Universidad Politécnica
de Madrid

Escuela Técnica Superior de Ingenieros Informáticos

Grado en Ingeniería Informática

Trabajo Fin de Grado

Implementación de Red de Neuronas en C en Entorno GNU Linux

Autor: Álvaro González Méndez
Tutor(a): José Crespo del Arco

Madrid, Junio 2024

Este Trabajo Fin de Grado se ha depositado en la ETSI Informáticos de la Universidad Politécnica de Madrid para su defensa.

Trabajo Fin de Grado
Grado en ingeniería informática
Título: Implementación de Red de Neuronas en C en Entorno GNU Linux
Junio 2024

Autor: Álvaro González Méndez

Tutor:
José Crespo del Arco
Departamento de Lenguajes y Sistemas Informáticos e Ingeniería de Software
ETSI Informáticos
Universidad Politécnica de Madrid

Resumen

Durante los últimos años hemos visto como las diferentes inteligencias artificiales como los modelos GPT o DALL-E de OpenAI han sido el punto de mira para toda la sociedad, siendo así que a diario oímos sobre nuevas tecnologías y avances en este campo tanto de personas cercanas como de los medios de comunicación. Muchos de los usuarios con acceso a internet hoy en día dan uso estas emergentes tecnologías en el trabajo y educación como herramientas de apoyo y consulta, siendo cada vez más común encontrar soluciones y sistemas que usen estas inteligencias artificiales.

Las redes neuronales artificiales forman parte del conjunto de instrumentos de Aprendizaje Automático (Machine Learning), que se utilizan para diseñar y desarrollar modelos que permiten dar respuestas a problemas complejos mediante un entrenamiento iterativo. Como su nombre indica, las redes neuronales artificiales se inspiran en las neuronas del sistema nervioso biológico y en la conexión entre ellas con el fin de simular una falsa inteligencia.

Pese a que ahora estén en pleno auge, las redes neuronales no son tan modernas como aparentan, los primeros intentos de diseños que proponían la estructura de una red de neuronas datan de la década de los 1940s. Pero hasta el final de los 2000s y principios de los 2010s no se pudieron lograr los hitos masivos que vemos que cada poco tiempo incrementan la capacidad de las IAs y nos acercan a los futuros descritos en la ciencia ficción donde las maquinas son capaces de tomar decisiones y razonar por si solas.

En este trabajo nos adentraremos en una de las bases de las nuevas tecnologías emergentes, las redes de neuronas artificiales, y sus principios más básicos, construyendo una librería en el lenguaje C que nos permita crear redes y entrenarlas de forma sencilla. Exploraremos la raíz de este modelo matemático y diferentes formas de aplicarlo para resolver problemas de clasificación, predicción y reconocimiento de patrones.

Empezaremos con el estudio de los modelos más sencillos y la implementación de diferentes de estos aumentando la complejidad y añadiendo ajustes y técnicas cada vez. Pasando desde el perceptrón y redes de una a dos capas con ajuste lineal a redes multicapa con aprendizaje de retro propagación hasta llegar a implementar una solución que con pocos parámetros nos permita generar redes con diferentes configuraciones y entrenarlas.

Abstract

During the last few years, we have seen how the different artificial intelligences such as OpenAI's GPT or DALL-E models have been the focus of attention for the whole society, so we hear daily about new technologies and advances in this field from people close to us as well as from the media. Many of the users with access to the Internet today make use of these emerging technologies in work and education as support and consultation tools, being increasingly common to find solutions and systems that use these artificial intelligences.

Artificial neural networks are part of the Machine Learning toolset, which are used to design and develop models that provide answers to complex problems through iterative training. As their name suggests, artificial neural networks are inspired by the neurons of the biological nervous system and the connection between them to simulate a false intelligence.

Although they are now booming, neural networks are not as modern as they appear to be the first attempts at designs proposing the structure of a network of neurons date back to the 1940s. But it was not until the late 2000s and early 2010s that the massive milestones witnessed by AIs were achieved, increasing the capacity of Ais, and bringing us closer to the futures described in science fiction where machines can make decisions and reasoning on their own.

In this work we will go into one of the foundations of the new emerging technologies, artificial neural networks, and their most basic principles, building a library in the C language that allows us to create networks and train them in an effortless way. We will explore the root of this mathematical model and unusual ways to apply it to solve classification, prediction and pattern recognition problems.

We will start with the study of the simplest models and the implementation of different of these increasing the complexity and adding adjustments and new techniques each time. Going from the perceptron and one to two-layer networks with linear adjustment to multi-layer networks with backpropagation learning until we get to implement a solution, that with few parameters allows us to generate networks with different configurations and train them.

Tabla de contenidos

1 Introducción 1
1.1 Motivación del proyecto 1
1.2 Hitos y Objetivos 1
1.3 Estructura del proyecto 2
2 Estado del Arte 3
2.1 Funcionamiento de las Redes Neuronales 3
2.2 Historia de las Redes de Neuronas 4
2.3 Modelos más comunes de las Redes de Neuronas 6
2.3.1 Perceptrón 6
2.3.2 ADALINE 7
2.3.3 Perceptrón Multicapa 7
2.3.4 Algoritmo de Retropropagación 8
2.3.5 Redes de Neuronas Convolucionales 9
3 Desarrollo de la librería 10
3.1 Herramientas utilizadas para la realización del trabajo 10
3.2 Modelos Mononeurona 10
3.2.1 ADALINE 11
3.2.2 Perceptrón 13
3.3 Código Auxiliar 14
3.3.1 Matrices – matrix.h 15
3.3.2 Manejador de datos – data_handler.h 15
3.4 Redes Multicapa Sencillas y Retropropagación 16
3.4.1 Perceptrón multicapa 16
3.4.2 Retropropagación 20
3.4.3 Optimizadores 25
3.4.4 Refactorización de red 29
3.5 Librería GMLNN 30
3.5.1 Estructuras de datos y Parámetros de estas 30
3.5.2 Funciones de Activación 34
3.5.3 Alimentación de la red de neuronas 38
3.5.4 Funciones de Error y Optimizadores 39
3.5.5 Aprendizaje Supervisado mediante Retropropagación 40
3.5.6 Guardado y Cargado de Redes de Neuronas 43
3.5.7 Función de entrenamiento simplificado 45
3.5.8 Funciones extra de visualizado y otros métodos auxiliares 46
4 Ejemplos de uso, evaluación y discusión 48
4.1 Uso de la librería como programador 48
4.1.1 Puerta lógica XOR 48

4.1.2 Funciones lógicas más complejas ... 49
4.1.3 Clasificación de puntos en un espacio 2D ... 51
4.1.4 Predicción con datos reales (Diabetes y VIH). ... 53
4.1.5 Reconocimiento de dígitos MNIST ... 56
4.2 Evaluación de los resultados ... 57
5 Resultados y conclusiones ... 59
5.1 Repaso de objetivos ... 59
5.2 Conclusiones sobre los resultados ... 60
5.3 Futuras líneas de Trabajo ... 60
6 Análisis de Impacto ... 62
Bibliografía ... 64
Anexo A - Manual de uso de matrix.h ... 66
Interfaz de uso (listado de métodos) ... 66
Ejemplos de código ... 70
Anexo B - Manual de uso de data_handler.h ... 73
Interfaz de uso (listado de métodos) ... 73
Ejemplos de código ... 76
Anexo C – Manual de la librería gml_nn.h ... 78
Interfaz de uso (listado de métodos) ... 80
Ejemplos de código ... 88

1 Introducción

Muchas veces, se habla de redes neuronales sin saber realmente que está detrás, como funcionan o que aspecto tienen y como interactuar con ellas. Esta es una de las inquietudes que personalmente me han empujado a la realización de este trabajo. En este primer capítulo se hablará de cómo se afrontará y estructurara el proyecto, que pasos se seguirán y que objetivos y tareas tienen que ser realizadas para dar con la solución propuesta de una pequeña librería en C que nos permita crear Redes Neuronales y realizar entrenamientos para resolver diferentes problemas.

En las siguientes secciones, se expondrán los diferentes factores que motivan a la realización del proyecto (sección 1.1). Después, hablare de los objetivos que propone este durante el TFG (sección 1.2). Finalmente, describiré la estructura que este documento va a seguir con unas breves descripciones de lo que se podrá leer en cada capítulo y la finalidad de estos (sección 1.3).

1.1 Motivación del proyecto

Con los nuevos avances en la inteligencia artificial y una gran mayoría de modelos avanzados abiertos al uso público de forma gratuita o por suscripción bajo un precio aceptable por la mayoría de los usuarios de Internet, términos como "inteligencia artificial", "redes neuronales", "modelos generativos" o "machine learning" se encuentran en nuestro día a día. Se pueden escuchar en las noticias, en cualquier conversación de amigos (sin necesidad de que sea entre técnicos o entendidos en el campo) y hasta en reuniones familiares. Pero realmente no son tantas las personas que realmente saben que es lo que tienen enfrente y que es lo que no.

Se ha llegado a un punto en el que la mayoría de las personas creen que estamos frente a tecnologías imposibles o "místicas", cuando realmente las bases de estas existen bajo conceptos sencillos que cualquiera con conocimiento medio de matemáticas podría comprender. No hablo de los últimos modelos como GPT de OpenAI, Watson de IBM o AplhaGo de Google, sino de modelos y conceptos más primitivos que asientan los pilares de las más avanzadas.

Estamos hablando de un tema que está en pleno "boom" y no para de crecer a velocidades exponenciales. Se trata de algo revolucionario como pudo ser la introducción de la máquina de vapor, la mecanización de procesos industriales y la creación de Internet. Claramente, en este proyecto no se pretende hacer competencia a ningún producto, programa o código, sino que sea una oportunidad para investigar las bases de las redes neuronales y aprender cómo se construyen y que mecanismos esconden debajo los complejos modelos que están dando tanto de que hablar.

1.2 Hitos y Objetivos

El objetivo principal del trabajo a realizar es implementar una librería en C que permita en pocas líneas de código describir una red de neuronas mediante una serie de parámetros como la cantidad de capas, el ancho de cada capa, la función umbral o de activación que realiza cada célula… y después entrenarla para que pueda resolver problemas de clasificación, predicción y reconocimiento de patrones.

Como dice el viejo refrán, "no se puede construir la casa por el tejado" y en este caso, al momento de empezar con el proyecto no se sabía por dónde empezar los cimientos. Por ende, antes de empezar con cualquier tipo de desarrollo sobre lo que va a ser el código final de la librería, se marcan ciertos hitos y objetivos que vayan aportándome cierto conocimiento previo acerca de las redes de

neuronas para poder afrontar la implementación de la librería final. Estas objetivos son las siguientes:

- Estudio de los principios básico de una Red Neuronal Artificial, del algoritmo de retro propagación para el aprendizaje supervisado y revisión de iniciativas relacionadas.
- Primeras Implementaciones de redes básicas (perceptrón, redes de dos capas…) y mecanismos de aprendizaje sencillos.
- Implementación de redes un más complejas y mecanismo de aprendizaje de retro propagación junto a los distintos tipos de descenso de gradiente para el aprendizaje y calculo mediante análisis de las funciones de minimizado de error para los distintos modelos.
- Análisis de los resultados de las redes implementadas anteriormente.
- Implementación de librería en C que permita crear y entrenar Redes Neuronales.
- Análisis de los resultados y comparación de rendimiento de las redes neuronales generadas y las librerías ya existentes.

1.3 Estructura del proyecto

La memoria seguirá la siguiente estructura:

- Capítulo 2 – **Estado del Arte**. Se discutirá sobre la historia de las redes neuronales desde el primer diseño hasta las más notables hoy en día y de las actuales soluciones relacionadas con el proyecto.
- Capítulo 3 – **Desarrollo de la Librería**. Se hará un seguimiento de todas las tareas previas realizadas antes de realizar el código que implementa la librería. Se estudiará a detalle cada componente que pueda ser agregado y finalmente se desarrollará el código final que soluciona el objetivo final planteado.
- Capítulo 4 – **Ejemplos de uso, Evaluación y Discusión**. Completado el código, serán realizadas ciertas pruebas, mostrado la forma de utilizar la librería y comparado con otras soluciones encontradas en internet para evaluar si la librería puede ser usada para generar redes que permitan resolver diferentes problemas.
- Capítulo 5 – **Resultados y Conclusiones**. Se hará una reflexión personal sobre los resultados obtenidos y que tan satisfecho he quedado con el trabajo realizado. A parte analizare los fallos y errores cometidos durante el desarrollo del trabajo para poder encontrar posibles puntos a mejorar y posibles futuras vías de desarrollo de la librería.
- Capítulo 6 – **Análisis de Impacto**. Finalmente se hablará del impacto de carácter personal, social, económico, medioambiental y cultural que pueda tener el proyecto y las decisiones tomadas a lo largo del trabajo que tienen como base la consideración del impacto.

2 Estado del Arte

Este capítulo nos sirve como punto de entrada al proyecto, ya que hablaremos de forma resumida cómo funciona una red neuronal y discutiremos sobre la historia de estas a través del tiempo, así partiendo de sus primeros diseños y los más modernos llegando a la actualidad.

Primero entenderemos el funcionamiento y los componentes más básicos de una red y sus neuronas (sección 2.1), luego revisaremos las propuestas que han ido surgiendo a lo largo de la historia (sección 2.2) y finalmente entraremos un poco más a fondo y hablaremos del funcionamiento de los modelos más comunes y sencillos (sección 2.3).

2.1 Funcionamiento de las Redes Neuronales

Basándonos en los sistemas nerviosos biológicos, durante el siglo XX se han diseñado varios modelos matemáticos que intentan simular el funcionamiento de las neuronas y la interconexión entre ellas. El objetivo es crear un modelo que aprenda como hace un animal de sus experiencias pasadas sobre un problema y consiga solucionar nuevos casos relacionados con autonomía. A estos modelos reciben el nombre de **Redes de Neuronas Artificiales**.

Las neuronas biológicas están compuestas por tres partes principales, el cuerpo celular o soma, que se encarga de mantener el núcleo con el ADN y generar la energía para que la célula siga funcionando de forma correcta; los axones, que se encargan de transportar información en forma de pulsos eléctricos mediante sinapsis a otras neuronas; y las dendritas, que reciben mediante sinapsis de otras neuronas impulsos eléctricos. Tomando de referencia estas tres partes, se crean las neuronas artificiales, que tienen unos valores de entrada con sus respectivos pesos (dendritas), un núcleo que procesa estas entradas (soma) y una salida del dato procesado (axón).

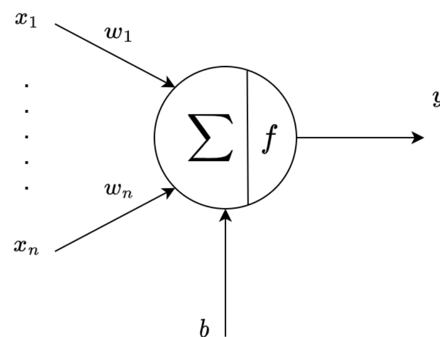


Figura 2.1 Representación de una Neurona Artificial

Nuestra neurona recibe n valores x y los multiplica por sus respectivos pesos, w y posteriormente los suma todos, la mayoría de las neuronas también cuentan con un valor de sesgo b que en caso de que cuente con ello, también se incluye en la operación de sumatorio. Después, el resultado se usa como parámetro de una función umbral o de activación y el resultado obtenido y se propaga a otras neuronas que realizaran el mismo proceso. Resumiendo, el resultado de una sola neurona puede ser descrito de la siguiente forma:

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

Dentro de las redes artificiales, las neuronas están organizadas en grupos que operan simultáneamente en orden, estos son llamados capas en los modelos que implementaremos posteriormente, todas las salidas de las neuronas de una capa sirven como las entradas para todas las neuronas de la siguiente. Considerando como salida de la red, el conjunto de salidas de las neuronas de la última capa.

A la primera capa se le denomina capa de entrada, ya que es la única que obtiene los valores de entrada, a la ultima la de salida, puesto que devuelve la salida de la red y a las intermedias ocultas porque sus resultados pese a ser necesarios, no son directamente accesibles desde el exterior y no están involucradas en la entrada o salida directa de datos. Un simple ejemplo de esto podría ser la siguiente que cuenta con 3 capas (entrada, salida y 1 capa oculta).

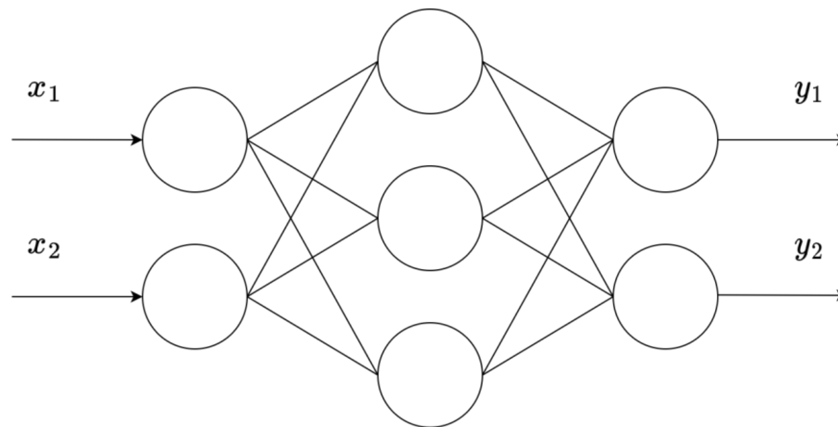


Figura 2.2 Red de Neuronas con una capa oculta

Las redes pueden ser de distintas formas y contener modificaciones que les permitan tener mejor rendimiento para distintos tipos de problemas, como el objetivo del TFG es crear una librería general, los ejemplos anteriores muestran ejemplos muy sencillos sin modificaciones. En el Capítulo 3 se profundizará más en los conceptos expuestos en este epígrafe.

2.2 Historia de las Redes de Neuronas

Las redes de neuronas que ahora tienen tanto impacto llevan entre nosotros más de 80 años, en esta sección hablaremos de las distintas propuestas y soluciones que se han ido diseñando durante el tiempo hasta la actualidad apoyándonos en el artículo académico de la Universidad Tecnológica de Varsovia (en la cual curse el año pasado dos semestres) escrito por Bohdan Macukow [1] y el libro *Redes de Neuronas: Un enfoque practico* de Pedro Isasi e Inés Galván[17].

El neurofisiólogo Warren McCulloch junto al matemático Walter Pitts en el año 1943 escribirían el articulo *The Logical Calculus of the Ideas Immanent in Nervous Activity* que se consideraría como los primeros pasos que se dieron sobre las redes neuronales artificiales [2]. Estos dos científicos crearon la llamada lógica umbral, que explora la idea del “todo o nada” e introducen un modelo simplificado de las neuronas que siguen esta regla, estableciendo así una relación entre la actividad neuronal y la lógica proposicional, sugiriendo que se podrían describir las redes de neuronas mediante circuitos lógicos. Cabe destacar que las investigaciones descritas en el artículo de McCulloch y Pitts darían cavidad a dos ramas de estudio diferentes, los procesos biológicos en el cerebro y en la aplicación de las redes neuronales para la inteligencia artificial.

A finales de los 40s, Donald Hebb escribiría el articulo *The Organization of Behavior* [3], el cual expone la idea de que el aprendizaje trata de reforzar los caminos que toman las neuronas a medida que estos se van usando como pasa en el aprendizaje de los seres humanos (cuantas más veces realizas una tarea mejoras los futuros resultados de esta). Este mecanismo de aprendizaje no supervisado pasara a llamarse el **Aprendizaje de Hebb**.

En el año 1958 Frank Rosenblatt creo el **perceptrón**, que sería el precursor a lo que conocemos como neurona artificial. El americano pudo crear con esta tecnología redes de dos capas que mediante adición propagarían datos para solucionar problemas de circuitería y reconocimiento de patrones muy básicos, siendo el primero en toparse con el problema que suponía entrenar a una red para solucionar la disyunción exclusiva o XOR que no sería resuelto hasta 20 años después. Rosenblatt también escribió el libro *Principles of Neurodynamics,* en el 1968 que trataría de los principios básicos de la neuro computación [4].

En el 1960 el profesor universitario Bernard Widrow y su alumno Ted Hoff en Stanford desarrollarían el modelo **ADALINE** (Adaptive Linear Neuron) [5], que tomaría el primer modelo de neurona McCulloch-Pitts como base y lo expandirían añadiendo el sesgo a cada neurona y haciendo que el algoritmo de aprendizaje tuviera en cuenta el grado de corrección de la salida obtenida respecto a la esperada usando el error cuadrático medio como estimador del error.

Durante los últimos años de los 60 y los 70s, la investigación sobre las redes de neuronas artificiales decayó, se perdió el interés en estas tecnologías posiblemente a que no se encontraba la forma de resolver problemas no lineares como la disyunción exclusiva y que los ordenadores de la época no eran lo suficientemente potentes para manejar de forma eficaz el tiempo de ejecución de las redes más grandes. Los investigadores Marvin Minsky y Seymour Papert en 1969 describieron estos problemas en el libro *Perceptrones: Introducción a la Geometría Computacional* [6].

Después de esta oscura etapa en el campo de las redes neuronales, el interés hacia esta tecnología se volvió a encender gracias a el estudiante Paul Werbos, que presento su tesis doctoral en 1974 acerca de la **Retropropagación** [7], algoritmo que hoy en día es usado para entrenar las redes y permitió deshacerse del problema del o-exclusivo y hacer aprender a redes con un número elevado de capas. Aunque esto en aquel momento no supondría un antes y un después para las redes neuronales, el redescubrimiento de este algoritmo en 1986 por David Rumelhart y sus colaboradores en el artículo *Neural Networks and Learning Internal Representation by Error Propagation,* sería un punto clave para el campo [8].

Mientras la retropropagación estaba en un segundo plano, John Hoppfield en el 1982 con su artículo *Neural Networks and Physical Systems with Emergent Collective Computational Abilities* [9], hizo que muchos matematicos, cientificos y tecnológicos se unieran a la investigación y se hiciera más hincapié en el estudio acerca de las redes de neuronas gracias a su idea expuesta en el artículo de hacer que las redes de neuronas pudieran actuar como sistemas de memoria direccionable por contenido.

El Instituto Americano de Física en 1985 realizo el primer congreso de Redes Neuronales para computación, que se ha repetido anualmente hasta día de hoy. En 1987, se formó la conocida Sociedad Internacional de Redes Neuronales (INNS) y posteriormente esta organización creo la revista *Neural Networks*.

En 1992 John Weng, Narenda Ahuja y Tomas Huang de la universidad de Illinois introdujeron la técnica del max-pooling (submuestreo de máximos) para ayudar con el reconocimiento de objetos tridimensionales [10].

Partiendo a la corriente hardware, que se desvía un poco del enfoque que tiene este trabajo. Durante los siguientes 20 años la investigación siguió creciendo de forma exponencial, haciendo que grandes empresas del sector tecnológico-informático como IBM empezaran a desarrollar sus propios chips basados en arquitecturas neuronales. Tomando como hito la creación del procesador TrueNorth en 2014 que asegura tener la estructura similar a la de un cerebro humano, permitiendo realizar entre 4.600 y 400.000 millones de operaciones sinápticas al segundo. Todo esto en un procesador del tamaño de uno comercial estándar.

Todos estos avances han hecho que las mayores empresas tecnológicas empiecen a desarrollar sus propias Inteligencias Artificiales basadas en Redes Neuronales para ofrecer soluciones innovadoras a los usuarios de sus sistemas. En 2010 Apple lanzo Siri, un asistente de búsqueda que reconoce la voz y responde a consultas para sus teléfonos IPhone. Este movimiento hizo que otras empresas desarrollaron sus propios asistentes como Google Assistant de Google, Alexa de Amazon y Cortana de Microsoft ente otros. Mas tarde se empezaron a usar modelos más potentes que permitían crear datos automáticamente a partir de entrenamiento realizado con otros datos, nacieron los Modelos Generativos, entre los que podemos destacar los generadores de texto GPT de OpenAI, Gemini de Google y Copilot de Microsoft, los generadores de imagen DALL-E, Stable-Diffusion y MidJouraney y el en desarrollo modelo de generación de video Sora.

2.3 Modelos más comunes de las Redes de Neuronas

En esta sección se introducirán los modelos más básicos y primitivos que se han mencionado en la sección 2.2, ya que son la base de las primeras implementaciones y experimentos que serán realizados durante el proyecto. Además, aunque no vayan a ser objeto de mucho interés para el TFG, se hablara sobre las redes de neuronas convolucionales dado que si se pudiera seguir con el proyecto seria de interés implementar en un futuro estas.

2.3.1 Perceptrón

El perceptrón es el modelo más simple por imaginar como red neuronal, consta solo con una neurona parecida a la descrita en la Figura 2.1 Representación de una Neurona Artificial. Los primeros diseños de esta neurona no contaban con valor de sesgo (aunque posteriormente, se incluirá) y su función de activación hace que la salida sea binaria, que discrimina datos en dos grupos de forma lineal.

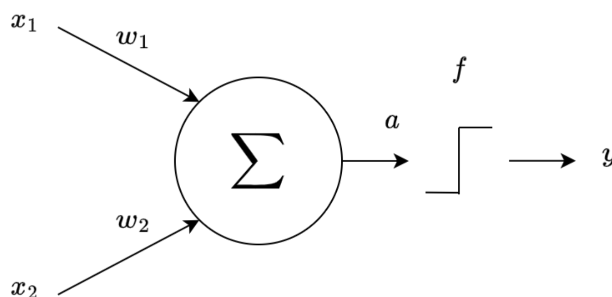


Figura 2.3 Perceptrón simple con dos entradas

$$f(a) = \begin{cases} 1 & si\ a > 0 \\ 0 & si\ a \leq 0 \end{cases}$$

Este modelo permite solucionar problemas sencillos como las puertas lógicas AND, OR y sus respectivas negativas NAND y NOR. Su limitado tamaño, hace que no pueda resolver problemas no lineales como XOR, problema que en un futuro se podría resolver con la combinación de varios perceptrones (perceptrón multicapa), oras funciones de activación y el algoritmo de retropropagación.

2.3.2 ADALINE

Las neuronas lineares adaptativas o ADALINE (su acrónimo en inglés) fueron descritas en un artículo por Bernard Widrow y su alumno Ted Hoff en 1960, dos años después de la creación de los primeros perceptrones. Aunque comparten una gran similitud, estas difieren de los perceptrones en que carecen de una función de activación de escalón, su aprendizaje usa el error medio cuadrático para calcular la cantidad de error y sus salidas no tienen un dominio binario. El uso de estas neuronas permite resolver problemas de regresión linear, pero al igual que los perceptrones, no son capaces de aprender a resolver problemas no lineares.

2.3.3 Perceptrón Multicapa

La combinación de varios perceptrones organizados por capas junto al uso de funciones de activación y el algoritmo de retropropagación nos permite deshacernos de la limitación más grande que tenían los modelos de neuronas más primitivas, la incapacidad de resolver problemas no lineares. El uso de funciones de activación no lineares le proporciona esta cualidad, haciendo que los resultados de las sumas ponderadas en cada neurona se "doblen" para ajustarse a los resultados esperados. Las tres funciones de activación más comunes son la sigmoidal, la tangente hiperbólica y le Unidad Linear Rectificada (ReLU).

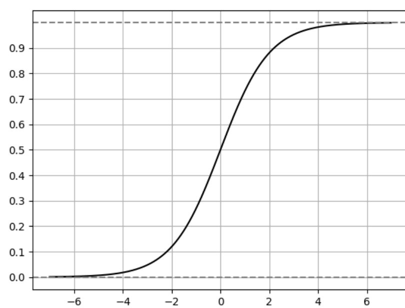


Figura 2.4 Función Sigmoidal

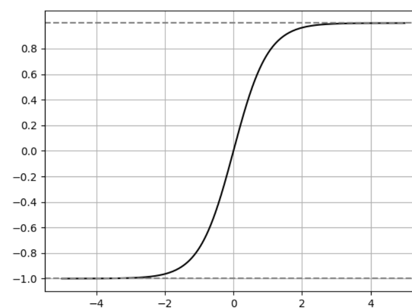


Figura 2.5 Función Tangente Hiperbólica

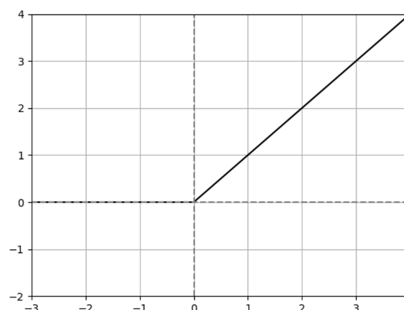


Figura 2.6 Función ReLU

2.3.4 Algoritmo de Retropropagación

La pieza clave que hace que las redes de neuronas florecieran de finales de los 80s en adelante sería el algoritmo de retropropagación, que permite a una red de ilimitadas capas aprender propagando el error de salida calculado con una función de error hasta la capa de entrada, ajustando los pesos y los sesgos cada entrenamiento. Para calcular el ajuste a realizar en cada parámetro, se calcula con descenso de gradiente estocástico o alguna variación de este la diferencia y el salto que debe tomar ese parámetro para optimizar el resultado.

Debido a que se le hará más hincapié al algoritmo en el Capítulo 3, no se hablará en esta sección del funcionamiento del algoritmo en detalle, pero podemos dividir la tarea de aprendizaje en las siguientes:

1. La red neuronal recibe los datos de entrada y los propaga hacia delante calculando los resultados en base a sus pesos, sesgos y la función de activación hasta dar con la salida de la iteración.
2. Con el resultado obtenido calculamos el error entre el valor obtenido y el esperado mediante una función de error $E(y)$, que puede variar según el modelo que se quiera implementar.
3. Mediante el descenso de gradiente se calcula la variación que debe de tomar el valor de cada peso (y sesgo) de la capa de salida, y se propaga el error a la capa anterior.
4. Para todas las capas restantes se ajustan los pesos por descenso de gradiente y se propaga hacia la capa anterior. Así hasta que ya no se pueda propagar más porque se haya llegado a la capa de entrada.

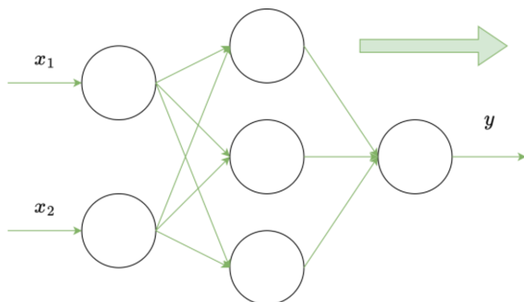


Figura 2.7 Propagación hacia delante

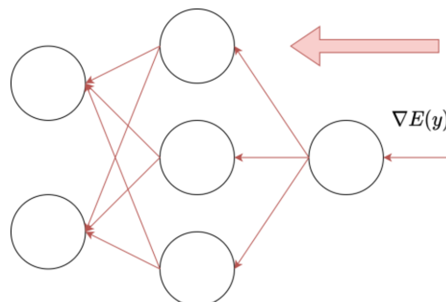


Figura 2.8 Propagación del error hacia atrás

El cálculo del descenso de gradiente puede ser implementado varias formas: La más sencilla es aplicando el método de diferencias finitas, que en el Capítulo 3 veremos cómo, aunque es sencillo de usar, consume más recursos de procesado y el afinado del modelo se ve limitado por el parámetro de salto que se aplique. La siguiente manera, aunque más laboriosa seria hacer análisis matemático de las funciones de error, activación y el sumatorio de entradas para encontrar una definición matemática que calcule la derivada para un punto dado, este método, aunque puede ser más complejo a la hora de diseñar un modelo, nos permite reducir a la mitad el número de operaciones que se tienen que procesar y hace que el modelo se afine con más precisión ya que no se hace uso del parámetro limitante de salto. En esta memoria se recogerán los resultados de usar ambos métodos y la implementación final de la librería permitirá alternar entre ambos.

2.3.5 Redes de Neuronas Convolucionales

Las redes convolucionales fueron introducidas en el año 1980 por Kunhiko Fukushima [11], aunque hasta 1998, que Yann LeCun y compañía introdujeron un método de aprendizaje basado en la retropropagación para entrenar estos modelos [12] no fueron usados. El algoritmo resulta ser excelente para el reconocimiento de objetos y análisis de imágenes en 2 dimensiones ya que se basa en el sistema nervioso de la percepción visual del ser humano. Las capas en vez de ser representadas por filas de neuronas son planos bidimensionales que se interconectan con una reducida parte de la siguiente capa. Este método, podría resultar similar al escalado de pixeles de una imagen, en el que las celdas de un cuadrante de la capa se usan para procesar las entradas de las celdas de un cuadrante en posición similar en la siguiente (véase en la Figura 2.9 Conexión de neuronas en una red convolucional).

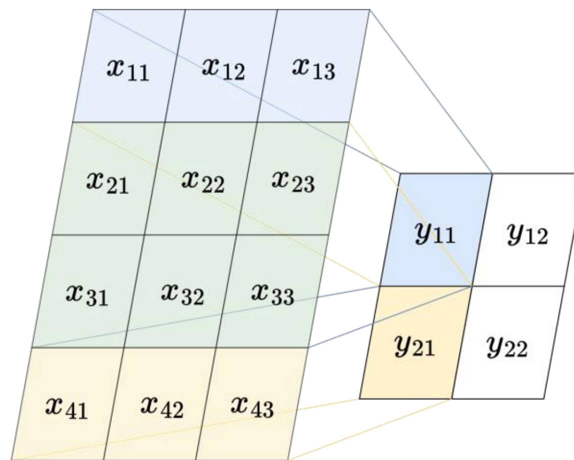


Figura 2.9 Conexión de neuronas en una red convolucional

En la figura observamos como el resultado de las neuronas azules se propagan hacia y_{11} y el de las amarillas a y_{21}, teniendo en cuenta que las neuronas coloreadas de verde son compartidas entre las dos salidas, así observando que cada neurona recibe como entrada el valor dado por la siguiente expresión, siendo f la función de activación, x_{nm} un valor de entrada a la célula, w_{nm} el peso asociado a dicha entrada, b_{ij} el sesgo de la neurona e y_{ij} el resultado del procesado de dicha neurona.

$$y_{ij} = f\left(\sum_{n=i}^{i+2}\sum_{m=j}^{j+2} x_{nm}w_{nm} + b_{ij}\right)$$

Aunque implementar este tipo de redes no se encuentra dentro de los objetivos del proyecto, se considera interesante que en futuras líneas de desarrollo (si hubieran) se considerara la implementación de un método que permitiera a la red establecer capas que usaran convoluciones.

3 Desarrollo de la librería

Este capítulo recoge todos los experimentos realizados durante la realización del TFG, la definición de la librería GMLNN y los detalles del desarrollo de esta. La realización del código del trabajo se puede describir en 3 fases. La primera, en la que se han estudiado los modelos as sencillos e implementado un par sirviendo como base a las futuras implementaciones a realizar en el proyecto, aquí se recogen implementaciones de modelos funcionales con una sola neurona. La segunda fase contempla estudiar al detalle el funcionamiento de la retropropagación y como desarrollar el algoritmo paso a paso, dando lugar a implementaciones de perceptrones multicapa que sean capaces de resolver problemas no lineales. Finalmente, en la tercera fase se desarrolla el objetivo principal del trabajo, una librería que permite crear redes de neuronas y entrenarlas en C.

