para la mina y una última para no realizar ninguna acción.

De esta manera, tras procesar las entradas, la neurona con el valor más alto será la que indique que acción realizar, si está asociada a una tropa invocar esa tropa, si es la neurona de la mina mejorar la mina y si es la última neurona no realizar ninguna acción.

4.3.- Diseño del aprendizaje

Una vez definida la red neuronal, se debe establecer como los pesos de las neuronas de la red serán modificados para lograr obtener el resultado deseado. Como se ha establecido anterior, se hará uso de una red evolutiva para el aprendizaje, haciendo uso de la implementación que se ha hecho de ella en el apartado 3.5.- Algoritmos genéticos.

Sin embargo, en esta ocasión, la red ha de enfrentarse a un rival con inteligencia que puede reaccionar de maneras diferentes a una misma situación en momentos diferentes. Esto dificulta enormemente el aprendizaje de la red, ya que no hay un marco fijo para establecer lo buenas o malas que son las acciones de una red respecto a las de otras redes. Es difícil decidir entre dos redes que han ganado una partida cuál de las dos es mejor.

4.3.1.- Organización de múltiples redes.

Como se ha mencionado con anterioridad, se hará uso de diferentes redes neuronales independientes para el aprendizaje. Esto tiene una multitud de ventajas:

- Aunque una red no demuestre mejoría, se atasque en una estrategia determinada o simplemente no aprende correctamente, existen otras redes que aprenden de manera simultánea y existen más posibilidades de que alguna de ellas obtenga buenos resultados.

- Asegura variedad de comportamientos, lo cual no solo es útil para determinar diferentes niveles de dificultad una vez terminado el entrenamiento, si no que será útil en el aprendizaje al poder competir contra diferentes estrategias, asegurando un mejor resultado general.

- Permiten establecer un entorno de aprendizaje en el que no son necesarios agentes externos. Las redes pueden entrenar solo compitiendo entre ellas y obteniendo las mejores de cada generación.

Para que todas las redes tengan las mismas oportunidades y aprendan simultáneamente, el entrenamiento se organizara en tantos lotes como diferentes redes haya. En cada lote entrenaran una cantidad igual de miembros de cada tipo de red, enfrentándose a una misma inteligencia rival. Esta inteligencia se corresponderá con el mejor miembro de la generación anterior de la red que corresponda con el lote, es decir, en el primer lote todas las inteligencias se enfrentan al mejor miembro en la

generación anterior de la primera red, en el segundo lote el mejor miembro de la segunda red y así sucesivamente.

Cuando termine una generación cada miembro de cada red se habrá enfrentado al mejor miembro de todas las demás, incluyendo la suya misma.

4.3.2.- Puntuación de la red

La puntuación que obtiene la red después de una sesión de aprendizaje es fundamental para establecer que redes se desarrollan en posteriores generaciones y cuales son eliminadas del proceso. Aunque se puede facilitar que la red se dirija hacia estrategias consideradas mejores por el desarrollador (por ejemplo, se puede contabilizar el oro generado por una red en la puntuación para priorizar la mejora temprana de minas), se ha intentado mantener al mínimo la intervención externa en el aprendizaje de la red. Por lo tanto, solo se tienen en cuenta tres factores a la hora de establecer la puntuación:

- Si la partida ha sido una victoria o una derrota,
- El daño infligido a la torre rival
- El daño recibido en la torre aliada.

El conjunto de estos factores establece que la máxima puntuación se obtiene al derrotar al rival sin recibir daño, lo cual, si se están entrenando varias redes simultáneamente, puede provocar empates con facilidad. Para reducir lo mínimo posible este efecto, y que el entrenamiento contra múltiples enemigos sea efectivo, la puntuación total es la suma de las puntuaciones de cada partida en cada uno de los lotes de entrenamiento enfrentándose a una red distinta. Esto permite puntuar a cada red en función de su desempeño general en contra de diferentes estrategias y favorecer la aparición de comportamientos más generales.

4.3.3.- Diseño de las nuevas generaciones

Una vez diseñada la puntuación de cada instancia de la IA, es sencillo obtener la mejor de cada una de las redes que se están entrenando. Sin embargo, parte fundamental de un algoritmo genético es que sea capaz de crear nuevas generaciones basándose en los mejores miembros de generaciones anteriores.

Con el objetivo de lograr este cometido, una vez terminada una generación de entrenamiento, cada instancia de cada red neuronal se ordena en función de su puntuación de red. Los primeros dos

miembros de cada red se replican de manera exacta en la siguiente generación, mientras que los demás miembros se van generando de dos en dos.

Para cada dos miembros a generar se seleccionan dos padres de la generación que acaba de completar el entrenamiento, de manera aleatoria, pero con una mayor probabilidad de seleccionar los miembros con mayor puntuación de la red.

Cada parámetro de cada hijo es copiado de uno de los dos padres aleatoriamente, si uno de los hijos obtiene un parámetro del primer padre, el otro hijo obtiene el parámetro correspondiente del segundo padre, de manera que los parámetros quedan recombinados en la nueva generación sin perder ninguno de ellos.

