Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

Librería Python para el aprendizaje y la implementación de redes neuronales.

Trabajo Final de Grado

Grado en Ingeniería Informática

Autor: Omar Caja García

Tutor: Carlos David Martínez Hinarejos

Curso 2019-2020

Resumen.

El presente TFG consta de la realización de una memoria y una librería escrita en el lenguaje de programación Python para el desarrollo de redes neuronales.

Se desarrollarán los tipos de redes neuronales vistos a lo largo de la rama de computación en las asignaturas de SIN y APR: perceptrón (clasificador binario), redes neuronales de una sola capa (clasificador multiclase) y redes neuronales profundas.

Partiendo de los elementos más simples que componen una red, tales como el perceptrón, se irá ampliando dicha librería con elementos más complejos hasta llegar a la construcción de redes neuronales más completas tales como las redes neuronales multicapa.

Dicha librería irá acompañada de un Jupyter Notebook que hará las veces de guía sobre el uso de ésta, explicando su funcionamiento de forma interactiva. Este *notebook* es una interfaz web que permite la ejecución de código Python en un navegador.

El principal objetivo de este TFG es didáctico, es decir, entender cómo se comporta una red neuronal y poder utilizarla en tareas de clasificación. En la actualidad existen infinidad de librerías para la implementación de redes neuronales, pero el objetivo no es aprender a usar una librería, la idea es crear una librería partiendo de cero para entender bien su funcionamiento.

Palabras clave: Python, Jupyter Notebook, red neuronal.

Abstract.

This FWD consists of the realization of a report and a library written in the Python programming language for the development of neural networks.

The types of neural networks seen throughout the computing branch in the SIN and APR subjects will be developed: perceptron (binary classifier), single-layer neural network (multiclass classifier), and deep neural networks.

Starting from the simplest elements that make up a network, such as the perceptron, this library will be expanded with more complex elements until reaching the construction of more complete neural networks, such as multilayer networks.

This library will be accompanied by a Jupyter Notebook that will serve as a guide on its use, explaining its operation interactively. This notebook is a web interface that allows Python code to be executed in a browser.

The main objective of this FWD is didactic, that is, to understand how a neural network behaves and how to use it in classification tasks. Currently there are countless libraries for the implementation of neural networks, but the objective is not to learn how to use a library, the idea is to create a library from scratch to understand its operation well.

Keywords: Python, Jupyter Notebook, neural network.

Índice.

Índice de figuras.

Índice de algoritmos.

1. Introducción.

Este trabajo final de grado, en adelante TFG, está centrado en el estudio de las redes neuronales artificiales, en concreto: perceptrón, redes neuronales de una sola capa (clasificador multiclase) y redes neuronales profundas. Además, se enfoca en el desarrollo de una librería con fines didácticos en el lenguaje de programación Python y un Jupyter Notebook sobre su implementación.

La aparición del concepto del perceptrón, desarrollado por Frank Rosenblatt en 1958 [1] basándose en la biología de la neurona propuesta por Santiago Ramón y Cajal y Charles Scott Sherrintong, ha propiciado el desarrollo de sistemas cada vez más complejos. Gran parte de estos sistemas se basan en la idea del perceptrón y se construyen a partir de éste.

La evolución de las redes neuronales y sus aplicaciones en el ámbito de la inteligencia artificial, y más concretamente en el campo del *Deep Learning,* han supuesto un gran cambio en la forma de plantear y resolver problemas que hace un par de décadas resultaban inimaginables, ya no solo en el sector de la informática. Sus aplicaciones hoy en día son muy diversas y extendidas. Algunos ejemplos como los siguientes pueden ilustrar su importancia: sistemas de apoyo al diagnóstico en medicina, aplicaciones en campañas electorales para dirigir mensajes a potenciales electores, el envío de publicidad de forma selectiva e individualizada usado por compañías como Amazon [2].

El Foro Económico Mundial realizó un anticipo en enero del 2019 de algunas de las tecnologías que formarán parte de la cuarta revolución industrial, también conocida como Revolución 4.0, y entre ellas se encuentra la inteligencia artificial [3]. Todo esto nos invita a pensar que la demanda de este tipo de sistemas a corto plazo aumentará y, por ende, la de los profesionales con capacidad para implementar dichos sistemas.

Actualmente existen múltiples librerías para la implementación de redes neuronales. Algunas de las más populares son desarrolladas y soportadas por empresas como Google, en el caso de TensorFlow [4] o Facebook, en el caso de PyTorch [5].

Tras esta breve contextualización se expondrá la motivación por la cual se decidió desarrollar este TFG.

1.1. Motivación.

Tras completar todas las asignaturas que componen el grado en ingeniería informática, troncales y específicas, estas últimas correspondientes a la rama de computación, hubo dos asignaturas que particularmente llamaron mi atención: EDA (Estructuras de datos y algoritmos), correspondiente al segundo cuatrimestre del segundo curso, y SIN (Sistemas inteligentes), correspondiente al primer cuatrimestre del tercer curso.

En EDA se estudian y se implementan algunas de las técnicas de diseño de algoritmos más comunes como *divide y vencerás* y una librería de estructuras de datos (listas, diccionarios, árboles y grafos) en el lenguaje de programación Java. Esta asignatura hizo que tomara la decisión de estudiar la rama de computación.

Crear esta librería de estructuras de datos durante la asignatura de EDA partiendo de cero me hizo comprender cómo funcionan realmente las estructuras de datos que estaba

implementando y qué diferencia existe, por ejemplo, entre una lista enlazada o un *array* y por qué es mejor usar una u otra según el caso.

El propósito de la librería que implementamos durante la asignatura no era el de reemplazar o competir con las librerías de estructuras de datos que cualquier lenguaje de programación tiene implementadas. El objetivo era poder conocer la base subyacente de las estructuras de datos más comunes para poder elegir de forma razonada qué estructura de datos utilizar en cada momento, estructuras de datos que no habremos implementado nosotros pero que conoceremos.

Por otra parte, SIN es la asignatura que ha propiciado la temática del TFG. En esta asignatura se presenta y se estudia el algoritmo del perceptrón propuesto por Rosenblatt en 1957 [1] y su uso en tareas de clasificación multiclase. Fue un concepto que, por su simplicidad y potencia, me fascinó e hizo que investigase por mi cuenta sobre su historia, implementación, evolución, etc.

Tras estos conocimientos adquiridos durante en el grado, me pareció una buena forma de continuar mi desarrollo en el campo de las redes neuronales y la inteligencia artificial realizar una librería en el lenguaje de programación Python que implemente algunas de las redes neuronales artificiales más comunes, acompañándola de una guía sobre su uso e implementación en un Jupyter Notebook.

Como pasaba con la librería implementada en EDA, esta librería no pretende competir con las librerías de redes neuronales actuales. Es, por tanto, un ejercicio didáctico para que quien tenga interés en esta temática pueda comprender, de forma didáctica, los fundamentos en la implementación de una red y las diferencias entre las distintas redes. Así, cuando se tenga que hacer uso de cualquier librería de redes neuronales, quien haya usado esta librería podrá razonar qué tipo de red es mejor utilizar y sus ventajas respecto a otras redes.

1.2. Objetivos.

Una vez finalizado el TFG los objetivos que se pretenden alcanzar son los siguientes:

- La creación de una memoria que contendrá el desarrollo del TFG.
- La creación de la librería de código abierto en Python con la implementación de, al menos, los siguientes sistemas: perceptrón, redes neuronales de una sola capa (clasificador multiclase) y redes neuronales profundas.
- La creación de una guía sobre el uso de la librería y la implementación de los diferentes tipos de redes neuronales en un Jupyter Notebook.
- Un repositorio público en GitHub que contendrá dicha librería.

1.3. Impacto esperado.

El impacto esperado con la realización de este TFG a nivel personal es ampliar mis conocimientos en el área de la inteligencia artificial y más concretamente en las redes neuronales, así como mejorar y adquirir destreza en el uso de uno de los lenguajes de programación con más crecimiento en los últimos años como es Python. Además, se busca

tener una base sólida a la hora de utilizar librerías para la implementación de redes neuronales.

Por otra parte, me gustaría que esta librería y el Jupyter Notebook puedan servir a otras personas para su crecimiento y desarrollo en el ámbito de las redes neuronales, de forma que puedan acceder al repositorio, descargar la librería y la documentación, e incluso que sigan desarrollando el proyecto. Es decir, que la gente pueda colaborar, implementar y documentar nuevas arquitecturas de redes, añadir nuevas funciones a las redes ya existentes, etc.

En otras palabras, se pretende que el proyecto no termine con este TFG, si no que pueda seguir creciendo con esta filosofía didáctica. Esto implica que en paralelo a cada implementación haya una documentación en el Jupyter Notebook sobre cómo se ha hecho, porque lo importante no es la librería, sino la comprensión de los fundamentos básicos que componen una red. Librerías hay muchas y en el ámbito profesional se usarán unas u otras, pero sí es interesante comprender los elementos comunes y subyacentes a todas ellas.

1.4. Estructura de la memoria.

El contenido en el que se ha estructurado el presente documento es el siguiente.

- 1. Introducción: en este apartado se exponen las motivaciones, los objetivos y el impacto esperado, durante y tras el desarrollo del proyecto.
- 2. Estado del arte: se hace una revisión de las librerías utilizadas en inteligencia artificial más relevantes del momento, resaltando sus aspectos positivos y negativos y destacando qué ventajas puede aportar la librería desarrollada en el presente TFG.
- 3. Redes neuronales artificiales:
    - 3.1. Historia: en esta sección se realiza una contextualización y un recorrido a lo largo de la historia de las redes neuronales artificiales.
    - 3.2. Definiciones: se realiza una definición de los elementos que componen un sistema neuronal artificial.
    - 3.3. Característica de las redes neuronales artificiales: se exponen algunos puntos que evidencian las bondades de las redes neuronales artificiales.
    - 3.4. Sistemas neuronales implementados: se realiza una definición y formalización matemática de cada uno de los sistemas desarrollados.
- 4. Diseño de la librería: en este capítulo se describen las diferentes tecnologías de las que se ha hecho uso para la implementación de la librería.
- 5. Desarrollo de la librería: en este apartado se describe la arquitectura de la librería, así como el lenguaje de programación utilizado y se detalla cada uno de los módulos implementados.
- 6. Creación del Jupyter Notebook: se describe cómo ha sido el proceso de documentación de la librería y los diferentes formatos en los que esta documentación está disponible.

- 7. Conclusiones: se exponen qué objetivos iniciales del proyecto han sido conseguidos y cuales no y qué aprendizajes he obtenido durante la realización del proyecto.
- 8. Trabajos futuros: se enumeran qué posibles trabajos o desarrollos futuros pueden realizarse tras la finalización del proyecto y qué líneas de desarrollo no sería conveniente que se siguiesen.
- 9. Bibliografía: en este apartado se enumeran las fuentes consultadas durante el desarrollo del proyecto.

2. Estado del arte.

En este apartado se realizará una revisión de las librerías actuales más relevantes sobre redes neuronales artificiales utilizadas en Python. Se documentarán los aspectos más significativos de cada una de ellas y, por último, se expondrá qué diferencia la librería que se implementará en este TFG con el resto de las librerías descritas.

Debido al incremento de la demanda en la utilización de la inteligencia artificial, existen infinidad de librerías para la implementación de redes neuronales; por ello, se enumeran las librerías más relevantes y se hará un estudio más detallado de aquellas más utilizadas.

Tal y como se expone en el apartado 4.1.1, Python es el lenguaje de programación utilizado para la creación de la librería del TFG, ya que se trata del lenguaje de programación más utilizado en el ámbito de la inteligencia artificial según el *Community insights* publicado por GitHub en 2019 [6]. Este es el motivo por el cual el estudio del estado del arte solo comprende las librerías utilizadas en Python y no en otros lenguajes de programación.

En la actualidad conviven multitud de librerías para la implementación de modelos de aprendizaje automático. Las tres más populares de acuerdo con diferentes artículos publicados en los últimos años son: Keras, TensorFlow y PyTorch [7] [8].

En la Figura 2.1 podemos observar el interés en el último año en todo el mundo de los términos de las tres librerías enumeradas anteriormente junto con otras dos librerías utilizadas en Python. Esta figura hace evidente la diferencia de popularidad entre las tres primeras y el resto.

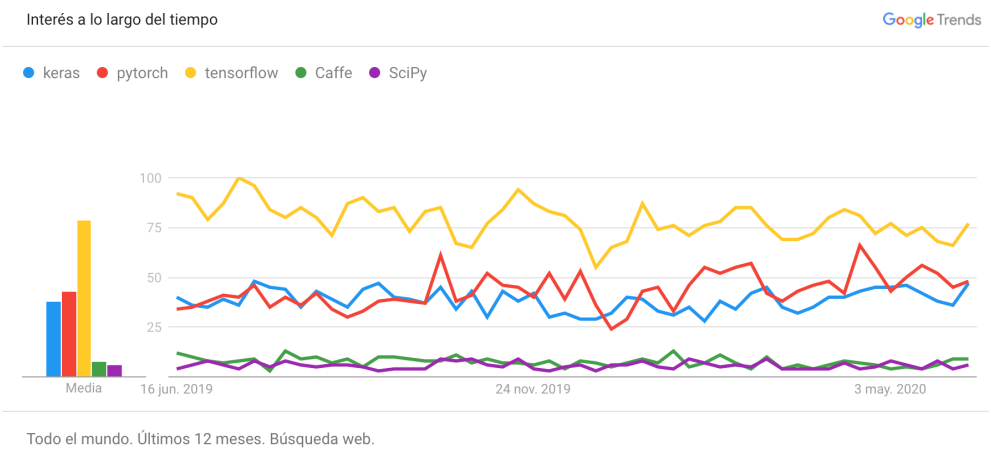


Figura 2.1. Interés en el último año de las principales librerías de inteligencia artificial

Una vez identificadas las tres librerías más populares (Keras, TensorFlow y PyTorch), se procederá a describir las características de cada una de ellas, así como sus ventajas y desventajas.

2.1. Keras.

Keras[1] es una librería de redes neuronales artificiales de código abierto escrita en Python, creada en el año 2015 por el ingeniero de Google François Chollet.