3.1 Herramientas utilizadas para la realización del trabajo

Usualmente cuando se piensa en soluciones de aprendizaje automático o "Machine Learning", aparecen en nuestras mentes *TensorFlow*, *Keras* y *PyTorch*, todas ellas escritas en su mayoría en Python, que es un lenguaje interpretado y esto podría hacer que los resultados de diferentes algoritmos sean más lentos. En un mundo en el que avanzas cuanto más rápido eres (como lo es en el ML el aprendizaje), interesaría buscar las soluciones más rápidas y eficientes, y que mejor manera de intentar reducir esos tiempos que usando C como lenguaje de implementación, ya que al ser compilado parte con ventaja sobre Python y al ser de nivel más bajo y menos pesado puede ser más rápido. De todas formas, aunque el objetivo de este proyecto no es ser más rápido que lo ya existente, podemos adoptarlo como objetivo secundario.

El entorno de trabajo base que usaremos para el desarrollo es sencillo, una maquina Linux con un compilador C.

Para ser más específicos, se ha usado un MacBook Pro del año 2009 con un procesador Intel Core I5 de 2,4Hz corriendo una versión de Ubuntu LTS 20.04, así aprovechando un ordenador antiguo y dándole una segunda vida, aunque en ocasiones pueda causar errores de rendimiento.

Para la visualización de datos, se usa *gnuplot* y para la generación de figuras y gráficos para la memoria también me apoyo en la librería *matplotlib* de Python y MATLAB ya que quedan resultados más vistosos.

Todo el código usado para producir los siguientes modelos se encuentra en el repositorio de GitHub: https://github.com/itsTwoFive/gml_nn

3.2 Modelos Mononeurona

Para los dos primeros modelos que se implementaron use como referencia los videos del divulgador Tsoding, que tiene una serie de videos publicados en YouTube sobre Machine Learning [13]. También me ayudaron a entender mejor las redes y su funcionamiento los canales de 3blue1brown y Emergent Garden [14][15].

Estos primeros modelos con los que se experimentó son un perceptrón y una neurona lineal adaptativa (ADALINE), que fueron implementados de forma paralela al comienzo del trabajo ya que son modelos muy parecidos. Al tratarse de las primeras propuestas en la historia de las redes de neuronas artificiales, su aplicación estaba limitada a la resolución de problemas lineales con un solo valor de salida.

3.2.1 ADALINE

El modelo de neurona adaptativa linear, en ingles Adaptive Linear Neuron trata de una sola neurona que multiplica n entradas x_i por sus correspondientes pesos w_i y posteriormente las suma junto a un sesgo b que es independiente de las entradas, dando como resultado un único valor numérico y.

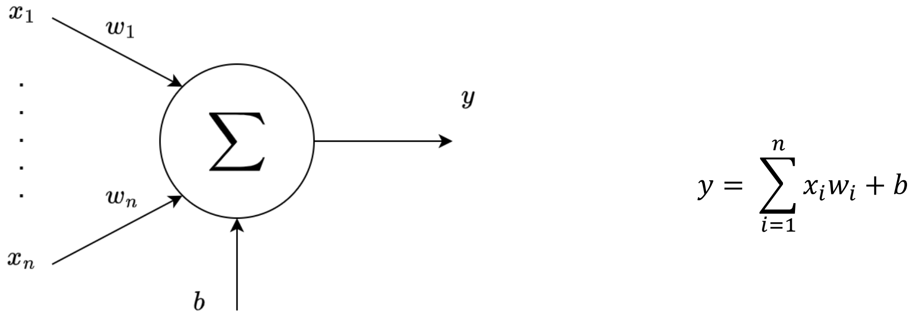


Figura 3.1 Representación de ADALINE

La salida predicha por la anterior expresión y en todos los m casos de entrenamiento es comparada con las esperadas $\hat{y}$ usando el error cuadrático medio para obtener el error de un entrenamiento E.

$$E = \frac{1}{m}\sum_{i=1}^{m}(y_i - \hat{y}_i)^2$$

La forma de hacer que nuestra neurona aprenda y cambie su comportamiento es ajustando los pesos y el sesgo, esto se puede hacer mediante un descenso de gradiente de la propia función de error. Con lo cual se debe calcular para cada w_i su gradiente $\nabla\mathrm{E}(w_i)$ y el gradiente del sesgo $\nabla\mathrm{E}(b)$. Para realizar, el ajuste tendremos que realizar una selección de casos de entrenamiento m que se usaran por etapa o iteración, en el primer modelo del trabajo, ADALINE, se decidió usar $m = 1$, haciendo que la neurona aprenda cada vez que analiza un caso. Este método es llamado método estocástico.

$$\nabla\mathrm{E}(w_i) = \frac{\partial E}{\partial w_i} = \frac{1}{m}\sum_{j=1}^{m} 2\left(y_j - \hat{y}_j\right) x_i \overset{m=1}{\Longrightarrow} \nabla\mathrm{E}(w_i) = 2(y_0 - \hat{y}_0)\, x_i$$

$$\nabla\mathrm{E}(b) = \frac{\partial E}{\partial b} = \frac{1}{m}\sum_{j=1}^{m} 2\left(y_j - \hat{y}_j\right) \overset{m=1}{\Longrightarrow} \nabla\mathrm{E}(b) = 2(y_0 - \hat{y}_0)$$

Lo último que falta para realizar el aprendizaje es establecer una tasa de aprendizaje η que permite controlar cuanto influye el error cometido a la variación de los pesos durante el entrenamiento de la red (en este caso de la neurona). El valor de η suele ser inferior a 0.5, aunque dependiendo del modelo y de los datos usados para entrenarlo puede necesitarse uno más alto. Con todo lo calculado previamente ya se puede reajustar de forma iterativa (siendo c la iteración o época) tanto los peso como el sesgo de nuestra neurona siguiendo las siguientes formulas.

$$w_i^c = w_i^{c-1} + \eta\, \nabla\mathrm{E}\left(w_i^{c-1}\right)$$

$$b^c = b^{c-1} + \eta\, \nabla\mathrm{E}(b^{c-1})$$

Antes de realizar la primera iteración o época de entrenamiento, debemos inicializar los valores de los pesos y el sesgo de forma aleatoria. Aunque con

modelos lineales no es estrictamente necesario, en un futuro permitirá evitar estancamientos en mínimos locales y promueve una mejor generalización de los datos nuevos.

Con la finalidad de hacer más comprensible el entrenamiento descrito, se resume en los siguientes pasos:

1. Se inicializan todos los pesos w_i y el sesgo b de la neurona.
2. Se específica la tasa de aprendizaje η y el numero de iteraciones C que tendrá el entrenamiento.
3. Para cada iteración c hasta llegar a C:
    a. Se calculan los gradientes $\nabla \mathrm{E}(w_i)$ para cada peso w_i y $\nabla \mathrm{E}(b)$ del sesgo b.
    b. Se actualizan los pesos siguiendo $w_i^c = w_i^{c-1} + \eta \nabla \mathrm{E}\left(w_i^{c-1}\right)$
    c. Se actualiza el sesgo siguiendo $b^c = b^{c-1} + \eta \nabla \mathrm{E}(b^{c-1})$

Una vez acabado el entrenamiento, si ha sido satisfactorio, se podría alimentar a la neurona con n datos y obtener un valor y que se ajuste al comportamiento de los casos de entrenamiento.

Se decidió usar este modelo en primer lugar para ajustar puntos en un espacio $\mathbb{R}^2$ a una función lineal que dado el valor de una abscisa la neurona te devolviera su ordenada. Usando puntos que siguen la expresión $f(x) = 3 - 2x$ el modelo se ajusta en pocas iteraciones para devolver las ordenadas de cualquier valor x.

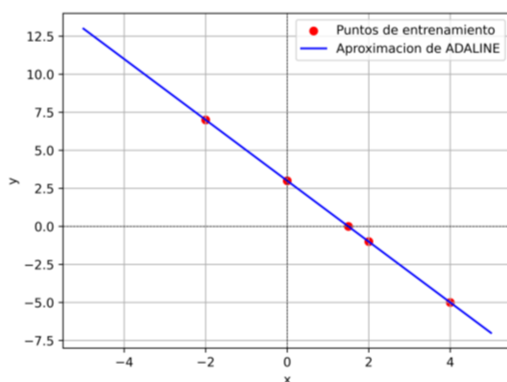


Figura 3.2 Ajuste a la función lineal y = 3 - 2x

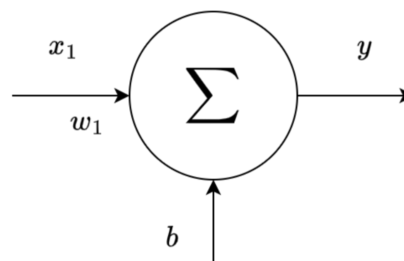


Figura 3.3 ADALINE con una entrada

Del mismo modo se puede ajustar funciones lineales en n dimensiones para obtener el hiperplano que satisfaga a los puntos deseados. Podemos por ejemplo usar una ADALINE para encontrar el plano $y = {x_2}/{4} - {x_1}/{2} + 1$ en un espacio $\mathbb{R}^3$.

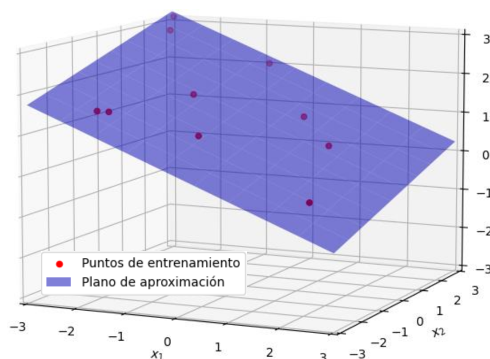


Figura 3.4 Plano generado por el ajuste de ADALINE

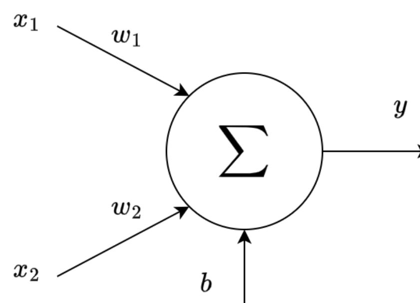


Figura 3.5 ADALINE con dos entradas

El código usado que realiza estas aproximaciones se encuentra en la carpeta del repositorio del proyecto "experimental" como *adaline1.c y adaline2.c*.

3.2.2 Perceptrón

Simultáneamente, mientras se experimentaba con la implementación de ADALINE, se desarrolló un fichero que implementaba el perceptrón de Frank Rosenblat, esta tomaba parte del código de la neurona adaptativa linear, pero con ciertos cambios. Lo primero es que se buscaba que la salida fuera binaria, con lo cual se hará uso de una función de activación o umbral sigmoide que permite acotar el rango de salida de la neurona entre 0 y 1. Además, más adelante se hará uso de estas funciones y era un buen momento para probar su uso. La función sigmoidal sigue la siguiente regla:

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

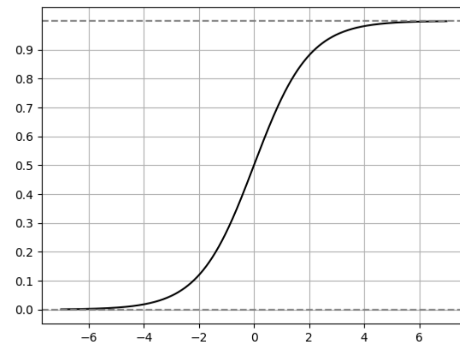


Figura 3.6 Representación gráfica del Sigmoide

Al añadir esta, hay que modificar el comportamiento de la neurona respecto al modelo ADALINE, haciendo que después de la suma ponderada de las entradas se calcule $\sigma(salida)$ así obteniendo:

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

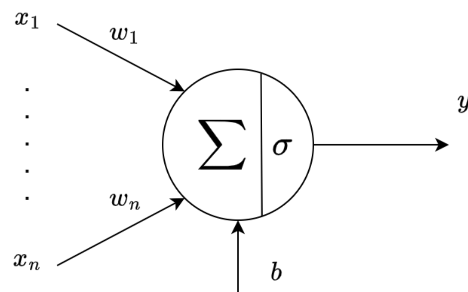


Figura 3.7 Perceptrón

Aunque los primeros modelos de perceptrón solo usaban como error la diferencia simple entre el resultado obtenido y el esperado, en este la función de error sigue siendo la misma y no se ve modificada frente a la implementación de la Sección 3.2.1.

La otra diferencia notable entre los dos modelos reside en el uso del método de diferencias finitas para realizar el ajuste de los pesos w_i y el sesgo b en vez de usar el cálculo de gradiente. El método de diferencias finitas permite calcular la derivada de una función en un cierto punto sin necesidad de realizar el cálculo de derivada analíticamente mediante la diferencia de dos puntos muy cercanos separados por un valor ε. Este método permite encontrar la derivada de forma muy sencilla y de forma generalizada, aunque también resulta en mayor coste computacional.

$$f'(x) = \lim_{\varepsilon \to 0} \frac{f(x+\varepsilon) - f(x)}{\varepsilon}$$

Haciendo uso de la tasa de aprendizaje η, siendo c la iteración actual de entrenamiento y $E'(x)$ la diferencia finita de la función de error cuadrático medio usando esta vez todos los casos de entrenamiento m por época a diferencia de ADALINE, los pesos y el sesgo son calculados de la siguiente forma:

$$E = \frac{1}{m} \sum_{i=1}^{m} (y_i - \hat{y}_i)^2$$

$$w_i^c = w_i^{c-1} + \eta \, \mathrm{E}'(w_i^{c-1})$$

$$b^c = b^{c-1} + \eta \, \mathrm{E}'(b^{c-1})$$

Igual que en ADALINE, antes de la primera época o iteración de entrenamiento se deben inicializar los pesos y el sesgo de forma aleatoria.

Este modelo podría ser usado para la clasificación binaria entre dos conjuntos de datos separados linealmente. En el fichero *perceptrón.c* de la carpeta "experimental" del repositorio, encontramos la implementación del perceptrón realizada, que permite resolver problemas de clasificación como el siguiente de forma sencilla:

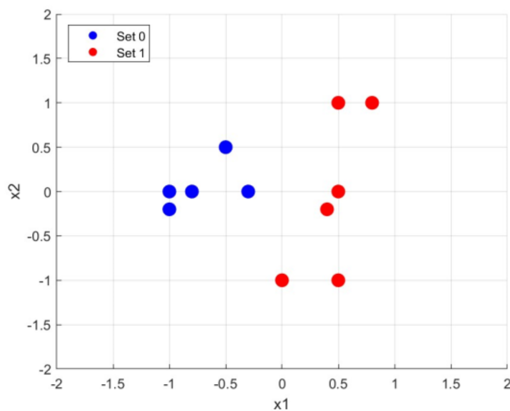


Figura 3.8 Conjunto de puntos a clasificar por el perceptrón

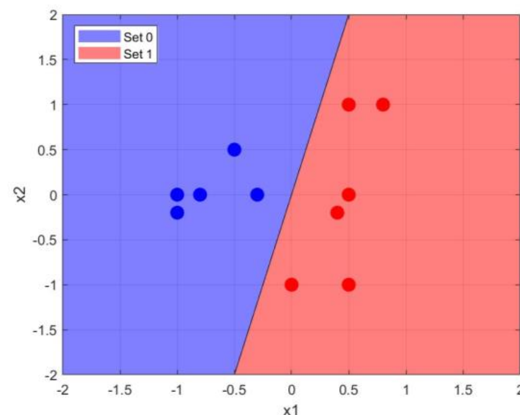


Figura 3.9 Conjunto de puntos clasificados por el perceptrón

También este mismo se puede usar para ajustar el modelo a una puerta lógica lineal alimentándolo con una tabla de verdad como entrada de datos. Probando con AND, OR y sus inversas podemos observar que en pocas épocas con los parámetros correctos el perceptrón ajusta sus pesos y sesgo para copiar el funcionamiento de las puertas lógicas. Intentando aprender más puertas lógicas nos topamos con que es imposible aproximar la puerta XOR, ya que se trata de una función no lineal y con un solo perceptrón no es posible trabajar con problemas no lineales.

3.3 Código Auxiliar

Con el fin de hacer más sencilla la programación de los siguientes modelos y sobre todo de la librería, se decide trabajar en diferentes módulos auxiliares que facilitan la escritura de código y permiten reducir el tamaño final del código la librería. Además, estos módulos pueden ser usadas por separado para futuros trabajos y mejoras.

3.3.1 Matrices – matrix.h

Usar matrices para representar y trabajar con los pesos de las redes neuronales proporciona un mayor grado de control y sencillez a la hora de realizar operaciones que involucren los pesos de las redes de neuronas artificiales (esto se verá en la sección 3.4). Pese a que existen propuestas que permiten el uso de matrices de forma sencilla e incluso se pueden usar arrays de arrays en C para representar y trabajar con matrices (esto puede ser un poco más lioso), decidí crear una estructura de datos y sus respectivas funciones que implementaran la funcionalidad de una matriz y las operaciones matriciales necesarias para el código a desarrollar de aquí en adelante. La cabecera de este código recibe el nombre de "matrix.h" y su implementación se encuentra en "matrix.c".

Estos ficheros definen el tipo *matrix*, que es un struct con los siguientes campos:

```
typedef struct {
    int cols;
    int rows;
    double *d;
} matrix;
```

Código 3.1 Estructura del tipo matrix

Estas matrices antes de usarse deben ser previamente inicializadas con el método `mat_alloc(rows,cols)` y permiten realizar operaciones matriciales como suma, resta, multiplicación escalar y multiplicación matricial, entre varias otras. La estructura es sencilla, cuenta con dos parámetros enteros `cols` y `rows` que determinan el tamaño de la matriz y un puntero a una zona de memoria `*d` que se reserva de forma dinámica dependiendo de los parámetros anteriores en el montículo de la memoria de nuestra máquina. Es importante una vez acabado el uso de cualquier matriz generada en el código liberar su espacio. Como ya se ha mencionado, esta hace uso de la memoria dinámica de la máquina, que puede producir un desbordamiento de memoria si se instancia un número muy grande de estructuras tipo matriz y no se liberan las innecesarias. Para hacer esto basta con usar `mat_free(*mat)`. Toda la descripción de este código se encuentra en el Anexo A – Manual de uso de matrix.h.

Cabe recalcar que en el repositorio del trabajo se encuentran dos versiones de este código ya que ha sufrido cambios notorios durante el desarrollo. Ambas comparten tanto el nombre de la cabecera como de la implementación, pero la primera versión de este se encuentra en la carpeta "experimental" y la más nueva, que cuenta con funcionalidad más amplia y extendida, que resuelve ciertas fallas de la anterior en la carpeta principal de Código "src".

3.3.2 Manejador de datos – data_handler.h

Este trozo de código se encarga de leer archivos `.csv` y transformar su contenido a una estructura `parser_result`, esta está formada por un array de arrays de tipo doble coma flotante `data_input`, que contiene las entradas de los casos de prueba, otro array de arrays `data_output` que contiene las salidas de dichos casos en el mismo orden, una lista con los nombres de los atributos descritos en el fichero y tres enteros `num_case`, `num_in` y `num_out` que especifican el número de casos leídos, el número de entradas que tiene cada caso y sus salidas.

```
typedef struct{
    double** data_input;
    double** data_output;
    char** atrib_names;
    int num_case;
    int num_in;
    int num_out;
```

```
} parser_result;
```
Código 3.2 Estructura del tipo parser_result

La cabecera data_handler.h incluye varios métodos que permiten manejar y transformar los datos, estos son implementados por "data_handler.c". Aunque en Anexo B – Manual de uso de data_handler.h se encuentra toda la información respecto a métodos, existen tres que ahorran trabajo a la hora de preprocesar los datos para hacerlos legibles y que se puedan usar junto a la librería:

Método **`parse_data(char [] filename, int num_inputs)`**, que abre el fichero con ruta `filename` y genera un tipo `parser_result` conteniendo en si los para cada caso una lista *n* de atributos establecidos por el parámetro `num_inputs` y de salidas, siendo las columnas restantes del fichero. Es importante tener en cuenta que este método devuelve todos los casos en un fichero en el mismo orden en el que están presentados en el archivo `.csv`.

Método **`change_all_values_for(double **data, int size_x, int size_y, float actual, float new)`**, que permite cambiar dentro de un array de arrays de números en coma flotante doble todas las instancias que hay del número `actual` a `new`. Al lidiar con números en coma flotante, podemos obtener errores de redondeo, que pueden hacer que la maquina no identifique un numero como el que creemos que es, por esto se opta por usar una macro `EPSILON_DISTANCE` con el valor máximo de diferencia que puede haber entre dos números para ser considerados iguales.

Método **`data_div(parser_result in, int div_size)`**, que devuelve la división de los datos guardados en `in` a dos partes, una de entrenamiento y otra de prueba. Estos dos están contenidos en un array de tipo `parser_result` con un ancho de 2, el tamaño de las partes depende del parámetro `div_size`, que es el número de casos que queremos conservar en el primer set. Este método aparte de la división de los datos debería implementar un método de barajado de datos para evitar que la red de neuronas aprenda patrones no deseados relacionados con el orden de los datos. Durante el desarrollo, tuve en cuenta varios diferentes, pero resultaban tener una complejidad computacional muy alta y en casos con un set de datos de tamaño elevado era un problema. Así que buscando por internet tope con el que sería el algoritmo que finalmente fue implementado. El algoritmo de permutación de Fisher-Yates [16]. Este tiene una complejidad lineal y permite trabajar con este número de casos elevados que tienen algunos sets de datos.

3.4 Redes Multicapa Sencillas y Retropropagación

Una vez entendidas las bases de las neuronas artificiales, es necesario aprender a conectarlas y hacer que estas aprendan. En esta sección, se discutirá de los conceptos básicos de la alimentación de las redes artificiales, la conexión entre las neuronas que las componen y de cómo hacer que aprendan de su error de forma efectiva. También se buscará acabar con la limitación de resolución únicamente de problemas lineales como XOR.

Quiero hacer mención especial al libro de Pedro Isasi Viñuela y Inés M. Galván, *Redes de Neuronas Artificiales: Un Enfoque Practico* [17], que ha sido de gran utilidad a la hora de entender el algoritmo de retropropagación descrito más adelante en la Subsección 3.4.2.

3.4.1 Perceptrón multicapa

Es el momento de evolucionar, nuestros seres unicelulares por si solos no acaban siendo nada más que simples sumatorios ponderados. El primer paso para crear una red ya está dado, se tiene el funcionamiento de una neurona

desarrollado, en esta subsección conectaremos varias y usaremos las salidas de unas como entradas de otras para crear estructuras más complejas.

En esta subsección exploraremos la forma de crear nuestra primera red de neuronas artificiales compuesta por perceptrones distribuidos en diferentes capas. Estas redes suelen adoptar el nombre de perceptrón multicapa o MLP, siglas del inglés “multi-layer perceptron” y son el tipo de red que puede ser generada por la librería a desarrollar. Claramente, en toda la Sección 3.4 como en la 3.3 se ceñirá todo diseño a mantenerse lo más simple y sencillo posible, las redes creadas durante esta experimentación previa al desarrollo de la librería son sujeto de cambio y se verán modificadas respecto al código de la librería.

El primer intento de implementación de red fue llamado 3neurons.c. Como todo código anterior, este se encuentra en el directorio “experimental” del repositorio del trabajo. La red cuenta con una estructura de dos capas: la primera con dos neuronas y la segunda con una única. La red está completamente conectada ya que todas las neuronas de una capa pasan información a todas las de la siguiente. Usa la función sigmoidal como función de activación para todas las capas y funciona con dos valores de entrada. El aprendizaje este modelo se basa en el método de diferencias finitas como anteriormente hicimos con el perceptrón. Esta forma de entrenamiento resulto estar planteado de manera errónea (ya que no se propagaba el error de manera efectiva hacia atrás) y no era más efectivo que usar un solo perceptrón. De todas formas, nos servirá como punto de partida para experimentar con las estructuras multicapa. La red sigue la siguiente estructura:

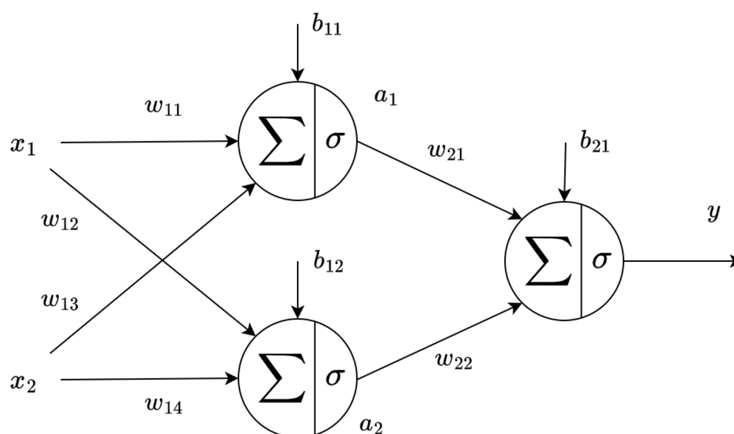


Figura 3.10 Red neuronal 3neurons

Podemos comprobar en la Figura 3.10 que ambos valores x_1 y x_2 actúan como valores de entrada en las neuronas de la primera capa, estas realizan la suma ponderada junto a el sesgo correspondiente y posteriormente el resultado es pasado por la función de activación para obtener los valoras a_1 y a_2, que serán usados como valores de entrada para la neurona de la capa de salida multiplicándose así por w_{21} y w_{22} respectivamente, siendo sumados junto al sesgo de la neurona y posteriormente realizado la operación sigmoidal sobre el resultado para dar con el valor final de la alimentación de la red.

Respecto al código, la idea original era usar estructuras llamadas `Neuron` que contuvieran el valor y el sesgo de los pesos. Así por cada neurona en el sistema que fuera instanciada una estructura del siguiente tipo.

```
typedef struct {
    float w1;
```

```
    float w2;
    float bias;
} Neuron;
```
Código 3.3 Estructura del tipo preliminar Neuron

Esa solución, aunque sea la más entendible por alguien que ve el código por primera vez, no resulta muy practica a la hora de trabajar con redes más grandes por varios motivos:

1. La estructura cuenta con un numero de valores limitados, que bien se podría modificar para que mediante memoria dinámica se instancien el numero necesario de pesos, pero requeriría realizar una asignación de memoria por cada neurona creada.
2. Se debería crear otra estructura que permitiera organizar las neuronas en capas y como el tamaño depende del uso que se le quiera dar, deberíamos trabajar con más asignaciones de memoria por cada capa.
3. Al tener tantas estructuras dentro de otras, escribir el código puede ser algo tedioso y se podrían desencadenar errores difíciles de resolver a la hora de trabajar en el desarrollo de la librería.

Otra manera, que resulta mejor y resuelve la forma de crear una estructura de datos que permita el comportamiento de las redes y sea menos costosa es mediante el uso de matrices como estructura para guardar los pesos y operar sobre estos. De esta forma no se debe instanciar ninguna estructura que represente una sola neurona, sino que se crean las capas como unidad mínima dentro del código. Aunque el uso de matrices parezca complejo para este tipo de problemas, es muy conveniente ya que la propagación hacia delante se puede describir como una multiplicación matricial precedida de una suma. Por ejemplo, en la red de dos capas mostrada previamente si quisiéramos calcular la salida de la capa de entrada $[a_1, a_2]$ deberíamos hacer lo siguiente:

$$a_1 = \sigma(x_1 w_{11} + x_2 w_{13} + b_{11})$$

$$a_2 = \sigma(x_1 w_{12} + x_2 w_{14} + b_{12})$$

Pero creando una matriz W_1que contenga todos los pesos w_{ij} ordenados de forma que las neuronas estén representadas por columnas y las filas sean del número de entradas.

$$W_1 = \begin{bmatrix} w_{11} & w_{12} \\ w_{13} & w_{14} \end{bmatrix}$$

Después, introduciendo los valores de sesgo b_{ij} en otra matriz B_1 y las entradas x_i en una matriz X para poder operar con la matriz W_1.

$$B_1 = \begin{bmatrix} b_{11} & b_{12} \end{bmatrix} \qquad X = \begin{bmatrix} x_1 & x_2 \end{bmatrix}$$

Se puede realizar la operación de alimentación hacia delante de la capa de forma sencilla independientemente del tamaño de la red siguiendo la siguiente ecuación:

$$A = f\ (X \cdot W + B)$$

Para simplificar la demostración se va a usar $\hat{A}$ como la matriz del resultado del sumatorio previo sin haber pasado por la función de activación:

$$\hat{A} = \begin{bmatrix} \hat{a}_1 & \hat{a}_2 \end{bmatrix}$$

$$\hat{a}_1 = x_1 w_{11} + x_2 w_{13} + b_{11}$$

$$\hat{a}_2 = x_1 w_{12} + x_2 w_{14} + b_{12}$$

Que siguiendo el caso de red de neuronas expuesto anteriormente la primera capa podría ser resuelta como:

$$\hat{A} = [\hat{a}_1 \quad \hat{a}_2] = X \cdot W_1 + B_1$$
$$= [x_1 \quad x_2] \cdot \begin{bmatrix} w_{11} & w_{12} \\ w_{13} & w_{14} \end{bmatrix} + [b_{11} \quad b_{12}]$$
$$= [x_1 w_{11} + x_2 w_{13} \quad x_1 w_{12} + x_2 w_{14}] + [b_{11} \quad b_{12}]$$
$$= [x_1 w_{11} + x_2 w_{13} + b_{11} \quad x_1 w_{12} + x_2 w_{14} + b_{12}]$$
$$= [\hat{a}_1 \quad \hat{a}_2]$$

Claramente para obtener el resultado final de la capa A, se debe usar como parámetro de entrada de la función de activación cada uno de los valores almacenados en la matriz $\hat{A}$. En este caso como (por ahora) estamos trabajando tan solo con el sigmoide deberíamos realizar la siguiente operación antes de avanzar al cálculo de la siguiente capa.

$$A = \sigma(\hat{A}) = [\sigma(\hat{a}_1) \quad \sigma(\hat{a}_2)]$$

Para la capa de salida se debe realizar la siguiente operación:

$$Y = \sigma(A \cdot W_2 + B_2) = \sigma\left([a_1 \quad a_2] \cdot \begin{bmatrix} w_{21} \\ w_{22} \end{bmatrix} + [b_{21}]\right) = \sigma([a_1 w_{21} + a_2 w_{22} + b_{21}])$$

Siguiendo estas normas podemos definir finalmente los pesos de una capa como una matriz con un numero de columnas igual a neuronas contenidas en esta y filas igual al número de entradas que recibe de otra capa o del exterior. También las capas deben contar con una matriz del número de columnas igual a las neuronas y 1 sola fila para guardar y operar con los valores del sesgo. Aun así, en un la Sección 3.5 se modificará esto para facilitar aún más la forma de operar y no tener que lidiar con una matriz para los sesgos.

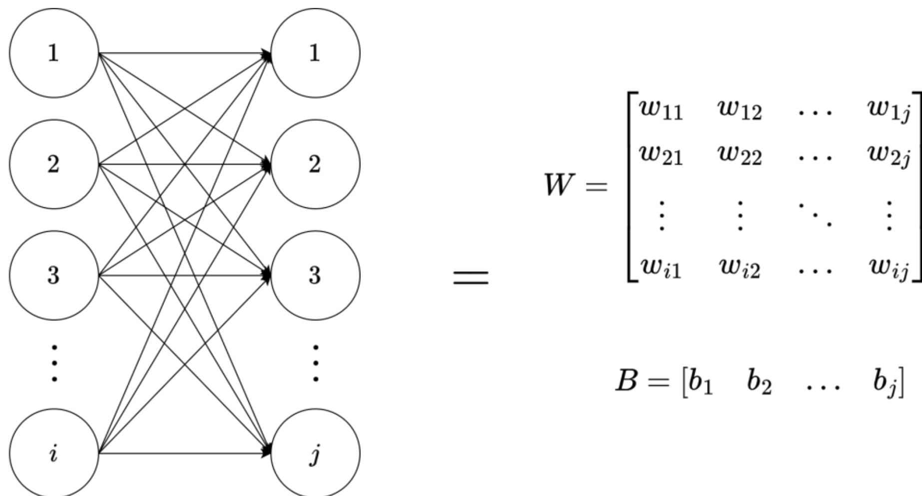


Figura 3.11 Pesos y sesgo de una capa en forma de matriz

Ahora, sabiendo que el uso de matrices puede facilitar en gran medida la implementación de los siguientes modelos, se decide implementar la primera versión de la librería descrita en la Subsección 3.3.1, matrix.h, que permite la creación de matrices de n x m tamaños y realización de operaciones matriciales como la suma, resta, multiplicación matricial, multiplicación escalar… Toda la funcionalidad de la librería en Anexo A – Manual de uso de matrix.h.

Con el código auxiliar que permite operar con matrices se presentó el siguiente fichero en que se trabajo fue "matrixbasednn.c", que implementa tan solo la

operación de propagación hacia delante o “feedforward” sin ningún tipo de aprendizaje, ya que lo que se buscaba era comprobar el correcto funcionamiento de las matrices y el uso de estas en nuestras redes.

```
matrix layer_forward(matrix inputs, matrix weights, matrix biases)
{
    matrix multiply_sol;
    mat_productf(inputs,weights,&multiply_sol);
    matrix sum_sol;
    mat_sumf(multiply_sol,biases,&sum_sol);
    for (int j = 0; j < sum_sol.cols; j++)
    {
        float *value = mat_seek(sum_sol,0,j);
        *value = sigmoid(*value);
    }
    mat_free(&multiply_sol);
    return sum_sol;
}
```

Código 3.4 Implementación de feed-forward usando matrices

Para asegurarme que el modelo actúa como debería, realice un entrenamiento con el modelo “3neurons.c” de una función lineal (ya que en este punto del proyecto no se ha dado con la forma de resolver estos problemas) y copie los pesos tras el entrenamiento al nuevo modelo que usa las matrices para ejecutarlo. Al comprobar que los resultados eran exactamente los mismos se decidió no implementar más funcionalidad a este último código ya que para el correcto funcionamiento de los perceptrones multicapa o MLP se necesitaba implementar la retropropagación y no se había comenzado ello en ese momento.

3.4.2 Retropropagación

La retropropagación o “Backpropagation” en inglés, es el algoritmo que permitirá entrenar a las redes de neuronas generadas por el código de la librería. Se trata de un algoritmo de aprendizaje supervisado, por lo que se debería seguir trabajando con los pares de conjuntos de datos de entrenamiento (entrada y salidas esperadas). El algoritmo calcula el error de cada salida mediante una función de error y un descenso de gradiente que busca la convergencia en el mínimo de los resultados de esta función. El nombre de retropropagación es dado por la forma en la que la red propaga el error desde las ultimas capas hasta la primera pasando por todas las capas ocultas (de forma contraria a la que realiza la operación “feedforward”).