Tras la recombinación genética cada uno de los hijos se ve afectado por una mutación aleatoria de la siguiente lista:

- **No mutar**: El individuo no sufre ninguna mutación.
- **Mutación parcial**: Cada parámetro del individuo puede ser alterado añadiéndole, con cierta probabilidad, una cantidad aleatoria.
- **Mutación imparcial**: Cada parámetro del individuo puede ser restablecido, con cierta probabilidad, a una cantidad aleatoria.
- **Mutar nodos**: Se selecciona una cantidad determinada de neuronas a mutar. Cada parámetro de cada neurona mutada es alterado agregándole una cantidad aleatoria.

Tras crear todos los miembros necesarios para una nueva generación, se duplica y almacena el mejor miembro de cada generación para su uso como rivales en el próximo entrenamiento.

Una vez establecida la nueva generación, se sustituirán los antiguos individuos por los nuevos y se reiniciara el entrenamiento.

5.- DESCRIPCIÓN INFORMÁTICA

Una vez planteado el diseño del juego y de la inteligencia artificial, se procede a su implementación en el motor de videojuegos Unity usando el lenguaje de programación C#. El código fuente se puede ver en *GitHub*, pero se va a hacer una explicación esquemática de la implementación del proyecto en este documento.

5.1.- Implementación del videojuego

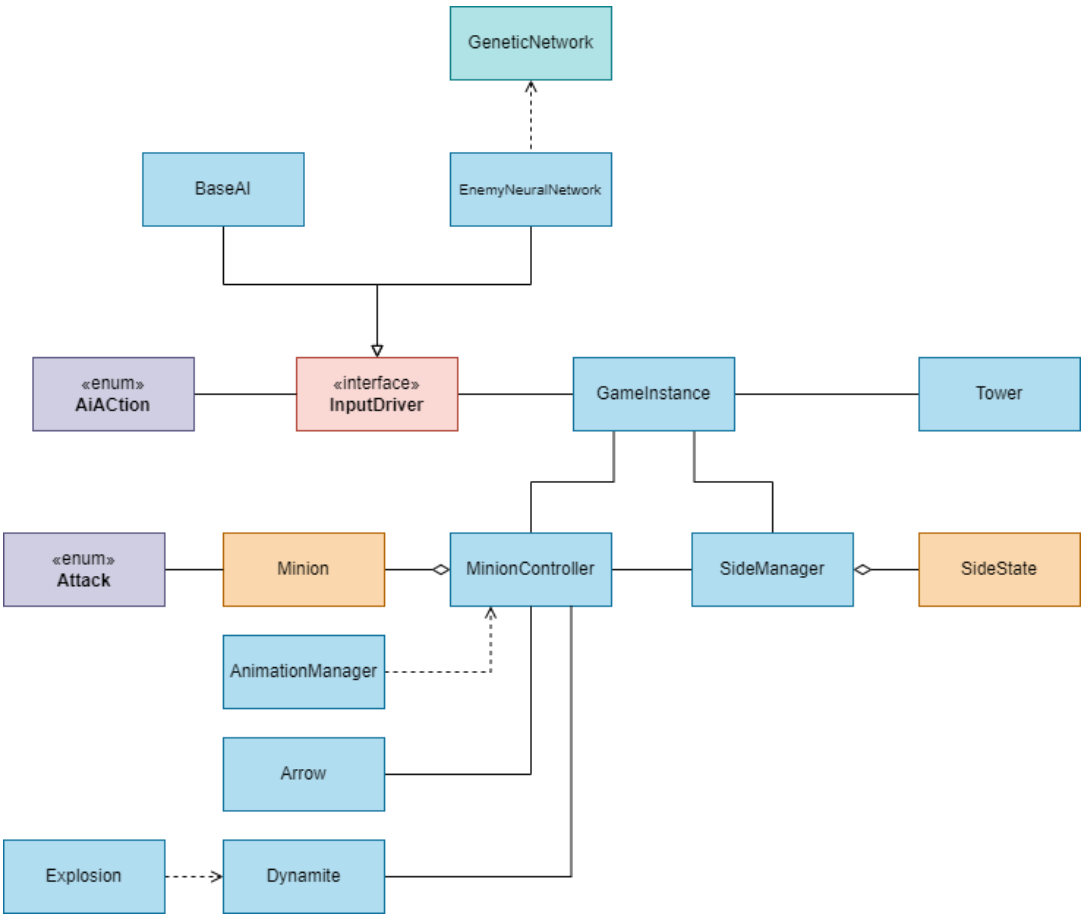

Figura 30: Diagrama de clases del videojuego

En la figura presentada se puede observar el diagrama de clases que se han creado para el funcionamiento del videojuego. La clase principal de donde el resto parten es la clase *GameInstance*, la cual es la encargada de controlar el funcionamiento de una instancia de una partida del juego. Entre sus diversas funciones se pueden encontrar algunas de las más importantes del juego como:

- Comprobar y actualizar el estado de la partida, si esta ha finalizado o tiene que comenzar.