Esta librería dispone de una *frontend* para la modelización de sistemas neuronales y un *backend* computacional para el entrenamiento de éstos. La modelización es el punto fuerte de esta librería, ya que ofrece una API sencilla y modular para la definición de sistemas neuronales.

Dado que la parte computacional de la librería se encuentra escrita en Python, a nivel de rendimiento presentaba grandes desventajas respecto a otras librerías como: TensorFlow, PyTorch, Caffe o SciPy. Además del problema del rendimiento, tareas como la depuración de código resultan complicadas.

En el año 2017 el equipo de TensorFlow (Google) asumió el soporte de Keras y la integró al núcleo de TensorFlow como API *frontend* de alto nivel para la definición y el modelado de redes neuronales. A partir de este momento Keras pasó a ser una API de alto nivel dependiente de un *backend* computacional capaz de procesar el entrenamiento de los modelos definidos.

En el año 2019 Google anunció la nueva versión de TensorFlow 2.0 en la cual Keras pasó a ser oficialmente la API de alto nivel.

Cabe destacar que Keras puede ser utilizada como API de alto nivel para la definición de modelos junto a otros motores computacionales como pueden ser Microsoft Cognitive Toolkit o Theano.

Es interesante observar la evolución y supervivencia de esta librería gracias a su simplicidad a la hora de definir modelos y cómo se ha convertido en parte fundamental de otras librerías para la definición de modelos de alto nivel.

2.2. TensorFlow.

TensorFlow[2] es una librería de código abierto para el aprendizaje automático desarrollada por Google.

La fecha de lanzamiento de la primera versión estable es del 9 de noviembre del 2015 y la última versión estable 2.0 es del 1 de noviembre del 2019.

El nombre de la librería hace referencia a cómo ésta procesa la información, ya que un tensor es una matriz de números y el flujo de éstas son las operaciones que se realizan para transformarlas.

TensorFlow está programada en Python y C++, ofreciendo soporte a varios sistemas operativos como Linux, Mac OS, Windows, iOS y Android. A su vez integra APIs de alto nivel para diferentes lenguajes de programación como Python, Java, C++, Haskell, Go y Rust.

[1] https://keras.io
[2] https://www.tensorflow.org

Ofrece la posibilidad de ser ejecutada tanto en CPUs como GPUs (Unidades de procesamiento gráfico), siendo esta última opción posible gracias a herramientas de terceros como CUDA, desarrollada por NVIDIA[3].

Esta librería ha ido evolucionando con el tiempo hacia la simplicidad y la facilidad de uso, especialmente en el apartado del modelado de las redes. Como se comentó en el apartado 2.1, TensorFlow incluyó en la versión 2.0 de manera oficial la API de alto nivel de Keras con el objetivo de mejorar y ofrecer una versión más accesible para los desarrolladores.

TensorFlow está orientada a ambientes de producción, esto es, prima la eficiencia por encima de la sencillez, sobre todo en el apartado *backend* de la librería. El entrenamiento de los modelos se puede realizar con múltiples hilos, reduciendo los tiempos de ejecución, especialmente al ejecutare sobre GPUs.

Tradicionalmente, y hasta la versión 2.0 de TensorFlow publicada en 2019, el modo de ejecución de la librería se basaba en grafos. Esto está cambiando hacia un modo imperativo más familiar con la forma en la que se programa en Python denominado *Eager Execution*. A continuación, se describen las características de cada modo de ejecución.

- El modo de ejecución mediante grafos consiste en definir unas dependencias y relaciones entre los diferentes elementos que componen nuestro problema. Una vez definido el grafo éste se ejecuta.
    - Ventajas:
        - Permite independizar partes el grafo para poder realizar ejecuciones en paralelo sin riesgo de cometer errores.
        - Descarta operaciones innecesarias ahorrando así tiempo de ejecución.
    - Desventajas:
        - La depuración en tiempo de ejecución resulta muy complicada.
        - No es posible modificar los modelos en tiempo de ejecución.
        - El nivel de abstracción dificulta el aprendizaje de la librería.
- El modo *Eager Execution* pretende acercar la arquitectura del tratamiento de los datos a la forma en la que convencionalmente se trabaja en Python.
    - Ventajas:
        - Facilita la depuración del código en tiempo de ejecución.
        - Mejora la curva de aprendizaje.
        - El desarrollo es más intuitivo y con menos líneas de código.

[3] https://developer.nvidia.com/cuda-toolkit

- Desventajas:
    - Existe una penalización en los tiempos de ejecución.

Se puede observar cómo TensorFlow, con la entrega de la versión 2.0, está relajando cada vez más la eficiencia de cálculo de la librería a cambio de acercarla más al desarrollador, facilitando de este modo su uso mediante la incorporación de la API de Keras y el nuevo modo de ejecución *Eager Execution*.

TensorFlow no es una librería recomendada para iniciarse en el mundo de la inteligencia artificial si se pretende comprender con cierta profundidad cómo son los algoritmos de entrenamiento de una red neuronal, cómo se clasifica un dato nuevo, etc. El nivel de abstracción es tan grande que lo que ocurre cuando un modelo pasa a entrenarse es opaco al desarrollador. Esta es una de las características de la librería que con las nuevas versiones se está tratando de cambiar.

2.3. PyTorch.

PyTorch[4] es una librería de aprendizaje automático de código abierto desarrollada por el laboratorio de inteligencia artificial de Facebook.

El lanzamiento de la primera versión estable es de octubre de 2016 y la última versión 1.4.0 fue lanzada el 15 de enero de 2020.

Al igual que TensorFlow, PyTorch está programada en Python y C++, ofreciendo soporte a diferentes sistemas operativos como Linux, Mac OS, Windows, iOS y Android. En este caso, y a diferencia de TensorFlow, esta librería cuenta con tan solo tres APIs de alto nivel para los siguientes lenguajes: Python, Java y C++.

PyTorch puede ser ejecutado tanto en CPUs como GPUs. Para la modalidad de GPU es obligatorio hacer uso de la librería CUDA desarrollada por NVIDIA, del mismo modo que sucede con TensorFlow.

En cuanto a la arquitectura de la librería y la forma en la que procesa los datos, PyTorch presenta dos bloques claramente diferenciados: un *frontend* escrito en Python y un *backend* escrito en C++, tal y como comenta en un artículo uno de los principales desarrolladores de la librería [9].

Una de las principales ventajas de PyTorch es que trabaja con grafos dinámicos y, a diferencia de lo que hasta ahora venía haciendo TensorFlow, PyTorch permite la modificación de los modelos en tiempo de ejecución. Esta era una de las características principales que desmarcaba a esta librería respecto de TensorFlow.

Esta librería presenta una curva de aprendizaje menos inclinada que TensorFlow, es más intuitiva para el desarrollador y permite una depuración del código más sencilla.

[4] https://pytorch.org

2.4. Conclusiones.

Una vez realizada una revisión de las principales librerías de aprendizaje automático disponibles en el mercado, es notoria la tendencia de éstas a acercarse cada vez más al desarrollador, aunque esto suponga alguna penalización a nivel de ejecución.

El nivel de abstracción proporcionado por la parte *backend* de estas librerías está pensado para que el desarrollador únicamente tenga que encargarse del modelado, dejando en manos de la propia librería la parte del procesamiento. Esta característica común en la gran mayoría de librerías actuales puede no resultar una ventaja si se pretende conocer con cierta profundidad cómo funcionan estos sistemas.

Es en este contexto en el que se presenta y justifica la creación de la librería que tiene por objetivo el presente TFG: una librería que pueda servir como paso intermedio y enlace hacia librerías mucho más complejas y opacas.

De este modo, la implementación de la librería del TFG está completamente escrita en Python; esto mejora la legibilidad asumiendo una penalización en el rendimiento, pues el objetivo no es crear una librería que pueda ser usada en ambientes de producción o que trabaje con grandes conjuntos de datos.

Otra diferencia fundamental es la documentación de la librería, pues no solo trata de exponer las diferentes funciones que se pueden utilizar y cómo hacerlo. La documentación de esta librería comprende también su implementación. De este modo no solo se expone lo que esta librería es capaz de hacer sino cómo lo hace.

3. Redes neuronales artificiales.

Desde la primera piedra usada como herramienta hasta nuestros días, el ser humano siempre ha buscado nuevos y más sofisticados sistemas capaces de mejorar nuestras condiciones de vida, optimizando la forma en la que se resuelven problemas ya resueltos o desarrollando tecnología capaz de resolver problemas que hasta entonces no tenían solución.

Las redes neuronales artificiales son un claro ejemplo de este proceso evolutivo y de sofisticación propio del ser humano. Partiendo de una idea más o menos simple, en pocos años esta tecnología ha sufrido un crecimiento vertiginoso y nos ha ayudado a resolver problemas que hasta entonces resultaban imposibles.

Durante la Segunda Guerra Mundial se creó el primer computador electrónico, ENIAC. Era capaz de realizar operaciones matemáticas, lo que permitía la implementación de algoritmos capaces de calcular tablas de tiro de artillería, que hasta ese momento se realizaban de forma manual y requerían de una gran cantidad de tiempo.

Este tipo de máquinas permiten implementar algoritmos capaces de resolver problemas que antes eran costosos de resolver. Esta forma de resolver problemas desde un punto de vista algorítmico presenta limitaciones en aquellas tareas que requieren de un cierto aprendizaje, una experiencia, y que intentan emular el comportamiento humano, ya sea en reconocimiento de formas, toma de decisiones, etc.

Desde este punto de vista, la inteligencia artificial es un intento de emular aspectos de la inteligencia humana mediante máquinas. Las redes neuronales artificiales en concreto están inspiradas en el modelo biológico del cerebro humano, formados ambos por un conjunto de unidades de procesamiento más simples como es la neurona y donde cada uno de estos elementos más simples está conectado con otros, formando una red mucho más compleja y con una capacidad de cálculo mucho más potente y abstracta.

Las redes neuronales artificiales son sistemas capaces de aprender por sí mismos en lugar de ser programados de forma explícita, siendo esta una de sus grandes ventajas.

3.1. Historia.

En 1943 McCulloch y Pitts elaboran un modelo matemático de neurona [10]. Años después, en la década de los 50, algunos investigadores como Rochester, Holland, Haibt y Duda, realizaron una de las primeras simulaciones de una red neuronal en un computador.

En 1957 Rosenblatt introdujo el perceptrón como un sistema con capacidad de autoaprendizaje [1] y en 1962 publicó su libro *Principles of Neurodynamics* [11], en el que se presentó formalmente el concepto del Perceptrón como un modelo para la construcción de Redes Neuronales Artificiales.

En 1969 Minsky y Papert publicaron un libro llamado *Perceptron: An Introduction to Computational Geometry* [12], en el que demostraron que el perceptrón básico era incapaz de implementar funciones discriminantes no lineales y que esta limitación no se puede superar con el uso de múltiples capas, ya que toda red formada por perceptrones básicos es equivalente a una red compuesta por un único perceptrón. Para solucionar este problema sería necesario la introducción de funciones de activación no lineales. Además

de esta cuestión, resaltaron que los computadores en aquel momento no disponían de la suficiente potencia de procesamiento para poder resolver eficazmente los cálculos que una red neuronal artificial requiere.

A partir de este momento, y durante los siguientes 20 años, el desarrollo de las redes neuronales se vio mermado y fueron pocas personas las que se dedicaron a ello. Algunas de éstas trataron de desarrollar algoritmos de descenso por gradiente con funciones no lineales, pero el cálculo de derivadas parciales no resultaba convincente; seguía sin ser eficaz.

No es hasta el año 1986 cuando el campo de las redes neuronales resurgió por completo gracias al algoritmo *Backpropagation* diseñado por Rumelhart, Hinton y Williams [13], que permitió la simplificación del cálculo de derivadas parciales necesarias para el descenso por gradiente mediante el uso de ciertas funciones de activación no lineales.

Este algoritmo permitió añadir capas ocultas a las redes, lo que provoca sistemas capaces de formar regiones de decisión más complejas e individualizadas. En cierto modo, provoca que la red pueda especificarse y adquirir un comportamiento más jerárquico: cada parte de la red se especializa en subproblemas más concretos del problema principal.

En el año 2006 Hinton publicó en *Science* un artículo sobre una nueva tendencia denominada *Deep Learning,* así como nuevas técnicas que permitieron mejorar el algoritmo de *Backpropagation* [14].

3.2. Definiciones.

Se definirá primero el elemento más básico de cualquier red, el perceptrón o neurona artificial, el cual se organiza en capas constituyendo sistemas más complejos como las redes neuronales artificiales que se definen a partir de éste.

Posteriormente se introducirá el concepto de sistema neuronal artificial capaz de aprender y poder generar una salida para unos datos de entrada.

3.2.1. Perceptrón.

Podemos definir el perceptrón como una unidad básica de procesamiento capaz de transformar unos datos de entrada en un único dato de salida. Es decir, una función escalar $f: \mathbb{R}^n \to \mathbb{R}$.

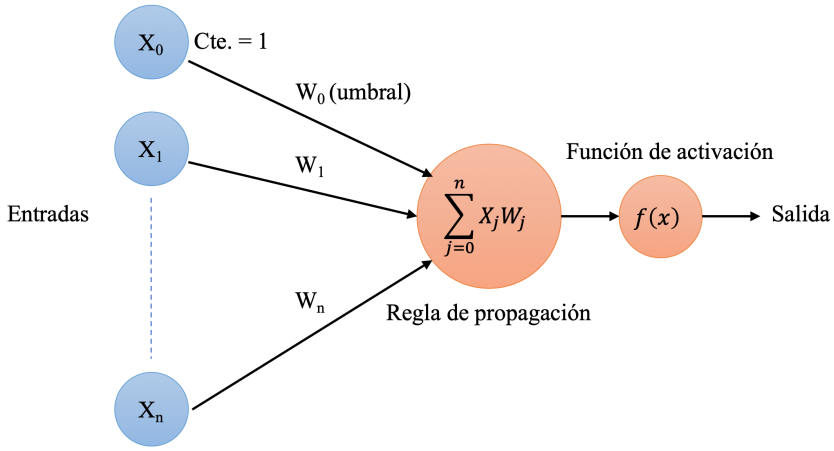


Figura 3.1. Esquema del perceptrón

Como se puede observar en la Figura 3.1, el perceptrón consta de 4 elementos principales:

- Entradas: datos de entrada que el elemento debe procesar y para los cuales producirá un valor de salida. Nótese que existe una entrada con valor constante igual a uno, denominada *bias*, pues es la responsable de que la frontera de decisión no corte siempre al origen de coordenadas. A partir de este momento definiremos el vector de entrada a la red como X, donde el elemento X_0 corresponde a la constante de valor uno.