El algoritmo se usa de forma iterativa por etapas comúnmente conocidas como épocas o “epochs” en inglés, en cada etapa evalúa una función de error E que dado N casos de entrenamientos calcula como de correctas son las salidas de la red y respecto a los datos esperados $\hat{y}$. Esta función puede variar dependiendo del modelo con el que se quiera trabajar, pero una de las más usadas, sobre todo en redes sencillas es el error cuadrático medio usado en las Subsecciones 3.2.1 y 3.2.2. A diferencia de las secciones anteriores vamos a tomar $e(n)$ como nuestra función de error cometido en el caso n y E como el error medio de todos los caso.

$$E = \frac{1}{N}\sum_{n=1}^{N} e(n)$$

Como se ha dicho, la red busca minimizar este error hasta llevarlo a converger con el mínimo mediante la modificación de los pesos y el sesgo de cada neurona,

lo cual nos lleva a usar el método de descenso de gradiente en cada época g con una tasa de aprendizaje η para actualizar estos:

$$w(g) = w(g-1) + \eta\nabla\mathrm{E}(w) = w(g-1) + \eta\frac{\partial E}{\partial w}$$

Como se cuenta con redes de neuronas organizadas por capas a la hora de aplicar el descenso de gradiente hay que encontrar la forma de trabajar con todas las neuronas de cada capa de forma repetitiva, esta regla se llama regla delta generalizada y tendremos que encontrar una con casos diferentes, primero para la capa de salida y después las demás capas ocultas ya que las reglas usadas para el cálculo de afinación de los pesos difieren entre las capas ocultas y la capa de salida. Estas reglas varían entre modelos, ya que dependen de la función de activación y error de cada red.

En esta subsección se fabricará un modelo multicapa para comprobar el funcionamiento de la retropropagación, este trabajará sobre dos capas (sin contar con la de entrada), la función sigmoidal como función de activación y el error cuadrático medio como función de error. La red solo tendrá una neurona en la capa de salida y un numero n de neuronas en la oculta, ya que se usan matrices para guardar los pesos puede variar el número de neuronas en esta capa.

Como el objetivo es minimizar el error mediante el ajuste de los pesos y el sesgo usando el descenso de gradiente, debemos encontrar ∇w y ∇b para cada peso y sesgo respectivamente.

$$\nabla\mathrm{E}(w) = \frac{\partial E}{\partial w} \qquad \nabla\mathrm{E}(b) = \frac{\partial E}{\partial b}$$

La retropropagación se hace de atrás hacia delante, primero debemos encontrar el primer caso de la regla delta sobre la capa de salida. Así, debemos evaluar la derivada de la función de error para encontrar el gradiente necesario para cada peso de la capa de salida C. Siendo i la neurona éxodo de la que se parte y j la destino que recibe el valor, N el número de muestras y n la muestra sobre la que se actúa:

$$\frac{\partial E}{\partial w_{ij}^{C-1}} = \frac{\partial}{\partial w_{ij}^{C-1}}\left(\frac{1}{N}\sum_{n=1}^{N} e(n)\right) = \frac{1}{N}\sum_{n=1}^{N}\frac{\partial e(n)}{\partial w_{ij}^{C-1}}$$

Usando la función de error:

$$e(n) = (y(n) - \hat{y}(n))^2$$

Evaluamos la derivada de $e(n)$ respecto a w_{ij}^{C-1}:

$$\frac{\partial e(n)}{\partial w_{ij}^{C-1}} = 2\big(y_j(n) - \hat{y}_j(n)\big)\frac{\partial y_j(n)}{\partial w_{ij}^{C-1}}$$

Aplicando la regla de la cadena, después de calcular la derivada de la función de error deberíamos calcular la del valor de salida y, que resulta ser nuestra función de activación, en este caso, siendo l_c el ancho (número de neuronas) de la capa c:

$$\frac{\partial y_j(n)}{\partial w_{ij}^{C-1}} = \frac{\partial}{\partial w_{ij}^{C-1}}\sigma\left(\sum_{i=1}^{l_{C-1}} w_{ij}^{C-1}a_i^{C-1} + b_j^C\right)$$

Necesitamos realizar el cálculo de la derivada de la función sigmoidal respecto a un peso, recordando la expresión que sigue esta:

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

Volviendo a aplicar la regla de la cadena podemos obtener la siguiente ecuación:

$$\frac{\partial y_j(n)}{\partial w_{ij}^{C-1}} = \sigma\left(h_j^{C-1}(n)\right)\left(1-\sigma\left(h_j^{C-1}(n)\right)\right)\frac{\partial h_j^{C-1}(n)}{\partial w_{ij}^{C-1}}$$

Siendo $h_j^{C-1}(n)$ el contenido dentro del sigmoidal:

$$h_j^C(n) = \sum_{i=1}^{l_{C-1}} w_{ij}^{C-1} a_i^{C-1}(n) + b_j^C$$

Evaluándolo y poniéndolo en común obtenemos la siguiente expresión:

$$\frac{\partial e(n)}{\partial w_{ij}^{C-1}} = 2\big(y_j(n) - \hat{y}_j(n)\big)\,\sigma\left(h_j^{C-1}(n)\right)\left(1-\sigma\left(h_j^{C-1}(n)\right)\right)a_i^{C-1}(n)$$

Podemos simplificar el uso de esta definiendo el termino común para cada neurona destino δ_j^C:

$$\delta_j^C(n) = 2\big(y_j(n) - \hat{y}_j(n)\big)\,\sigma\left(h_j^{C-1}(n)\right)\left(1-\sigma\left(h_j^{C-1}(n)\right)\right)$$

Quedando como expresión de la derivada del error e por caso n respecto a w_{ij}^{C-1}:

$$\frac{\partial e(n)}{\partial w_{ij}^{C-1}} = \delta_j^C(n)\, a_i^{C-1}(n)$$

Siguiendo los mismos pasos para calcular el gradiente del sesgo $\nabla \mathrm{E}\big(b_j^C\big)$:

$$\frac{\partial e(n)}{\partial b_j^C} = \delta_j^C(n)$$

Estas expresiones permiten calcular los gradientes en base a los parámetros de la capa de salida, pero no de las ocultas. Debemos encontrar un procedimiento que permita dar con la regla delta desde el caso de la capa oculta. Fijándonos y comparando los dos casos podemos notar cambios respecto a la forma de operar con la capa de salida, ya que cierto peso w_{ij} solo influenciaba a la salida de la neurona j y ahora influye a todas las salidas.

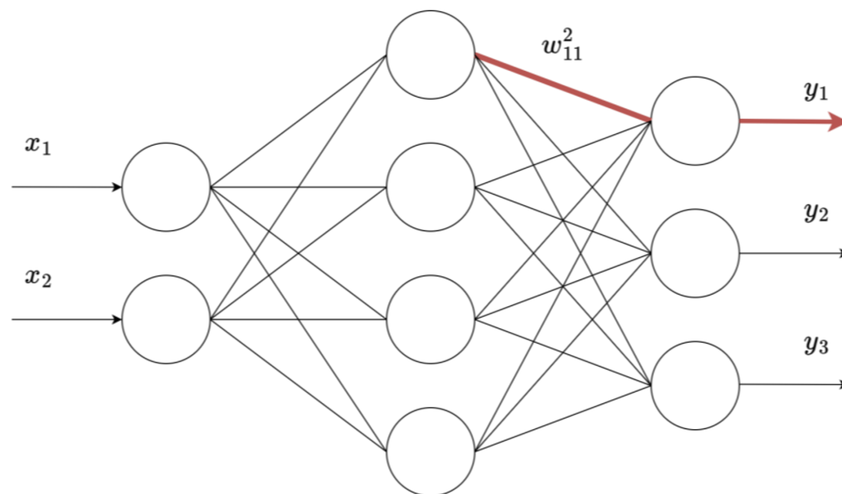


Figura 3.12 Influencia de un peso en la última capa a la red

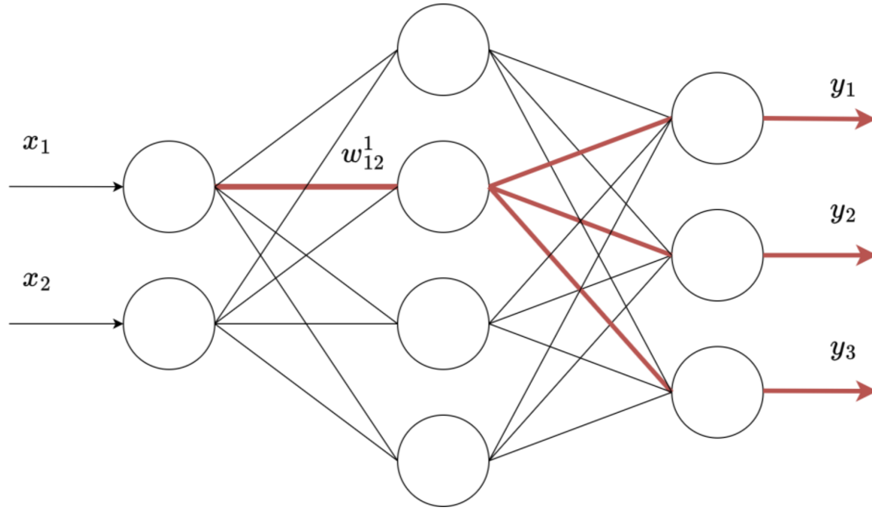


Figura 3.13 Influencia de un peso en una capa oculta a la red

Esto hace que debamos tener en cuenta en la derivada todas las salidas de las capas de la red desde de la capa siguiente a la que se está operando $c+1$ hasta la $C-1$. Importante es tener en cuenta que en las capas ocultas ya no se trabaja con las salidas de la propia red, sino con las salidas de las neuronas de la capa, por lo que se cambia de la variable y a a cuando nos referimos a la salida de una neurona que no esté en la última capa. Siguiendo un procedimiento similar al caso anterior, podemos encontrar la solución a este caso. Usamos el iterador k que indica de que neurona de la capa $c+1$ es cierta salida a_k actuando sobre la capa c obtenemos:

$$\frac{\partial e(n)}{\partial w_{ij}^{c-1}} = \sum_{k=1}^{l_{c-1}} 2(y_k(n) - \hat{y}_k(n)) \frac{\partial y_k(n)}{\partial w_{ij}^{c-1}}$$

Para calcular la derivada de la salida $y_k(n)$ respecto al peso w_{ij}^{c-1} debemos tener en cuenta que el peso influye en la activación de la neurona k de la capa c pero el resto no, por lo que:

$$\frac{\partial y_k(n)}{\partial w_{ij}^{C-1}} = \frac{\partial}{\partial w_{ij}^{C-1}} \sigma \left(\sum_{i=1}^{l_{C-1}} w_{jk}^{C-1} a_j^{C-1} + b_k^C \right)$$

$$\frac{\partial y_k(n)}{\partial w_{ij}^{c-1}} = \sigma\left(h_k^{c-1}(n)\right)\left(1 - \sigma\left(h_k^{c-1}(n)\right)\right) \frac{\partial h_k^{c-1}(n)}{\partial w_{ij}^{c-1}}$$

Calculando la derivada para $h_k^{c-1}(n)$ respecto a w_{ij}^{c-1}:

$$\frac{\partial h_k^{c-1}(n)}{\partial w_{ij}^{c-1}} = w_{jk}^c \frac{\partial a_j^c(n)}{\partial w_{ij}^{c-1}}$$

Sustituyendo en la derivada de $e(n)$ respecto a w_{ij}^{C-1}:

$$\frac{\partial e(n)}{\partial w_{ij}^{c-1}} = \sum_{k=1}^{l_{c-1}} \left(2(y_k(n) - \hat{y}_k(n)) \sigma\left(h_k^{c-1}(n)\right)\left(1 - \sigma\left(h_k^{c-1}(n)\right)\right) w_{jk}^c \frac{\partial a_j^c(n)}{\partial w_{ij}^{c-1}} \right)$$

Encontramos que el termino común $\delta_k^c(n)$ se da dentro del sumatorio, por lo que podemos reducir el tamaño de la regla de la siguiente forma:

$$\frac{\partial e(n)}{\partial w_{ij}^{c-1}} = \sum_{k=1}^{l_{c-1}} \delta_k^c(n) w_{jk}^c \frac{\partial a_j^c(n)}{\partial w_{ij}^{c-1}}$$

Ahora solo nos quedaría encontrar el valor de la derivada de $a_j^c(n)$ para dar con el segundo caso de la regla delta generalizada. Sabiendo:

$$a_j^c(n) = \sigma\left(\sum_{i=1}^{l_{c-1}} w_{ij}^{c-1} a_i^{c-1}(n) + b_j^c\right)$$

Ahora si calculando la derivada de esta:

$$\frac{\partial a_j^c(n)}{\partial w_{ij}^{c-1}} = \sigma\left(h_j^{c-1}(n)\right)\left(1 - \sigma\left(h_j^{c-1}(n)\right)\right) a_i^{c-1}(n)$$

Y sustituyendo en $e(n)$:

$$\frac{\partial e(n)}{\partial w_{ij}^{c-1}} = \sigma\left(h_j^{c-1}(n)\right)\left(1 - \sigma\left(h_j^{c-1}(n)\right)\right) a_i^{c-1}(n) \sum_{k=1}^{l_{c-1}} \delta_k^c(n) w_{jk}^c$$

Ahora podemos definir el valor δ para las neuronas de las capas ocultas, que es similar al de la capa de salida, pero sin la influencia de la función de error y añadiendo el sumatorio de valores δ de las capas previas:

$$\delta_j^c(n) = \sigma\left(h_j^{c-1}(n)\right)\left(1 - \sigma\left(h_j^{c-1}(n)\right)\right) \sum_{k=1}^{l_{c-1}} \delta_k^c(n) w_{jk}^c$$

Finalmente obtenemos la expresión que define la derivada de $e(n)$ respecto a w_{ij}^{c-1} en el segundo caso de la regla:

$$\frac{\partial e(n)}{\partial w_{ij}^{c-1}} = \delta_j^c(n)\, a_i^{c-1}(n)$$

En caso de que deseemos trabajar con el sesgo b_j, es tan fácil como igualar el valor de $a_i^{c-1}(n) = 1$ y operar como si el sesgo fuera otro peso más. Esto se usará más adelante en la implementación de la librería. Los ajustes de los pesos vienen dados al igual de la misma forma que lo hacían en la capa de salida. Para la una capa c, si se quiere calcular el nuevo valor de los pesos y los sesgos en una generación g debemos aplicar las siguiente expresiones:

$$w_{ij}^{c-1}(g) = w_{ij}^{c-1}(g-1) + \eta \frac{\partial E(g)}{\partial w_{ij}^c}$$

$$b_j^c(g) = b_j^c(g-1) + \eta \frac{\partial E(g)}{\partial b_j^c}$$

Aunque todos estos cálculos parecen complejos, en la práctica tan solo basta con aplicar bien la regla de la cadena para averiguar la derivada del error por caso e respecto a un peso w. Simplificándose a la siguiente regla de la cadena:

$$\frac{\partial e}{\partial w} = \frac{\partial e}{\partial y} \times \frac{\partial y}{\partial h} \times \frac{\partial h}{\partial w}$$

El código “bp_matrix_experiment.c” dentro de la carpeta experimental que se encuentra en repositorio del proyecto es la implementación del primer modelo que consigue usar la retropropagación en el proyecto. Usando este código se consigue acabar con la linealidad a la hora de resolver problemas, ahora gracias

a la retropropagación y el uso de funciones de activación no lineares, la red puede resolver problemas no lineares como el XOR. De hecho, el primer caso de entrenamiento con el que se ha probado esta red ha sido con la clasificación de la tabla de verdad de la función lógica XOR usando 2 neuronas en la capa oculta.

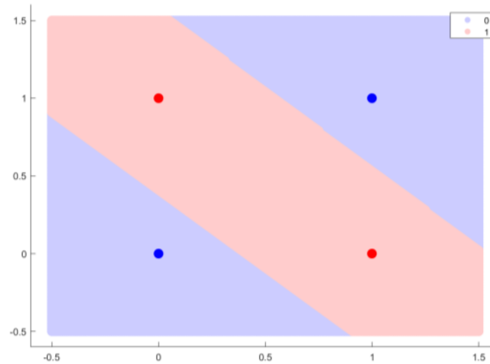


Figura 3.14 Clasificación de los puntos de la tabla de verdad de XOR

Las Figuras 3.14, 3.15 y 3.16 son generadas usando el set de puntos que guarda el script "bp_matrix_experiment.c" tras su ejecución en un archivo .csv, este contiene los resultados de la red tras el entrenamiento para múltiples puntos cercanos entre sí en el rango de dominio que queramos mostrar, después, se usa MATLAB para "plotear" los puntos en el archivo .csv. Probando con otros sets de puntos un poco más complejos y una red que cuenta en la capa oculta con 4 neuronas obtenemos:

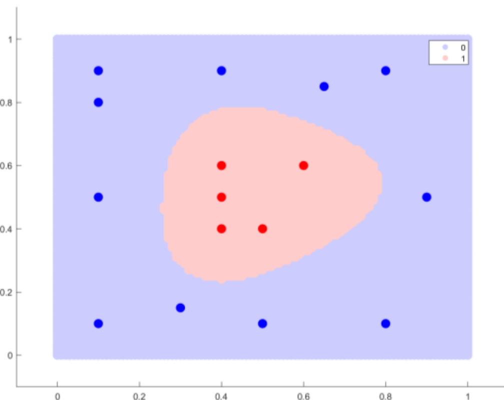


Figura 3.15 Clasificación de set de puntos en dos clases

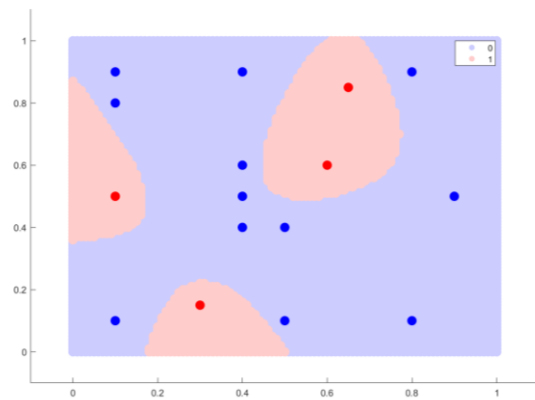


Figura 3.16 Clasificación de set de puntos en dos clases

Podemos comprobar como la red se va ajustando para encontrar en que regiones del plano bidimensional se encuentran los puntos de cada clase. Estos ejemplos junto al previo de la puerta lógica XOR son muy sencillos, pero demuestran que el algoritmo de retropropagación implementado es correcto y se puede seguir experimentando con modelos más complejos y grandes.

3.4.3 Optimizadores

Una vez teniendo claros los conceptos detrás de la retropropagación y cómo aplicarlos para realizar el aprendizaje supervisado de nuestra red de neuronas artificiales, tenemos casi todo lo necesario para implementar la librería. Tan solo falta hablar de los optimizadores o algoritmos de optimización. Estos son una parte clave de las redes y son los encargados de hacer que la red minimizara la función de pérdida o error cometido. Hasta ahora, en el proyecto solo se ha trabajado con el descenso de gradiente por lotes (todo el set de datos por lote) a

excepción de en ADALINE que se trabajó con el descenso de gradiente estocástico o SGD usando las siglas en ingles que usaba solo un caso por época. Siguiendo con la búsqueda en internet, tope con varios tipos de optimizadores que dotan a las redes de distintas características al minimizar el error cometido. Decidí probar a extender el código de la Subsección 3.4.2 e implementar uno de los más sencillos que existen, el método del momento o de la bola pesada [18].

Antes de entrar en detalle del optimizador con momento, quiero especificar el funcionamiento del descenso de gradiente para poder comparar los optimizadores y comprender las diferencias entre ellos. Para expresarlos de forma gráfica, podemos tomar la función de error como una superficie con picos y hoyos, el valor del peso como una bola en esa superficie, el descenso de gradiente es la dirección en la que la gravedad empuja a nuestra bola para trasladarla al punto más bajo cercano, que automáticamente busca hacer que el objeto llegue al mínimo de la superficie.

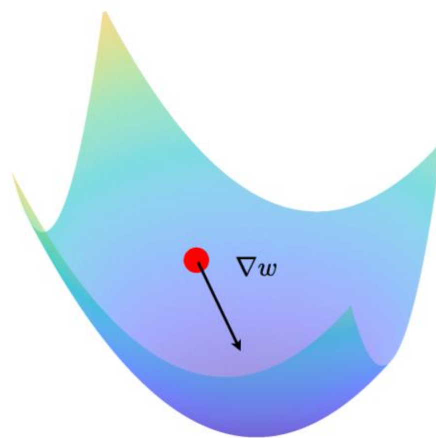


Figura 3.17 Descenso de gradiente en un espacio en tres dimensiones

Usando esta dirección que nos otorga el gradiente con nuestra tasa de aprendizaje η podemos mover la bola a través del espacio hasta llegar al final del hoyo, a esto se le llama convergencia. Es importante establecer un valor adecuado a η ya que, si se establece uno erróneo, el ajuste será incorrecto y podría suponer que la eficiencia del algoritmo baje o incluso que dispare a nuestra “bola” lejos del hoyo al que está cayendo. Si el valor de η es muy bajo, la red convergerá lentamente y causará problemas de rendimiento. Sin embargo, si el valor es demasiado elevado causara el efecto contrario a la convergencia, la divergencia y hará que la red “olvide” lo aprendido.

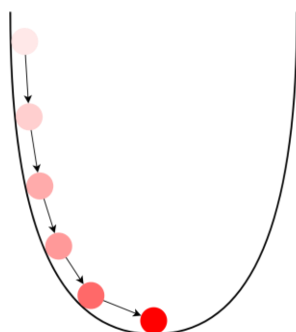


Figura 3.18 Convergencia de la función de pérdida

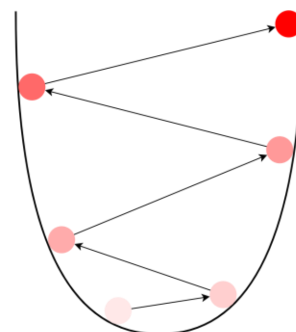


Figura 3.19 Divergencia de la función de pérdida

Cuando se aplica de forma correcta este método puede parecer perfecto y poco costoso, pero existen casos comunes que hacen que la red de neuronas no aprenda de forma correcta. El primer caso es, si nuestro peso se encuentra en una zona llana o con muy poco desplome, este apenas se moverá en cada entrenamiento, haciendo que tome muchas etapas para apenas reducir el error de la red o incluso provocando un estancamiento inesperado. El segundo caso es que las funciones de error no suelen tener solo un mínimo y que, aunque el gradiente este acercándonos a converger con uno de ellos, no garantiza que este sea el mínimo global o absoluto pudiendo hacer que nuestro peso converja en un mínimo local y sufrir un estancamiento.

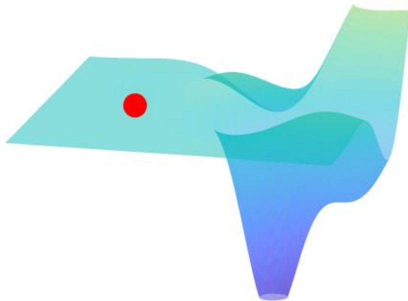


Figura 3.20 Estancamiento en zona llana de la función de perdida

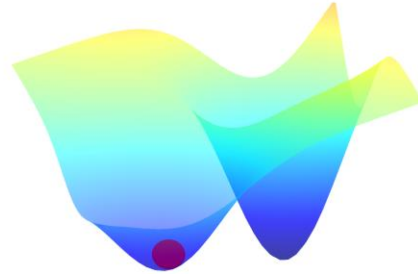


Figura 3.21 Convergencia en mínimo local de la función de perdida

Siguiendo con el supuesto de bola que cae por gravedad hacia los mínimos de la función de perdida, como nuestro descenso de gradiente calcula la siguiente posición teniendo en cuenta tan solo el gradiente y la posición actual, realmente no acelera a medida que cae como pasaría en el mundo real, sino que saltaría poco a poco hasta dar con un gradiente en la dirección contraria que le haga saltar de vuelta. Esto se soluciona añadiéndole momento a la bola (como sucede en la realidad). Nuestra bola debe de ir cogiendo velocidad a medida que cae empujada por la gravedad (gradiente), así permitiendo que llegue a la convergencia más rápido cuando tiene que bajar una rampa poco pronunciada ya que empezaría a "rodar" acelerando. Esto podría solucionar los problemas de estancamiento vistos anteriormente, haciendo que cuando llegue a una zona plana, la bola siga moviéndose en la dirección en la que se movía anteriormente y que, en caso de pasar por un mínimo local, la bola pueda pasar a través de este y dar con otro mínimo más pronunciado.

Si recordamos las fórmulas del optimizador de descenso de gradiente:

$$w_i = w_{i-1} + \eta \, \nabla \mathrm{E}(w_{i-1})$$

$$b_i = b_{i-1} + \eta \, \nabla \mathrm{E}(b_{i-1})$$

Podemos variarlas para hacer que actúen de forma que usen el momento:

$$w_i = w_{i-1} + \Delta w_i \qquad b_i = b_{i-1} + \Delta b_i$$

$$\Delta w_i = \Delta w_{i-1} - \eta \, \nabla \mathrm{E}(w_{i-1}) \qquad \Delta b = \Delta b_{i-1} - \eta \, \nabla \mathrm{E}(b_{i-1})$$

Con estas ecuaciones añadimos el momento a nuestro descenso de gradiente, pero no tenemos forma de hacer que se pare la "bola" ya que no existe ningún tipo de fricción que haga que frene ni que disminuya la velocidad. Esto se soluciona añadiendo a nuestra ecuación la tasa de decadencia β que describe la cantidad de velocidad que pierde el descenso cada vez que se produce una iteración. De tal forma, nos quedamos con las siguientes expresiones:

$$w_i = w_{i-1} + \Delta w_i \qquad\qquad b_i = b_{i-1} + \Delta b_i$$

$$\Delta w_i = \beta \Delta w_{i-1} - \eta\, \nabla \mathrm{E}(w_{i-1}) \qquad\qquad \Delta b = \beta \Delta b_{i-1} - \eta\, \nabla \mathrm{E}(b_{i-1})$$

Aunque este método soluciona ciertos inconvenientes del descenso de gradiente común, también puede provocar otros como hacer que la convergencia sea más lenta debido a revotes entre las paredes de un mínimo.

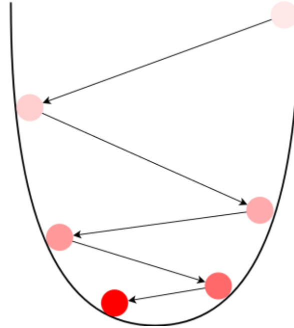


Figura 3.22 Rebotes durante la convergencia

Este método es usado en el programa escrito dentro de la carpeta "experimental" del repositorio del trabajo bajo el nombre de "bp_matrix_momentum.c". Trata de ser una ampliación del código "bp_matrix_experiment.c" utilizado en la Subsección 3.4.2 con el añadido del momento. Comparando el entrenamiento entre el modelo sin momento y el modelo con momento con $\beta = 0.5$ sobre los mismo datos, la misma tasa de aprendizaje η y los mismos pesos iniciales podemos comprobar que el uso del momento hace que el entrenamiento converja en la mitad de tiempo.

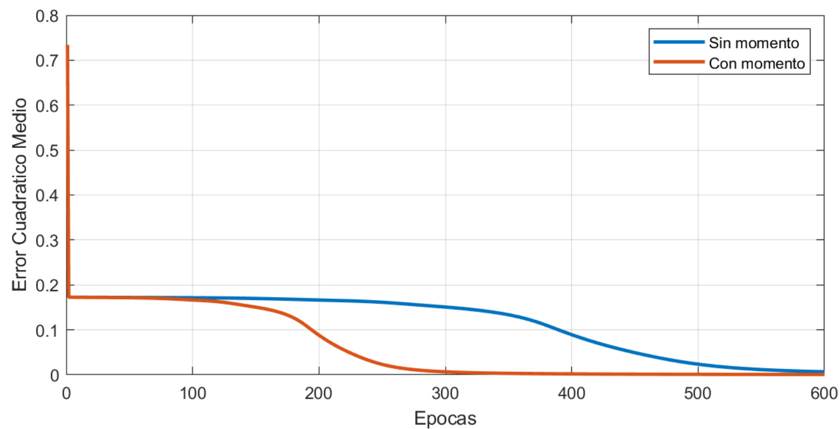


Figura 3.23 Comparación de tiempo de entrenamiento entre un optimizador con descenso y uno sin descenso

También los algoritmos de optimización pueden variar el tamaño de los datos recogidos por cada época, haciendo que existan 3 tipos de optimización dependiendo del número de datos usado:

- Optimizadores **estocásticos** – Que usan un solo caso de entrenamiento por cada época. Estos hacen que el entrenamiento sea más rápido, pero también producen casos de sobreentrenamiento que pueden hacer que solo funcione correctamente la red con los datos usados para entrenarlo. Este tipo de optimizador se utiliza cuando se trabajan con redes y conjuntos de datos muy grandes y se busca un entrenamiento rápido. Otra ventaja de estos es que permiten hacer seguimiento de la actualización de los pesos de forma más sencilla ya que se puede saber qué caso ha hecho que la red se ajuste a un estado nuevo.

- Optimizadores de **batch** o **lote** – Estos son los que hemos usado en la mayoría de los modelos implementados hasta el momento. Toman todos los casos de prueba del conjunto de datos y lo usan para entrenar a la red en cada entrenamiento. La ventaja principal frente a los estocásticos es que producen menos casos de sobreentrenamiento ya que al usar muchos más casos, estos permiten generalizar más los resultados. En contrapartida, estos optimizadores tienen tiempos de entrenamientos N veces superior (siendo N el número de casos en un set) y en conjuntos muy grandes los errores se pueden suavizar y provocar fallos en el funcionamiento de la red ya entrenada.
- Optimizadores **mini-batch** o **mini-lotes** – Estos juntan ambos tipos para crear un punto medio en el que no se gaste tanto tiempo en realizar un entrenamiento y que los resultados postentrenamiento sean coherentes ya que generaliza los datos. También otra ventaja es que al poder cambiar el tamaño del batch, podemos ajustar un mismo modelo a distintos tamaños para ver cual obtiene mejores resultados. Este es el más común debido a la versatilidad que presenta a la hora de ajustar las épocas de entrenamiento.

A demás de estos dos tipos de optimizadores existen gran variedad que son hoy en di preferiblemente usados antes que los básicos ya mencionados. Entre los más conocidos se encuentran: AdaGrad, RMSProp y Adam. Durante lo que queda de trabajo no se hablara de estos, pero podrían ser parte de una nueva versión del código si se quisiera ampliar.

3.4.4 Refactorización de red

Como punto final a esta Sección 3.4 quiero hacer un breve inciso en el último código realizado en la carpeta experimental “nngen_pre.c”, trata de una versión refactorizada del código encontrado en “bp_matrix_momentum.c” que trata usa la versión final de “matrix.h” ya que se encontró un error a la hora de asignar y liberar espacio dinámico a las matrices y era necesario un cambio en el código de esta.

También este código elimina las matrices que contienen los valores de sesgo b de las neuronas, ya que estos pueden ser tratados como un peso (en el ámbito del proyecto w_{N+1}, siendo N el número de entradas), que se multiplica por una entrada que siempre es 1. De esta forma ahorramos en tamaño de código y el número de operaciones que se deben realizar ya que los sesgos y los pesos son almacenados en una sola matriz.

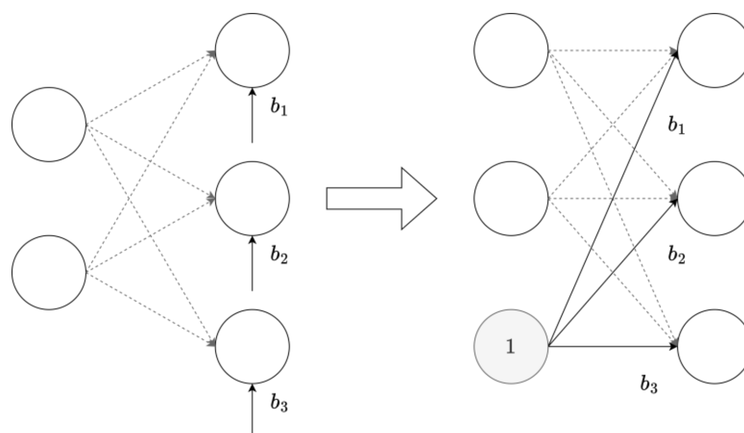


Figura 3.24 Reajuste en la forma de operar con los sesgos

Esta forma de trabajar con los sesgos y pesos en una sola estructura es muy común en el mundo del aprendizaje automático. Ajustando el cambio a las matrices usadas obtenemos matrices con la siguiente forma:

$$W = \begin{bmatrix} w_{11} & \cdots & w_{n1} \\ \vdots & \ddots & \vdots \\ w_{1m} & \cdots & w_{nm} \\ b_1 & \cdots & b_n \end{bmatrix}$$

Y por supuesto las matrices que contienen los resultados de la capa anterior deben ser modificados también para que se pueda hacer la multiplicación con la modificación a las matrices W:

$$A = [\, a_1 \quad \cdots \quad a_n \quad 1]$$

Estos cambios junto a otras heurísticas aplicadas al nuevo código no solo mejoran el rendimiento de la red, sino que también reducen el tamaño de este en 60 LOC.

3.5 Librería GMLNN

Esta es la última fase del proyecto se desarrollará el objetivo principal de este. Se pretende crear desde cero (solo usando C moderno estándar) una librería que permita la creación, entrenamiento y uso de redes de neuronas artificiales. En concreto se ha diseñado para que las redes creadas sean perceptrones multicapa que realizan entrenamientos supervisados mediante el algoritmo de retropropagación. Se partirá del código "nngen_pre.c" ya que es el más avanzado dentro de los experimentos realizados. Respecto al nombre escogido para esta se ha decidido llamarla **gml_nn** tomando las primeras consonantes de mis apellidos seguido de mi nombre y finalmente nn de "neural network".

Esta librería hace uso del código auxiliar descrito en la Sección 3.3, especialmente de "matrix.h", ya que se usa para realizar el "feed-forward" de cada capa. Sin embargo, aunque también se proporcione el código manejador de datos "data_handler.h", este no es necesario para el funcionamiento de la librería, pero sí que es recomendable usarlo junto a esta a la hora de introducir datos para entrenar a las redes.

Se discutirá en diferentes subsecciones el diseño y el trabajo realizado sobre las partes de la librería que permiten el creado y entrenamiento de redes neuronales con en pocas líneas de código. También se ha optado por crear un manual de uso de la librería, que puede ser encontrado tanto en GitHub como en el final del documento en Anexo C – Manual de la librería gml_nn.h.

3.5.1 Estructuras de datos y Parámetros de estas

Anteriormente se han realizado scripts o programas que implementaban redes que tan solo usaban tamaños fijos para el numero de capas y las neuronas que se encuentran en ellas. Cambiar el tamaño de la red o el ancho de una capa en estas resultaba en la creación de un script con cientos de líneas de código diferentes y precedía a una pérdida de tiempo si el modelo creado no se ajustaba a las necesidades de los datos con los que se quiere operar. El primer objetivo marcado para realizar la librería es encontrar una forma de poder guardar datos en el entorno de ejecución de forma que se pudiera trabajar con redes de cualquier tipo de dimensiones.