- Comprobar el estado de cada unidad (instancias de la clase *MinionController*) y tener una lista ordenada de ellas según su posición para poder trabajar con sus datos.
- Enviar a cada *InputDriver* el estado de la partida para que pueda tomar una decisión.
- Obtener el resultado de las redes neuronales desde el *InputDriver* y procesarlas para comunicar a cada *SideManager* acción tiene que realizar.
- En el caso de que la partida la juegue una persona y no dos redes neuronales, no se le asignara ningún *InputDriver* al lado azul de la partida.

Intrínsicamente relacionada con la clase *GameInstance* se encuentra la clase *SideManager*, la cual representa cada uno de los bandos presente en el juego (azul o rojo) y se encarga del manejo del estado y los elementos de su lado del juego. En concreto cuenta con funciones para:

- Controlar el gasto y la ganancia de oro
- Generar las unidades
- Actualizar el estado de su bando
- Actualizar las minas

Para mantener el estado del bando en un contenedor que pueda ser compartido fácilmente entre las clases *SideManager* y *GameInstance*, los datos relevantes a este se han visto envueltos en la clase no componente *SideState* (al trabajar en Unity, todas las clases por defecto son componentes que se pueden agregar a los objetos del juego). Entre estos datos se pueden encontrar la vida actual y máxima de la torre, el nivel de la mina o el oro.

En el caso de que un jugador controle uno de los bandos, se establecen una serie de botones en la interfaz de juego para que pueda invocar tropas al pulsarlos. Dichos botones llaman a las funciones necesarias del *SideManager* para cumplir su cometido.

5.1.1.- Unidades

Cada una de las unidades del juego se ve controlada por la clase de *MinionController*, siendo invocados por el *SideManager* y monitorizados por el *GameInstance*. Dentro de esta clase, la parte más compleja

es detectar a los aliados y enemigos para saber cuándo dejar de moverse y cuando atacar, ya que no deben moverse si hay una unidad demasiado cerca de ellos y tienen que atacar a un único enemigo si está en su rango de ataque.

El uso de colisiones es complejo ya que hay que almacenar de alguna manera todos los elementos con los que esta colisionando la unidad (y eliminarlos si dejan de estarlo), ordenarlos y procesarlos para poder cumplir los requisitos. Este sistema se descartó debido a que el funcionamiento del sistema de colisiones de Unity solo permite tratar con una colisión a la vez, además de que no permite detectar si un objeto que estaba colisionando ha sido destruido o inhabilitado.

El uso de *raytracing*, en concreto la función de las físicas de Unity *raycastAll*, ha ofrecido muy buenos resultados, permitiendo detectar fácilmente todos los elementos en frente de una unidad en una cómoda lista para trabajar con ellos. Sin embargo, dicha función es muy costosa computacionalmente, y lo es más cuantos más elementos hay en la escena, por lo que su uso en los entrenamientos donde existen cientos de unidades haciendo uso de ella en cada fotograma del juego ralentizaba enormemente el sistema.

Por último, tras la implementación del sistema de entrenamiento y la clase *GameInstance*, se ha decidido no comprobar ningún tipo de colisión, si no que las unidades vivas se almacenan en una lista de la mencionada clase ordenada por la posición de cada tropa y cada unidad hace las comprobaciones de colisión calculando la distancia con el siguiente aliado en la lista y el primer enemigo de su correspondiente lista. Este método es prácticamente instantáneo, siendo solo necesario el cálculo de dos raíces cuadradas por cada unidad.

Si no se detecta ningún aliado, enemigo o torre que bloquee el paso, la clase *MinionController* se encarga del movimiento de la tropa, y si se encuentra un enemigo o torre invoca la rutina de ataque correspondiente al tipo de unidad, ya sea ataque melee, flecha o dinamita. En cualquiera de los casos, se hace uso de la clase *AnimationManager* para actualizar los visuales de la unidad.

Dicha clase únicamente contiene los mensajes que activan los *triggers* de las animaciones, ya que dicho sistema se ha implementado de manera simplista, donde cada animación tiene un *trigger* que al ser activado cancela la animación que estuviese ejecutándose y la activa a ella en bucle. La excepción es la animación de muerte, que solo se ejecuta una vez tras lo cual la clase *AnimationManager* desactiva los visuales de la unidad para que esta desaparezca de la vista del juego.

La información de cada tipo de unidad viene definida en la clase *Minion*, un *scriptable object* (una clase especial de Unity que puede definirse como un archivo dentro del editor y modificarse desde él directamente). Esta clase, contiene los atributos correspondientes a su tipo de unidad como la vida, el daño, el coste en oro o el animador que se usara del lado rojo y el lado azul. Además de eso, contiene una referencia al enumerador *Attack*, que indica el tipo de ataque que realiza la unidad.

- En un ataque cuerpo a cuerpo simplemente se inflige el daño de ataque al *MinionController* o la Torre enemiga.

- En un ataque de flecha se crea una instancia de la clase *Arrow*, la cual se mueve en la dirección de las tropas hasta colisionar con una unidad o torre enemiga y le inflige el daño de ataque siendo destruida en el proceso. Además, la flecha tiene un rango máximo de movimiento, si lo sobrepasa la flecha se destruye sin infligir daño-

- En un ataque de dinamita se crea una instancia de la clase *Dynamite*. Funciona de manera similar a la clase *Arrow*, pero al llegar al rango máximo o impactar a una unidad, se desactiva su parte visual y activa un objeto hijo de la clase *Eplosion*. La explosión inflige daño en un área una única vez a todos los enemigos dentro de ella y cuando termina su animación ambos objetos son destruidos.