- Pesos: ponderan el valor recibido en la entrada; son los valores responsables de la clasificación y se aprenden en cada iteración. Denominaremos a este vector como W, donde el elemento W_0 es el peso correspondiente al valor del *bias*.

- Regla de propagación $g: \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$: encargada de procesar todas las entradas y sus pesos asociados para convertirlos en un único valor, el cual será procesado en la siguiente fase. La regla de propagación más utilizada es el producto escalar del vector de entrada y el vector de peso $g(X, W) = X \cdot W$.

- Función de activación $f: \mathbb{R} \to \mathbb{R}$: es la encargada de procesar el dato entregado por la regla de propagación y define la salida del perceptrón. Existen multitud de funciones de activación; la elección de una u otra dependerá fundamentalmente del tipo de problema a resolver. Anteriormente se comentó la importancia de este tipo de funciones, ya que permiten la agregación de múltiples capas sin provocar que la red colapse en un modelo lineal. Funciones de activación no lineales permiten romper la linealidad de la red.

Una vez definidos los diferentes elementos que componen un perceptrón podemos formalizarlo matemáticamente de la siguiente forma. Siendo $g: \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$ y $f: \mathbb{R} \to \mathbb{R}$ las funciones de propagación y activación respectivamente, un perceptrón se define como: $f \circ g: \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R},\ f\big(g(X, W)\big) = x$.

3.2.2. Red neuronal artificial.

Existen numerosas definiciones de redes neuronales artificiales, algunas de ellas desde diversos puntos de vista: la computación, modelos matemáticos, modelos biológicos, etc.

Según el IBM Knowledge Center [15]: "*Una red neuronal es un modelo simplificado que emula el modo en que el cerebro humano procesa la información. Las unidades básicas de procesamiento son las neuronas, que generalmente se organizan en capas interconectadas*".

Las unidades de procesamiento se organizan en capas. Una red neuronal está formada por tres partes principales: una **capa de entrada** encargada de recibir los datos, cero o más **capas ocultas** y una **capa de salida** con una o varias unidades que representan las diferentes clases.

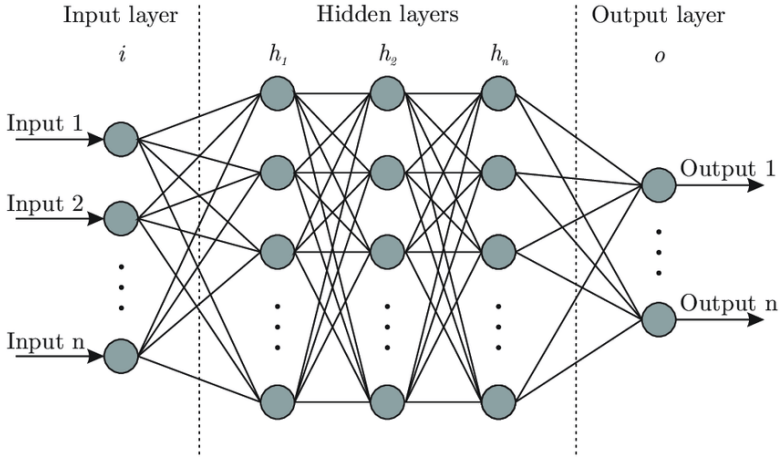


Figura 3.2. Arquitectura de una red neuronal artificial

Podemos definir una red neuronal artificial de una forma más formal como un grafo dirigido, semejante al de la Figura 3.2, donde:

- A cada nodo (neurona) i se le asocia una variable de estado X_i.
- A cada arista (i, j) entre dos nodos (neuronas) i y j se le asocia un peso W_{ij}.
- Para cada nodo i se define una función $f(x)$ que, dependiendo del vector de entrada y los pesos, genera el valor de salida de la neurona.

Desde este punto de vista, cada nodo representa una neurona del modelo biológico y cada arista una sinapsis. Además, el flujo de datos es direccional, es decir, los datos solo fluyen en un único sentido, desde la neurona presináptica a la postsináptica. Los nodos de entrada son aquellos sin aristas entrantes y, por extensión, los nodos de salida son aquellos que carecen de aristas de salida; al resto de nodos se les denomina nodos ocultos.

La arquitectura de una red se define a partir de la disposición de las neuronas y la relación entre ellas. Generalmente se disponen en capas y el conjunto de una o más capas constituye la red.

Podemos clasificar los diferentes tipos de redes atendiendo a dos parámetros:

- Arquitectura:
  - Redes monocapa.
  - Redes multicapa.
- Flujo de datos:
  - Unidireccionales: no existen aristas hacia atrás en el grafo que define la red, es decir, no hay bucles.
  - Recurrentes: existen aristas hacia atrás.

3.2.3. Sistema neuronal artificial.

Por último, y una definidos todos los elementos necesarios, se aborda el concepto de sistema neuronal, el cual está compuesto por una red neuronal con una determinada arquitectura y flujo de datos, un algoritmo de aprendizaje y unos datos de entrada. Este sistema es capaz de aprender y generar datos de salida tras un proceso de entrenamiento.

Dentro de los paradigmas de aprendizaje, supervisado o no supervisado, los algoritmos de aprendizaje de las redes neuronales artificiales forman parte del primero, y para su entrenamiento es necesario el par (X, C), donde X es la muestra y C la clase a la que pertenece. Por el contrario, en el paradigma no supervisado no es necesario conocer a qué clase pertenece cada muestra.

Un sistema neuronal se puede definir como una tupla (A, L, I, O), donde A es la arquitectura de la red, L su algoritmo de aprendizaje, I sus entradas y O sus salidas.

Los sistemas neuronales que se desarrollarán a continuación son todos **unidireccionales** y hacen uso de algoritmos de **aprendizaje supervisado**.

3.3. Características de las redes neuronales artificiales.

Por su arquitectura y modelado, las redes neuronales artificiales presentan algunas características semejantes a las del cerebro: son capaces de aprender de la experiencia, abstraer características esenciales a partir de los datos de entrada e incluso pueden llegar a competir con el ser humano, detectando patrones a veces indetectables para nosotros.

Destacaremos los cinco principios más relevantes de esta tecnología enumerados por Hilera y Martínez [16]:

- Aprendizaje adaptativo: las redes neuronales aprenden en base a un entrenamiento, no es necesario establecer un modelo *a priori*.
- Autoorganización: una red neuronal va creando su propia organización en base al entrenamiento y la experiencia adquirida con cada dato. Así pues, diferentes partes de la red se encargan de reconocer diferentes patrones.

- Tolerancia a fallos: este punto está muy ligado con el anterior; las redes neuronales son capaces de eliminar el ruido de los datos, enfatizando (ponderando) más las características esenciales de éstos. Incluso aunque se destruya una parte de la red, ésta podría seguir funcionando debido a su estructura distribuida y redundante.

- Operación en tiempo real: una vez entrenadas las redes neuronales realizan el reconocimiento y la clasificación de los datos en tiempo real.

- Fácil inserción en la tecnología existente: gracias a la naturaleza matricial de las operaciones necesarias para el entrenamiento de la red se pueden fabricar chips encargados de agilizar estas tareas, como es el caso las GPUs[5], capaces de realizar múltiples operaciones matriciales en paralelo.

3.4. Sistemas neuronales implementados.

Siguiendo el orden cronológico de aparición de los diferentes sistemas neuronales, el estudio de éstos se realizará de la forma que se describe a continuación.

En primer lugar, se definirá el perceptrón como unidad básica de procesamiento y su capacidad como clasificador binario. A continuación, se generalizará este algoritmo como clasificador multiclase, también denominado red neuronal de una sola capa, el cual es considerado como una de las implementaciones más sencillas de una red neuronal. Tras este desarrollo se expondrá el principal problema de este tipo de redes, incapaces de resolver regiones linealmente no separables, lo que nos llevará a las redes neuronales multicapa y el algoritmo *Backpropagation*, que resuelven este problema.

La definición de cada sistema estará dividida en dos bloques principales: arquitectura de la red y algoritmo de aprendizaje.

3.4.1. Clasificador binario.

Desde el punto de vista de un clasificador, el perceptrón puede ser definido como un algoritmo capaz de aprender una frontera de decisión lineal entre dos regiones, es decir, una función que mapea una entrada x en una salida $f(x)$ con dos únicos valores posibles, 0 o 1.

Esta es la implementación más sencilla y trivial de un clasificador. De hecho, ni si quiera podemos denominarlo red neuronal ya que solo participa una única neurona capaz de discriminar el espacio en dos regiones.

3.4.1.1. Arquitectura.

La arquitectura de este clasificador es la misma que la de la Figura 3.1, haciendo uso de la función de activación escalón (*binary step*) visible en la Figura 3.3.

[5] Aunque las GPUs no tenían inicialmente este propósito, su naturaleza ha permitido usarlas para el entrenamiento de redes neuronales de forma más eficaz.

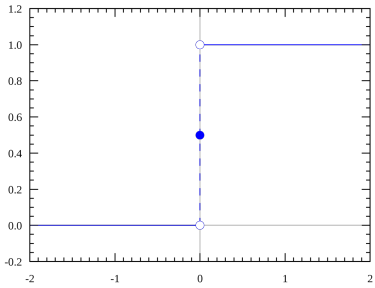


Figura 3.3. Función de activación escalón

Por lo tanto, la función que define a un clasificador binario es la siguiente:

$$f(x) = \begin{cases} 1, & X \cdot W > 0 \\ 0, & X \cdot W \leq 0 \end{cases} \qquad \forall\, X, W \in \mathbb{R}^n$$

3.4.1.2. Algoritmo de aprendizaje.

El algoritmo de aprendizaje se muestra en el Algoritmo 3.1, escrito en un pseudo lenguaje similar a Python.

En 1962 Novikoff demostró que este algoritmo converge después de un número finito de iteraciones si las muestras son linealmente separables [17].

Se define este algoritmo con la modalidad del factor de aprendizaje, un valor mayor o igual a cero que determina la velocidad de aprendizaje. Con un tamaño cercano a cero el algoritmo converge suavemente y con más iteraciones. A su vez se dispone de un número máximo de iteraciones, ya que si los datos no son linealmente separables se necesita de un punto de parada, pues el algoritmo nunca llegaría a converger.

Las limitaciones principales de este clasificador son dos:

- Solo es capaz de clasificar datos linealmente separables.
- Clasifica en dos clases (0, 1); esta limitación se resolverá en apartado 3.4.2 con un clasificador multiclase haciendo uso de varios perceptrones.

```
"""
Sean:
    - W: vector de pesos inicializados aleatoriamente.
    - (X0, C0), ..., (Xn-1, Cn-1), N muestras de aprendizaje,
    donde Xi es el vector de datos y Ci la clase a la cual pertenece.
    - rate: factor de aprendizaje.
    - max_iterations: máximo número de iteraciones.
"""

iteration = 0

while True:
    well_classified_samples = 0

    for sample in range(N):
        y = 1 if (X[sample] * W >= 0) else 0
        error = C[sample] - y

        if error != 0: # Muestra mal clasificada
            W = W + (rate * delta * X[sample])
        else:
            well_classified_samples += 1

    iteration += 1

    if iteration = max_iterations or well_classified_samples = N:
        break
```

Algoritmo 3.1. Algoritmo de aprendizaje del clasificador binario

3.4.2. Clasificador multiclase.

Una vez definido el clasificador binario podemos extenderlo para superar una de sus principales limitaciones, la cual es la incapacidad de discernir entre más de dos clases. De este modo, agregando varios clasificadores binarios podemos crear un clasificador capaz de clasificar los datos en C clases, siendo $C \in \mathbb{N}$.

3.4.2.1. Arquitectura.

Un clasificador multiclase en C clases puede ser definido por la agregación de C perceptrones capaces de clasificar cada muestra en una clase.

Cada perceptrón recibe la muestra, pondera sus valores con los pesos asociados a cada una de las entradas y devuelve un valor de salida. En última instancia la clasificación de la muestra se realiza comparando las salidas de cada perceptrón y seleccionando la clase

asociada al perceptrón que proporcione un valor de salida superior al resto. Esto se muestra en la Figura 3.4.

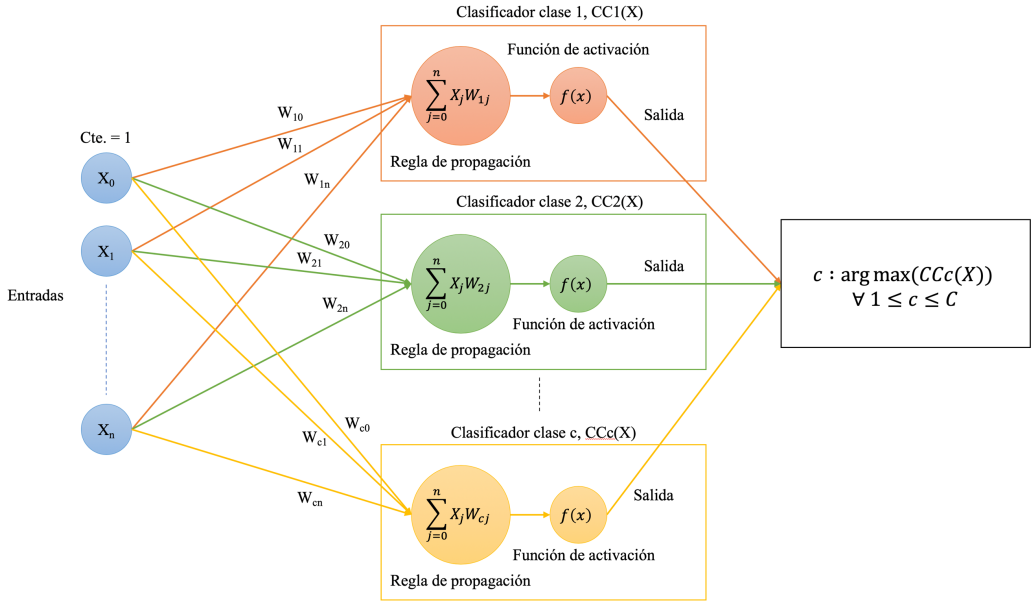


Figura 3.4. Arquitectura clasificador multiclase

La función de activación utilizada en este caso para cada perceptrón se trata de la función identidad $f: \mathbb{R} \to \mathbb{R},\ f(x) = x$ (Figura 3.5), de tal forma que el valor de salida de cada perceptrón es simplemente el obtenido tras la regla de propagación, es decir, el producto escalar entre las entradas y los pesos $f(X, W) = X \cdot W\ \forall\, X, W \in \mathbb{R}^n$.