En la Subsección 3.4.1, se intentaba implementar una estructura de datos que almacenara los parámetros necesarios para una sola neurona. Como ya se discutió en aquel punto, aunque esta forma de estructurar nuestros datos facilite el entendimiento del código, esto no resultaría ser muy apropiado por varias razones de rendimiento. Tras los experimentos realizados a posteriori de

la Subsección 3.4.1, se ha decidido que la forma más conveniente para trabajar con los datos dividiéndolos por capas. Así que se decide implementar un tipo `layer` que define una estructura de datos la cual contiene punteros a la matriz de pesos, otras matrices necesarias para el cálculo de gradiente y una matriz vector en la que se guaran las salidas de la capa. También incluye información acerca de la función de activación que se aplica al final de cada neurona (de esto se hablara en la siguiente Subsección) y el ancho de la capa.

```
typedef struct {
    int layer_width;
    int act_func;
    double (*c_act_func)(double);
    double alpha_rate;
    matrix *W;
    matrix *oW;
    matrix *dW;
    matrix *vW;
    matrix *cW;
    matrix *out;
} layer;
```

Código 3.5 Estructura de datos de capa "layer"

Como se puede observar en el trozo de Código 3.5, la capa usa cinco matrices apuntadas por sus correspondientes punteros para almacenar y calcular el ajuste de los pesos, estas tienen todas las mismas dimensiones ya que todas almacenan información sobre los pesos de la red.

- **`W`** – Como se podría esperar, almacena el valor actual de los pesos y sesgos de la capa, sería el equivalente a la matriz usada en todo el documento como W.
- **`dW`** – Guarda los valores del gradiente de la función de error E respecto a los respectivos w_{ij} contenidos en W o `W` en el código.
- **`vW`** – Es la matriz que se usa para representar el momento a la hora de trabajar con un optimizador que usa el método de la bola pesada o momento.
- **`cW`** – Es una matriz usada para el cálculo intermedio matricial durante el entrenamiento, inicialmente `cW` no iba a ser incluida en la estructura, pero como era necesario el uso de una matriz intermedia y en cada ajuste de pesos se debe usar una, se opta por reservar espacio para esta inicialmente de forma única y sobrescribir los datos de esta para no tener que realizar en cada iteración ambas operaciones de reservado y liberado de memoria dinámica, que causarían un letargo en el entrenamiento de la red.
- **`oW`** – Guarda los valores de las llamadas funciones δ en la especificación de la retropropagación en la Subsección 3.4.2.

Los demás campos de forma intuitiva se puede entender su uso y por qué han sido seleccionados para formar parte de la estructura, aun así, **`layer_width`** indica el número de neuronas en la capa, también llamado ancho de capa. El puntero a la matriz **`out`** guarda los valores de salida de la alimentación de la capa. **`act_func`**, **`c_act_func`** y **`alpha_rate`** se usan para seleccionar la función de activación de las neuronas en la capa, permitiendo al programador definir diferentes funciones a cada capa e incluir funciones personalizadas (más información de estas características en la Subsección 3.5.2).

Existe una segunda estructura de datos definida en la cabecera de la librería llamad `data` la cual fue agregada como parte de la función de entrenamiento

sencillo a ultimas instancias del desarrollo, por ahora no se hablará de esta hasta la Subsección 3.5.7.

Una vez las capas son descritas y definidas por el código, necesitamos organizar estas de forma que puedan usarse entre ellas de forma sencilla y genérica permitiendo usar tantas capas como la memoria de la maquina donde se ejecute el código soportase. Al igual que con el tipo `layer`, se decide crear una definición de tipo `neural_net` a una estructura que guarde toda la información de la red que se quiere crear.

```
typedef struct{
    int input_count;
    int err_func;
    double (*c_err_func)(double,double);
    int layer_count;
    double learning_rate;
    double decay_rate;
    double epsilon_rate;
    int rand_seed;
    int batch_size;
    int cost_output;
    int console_out;
    layer **layers;
    data *dataset;
}neural_net;
```

Código 3.6 Estructura de datos de red neuronal "neural_net"

Esta estructura contiene toda la información y parámetros que la red necesita para funcionar. Por ahora los parámetros que parten de las bases obtenidas en las Secciones anteriores e influyen en la estructura de la red de neuronas son:

- **`input_count`** – Número de entradas que acepta la red, específicamente la capa de entrada.
- **`layer_count`** – Que guarda la cuenta de cuantas capas forman la red de neuronas.
- **`learning_rate`** – Como su nombre en inglés indica, es la tasa o ratio de aprendizaje η, que se aplica al ajuste de todos los pesos de la red.
- **`decay_rate`** – Que es la tasa de decadencia usada en el método de la bola pesada al aplicar momento al optimizador. Si el valor de esta es 0, la red actuara como si el algoritmo de optimización no contara con momento.
- **`rand_seed`** – Semilla usada para la inicialización pseudoaleatoria de los pesos y sesgos de la red. Aunque pueda parecer innecesario este es realmente clave para la volver a reproducir casos de entrenamiento sobre una red y reproducir experimentos anteriores de forma completamente exacta. Si el valor de este entero se establece a 0, la semilla se asignará de forma automática dependiendo de la fecha y la hora de la máquina.
- **`batch_size`** – Determina el tamaño del lote de entrenamiento. Por defecto este es del tamaño del set de datos de entrenamiento. Se encontrará más información de este en la Subsección 3.5.4.
- El doble puntero **`layers`** – Guarda en orden la posición en memoria de las capas `layer` de la red de neuronas.

Los parámetros **`epsilon_rate`**, **`err_func`** y **`c_err_func`**, se usan para establecer la función de error del entrenamiento de la red. El uso en detalle de estos se especifica en la Subsección 3.5.4. Al igual que el uso de **`cost_output`**, **`console_out`** y el puntero **`dataset`** se especificarán en la Subsección 3.5.7.

Hasta ahora durante los experimentos se ha usado la capa y los valores de entrada de forma un poco ambigua, lo normal en el uso de redes es tomar los valores de entrada como la salida de la capa de entrada, distinguiendo el uso entre esta y las ocultas. Teóricamente ambas formas, tanto el uso como el no uso de esta capa es correcto y en ciertos libros y artículos tratan a la entrada de información de diferentes maneras. En el código implementado se trata a los "inputs" como la salida de una primera capa de entrada de la forma mostrada en la Figura 3.25.

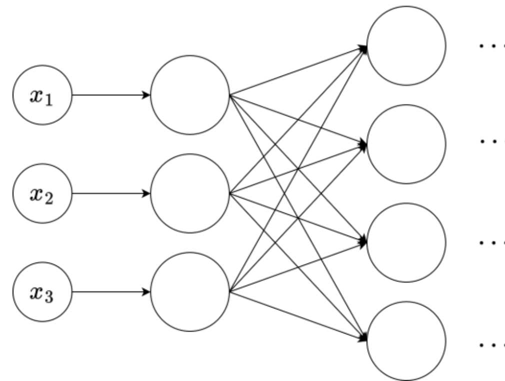


Figura 3.25 Entrada de datos a la red de neuronas

Para hacer que cada entrada funcione como una neurona, pero solo este conectada con una neurona de la siguiente capa, se hace uso de una matriz de pesos especial, que no se ajusta durante el entrenamiento. Esta es una matriz de identidad de tamaño $N_{entradas} \times N_{entradas}$ ya que a la hora de multiplicar una matriz de identidad por la entrada de datos devuelve la misma entrada. En el caso de la red mostrada en la Figura 3.25, la matriz seria:

$$W = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Una vez diseñada la forma de almacenar la información sobre las redes y sus neuronas, se implementa la forma de crearlas con el método `nn_create`, este es un constructor que crea un tipo `neural_net` y las capas contenidas en esta descritas por un array de enteros que debe ser usado como entrada de la rutina entre otros parámetros. Por ejemplo, si al método se le entrega el siguiente vector de enteros [4 8 6 4] generará la siguiente red:

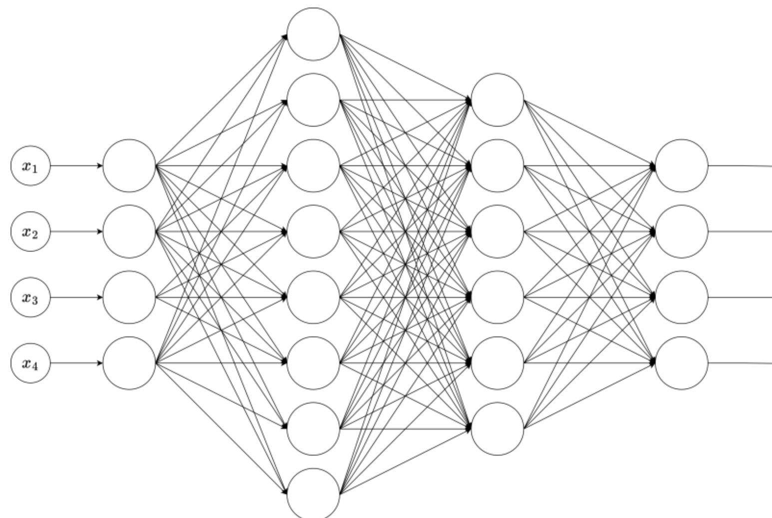


Figura 3.26 Red de neuronas con dos capas ocultas con anchos de capa [4 8 6 4]

3.5.2 Funciones de Activación

En las anteriores secciones del Capítulo 3, se ha hablado por encima sobre las funciones de activación y solo se ha trabajado con la función sigmoidal, pero en realidad existen varias funciones y dependiendo de los datos a usar en la red, unas pueden ser más adecuadas que otras.

Las funciones de activación permiten modelar la salida de los datos de cada neurona de la red otorgando a la red diferentes cualidades dependiendo de los resultados que se esperen. La primera ventaja obtenida al usar estas es que permiten es la transformación de salidas lineales a no lineales mediante el "doblado" del espacio donde se encuentran los datos obtenidos en cada capa. Esta propiedad viene dada al usar funciones no lineales como función de activación. La segunda ventaja que nos pueden proporcionar estas funciones es la posibilidad de acotar la salida de una red a un rango o imagen deseado. Por ejemplo, si queremos que la salida de una capa sea binaria (0 o 1) podemos usar un discriminador binario como función de activación.

En el desarrollo de la librería se ha buscado implementar varias funciones de activación comunes usadas en modelos de aprendizaje automático y permitir a cada capa que actúe con una diferente si se desea. Mediante el uso de la función `layer_set_act_func` se puede establecer a cada red la función deseada usando una de las siguientes macros que se definen en la cabecera de la librería:

`ACT_NONE` – Indica que no se desea usar ninguna función de activación y la salida de la neurona es el resultado del sumatorio de esta. Es lo equivalente a usar una función $f(x) = x$, resulta carecer de utilidad en la mayoría de los casos ya que no aporta ningún tipo de ventaja a la red. Al usar una capa extra de entrada, y no querer modificar los valores de entrada, es utilizada en esta.

`ACT_SIGMOID` – Es la ya nombrada anteriormente sigmoide o logística σ, usada en los experimentos realizados previamente a la implementación de la librería, esta discrimina todos los valores en un rango de $(0, 1)$, con parte de la función siendo lineal cuando los valores de x están alrededor de 0 y dos asíntotas horizontales en 0 y 1, haciendo que los valores positivos más elevados se aproximen a 1 y los negativos a 0.

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

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

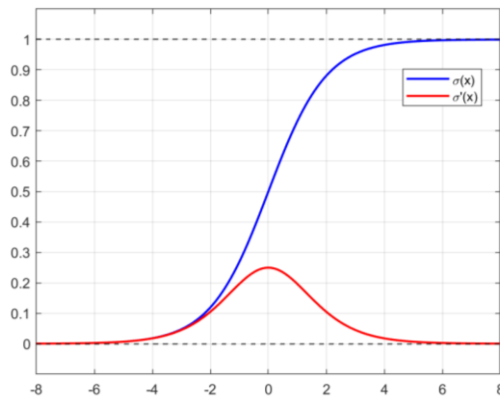


Figura 3.27 Grafica de la función sigmoidal y su respectiva derivada

Esta función es un excelente discriminador binario y es muy común sobre todo en los modelos más sencillos que esperan este tipo de salidas. Pero esta al carecer de valores negativos y no estar centrada en 0, puede conllevar a ciertos fallos de optimización y rendimiento en el entrenamiento de la red. Para solucionar estos problemas se recomienda usar la función tangente hiperbólica.

ACT_TANH – La tangente hiperbólica o *tanh* en notación matemática, es otro tipo de función sigmoidal, está a diferencia del sigmoide estándar, si contempla valores negativos en sus resultados ya que su imagen es $(-1,1)$ y se encuentra centrada en el $(0,0)$.

$$\tanh(x) = \frac{\sinh(x)}{\cosh(x)} = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

$$\tanh'(x) = 1 - \tanh^2(x)$$

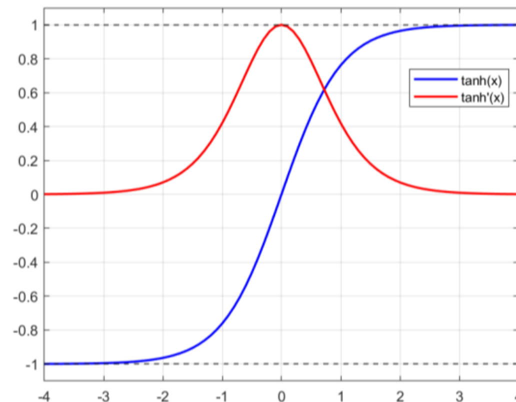


Figura 3.28 Grafica de la función tangente hiperbólica y su respectiva derivada

Esta función es más adecuada que la logística σ para las redes de neuronas profundas ya que se encuentra centrada en 0 y contempla salidas tanto negativas como positivas, estas propiedades hacen que el entrenamiento para una clasificación binaria sea más efectivo, aunque hay que tener en cuenta que los resultados binarios son $\{-1,1\}$ y no $\{0,1\}$. Aun siendo mejor que la anterior esta al igual que su predecesora comparten el siguiente problema, para hacer que una neurona devuelva 1 debe tender a infinito el sumatorio de esta y viceversa con -1 ya que la función $\tanh(x)$ presenta dos asíntotas horizontales en estos puntos. Esto se solucionará con la optimización del sigmoide nombrada a continuación.

ACT_OPSIGMOID – Esta es de las versiones más adecuadas de la función logística para trabajar con redes neuronales. Siguiendo los pasos de Yann LeCun y compañía en el artículo *Efficent BackProp* publicado en 2012 [19], se obtiene esta, que tomara el nombre de sigmoide optimizado en el ámbito del trabajo. La razón del porque esta función es superior a las dos ya mencionadas antes, es que la imagen de la función es más amplia, haciendo que la zona lineal de la función se encuentra entre $[-1,1]$, que son los valores esperados de la clasificación binaria. Haciendo que las neuronas no cuenten con valores muy pronunciados para llegar a estos valores.

$$op\sigma(x) = 1.7159 \tanh\left(\frac{2}{3}x\right)$$

$$op\sigma'^{(x)} = 1.14393\,\mathrm{sech}^2(0.\hat{6}x)$$

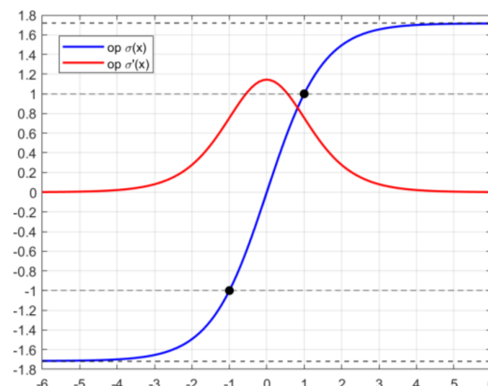


Figura 3.29 Grafica de la función sigmoide optimizada y su respectiva derivada

Como se puede observar en la gráfica representada en la Figura 3.29, la región entre -1 y 1 sigue prácticamente la linealidad descrita por $f(x) = x$. También, es importante si se quiere mejorar aún más el entrenamiento, que el valor de inicialización de los pesos se encuentre en la zona de linealidad de la función de activación siguiendo una distribución normal.

Al reescalar las gráficas en cada una de las operaciones sigmoidales, tal vez no se aprecia la diferencia entre estas, por ello en la Figura 3.30 se visualizan las 3 juntas para poder comparar estas.

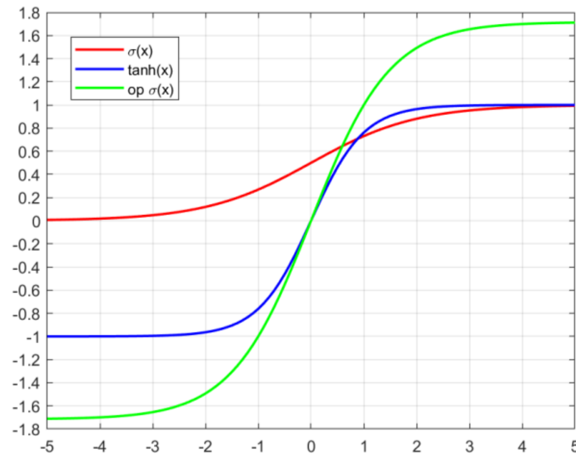


Figura 3.30 Comparación entre las funciones sigmoidales de activación

ACT_RELU – La función de Unidad Rectificada Uniforme (ReLU) discrimina la parte negativa de los reales, así haciendo que todo valor x que sirva de entrada que sea negativo devuelva 0 y todo positivo ese mismo. Analizando la función, podemos observar que no tiene derivada cuando x=0, como esta derivada acercándose por la izquierda es 0 y por la derecha es 1, normalmente se decide elegir uno de los dos números, en la mayoría de los casos no importa le número seleccionado, así que, en el proyecto, de forma arbitraria se seleccionó el valor 1.

$$ReLU(x) = \max(0, x)$$

$$ReLU'(x) = \begin{cases} 0 & si \ \ x < 0 \\ 1 & si \ \ x \geq 0 \end{cases}$$

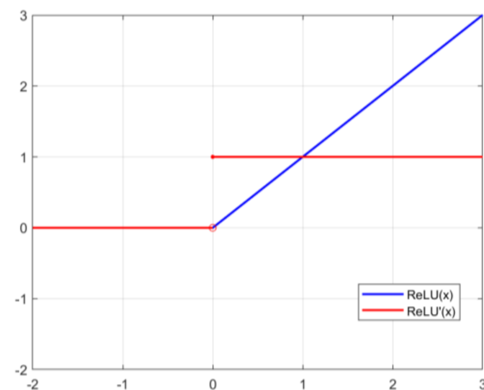


Figura 3.31 Grafica de la función ReLU y su respectiva derivada

Esta función de activación es considerada como "el nuevo sigmoide" ya que antiguamente la función por defecto para toda re era el sigmoide, y en las redes más modernas se está empezando a usar ReLU debido a su sencillez y a que requiere menos coste computacional que las sigmoidales. La función también tiene sus contrapuntos como el error de "Muerte de ReLU" que sucede cuando el gradiente de un peso llega a una región donde es cero y produce que la

neurona deje de aprender por completo. La muerte del ReLU puede ser solventada usando diferentes variantes del rectificador como el leaky ReLU. También al no ser una función acotada, sobre todo en redes con varias capas y datos de entrenamiento grandes y sin normalizar, pocas épocas de ajuste pueden ocasionar que los valores de los pesos se disparen y la maquina no pueda lidiar con valores tan grandes.

ACT_LRELU – Para lidiar con el error de muerte de ReLU, se pueden usar varias funciones como el Leaky ReLU o la Unidad de Rectificado Exponencial ELU, en el ámbito del trabajo se trabajará solo con la primera nombrada, aunque sería posible utilizar neuronas con ELU mediante la funcionalidad de función de activación personalizada (de la que se hablará más tarde en esta misma Subsección). La función Leaky, es similar a ReLU, solo que para el dominio negativo de esta no se establecen todos los valores a 0 sino que se multiplican por un parámetro α que por defecto toma el valor $\alpha = 0.1$, y puede ser cambiado mediante el uso de la función `layer_set_alpha`.

$$LReLU(x) = \max(\alpha x, x)$$

$$LReLU'^{(x)} = \begin{cases} \alpha & si\ \ x < 0 \\ 1 & si\ \ x \geq 0 \end{cases}$$

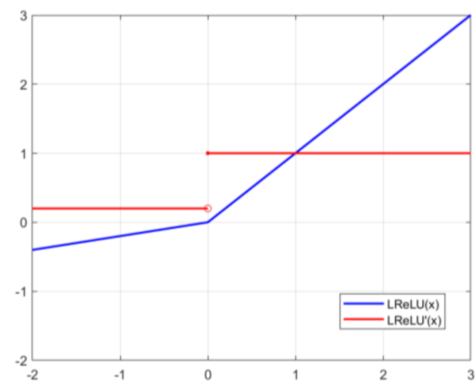


Figura 3.32 Grafica de la función LReLU con valor $\alpha = 0.2$ y su respectiva derivada

Aunque esta solución hace que las neuronas no mueran como en el ReLU simple, sigue siendo susceptible a provocar errores cuando se usan valores grandes y sin normalizar en redes profundas con varias capas. Una forma de solucionar este problema puede ser el uso de capas intermedias con funciones de activación que acoten o suavicen los valores más altos.

ACT_SOFTPLUS – La función softplus también llamada smoothReLU, es una aproximación suavizada del rectificador ReLU, que elimina el salto de la derivada de esta. Curiosamente, la derivada de softplus es la propia función sigmoidal.

$$softplus(x) = \ln(1 + e^x)$$

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

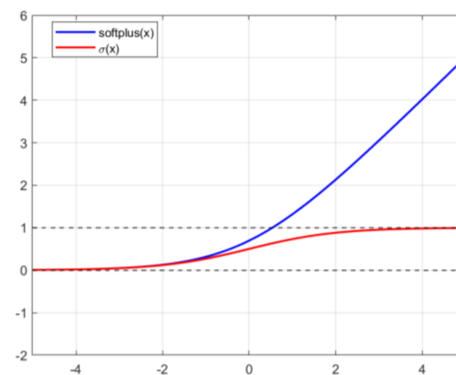


Figura 3.33 Grafica de la función softplus y su respectiva derivada (sigmoide)

A parte de todas estas funciones de activación, se ha implementado la forma para hacer que el programador pueda definir una función de tipo `double` que permita un parámetro también `double` y se le asigne a la capa deseada usando el método `layer_custom_act_func`. Como el algoritmo de aprendizaje hace uso de la derivada de la función de activación se ha tomado la decisión de permitir que se use el método de diferencias finitas para las funciones de activación personalizadas. De esta forma el programador no debe encontrar la derivada mediante el análisis matemático de la función. Recordando cómo funcionaba el método de diferencias finitas:

$$f'(x) = \lim_{\varepsilon \longrightarrow 0} \frac{f(x+\varepsilon) - f(x)}{\varepsilon}$$

El valor de épsilon se guarda en la estructura tipo `neural_net`, y se modifica usando el método `nn_set_epsilon`. Por defecto viene con un valor de $\varepsilon = 10^{-3}$.

3.5.3 Alimentación de la red de neuronas

La alimentación comúnmente conocida como feedforward como previamente ya se ha discutido en la memoria, es la operación principal de la red de neuronas artificiales, esta toma una serie de datos y los introduce en la primera capa o capa de entrada de la red, después iterativamente los va propagando de capa en capa hacía delante con el fin de obtener un resultado por cada neurona en la capa de salida. Como ya se ha discutido el funcionamiento de la alimentación de la red en todas las demás secciones del Capítulo 3, no pretendo profundizar en ello en esta sección.

Al implementar la librería se quería hacer que esta operación se realizara tan solo con la llamada de un único método llamado **`feed_forward`**, que toma como parámetros la red de neuronas tipo `neural_net`, una cadena de datos de tipo double que almacena las entradas de la red y el tamaño de esta cadena. El método descrito devuelve un puntero a una matriz vector horizontal que contiene todas las salidas de la red. Si se quiere consultar de forma sencilla estas se puede hacer uso del método `mat_print` implementado en el código de “matrix.h”.

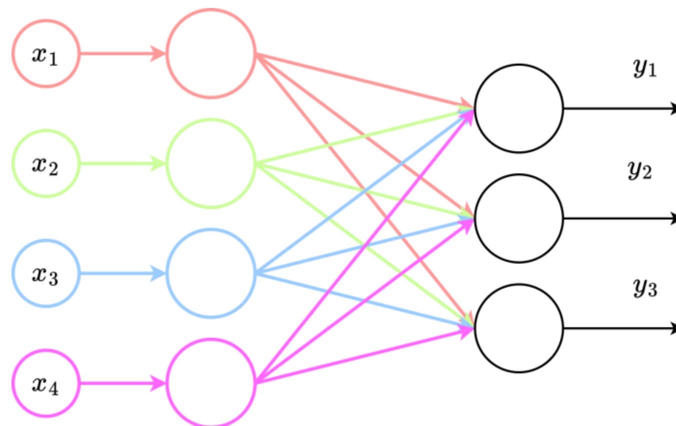


Figura 3.34 Visionado de la propagación de los valores de entrada al realizar feedforward

Para conocer el error cometido por la red en una iteración del entrenamiento se ha desarrollado la función **`cost`**, que calcula el error entre los datos esperados y los obtenidos dándole un set de datos (con los resultados esperados). Esta rutina hará uso de la función de error establecida por el programador. Por defecto se usa la media de errores cuadráticos. Mas información sobre las funciones de error y como cambiarlas en la Subsección 3.5.4. Al igual que con la operación de `feed_forward`, esta devuelve una matriz vector horizontal que

contiene los valores para el coste de cada salida. Como con la operación anterior, se puede usar `mat_print` para imprimir por consola el valor de la matriz o usar `mat_seek` para acceder a cada valor de los elementos contenidos en la matriz.

Si se tiene un set de datos de entrada y sus respectivos datos esperados para operaciones de clasificación entre dos clases, se puede usar la función de la librería **`single_binary_acurracy_rate`** para determinar la tasa de acierto a la hora de realizar una clasificación. Esta calcula la distancia entre cada par de valores esperado y obtenido y lo compara con la distancia mínima establecida mediante la llamada a la función por el programador, si es menor cuenta como clasificación exitosa y si es mayor como operación fallida, finalmente realizando la media entre todos los casos se obtiene una tasa de acierto entre 0 y 1.

3.5.4 Funciones de Error y Optimizadores

La idea original de la librería contemplaba implementar varios tipos de función de error al igual que se ha descrito en la Subsección 3.5.2. La opción de cambiar de función de error ha sido implementada, pero por falta de tiempo se ha decidido tan solo implementar dos funciones. La forma de seleccionar la función que el programador desea usar es similar a la que se debe usar, la diferencia es que como la función de error afecta a toda la red y no tiene sentido tener funciones de error intermedias, esta se debe definir en el espacio de la red y no en el de las capas como se hacía con las funciones de activación. El programador debe usar el método **`nn_set_err_func`** junto a la macro definida para el error deseado si quiere seleccionar una entre las ya implementadas:

`ERR_SQRDIFF` – Se trata de la función de error por defecto, error cuadrático medio. Es la ya usada anteriormente en todos los scripts realizados durante la fase de experimentación. A modo de recuerdo, para cada salida k de la red y N el número de casos de entrenamiento que se usa por cada lote, tomando el valor esperado como $\hat{y}_k$ y el valor obtenido como y_k, obtenemos el valor de error cuadrático medio de la siguiente forma:

$$E_k = \frac{1}{N}\sum_{n=1}^{N}(y_{nk} - \hat{y}_{nk})^2$$

Obteniendo su derivada respecto a y_k como:

$$\frac{\partial E_k}{\partial y_{nk}} = 2N(y_{nk} - \hat{y}_{nk})$$

`ERR_HSQRDIFF` – De la misma forma también se puede seleccionar la única alternativa que la librería lleva implementada, la mitad del error cuadrático medio. Aunque funcionen de la misma manera, se quería probar la modularidad de la librería con la implementación de una alternativa (aunque sea parecida) a la función de error por defecto. Esta es una variante del error cuadrático medio que toma el valor obtenido y lo divide a la mitad, así haciendo que cuando se derive este para encontrar un mínimo se simplifique el exponencial que pasa a multiplicar. Usando la misma nomenclatura que en las anteriores formulas este error es definido de la siguiente manera:

$$E_k = \frac{1}{N}\sum_{n=1}^{N}\frac{1}{2}(y_{nk} - \hat{y}_{nk})^2$$

Obteniendo su derivada respecto a y_{nk} como:

$$\frac{\partial E_k}{\partial y_{nk}} = N(y_{nk} - \hat{y}_{nk})$$

ERR_SIMPDIFF – Con motivo de investigar sobre la posible aplicación del error simple como función de error, se implementa también esta, aunque resulta no funcionar ya que su derivada elimina información necesaria para el ajusta así que, aunque esta implementada no se recomienda su uso al no ser que se quiera experimentar con esta para valorar su aplicación de otro modo.

También al igual que con las funciones de activación se puede usar una función personalizada mediante su especificación con el método **nn_custom_err_func**, este acepta un puntero a una función que devuelve un double y acepta 2 double distintos que representan el valor de y e $\hat{y}$. El valor de la derivada del error se obtiene mediante el método de las diferencias finitas usando el valor de la red ε. Que se comparte cuando el programador quiere usar funciones de activación personalizadas. Para cambiarlo, de la misma forma que en la Subsección 3.5.2, se debe usar la llamada al método nn_set_epsilon.

Como repunte, se han investigado otros tipos de funciones de error usados en el aprendizaje automático como el error de entropía cruzada binaria o “Binary Cross-Entropy” en inglés, el cual es usado por defecto en muchas librerías de redes neuronales artificiales. En caso de que se trabajara en una actualización del código, implementar este tipo de error sería un objetivo atractivo.

Respecto al algoritmo de optimización, la red creada por el programador usara el descenso de gradiente con lote completo por defecto. Es posible añadir el método de momento o bola pesada (Subsección 3.4.3) al descenso de gradiente tan solo estableciendo en la red un valor para la tasa de decadencia mediante la rutina **nn_set_decay_rate**. Si se desea volver una red al uso sin momento se debe volver a utilizar el método y establecer el valor a **0.0f**.

Por último, la librería permite cambiar el número de casos usados (lote) en cada entrenamiento. Esto se hace mediante la función **nn_set_batch_size**.

Dependiendo el valor establecido como parámetro en esta función puede variar el funcionamiento del optimizador. Existen tres casos:

- **Tamaño de lote < 1** – La red usara todos los casos de prueba en cada entrenamiento.
- **Tamaño de lote = 1** – La red usara el descenso de gradiente estocástico (SGD) como optimizador, tomando solo una muestra cada entrenamiento.
- **Tamaño de lote > 1** – La red usara mini-lotes seleccionados de forma pseudoaleatoria durante el descenso de gradiente de cada entrenamiento.

De estos tres tipos de optimizado se habla en la Subsección 3.4.3 de forma más detallada y con mayor profundidad.

3.5.5 Aprendizaje Supervisado mediante Retropropagación

El entrenamiento de la red viene dado por el algoritmo de retropropagación o “backpropagation”. Como se explica en la Subsección 3.4.2, la retropropagación permite a los perceptrones multicapa aprender usando un optimizador como el descenso de gradiente y propagando el error desde la capa de salida hasta la primera. Se vio en la Subsección 3.4.2 que el algoritmo al trabajar junto al descenso de gradiente requiere de un análisis matemático de tanto las funciones de activación como las funciones de error. En el ejemplo descrito junto al algoritmo previamente en este documento solo se tenía en cuenta una única función de activación y una sola de error. Cuando se implementa un solo modelo con las funciones fijas, basta simplemente con escribir las matemáticas detrás de ellas en un papel y cuando se comprueba que son adecuadas se pasa a código. Esto es fácil y no muy costoso si se quiere implementar una solución que no permita modificación, pero si tratamos de hacer algo más generalizado (como es

el caso de “gml_nn”) se debería hacer el análisis de $a \times e$ casos, siendo a el numero de funciones de activación y e el numero de funciones de error limitando el crecimiento de la librería y haciendo que se deban de programar bloques muy grandes de código por cada nueva adición.

En el caso de la librería, se deberían contemplar $7 \times 3 = 21$ casos distintos sin cotar con la posibilidad de que el programador pudiera dotar a diferentes capas de funciones de activación diferentes entre sí y que tampoco pudiera usar funciones personalizadas. En términos de tiempo, podría tomar tiempo muy elevado analizar cada caso y comprobar su correcto funcionamiento, otra parte donde se verá afectada la librería por el uso de esta práctica seria en las líneas de código para implementarla. Contando que en los scripts realizados en la Sección 3.4 “bp_matrix_experiment”, “bp_matrix_momentum” y “nngen_pre” contaban solo con un caso único (sigmoide y error cuadrático medio) y ocupaba alrededor de 50 LOC la implementación del retro propagador. Hacemos cálculos sencillos y obtendríamos que obtenemos $50 \times 21 = 1050$ LOC (que son más de las líneas que tiene la implementación de la librería desarrollada) únicamente para implementar esta parte sin contar las adiciones que permiten hacer cambios en el tamaño del lote de entrenamiento y las adiciones del optimizador con momento.

La forma de solucionar esto está en buscar de qué forma se puede dividir la derivada de nuestra función de error respecto al peso w_{ij} (que se multiplica al valor que a_i va desde la neurona i a la j) que se quiera ajustar de forma que se tenga en cuenta las diferentes funciones de activación de las capas de la red. Sabiendo que queremos minimizar la función de error e para minimizar el error total E, como en la Subsección 3.4.2 partimos de una primera aplicación de la regla de la cadena:

$$\frac{\partial e}{\partial w_{ij}} = \frac{\partial e}{\partial a_j} \times \frac{\partial a_j}{\partial w_{ij}}$$

Siendo a_j el resultado de la operación feedforward de la neurona j en la capa anterior, asignando a la variable h_j el valor de la suma ponderada de la misma neurona y aplicando la regla de la cadena una última vez sobre la derivada de a_i respecto a w_{ij}:

$$\frac{\partial e}{\partial w_{ij}} = \frac{\partial e}{\partial a_j} \times \frac{\partial a_j}{\partial h_j} \times \frac{\partial h_j}{\partial w_{ij}}$$

Si prestamos atención, la derivada de h_j respecto a w_{ij}^c no es mas que el valor de la propia entrada por la que se ve multiplicado nuestro peso, por lo que podemos simplificar a la siguiente ecuación:

$$\frac{\partial e}{\partial w_{ij}} = \frac{\partial e}{\partial a_j} \times \frac{\partial a_j}{\partial h_j} \times a_i$$

En el caso de querer ajustar un sesgo b_j siendo $a_i = 1$.

$$\frac{\partial e}{\partial b_j} = \frac{\partial e}{\partial a_j} \times \frac{\partial a_j}{\partial h_j} \times 1$$

Habiendo reducido con la regla anterior, observamos que quedan 2 derivadas sin resolver en la ecuación. Siendo la derivada de a_j sobre el sumatorio h_j la derivada de nuestra función de activación sea cual fuere. Y la que queda siendo la derivada de la función de error respecto al valor obtenido por la función de activación.