5.1.2.- Comportamiento enemigo

Para que la IA pueda interactuar con la clase *GameInstance* y generar tropas se ha generado la interfaz *InputDriver*, la cual define funciones para procesar los datos vistos en el apartado 4.2.2.- Diseño de las entradas de la red neuronal, empaquetados dentro de la clase *NetworkInput*. De esta clase heredan otras dos:

BaseAI

La clase *BaseAI* modela un modelo de inteligencia artificial basado en probabilidades que se ha usado en el proyecto para el testeo y de prueba contra las redes neuronales. Este modelo asigna a cada posible acción una probabilidad base, tras lo cual interpreta el *NetworkInput* para aumentar o disminuir la probabilidad de ciertas acciones.

Una vez establecida la probabilidad de cada acción se suman las probabilidades de todas las acciones y se escoge un valor aleatorio dentro de esa cantidad. La acción que en la suma correspondiese a dicho número es la que se ejecuta.

EnemyNeuralNetwork

Esta clase trabaja como un intermediario entre la red neuronal en sí y el juego, implementa las funciones de *InputDriver* para pasarle los datos de entrada, recibir la salida, actualizar y obtener los datos sobre la puntuación de la red y reiniciar esta de ser necesario.

5.2.- Implementación de la Red Neuronal

Para la implementación de la red neuronal se ha usado el modelo computacional descrito en el apartado 3.5.- Algoritmos genéticos.

5.2.1.- Red Neuronal general

La clase base para la implementación de la red neuronal es la clase *NeuralNetwork.* Esta clase está formada por una lista (codificadas en un arreglo o *array*) de capas de neuronas. Dentro de esta clase se encuentran funciones para iniciar las capas de la red, obtener el resultado de las capas de salida (la neurona con el valor más alto en la salida), algunas funciones para la retropropagación que no son empleadas en los algoritmos genéticos y funciones para codificar y descodificar la red.

Estas dos últimas funciones son muy importantes, ya que permiten almacenar fácilmente una red en un archivo de texto, además de servir para copiar una red neuronal sin necesidad de usar la misma instancia (esto se usa, por ejemplo, en el entrenamiento de la red, cuando se obtienen los enemigos para él siguiente entrenamiento se copian los valores de la red codificando y decodificando las redes con mejores puntuaciones, ya que la red original se verá modificada por las mutaciones y si simplemente se asigna la red en vez de usar la descodificación lo que se copia es la referencia en memoria de la red y la modificación de esta alterara la IA enemiga). El algoritmo usado para la codificación es bastante simple, se calcula la cantidad de valores que tiene la red sumando la cantidad que tiene cada capa (una capa de n neuronas que está a continuación de una capa de m neuronas contiene $n*m + n$ valores, correspondiente a los pesos y las inclinaciones de cada neurona).

Una vez obtenidas la cantidad de valores, se usará un triple bucle *for* (uno para las capas y los otros dos para los pesos, que están almacenados en un *array* bidimensional). Como se quiere almacenar el resultado en un *array* de una sola dimensión de decimales, se hace uso de una variable índice que indique en que posición hay que escribir dentro del *array* de salida según se vaya avanzando por cada uno de los bucles. Tras completar todas las iteraciones de la red se devuelve el nuevo *array* completo con todos los datos listo para su uso. El algoritmo usado para la decodificación es emplear la misma técnica, pero de manera inversa, obteniendo los valores de un *array* y asignándolos a la propia red.

La clase *Layer* representa cada capa de neuronas dentro de una red neuronal, y es la que implementa la mayoría de las funciones matemáticas para el funcionamiento de la red. Los parámetros que más importan para el proyecto que se está implementando son los pesos, almacenado en un *array* bidimensional de valores decimales, las inclinaciones de cada neurona, almacenadas en un *array* de una sola dimensión de decimales y la activación, una instancia del *enum Activation* que contiene las diferentes funciones de activación que puede usar la red (definidas en el proyecto se encuentran la función de paso, la sigmoide, la ReLU, Softmax y tangencial, pero en el algoritmo genético se usara únicamente la sigmoide). Al estar definida la activación en cada capa en lugar de dentro de la propia red permite definir diferentes funciones de activación para las capas ocultas y las de salida.

Para inicializar las variables de la capa esta consta de dos diferentes funciones, la función de obtener un valor aleatorio para los pesos y la función de obtener uno para las inclinaciones. Estas funciones generan un valor aleatorio entre -1 y 1 y dividen el resultado entre la raíz cuadrada de la cantidad de pesos o inclinaciones que existen en la capa. De esta forma, el efecto en de la red es más pequeña cuando más compleja es esta para compensar el efecto mayor que causa la variación de una gran cantidad de neuronas.