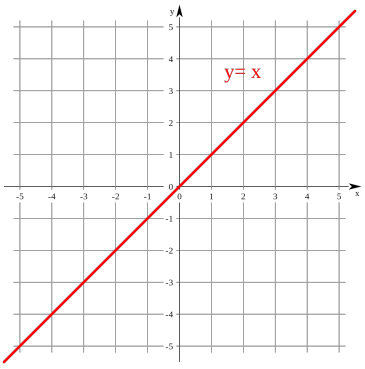


Figura 3.5. Función de activación identidad

Tras esta presentación del sistema se definirá a continuación formalmente el mismo. Siendo C el número de clases, consideraremos a cada clasificador $CCc(C)$ como una función $f: \mathbb{R}^n \to \mathbb{R}$, tal y como se definió en el apartado 3.2.1.

De este modo el sistema clasifica cada muestra mediante la función $f: \mathbb{R}^n \to \{1, \dots, C\} :\ f(x) = \arg\max(CCc(X))\ \forall\, 1 \leq c \leq C$, siendo n el tamaño de las muestras y C el número de clases.

Un clasificador multiclase divide el espacio de representación en C fronteras de decisión. Si el espacio de representación es $\mathbb{R}^3$ las fronteras serán planos, si el espacio de representación es $\mathbb{R}^2$ las fronteras serán rectas y en espacios de $\mathbb{R}$ serán puntos.

3.4.2.2. Algoritmo de aprendizaje.

El algoritmo de aprendizaje se encuentra definido en el Algoritmo 3.2.

```
"""
Sean:
    - W[c]: el vector de pesos inicializados aleatoriamente
    correspondientes al clasificador c.
    - (X0, C0), ..., (Xn-1, Cn-1), N muestras de aprendizaje,
    donde Xi es el vector de datos y Ci la clase a la cual pertenece.
    - rate: factor de aprendizaje.
    - max_iterations: máximo número de iteraciones.
"""

iteration = 0

while True:
    well_classified_samples = 0

    for sample in range(N):
        classifier = C[sample]
        y = X[sample] * W[sample]
        error = False

        for classifier2 in range(C) and classifier2 != classifier:
            y2 = X[sample] * W[classifier2]
            if y2 > y:
                W[classifier2] = W[classifier2] - rate * X[sample]
                error = True

        if error:
            W[classifier] = W[classifier] + rate * X[sample]
        else:
            well_classified_samples += 1

    iteration += 1

    if iteration = max_iterations or well_classified_samples = N:
        break;
```

Algoritmo 3.2. Algoritmo de aprendizaje del clasificador multiclase

Este algoritmo es una generalización del algoritmo del perceptrón definido anteriormente (Algoritmo 3.1), extendiéndolo para poder clasificar en más de dos clases. Nótese cómo en caso de error en la clasificación de una muestra se corrigen tanto los valores de las clases causantes del error, disminuyendo el valor de sus pesos, como el de la clase correcta, aumentándolos en este caso.

Si bien es cierto que la limitación de dividir el espacio de representación en más de dos clases ha sido posible superarla con este nuevo sistema, el problema de clasificar datos que no son linealmente separables sigue sin resolverse con este clasificador.

3.4.3. Redes neuronales multicapa.

El sistema de redes neuronales multicapa combina los elementos descritos y desarrollados en los sistemas previos, de los apartados 3.4.1 y 3.4.2. Este sistema hace uso de múltiples perceptrones organizados en diferentes capas.

Del mismo modo que el clasificador multiclase, esta arquitectura permite la clasificación de los datos en C clases, siendo $C \in \mathbb{N}$. La diferencia fundamental con el clasificador multiclase se encuentra en la capacidad de generar fronteras de decisión no lineales. Esto es posible gracias al uso de funciones de activación no lineales y la organización de los perceptrones en diferentes capas, lo que permite que esta arquitectura no colapse en un sistema de clasificación lineal.

3.4.3.1. Arquitectura.

La arquitectura de este tipo de sistemas se compone de una capa de entrada, una o varias capas ocultas y una capa de salida. El número de elementos que componen la capa de entrada es igual a la dimensión de las muestras a clasificar y el número de perceptrones que componen la capa de salida es igual al número de clases C.

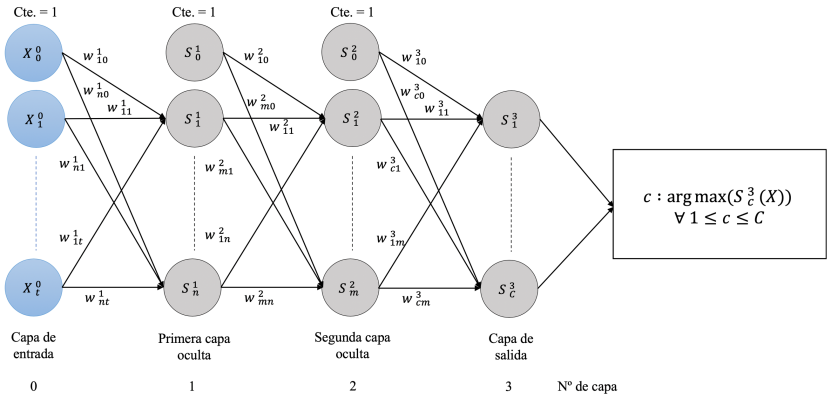


Figura 3.6. Arquitectura red neuronal multicapa de dos capas ocultas

Para simplificar, pero sin perdida de generalidad, la Figura 3.6 representa una red neuronal multicapa con dos capas ocultas. La arquitectura y las notaciones de las ecuaciones de propagación hacia delante y hacia atrás son extensibles a cualquier número de capas ocultas.

La función de activación definida en los perceptrones que componen este sistema debe ser derivable y se pueden implementar diferentes funciones de activación por cada capa oculta. En este caso se desarrollará e implementará la función de activación sigmoide, visible en la Figura 3.7, ampliamente utilizada en este tipo de sistemas de clasificación.

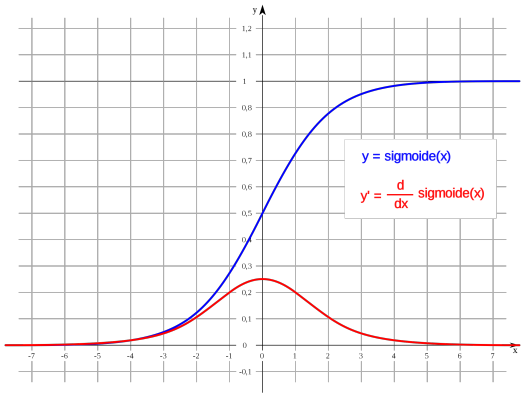


Figura 3.7. Función de activación sigmoide

De este modo la función es definida de la siguiente forma; $f: \mathbb{R} \to \mathbb{R},\ f(x) = \frac{1}{1+e^{-x}}$, cuya derivada es; $f': \mathbb{R} \to \mathbb{R},\ f'(x) = f(x)(1-f(x))$.

Atendiendo a la notación de los elementos que componen la Figura 3.6, encontramos las siguientes definiciones:

- Datos $X\,{}_{z}^{y}$: estos son valores inmutables en cada iteración, los datos entregados al sistema y que éste debe ser capaz de clasificar. El valor del superíndice y en $X\,{}_{z}^{y}$ hace referencia a la capa en la que se encuentra el elemento y el valor del subíndice z en $X\,{}_{z}^{y}$ hace referencia a la posición que ocupa dentro de la capa.

- Perceptrones $S\,{}_{y}^{x}$: estos elementos hacen referencia a la salida de cada uno de los perceptrones que componen la red. Al igual que sucede con los datos, el superíndice y subíndice hacen referencia a la capa y la posición en la que se encuentra el perceptrón respectivamente.

- Pesos $W\,{}_{yz}^{x}$: estos elementos hacen referencia a los pesos de la red, los cuales son modificados en cada iteración por el algoritmo de entrenamiento. El superíndice x en $W\,{}_{yz}^{x}$ hace referencia a la capa a la que acomete, el primer subíndice y en $W\,{}_{yz}^{x}$ representa la posición que ocupa el perceptrón al que acomete y por último el segundo subíndice z en $W\,{}_{yz}^{x}$ indica la posición del perceptrón de la capa inmediatamente anterior $(x-1)$ de la que parte el peso. La ausencia de un segundo superíndice se omite a favor de una notación más compacta, ya que la capa de la que parte el peso es siempre la anterior a la que acomete.

Nótese como todas las capas excepto la de salida disponen de un elemento $X\,{}_{0}^{y}$ en la capa de entrada y $S\,{}_{0}^{x}$ en las ocultas que hace referencia a la constante de valor igual a 1 correspondiente al *bias*.

La propagación hacia delante de las diferentes capas, suponiendo que la dimensión de entrada es *t*, el número de perceptrones de la primera capa oculta es *n*, el de la segunda capa oculta es *m*, el número de clases es *C* y una muestra X, se define a continuación:

- Primera capa oculta: $S_i^1(X) = f\left(\sum_{j=0}^{t} W_{ij}^1 X_j\right) \forall\, 1 \leq i \leq n$.
- Segunda capa oculta: $S_i^2(X) = f\left(\sum_{j=0}^{n} W_{ij}^2 S_j^1(X)\right) \forall\, 1 \leq i \leq m$.
- Capa de salida: $S_i^3(X) = f\left(\sum_{j=0}^{m} W_{ij}^3 S_j^2(X)\right) \forall\, 1 \leq i \leq C$.

Una vez definidos cada uno de los elementos del sistema se procede a la formalización de éste. La función que representa al sistema y se encarga de clasificar cada muestra es $f: \mathbb{R}^t \rightarrow \{1, \dots, C\} : f(x) = \arg\max(S_c^3(X)) \;\forall\, 1 \leq c \leq C$, siendo t la dimensión de las muestras y C el número de clases.

3.4.3.2. Algoritmo de aprendizaje.

El algoritmo de aprendizaje utilizado en este sistema es el desarrollado en el año 1986 por Rumelhart, Hinton y Williams, denominado *Backpropagation* [13]. Este algoritmo simplificó el cálculo de las derivadas parciales necesarias en el proceso de descenso por gradiente.

El problema que debe resolver el algoritmo es, dada una red neuronal multicapa y un conjunto de datos, obtener unos pesos que minimicen el error de clasificación del sistema. La solución al problema se basa en la idea del descenso por gradiente.

En primer lugar, se debe definir una función de error derivable capaz de medir el error de clasificación del sistema. Una vez definida esta función se obtienen las ecuaciones de propagación hacia atrás que serán utilizadas por el algoritmo para corregir los pesos en cada iteración.

El algoritmo *Backpropagation* se apoya en la definición de las derivadas parciales de la función de error respecto a los pesos del sistema. De este modo, y tras cada iteración, el algoritmo corrige los pesos de la red en el sentido opuesto a la pendiente de la derivada de la función de coste, obteniendo un error de clasificación menor tras cada iteración. Cuánto desciende la pendiente tras cada iteración viene determinado por un parámetro de la red denominado factor de aprendizaje ρ.

La función que se utilizará para determinar el error de clasificación del sistema es la del error cuadrático medio: $ECM = \frac{1}{2}\sum_{i=1}^{C}(t_i - S_i^3)^2$, siendo t_i el valor de salida esperado en el elemento i de la última capa y S_i^3 el valor obtenido por el perceptrón *i* de la última capa.

De este modo, las ecuaciones de propagación hacia atrás correspondientes a la función del error cuadrático medio para la corrección de los pesos dada una muestra X son las siguientes.

- Capa de salida ($1 \leq i \leq C, 0 \leq j \leq m$):
  - Error: $\delta_i^3(X) = \left(t_i - S_i^3(X)\right) f'(\sum_{j=0}^{m} W_{ij}^3 S_j^2(X))$.

- Corrección de los pesos: $W_{ij}^{3} = W_{ij}^{3} + \rho\delta_{i}^{3}(X)S_{j}^{2}(X)$.

- Segunda capa oculta ($1 \leq i \leq m, 0 \leq j \leq n$):
  - Error: $\delta_{i}^{2}(X) = \left(\sum_{r=1}^{c} \delta_{r}^{3}(X)W_{ri}^{3}\right)f'\left(\sum_{j=0}^{n} W_{ij}^{2}S_{j}^{1}(X)\right)$.
  - Corrección de los pesos: $W_{ij}^{2} = W_{ij}^{2} + \rho\delta_{i}^{2}(X)S_{j}^{1}(X)$.

- Primera capa oculta ($1 \leq i \leq n, 0 \leq j \leq t$):
  - Error: $\delta_{i}^{1}(X) = \left(\sum_{r=1}^{m} \delta_{r}^{2}(X)W_{ri}^{2}\right)f'\left(\sum_{j=0}^{t} W_{ij}^{1}X_{j}\right)$.
  - Corrección de los pesos: $W_{ij}^{1} = W_{ij}^{1} + \rho\delta_{i}^{1}(X)X_{j}$.

Una vez identificados los elementos del algoritmo podemos definir el mismo de forma general, tal como se ve en el Algoritmo 3.3.

```
Sean:
    - W: el vector de pesos inicializados aleatoriamente
    correspondientes al clasificador c.
    - (X0, C0), ..., (Xn-1, Cn-1), N muestras de aprendizaje,
    donde Xi es el vector de datos y Ci la clase a la cual pertenece.
    - rate: factor de aprendizaje.
    - max_epochs: número máximo de iteraciones en el que el algoritmo procesa todos los datos.

    1. epoch = 0
    2. Mientras epoch <= max_epochs
        3. Para 0 <= i < N
        4. Calcular la salida de la red para la muestra Xi (propagación hacia delante)
        5. Calcular el error cometido por la última capa.
        6. Propagar el error y ajustar pesos desde la última capa a la primera (propagación
hacia atrás)
        7. epoch++
```

Algoritmo 3.3. Algoritmo Backpropagation

4. Diseño de la librería.

Una vez identificados y formalizados los elementos que se desarrollarán en la librería, en este capítulo se expondrán las diferentes tecnologías utilizadas para su posterior implementación.