Dando un paso más adelante nos topamos al igual que en la Subsección 3.4.2 con que existen dos casos diferentes dependiendo de si el peso a ajustar apunta a una neurona en la capa de salida o a otra en la capa oculta. Ya que el valor de la derivada de la función de error depende de las derivadas de esta y la de la función de activación de las capas posteriores. En la capa exterior no pasa esto ya que no existen otras capas posteriores. Pues podemos nombrar a la variable δ^C como el termino común de la capa de salida.

$$\delta^C = \frac{\partial e}{\partial y_j} \times \frac{\partial y_j}{\partial h_j}$$

Y para las demás capas podemos calcular el valor realizando un sumatorio de todos los pesos alterados en la siguiente capa multiplicados por el termino común de esa misma, haciendo que para calcular el ajuste en una capa c se calculan todos los posteriores.

$$\delta^c = \sum_{k=1}^{N_{c+1}} w_{jk} \delta_k^{c+1} \frac{\partial a_j}{\partial h_j}$$

Uniendo δ^c a nuestra expresión inicial obtenemos:

$$\frac{\partial e}{\partial w_{ij}} = \delta^c a_i$$

Para optimizar el algoritmo en la librería, se calculan los valores de las capas empezando por la ultima capa hasta la primera almacenando δ^c en memoria para no tener que repetir el cálculo de este. Este método reduce la complejidad computacional del método de exponencial a lineal.

Volviendo a la implementación realizada de la librería, se decide implementar las derivadas de tanto las posibles funciones de activación como las de error y se implementa el método de diferencias finitas para poder usar punteros a otras funciones personalizadas como se mostró en las Subsecciones 3.5.2 y 3.5.4. El algoritmo de retropropagación en la capa de salida resulta ser una simple multiplicación entre los resultados obtenidos de las derivadas de las funciones deseadas para los modelos y el valor de salida de la neurona afectada por el ajuste. En las demás capas (ocultas), resulta en la misma multiplicación, pero cambiando el valor de la derivada de la función de error por el sumatorio del resultado de multiplicar el delta δ con su respectivo peso w que parte de la neurona afectada a todas las de las siguientes capas.

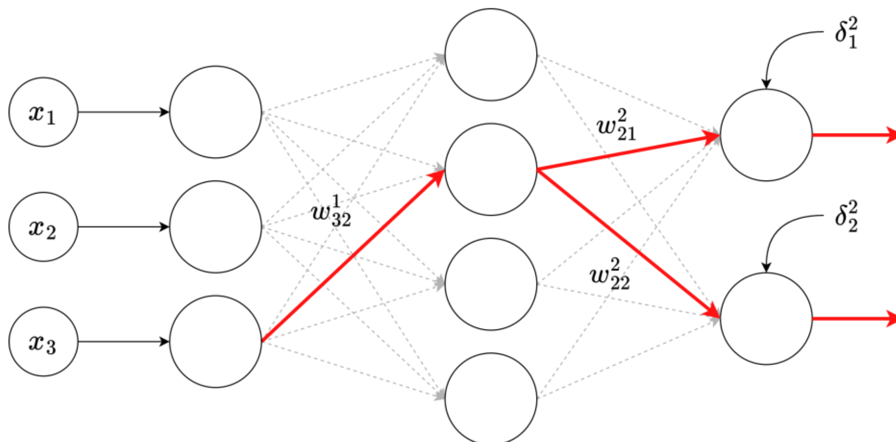


Figura 3.35 Visualización de los deltas y pesos que afectan al ajuste de w_{32}^1

La librería permite al programador accionar una sola época del entrenamiento mediante la llamada a la función **train_network_epoch**. Esta función reproduce el entrenamiento supervisado mediante retropropagación y automáticamente reajusta los valores de todos los pesos que hayan sido alterados (normalmente todos).

3.5.6 Guardado y Cargado de Redes de Neuronas

La librería ya está dotada de las propiedades necesarias para crear redes capaces de entrenarse y dar resultados vasados en sus entrenamientos, pero solo mientras el código siga ejecutando.

Pongamos el hipotético de que se quiere desarrollar un modelo que prediga si un paciente de cierto hospital está enfermo de una enfermedad “X” en función de sus constantes y muestras biológicas. Dicho modelo tarda en entrenar digamos que 15 minutos y su validado otros minutos. El resultado de la prueba interesa que sea rápida y eficiente para poder hacer una prueba en el mínimo tiempo y se consuma el menor número de recursos posibles. Viene un paciente y quiere saber si tiene dicha enfermedad, no conviene hacer que el medico espere a que la maquina entrene y tenga que comprobar si el entrenamiento ha sido correcto cada vez que un paciente quiere comprobar si tiene “X”. Es sentido común, el modelo debe estar ya entrenado y validado por algún especialista.

Pues las redes creadas por nuestra librería no son capaces de ser guardadas ni mucho menos de cargarse y estar listas para funcionar. Por ello se comienza la implementación de un sistema para poder guardar modelos entrenados en un fichero y poder usarlos posteriormente, o seguir con su entrenamiento. Para implementar esta funcionalidad se decidió especificar un formato propio de fichero que guardara texto entendible tanto por la maquina como por el usuario que quiera hacer uso de ello, permitiendo realizar consultas a la red neuronal sin necesidad de cargarla programáticamente.

Lo primero que se necesita es definir un sufijo para el formato del fichero. Se selecciono “.nn”, ya que son las siglas en ingles de redes de neuronas y parece ser intuitivo el uso de este nombrado. También se comprobó si dicho sufijo es usado comúnmente en sistemas Linux en el manual de la distribución Ubuntu [20] y no se encontró nada al respecto por lo que se puede dar por hecho que no se está nombrando a nuestras redes con un sufijo improcedente.

Para realizar el guardado de una red creada con la librería, el programador debe utilizar el método **nn_save** estableciendo como parámetro el nombre deseado para la red. Este se guardará automáticamente en la carpeta “saved”.

El fichero puede contener comentarios creados con una almohadilla como primer carácter de una línea en las primeras líneas del fichero creado. Después de los comentarios, cuenta con una cabecera que guarda los datos globales de la red de neuronas artificiales donde se especifica el número de capas contenidas dentro de la red, el número de parámetros de entrada que acepta, ambas tasas de aprendizaje η y de decadencia β, el valor de salto usado para el método de diferencias finitas ε, el tamaño del lote que opera el optimizador, la función de error seleccionada, la semilla usada para generar los valores iniciales de la matriz de pesos y el ancho de todas las capas en la red.

Después se especifica capa a capa primero las cualidades de la capa incluyendo el tamaño la función de activación asignada y el valor Alpha y después el valor de los pesos y sesgos de toda la capa en forma de matriz.

A modo de ejemplo se muestra a continuación el contenido del fichero generado por la red que tiene 2 capas ocultas con la función de activación Leaky ReLU y una de salida con el Sigmoide optimizado sin inicializar los pesos.

```
Input Number: 2
Learning Rate: 0.10000000
Decay Rate: 0.00000000
Epsilon: 0.00100000
Batch Size: 0
Error Function: 1
Random Seed: 0
Number of Layers: 3
Layer Widths: 3, 4, 2

*Layer
       Width: 3
       Alpha Value: 0.10000000
       Activation Function: 5
  -Weights
       0.0000000000 0.0000000000 0.0000000000
       0.0000000000 0.0000000000 0.0000000000
       0.0000000000 0.0000000000 0.0000000000
*Layer
       Width: 4
       Alpha Value: 0.10000000
       Activation Function: 5
  -Weights
       0.0000000000 0.0000000000 0.0000000000 0.0000000000
       0.0000000000 0.0000000000 0.0000000000 0.0000000000
       0.0000000000 0.0000000000 0.0000000000 0.0000000000
       0.0000000000 0.0000000000 0.0000000000 0.0000000000
*Layer
       Width: 2
       Alpha Value: 0.10000000
       Activation Function: 4
  -Weights
       0.0000000000 0.0000000000
       0.0000000000 0.0000000000
       0.0000000000 0.0000000000
       0.0000000000 0.0000000000
       0.0000000000 0.0000000000
```

Código 3.7 Fichero de ejemplo de guardado de red de neuronas

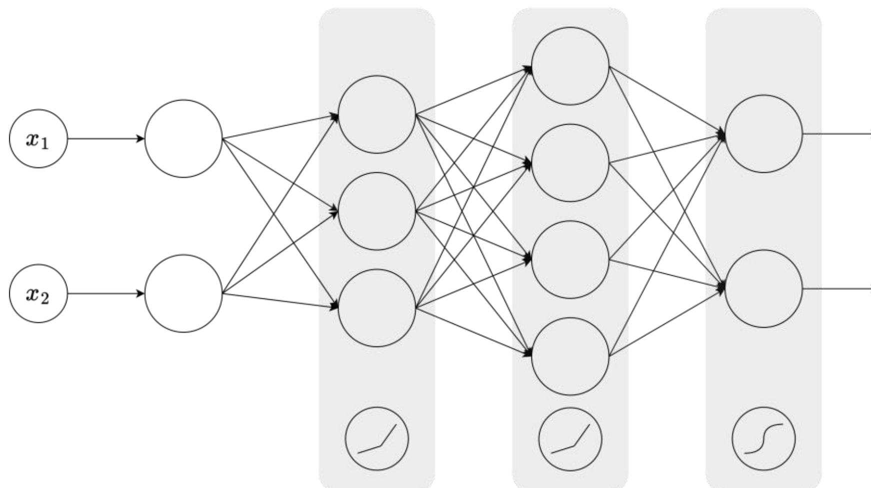


Figura 3.36 Red de neuronas descrita por el fichero de ejemplo

Como no se puede guardar de forma sencilla en el mismo archivo de texto la función personalizada (en caso de que el programador haya establecido una), se decide guardar que esta usa una personalizada pero no la función como tal, así mostrando por pantalla al cargar la red que necesita usar de nuevo el método **nn_custom_err_func** o **layer_custom_act_func** para volver a establecerla.

También se implementa una función de cargado de neuronas, obviamente ya que carece de sentido poder guardarlas, pero no cargarlas. Por defecto si se le entrega al método **nn_load** una cadena de caracteres con solo el nombre de la red guardada anteriormente acedera a ella si se encuentra en la carpeta "saved", en caso contrario mostrara un error por consola y parara la ejecución del código. Si se decide usar una dirección relativa como ruta del fichero la rutina abrirá esta.

3.5.7 Función de entrenamiento simplificado

Librerías como *Keras* permiten al programador hacer entrenamientos de forma sencilla con la llamada de una sola función. En el proyecto se ha optado por proporcionar una solución genérica al entrenamiento de una red (aun así, el programador puede usar la función **train_network_epoch** dentro de un bucle si desea personalizar el entrenamiento). Para usar la función de entrenamiento simplificado con el nombre **train_network** el programador debe primero cargar dentro de la red de neuronas los datos de entrenamiento usando el método vacío `nn_set_training_data` y si se dispone de datos de prueba y se quiere que se muestre información durante la ejecución de estos también deben se añadidos mediante su función equivalente `nn_set_training_data`.

La función de entrenamiento es un bucle que cada ciertas épocas (elegidas por el programador) muestra información de como coste o error durante la ejecución del entrenamiento se va actualizando. **train_network** admite como parámetro de entrada una macro que indica que costes calcular durante y después del entrenamiento. La librería permite las siguientes opciones.

COST_NONE – No hará ningún cálculo de error durante el entrenamiento. Ideal si se trabaja con sets de datos muy grandes y se busca el máximo rendimiento del equipo. No es recomendable si no se sabe que el entrenamiento va a ser exitoso, ya que si el modelo no testa bien preparado no se sabrá.

COST_TRAIN – Solo calculara los costes o error del set de entrenamiento.

COST_TEST – Al igual que `COST_TRAIN` solo calculara los costes o error del set de pruebas. Es necesario haber establecido este antes mediante el uso de la función `nn_set_training_data` del entrenamiento. Aconsejable si se utilizan datos de entrenamiento muy grandes y se quiere ganar rendimiento a costa de no mostrar esta información.

COST_BOTH – Calculara ambos costes de forma paralela, como usar las macros `CONST_TRAIN` y `COST_TEST` a la vez. Deseable su uso siempre, aunque cuando se trabajan con sets muy grandes es necesario acotar los sets o usar otro modo.

Estos datos por defecto saldrán por consola siguiendo el siguiente formato:

```
EPOCH: 0
Train: [1, 9]
0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
Test: [1, 9]
0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
```

Código 3.8 Salida del coste de una época de entrenamiento sobre una red con 9 neuronas en la capa de salida

También se puede configurar si queremos exportar los costes calculados como csv usando el método **nn_set_cost_output** con la opción **COUT_CSV**, que guardara la información en el fichero costs.csv o si queremos que se nos muestre un gráfico creando un subproceso de *gnuplot* que muéstrela evolución de la media de los costes de los sets de datos especificados mostrando en las abscisas la época en la que el error fue medido y en las ordenadas el valor de este usar la opción **COUT_GNUPLOT**. Siempre se mostrará el error del conjunto de entrenamiento en gris y el del conjunto de pruebas en negro.

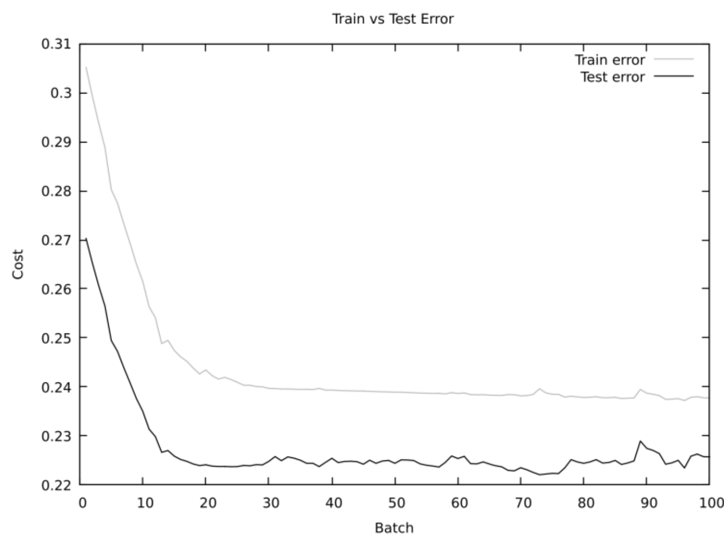


Figura 3.37 Ejemplo de salida de la gráfica de errores de entrenamiento y prueba usando gnuplot

Como adición, también se puede cambiar el comportamiento de la salida por consola de los errores, se ha implementado la función **nn_set_console_out** con las siguientes opciones:

PRT_CONSOLE – Es la que viene por defecto configurada al crear una red. Permite la impresión de los costes por consola de la forma mostrada previamente.

PRT_NOCONSOLE – Deshabilita el mostrado de los datos por consola, pero no hace que no se calculen estos, si se quiere hacer que no se calculen se tiene que usar en la llamada a la función de entrenamiento simplificado el flag **COST_NONE**.

PRT_ONLYBATCH – Al igual, deshabilita el mostrado de la impresión de los costes por consola, pero sí que muestra la época en la que el entrenamiento este. Idóneo para hacer un seguimiento de cuánto tiempo está tomando o le queda a un entrenamiento muy pesado.

3.5.8 Funciones extra de visualizado y otros métodos auxiliares

Como último apartado de la sección, se quiere hablar sobre las funciones que se han implementado durante el desarrollo de la librería para el seguimiento y verificación de las funcionalidades principales de la librería y se han dejado a disposición del programador. No se recomienda el uso de estos métodos como solución general para diferentes problemas. Lo ideal es que el programador cree su propia implementación de visualizadores, normalizadores de datos o validadores ajustándose a las necesidades del modelo a crear usando la librería. Mirando la otra cara de la moneda, estos métodos si pueden servir como guía para validar si los modelos creados parecen funcionar de forma correcta sin la necesidad de implementar más código.

Las primeras funciones que fueron implementadas son las de visualización de áreas y puntos para clasificación de un conjunto de datos en un espacio $\mathbb{R}^2$,

generando mediante la creación de un proceso de *gnuplot* (haciendo un pipe entre el proceso principal y este) diferentes imágenes. El primer método permite al usuario guardar el grafico de dispersión de los datos en una imagen con el nombre “Points.png”. La llamada de este es **plot_2d_data_for_binary**. Después, se implementó el segundo método **show_areas_2d_plot**, que dibuja las áreas donde la red clasificaría a los diferentes puntos en sus clases, esto, aunque es algo costoso y no resulta en resultados muy vistosos se consigue definiendo un valor de salto y dibujando puntos separados por este valor de salto en todo el espacio que se defina en la llamada al método.

En las Figura 3.38 se muestra los puntos de entrada de diferentes clases y en la Figura 3.39 el resultado tras el entrenamiento de una red con estos usando los dos métodos de visualización a disposición del programador.

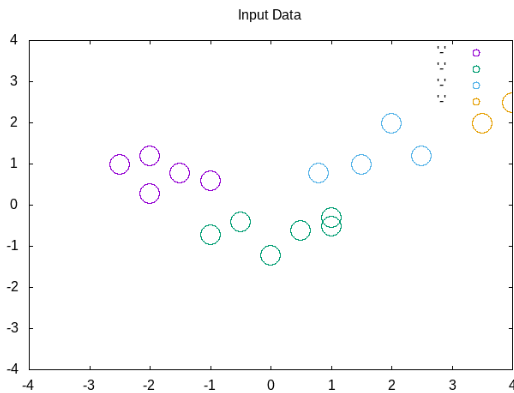


Figura 3.38 Grafico de dispersión generado por el método de visualización de puntos

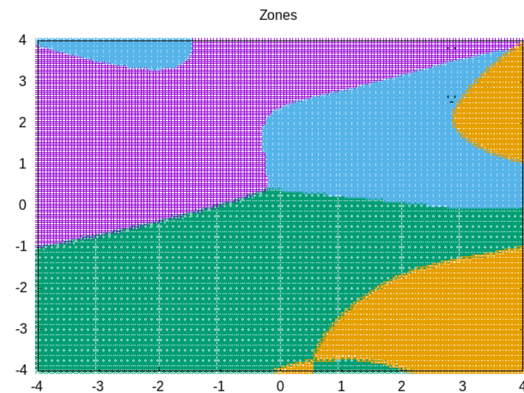


Figura 3.39 Grafico generado de una red mediante el método de visualización de áreas

A parte de los métodos de visualizado también se ha decidido incluir en el código los él ya nombrado en la Subsección 3.5.3 **single_binary_acurracy_rate** que calcula para un conjunto de datos su salida en la red usando la alimentación de esta y después decide si la clasificación de estos sets ha sido correcta usando un valor como espacio máximo entre un resultado y su valor esperado para que este sea considerado del mismo tipo. Claramente este método está pensado para comparar resultados de clasificación, pero si se desea puede ser usado para otro tipo de verificaciones. El método **choose_class** implementa un algoritmo que lee una lista de salidas binarias para un caso y decide cual de todas las salidas es las clases representadas por esta salida es la más oportuna para clasificar la muestra mediante la obtención del valor más grande.

Por último, a la librería se le dota de una función de normalizado. Normalizar los datos de entrada a la red es esencial cuando se trabaja con datos muy dispersos o grandes. La forma ideal de trabajar con los datos como se puede ver en artículo publicado por Yann LeCun y compañía *Efficent BackProp* [19] ya nombrado en anteriores secciones, es usando datos de entrada centrados en 0 y con valores negativos y positivos. Esto puede ser logrado transformando los datos con un normalizador tipo “MinMax”, que ha sido implementado bajo el nombre de **minmax_normalization** para su uso por el programador. La manera en la que funciona este es obteniendo el par de valores (mínimo, máximo) del conjunto de datos para un atributo específico y aplicando la siguiente formula a cada entrada en este para acotar el valor en el rango [-1,1]:

$$minmax(X) = \frac{2(X - \min(X))}{\max(X) - \min(X)} - 1$$

4 Ejemplos de uso, evaluación y discusión

En este capítulo se mostrará la forma de usar la librería con varios ejemplos de uso, se utilizará esta para solucionar diferentes problemas como si fuésemos un programador externo al desarrollo del trabajo. Después se realizarán experimentos para comparar una red de neuronas creada con *gml_nn* tras realizar entrenamientos similares con otra usando los mismos datos con la librería *Keras* en Python, así determinando si se consigue bajar el tiempo de entrenamiento al usar un lenguaje compilado.

4.1 Uso de la librería como programador

En esta sección se tratará de utilizar la librería desarrollada para resolver ciertos problemas de clasificación. Primero se tomarán casos más simples y después se ira complicando hasta poder crear un modelo que resuelva el famoso set de datos de reconocimiento de dígitos MNIST [23]. Se ira intercalando código con comentarios sobre el uso de la librería durante la creación de estos clasificadores. Si se desea descargar el código fuente de estas pruebas se encuentran en el repositorio del proyecto bajo el directorio de “code_examples”.

Para toda compilación realizada para hacer uso de la librería en necesario incluir los ficheros “matrix.c”, “matrix.h”, “gml_nn.c” y “gml_nn.h”, y si se hace uso del manejador de datos también “data_handler.c” y “data_handler.h”. En el caso de los ejemplos de prueba se utiliza un fichero Makefile con el siguiente contenido para generar los ejecutables:

```
CC = gcc
CFLAGS = -Wall -Wextra -g
LDFLAGS = -lm

SRC = matrix.c gml_nn.c data_handler.c NOMBRE_CODIGO.c
OBJ = $(SRC:.c=.o)
TARGET = NOMBRE_EJECUTABLE

$(TARGET): $(OBJ)
    $(CC) $(CFLAGS) -o $@ $^ $(LDFLAGS)

%.o: %.c
    $(CC) $(CFLAGS) -c $< -o $@

clean:
    rm -f $(OBJ) $(TARGET)
```

Es necesario rellenar los campos `NOMBBRE_CODIGO` y `NOMBRE_EJECUTABLE` con los nombres del fichero que contenga el código que usa la librería y el del fichero que se quiere usar como ejecutable del código.

4.1.1 Puerta lógica XOR

El primer problema que se quería resolver usando la librería es la aproximación a la puerta lógica XOR, que, aunque ya se resolvió en la Sección 3.4, se quería comprobar que con la librería podía resolver este problema de forma sencilla.

Lo primero que se debe hacer para crear una red neuronal es usar el método `nn_create`, en nuestro caso como queremos trabajar con una red sencilla con una sola capa de entrada y una de salida nos sirve, por lo que se debe definir el tamaño y el número de capas a la hora de declarar nuestra red de la siguiente forma:

```
int lay_count[] = {2,1};
neural_net nn = nn_create(ACT_OPSIGMOID,2,lay_count,2);
```

Después asignamos un valor a la tasa de aprendizaje, e inicializamos los pesos de la red de forma aleatoria con las líneas:

```
nn_set_learning_rate(&nn,0.3);
nn_weight_randf(&nn);
```

Finalmente cargamos los datos contenidos en un fichero xor.csv con el código de *data_handler* y los normalizamos a binario [−1 1] para una clasificación más eficiente:

```
int num_in = 2, num_cases = 4;
parser_result data = parse_data("../datasets/xor.csv",num_in);
nn_set_training_data(nn,num_cases,data.data_input,data.data_output);
change_all_values_for(data.data_output,data.num_out,data.num_case,0.0,-1.0);
change_all_values_for(data.data_input,data.num_in,data.num_case,0.0,-1.0);
```

Establecemos el número de épocas, y que deseamos obtener una gráfica con el coste tras el entrenamiento.

```
int epoch = 80;
nn_set_cost_output(&nn,COUT_GNUPLOT);
```

Ahora usamos la función de entrenamiento simplificado:

```
train_network(nn,epoch,1,COST_TRAIN);
```

Y establecemos más código para comprobar el correcto tras entrenamiento de la red de neuronas. Este comprueba para todos los casos del set los dos valores, el obtenido y el esperado. Después calcula el acierto obtenido. Todo se acaba imprimiendo por consola una vez finalizado el entrenamiento obteniendo los siguientes resultados y gráfica:

```
Case 0. -1 -1 = -0.996500 expected: -1
Case 1. -1 1 = 0.997855 expected: 1
Case 2. 1 -1 = 0.997855 expected: 1
Case 3. 1 1 = -0.996419 expected: -1
Acurracy: 1.000000
```

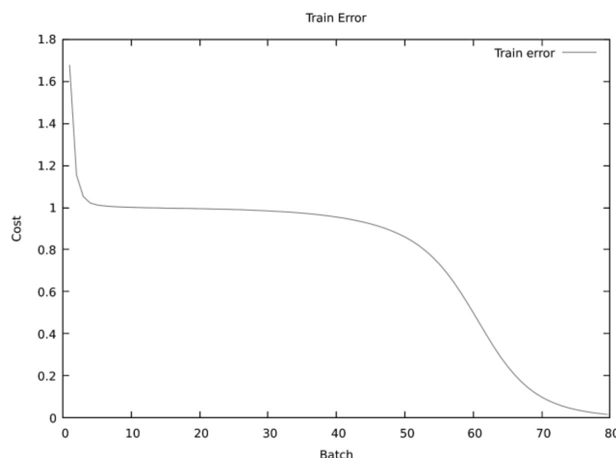


Figura 4.1 Grafica de coste por época de entrenamiento del modelo xor

Este código puede ser consultado en la carpeta de ejemplos del repositorio del proyecto bajo el nombre de xor.c.

4.1.2 Funciones lógicas más complejas

En el siguiente ejemplo se sigue trabajando con entradas en binario, pero se busca aprender una función un poco más complicada, esta cuenta con 6 bits de entrada y 2 de salida. Se puede describir por la siguiente formula lógica:

```
Y0 = (A AND B) OR (C AND D)
Y1 = (E XOR F) AND (A OR B)
```

El set de datos generado con estas funciones (tabla de verdad) se guarda en un csv con el nombre `logic.csv` que debe ser cargado con el siguiente código:

```
int num_in = 6;
parser_result data = parse_data("../datasets/logic.csv",num_in);
```

Y preprocesado por el siguiente código:

```
change_all_values_for(data.data_input,data.num_in,data.num_case,0.0,-1.0);
change_all_values_for(data.data_output,data.num_out,data.num_case,0.0,-1.0);
```

Después, se crea la red usando el método constructor para indicar que se quiere crear la siguiente red con sus respectivos selectores de parámetros:

```
int lay_count[] = {6,2};
neural_net nn = nn_create(ACT_OPSIGMOID,2,lay_count,num_in);
nn_set_learning_rate(&nn,0.05);
```

Que crea una red con una tasa de aprendizaje $\eta = 0.05$ de 3 capas, teniendo la capa de entrada el número de valores aceptados por la red como entrada de datos, una capa oculta con 6 neuronas y una capa de salida con dos neuronas que cada una devuelve un valor `Y0` o `Y1`.

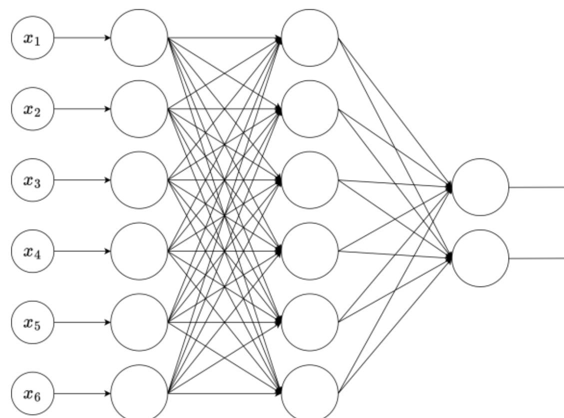


Figura 4.2 Red de neuronas para solucionar la función lógica de ejemplo

Después, inicializamos los pesos, establecemos el método de salida, el número de épocas a realizar en el entrenamiento e indicamos que se realice este.

```
nn_weight_randf(&nn);
int epoch = 150;
nn_set_cost_output(&nn,COUT_GNUPLOT);
train_network(nn,epoch,1,COST_TRAIN);
```

Como en el caso de XOR, se hace uso de `single_binary_accuracy_rate` para obtener la tasa de acierto y se imprime por pantalla una vez se ha completado el entrenamiento. Al haber usado el método `nn_set_cost_output` también obtendremos la gráfica de coste del entrenamiento. Ambas salidas se muestran en la Figura 4.3 en forma de media de costes y las líneas de texto mostradas por consola:

```
EPOCH: 149
Train: [1, 2]
0.106979 0.128231
Acurracy: 1.000000
```

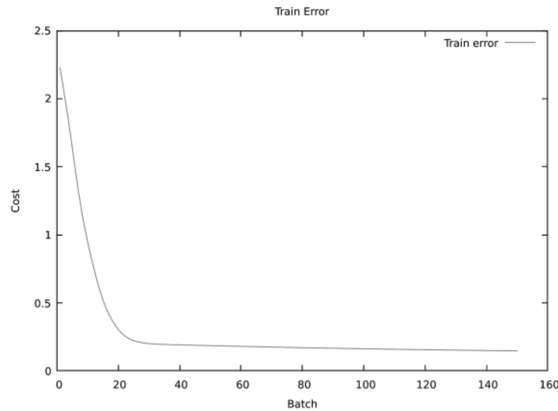


Figura 4.3 Grafica de coste por época de entrenamiento del modelo logic

Como se puede observar en los costes, no se consigue minimizar en tan pocas etapas el error a 0 para ninguna de las dos salidas, pero, aun así, el resultado del entrenamiento es óptimo porque se puede discriminar los valores como se hace con la función `single_binary_accuracy_rate` usando un valor de distancia máxima entre valor esperado y obtenido, que operar con resultados binarios $[-1\ 1]$ con discriminar positivos como 1 y negativos como -1 basta.

El código completo se encuentra bajo el nombre "logic.c" en el repositorio del TFG.

4.1.3 Clasificación de puntos en un espacio 2D

Saliendo de la dinámica del uso de datos de entrada en binario, se quieren implementar varios modelos que dadas las ordenadas y las abscisas en un espacio bidimensional puedan predecir la clase de este punto entre las distintas contempladas por la red. No es estricto hacer esto con espacios en $\mathbb{R}^2$, ya que la librería permite trabajar con un número elevado de atributos, pero como se quiere hacer uso de las herramientas de visualizado que proporciona la librería se escoge las más adecuadas para ello.

Primero se quiere hacer un modelo que dados puntos que puedan pertenecer a 6 clases distintas sea capaz de predecir a que clase pertenece pertenecen estos. Como conjunto de entrenamientos, se preguntó a ChatGPT por un conjunto de datos que contenga 6 cúmulos distintos de datos de diferentes clases separados para obtener un archivo csv que pueda ser usado con suficientes puntos. Para entrenar al modelo se debe cargar los datos usando el método `parse_data`, dividiendo la salida única entera como distintas binarias usando el método `from_integer_to_binary_classes` y configurando nuestro set de datos de la siguiente forma:

```
char filename[]  = "../datasets/6class.csv";
int num_input =2;
parser_result pre_data = parse_data(filename,num_input);
int num_classes;
double ** res_data = from_integer_to_binary_classes(pre_data.data_output,
       pre_data.num_case,&num_classes);
parser_result data;
data.num_case = pre_data.num_case;
data.num_in = pre_data.num_in;
data.num_out = num_classes;
data.data_input = pre_data.data_input;
data.data_output = res_data;
```

Después, se crea una red de neuronas con 32 neuronas en la capa oculta y tantas como clases haya en la de salida. Se le establece como función de

activación Softplus para toda la red. Se le da un valor $\eta = 0.005$ y un tamano de mini-lotes de 5 casos por entrenamiento. Finalmente se inicializan todos los pesos de forma aleatoria.

```
int lay_count[] = {32,num_classes};
neural_net nn = nn_create(ACT_SOFTPLUS,2,lay_count,num_input);
nn_set_learning_rate(&nn,0.005);
nn_weight_randf(&nn);
nn_set_batch_size(&nn,5);
```

Una vez configurada la red y los datos de entrenamiento preprocesados, se debe realizar el entrenamiento del modelo. En este caso se seguirá usando el entrenamiento simplificado, pero solo se pedirá que muestre los costes por la salida estándar.

```
int epoch = 5000;
int print_each = 100;
nn_set_training_data(nn,data.num_case,data.data_input,data.data_output);
train_network(nn,epoch,print_each,COST_TRAIN);
```

Finalmente, con el objetivo de visualizar los resultados del entrenamiento y comprobar si el modelo actúa de la forma esperada se usan los dos métodos de visualizado que presenta la librería que nos permiten guardar dos gráficas, una con los casos del conjunto de entrenamiento y otra con las áreas donde la red clasificara los puntos contenidos en estas.

```
double *ranges = (double *) malloc(sizeof(double)*4);
ranges[0] = -8;
ranges[1] = 15;
ranges[2] = -8;
ranges[3] = 15;
plot_2d_data_for_binary(pre_data.data_input,
        pre_data.data_output,pre_data.num_case,num_classes,ranges);
show_areas_2d_plot(nn,num_classes,0.2,ranges);
free(ranges);
```

Así, cuando se compila y ejecuta el fichero se obtienen las siguientes graficas tras el entrenamiento:

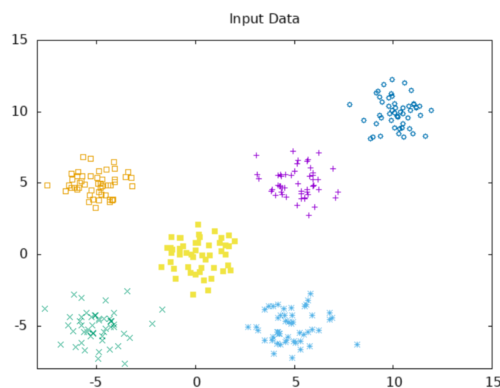


Figura 4.4 Conjunto de datos bidimensionales de entrenamiento

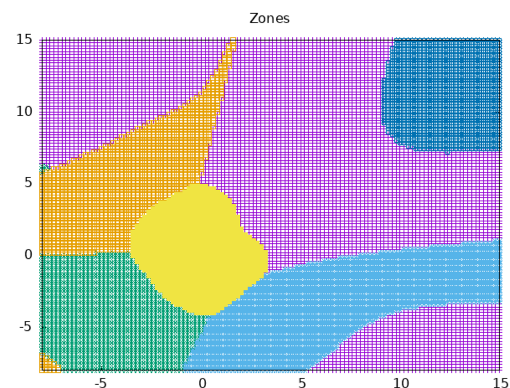


Figura 4.5 Áreas de clasificado creadas tras el entrenamiento de una red

Como es visible en las imágenes, la red clasificara de forma correcta a los puntos semejantes a los dados como entrada de datos, por lo que podemos decir que el entrenamiento ha sido efectivo. Pero, podemos pedirle casos más complejos, ya que resulten más difíciles de resolver, ya que este ejemplo se resolvería de forma

muy sencilla con otros métodos de clasificación. Modificando el código podemos hacer un modelo que intente hacer algo parecido, pero con puntos más dispersos en el espacio, para ello se utiliza un conjunto de datos de dos clases que forman dos espirales concéntricas. Cargamos de la misma forma los nuevos datos, usaremos la misma red, pero con una tasa de aprendizaje $\eta = 0.001$ y se realizara un entrenamiento más largo. También, por supuesto se usarán los mismos métodos de visualizado, que devolverán las siguientes imágenes:

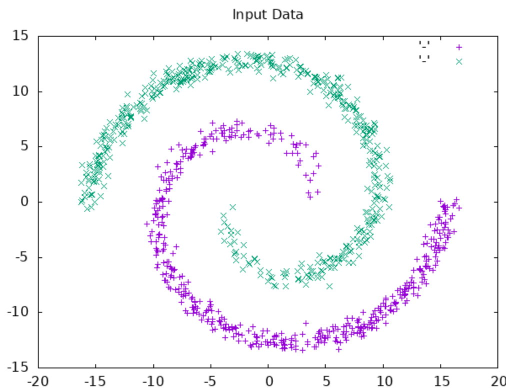


Figura 4.6 Conjunto de clases de puntos en forma de espiral

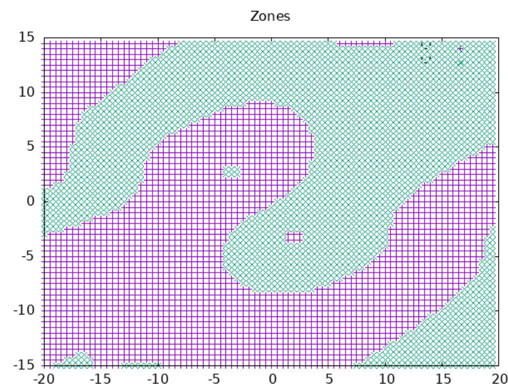


Figura 4.7 Áreas tras el entrenamiento de la red con el conjunto espiral

Todo el código completo se encuentra en la carpeta de pruebas del repositorio del proyecto. Estos dos casos se guardaron bajo los nombres de 6class.c. y spiral.c

4.1.4 Predicción con datos reales (Diabetes y VIH).

Por ahora hemos trabajado con la librería para crear redes que resuelvan problemas de clasificación muy sencillos (circuitos lógicos y espacios $\mathbb{R}^2$). Sin embargo, la librería permite crear redes más potentes y realizar entrenamientos con conjuntos de datos de tamaños mucho mayores. En esta subsección se trabajará con la librería para crear dos clasificadores que permitan predecir si una persona padece diabetes o VIH.