Dicha clase también contiene las funciones matemáticas necesarias para aplicar cada tipo de función de activación, aplicar los pesos de cada neurona sobre un conjunto de entradas y usar la inclinación de cada neurona para calcular su salida. Es importante notar que las neuronas no tienen ningún tipo de representación individual, si no que se modelan de manera abstracta dentro de los *arrays* de pesos e inclinaciones.

5.2.2.- Red neuronal genética

La red anterior es una red básica planificada para la retropropagación. Como no es necesario el uso de esta técnica en una red genética, la clase *GeneticNetwork* es una clase hija de *NeuralNetwork*, lo cual le permite sobrescribir y añadir funciones y atributos para implementar los algoritmos necesarios.

Dentro de los parámetros se añade la puntuación de la red, de manera que esta quede almacenada dentro de la red y se pueda usar fácilmente. Por otro lado, se define el *enum GeneticOperation*, que contiene la definición de varios operadores genéticos para su uso por la red, los cuales han sido vistos en el apartado 3.5.1.- Operadores genéticos. Dicho operador se almacena en una variable dentro de la red.

Las funciones agregadas en la clase son las necesarias para el uso de los diferentes operadores genéticos, siendo los usados en el proyecto la mutación parcial e imparcial, la no mutación y la mutación de nodos. Todos los operadores genéticos constan de un parámetro que es la red genética que se usa como base para la mutación, además de la probabilidad de mutación en el caso de las mutaciones parciales e imparciales y la cantidad de nodos a mutar en la mutación de nodos.

En cualquiera de los casos, la mutación que se genera para una neurona es un valor de la distribución inicial de valores aleatorios.

Dentro de la implementación de estos métodos, cabe destacar que la mutación copia todos los valores de la red padre, independientemente de si estos mutan o no. De esta manera se pueden copiar redes neuronales sin preocupación de la copia de la referencia de estos. Este es el motivo por el cual existe una función para la no mutación, pese a que no se alteren los valores se usa para copiar todos los datos de la red padre.

La red neuronal que se usa en el proyecto es una red con función de activación sigmoide, tanto en las capas ocultas como en las de salida, que cuenta con una capa de entrada de 15 neuronas correspondientes a los valores vistos en el apartado 4.2.2.- Diseño de las entradas de la red neuronal. Tras varias iteraciones de entrenamiento y comprobación se han establecido dos capas ocultas en la red de 40 y 30 neuronas respectivamente. Por último, la capa de salida cuenta con 7 neuronas correspondientes a las 7 posibles decisiones que puede tomar la red.

5.3.- Implementación del entrenamiento.

El entrenamiento es llevado a cabo por la clase *GeneticManager*, una clase capaz de crear y controlar grandes cantidades de *GameInstance* de manera simultánea. Dentro de los parámetros que usa esta clase se encuentran las variables que controlan las mutaciones y el control de las nuevas generaciones, los datos sobre las redes neuronales a crear, listas con todas las redes neuronales genéticas siendo entrenadas y variables para almacenar las mejores redes de cada generación y tipo.

Para los entrenamientos, se ha creado una escena diferente de la de juego usada solo en el desarrollo, que no esta disponible en la aplicación final. Esta escena contiene una instancia de la clase *GeneticManager*, la cual al iniciar la escena instancia una cantidad *K* de instancias de la clase *InstanceManager*. Dentro de la clase esta definida la cantidad de tipos diferentes de redes a emplear simultáneamente *N* (para el entrenamiento final de la red se han usado los valores *K*=500, *N*=4).

5.3.1.- Primera generación

Tras la creación de los elementos necesarios para el entrenamiento, se procede a la creación de la primera generación de agentes para el entrenamiento. Para ello, se dividen las instancias creadas entre la cantidad *N* de tipos de redes empleadas. Cada una de las divisiones será denominada familia de ahora en adelante.

Para cada miembro de cada familia se crea una red neuronal genética con parámetros aleatorios, la cual se añade a la lista correspondiente de redes y a la instancia de *InstanceManager* correspondiente se le asocia la red mediante la creación de un miembro de *EnemyNeuralNetwork* al que se le ha asociado la red creada y un índice correspondiente a la familia. Este objeto se asocia al lado azul de la instancia de juego, que es el lado que se entrena, al lado rojo se le asociaran más tarde los enemigos para ver la eficacia de la inteligencia.

Para facilitar el entrenamiento y no perder datos entre diferentes ejecuciones de este, si existen archivos de guardado de las redes, estos se cargan sobrescribiendo las primeras redes generadas aleatoriamente de cada familia.

Por último, se seleccionan los primeros miembros de cada red como enemigos para practicar de base y se comienza el entrenamiento.

5.3.2.- Archivos de guardado

Para permitir el entrenamiento a lo largo de diferentes ejecuciones del programa y poder usar el resultado de este en la aplicación de manera permanente, tras la ejecución de cada instancia de entrenamiento se almacenan una cantidad determinada *P* de redes de cada familia en un archivo de texto. Cada familia tiene su propio archivo individual.