4.1. Tecnología utilizada.

A continuación, se describirán las diferentes herramientas utilizadas en la elaboración del proyecto.

4.1.1. Lenguaje de programación.

El lenguaje de programación utilizado para la implementación de la librería ha sido *Python,* concretamente la versión *Python 3.7.5.*

La elección de este lenguaje se debe fundamentalmente a tres motivos:

- Es el lenguaje de programación más utilizados actualmente en el campo de la inteligencia artificial según el *Community insights* publicado por *GitHub* en 2019 [6].
- Ha sido el segundo lenguaje de programación más utilizado durante 2019 según el *Developer Survey Results* publicado por *Stack Overflow* [18].
- Por los motivos expuestos anteriormente, entre otros, es uno de los lenguajes de programación que quería aprender en más profundidad. Consideré que este proyecto podría ser el punto de partida para aprender más sobre este lenguaje.

Además de los tres puntos mencionados anteriormente cabe destacar algunos de los aspectos que han convertido a *Python* en uno de los lenguajes de programación líderes:

- Es un lenguaje interpretado, dinámico y multiplataforma.
- Permite flexibilidad y rapidez en el desarrollo; con muy pocas líneas de código se pueden realizar implementaciones que en otros lenguajes serían más tediosas.
- Es multiparadigma: permite la programación orientada a objetos, funcional e imperativa.
- Dispone de gran cantidad de librerías que facilitan los desarrollos, sobre todo en el campo de la inteligencia artificial y el *Big Data*.

4.1.2. Sistema de control de versiones.

Como sistema de control de versiones se ha utilizado *Git*. En este sentido, y en comparación con la elección del lenguaje de programación, no había dudas, pues es prácticamente el estándar de los sistemas de control de versiones, gratuito y libre.

Se ha utilizado a su vez un repositorio remoto en https://github.com/OmarCaja/SNN en el que se mantiene una copia en línea del proyecto. Este repositorio se dispondrá de forma privada hasta que se finalice el TFG. Una vez finalizado se hará público para que pueda ser utilizado e incluso desarrollado por otras personas.

4.1.3. Sistema de gestión de paquetes.

Para la gestión de las librerías (dependencias) necesarias para desarrollar la aplicación se utiliza *Anaconda.*

Este programa permite la gestión de entornos de forma separada, gestionando así diferentes entornos cada uno con sus propias dependencias y todos ellos en una misma máquina.

En el desarrollo de la librería se utilizan dos paquetes que no forman parte de *core* de *Python*. Estos paquetes son *numpy* y *matplotlib*, utilizados para el manejo de matrices y la impresión de datos, respectivamente.

4.1.4. Jupyter Notebook.

En el contexto didáctico en el que se presenta esta librería es necesario, más allá del propio código, un documento que sirva de apoyo para dicho fin. Éste describe y justifica la implementación de cada uno de los módulos que componen la librería

Jupyter Notebook es una aplicación web que permite la documentación de código de una forma interactiva y de fácil acceso desde un navegador web.

Se compone de un documento a modo de página web dividido en celdas en las que se puede insertar y ejecutar código, texto en formato *Markdown*, fórmulas matemáticas y contenido multimedia. Existe la posibilidad de exportar el documento a múltiples formatos como *HTML*, *PDF*, *Python* o *Markdown*.

De esta forma, la guía sobre la implementación de cada uno de los módulos y la documentación web del repositorio han sido generados con esta herramienta.

Este *Notebook* se aloja en el repositorio en el que se encuentra la librería y es una parte más de la misma. La idea es que el desarrollo futuro de la misma sea completado en paralelo junto al del *Notebook*.

En la Figura 4.1 se muestra un fragmento del Jupyter Notebook, en el que se puede observar la presencia de texto descriptivo en formato *Markdown*, de código que puede ser modificado y ejecutado en el momento desde el propio navegador y, por último, de los archivos multimedia resultados de la ejecución de ese bloque de código.

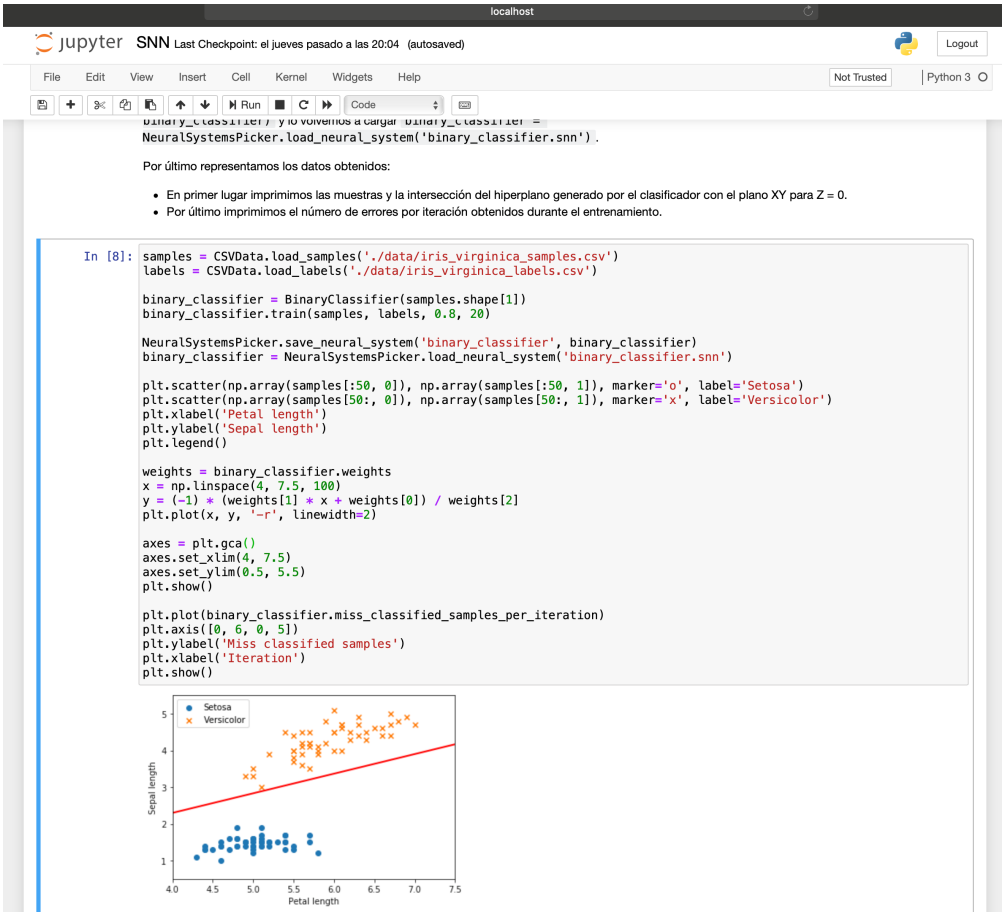


Figura 4.1. Documentación del Jupyter Notebook

4.1.5. Entorno de desarrollo.

La herramienta utilizada para la escritura del código de una forma cómoda y robusta ha sido *PyCharm.* Se trata de un IDE desarrollado por la compañía *JetBrains* que incluye múltiples funcionalidades para el desarrollo de aplicaciones en *Python*. Algunas de sus características son:

- Gestión de los entornos de desarrollo desde el propio IDE gracias a su integración con *Anaconda.*
- Módulo para el sistema de control de versiones integrado con *Git.*
- Herramientas para el depurado de la aplicación y la refactorización de código.
- Soporte para la edición y ejecución de Jupyter Notebooks.

Como se puede observar, esta herramienta se integra con las herramientas descritas en los puntos anteriores. De esta forma, desde una sola aplicación es posible controlar el resto de las herramientas utilizadas para desarrollar la librería.

4.1.6. Trello.

Trello[6] es una herramienta gratuita y ampliamente utilizada para la gestión y planificación de proyectos. Mediante el uso de tableros, tarjetas, etiquetas y listas, Trello permite la organización del proyecto de una forma intuitiva y flexible.

El proyecto se ha divido por tareas y fases, donde cada una de las tareas debe pasar por cada una de las fases antes de haberse completado. Las fases que componen el proyecto son las siguientes: por hacer, en proceso, en espera, terminado y dudas. La Figura 4.2 muestra un estado del tablero Trello del proyecto.

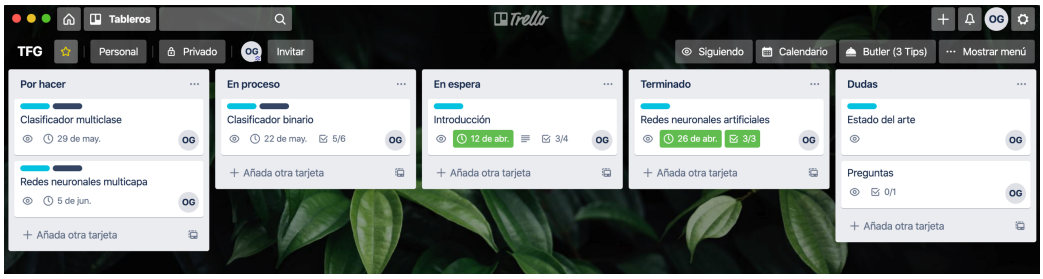


Figura 4.2. Tablero de Trello del proyecto

Cada tarea comienza en la fase “por hacer” y se le asigna un plazo de vencimiento, una lista de subtareas que componen la tarea principal y etiquetas sobre el tipo de tarea (redacción y/o código). Una vez la tarea se está realizando, se avanza a la fase “en proceso” hasta que finaliza y pasa a la fase final “terminado”. Hay dos fases extra; la primera es “en espera”, donde se acumulan las tareas que tienen dependencias con otras; por último, se tiene la fase “dudas”, donde se acumulan las tareas que deben ser consultadas con el tutor.

Los detalles de cada tarea se almacenan dentro de cada una de las tarjetas Trello. La Figura 4.3 muestra un ejemplo de una tarjeta Trello.

Este tipo de herramientas permite tener un control sobre los tiempos del proyecto y el número de tareas que componen el mismo, ofreciendo una perspectiva global del mismo.

[6] https://trello.com

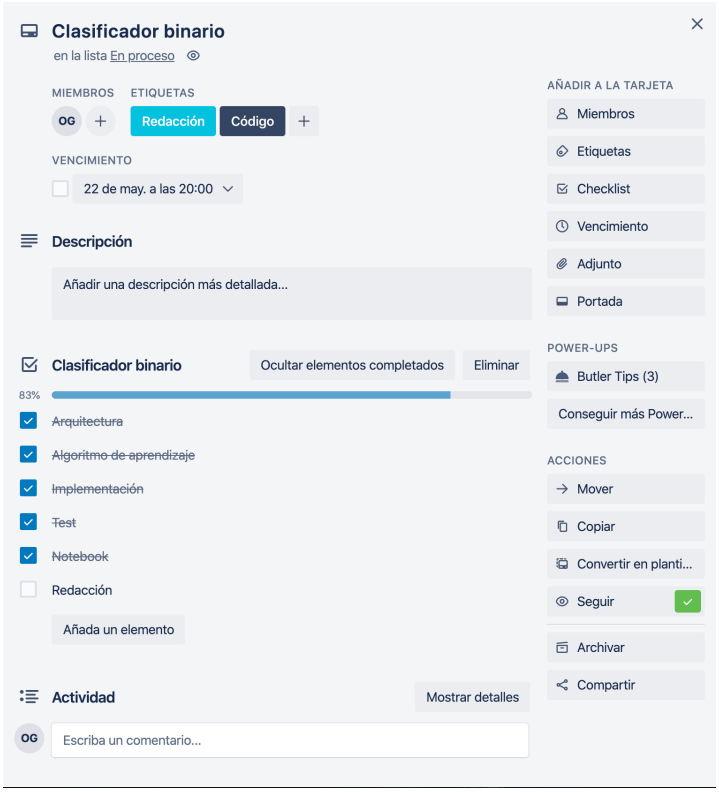


Figura 4.3. Detalle tarjeta de Trello

5. Desarrollo de la librería.

A continuación, se abordará con más detalle la implementación y el desarrollo de la aplicación. Del mismo modo que se formalizaron cada uno de los sistemas neuronales que van a ser desarrollados en el punto 3.4, en esta sección expondremos cada uno de los mismos desde el punto de vista de su implementación y las decisiones tomadas en cuanto a su diseño.

En primer lugar, se representará de forma conceptual cada uno de los elementos de la aplicación, sus características y la relación entre ellos. Tras esta contextualización se definirá en detalle cada uno de elementos, describiendo sus atributos, funciones y diseño.

A partir de este momento se hará referencia a cada una de las ubicaciones en las que se encuentra cada elemento desde el directorio raíz de la librería. Así pues, considerando como *./* el directorio de trabajo en el que se encuentra la librería, la raíz de la misma será: *./simple_neural_network*.

5.1. Arquitectura de la librería.

La librería se ha desarrollado íntegramente en el lenguaje de programación *Python*, haciendo uso del paradigma de programación orientado a objetos, lo que facilita la encapsulación y modelización de las diferentes clases.

La estructura de la librería se compone de los siguientes módulos:

- *activation_functions:* contiene la definición de las diferentes funciones de activación.
- *constants:* en este módulo se dispone de un archivo encargado de almacenar los valores de las diferentes constantes utilizadas en la librería.
- *loss_functions:* contiene la definición de las diferentes funciones de coste utilizadas por el algoritmo *Backpropagation*.
- *neural_systems:* este paquete contiene los diferentes sistemas de clasificación implementados: clasificador binario, clasificador multiclase y red neuronal artificial multicapa.
- *neuron:* este módulo contiene la implementación del perceptrón, elemento sobre el que pivotan todos los sistemas de clasificación implementados.
- *utilities:* este paquete contiene las diferentes utilidades necesarias para facilitar el uso de la librería: lectura de datos desde archivos *CSV*, normalización de datos, sistema de impresión de mensajes y guardado y carga de sistemas neuronales.