Primero se empezará vera el caso de la detección de diabetes. Es necesario leer los datos del set encontrado en Kaggle [21], hacer un preprocesado, que en este caso será una normalización a los atributos de los casos.

```
char filename[]  = "../datasets/Diabetes.csv";
int num_output = 1;
int num_input = get_number_atributes(filename)-num_output;
parser_result data = parse_data(filename,num_input);
for (int i = 0; i < num_input; i++)
{
       minmax_normalization(data.data_input,i,data.num_case);
}
```

A diferencia de los otros casos como contamos con un set de gran tamaño, podemos dividirlo en dos (prueba y entrenamiento) con la siguiente función de data_handler.c:

```
double div_num = 75.0/100.0;
    parser_result * div_data = data_div(data,(int) data.num_case*div_num);
    parser_result train_data = div_data[0];
    parser_result test_data  = div_data[1];
```

Así entrenando el set tan solo con los datos de entrenamiento, y comprobando al final de este con los del conjunto de pruebas la tasa de acierto. Posterior al preprocesado de los datos, se crea una red neuronal con la estructura formada por 2 capas ocultas de tamaño 12 y 8 ambas usando ReLU como función de activación, una tasa de aprendizaje $\eta = 0.03$ y aparte, se le añade momento al optimizador mediante la implantación de una tasa de decadencia $\beta = 0.2$ y se establece un tamaño de mini-lotes de 20 casos. Se inicializan los pesos usando una semilla personalizada y, por último, también se le establece la función sigmoide como la función de la última capa.

```
int lay_count[] = {12,8,1};
neural_net nn = nn_create(ACT_RELU,3,lay_count,num_input);
nn_set_learning_rate(&nn,0.03);
nn_set_decay_rate(&nn,0.2);
nn_set_batch_size(&nn,20);
nn_set_rand_seed(&nn,25);
nn_weight_randf(&nn);
layer_set_act_func(nn,3,ACT_SIGMOID);
```

Una vez tenemos la red preparada, cargamos los datos usando los métodos de cargado a red.

```
nn_set_training_data(nn,train_data.num_case,
       train_data.data_input,train_data.data_output);
nn_set_testing_data(nn,test_data.num_case,
       test_data.data_input,test_data.data_output);
```

Establecemos el número de épocas, cada cuanto queremos que se calcule el coste sobre el entrenamiento, configuramos la red para que muestre tras acabar el entrenamiento una gráfica con los costes y empezamos el entrenamiento con el método de entrenamiento simplificado especificando que queremos que se calcule el coste de ambos conjuntos de datos.

```
int epoch = 4000;
int print_each = 10;
nn_set_cost_output(&nn,COUT_GNUPLOT);
train_network(nn,epoch,print_each,COST_BOTH);
```

Por último, calculamos una vez acabado el entrenamiento la tasa de acierto con los casos de prueba y lo mostramos por pantalla:

```
double acurracy = single_binary_acurracy_rate(nn,test_data.data_input,
       num_input,test_data.data_output,.5,test_data.num_case);
printf("Acurracy: %f\n",acurracy);
```

Realizando varios entrenamientos y evaluaciones del modelo, se obtiene entre un 72% y un 80% de tasa de acierto, que comprobando en internet con otros experimentos similares [22][23] que usan el mismo set de datos con otras librerías que permiten la creación de redes son razonablemente parecidos. La grafica de los costes es la siguiente:

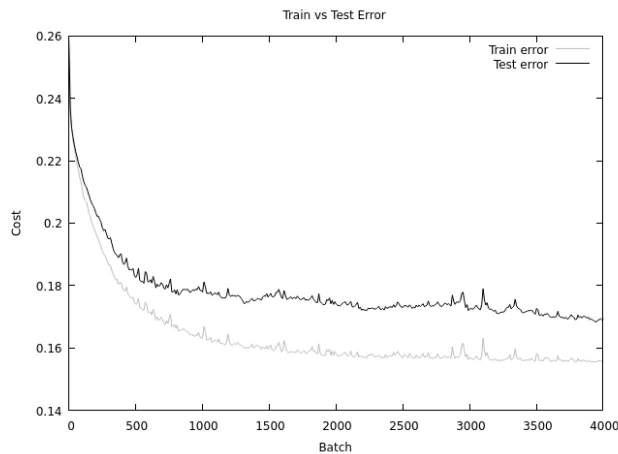


Figura 4.8 Coste durante el entrenamiento de predicción de diabetes de los casos de entrenamiento y prueba

Una vez realizado este, se proba a modificar el código del anterior experimento para realizar un entrenamiento parecido con datos sobre el VIH, el set de datos también se encontró en Kaggle [24].

En este caso se usa la misma estructura de red, pero con un numero de épocas superior, una tasa de aprendizaje $\eta = 0.1$ y usando como función de activación la aproximación al rectificador (Softplus).

```
int lay_count[] = {12,8,1};
neural_net nn = nn_create(ACT_SOFTPLUS,3,lay_count,num_input);
nn_set_learning_rate(&nn,0.1);
nn_set_decay_rate(&nn,0.2);
nn_set_batch_size(&nn,20);
nn_set_rand_seed(&nn,25);
nn_weight_randf(&nn);
layer_set_act_func(nn,3,ACT_SIGMOID);
```

Se obtiene un acierto mayor, en un intervalo del 84% al 91%. A parte, se observa en la Figura 4.9 La grafica de coste obtenida tras el entrenamiento de la red.

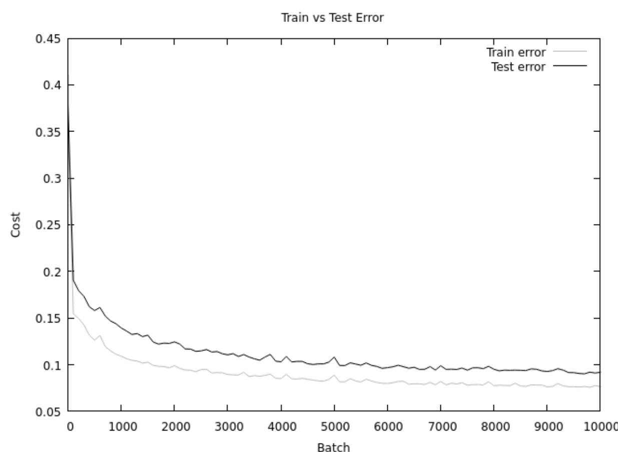


Figura 4.9 Grafica de coste obtenida tras el entrenamiento del predictor de VIH

Tanto el código del predictor de diabetes (diabetes_classifier.c) como el del VIH (aids_classifier.c) se encuentran en la carpeta de pruebas del repositorio de la librería.

4.1.5 Reconocimiento de dígitos MNIST

Finalmente, la última prueba realizada para comprobar que las redes creadas pueden ser aplicadas para el desarrollo de programas escritos en C. Es el famoso programa de reconocimiento de dígitos con el set de imágenes MNIST [25]. En especial, se usará un archivo encontrado en Kaggle que contiene todos los casos en una tabla csv [26].

Primero debemos tratar los datos, ya que vienen en un formato diferente a los usados previamente. Primero aparece el resultado de la clasificación como un entero entre 0 y 9, y luego seguidas 784 filas mostrando el valor en una escala de grises de cada pixel. El cargado de datos y su preprocesado es el siguiente:

```
char * dataset = "../datasets/mnist_test.csv";
int input_count = 784;
parser_result pre_data = parse_data(dataset,input_count);
for (int i = 0; i < input_count; i++)
{
	minmax_normalization(pre_data.data_output,i,pre_data.num_case);
}
int num_classes;
double ** res_data = from_integer_to_binary_classes(pre_data.data_input,
	pre_data.num_case,&num_classes);
parser_result data;
data.num_case = pre_data.num_case;
data.num_in = pre_data.num_out;
data.num_out = 10;
data.data_input = pre_data.data_output;
data.data_output = res_data;
change_all_values_for(data.data_output,data.num_out,data.num_case,0,-1);
parser_result * div_datas = data_div(data,(int)(data.num_case*7/10));
parser_result train_data = div_datas[0];
parser_result test_data  = div_datas[1];
```

Después creamos una red de neuronas con una capa oculta de 100 neuronas y una de salida con 10 (una para cada digito).

```
int widths[] = {100,10};
neural_net nn = nn_create(ACT_OPSIGMOID,2,widths,input_count);
```

Y ajustamos la tasa de aprendizaje $\eta = 0.01$, establecemos el tamaño de los lotes de entrenamiento a 10 e inicializamos los pesos de forma aleatoria:

```
nn_set_learning_rate(&nn,0.01);
nn_set_batch_size(&nn,10);
nn_weight_randf(&nn);
```

Establecemos el número de épocas y recortamos el tamaño del set de pruebas ya que, aunque trabajar con un número elevado de casos a la hora de comprobar el entrenamiento, muestra resultados más fiables, también hace que el tiempo de ejecución se dispare.

```
int training_it = 3000;
int trim_size = 100;
parser_result rt_data = random_trim(test_data,trim_size);
```

Realizamos un entrenamiento personalizado usando `train_network_epoch` en vez de `train_network`.

```
for (int i = 0; i < training_it; i++)
{
	printf("Test it %i\n",i);
	train_network_epoch(nn,train_data.num_case,
		train_data.data_input,train_data.data_output);
```

```
}
```

Y para comprobar la tasa de acierto se hace uso del siguiente código, que escoge el índice con el resultado más alto en las salidas de la red neuronal y lo establece como resultado final.

```
int success = 0;
for (int i = 0; i < trim_size; i++)
{
        double *current_out =
            mat_toarray(*feed_forward(nn,rt_data.data_input[i],input_count));
        if (choose_best_class(current_out,10) ==
            choose_best_class(rt_data.data_output[i],10))
        {
            success++;
        }
}
printf("Accuracy: %f\n",(double)success/(double)trim_size);
```

Por último, se hace uso de la función nn_save para guardar el estado tras el entrenamiento de la red:

```
nn_save(nn,"MNIST")
```

Revisando los resultados obtenidos podemos observar como la red no soluciona de la forma esperada el problema. Realizando varios entrenamientos se comprueba que los valores de la tasa de acierto rondan el 37% obteniendo de algún entrenamiento resultados hasta de 58%, como es el caso de la red guardada MNIST38.nn en el repositorio. En la Sección 4.2 se hablará más a fondo de los problemas que podrían estar haciendo que la red generada no alcance mejores tasas de acierto.

El código de este experimento se encuentra bajo el nombre de “mnist.c” de la carpeta de pruebas. También se decide crear otro código, que cargue la red creada en este y haga un análisis de acierto. La red se carga usando:

```
neural_net nn = nn_load("MNIST58");
```

Se cargan los datos de la misma forma que en el caso de entrenamiento, con un normalizado, desglosando la clases en distintas salidas y cambiando los valores 0 a -1 de estas. También se utiliza el mismo código para comprobar la tasa de acierto, y se obtienen valores cerca del 50%.

Este último código se encuentra bajo el nombre de “mnist_load.c” en la carpeta de pruebas.

4.2 Evaluación de los resultados

En la sección anterior se ha revisado el uso de la librería en varios casos como si fuéramos programadores haciendo uso de esta, en los primeros casos lógicos y clasificación en espacios bidimensionales se consigue crear modelos que aciertan casi el 100% de los casos. Esto es debido a que, al tratarse de datos más simples, el preprocesado de datos necesario para realizar entrenamientos sobre la red es menos crucial que en modelos con más capas y neuronas por cada una de estas.

Después los modelos predictores de las enfermedades que reconocían casos de diabetes y el virus de la inmunodeficiencia humana VIH, aunque no se obtenga una clasificación perfecta, los resultados comparados con otros experimentos en la red [22][23] sí que eran semejantes y se podrían dar como óptimos y válidos. El mayor problema surge a la hora de realizar el programa de reconocimiento

de dígitos. El comportamiento del modelo no resulta ser el esperado, en los mejores casos obtenemos hasta un 58% de probabilidad de acierto, lo cual indica que la red está aprendiendo ya que, si el resultado se formara de forma aleatoria, se obtendrían valores cerca del 10% puesto que existen 10 clases entre las que elegir una única. Tras un análisis, no se llega a encontrar un único motivo seguro que produzca un resultado tan pobre, pero si varios aspectos por los que el reconocimiento podría haberse visto afectado.

- El uso únicamente **redes tipo MLP**, puede hacer que el tiempo necesario para realizar una etapa del entrenamiento sea mayor que los tiempos usando otras técnicas como las redes convolucionales y mediante el uso de max-pooling. Implementar modelos que usen estos dos métodos reduciría el tiempo de ejecución y podrían tener mejores tasas de acierto.
- Obligatoriedad de **conexión completa** entre capas. Aunque permite que sea más simple algoritmo de propagación hacia delante o feedforward, esta técnica puede causar sobrexcitación en neuronas que haga que no se tomen en cuenta datos relevantes porque se rodean de muchos datos no tan relevantes repetidos. Permitiendo eliminar conexiones entre neuronas, se podría reducir la ocurrencia de estos casos y se dotaría de comportamientos diferentes a la red.
- El uso de una **normalización MinMax** puede acarrear problemas, las imágenes del conjunto de datos MNIST contienen en su mayoría pixeles en blanco, que, tras la normalización de mínimos y máximos, estos se denotan como -1s. Al tener tantos -1s y tan pocos números negativos, el sumatorio de cada neurona tiende a devolver números lo suficientemente grandes como para que la función sigmoide optimizado usada en todas las neuronas como función de activación devuelva valores muy cercanos a sus asíntotas horizontales alrededor de -1.71 y 1.71. Esto se podría solucionar si el programador implementaras un normalizador más adecuado para estos datos.
- Uso del optimizador con **descenso de gradiente con momento** en vez de otras propuestas, buscando en internet [27][28], se encuentran soluciones con diferente tipo de optimizadores, sobre todo Adam. Como la librería no permite el uso de otras técnicas de optimizado puede influenciar en que no se pueda obtener mejores resultados.
- Por último, la mayoría de las soluciones que he encontrado en internet [27][28], cuentan con funciones de error más avanzadas, que no se encuentran implementadas en la librería. En el caso del modelo propuesto para la resolución del problema de los dígitos escritos a mano se ha usado la **media de diferencias cuadráticas**. La que se suele usar más a menudo en modelos de este estilo suele ser la llamada “Binary Cross-Entropy”.

Debido a el uso de estas técnicas ya mencionadas, en vez de las más optimas, la red del programa no consigue ser entrenada de la forma más adecuada, de hecho, se queda lejos de esto ya que falla 1 de cada 2 veces. Como se puede comprobar en los otros casos de ejemplo la librería funciona correctamente, por lo que podemos asumir que el código que está escrito tiene un correcto funcionamiento, pero no es suficiente. En futuras líneas de desarrollo (si se continuara con nuevas versiones de la librería), debería implementarse varias de estas técnicas para mejorar el rendimiento de este tipo de problemas. Cabe destacar que la normalización no debería ser parte de la mejora, ya que no es uno de los objetivos de trabajo y se debería normalizar los datos antes del uso de la librería.

5 Resultados y conclusiones

Durante el trabajo se han revisado los conceptos más básicos de las redes de neuronas artificiales, explorando a fondo su uso y funcionamiento. Buscando crear una librería que permita utilizar estos conceptos para crear modelos de forma sencilla para el programador. En esta sección, se va a discutir el resultado del proyecto y que objetivos se han cumplido respecto a la lista marcada en la Sección 1.2 (sección 5.1). También se realizará un análisis personal en el que se dará opinión sobre la realización del trabajo y los resultados obtenidos frente a los esperados (sección 5.2). Y, por último, se expondrán varias ideas que no han sido implementadas que podrían ser de interés en futuras líneas de trabajo para lidiar con más tipos de problemas y optimizar el funcionamiento de la librería (sección 5.3).

5.1 Repaso de objetivos

El proyecto tenía como objetivo principal el desarrollo de una librería escrita en C que permita al programador crear, entrenar y guardar redes de neuronas de forma sencilla con el fin de poder resolver problemas de aprendizaje automático en este lenguaje. En la sección primera, a parte de este objetivo, se marcaron otros objetivos e hitos que ir completando a medida que se avanzaba con el trabajo. Estos fueron los siguientes:

1. Estudio de los principios básico de una Red Neuronal Artificial, del algoritmo de retro propagación para el aprendizaje supervisado y revisión de iniciativas relacionadas.
2. Primeras Implementaciones de redes básicas (perceptrón, redes de dos capas…) y mecanismos de aprendizaje sencillos.
3. Implementación de redes un más complejas y mecanismo de aprendizaje de retro propagación junto a los distintos tipos de descenso de gradiente para el aprendizaje y calculo mediante análisis de las funciones de minimizado de error para los distintos modelos.
4. Análisis de los resultados de las redes implementadas anteriormente.
5. Implementación de librería en C que permita crear y entrenar Redes Neuronales.
6. Análisis de los resultados y comparación de rendimiento de las redes neuronales generadas y las librerías ya existentes.

El primer objetivo, fue marcado debido a que no se conocía lo suficiente el campo como para poder aventurarse de forma inmediata al desarrollo de redes de neuronas. Este puede considerarse como cumplido tras leer sobre el tema y ver videos de diferentes divulgadores. Pero siempre se puede aprender más ya que es un campo en el que hay muchas ramas y diferentes tipos de modelos. Aun así, para el ámbito del proyecto, se alcanzaron los conocimientos necesarios.

Siguiendo, el segundo, el tercero y el cuarto conllevaron a fases de desarrollo experimental en el que mientras se aprendía sobre las redes y su funcionamiento, se creaban modelos cada vez más complicados. En esta fase pese a haberse presentado varios ficheros encontrados en el repositorio, se desarrollaron más, pero fueron desechados porque eran muy similares a otros o porque carecían del correcto funcionamiento que se les quería dar. Puesto a que se consiguió dar con varios modelos funcionales y se acabó con la implementación del fichero “nngen_pre.c”, que establecería las bases de cómo se trabajaría con las neuronas generadas por la librería, también podemos considerar como realizado con éxito el hito.

El objetivo principal del proyecto sería el quinto de la lista. Para realizar este se tomó como como punto de partida todos los modelos realizados hasta entonces.

Esta parte fue la más complicada, ya que se tenía que poner en conjunto todos los conocimientos obtenidos previamente. Tras mucho esfuerzo el objetivo se consigue, aunque como se ve en la Subsección 4.1.5, la librería tiene ciertas limitaciones que hacen que no se obtenga los resultados esperados en la último caso de uso. También, como trabajo a parte, se creó una serie de manuales de uso para facilitar el uso de la librería al programador.

Finalmente, el sexto y último objetivo no se ha podido cumplir de la forma que se esperaba, como se observa en la Sección 4.1 los ejemplos de uso fueron analizados y comprobados con otros modelos creados on-line. Pero por falta de tiempo no se pudo realizar grandes comprobaciones y comparaciones con otras librerías como Keras.

5.2 Conclusiones sobre los resultados

Debido a la falta de tiempo, no se pudiera realizar la comparación de rendimiento entre las soluciones existentes y la librería desarrollada, el objetivo principal de la librería se cumplió con éxito, implementando una librería que puede ser usada para obtener resultados semejantes a los obtenidos por otras soluciones. Claramente, el código tiene sus limitaciones debido a que se han implementado solo las funcionalidades de las redes de neuronas más básicas, pero permite resolver una amplia variedad de problemas de forma sencilla y fiable.

Aunque el resultado es muy satisfactorio, sí que me hubiese gustado haber contado con más tiempo para expandir aún más la librería y realizar más comprobaciones de rendimiento frente a otras librerías.

Personalmente estoy contento de haber terminado el proyecto con los resultados obtenidos, aunque los del último ejemplo no fueran los deseados. He dedicado mucho tiempo y esfuerzo a la realización de este durante el semestre. También, este trabajo marca un hito en mi formación académica, ya que se trata del proyecto más grande realizado y posiblemente del que más cosas he aprendido en toda la carrera.

También creo que esta misma memoria creada para documentar el trabajo realizado, es una buena guía y manual para ponerse manos a la obra con las redes de neuronas, ya que se explica cautelosamente paso por paso todos los mecanismos y procedimientos para implementar redes sencillas. De la misma forma que estoy orgulloso de haber realizado la librería, también me enorgullece haber creado esta memoria.

5.3 Futuras líneas de Trabajo

Una vez acabado el trabajo, se puede pensar en futuras líneas de desarrollo y actualizaciones que se debería incluir en caso de ampliar la librería o reabrir el proyecto. A lo largo del documento se ha ido mencionando distintas posibles futuras adiciones que podrían resultar interesantes en el proyecto. Me he tomado la libertad de realizar un listado de varios untos que podrían ser objeto de futuras versiones:

- **Refactorización del uso y selección de las funciones de activación**, aunque el código de la versión final de la librería funciona correctamente, a la hora de realizar la función de activación en cada neurona se comprueba cual se debe usar. Esto pasa cada vez que se calcula la salida de una neurona y hace que el tiempo de ejecución se vea afectado. Se plantea la siguiente solución: se crea un puntero a una función, y a la hora de seleccionar la función de activación de una capa, la función de selección cambia a que función esta dicho puntero apuntando, así

ejecutando en cada neurona la función apuntada, teniendo que realizar solo una vez la comprobación en todo el código. El mismo método se podría usar con la función de error.

- Añadir la opción de seleccionar para las funciones existentes en la librería si se quiere utilizar el método de las **diferencias finitas** o usar los métodos analíticos preestablecidos. Esto actualmente se puede hacer definiendo una copia de la función a usar y pasándola como función personalizada a la red mediante los respectivos métodos de la librería `nn_custom_err_func` y `lay_custom_act_func`.
- Expandir el catálogo de **funciones de error**, incluyendo funciones de perdida de clasificación como "Binary Cross-Entropy" o "Categorical Cross-Entropy". Esto solucionaría las limitaciones a la hora de realizar tareas de clasificación como la de reconocimiento de dígitos MNIST.
- Agregado, al igual que con las funciones de error o perdida, de nuevas **funciones de activación** como ELU o unidad lineal exponencial, Swish y Mish.
- Agregar la posibilidad de **eliminar conexiones** entre neuronas, haciendo posible crear redes conexiones personalizadas y eliminar las limitaciones que produce tener las redes totalmente conectadas. Esto permitiría al programador crear redes con estructuras similares a la siguiente:

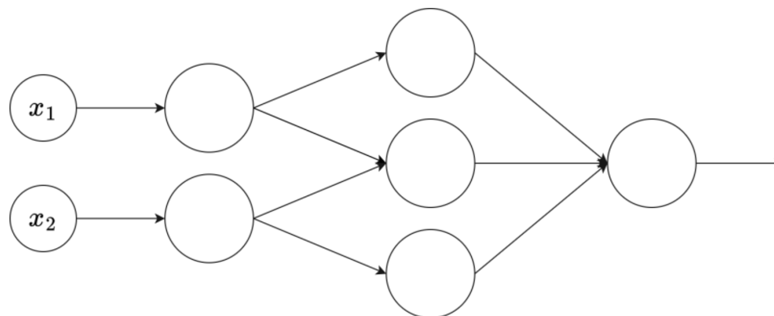


Figura 5.1 Red de neuronas con conexiones eliminadas

- Implementación de diferentes tipos de **optimizadores** y permitir al usuario seleccionar cual de ellos quiere usar al igual que las funciones de activación o de error. Se crearía una lista de los diferentes algoritmos de optimización mas conocidos, con algunos como Adam y RMSProp.
- Añadir la posibilidad de crear **capas convolucionales y max-pooling** para facilitar el tratado de imágenes como entrada de datos, lo que permitiría crear mejore modelos de reconocimiento de patrones.
- Agregado de varias **funciones de visualización** que permitan dibujar o crear esquemas de las redes creadas en las que se muestre la influencia de las entradas de datos en cada neurona, ver los costes dibujados en la gráfica de costes y mostrar las áreas de predicción sobre un problema mientras se entrena en tiempo real, de forma similar al playground de TensorFlow [29].

6 Análisis de Impacto

Finalmente, en este breve capitulo se hablará del impacto potencial que puede tener el proyecto en distintos contextos. Primero, el mayor impacto que este va a tener es personal, ya que durante el desarrollo del trabajo he aprendido mucho sobre el campo del aprendizaje automático y las redes de neuronas artificiales complementado las lecciones que se imparten en el grado. También me ha ayudado a descubrir un área de especialización en la que posiblemente acabe dirigiendo mi carrera profesional. A parte, el trabajar en C me ha hecho afianzar mis nociones en este lenguaje y el entorno Linux. Sin dejar atrás que la realización del trabajo como ya se ha dicho en la Sección 5.2 me ha proporcionado una gran satisfacción y ha marcado varios hitos en mi formación académica, entre ellos, es la primera vez que supero en código escrito las 1000 líneas en un solo fichero creado por mí, también la primera vez que tengo que documentar al detalle una ida propia y por supuesto es el trabajo en el que más tiempo he invertido. Finalmente, creo que profesionalmente este trabajo puede ser una buen proyecto de referencia a la hora de encontrar experiencias profesionales similares.

Respecto a otros tipos de impacto, no creo que el proyecto influya actualmente en otras áreas, ya que existen alternativas a la librería mas eficientes y con mas funcionalidades. Pero si que si el desarrollo de esta continuara y se desarrollan varias de las funcionalidades descritas en la Sección 5.3 (sobre todo las de visualización en tiempo real), se podría utilizar en el ambiente académico como software para realizar pequeños experimentos para introducirse en el mundo de las redes de neuronas artificiales.

Hablando del impacto medioambiental, es sabido que el entrenamiento, uso y mantenimiento de modelos de inteligencia artificial producen huellas de carbono masivas debido a la cantidad de energía requerida para entrenarlos principalmente. En el caso de la librería desarrollada, si fuese utilizada por otras personas este problema no se vería reflejado de semejante manera, ya que no esta pensado para crear modelos masivos con miles de millones de datos. A esto añadiendo que el código al estar escrito en C es software de lo más eficiente dado que es un lenguaje compilado y bastante optimizado consumiría menos que librerías parecidas programadas en otros lenguajes interpretados o más pesados.

En caso de que apicare la librería para generar código, respecto a los Objetivos de desarrollo Sostenible de la Agenda 2030 se podría realizar un potencial impacto en todos los puntos de esta ya que al poder crearse un amplio rango de modelos para solucionar diferentes problemas se puede aplicar a todo lo que se pueda automatizar o controlar usando un ordenador. Aun así, los puntos en los que habría mayor potencial de impacto positivo serian:

- **Salud y Bienestar** ODS3 – Se pueden desarrollar programas que permitan predecir si diferentes pacientes contraen una enfermedad basándose en varios tipos de datos adquiridos mediante pruebas, análisis y observaciones.
- **Educación de Calidad** ODS4 – Como ya se ha explicado antes en este mismo capitulo, la librería podría usarse como código académico para realizar experimentos sencillos que futuros estudiantes o interesados en aprender las bases de las redes de neuronas.
- **Industria, innovación e Infraestructura** ODS9 – Puesto que la librería permite desarrollar programas que clasifiquen y analicen datos, se puede

utilizar en todo proceso industrial para detectar fallos o productos en estado defectuoso.

- **Producción y Consumo Sostenible** OSD12 – De forma parecida al punto anterior, se puede usar la librería para crear programas que ayuden a optimizar procesos, prediciendo fallos antes de que sucedan y ahorrando tener que arreglar estos.

Bibliografía

Publicaciones utilizadas en el estudio y desarrollo del trabajo:

[1] MACUKOW, Bohdan. Neural networks–state of art, brief history, basic models and architecture. En Computer Information Systems and Industrial Management: 15th IFIP TC8 International Conference, CISIM 2016, Vilnius, Lithuania, September 14-16, 2016, Proceedings 15. Springer International Publishing, 2016. p. 3-14.

[2] MCCULLOCH, Warren S.; PITTS, Walter. A logical calculus of the ideas immanent in nervous activity. The bulletin of mathematical biophysics, 1943, vol. 5, p. 115-133.

[3] DO, Hebb. The organization of behavior. New York, 1949.

[4] ROSENBLATT, Frank, et al. Principles of neurodynamics: Perceptrons and the theory of brain mechanisms. Washington, DC: Spartan books, 1962.

[5] WIDROW, Bernard, et al. Adaptive switching circuits. En IRE WESCON convention record. 1960. p. 96-104.

[6] MINSKY, Marvin; PAPERT, Seymour. An introduction to computational geometry. Cambridge tiass., HIT, 1969, vol. 479, no 480, p. 104.

[7] WERBOS, Paul. New tools for prediction and analysis in the behavioral science. Ph. D. dissertation, Harvard University, 1974.

[8] RUMELHART, David E.; HINTON, Geoffrey E.; WILLIAMS, Ronald J. Learning Internal Representations by Error Propagation, Parallel Distributed Processing, Explorations in the Microstructure of Cognition, ed. DE Rumelhart and J. McClelland. Vol. 1. 1986. Biometrika, 1986, vol. 71, p. 599-607.

[9] HOPFIELD, John J. Neural networks and physical systems with emergent collective computational abilities. Proceedings of the national academy of sciences, 1982, vol. 79, no 8, p. 2554-2558.

[10] WENG, Juyang; AHUJA, Narendra; HUANG, Thomas S. Cresceptron: a self-organizing neural network which grows adaptively. En [Proceedings 1992] IJCNN International Joint Conference on Neural Networks. IEEE, 1992. p. 576-581.

[11] FUKUSHIMA, Kunihiko. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. Biological cybernetics, 1980, vol. 36, no 4, p. 193-202.

[12] LECUN, Yann, et al. Gradient-based learning applied to document recognition. Proceedings of the IEEE, 1998, vol. 86, no 11, p. 2278-2324.

[13] Tsoding Daily. YouTube. Machine Learning Playlist, 2023. Disponible en: https://www.youtube.com/playlist?list=PLpM-Dvs8t0VZPZKggcql-MmjaBdZKeDMw

[14] 3blue1brown. YouTube. Neural Networks Playlist, 2018. Disponible en: https://www.youtube.com/playlist?list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi

[15] EmergentGarden. YouTube. Watch Neural Networks Learn, 2023. Disponible en: https://www.youtube.com/watch?v=TkwXa7Cvfr8

[16] KNUTH, Donald Ervin. The art of computer programming. Pearson Education, 2005. pp. 124-125

[17] ISASI VINUELA, Pedro, et al. Redes de neuronas artificiales: Un enfoque práctico. 2004. pp. 45-69

[18] RUMELHART, David E.; HINTON, Geoffrey E.; WILLIAMS, Ronald J. Learning representations by back-propagating errors. nature, 1986, vol. 323, no 6088, p. 533-536.

[19] LECUN, Yann, et al. Neural networks: Tricks of the trade. Springer Lecture Notes in Computer Sciences, 1998, vol. 1524, no 5-50, p. 6.

[20] Manual de Ubuntu 24.04, 2024 [consulta 1-05-2024], Disponible en: https://manpages.ubuntu.com/manpages/noble/es/man7/suffixes.7.html
[21] Khare, A. Kaggle. Diabetes Dataset, 2022 Disponible en: https://www.kaggle.com/datasets/akshaydattatraykhare/diabetes-dataset
[22] Atul, A. Kaggle. Diabetes, Keras Implementation, 30-11-2016 [consulta 25-05-2024], Disponible en: https://www.kaggle.com/code/atulnet/pima-diabetes-keras-implementation
[23] Your First Deep Learning Project in Python with Keras, 16-08-2024. Disponible en: https://machinelearningmastery.com/tutorial-first-neural-network-python-keras/
[24] Velu, A. Kaggle, Set de datos AIDS 04-2024. Disponible en: https://www.kaggle.com/datasets/aadarshvelu/aids-virus-infection-prediction
[25] LeCun, Y., Cortes, C., Burges, C. MNIST Database, 1998. Disponible en: http://yann.lecun.com/exdb/mnist/
[26] Dato-On, D. Kaggle. Set de datos MNIST en formato csv, 2018, Disponible en: https://www.kaggle.com/datasets/oddrationale/mnist-in-csv
[27] Manral, Nipul GitHub. Multilayer Perceptron Training for MNIST Classification, 2019. Disponible en: https://github.com/nipunmanral/MLP-Training-For-MNIST-Classification
[28] Yu-Cheng, K. PyTorch – MLP on MNIST 19-12-2020 Disponible en: https://medium.com/analytics-vidhya/ml14-f03f75254934
[29] PlayGround de TensorFlow, 2022, Disponible en: https://playground.tensorflow.org/
[30] Repositorio GitHub de la Librería gml_nn Disponible en: https://github.com/itsTwoFive/gml_nn

Anexo A - Manual de uso de matrix.h

Esta es una versión adaptada del manual de uso de la librería matrix.h, el manual suelto se puede encontrar en el repositorio de GitHub del proyecto.