Los miembros seleccionados de cada familia serán aquellos con la puntuación más alta obtenida en el entrenamiento, asegurando la supervivencia de los miembros más aptos de cada tipo de red. Si se ejecuta una partida normal, se carga únicamente el primer miembro de la familia correspondiente a la dificultad escogida como rival para el jugador. En el caso de iniciar un entrenamiento existiendo dichos archivos de guardado, al crear las redes base para el entrenamiento se sustituyen los *P* primeros miembros de cada familia por cada una de las redes almacenadas en su archivo correspondiente.

Para guardar una red en un archivo de texto se utilizan las funciones *Encode* para escribir y *Decode* para leer, ambas definidas en la clase *NeuralNetwork*. En los archivos de texto se separa cada red usando un salto de línea. Al trabajar con archivos de texto es importante el uso de la cultura no variante para comunicarse entre el código y el archivo, ya que en diferentes ordenadores puede cambiar la manera de leer y escribir números decimales dependiendo del idioma empleado por el dispositivo.

5.3.3.- Entrenamiento

La ejecución del entrenamiento es un ciclo que se repite hasta la pausa manual de la ejecución. En cada iteración se realizan los siguientes pasos:

1. Reiniciar la puntuación de cada red.
2. Por cada familia de redes:
    2.1. Establecer como rival de entrenamiento el mejor miembro de la familia en la generación anterior (o el primero de la lista en la primera generación) y reiniciar el estado de cada instancia.
    2.2. Esperar cierta cantidad de tiempo *T* para dar tiempo a que todas las redes entrenen.

2.3. Tras finalizar el tiempo de espera se da el entrenamiento contra el rival por completo, se actualiza la puntuación de la red y se repite el proceso para enfrentarse a la siguiente familia.

3. Tras entrenar cada instancia contra los mejores miembros de cada familia, se ordena cada lista de redes según la puntuación obtenida por cada una.

4. Se almacenan los *P* mejores miembros de cada familia en su archivo correspondiente.

5. Se duplica el mejor miembro de cada familia para su uso como rival en la siguiente iteración de entrenamiento.

6. Se genera la siguiente generación de redes siguiendo lo definido en el apartado 4.3.3.- Diseño de las nuevas generaciones y se actualiza cada agente para usar las nuevas redes.

5.4.- Optimizaciones para la ejecución de múltiples instancias.

Como se ha visto, para realizar una generación de entrenamiento se necesita esperar una cantidad de tiempo $T * N$. En los entrenamientos realizados se ha usado una cantidad $T = 80$ segundos. Esto significa se necesita esperar 5 minutos y 20 segundos por generación. En un contexto en el que para obtener resultados se necesitan cientos o miles de generaciones, esta cantidad de tiempo supone un gasto excesivo de recursos. Para acelerar el entrenamiento se ha modificado un modificador de la escala del tiempo *S* (se ha usado un valor final de $S = 4$ finalmente en el proyecto). Este valor acelera el juego *S* veces, es decir, un valor de 2 hace que el programa se ejecute el doble de rápido, de 3 el triple y así sucesivamente.

Es importante tener cuidado con este valor, porque acelerar mucho el programa mientras se ejecutan una gran cantidad de instancias puede bajar drásticamente los FPS de la aplicación, pudiendo provocar errores en el sistema. Uno de estos problemas ha sido el *Pass Through*, un error común en los videojuegos en los que si un objeto se mueve muy rápido puede atravesar otros objetos al no detectarlos a tiempo. Como la cantidad movida por cada unidad depende del tiempo entre *frames*, si este es muy alto pueden atravesarse sin llegarse a detectar, además de que produce que la distancia entre unidades sea irregular, pudiendo estar mas cerca de lo indicado por el sistema dependiendo de la distancia que se hayan movido en este fotograma. Para solventar este problema, antes de realizar el movimiento de

cada unidad se almacena la unidad más cercana (las unidades deben estar a una distancia determinada de la siguiente unidad en el juego, ya sea aliada o enemiga). Si tras el movimiento la unidad la ha atravesado o esta más cerca de lo indicado, se hace retroceder la unidad hasta estar a la distancia ideal. De esta manera se logra que la distancia entre unidades sea regular y se evita en gran parte los problemas de atravesar unidades.

5.4.1.- Optimización del movimiento de las unidades

El movimiento de cada unidad del juego ha sido sujeto de cambio a lo largo del proyecto. La iteración inicial de este proceso hacia uso de *RaycastAll para* detectar las siguientes unidades en el terreno y moverse o atacar adecuadamente. Aunque este proceso funciona correctamente, a la hora de realizar el entrenamiento (que consta de 500 instancias, cada una con dos IAs generando unidades simultáneamente) se podían generar varios miles de unidades usando *Raycast* constantemente. Este proceso es pesado, sobre todo al tener en cuenta que dicha función comprueba todos los objetos validos existentes, por lo que al crear más unidades no solo se hacen más llamadas, si no que estas requieren más computo. Se intento hacer un cambio a *RaycastAllNonAlloc*, ya que la función *RaycastAll* genera una lista nueva cada vez que se ejecuta, mientras que dicha nueva función usa una lista ya creada para ejecutarse. Sin embargo, la mejora en rendimiento fue mínima y se tuvo que cambiar el sistema de detección.