5.2. Utilidades.

Se cree conveniente que, además de los tres principales sistemas neuronales de los que se compone la librería, se deben desarrollar una serie de módulos para ciertas tareas inherentes a la misma. Se dispone por tanto de una serie de utilidades tales como lectura de

datos desde archivos CSV, normalización de datos, guardado en disco y carga de sistemas neuronales entrenados.

Estas utilidades pretenden facilitar el uso de la librería, permitiendo al usuario final centrarse únicamente en el entrenamiento de sistemas y la clasificación de datos de una forma más cómoda.

5.2.1. Lectura de datos desde archivos CSV.

Ubicación: *./simple_neural_network/utilities/data_loader/csv_data_loader.py.*

Este módulo está dedicado a la lectura y carga en memoria de datos desde un archivo .CSV. Estos archivos son utilizados frecuentemente por bancos de datos *online* para representar información.

Este archivo contiene una clase con tres métodos estáticos, dos públicos y uno privado:

- *__load_data(path_to_csv_file, delimiter, discard_first_row)*: método privado encargado de la lectura y carga de datos en crudo desde el archivo pasado como argumento. Este método es consumido por los dos métodos públicos, encargados únicamente de dar un formato a los datos.

- *load_samples(path_to_csv_file, delimiter, discard_first_row)*: método público que realiza una llamada a *__load_data(path_to_csv_file)* para cargar los datos de las muestras. Una vez obtenidos los datos los formatea para que puedan ser utilizados por los algoritmos de la librería. Así pues, todas la muestras cargadas utilizando este método serán del tipo *numpy.double*.

- *load_labels(path_to_csv_file, delimiter, discard_first_row)*: al igual que el método descrito anteriormente, éste también formatea los datos devueltos por *__load_data(path_to_csv_file)*, de modo que las etiquetas obtenidas a través de este método serán del tipo *numpy.intc*.

  Parámetros:

  - *path_to_csv_file*: ruta al archivo CSV que se desea leer.

  - *delimiter*: caracter delimitador para separar los valores de los datos en el fichero CSV, normalmente son: “,” o “;”.

  - *discard_first_row*: booleano que indica si deseamos o no descartar la primera línea del archivo CSV. Es común que la primera línea de este tipo de archivos se trate de una cabecera descriptiva de cada una de las columnas de éste.

Es importante realizar esta homogenización de los datos previa a su utilización por los diferentes sistemas neuronales. Se dispone por tanto esta utilidad como una facilidad para llevar a cabo esta tarea de carga y lectura que en otras ocasiones corre a cargo de los usuarios.

5.2.2. Normalización de datos.

Ubicación: *./simple_neural_network/utilities/normalization/normalization.py*.

Esta utilidad corresponde a un módulo encargado de la implementación de las diferentes funciones de normalizado de datos.

La función de normalizado implementada en el presente TFG es la de *unidad tipificada (Z-score)* $z = \frac{x-\mu}{\sigma}$, donde cada muestra es restada y dividida por la media y la desviación típica del conjunto de datos respetivamente.

Este archivo contiene una clase con un método estático público:

- *z_score(data)*: método público que recibe una lista de datos y devuelve una lista de datos del mismo tipo normalizados.

  Parámetros:

  - *data*: un *numpy.array* de tipo *numpy.double* a normalizar.

5.2.3. Sistema de impresión de mensajes.

Ubicación: *./simple_neural_network/utilities/logger/logger.py*.

En este paquete se implementarán las diferentes funciones encargadas de la impresión de los mensajes que se mostrarán por consola en el proceso de ejecución de la librería.

En este caso se ha desarrollado una función que da formato al mensaje que se imprime en el proceso de entrenamiento, siendo de vital importancia este tipo de mensajes en el entrenamiento de redes neuronales multicapa ya que el tiempo transcurrido necesario para el entrenamiento puede llegar a ser de horas. Por tanto, es interesante conocer cómo evoluciona nuestra red durante cada iteración, siendo posible que el algoritmo deje de converger en las primeras iteraciones y no sea necesario esperar la ejecución del resto de iteraciones.

Este archivo contiene una clase con un método estático público:

- *print_error_rate_message(epoch, misclassified_samples, samples, error_rate)*: método público que formatea e imprime los datos entregados como parámetros en cada iteración del proceso de entrenamiento. El mensaje mostrado es el siguiente: *Epoch 5: 3 misclassified samples out of 60 -> error rate = 0.05.* En el que se indica que en la iteración número 5 hubo 3 muestras mal clasificadas de 60 correspondiente a una tasa de error del 0.05.

  Parámetros:

  - *epoch*: iteración actual.
  - *misclassified_samples*: número de muestras mal clasificadas en la iteración actual.

- *samples*: número total de muestras de entrenamiento.
- *error_rate*: tasa de error cometida en la iteración actual.

5.2.4. Guardado y carga de sistemas neuronales.

Ubicación: *./simple_neural_network/utilities/neural_system_picker/neural_system_picker.py*.

Otra de las funcionalidades disponibles en la librería es la encargada de guardar y cargar sistemas neuronales en disco. De esta forma podemos entrenar una red y reutilizarla en nuevas ejecuciones.

La extensión de los objetos guardados en disco es: ".snn". Este archivo contiene una clase que dispone de dos métodos públicos y estáticos dedicados al guardado y carga de sistemas neuronales:

- *save_neural_system(file_name, neural_system)*: función encargada de guardar el sistema neuronal pasado como argumento en disco.

  Parámetros:

  - *file_name*: ruta al archivo donde se guardará el sistema neuronal.
  - *neural_system*: sistema neuronal a guardar.

- *load_neural_system(file_name)*: función encargada de cargar el archivo pasado como parámetro como un sistema neuronal en memoria.

  Parámetros:

  - *file_name*: ruta al archivo que se desea cargar como sistema neuronal.

5.3. Perceptrón.

Ubicación: *./ simple_neural_network/neuron/neuron.py*.

Se trata del elemento principal de la librería, y forma parte de cada uno de los sistemas neuronales implementados.

Los atributos y métodos que definen su clase hacen referencia directa a los elementos que componen al perceptrón visto en la Figura 3.1.

De este modo, tenemos una clase definida de la siguiente forma:

- Constructor:

  Neuron(number_of_inputs, activation_function): constructor de un objeto de tipo *Neuron*.

Parámetros:

- *number_of_inputs*: entero que indica el número de entradas del sistema, es decir, la dimensión de las muestras. Al número de entradas se le añade una extra de valor igual a 1 que hace referencia a la constante que multiplica el valor del *bias*.
- *activation_function*: tipo de función de activación de la neurona.

- Atributos:
  - *__weights:* representan los pesos de la neurona. Son los valores encargados de ponderar cada uno de los datos de entrada a la misma.
  - *__activation_function*: hace referencia al tipo de función de activación definida para la neurona.

- Funciones:
  - *weights()*: consultor del atributo *__weights*.
  - *weights(value)*: modificador del atributo *__weights*.

    Parámetros:

    - *value*: valor asignado al atributo *__weights*.
  - *activation_function()*: consultor del atributo *__activation_function*. En este caso no se define un modificador ya que la creación de este valor se realiza en el constructor y no debe cambiar durante la ejecución del programa.
  - *__calculate_propagation(input_values)*: función de propagación de los valores de entrada y los pesos de la neurona. En este caso se trata del producto escalar entre ambos vectores. Es un método privado ya que solo debe hacer uso de éste la función *calculate_output(input_values)*.

    Parámetros:

    - *input_values*: valores de entrada de la muestra sobre los que aplicar la regla de propagación.
  - *calculate_output(input_values)*: función de activación de la neurona, encargada de calcular la salida de ésta en función del tipo de función de activación definida en el constructor.

    Parámetros:

    - *input_values*: valores de entrada de la muestra sobre los que obtener el valor de salida de la neurona.

5.4. Clasificador binario.

Ubicación:
./simple_neural_network/neural_systems/binary_classifier/binary_classifier.py.

El clasificador binario se trata del primer sistema neuronal definido en la librería, ya que es el más sencillo de los tres implementados. Este sistema, tal y como se definió en el apartado 3.4.1, es capaz de definir una frontera de decisión lineal entre dos clases.

Los elementos que componen al clasificador binario son:

- Constructor:
  - *BinaryClassifier(number_of_inputs)*; constructor de un objeto de tipo *BinaryClassifier*.

    Parámetros:

    - *number_of_inputs*: entero que indica el número de entradas del sistema, es decir, la dimensión de las muestras.

- Atributos:
  - *__neuron:* atributo de tipo *Neuron*, es el encargado de realizar la clasificación y el objeto de entrenamiento.
  - *__learning_rate:* este atributo define la velocidad de aprendizaje del algoritmo; valores próximos a cero suponen convergencias más suaves, pero con más iteraciones.
  - *__max_epochs:* atributo que define la cantidad máxima de iteraciones que realizará el algoritmo; se alcanzará este valor cuando las muestras no sean linealmente separables.
  - *__misclassified_samples_per_epoch:* se trata de una lista de enteros donde cada entrada corresponde al número de muestras mal clasificadas en la iteración posición + 1 de la lista.

    De este modo *__misclassified_samples_per_epoch [x]* corresponde al número de muestras mal clasificadas en la iteración x + 1.

- Funciones:
  - *learning_rate()*: consultor del atributo *__learning_rate*; no se dispone de un modificador para este atributo ya que la definición del mismo se realiza en la llamada a la función *train(samples, labels, learning_rate, max_epochs)* y su valor no debe cambiar durante el proceso de entrenamiento.
  - *max_epochs()*: consultor del atributo *__max_epochs*; al igual que sucede con el atributo *__learning_rate,* éste no dispone de un modificador, ya que su definición se realiza en la llamada a la función *train(samples, labels, learning_rate, max_epochs).*

- *miss_classified_samples_per_epoch():* consultor del atributo *__miss_classified_samples_per_epoch;* no se dispone de un modificador ya que este atributo se genera en tiempo de ejecución.

- *epochs()*: esta función nos devuelve el número total de iteraciones realizadas durante el proceso de entrenamiento; su valor es la dimensión de la lista *__miss_classified_samples_per_epoch.*

- *weights()*: consultor del atributo *__neuron.weights()*, definido en la clase *Neuron* de la que hace uso el clasificador binario. Este atributo no dispone de modificador, ya que su escritura se debe realizar únicamente en el proceso de entrenamiento.

- *train(samples, labels, learning_rate, max_epochs)*: la llamada a esta función desencadena el entrenamiento del sistema; el algoritmo utilizado para el entrenamiento es el definido anteriormente en el Algoritmo 3.1.

  Parámetros:

  - *samples*: un *numpy.array* de tipo *numpy.double* para las muestras.
  - *labels*: un *numpy.array* de tipo *numpy.intc* para las etiquetas.
  - *learning_rate*: velocidad de aprendizaje del algoritmo; por defecto, su valor es 1.
  - *max_epochs*: número máximo de iteraciones en caso de que las muestras no sean linealmente separables; por defecto, su valor es 10.

- *classify(sample)*: función encargada de clasificar una muestra; devuelve 0 o 1.

  Parámetros:

  - *sample*: un *numpy.array* de tipo *numpy.double* para las muestras.

- *calculate_error_rate(samples, labels)*: función encargada de calcular la tasa de error cometido por el sistema al clasificar las muestras entregadas como parámetros con sus respectivas etiquetas.

  Parámetros:

  - *samples*: un *numpy.array* de tipo *numpy.double* para las muestras.
  - *labels*: un *numpy.array* de tipo *numpy.intc* para las etiquetas.

Las muestras y etiquetas se pueden extraer de archivos CSV con el uso de la funcionalidad descrita en el apartado 5.2.1.

Las dimensiones y formato de ambos datos serán NxM para las muestras y Nx1 para las etiquetas, donde N es el número de muestras y M es la dimensión de estas y, por tanto, *labels[n]* es la etiqueta correspondiente a la muestra *samples[n].*

5.5. Clasificador multiclase.

Ubicación:
./simple_neural_network/neural_systems/multiclass_classifier/multiclass_classifier.py.

Este sistema, tal y como se definió en el apartado 3.4.2, es capaz de clasificar una serie de muestras en N clases diferentes. Al igual que ocurría con el clasificador binario, la frontera de decisión entre dos clases sigue siendo lineal.

Los elementos que componen al clasificador multiclase son:

- Constructor:
  - *MulticlassClassifier(number_of_inputs, number_of_classes)*; constructor de un objeto de tipo *MulticlassClassifier*.

    Parámetros:
    - *number_of_inputs*: número de entradas del sistema, es decir, la dimensión de las muestras.
    - *number_of_classes*: número de clases del sistema.
- Atributos:
  - *__number_of_classes:* atributo de tipo entero que indica el número de clases del sistema.
  - *__neurons:* lista de tipo *Neuron*, cuya dimensión es igual al número de clases del sistema. Cada una de estas neuronas será la responsable de realizar la clasificación de cada muestra en cada una de las diferentes clases.
  - *__learning_rate:* este atributo define la velocidad de aprendizaje del algoritmo; valores próximos a cero suponen convergencias más suaves, pero con más iteraciones.
  - *__max_epochs:* atributo que define la cantidad máxima de iteraciones que realizará el algoritmo; se alcanzará este valor cuando las muestras no sean linealmente separables.
  - *__miss_classified_samples_per_epoch:* se trata de una lista de enteros donde cada entrada corresponde al número de muestras mal clasificadas en la iteración posición + 1 de la lista.

    De este modo *__miss_classified_samples_per_iteration[x]* corresponde al número de muestras mal clasificadas en la iteración x + 1.
- Funciones:
  - *number_of_classes():* consultor del atributo *__number_of_classes;* no se define un modificador ya que este atributo se define en el constructor y no debe cambiar su valor durante la ejecución.

- *learning_rate()*: consultor del atributo *__learning_rate*; no se dispone de un modificador para este atributo ya que la definición del mismo se realiza en la llamada a la función *train(samples, labels, learning_rate, max_epochs)* y su valor no debe cambiar durante el proceso de entrenamiento.

- *max_epochs()*: consultor del atributo *__max_epochs*; al igual que sucede con el atributo *__learning_rate,* éste no dispone de un modificador, ya que su definición se realiza en la llamada a la función *train(samples, labels, learning_rate, max_epochs).*

- *miss_classified_samples_per_epoch():* consultor del atributo *__miss_classified_samples_per_epoch;* no se dispone de un modificador ya que este atributo se genera en tiempo de ejecución.