La librería matrix.h permite al usuario hacer uso de la estructura tipo `matrix`, cuya funcionalidad reside en poder trabajar en C con este tipo de estructura matemática de forma sencilla y sin tener que preocuparse de la inicialización correcta de arrays de arrays cada vez que se desea instancia una matriz. También cuenta con un conjunto de operaciones aritméticas y de manejo de datos que facilitan aún más su uso.

Podemos dividir los métodos y operaciones de la librería en los siguientes campos

- **Creación y borrado** de matrices:
  - Asignación de espacio dinámico
  - Liberado de espacio dinámico
- **Manejado de los valores** de la matriz:
  - Asignación de valores
  - Inicialización de matriz con todos los valores en cero
  - Inicialización de matriz con todos los valores aleatorios
  - Desplazado de dirección de valor dentro de la matriz
- **Representación de valores** de la matriz:
  - Mostrado por consola de la matriz
- **Operaciones aritméticas** de la matriz:
  - Suma
  - Resta
  - Multiplicación matricial
  - Multiplicación escalar
  - Suma por columnas
- **Transformaciones** de tipo
  - Transformación de matriz a array de arrays
  - Transformación de array de arrays a matriz

El tipo de datos `matrix` implementado en la librería es definido de la siguiente forma:

```
typedef struct {
    int cols;
    int rows;
    double *d;
}matrix;
```

Donde `cols` es un entero que representa el número de columnas que se encuentran en la matriz, `rows` el número de filas y el puntero a número real de precisión double `d` actúa como cadena de números reales y antes de usar una matriz debe ser inicializado mediante el asignado de memoria dinámica.

Interfaz de uso (listado de métodos)

```
matrix *mat_alloc(int rows, int cols);
```

La rutina **`mat_alloc`** crea una estructura de tipo matrix de tamaño `rows` x `cols`, reservando en el heap (montículo) de la computadora memoria dinámica suficiente para poder almacenar todos los valores que permitiría una matriz del mismo tamaño. Si la operación falla durante la asignación de memoria se

mostrará un error indicando que esto ha sucedido. El espacio reservado se da por: `sizeof(double) * rows * cols.` Después de la asignación de espacio, se establecen todos los valores de la matriz a cero. Finalmente, la rutina acaba devolviendo un puntero a la estructura creada.

```
void mat_free(matrix* mat);
```

Este método vacío permite liberar el espacio reservado a la matriz apuntada por el puntero `mat`, es imprescindible hacer uso de este cada vez que se deja de usar un tipo matrix ya que si existen muchas instancias de la estructura se puede llegar a desbordar la memoria del montículo. Una buena práctica para la programación usando los dos métodos **`mat_alloc`** y **`mat_free`** es igualar el número de veces que se usan en el código ya que así cercioramos que no se quedan reservadas en memoria dinámica estructuras que se han dejado de usar.

```
double* mat_seek(matrix m, int i, int j);
```

Esta función devuelve un puntero que apunta dentro de la matriz `m` a la posición `i` fila y `j` columna. El uso de esta puede ser útil para recolectar un valor o para modificar este. Se recomienda usar la función **`mat_set_number`** si solo se quiere modificar una vez puesto que puede ser más sencillo y acorta el código.

```
void mat_set_number(matrix m, int i, int j, double value);
```

La rutina **`mat_set_number`** permite al programador alterar el valor dentro de la celda `i x j` de la matriz `m` al nuevo valor `value`. Esta función es ideal para cambiar valores de forma momentánea, si se va a alterar un valor varias veces se recomienda usar **`mat_seek`** para obtener el puntero de la posición y trabajar sobre este ya que solo se realiza una operación de recolectado de datos dentro de la cadena de reales de esta forma.

```
void mat_print(matrix m);
```

La función vacía **`mat_print`** imprime en la consola de la maquina donde se esté ejecutando el Código la información sobre la matriz `m` (parámetros `rows` y `cols`) seguidos de los valores contenidos en la matriz. Para una matriz 4x3 con todos los valores a 0 el resultado de ejecutar la función **`mat_print`** seria:

```
[4, 3]
0.000000 0.000000 0.000000
0.000000 0.000000 0.000000
0.000000 0.000000 0.000000
0.000000 0.000000 0.000000
```

```
void mat_set_zeros(matrix m);
```

La rutina **`mat_set_zeros`** reestablece todos los valores de la matriz `m` a cero.

```
void mat_randf(matrix m);
```

El método **`mat_randf`** recorre la estructura `m` de tipo `matrix` y establece todos los valores de esta de forma aleatoria entre -1.0 y 1.0, estos límites se pueden cambiar cambiando el valor de las macros `MIN_R` y `MAX_R` respectivamente de la cabecera de la librería. La función hace uso del método de la librería stdlib.h `rand()`, si el programador quiere que los resultados siempre sean los mismo debe establecer su propia semilla con el método `srand(int seed)` de la misma librería para replicar los resultados.

```
void mat_sumf(matrix m1, matrix m2, matrix *result);
```

La función **`mat_sumf`** permite realizar la suma de valores 1 a 1 entre dos matrices `m1` y `m2`, el resultado queda guardado en la estructura tipo matrix apuntada por el puntero `result`. Antes del uso de la función **`mat_sumf`** es importante alocar el espacio correcto a la matriz en la posición `result` con el método **`mat_alloc`**. Importante recordar:

$$\underset{3\times 2}{\overset{m1}{\begin{bmatrix} a_1 & a_4 \\ a_2 & a_5 \\ a_3 & a_6 \end{bmatrix}}} + \underset{3\times 2}{\overset{m2}{\begin{bmatrix} b_1 & b_4 \\ b_2 & b_5 \\ b_3 & b_6 \end{bmatrix}}} = \underset{3\times 2}{\overset{*result}{\begin{bmatrix} a_1+b_1 & a_4+b_4 \\ a_2+b_2 & a_5+b_5 \\ a_3+b_3 & a_6+b_6 \end{bmatrix}}}$$

Si el espacio no está bien reservado o las matrices a sumar no son del mismo tamaño se mostrará por el canal de error un mensaje especificando que se ha cometido un error a la hora de realizar la operación de suma.

```
void mat_subsf(matrix m1, matrix m2, matrix *result);
```

La rutina **`mat_subsf`**, al igual que la función **`mat_sumf`** recibe dos matrices de entrada `m1` y `m2` que deben ser restadas y almacenado su valor en la matriz guardada en el espacio de memoria que el puntero `result` guarda.

$$\underset{3\times 2}{\overset{m1}{\begin{bmatrix} a_1 & a_4 \\ a_2 & a_5 \\ a_3 & a_6 \end{bmatrix}}} - \underset{3\times 2}{\overset{m2}{\begin{bmatrix} b_1 & b_4 \\ b_2 & b_5 \\ b_3 & b_6 \end{bmatrix}}} = \underset{3\times 2}{\overset{*result}{\begin{bmatrix} a_1-b_1 & a_4-b_4 \\ a_2-b_2 & a_5-b_5 \\ a_3-b_3 & a_6-b_6 \end{bmatrix}}}$$

También se deben respetar los tamaños necesarios para la resta (todas las matrices deben compartir el mismo número de filas y columnas) o si no se mostrará un mensaje de error especificando que se ha cometido este.

```
void mat_productf(matrix m1, matrix m2, matrix *result);
```

El método **`mat_productf`** realiza la multiplicación matricial entre dos matrices `m1` y `m2`, el resultado queda guardado en la estructura tipo matrix apuntada por el puntero `result`. Antes del uso de la función **`mat_productf`** es importante alocar el espacio correcto a la matriz en la posición `result` con el método **`mat_alloc`**. Siguiendo la descripción matemática de la operación de multiplicación entre matrices, el número de columnas en la primera matriz `m1` debe ser igual al número de filas en la segunda `m2`, esto hace que el resultado producido sea una matriz con el número de filas igual a `m1` y el número de columnas a `m2` de la siguiente forma:

$$\underset{1\times 3}{\overset{m1}{\begin{bmatrix} a_1 & a_2 & a_3 \end{bmatrix}}} \times \underset{3\times 2}{\overset{m2}{\begin{bmatrix} b_1 & b_4 \\ b_2 & b_5 \\ b_3 & b_6 \end{bmatrix}}} = \underset{1\times 2}{\overset{*result}{\begin{bmatrix} a_1b_1+a_2b_2+a_3b_3 & a_1b_4+a_2b_5+a_3b_6 \end{bmatrix}}}$$

Si el espacio no está bien reservado o las matrices a multiplicar no son del tamaño correcto se mostrará por el canal de error un mensaje especificando que se ha cometido un error a la hora de realizar la operación de multiplicación.

```
void mat_dot_productf(matrix m, double coeficent, matrix *result);
```

La función **mat_dot_productf** realiza el producto escalar entre una matriz `m` y un valor `coeficent` de la misma forma que se podría hacer en otros lenguajes de programación que contienen la implementación de matrices como Python o Matlab. El resultado queda guardado como en todas las demás operaciones aritméticas en la matriz apuntada por el puntero `result` (que debe ser alocada previamente con el método **mat_alloc**). La única restricción respecto al tamaño que deben tener las matrices `m` y `*result` es que deben ser del mismo tamaño. Analíticamente podemos observar usando `coeficent` como λ que el comportamiento de la función es:

$$\underset{3\times 2}{\overset{m}{\begin{bmatrix} a_1 & a_4 \\ a_2 & a_5 \\ a_3 & a_6 \end{bmatrix}}} \times \lambda = \underset{3\times 2}{\overset{*\,result}{\begin{bmatrix} \lambda a_1 & \lambda a_4 \\ \lambda a_2 & \lambda a_5 \\ \lambda a_3 & \lambda a_6 \end{bmatrix}}}$$

```
double mat_column_sum(matrix m, int col);
```

El método **mat_column_sum** realiza el sumatorio de todos los valores guardados en la columna `col` de la matriz `m`, así devolviendo como número real en coma flotante el resultado de este. Es importante no pasarle como parámetro `col` un valor mayor al número de columnas dentro de la matriz `m`, esto puede causar un error de desbordamiento y terminar con la ejecución del código o devolver resultados erróneos.

$$mcs(M,j) = \sum_{i=0}^{M_r} M(i,j)$$

Siendo M_r el número de filas en la matriz M, y j la columna `col` a sumar por completo.

```
matrix* mat_fromarray(int length, double array[]);
```

La rutina **mat_fromarray** permite transformar una cadena de números reales `array` en una matriz de longitud `length` vector con una fila y devuelve un puntero apuntando a la dirección de memoria donde se ha reservado el espacio dinámico de esta. Es importante usar el método **mat_free** cuando se deje de usar la matriz creada.

```
double* mat_toarray(matrix m);
```

El método **mat_toarray** funciona de forma inversa al ya descrito **mat_fromarray**, dada una estructura tipo `matrix m`, aloca espacio dinámico para una cadena de reales en punto flotante y devuelve la dirección a esta. Como se sigue usando memoria dinámica es recomendable liberar la matriz si ya no se va a volver a usar con método **mat_free** y cuando se acabe de usar la cadena de reales llamar a `free` de la stdlib.h para liberar esta también y evitar posibles colapsos del heap.

Ejemplos de código

En esta sección se mostrarán varios ejemplos de uso de la librería con el fin de mostrar al programador que vaya a hacer uso de este código de ejemplo por el cual poder guiarse.

Programa 1. Suma de dos matrices

```
#include <stdio.h>
#include <stdlib.h>
#include "matrix.h"

int main(int argc, char const *argv[])
{
    /* Se crean 2 matrices de tamano 4x3 */
    matrix * mat1 = mat_alloc(4,3);
    matrix * mat2 = mat_alloc(4,3);

    /* Establecemos valores aleatorios para todas la entradas de la matriz1 */
    srand(25);
    mat_randf(*mat1);

    /* La segunda matriz la transformamos sus valores a todo unos */
    for (int i = 0; i < mat2->rows; i++)
    {
        for (int j = 0; j < mat2->cols; j++)
        {
            mat_set_number(*mat2,i,j,1.0);
        }
    }

    /* Sumamos los mat1 y mat2 */
    matrix * mat3 = mat_alloc(4,3);
    mat_sumf(*mat1,*mat2,mat3);

    /* Imprimimos valores de ambas matrices y el resultado por consola */
    mat_print(*mat1);
    mat_print(*mat2);
    mat_print(*mat3);

    /* Liberamos la memoria alocada a las tres matrices */
    mat_free(mat1);
    mat_free(mat2);
    mat_free(mat3);

    return 0;
}
```

Programa 2. Multiplicación matricial y multiplicación escalar

```
#include <stdio.h>
#include <stdlib.h>
#include "matrix.h"

int main(int argc, char const *argv[])
{
    /* Se crean 2 matrices de tamano 2x5 y 5x3 */
    matrix * mat1 = mat_alloc(2,5);
    matrix * mat2 = mat_alloc(5,3);

    /* Establecemos valores aleatorios ara todas la entradas de ambas matrices */
    srand(25);
    mat_randf(*mat1);
    mat_randf(*mat2);

    /* Multiplicamos los mat1 y mat2 */
```

```
    matrix * mat3 = mat_alloc(2,3);
    mat_productf(*mat1,*mat2,mat3);

    /* Imprimimos valores de ambas matrices y el resultado por consola */
    mat_print(*mat1);
    mat_print(*mat2);
    mat_print(*mat3);

    /* Realizamos una multipplicacion escalar por 10 */
    mat_dot_productf(*mat3,10,mat3);

    /* Volvemos a imprimir el resultado para comprobar su funcionamiento */
    mat_print(*mat3);

    /* Liberamos la memoria alocada a las tres matrices */
    mat_free(mat1);
    mat_free(mat2);
    mat_free(mat3);

    return 0;
}
```

Programa 3. Operación de sumatorio ponderado usado en los algoritmos de feed-forward de las redes neuronales $A = X \times W + B$

```
#include <stdio.h>
#include <stdlib.h>
#include "matrix.h"

int main(int argc, char const *argv[])
{
    int num_inputs = 3;
    int num_neurons = 4;

    /* Se crean las matrices X, W, B y A */
    matrix *X = mat_alloc(1,num_inputs);
    matrix *W = mat_alloc(num_inputs,num_neurons);
    matrix *B = mat_alloc(1,num_neurons);
    matrix *A = mat_alloc(1,num_neurons);

    /* Se le agregan valores a X */
    *mat_seek(*X,0,0) = 1.5;
    *mat_seek(*X,0,1) = 1.0;
    *mat_seek(*X,0,2) = 2.25;

    /* Se inicializan los valores de W y B de forma aleatoria */
    mat_randf(*W);
    mat_randf(*B);

    /* Realizamos la multiplicacion X * W */
    matrix * temp = mat_alloc(1,num_neurons);
    mat_productf(*X,*W,temp);

    /* Sumamos el resultado de la multiplicacion con B */
    mat_sumf(*temp,*B,A);

    /* Liberamos la matriz temporal */
    mat_free(temp);

    /* Imprimimos el resultado */
    mat_print(*A);

    /* Liberamos las demas matrices */
    mat_free(X);
    mat_free(W);
    mat_free(B);
```

```
    mat_free(A);

    return 0;
}
```

Programa 4. Cálculo de la media de valores aleatorios usando `mat_column_sum`

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "matrix.h"

int main(int argc, char const *argv[])
{
    /* Establecemos el numero de valores a sumar*/
    int num_values = 100;

    /* Creamos un vector vertical de tamano 100 */
    matrix * mat = mat_alloc(num_values,1);

    /* Establecemos sus valores a valores aleatorios */
    time_t t;
    time(&t);
    srand((unsigned int) t);

    mat_randf(*mat);

    /* Visualizamos en consola los 100 valores */
    mat_print(*mat);

    /* Realizamos la suma de toda la columna */
    double sum = mat_column_sum(*mat,0);

    /* Dividimos entre el numero de valores para calcular la media*/
    double mean = sum/num_values;
    printf("Media = %f\n",mean);

    /* Liberamos la matriz */
    mat_free(mat);

    return 0;
}
```

Anexo B - Manual de uso de data_handler.h

La librería data_handler.h permite al programador cargar datos de archivos con sufijo .csv a una estructura de tipo parser_result y manipularlos para poder ser usados dentro del código y ahorrar tiempo en la programación del preprocesado de datos en C. La propia librería se encarga del manejo de los ficheros de datos y la lectura de ellos para que el programador no deba tener en cuenta estos factores y facilitar la programación de código.

Podemos dividir los métodos y operaciones de la librería en los siguientes campos

- **Obtención de información sobre los datos** de un fichero .csv:
  - Obtención del número de casos en una tabla
  - Obtención del número de atributos en una tabla
- **Lectura de datos** contenidos en un fichero .csv:
  - Parseador que recoge los datos y los guarda
- **Tratamiento de datos** y **asignación de memoria dinámica** para estos:
  - Asignar memoria para almacenar datos
  - Sustitución de un valor por otro en toda la tabla
- **Barajado de datos** de forma aleatoria:
  - Dividir los datos barajados en dos sets
  - Discriminar datos de forma aleatoria
  - Separar en clases binarias datos dependiendo de un valor entero

El tipo de datos `parser_result` implementado en la librería es definido de la siguiente forma:

```
typedef struct{
    double **data_input;
    double **data_output;
    char **atrib_names;
    int num_case;
    int num_in;
    int num_out;
}parser_result;
```

Donde `data_input` es un array de arrays que almacena toda la información de los casos que tomamos como entrada de datos. La cadena de cadenas de reales `data_output` sería los resultados esperados para cada caso de la tabla. El parámetro `atrib_names` es un array de cadenas que almacena los nombres de los atributos de una tabla. El numero entero `num_case` guarda el número de casos que hay dentro de una tabla. Otro entero llamado `num_in` cuenta el número de atributos que son entrada de datos independientes y, por último, `num_out` es el número de atributos que dependen de los independientes.

Además, el `data_handler` cuenta con la macro `LOG_PARSER` que muestra datos sobre la operación de lectura del fichero, si no se desea que aparezca por pantalla nada si se establece a 0 se desactiva.

Interfaz de uso (listado de métodos)

```
int get_number_cases(char filename[]);
```

El método **`get_number_lines`** se encarga de abrir el fichero cuya dirección es `filename` y contar el número de casos contenidos en el archivo de valores separados por comas para después devolver la cuenta total. La rutina maneja

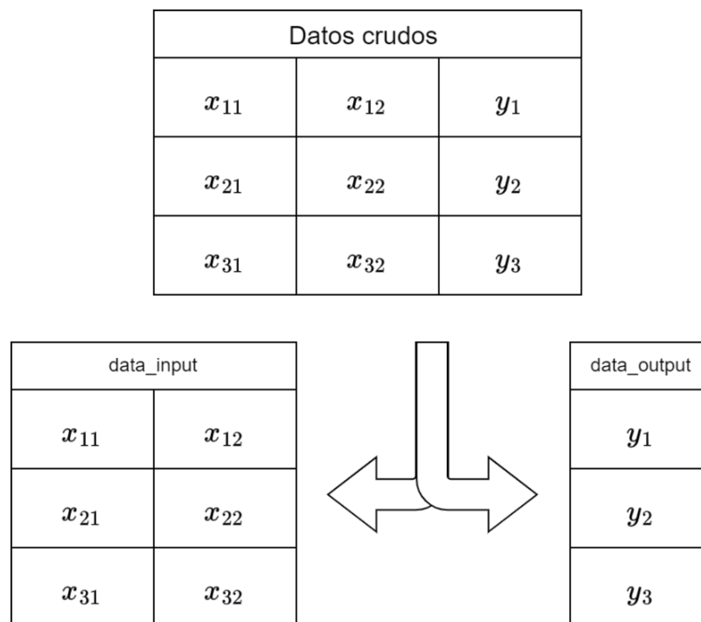
todas las operaciones de entrada y salida de datos con los ficheros de forma que el programador no tiene que encargarse de abrir un fichero y cerrarlo. En caso de que el nombre del fichero sea erróneo o no exista en la dirección especificada se mostrara un mensaje de error por pantalla y se detendrá la ejecución del código.

```
int get_number_atributes(char filename[]);
```

La función **get_number_atributes** mantiene un comportamiento similar a **get_number_cases**, este método cuenta el número de columnas o atributos que tiene una tabla en el fichero con la dirección `filename` y lo devuelve en forma de entero. Al igual que la rutina **get_number_cases**, esta maneja todas las operaciones de entrada de datos, por lo que el programador no debe manejar la apertura ni el cerrado del fichero .csv. En caso de que el nombre del fichero sea erróneo o no exista en la dirección especificada se mostrara un mensaje de error por pantalla y se detendrá la ejecución del código.

```
parser_result parse_data(char filename[], int num_inputs);
```

La rutina **parse_data** lee `num_inputs` número de casos de la tabla del fichero en la dirección `filename` y guarda todos los casos en una estructura `parser_result` dividiendo cada fila en dos partes: una de datos independientes `data_input` que sirven como atributos de entrada para cada caso y otra de atributos derivados o deducidos de los independientes `datas_output`, separando las entradas de las salidas para poder utilizar las listas de casos como conjuntos de datos de entrenamiento y de prueba. Los datos de entrada y salida de cada caso comparten el índice dentro de los dos arrays, de esta forma el programador puede asociar estos valores posteriormente al guardado en las dos listas. En la figura de abajo se explica de forma gráfica cómo funciona esta separación:



El campo `atrib_names` del `parser_result` se llena con los nombres de los atributos indicados en la primera fila del fichero .csv, y los valores `num_case`, `num_in` y `num_out` también se rellenan respectivamente con el número de casos recogido, el número de datos independientes por caso y el número de datos derivados de los independientes o esperados. Devolviendo finalmente un solo `parser_result` con todos los datos y los parámetros que la estructura maneja.

```
double **array_alloc(int rows, int cols);
```

El método **array_alloc** crea una cadena de cadenas de números reales de tipo double, el número de filas y columnas de las cadenas viene determinado por los parámetros `rows` y `cols` respectivamente. Este método usa memoria dinámica por lo que se debería después de usar las cadenas realizar un liberado de estas en memoria con el método `free` de la stdlib.h para prevenir desbordamientos en el heap que puedan causar comportamientos inesperados o el fallo del sistema.

```
parser_result random_trim(parser_result data, int size);
```

La rutina **random_trim** reduce una estructura `parser_result data` a una más pequeña de tamaño `size`, copiando el numero atributos de entrada `num_in` y de salida `num_out`. El método coje de forma pseudoaleatoria `size` número de casos guardados en la pareja de cadenas `data_input` y `data_output` de `data` y los introduce en un nuevo `parser_result`.

Cuidado al usar este método ya que puede duplicar varios casos. Recomendable cuando se trabaja con set de datos muy grandes y se quiere sacar una muestra pequeña de estos. Si se necesita estrictamente que no haya duplicados se puede usar el método **data_div** y escoger uno de los dos `parser_result` generados por este.

```
parser_result *data_div(parser_result in, int div_size);
```

La función **data_div** permuta y divide los datos leídos de un archivo en dos sets de tamaño `div_size` y `(num_cases - div_size)` permitiéndonos seleccionar `div_size` número de casos como set de entrenamiento y el resto de prueba. El barajado o permutación de los casos se hace antes de la separación y es una implementación del algoritmo de barajado de Fisher-Yates, ya que garantiza que no se dupliquen casos y, además, tiene una complejidad computacional lineal lo cual es asumible al trabajar con grandes sets de datos. El siguiente seudocódigo describe el comportamiento de Fisher-Yates siendo C un vector que contiene todos los casos:

1. Se da valor a $t = |C| - 1$
2. Mientras que $t > 0$:
    a. Se selecciona aleatoriamente un entero k entre 0 y t.
    b. Se cambia el valor de $C[k]$ por el de $C[t]$.
    c. $t = t - 1$.

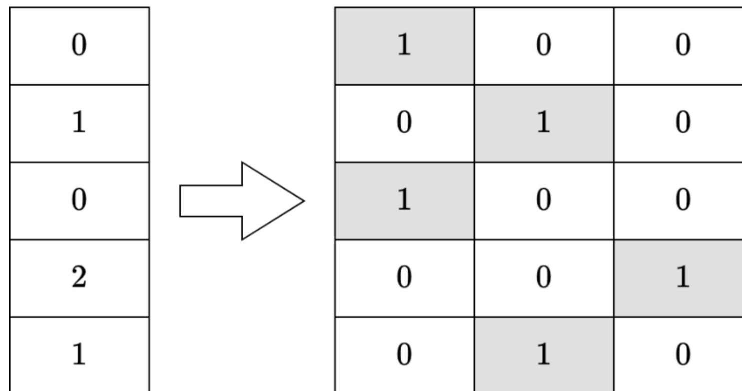
Una vez reorganizados los datos se separan en dos `parser_result` y se devuelven en forma de array. El primero contendrá `div_size` número de casos y el segundo el resto de los contenidos originalmente en `in`.

```
void change_all_values_for(double **data, int size_x, int size_y, double
actual, double new);
```

La rutina **change_all_values_for** recorre todos los valores almacenados en `data` y modifica los semejantes a `actual` al valor `new`, los parámetros `size_x` y `size_y` definen el tamaño de la array de arrays. Para lidiar con los problemas que puede suponer la comparación de dos números en representación de coma flotante, se usa la macro `EPSILON_DISTANCE` definida en la cabecera por defecto igual a 10^{-5} que establece la distancia entre dos valores en coma flotante máxima para que ambos se consideren semejantes.

```
double **from_integer_to_binary_classes(double **list, int num_case, int
*num_classes);
```

El método **`from_integer_to_binary_classes`** permite transformar una vector vertical de datos enteros (reales que representan enteros) `list` que establecen la clase de un caso a una matriz que contenga tantas columnas como posibles casos y se marca con un 1 la clase a la que pertenece un caso. Es necesario declarar el número de casos que tiene la lista con el parámetro `num_case` para que funcione de forma correcta. La siguiente imagen muestra como transforma el array de arrays en otro separando las clases por columnas:



El método finalmente cambia el valor que se encuentra en el puntero `num_clases` al número de clases que existían en el vector principal.

Ejemplos de código

En esta sección se mostrarán varios ejemplos de uso de la librería con el fin de mostrar al programador que vaya a hacer uso de este código de ejemplo por el cual poder guiarse.

Programa 1. Averiguar el número de clases y de atributos en una tabla .csv

```
#include <stdio.h>
#include <stdlib.h>
#include "data_handler.h"

int main(int argc, char const *argv[])
{
    /* Establecemos el nombre de fichero a leer */
    char filename[] = "FicheroDePrueba.csv";

    /* Obtenemos el numero de casos y atributos */
    int num_casos = get_number_cases(filename);
    int num_atrib = get_number_atributes(filename);

    /* Los mostramos por consola */
    printf("El fichero %s tiene %i casos con %i atributos\n"
        ,filename
        ,num_casos
        ,num_atrib);

    return 0;
}
```

Programa 2. Se leen los datos de un fichero y se dividen en set de entrenamiento y prueba

```
#include <stdio.h>
```

```c
#include <stdlib.h>
#include "data_handler.h"

int main(int argc, char const *argv[])
{
    /* Establecemos el nombre de fichero a leer */
    char filename[] = "FicheroDePrueba.csv";

    /* Leemos el archivo y guardamos los datos contenidos en pr */
    int num_inputs = get_number_atributes(filename)-1;
    parser_result pr = parse_data(filename,num_inputs);

    /* Dividimos los datos en dos set, de entrenamiento y prueba */
    parser_result* sets = data_div(pr,3);
    parser_result set_entrenamiento = sets[0];
    parser_result set_prueba = sets[1];

    /* ...... */

    return 0;
}
```

Programa 3. Se leen los datos de un fichero, se separan las clases de la columna de salida y se cambian todos los valores de las salidas que sean 0 por -1.

```c
#include <stdio.h>
#include <stdlib.h>
#include "data_handler.h"

int main(int argc, char const *argv[])
{
    /* Establecemos el nombre de fichero a leer */
    char filename[] = "FicheroDePrueba.csv";

    /* Leemos el archivo y guardamos los datos contenidos en pr */
    int num_inputs = get_number_atributes(filename)-1;
    parser_result pr = parse_data(filename,num_inputs);

    /* Separamos la columna final en casos */
    int num_clases = 0;
    double **separados =
       from_integer_to_binary_classes(pr.data_output,pr.num_case,&num_clases);

    /* Cambiamos el valor de 0 a -1 en todos los atributos de clase */
    change_all_values_for(separados,num_clases,pr.num_case,0.0,-1.0);

    /* Modificamos los valores del parser_result para utilizar los preprocesados */
    pr.data_output = separados;
    pr.num_out = num_clases;

    /* ...... */

    return 0;
}
```

Anexo C – Manual de la librería gml_nn.h

La librería gml_nn.h ayuda al programador especificar y crear redes de neuronas artificiales de tipo perceptrón multicapa MLP con todas las neuronas de una capa conectadas a todas las de la siguiente, que posteriormente pueden ser entrenadas para solucionar variedad de problemas mediante el algoritmo de retropropagación. La librería permite la selección de múltiples funciones de activación, funciones de coste o error y otros métodos de optimización para poder crear el modelo deseado en C que permita ser guardado y cargado en otros códigos. También cuenta con un conjunto de funciones que permiten el visualizado y validación de resultados de la red, aunque no es el principal objetivo.

La librería forma parte del trabajo de fin de grado de Álvaro González Méndez, estudiante de ingeniería informática en Escuela Técnica Superior de Ingenieros Informáticos de la Universidad Politécnica de Madrid. El código y la documentación de esta se encuentra en el repositorio de GitHub: https://github.com/itsTwoFive/gml_nn.

Para el uso del código, es necesario usar la librería *matrix.h* que está incluida en el repositorio y para ciertas funcionalidades de visualizado se requiere tener instalado en el equipo *gnuplot*.

Para la lectura de datos de archivos con el formato .csv se da la opción de usar la también implementada *data_handler.h* que también se encuentra en el repositorio de GitHub junto a su manual de uso.

Podemos dividir los métodos contenidos en la cabecera de la librería por su funcionalidad en los diferentes campos:

- Creación y configuración de redes de neuronas artificiales:
  - Creación de la red.
  - Configuración de los parámetros de inicialización de variables.
  - Configuración de los parámetros de entrenamiento.
  - Configuración de la función de error de la red.
  - Configuración del optimizador usado por la red.
  - Configuración de las funciones de activación de las capas.
- Entrenamiento y ajuste de las variables de la red:
  - Entrenamiento simple de una sola época.
  - Entrenamiento de la red en varias épocas.
- Alimentado de la red y cálculo del error cometido por esta:
  - Operación de alimentado “feedforward”.
  - Cálculo del error cometido respecto a una función de error.
- Operaciones de guardado del estado y cargado de diferentes redes:
  - Guardado en fichero del estado de la red.
  - Cargado de red guardada previamente.
  - Consulta de los parámetros y variables.
- Visualizado de los datos obtenidos por la red:
  - Graficado de datos establecidos como entrada de la red.
  - Graficado de las predicciones hechas por la red.
  - Visualización de graficas de error tras el entrenamiento.
- Funciones Extras para validación y manejado de datos:
  - Cálculo del acierto de una red.
  - Normalización de los datos de entrada.
  - Discriminador de clases.

La librería hace uso de varios tipos de estructuras de datos `struct` para trabajar con las redes de neuronas y los datos. El primer tipo es `layer`, que guarda la información respecto a las neuronas contenidas en una capa de la siguiente forma:

```
typedef struct {
    int layer_width;
    int act_func;
    double (*c_act_func)(double);
    double alpha_rate;
    matrix *W;
    matrix *oW;
    matrix *dW;
    matrix *vW;
    matrix *cW;
    matrix *out;
} layer;
```

Donde se guarda el valor `layer_width` que indica el número de neuronas que forman dicha capa (ancho de capa). Seguido, existe un entero `act_func` que indica la función de activación que usaran las neuronas de la capa, en caso de que se especifique una función personalizada se usara el puntero a función `c_act_func` para especificar el funcionamiento de esta. Aparte, si la función de activación se puede modular con algún parámetro extra este se almacenará en el número real `alpha_rate`. Finalmente, los punteros al tipo `matrix` definido en *matrix.h* de distintas matrices que contienen los valores de los pesos `W`, valores calculados en pasos intermedios como `oW`, `dW`, `vW`, y `cW`, y por último la salida de las neuronas contenidas en la capa `out`.

El tipo definido como `data`:

```
typedef struct{
    int num_cases_train;
    double **train_input;
    double **train_output;
    int num_cases_test;
    double **test_input;
    double **test_output;
} data;
```

Que almacena tanto los datos de entrada como los de salida de los conjuntos de datos de prueba y entrenamiento. El entero `num_cases_train` guarda el número total de casos en el conjunto de entrenamiento. Los punteros `train_input` y `train_output` apuntan respectivamente a las tablas donde se encuentran guardadas las entradas y salidas del conjunto de casos de entrenamiento. De la misma forma para con el conjunto de pruebas, se guardan en el entero `num_cases_test` y los punteros `test_input` y `test_output` el número de casos del conjunto de prueba, las entradas de este y sus salida. Nótese que las "matrices" o "tablas" de datos guardan los valores de números reales en un array de arrays en lugar del uso de la librería *matrix.h* ya que como no se necesita realizar operaciones matriciales y solo se quiere acceder a los datos, se evita el uso con el fin de ahorrar coste computacional.

La estructura más compleja y clave para el funcionamiento de la librería es el tipo `neural_net`, que define y almacena todos los parámetros globales (que se usan por igual en todas las capas) y la lista de capas. Aunque se pueden modificar estos campos accediendo a ellos como cualquier otra estructura, se

recomienda no hacerlo y usar los métodos de la librería que permiten cambiar esos valores. Esta viene dada por la siguiente especificación de tipo:

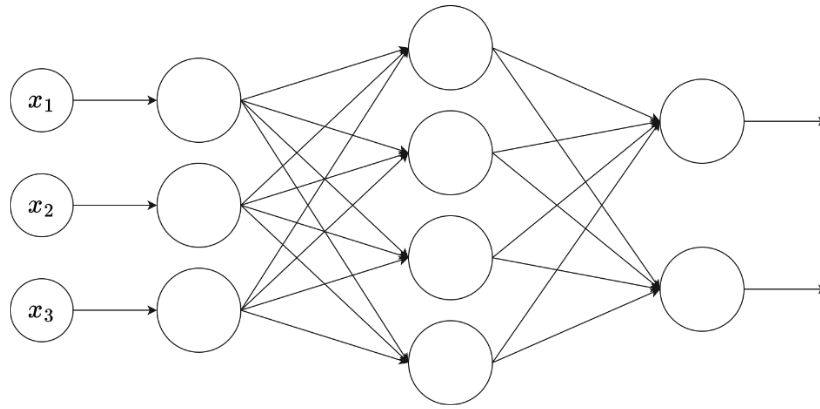
```
typedef struct{
    int input_count;
    int err_func;
    double (*c_err_func)(double,double);
    int layer_count;
    double learning_rate;
    double decay_rate;
    double epsilon_rate;
    int rand_seed;
    int batch_size;
    int cost_output;
    int console_out;
    layer **layers;
    data *dataset;
}neural_net;
```

El entero `input_count` almacena el numero da entradas que la red admite para su alimentación. Al igual que en las capas a la hora de seleccionar función de activación, en la red se almacena el valor que indica la función de error en el entero `err_func`, en caso de que se especifique que se quiere usar una función de error personalizada, el puntero a función `c_err_func` almacenara la dirección de dicha función. El siguiente número entero `layer_count`, indica el número de capas que existen en la red y el puntero de capas `layers`, almacena una lista de punteros a estructuras de tipo `layer` que almacenan la información acerca de las capas del modelo. El "learning rate" o tasa de aprendizaje se guarda en el valor de doble coma flotante `learning_rate`, al igual que la tasa de decadencia en `decay_rate` y el tamaño de salto usado para el método de diferencias finitas en `epsilon_rate`. En caso de que se quiera usar una semilla de generación personalizada para la inicialización de los pesos, esta se debe guardar en el entero `rand_seed`. Si el programador desea usar optimización por mini-lotes, el tamaño del lote se guarda en la variable entera `batch_size`. Las opciones de impresión de coste por consola y visualizado del error son almacenadas en `cost_output` y `console_out`. Por último, los datos necesarios para el entrenamiento y las pruebas cuando se usa la función de entrenamiento simplificado son almacenadas en la estructura de tipo `data` apuntada por `dataset`. El siguiente número entero `layer_count`, indica el número de capas que existen en la red.

Interfaz de uso (listado de métodos)

```
neural_net nn_create(int act_func, int layer_count, int layer_widths[], int input_count);
```

La rutina **`nn_create`** permite instanciar una red de neuronas que acepta `input_count` número de valores de entrada. Esta contara con un número fijo `layer_count` de capas ocultas y de salida (ya que la capa de entrada se genera dependiendo del número de entradas, no hace falta contar con ella en la inicialización de la red) siendo el tamaño especificado en la cadena de enteros `layer_widths` de forma ordenada. Es importante que el tamaño de la lista de enteros `layer_widths` sea igual al número de capas en la red. El primer parámetro `act_func` determina que función de activación se desea implantar en todas las capas por defecto. De forma gráfica, si se quisiera crear la siguiente red:



Se debería especificar un `input_count` de 3, un `layer_count` de 2 (número de capas que no sean de entrada), y como lista `layer_widths` se entregaría a la función [4,2]. La función devolverá la estructura creada tipo `neural_net`.