El nuevo sistema consiste en mantener una lista de todas las unidades vivas de cada bando del juego en el *InstanceManager.* Para obtener los enemigos o aliados cada unidad solo tiene que acceder a dicha lista y comparar las distancias, proceso mucho más rápido y eficiente que el anteriormente descrito, además de evitar interferencias con otras instancias del juego ya que las unidades solo pueden interactuar con las que existen dentro de su instancia, y es el sistema que se ha usado finalmente.

5.4.2.- Optimización de los efectos visuales

Para ahorrar en rendimiento de la aplicación, durante el entrenamiento se reducen enormemente los efectos visuales.

Para lograr este cometido, cada *InstanceManager* cuenta con una lista con todos los elementos visuales que contiene su instancia de juegos, y si se ejecuta la instancia en un entrenamiento todos estos elementos se desactivan para no tener la necesidad de renderizarlos. Sin embargo, el mayor coste de renderizado está en las animaciones de las unidades.

Aunque se podrían desactivar también completamente los visuales de las animaciones, se ha decidido sustituirlas por imágenes estáticas de la unidad, de manera que se pueda supervisar el entrenamiento reduciendo al mínimo el coste de renderizando, pero manteniendo una imagen real de lo que esta pasando. Otras imágenes que se mantienen renderizadas son las torres y las minas, ya que su estado se indica visualmente.

Para eliminar las animaciones, el controlador de animaciones de cada unidad cuenta con un estado especial en el que solo se visualiza la imagen estática de la unidad. Al crear la unidad, se le indica al animador que entre en ese estado y se bloquea cualquier llamada que intente actualizar las animaciones de la unidad.

5.4.3.- Pooling

Al constar de cientos de instancias creando unidades, que a su vez crean otros elementos como flechas o dinamita, el motor de juego deber estar creando y destruyendo objetos constantemente. Para disminuir enormemente el tiempo dedicado a ese proceso, se puede hacer del Pooling, una técnica que consiste en generar una gran cantidad de objetos al principio y mantenerlos inactivos hasta que sea necesario su uso. Al necesitarlo se activa uno de los objetos y se mueve a la posición deseada, eliminando todo el proceso de eliminar y crear objetos. Como punto negativo, esta técnica supone un mayor coste de memoria, pero reduce el consumo de procesador por lo que es útil para extraer unos cuantos fotogramas por segundo extra.

Para implementar esta técnica, las clases que son creadas y destruidas constantemente (*MinionController, Arrow, Dynamite y Explosion*) se establecen como hijas de la clase *Poolable*. Esta clase base consta de funciones que se pueden sobrescribir para activar el objeto, desactivarlo y eliminarlo, además de contar con una referencia a un controlador (*ObjectPooler*) asociado al objeto.

Existe uno de esos controladores para cada tipo de *Poolable*, contando con 4 en total. Este consta de una referencia al objeto a manejar y una lista de los objetos ya creados y disponibles. Este controlador cuenta con dos funciones, *GetItem*, que devuelve el ultimo objeto, lo elimina de la lista y lo activa, o, en caso de no existir objetos disponibles, crea uno nuevo y lo devuelve, y la función *ReturnItem*, función a la que hay que pasarle uno de los objetos controlados, lo desactiva y lo vuelve a añadir a la lista.

De esta manera, en vez de destruir objetos se desactivan y devuelven al controlador. Cuando se necesita crear un objeto se obtiene del controlador, usando uno creado anteriormente o creando uno nuevo si no existe ninguno disponible. Cada clase hija define sus propios métodos para el correcto funcionamiento de esta técnica, reiniciando sus valores o reasignándolos y cambiando el objeto padre al activarse y desactivarse.

6.- VALIDACIÓN

Problemas con la codificación de enemigos.
Validar si la solución funciona o no funciona con una escala SUS, mínimo 25 personas. Sexo y edad.

7.- CONCLUSIONES

Respuesta a los objetivos.

8.- APENDICE

8.1.- Mejoras en el aprendizaje de una red neuronal

-Diferentes funciones de activación
-Inercia del gradiente
La inetia

Sin Redactar

-Desarrollo de la aplicación

Para desarrollar la aplicación se ha usado el motor de videojuegos Unity.

Elementos gráficos de creación propia

Bola

Paletas

Tablero

Cada una de las paletas puede desplazarse libremente por su mitad del campo y el objetivo será empujar la bola hasta la portería rival (franjas azul y roja).
Los elementos del juego se mueven usando el motor de físicas de Unity:

- El tablero es estático, tiene las esquinas con forma triangular para evitar que la bola se quede atascada. Todas las paredes tienen una colisión donde la pelota puede rebotar, a excepción de las porterías que tienen un *trigger* que se activa cuando la pelota entra en ellas, dándole la victoria al jugador del campo opuesto del tablero.
- La pelota usa un *rigidbody* dinámico, un componente que le permite simular propiedades físicas en tiempo real. Consta de un material físico que modifica sus propiedades de fricción y rebote, permitiendo controlar el comportamiento físico de la bola.
- Las paletas usan, al igual que la pelota, un *rigidbody* dinámico, el cual es controlado por código para otorgarle una fuerza constante en una dirección normalizada de movimiento, en el caso de estar controlada por un jugador en dirección al ratón, y en el caso de estar controlada por la IA en la dirección determinada por la red neuronal. Como la aceleración es constante, para evitar comportamientos no deseados y darle más realismo a la simulación, la velocidad esta capada en un máximo, haciendo que las paletas no aceleren más al llegar a esa velocidad. (¿anexo 1?)