- *epochs()*: esta función nos devuelve el número total de iteraciones realizadas durante el proceso de entrenamiento; su valor es la dimensión de la lista *__miss_classified_samples_per_epoch.*

- *weights()*: devuelve una lista con los pesos de cada una de las neuronas del atributo *__neurons.*

- *train(samples, labels, learning_rate, max_epochs)*: la llamada a esta función desencadena el entrenamiento del sistema; el algoritmo utilizado para el entrenamiento es el definido anteriormente en el Algoritmo 3.2.

  Parámetros:

  - *samples*: un *numpy.array* de tipo *numpy.double* para las muestras.
  - *labels*: un *numpy.array* de tipo *numpy.intc* para las etiquetas.
  - *learning_rate*: velocidad de aprendizaje del algoritmo; por defecto, su valor es 1.
  - *max_epochs*: número máximo de iteraciones en caso de que las muestras no sean linealmente separables; por defecto, su valor es 10.

- *classify(sample)*: función encargada de clasificar una muestra; devuelve un valor c que pertenece al conjunto $\{1, \dots, C\}$, siendo C el número de clases.

  Parámetros:

  - *sample*: un *numpy.array* de tipo *numpy.double* para las muestras.

- *calculate_error_rate(samples, labels)*: función encargada de calcular la tasa de error cometido por el sistema al clasificar las muestras entregadas como parámetros con sus respectivas etiquetas.

  Parámetros:

  - *samples*: un *numpy.array* de tipo *numpy.double* para las muestras.
  - *labels*: un *numpy.array* de tipo *numpy.intc* para las etiquetas.

Las muestras y etiquetas se pueden extraer de archivos CSV con el uso de la funcionalidad descrita en el apartado 5.2.1.

Las dimensiones y formato de ambos datos serán NxM para las muestras y Nx1 para las etiquetas, donde N es el número de muestras y M es la dimensión de estas y, por tanto, *labels[n]* es la etiqueta correspondiente a la muestra *samples[n]*.

5.6. Redes neuronales artificiales multicapa.

Ubicación:
./simple_neural_network/neural_systems/multilayer_neural_network/multilayer_neural_network.py.

Este sistema, tal y como se definió en el apartado 3.4.3, es capaz de clasificar una serie de muestras en N clases diferentes. A diferencia de lo que ocurría con el clasificador multiclase, la frontera de decisión entre dos clases ya no es lineal.

Los elementos que componen al clasificador multiclase son:

- Constructor:
  - *MultilayerNeuralNetwork(layers_definition)*; constructor de un objeto de tipo *MultilayerNeuralNetwork*.

    Parámetros:

    - *layers_definition*: se trata de una lista con la definición de cada una de las capas que componen el sistema, siendo cada elemento una lista con dos elementos, el primero un entero que hace referencia al número de neuronas de la capa y el segundo al tipo de función de activación utilizada en las neuronas de esa capa.

      Ejemplo:

      [[784], [20, ActivationFunctionsEnum.SIGMOID_FUNCTION], [20, ActivationFunctionsEnum.SIGMOID_FUNCTION]]

      De esta forma, el primer elemento de la lista representa la capa 0 del sistema y, por tanto, la dimensión de las muestras; en este caso no es necesario definir una función de activación.

- Atributos:
  - *__number_of_classes:* atributo de tipo entero que indica el número de clases del sistema.
  - *__layers:* lista de listas de tipo *Neuron*, se construye a partir del atributo *layers_definition* recibido a través del constructor.
  - *__learning_rate:* este atributo define la velocidad de aprendizaje del algoritmo; valores próximos a cero suponen convergencias más suaves, pero con más iteraciones.

- *__max_epochs:* atributo que define la cantidad máxima de iteraciones que realizará el algoritmo; se alcanzará este valor cuando las muestras no sean linealmente separables.

- *__miss_classified_samples_per_epoch:* se trata de una lista de enteros donde cada entrada corresponde al número de muestras mal clasificadas en la iteración posición + 1 de la lista.

  De este modo *__miss_classified_samples_per_iteration[x]* corresponde al número de muestras mal clasificadas en la iteración x + 1.

- Funciones:

  - *number_of_classes():* consultor del atributo *__number_of_classes;* no se define un modificador ya que este atributo se define en el constructor y no debe cambiar su valor durante la ejecución.

  - *learning_rate()*: consultor del atributo *__learning_rate*; no se dispone de un modificador para este atributo ya que la definición del mismo se realiza en la llamada a la función *train(samples, labels, learning_rate, max_epochs)* y su valor no debe cambiar durante el proceso de entrenamiento.

  - *max_epochs()*: consultor del atributo *__max_epochs*; al igual que sucede con el atributo *__learning_rate,* éste no dispone de un modificador, ya que su definición se realiza en la llamada a la función *train(samples, labels, learning_rate, max_epochs).*

  - *miss_classified_samples_per_epoch():* consultor del atributo *__miss_classified_samples_per_epoch;* no se dispone de un modificador ya que este atributo se genera en tiempo de ejecución.

  - *weights()*: devuelve una lista con los pesos de cada una de las neuronas del atributo *__layers,* siendo *weights[l][n][w]* el peso *w* correspondiente a la neurona *n* de la capa *l.*

  - *__forward_propagation(sample)*: función privada encargada de calcular la propagación hacia delante de cada una de las capas de la red; devuelve una lista de valores correspondientes a las salidas obtenidas por las neuronas de cada capa, siendo *forward_propagation[l][n]* el valor de salida obtenido por la neurona *n* en la capa *l*.

    Parámetros:

    - *sample*: un *numpy.array* de tipo *numpy.double* correspondiente a una muestra.

  - *__generate_expected_output(label)*: función privada encargada de generar y devolver una lista de dimensión igual al número de clases con valor igual a uno en la posición de la lista correspondiente al valor *label* y valor 0 en el resto de las posiciones.

    Parámetros:

    - *label*: un *numpy.intc* que representa la etiqueta de una muestra.

- *__mse_calculate_errors_per_layer(outputs_per_layer, expected_output)*: función privada encargada de calcular los errores cometidos por cada neurona de cada una de las capas del sistema; devuelve una lista donde *mse_calculate_errors_per_layer[l][n]* corresponde al error de salida obtenido por la neurona *n* en la capa *l*.

  La llamada a esta función desencadena la llamada de las funciones *__mse_calculate_output_layer_errors(outputs_per_layer, expected_output)* y *__mse_calculate_hidden_layer_errors(layer, outputs_per_layer, errors_per_layer)* para calcular los errores generados en la última capa y las capas ocultas respectivamente.

  Esto es así ya que la ecuación para calcular el error de cada capa es diferente para la última capa y las ocultas.

  Parámetros:

  - *outputs_per_layer*: lista generada en la llamada a la función *__forward_propagation(sample)*.
  - *expected_output*: lista generada en la llamada a la función *__generate_expected_output(label)*.

- *__mse_calculate_output_layer_errors(outputs_per_layer, expected_output)*: función privada encargada de calcular los errores cometidos por cada neurona de la capa de salida; devuelve una lista donde *mse_calculate_output_layer_errors[n]* corresponde al error de salida obtenido por la neurona *n* de la capa de salida.

  Parámetros:

  - *outputs_per_layer*: lista generada en la llamada a la función *__forward_propagation(sample)*.
  - *expected_output*: lista generada en la llamada a la función *__generate_expected_output(label)*.

- *__mse_calculate_hidden_layer_errors(layer, outputs_per_layer, errors_per_layer)*: función privada encargada de calcular los errores cometidos por cada neurona de la capa oculta *layer*; devuelve una lista donde *mse_calculate_hidden_layer_errors[n]* corresponde al error de salida obtenido por la neurona *n* de la capa de oculta *layer*.

  Parámetros:

  - *layer*: valor entero que hace referencia a la capa oculta.
  - *outputs_per_layer*: lista generada en la llamada a la función *__forward_propagation(sample)*.
  - *errors_per_layer*: lista que contiene los errores cometidos por cada neurona de cada una de las capas del sistema posteriores a la capa

layer, pues para calcular el error cometido en la capa *n* se deben conocer los errores cometidos en la capa *n+1*.

- *__mse_correct_weights(outputs_per_layer, errors_per_layer)*: función privada encargada de corregir los pesos de cada una de las neuronas que conforman la red.

  Parámetros:

  - *outputs_per_layer*: lista generada en la llamada a la función *__forward_propagation(sample)*.
  - *errors_per_layer*: lista generada en la llamada a la función *__mse_calculate_errors_per_layer(outputs_per_layer, expected_output)*.

- *__mse_back_propagation(outputs_per_layer, expected_output)*: función privada encargada del entrenamiento del sistema para el tipo de función de coste *error cuadrático medio*. Su llamada desencadena la ejecución de las funciones *__mse_calculate_errors_per_layer(outputs_per_layer, expected_output)* y *__mse_correct_weights(outputs_per_layer, errors_per_layer)*.

  Parámetros:

  - *outputs_per_layer*: lista generada en la llamada a la función *__forward_propagation(sample)*.
  - *expected_output*: lista generada en la llamada a la función *__generate_expected_output(label)*.

- *train(samples, labels, loss_function, learning_rate, max_epochs)*: la llamada a esta función desencadena el entrenamiento del sistema; el algoritmo utilizado para el entrenamiento es el definido anteriormente en el Algoritmo 3.3.

  Parámetros:

  - *samples*: un *numpy.array* de tipo *numpy.double* para las muestras.
  - *labels*: un *numpy.array* de tipo *numpy.intc* para las etiquetas.
  - *loss_function*: función de coste del algoritmo, por defecto el *error cuadrático medio*.
  - *learning_rate*: velocidad de aprendizaje del algoritmo; por defecto, su valor es 0,1.
  - *max_epochs*: número máximo de iteraciones; por defecto, su valor es 20.

- *classify(sample)*: función encargada de clasificar una muestra; devuelve un valor *c* que pertenece al conjunto $\{1, \dots, C\}$, siendo *C* el número de clases.

Parámetros:

- *sample*: un *numpy.array* de tipo *numpy.double* para las muestras.

- *calculate_error_rate(samples, labels)*: función encargada de calcular la tasa de error cometido por el sistema al clasificar las muestras entregadas como parámetros con sus respectivas etiquetas.

  Parámetros:

  - *samples*: un *numpy.array* de tipo *numpy.double* para las muestras.
  - *labels*: un *numpy.array* de tipo *numpy.intc* para las etiquetas.

Las muestras y etiquetas se pueden extraer de archivos CSV con el uso de la funcionalidad descrita en el apartado 5.2.1.

Las dimensiones y formato de ambos datos serán NxM para las muestras y Nx1 para las etiquetas, donde N es el número de muestras y M es la dimensión de estas y, por tanto, *labels[n]* es la etiqueta correspondiente a la muestra *samples[n]*.

6. Creación del Jupyter Notebook.

Como se comentó en los apartados 1 y 1.1, el principal objetivo del proyecto es construir una librería con fines didácticos sobre redes neuronales artificiales. Esta es sin duda la característica más destacable de esta librería respecto a las principales librerías actuales más utilizadas.

La forma en la que se ha trabajado para generar esta documentación y hacerla más accesible es mediante la creación de dos *Jupyter Notebooks,* uno en castellano y otro en inglés. Como se expuso en el apartado 4.1.4, esta herramienta reporta múltiples ventajas a la hora de generar una documentación para aplicaciones escritas en *Python*.

Una vez finalizada la documentación de los *Jupyter Notebooks*, se han generado dos archivos en formato *Markdown* y dos documentos en formato *PDF*, ambos en castellano e inglés. Estos documentos permiten disponer de una documentación en un formato estático y de fácil lectura en el caso de los archivos *PDF* y una documentación web en la página principal del repositorio a partir de los documentos *Markdown*.

Al igual que el código que compone la librería, toda la documentación generada se encuentra disponible en repositorio del proyecto.

El proceso de documentación de la librería se ha realizado haciendo un recorrido por cada uno de los diferentes paquetes y archivos que la componen, explicando en detalle tanto su implementación como su relación con el resto de los elementos de la librería y realizando pruebas de ejecución al final de cada uno de los sistemas desarrollados.

6.1. Documentación web del repositorio.

Los archivos en formato *Markdown* son ampliamente utilizados para generar la documentación web de los repositorios. Estos archivos pueden ser renderizados por cualquier navegador y son una buena forma de exponer las características del proyecto en la página principal del mismo.

En las Figura 6.1 y Figura 6.2 se puede observar un extracto del aspecto de la web en la que se aloja el repositorio del proyecto. De este modo, toda la documentación del proyecto, aparte de poder consultarse en el propio *Notebook,* se encuentra accesible de una forma más directa a través de la web del repositorio.

Este archivo es generado directamente por el *Jupyter Notebook.* Esta herramienta ofrece la posibilidad de generar este tipo de archivos directamente a partir del código implementado y documentado en el propio *Notebook.* Esto permite que la documentación web del proyecto se mantenga siempre actualizada al mismo tiempo que se desarrolla y documenta el código en el *Jupyter Notebook;* de este modo, el desarrollador solo debe preocuparse por desarrollar y documentar las nuevas implementaciones en un único archivo.