```
void layer_print(neural_net nn, int layer_num);
```

El método **`layer_print`** permite mostrar por consola los pesos y el sesgo de cada neurona de la capa número `layer_num` en la red `nn`. La salida del método muestra neurona a neurona por la consola de comandos el valor de los parámetros de esta siguiendo la siguiente estructura:

```
wN.0 0.000000
wN.1 0.000000
   …
wN.M 0.000000
bN   0.000000
```

En el caso de la neurona número N de la capa seleccionada, con $M+1$ numero de neuronas en la siguiente capa.

```
void nn_set_learning_rate(neural_net *nn, double learning_rate);
```

La función **`nn_set_learning_rate`** permite al programador cambiar el valor de la tasa de aprendizaje aplicada a toda la red de neuronas `nn` al descrito en la llamada `learning_rate`. Por defecto las redes de neuronas generadas con el método `nn_create` cuentan con un valor para la tasa de aprendizaje $\eta = 0.1$.

```
void nn_set_decay_rate(neural_net *nn, double decay_rate);
```

La rutina **`nn_set_decay_rate`** permite al programador cambiar el valor de la tasa de decadencia aplicada a toda la red de neuronas `nn` al descrito en la llamada `decay_rate`. Este valor por defecto viene asociado $\beta = 0.0$ ya que, si se le proporciona un valor positivo, el optimizador de la red de neuronas usara momento para acelerar el descenso de gradiente como es descrito en el método de la bola pesada o descenso con momento. Si se quiere volver a desactivar este, basta con volver a cambiarlo a `0.0f`.

```
void nn_set_epsilon(neural_net *nn, double epsilon_value);
```

El método **`nn_set_epsilon`** permite al programador cambiar el valor épsilon usado en el método de las diferencias finitas para aproximar la derivada de una función. Este valor `epsilon_value` es aplicado a toda la red de neuronas `nn`. Por defecto viene dado como $\varepsilon = 10^{-3}$.

```
void nn_set_batch_size(neural_net *nn, int size);
```

La función **`nn_set_batch_size`** permite al programador cambiar el modo en el que el optimizador de la red neuronal `nn` trabaja con los casos de entrenamiento durante este. Dependiendo del valor del entero `size`, la red de neuronas se comportará de diferente manera:

- `size` = 0 – La red usara todos los casos de prueba en cada entrenamiento.
- `size` = 1 – La red usara el descenso de gradiente estocástico (SGD) como optimizador, tomando solo una muestra cada entrenamiento.
- `size` > 1 – La red usara mini-lotes seleccionados de forma pseudoaleatoria durante el descenso de gradiente de cada entrenamiento.

Si se establece un valor para `size` mayor que el número de casos en el set de entrenamiento se causara un error por desbordamiento de memoria.

```
void layer_set_act_func(neural_net nn, int layer_pos, int act_func);
```

El método **`layer_set_act_func`** permite al programador establecer una función de activación denotada por su macro en el campo `act_func` a la capa número `layer_pos` de la red de neuronas `nn`. Existen varios tipos de funciones de activación preprogramadas en la librería, si se quiere usar una de estas, se debe de pasar como parámetro `act_func` una de las siguientes macros:

`ACT_NONE` – Indica que no se desea emplear ninguna función de activación en la capa.
`ACT_SIGMOID` – Indica que se desea utilizar la función logística o sigmoidal.
`ACT_TANH` – Indica que se desea utilizar la función tangente hiperbólica o tanh.
`ACT_OPSIGMOID` – Indica que se quiere usar una variante optimizada de la función logística para el uso de datos con un rango [-1, 1] de salida esperada. Esta función viene dada por la siguiente expresión:

$$\sigma(x) = 1.7159 \tanh\left(\frac{2}{3}x\right)$$

`ACT_RELU` – Indica que se quiere usar el rectificador ReLU o unidad de rectificador lineal como función de activación.
`ACT_LRELU` - Indica que se quiere usar el rectificador Leaky ReLU o unidad de rectificador lineal con fuga como función de activación. Esta es una variante del ReLU que toma un valor α como segundo parámetro para modular la parte negativa de la función. Si se quiere modificar este valor que viene por defecto como $\alpha = 0.1$ se debe usar el método `layer_set_alpha`.
`ACT_SOFTPLUS` – Indica que se quiere usar la función softplus (aproximación suavizada de ReLU).
`ACT_HEAVISIDE` – Indica que se quiere usar la función escalón. No recomendable su uso ya que al carecer de derivada el entrenamiento falla.

Existe también la macro **`ACT_CUSTOM`**, que indica que se está usando una función personalizada por el programador. Si se quiere hacer uso de una función personalizada se debe usar el método `layer_custom_act_func`.

```
void layer_set_alpha(neural_net nn, int layer_pos, double alpha);
```

La función **`layer_set_alpha`** permite establecer el valor α usado como modulador en funciones de activación como Leaky ReLU que necesitan este valor. Por defecto $\alpha = 0.1$, pero si se usa esta función, puede establecerse al valor en punto flotante `aplha` a la capa número `layer_pos` de la red `nn`.

```
void layer_custom_act_func(neural_net nn, int layer_pos, double
(*func)(double));
```

La rutina **layer_custom_act_func** permite especificar un puntero func a otro método que use como entrada un solo parámetro para que sea usado como función de activación para la capa layer_pos de la red de neuronas nn. Como no se puede saber analíticamente la derivada de esta, la red usara el método de las diferencias finitas para calcular la derivada de esta usando el valor ε establecido con la función nn_set_epsilon. El método de diferencias finitas calcula la derivada de la siguiente forma:

$$f'(x) = \lim_{\varepsilon \to 0} \frac{f(x+\varepsilon) - f(x)}{\varepsilon}$$

A parte, se actualizará el entero que indica la función de activación a ACT_CUSTOM.

```
void nn_set_err_func(neural_net *nn, int err_func);
```

El método **nn_set_err_func**, al igual que layer_set_act_func, permite al programador establecer una entre las ya predefinidas funciones de error descritas por el entero err_func a toda la red de neuronas nn. Las funciones de error que el programador puede usar vienen definidas por los siguientes macros:

ERR_SQRDIFF – Es el valor por defecto al crear una red, esta indica que se quiere usar el método de la media de errores cuadráticos.
ERR_HSQRDIFF – Indica que se quiere utilizar la mitad de la media de errores cuadráticos.
ERR_SIMPDIFF – Indica que se quiere usar la media de errores simples. Al igual que la función de activación de escalón, no es recomendable su uso ya que no sirve para entrenar a la red.

Existe también la macro **ERR_CUSTOM**, que indica que se está usando una función personalizada por el programador. Si se quiere hacer uso de una función de error personalizada se debe usar el método nn_custom_err_func.

```
void nn_custom_err_func(neural_net *nn, double (*func)(double,double));
```

La rutina **nn_custom_err_func** permite especificar un puntero func a otro método que use como entrada de dos parámetros (valor obtenido y esperado) para que sea usado como función de error en toda la red de neuronas nn. Como no se puede saber analíticamente la derivada de esta, la red usara el método de las diferencias finitas para calcular la derivada de esta usando el valor ε establecido con la función nn_set_epsilon. El método de las diferencias finitas se calcula de igual forma que en el método layer_custom_act_func. A parte, se actualizará el entero que indica la función de error a ERR_CUSTOM.

```
void nn_set_rand_seed(neural_net *nn, int seed);
```

La rutina **nn_set_rand_seed** permite establecer una semilla seed para la inicialización del valor de los pesos de toda la red neuronal nn. Si se establece como valor 0, la semilla se generará en función a la fecha y el tiempo de la maquina donde corra el código.

```
void nn_weight_randf(neural_net *nn);
```

El método **nn_weight_randf** inicializa todos los pesos de la red nn de forma pseudoaleatoria, usando el método rand de la librería estándar de C. Si se desea cambiar el valor de la semilla de generación, es recomendable hacerlo mediante nn_set_rand_seed en vez de usar el método srand porque así permite a la hora de guardar la red en un fichero también encapsular este valor.

```
matrix *cost(neural_net nn, int data_length, double **data, double
**results);
```

La función **cost** permite calcular el coste usando la función de error establecida en la red nn. Esta necesita como entrada el conjunto de datos de entrada data y los resultados esperados de esta results. También es necesario especificar el tamaño del conjunto.

```
matrix *feed_forward(neural_net nn, double data[], int data_size);
```

La rutina **feed_forward** permite alimentar a la re de neuronas nn con los datos data para un caso específico. Es importante especificar en el campo data_size el número de atributos con los que cuenta el caso con el que estamos alimentando a la red. Tras haber realizado todas las operaciones necesarias, el método devuelve un tipo matrix de tamaño $1 \times N$ siendo N el numero de neuronas en la última capa. También se puede recoger el resultado del último **feed_forward** accediendo al atributo de tipo matrix out de la última capa layer de la red de neuronas nn.

```
void train_network_epoch(neural_net nn, int data_length, double** data,
double** results);
```

El método **train_network_epoch** permite realizar el entrenamiento de una sola época en base a los parámetros y valores establecidos previamente en la red de neuronas nn. Las entradas de la rutina de tipo doble puntero: data y results apuntan directamente al conjunto de datos con los que se quiere entrenar a la red por completo y los resultados esperados de cada uno de los casos. También es necesario especificar el número de casos con el que cuenta el conjunto por el valor data_length.

Este método es el recomendado para un aprendizaje más controlado y personalizado, ya que el programador puede acceder a cualquier dato entre época y época para establecer diferentes métodos de visualización o control de los resultados.

Si lo que se desea es una forma más sencilla y generalizada de entrenar a dicha red, es recomendable el uso de la función train_network, que permite hacer n número de entrenamientos con una sola llamada y contiene diferentes modos de visualización del estado del entrenamiento.

```
void nn_set_training_data(neural_net nn, int num_cases, double **train_input,
double **train_output);
```

La rutina **nn_set_training_data** permite cargar el conjunto de datos de entrenamiento a la red neuronal nn para su posterior entrenamiento usando el método de entrenamiento simplificado train_network. Es necesario especificar el número de casos en el entero num_cases y por supuesto pasar el conjunto de entradas del set y sus respectivas salidas a través de los dos punteros train_input y train_output respectivamente.

```
void nn_set_testing_data(neural_net nn, int num_cases, double **test_input,
double **test_output);
```

La rutina **nn_set_testing_data**, de la misma forma que nn_set_training_data permite cargar el conjunto de datos de pruebas a la red neuronal nn para su posterior cálculo de coste usando el método de entrenamiento simplificado train_network. Es necesario especificar el número de casos en el entero num_cases y por supuesto pasar el conjunto de entradas del set y sus

respectivas salidas a través de los dos punteros `test_input` y `test_output` respectivamente.

```
void train_network(neural_net nn, int epochs, int print_cost_each, int which_cost);
```

El método **`train_network`** facilita al programador la forma de entrenar a la red de forma simple. Su funcionamiento depende antes del cargado correcto de datos usando `nn_set_training_data` y `nn_set_testing_data`. Se debe especificar el número de épocas o iteraciones en el entero `epochs` y cada cuanto se quiere calcular el coste de la red durante entrenamiento en el entero `print_cost_each`. También se debe especificar con qué conjunto de datos se quiere imprimir el coste. Esto se hace usando una macro como entrada para el valor `which_cost`. Existen los siguientes modos:

`COST_NONE` – No se realizará ningún cálculo de coste durante el entrenamiento.
`COST_TRAIN` – Solo se calculará el coste de los datos de entrenamiento.
`COST_TEST` – Solo se calculará el coste de los datos de prueba.
`COST_BOTH` – Se calculará tanto el coste de los datos de prueba como los de entrenamiento por separado.

El mostrado de los datos por defecto se hace mediante la salida estándar donde se ejecute el código que usa la librería. Haciendo uso del método `nn_set_cost_output` este comportamiento puede ser cambiado, permitiendo la salida de datos en formato csv o un graficado usando *gnuplot.*

Cambiar la salida de los costes no impide que se siga mostrando por consola la información sobre estos. Tanto si se quiere silenciar como si solo se quiere mostrar la época en la que se encuentra la red durante el entrenamiento se debe usar el método `nn_set_console_out`.

```
void nn_set_cost_output(neural_net *nn, int cost_out);
```

La rutina **`nn_set_cost_output`** debe ser usada para indicar a la red de neuronas que se desea mostrar el resultado de los cálculos de coste. Dependiendo del valor entregado al parámetro entero `cost_out` la red de neuronas actuara de forma distinta durante la ejecución del método de entrenado simplificado `train_network`. Puede adquirir los siguientes comportamientos al usar las macros:

`COUT_ONLY_CONSOLE` – Valor por defecto. Indica que solo se quiere mostrar la información por la salida estándar.
`COUT_GNUPLOT` – Indica que se desea dibujar una gráfica mostrando los costes calculados por época. Si se calculan los de ambos conjuntos, se mostrarán en la misma grafica. Si la salida de la red de neuronas es múltiple, se calcula una media de los costes de todas las salidas para dibujar la media de coste.
`COUT_CSV` – Indica que se desea guardar los resultados del coste en una tabla de un fichero con formato csv. El nombre del fichero generado será `costs.csv`.

```
void nn_set_console_out(neural_net *nn, int console_out);
```

La función **`nn_set_console_out`** permite silenciar la salida de los costes por consola cuando se usa la función `train_network` sobre la red `nn`. Existen varios tipos de silenciado y estos se seleccionan dándole valor al entero `console_out` con una de las siguientes macros:

`PRT_CONSOLE` – Valor por defecto. Muestra todo cálculo de costes por la salida estándar.

`PRT_NOCONSOLE` – No muestra ningún mensaje de costes por la salida estándar.
`PRT_ONLYEPOCH` – Solo muestra el número de iteración en la que se encuentra el entrenamiento.

```
void nn_save(neural_net nn, char* name);
```

La rutina **nn_save** permite el guardado de la red apuntada por el puntero `nn` en un fichero llamado `name` son el sufijo .nn. Todas las redes son guardadas en la carpeta "saved" y siguen el siguiente formato:

```
Input Number: 2
Learning Rate: 0.10000000
Decay Rate: 0.00000000
Epsilon: 0.00100000
Batch Size: 0
Error Function: 1
Random Seed: 0
Number of Layers: 1
Layer Widths: 3

*Layer
       Width: 3
       Alpha Value: 0.10000000
       Activation Function: 5
  -Weights
       0.0000000000 0.0000000000 0.0000000000
       0.0000000000 0.0000000000 0.0000000000
       0.0000000000 0.0000000000 0.0000000000
```

Este tipo de ficheros permite consultar y editarlos parámetros de la red sin necesidad de ejecutar código. Para cargarlos basta con hacer uso del método `nn_load`. Cabe destacar que si se estaba usando alguna función de activación o de error personalizada habrá que volver a cargarla cuando se vuelva a cargar la red.

```
neural_net nn_load(char* filename);
```

La función **nn_load** como se puede intuir por su nombre carga una red de neuronas previamente guardada mediante el método `nn_save` por otro Código. Esta toma una cadena `filename` y busca dentro de la carpeta "saved" si existe alguna red con ese nombre. En caso de que no exista buscara tomando la cadena como una dirección relativa. Si tampoco encuentra el fichero .nn mostrara un error por consola y parara la ejecución del Código. Si el método encuentra la red a cargar en el sistema de ficheros devolverá una instancia `neural_net` con la información de la red de neuronas cargada.

Se debe tener en cuenta que si se estaba usando alguna función de activación o de error personalizada habrá que volver a cargarla cuando se vuelva a cargar la red usando el método de configuración correspondiente `nn_custom_err_func` o `layer_custom_act_func`.

```
void plot_2d_data_for_binary(double **data_in, double **data_out, int
num_cases, int num_out, double *ranges);
```

La rutina **plot_2d_data_for_binary** crea un proceso ejecutando *gnuplot* conectado mediante un pipe que muestra datos de un set con dos atributos dibujados con colores distintos dependiendo de la clase a la que pertenezcan. Se deben especificar dos punteros a los datos con sus correspondientes salidas `data_in` y `data_out`. También se debe pasar como parámetro `num_cases` el número de casos que se quiere dibujar y el número de clases a las que pueden

pertenecer los distintos casos `num_out`. Finalmente, también se debe entregar a la función un array de reales que especifican los límites de la gráfica `ranges`, siendo para el eje x el mínimo `ranges[0]` y el máximo `ranges[1]` y para el eje y `ranges[2]` y `ranges[3]`.

```
void show_areas_2d_plot(neural_net nn, int num_out, double step, double
*ranges);
```

El método **`show_areas_2d_plot`** al igual que `plot_2d_data_for_binary` muestra crea un pipe a otro proceso ejecutando *gnuplot* que mostrara un mapa de áreas en las que un clasificador usando una red neuronal clasificaría los puntos que se encuentren en dichas áreas. Se debe especificar la red de neuronas previamente entrenada `nn`, el número de clases que puede clasificar dicha red `num_out` y el salto entre punto y punto usado para calcular para las áreas `step` dentro de un rango `ranges`, siendo para el eje x el mínimo `ranges[0]` y el máximo `ranges[1]` y para el eje y `ranges[2]` y `ranges[3]`.

```
void minmax_normalization(double **data, int param, int size);
```

La rutina **`minmax_normalization`** realiza para el atributo en la posición `param` de todos los casos en el conjunto `data` de tamaño `size` una normalización de tipo Min Max siguiendo la siguiente formula:

$$minmax(X) = \frac{2(X - \min(X))}{\max(X) - \min(X)} - 1$$

Así estableciendo para todos los casos del conjunto en el atributo seleccionado un valor normalizado entre -1 y 1.

```
double single_binary_acurracy_rate(neural_net nn, double **input, int
data_size, double **expected_arr, double dist, int case_num);
```

El método **`single_binary_acurracy_rate`** provee una forma sencilla de calcular la tasa de acierto de una red de neuronas entrenada `nn`. Recibe como parámetros de entrada el puntero `input`, que apunta a los valores de entrada del conjunto de datos deseado para calcular el acierto, el puntero `expected_arr`, que igualmente apunta a los resultados esperados de dicho conjunto, el entero `data_size`, que es el número de atributos en un caso del conjunto, `case_num` que es el número de casos en el conjunto y por último el valor real en punto flotante `dist`, que es la distancia máxima entre el resultado obtenido y el esperado para considerarse semejantes.

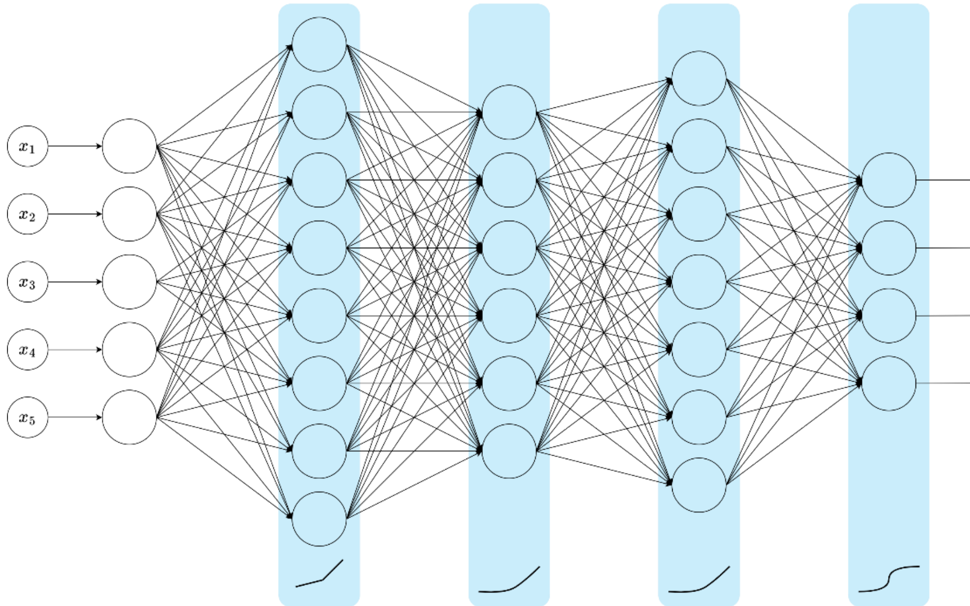
```
int choose_class(double *outputs, int num_out, int target);
```

La rutina **`choose_class`**, es un método selector de clase cuando se trabaja en la clasificación de más de dos clases. Recorre los resultados en el array de números reales outputs y escoge la clase que tenga el valor más cercano a un valor objetivo `target`. Es necesario especificar mediante la variable de entrada `num_out` el número de clases existentes (número de salidas de la red) a las que puede pertenecer el caso analizado por la red de neuronas `nn`.

Ejemplos de código

En esta sección se mostrarán varios ejemplos de uso de la librería con el fin de mostrar al programador que vaya a hacer uso de este código de ejemplo por el cual poder guiarse.

Programa 1. Creación de la siguiente red de neuronas con sus respectivas funciones de activación para cada capa:




```
#include <stdio.h>
#include <stdlib.h>
#include "gml_nn.h"
#include "data_handler.h"

int main(int argc, char const *argv[])
{
    // Creacion de la red
    int widths[] = {8,6,7,4};
    neural_net nn =nn_create(ACT_SOFTPLUS,4,widths,5);

    // Inicializacion de los pesos de forma aleatoria
    nn_weight_randf(&nn);

    // Seleccion de funciones de activacion para las capas 1 y 4
    layer_set_act_func(nn,1,ACT_LRELU);
    layer_set_act_func(nn,4,ACT_SIGMOID);

    return 0;
}
```

Programa 2. Creación de una red de neuronas con métodos como funciones de activación y error.

```
#include <stdio.h>
#include <stdlib.h>
#include "gml_nn.h"
#include "data_handler.h"

double funcion_de_prueba_act(double a){
    return 7.0f/(1.0f+ exp(-a));
}

double funcion_de_prueba_err(double a, double b){
    return pow(a-b,4);
```

```
}

int main(int argc, char const *argv[])
{
    // Creacion de la red
    int widths[] = {4,2};
    neural_net nn =nn_create(ACT_TANH,2,widths,3);

    // Establece funcion de activacion personalizada
    layer_custom_act_func(nn,1,&funcion_de_prueba_act);
    layer_custom_act_func(nn,2,&funcion_de_prueba_act);

    // Establece funcion de error personalizada
    nn_custom_err_func(&nn,&funcion_de_prueba_err);

    return 0;
}
```

Programa 3. Creación de una red de neuronas con 2 neuronas ocultas y 1 de salida que se entrena para resolver la tabla de verdad de XOR.

```
#include <stdio.h>
#include <stdlib.h>
#include "gml_nn.h"
#include "data_handler.h"

int main(void)
{
    // Seleccionamos el fichero que contenga el set de datos
    char filename[]  = "../datasets/xor.csv";

    int num_output = 1;
    int num_input = get_number_atributes(filename)-num_output;

    // Lo leemos y guardamos en memoria los datos
    parser_result data = parse_data(filename,num_input);

    // Normalizamos los datos usando Minmax
    for (int i = 0; i < num_input; i++)
    {
        minmax_normalization(data.data_input,i,data.num_case);
    }

    // Para mejor clasificacion cambiamos los 0 de los outputs por -1
    change_all_values_for(data.data_output,data.num_out,data.num_case,0.0,-1.0);

    // Estalecemos los tamanos de la red
    int lay_count[] = {2,1};

    // Creamos la Red Neuronal
    neural_net nn = nn_create(ACT_OPSIGMOID,2,lay_count,2);

    // Configuramos la tasa de aprendizaje
    nn_set_learning_rate(&nn,0.3);

     // Dar valor a la semilla del generador de pesos iniciales e inicializar estos
pesos
    nn_set_rand_seed(&nn,23241);
    nn_weight_randf(&nn);

    // Agregamos los datos de entrenamiento y prueba a la red
    nn_set_training_data(nn,4,data.data_input,data.data_output);
    // Entrenamos la red
    int epoch = 50;
    int print_each = 1;
    train_network(nn,epoch,print_each,COST_TRAIN);
```

```
    //Podemos calcular la tasa de acierto discriminando si >0 o si <0
    double acurracy = single_binary_acurracy_rate(nn,
        data.data_input,2,data.data_output,.5,4);

    printf("Acurracy: %f\n",acurracy);
    return 0;
}
```

Programa 4. Creación de una red de neuronas con 2 capas ocultas de (tamaño 16 y 8) y 1 neurona en la capa de salida. Tras ello se le entregan datos de entrenamiento y de prueba y se realiza el entrenamiento de 1000 épocas. Después muestra la tasa de acierto.

```
#include <stdio.h>
#include <stdlib.h>
#include "gml_nn.h"
#include "data_handler.h"

int main(void)
{
    // Seleccionamos el fichero que contenga el set de datos
    char filename[]  = "../datasets/Diabetes.csv";

    int num_output = 1;
    int num_input = get_number_atributes(filename)-num_output;

    // Lo leemos y guardamos en memoria los datos
    parser_result data = parse_data(filename,num_input);

    // Normalizamos los datos usando Minmax
    for (int i = 0; i < num_input; i++)
    {
        minmax_normalization(data.data_input,i,data.num_case);
    }

    // Para mejor clasificacion cambiamos los 0 de los outputs por -1
    change_all_values_for(data.data_output,data.num_out,data.num_case,0.0,-1.0);

    // Dividimos los datos en set de entrenamiento y pruebas
    double div_num = 75.0/100.0;
    parser_result * div_data = data_div(data,(int) data.num_case*div_num);
    parser_result train_data = div_data[0];
    parser_result test_data  = div_data[1];

    // Estalecemos los tamanos de la red
    int lay_count[] = {16,8,1};

    // Creamos la Red Neuronal
    neural_net nn = nn_create(ACT_SOFTPLUS,3,lay_count,num_input);

    // Configuramos tamano del minilote
    nn_set_batch_size(&nn,40);

    // Configuramos la tasa de aprendizaje
    nn_set_learning_rate(&nn,0.01);

    // Dar valor a la semilla del generador de pesos iniciales e inicializar estos
pesos
    nn_set_rand_seed(&nn,20201);
    nn_weight_randf(&nn);

    // Agregamos los datos de entrenamiento y prueba a la red
    nn_set_training_data(nn,train_data.num_case,train_data.data_input,train_data.data
_output);
```

```
    nn_set_testing_data(nn,test_data.num_case,test_data.data_input,test_data.data_out
put);

    // Entrenamos la red
    int epoch = 1000;
    int print_each = 10;
    nn_set_cost_output(&nn,COUT_GNUPLOT);
    train_network(nn,epoch,print_each,COST_BOTH);

    //Podemos calcular la tasa de acierto discriminando si >0 o si <0
    double acurracy = single_binary_acurracy_rate(nn,
        test_data.data_input,num_input,test_data.data_output,1,test_data.num_case);

    printf("Acurracy: %f\n",acurracy);
    return 0;
}
```

Programa 5. Creación de red que clasifica en 4 clases distintas puntos en un espacio de 2 dimensiones.

```
#include <stdio.h>
#include <stdlib.h>
#include "gml_nn.h"
#include "data_handler.h"

int main(void)
{
    // Seleccionamos el fichero que contenga el set de datos
    char filename[]  = "../datasets/4class.csv";

    int num_output = 4;
    int num_input = get_number_atributes(filename)-num_output;

    // Lo leemos y guardamos en memoria los datos
    parser_result data = parse_data(filename,num_input);

    // Para mejor clasificacion cambiamos los 0 de los outputs por -1
    change_all_values_for(data.data_output,data.num_out,data.num_case,0.0,-1.0);

    // Creamos la Red Neuronal
    int lay_count[] = {12,num_output};
    neural_net nn = nn_create(ACT_OPSIGMOID,2,lay_count,num_input);

    // Configuramos diversos parametros
    nn_set_decay_rate(&nn,0.0);
    nn_set_learning_rate(&nn,0.01);

    // Inicializacion de los pesos
    nn_weight_randf(&nn);

    // Podemos configurar tambien distintas Funciones de activacion para cada capa
    layer_set_act_func(nn,2,ACT_OPSIGMOID);

    // Entrenamos la red
    int epoch = 2e4;
    int print_each = 1000;
    for (int i = 0; i < epoch; i++)
    {
        train_network_epoch(nn,data.num_case,data.data_input,data.data_output);
        if(i%print_each == 0){
           matrix * act_cost = cost(nn,data.num_case,data.data_input,data.data_output);
            printf("EPOCA %i Coste de entrenamiento: ",i);
            mat_print(*act_cost);
            mat_free(act_cost);
        }
    }
```

```
    //Podemos calcular la tasa de acierto

    plot2DDataForBinary(data.data_input,data.data_output,data.num_case,data.num_out);
    showAreas2DPlot(nn,data.num_out);

    return 0;
}
```

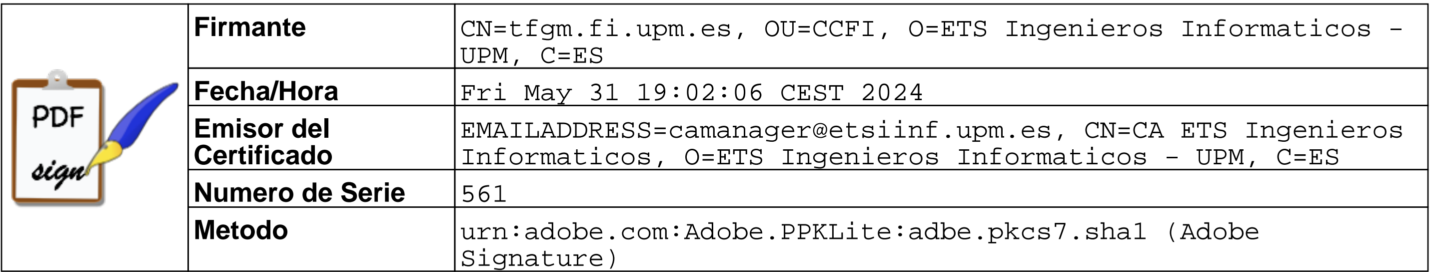
Programa 6 y 7. Creación de una red de neuronas con 2 neuronas ocultas y 1 de salida que se entrena para resolver la tabla de verdad de XOR. Una vez entrenado se guarda en "xor_model.nn" y el segundo programa lo carga y comprueba el resultado obtenido de la entrada [1,1].

```
#include <stdio.h>
#include <stdlib.h>
#include "gml_nn.h"
#include "data_handler.h"

int main(void)
{
    // Seleccionamos el fichero que contenga el set de datos
    char filename[]  = "../datasets/xor.csv";

    int num_output = 1;
    int num_input = get_number_atributes(filename)-num_output;

    // Lo leemos y guardamos en memoria los datos
    parser_result data = parse_data(filename,num_input);

    // Estalecemos los tamanos de la red
    int lay_count[] = {2,1};

    // Creamos la Red Neuronal
    neural_net nn = nn_create(ACT_OPSIGMOID,2,lay_count,2);

    // Configuramos la tasa de aprendizaje
    nn_set_learning_rate(&nn,0.3);

     // Dar valor a la semilla del generador de pesos iniciales e inicializar estos
pesos
    nn_set_rand_seed(&nn,23241);
    nn_weight_randf(&nn);

    // Agregamos los datos de entrenamiento y prueba a la red
    nn_set_training_data(nn,4,data.data_input,data.data_output);
    // Entrenamos la red
    int epoch = 50;
    int print_each = 1;
    train_network(nn,epoch,print_each,COST_TRAIN);

     //Podemos calcular la tasa de acierto discriminando si >0 o si <0
    double acurracy = single_binary_acurracy_rate(nn,
        data.data_input,2,data.data_output,.5,4);

     nn_save(nn,"xor_model");


    return 0;
}
```

```
#include <stdio.h>
#include <stdlib.h>
#include "gml_nn.h"
```

```c
#include "data_handler.h"

int main(void)
{
    // Cargamos la red guardada como xor_model
    neural_net nn = nn_load("xor_model");

    // Cargamos el caso a probar
    double xor_case[] = {1,1};

    // Ejecutamos el feedforward sobre la red con el caso
    matrix *result = feed_forward(nn,xor_case,2);

    // Mostramos por consola el resultado de la alimentacion
    mat_print(*result);

    // Se comprueba que el resultado contenido en la neurona es el esperado
    if(*mat_seek(*result,0,0) < 0.1){
        printf("El feedforward ha sido correcto\n");
    }
    else{
        printf("El feedforward ha sido erroneo\n");
    }
    return 0;
}
```

Este documento esta firmado por



	Firmante	CN=tfgm.fi.upm.es, OU=CCFI, O=ETS Ingenieros Informaticos - UPM, C=ES
	Fecha/Hora	Fri May 31 19:02:06 CEST 2024
	Emisor del Certificado	EMAILADDRESS=camanager@etsiinf.upm.es, CN=CA ETS Ingenieros Informaticos, O=ETS Ingenieros Informaticos - UPM, C=ES
	Numero de Serie	561
	Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)