-Sigmoide vs RElU

Sigmoide es muy lento y costoso de entrenar, las redes neuronales modernas usan ReLU (Rectified Linear Unit), una función que simplemente coge el valor máximo entre 0 y a, estando simplemente activada o no, teniendo un límite de activación-

USO de Math.net para el cálculo de matrices

qlearning

ENLACES

- Aplicación interactiva (visualización de redes neuronales): https://dakexd.itch.io/redesneuronales
- Código fuente de la aplicación interactiva: https://github.com/dakeXd/NeuralNetworkVisualization
- Código fuente del videojuego: https://github.com/dakeXd/TFGroyale
- Videojuego:
- *Suck Up: https://www.playsuckup.com*
- *Oasis: https://oasis-model.github.io*

BIBLIOGRAFÍA

[1] Michael A. Nielsen (2015). *Neural Networks and Deep Learning*. Determination Press

[2] *Lie, C. S. K., & Istiono, W. (2022). How to make npc learn the strategy in fighting games using adaptive AI?. Int J Sci Tech Res Eng, 7(4).*

[3] *Tewari, A., Fried, O., Thies, J., Sitzmann, V., Lombardi, S., Sunkavalli, K., ... & Zollhöfer, M. (2020, May). State of the art on neural rendering. In Computer Graphics Forum (Vol. 39, No. 2, pp. 701-727).*

[4] *Plut, C., & Pasquier, P. (2020). Generative music in video games: State of the art, challenges, and prospects. Entertainment Computing, 33, 100337.*

[5] Rouhiainen, L. (2018). *Inteligencia artificial*. Madrid: Alienta Editorial.

[6] Blue1Brown. (2017). *Neural networks* [Video Series]. YouTube. https://youtube.com/playlist?list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi

[7] REAL ACADEMIA ESPAÑOLA. (2023) *Diccionario de la lengua española, 23.ª ed.*, https://dle.rae.es/inteligencia

[8] Sebastian Raschka (2016) *Gradient Descent and Stochastic Gradient Descent* https://www.quora.com/Whats-the-difference-between-gradient-descent-and-stochastic-gradient-descent/answer/Sebastian-Raschka-1

[9] Michael A. Nielsen (2015). *Neural Networks and Deep Learning*. GitHub repository, https://github.com/mnielsen/neural-networks-and-deep-learning

[10] Frank Rosenblatt (1962*) Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms.* Universidad de Michigan

[11] Russell, Stuart J.; Norvig, Peter. (2021). *Artificial Intelligence: A Modern Approach (4th ed.)*. Hoboken: Pearson

[12] Luger, George; Stubblefield, William (2004). *Artificial Intelligence: Structures and Strategies for Complex Problem Solving (5th ed)*. Benjamin/Cummings.

[13] Poole, David; Mackworth, Alan; Goebel, Randy (1998*). Computational Intelligence: A Logical Approach*. New York: Oxford University Press.

[14] Sheth, D., & Giger, M. L. (2020). Artificial intelligence in the interpretation of breast cancer on MRI. *Journal of Magnetic Resonance Imaging*, *51*(5), 1310-1324.

[15] Abdi, H., Valentin, D., & Edelman, B. (1999). *Neural networks* (No. 124). Sage.

[16] Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning*. MIT press.

[17] Yann Lecun, Yoshua Bengio, Geoffrey Hinton. (2015). *Deep learning.* Nature, 521 (7553), pp.436-444.

[18] Feuerriegel, S., Hartmann, J., Janiesch, C., & Zschech, P. (2024). Generative ai. *Business & Information Systems Engineering*, *66*(1), 111-126.

[19] Kanal, L. N. (2003). Perceptron. In *Encyclopedia of Computer Science* (pp. 1383-1385).

[20] Guo-Xun Yuan; Chia-Hua Ho; Chih-Jen Lin (2012). *Recent Advances of Large-Scale Linear Classification*, Proc. IEEE. 100 (9)

[21] Hastie, Trevor; Tibshirani, Robert; Friedman, Jerome H. (2001*). The Elements of Statistical Learning*. Springer. p. 18.

[22] Bishop, Christopher (2008) *Pattern Recognition and Machine Learning*. Springer Verlag.

[23] George F. Simmons, *Calculus with Analytic Geometry*, McGraw-Hill Education

[24] Kaelbling, L. P., Littman, M. L., & Moore, A. W. (1996). Reinforcement learning: A survey. *Journal of artificial intelligence research*, *4*, 237-285.

[25] Forrest, S. (1996). Genetic algorithms. *ACM computing surveys (CSUR)*, *28*(1), 77-80.

[26] Montana, D. J., & Davis, L. (1989, August). Training feedforward neural networks using genetic algorithms. In *IJCAI* (Vol. 89, No. 1989, pp. 762-767)