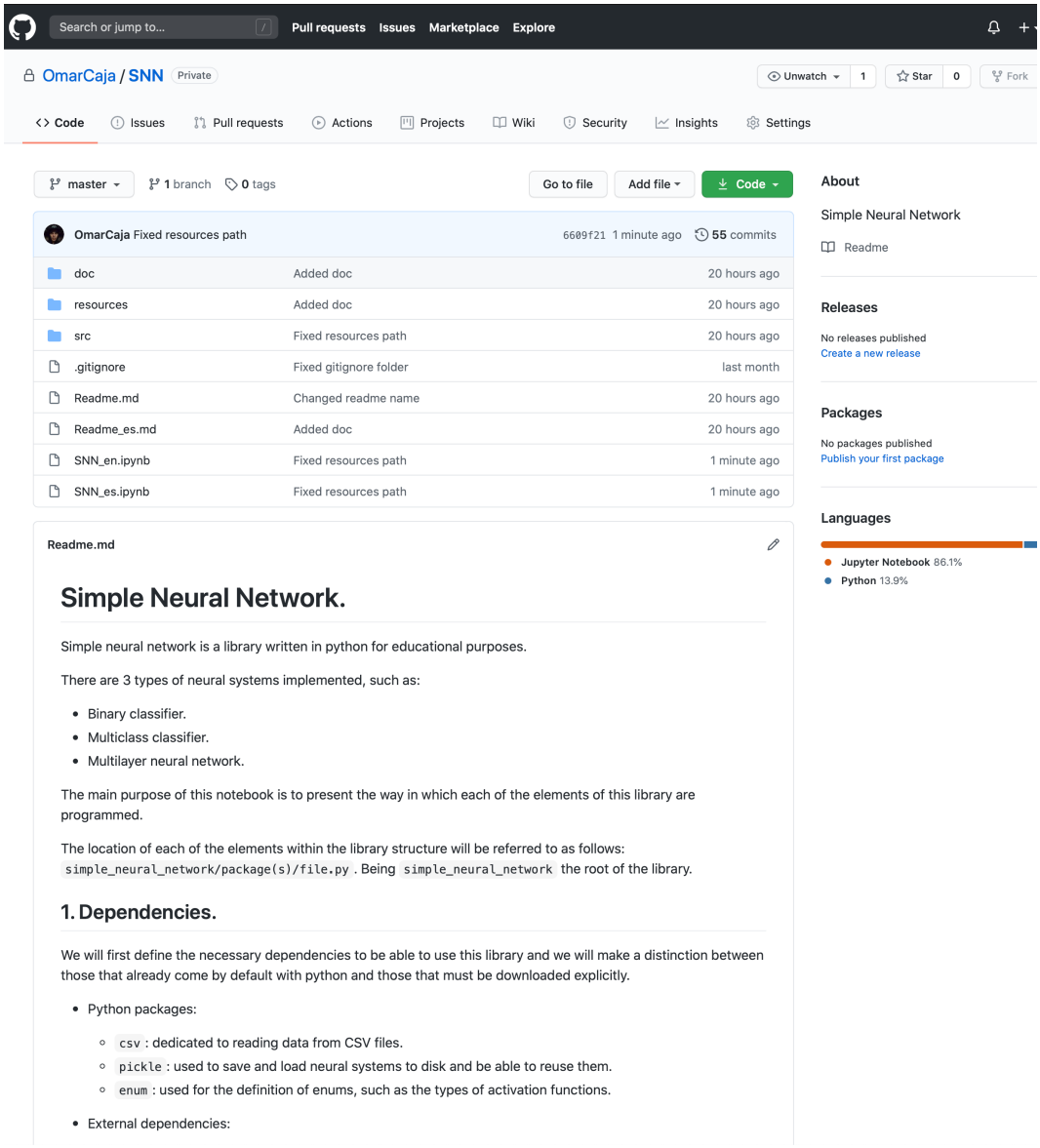


Figura 6.1. Repositorio de la librería

5. Loss function.

This module defines the different cost functions that will be used in the *Backpropagation* algorithm.

5.1 Enums.

Location: `simple_neural_network/loss_functions/loss_functions_enum.py`

This file lists the different types of cost functions. This allows us a more comfortable way to reference them.

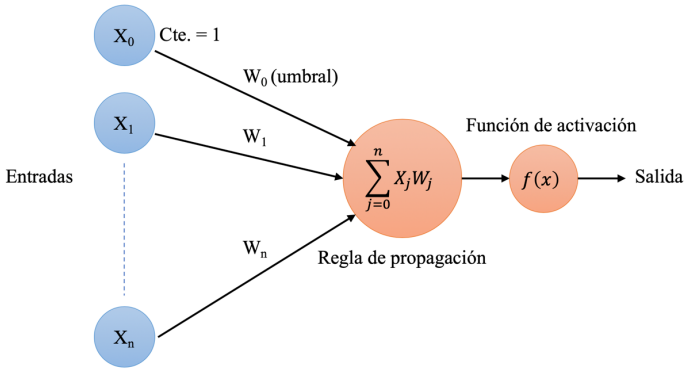
In that case the `MSE_FUNCTION` corresponds to *mean square error* function.

```
class LossFunctionsEnum(Enum):
    MSE_FUNCTION = 1
```

6. Perceptron.

Location: `simple_neural_network/neuron/neuron.py`

The `Neuron` class is the main element of this library, since it will be part of all the neural systems that will be defined below.



Elements that make up the neuron class:

- Constructor:
  - `Neuron(number_of_inputs, activation_function)` : returns a `Neuron` object.

    Parameters:

    - `number_of_inputs` : integer that indicates the number of inputs of the neuron (dimension of the samples). An extra value equal to 1 is added to the number of entries, which refers to the constant that multiplies the bias of the weights.
    - `activation_function` : activation function type.
- Attributes:
  - `__weights` : represent neuron's weights.
  - `__activation_function` : refers to the type of activation function defined for the neuron. Being a value of the enum `ActivationFunctionsEnum` defined above.

Figura 6.2. Detalle de la documentación del repositorio

6.2. Documentación del Jupyter Notebook.

Otra forma de presentar la documentación, desde el punto de vista de los desarrolladores y con un enfoque más interactivo, es la contenida en los propios *Jupyter Notebooks*. Estas herramientas nos permiten la visualización, modificación y ejecución de código al instante desde un navegador.

Esta documentación dinámica es fundamental para el propósito del proyecto, permitiendo que éste no sea una librería más con la que el usuario interactúa obteniendo unas salidas. Esta documentación hace transparente y didáctico el contenido de la librería, explicando el modo en el que se ha implementado y mostrando la relación entre sus diferentes elementos.

De esta forma, se ha documentado e implementado cada uno de los diferentes archivos que componen la librería. En general, la documentación relativa a cada archivo consta de las siguientes partes:

- Ubicación del archivo dentro de la librería del que se está realizando la documentación.
- Descripción y funcionamiento del elemento de la librería acompañado de las imágenes y ecuaciones matemáticas necesarias para su comprensión e implementación.
- Descripción de los diferentes elementos que componen el archivo, como son atributos, funciones y parámetros de cada función.
- Código que implementa dicho archivo.
- Ejemplo de uso de éste.

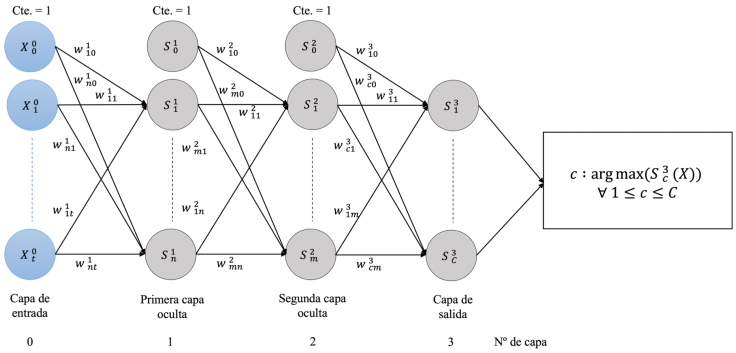
Las Figuras Figura 6.3, Figura 6.4, Figura 6.5 y Figura 6.6 muestran un extracto de la documentación que se ha redactado para la definición, implementación y ejemplo de uso de uno de los sistemas que componen la librería, una red neuronal artificial multicapa. Se ha elegido este sistema por ser el más completo y representativo de todos los que componen la librería.

En la Figura 6.3 se muestra la ubicación del archivo y definición del sistema que implementa haciendo uso de las imágenes y ecuaciones matemáticas.

7.3. Red neuronal artificial multicapa.

Ubicación: `simple_neural_network/neural_systems/multilayer_neural_network/multilayer_neural_network.py`

Este sistema es capaz de clasificar una serie de muestras en N clases diferentes, haciendo uso de N neuronas para ello. A diferencia de lo que ocurría con el clasificador multiclase, la frontera de decisión entre dos clases ya no es lineal.



Para la función del *error cuadrático medio* (`LossFunctionsEnum.MSE_FUNCTION`) las ecuaciones de propagación hacia delante, de cálculo del error y corrección de los pesos son las siguientes.

Para simplificar, pero sin perdida de generalidad, la figura representa una red neuronal multicapa con dos capas ocultas. La arquitectura y las notaciones de las ecuaciones de propagación hacia delante y hacia atrás son extensibles a cualquier número de capas ocultas.

- La propagación hacia delante de las diferentes capas, suponiendo que la dimensión de entrada es t, el número de perceptrones de la primera capa oculta es n, el de la segunda capa oculta es m, el número de clases es C y una muestra X se define a continuación:
  - Primera capa oculta: $S_i^1(X) = f(\sum_{j=0}^{t} W_{ij}^1 X_j \quad \forall 1 \leq i \leq n)$
  - Segunda capa oculta: $S_i^2(X) = f(\sum_{j=0}^{n} W_{ij}^2 S_j^1 \quad \forall 1 \leq i \leq m)$
  - Capa de salida: $S_i^3(X) = f(\sum_{j=0}^{m} W_{ij}^3 S_j^2 \quad \forall 1 \leq i \leq C)$

Figura 6.3. Documentación Jupyter Notebook, arquitectura de la red.

En la Figura 6.4 se puede observar un extracto de la descripción de algunos elementos que componen la red neuronal multicapa, tales como, el constructor y los parámetros que recibe o algunos de los atributos que conforman la clase.

Elementos que componen a la red neuronal artificial multicapa:

- Constructor:
  - `MultilayerNeuralNetwork(layers_definition)` : constructor de un objeto de tipo `MultilayerNeuralNetwork` .

    Parámetros:
    - `layers_definition` : se trata de una lista con la definición de cada una de las capas que componen el sistema, siendo cada elemento una lista con dos elementos, el primero un entero que hace referencia al número de neurona de la capa y el segundo al tipo de función de activación utilizada en las neuronas de esa capa. De tal forma que el primer elemento de la lista representa la capa 0 del sistema y por tanto la dimensión de las muestras, en este caso no es necesario definir una función de activación. `[[784], [20, ActivationFunctionsEnum.SIGMOID_FUNCTION], [10, ActivationFunctionsEnum.SIGMOID_FUNCTION]]` , en este caso se está definiendo una red con una capa de entrada de 784 entradas, una capa oculpa con 20 neuronas que implementan una función de activación de tipo sigmoide y una capa de salidar con 10 neuronas que implementan una función de activación de tipo sigmoide.
- Atributos:
  - `__number_of_classes` : atributo de tipo entero que indica el número de clases del sistema.
  - `__layers` : lista de listas de tipo `Neuron` , se construye a partir del atributo `layers_definition` recibido a través del constructor.
  - `__learning_rate` : atributo que define la velocidad de aprendizaje del algoritmo, valores próximos a cero suponen convergencias más suaves pero con más iteraciones.

Figura 6.4. Documentación Jupyter Notebook, descripción de los elementos de la red.

La Figura 6.5 muestra un extracto de código correspondiente a la implementación de la red neuronal multicapa.

```
class MultilayerNeuralNetwork:

    def __init__(self, layers_definition):
        self.__number_of_classes = layers_definition[-1][MULTILAYER_NEURAL_NETWORK.get('NEURON_DIMENSION')]
        self.__layers = [
            [
                Neuron(
                    layers_definition[layer - 1][MULTILAYER_NEURAL_NETWORK.get('NEURON_DIMENSION')],
                    layers_definition[layer][MULTILAYER_NEURAL_NETWORK.get('NEURON_ACTIVATION_FUNCTION')]
                )
                for _ in range(layers_definition[layer][MULTILAYER_NEURAL_NETWORK.get('NEURON_DIMENSION')])
            ]
            for layer in range(1, len(layers_definition))
        ]
        self.__learning_rate = MULTILAYER_NEURAL_NETWORK.get('LEARNING_RATE_DEFAULT_VALUE')
        self.__max_epochs = MULTILAYER_NEURAL_NETWORK.get('MAX_EPOCHS_DEFAULT_VALUE')
        self.__misclassified_samples_per_epoch = []

    @property
    def number_of_classes(self):
        return self.__number_of_classes

    @property
    def number_of_layers(self):
        return len(self.__layers)
```

Figura 6.5. Documentación Jupyter Notebook, implementación de la red.

La Figura 6.6 muestra un ejemplo de uso del sistema con un conjunto de datos reales como es el de *MNIST,*[7] con el que se entrena y prueba el sistema.

7.3.1. Ejemplo de uso de una red neuronal artificial multicapa.

Ubicación: `usage_multilayer_neural_network.py`

Uso: `python src/usage_multilayer_neural_network.py`

En este archivo se realiza un ejemplo de uso de una red neuronal artificial multicapa con un set de datos real como es el de mnist.

Se dispone de cuatro archivos .csv en el directorio `/resources/data/mnist` estos archivos contienen los valores de las imágenes de 28 x 28 píxeles en escala de grises de los dígitos manuscritos comprendidos entre el 0 y el 9, ambos inclusive:

- `mnist_train_40K_samples.csv` e `mnist_train_40K_labels.csv` son las muestras y etiquetas utilizadas para el entrenamiento del sistema.
- `mnist_test_10K_samples.csv` e `mnist_test_10K_labels.csv` son las muestras y etiquetas utilizadas para obtener la tasa de error obtenido por el sistema una vez entrenado.

En primer lugar se cargan los datos `train_samples`, `train_labels`, `train_labels` y `test_labels` haciendo uso de `CSVData`. Posteriormente se normalizan estos haciendo uso de `Normalization.z_score(data)`.

Instanciamos un `MultilayerNeuralNetwork(train_samples_normalized.shape[1], 10)`, donde `train_samples_normalized.shape[1] = dimensión de la primera muestra = 784` y `10` es el número de clases diferentes. Entrenamos el sistema con `multilayer_neural_network.train(train_samples_normalized, train_labels, LossFunctionsEnum.MSE_FUNCTION, 0.1, 20)`.

Calculamos la tasa de error obtenida por el sistema haciendo uso de la función `multilayer_neural_network.calculate_error_rate(test_samples_normalized, test_labels)` y lo mostramos por consola.

```
train_samples = CSVDataLoader.load_samples('./resources/data/mnist/mnist_train_40K_samples.csv', ';', False)
train_labels = CSVDataLoader.load_labels('./resources/data/mnist/mnist_train_40K_labels.csv', ';', False)
test_samples = CSVDataLoader.load_samples('./resources/data/mnist/mnist_test_10K_samples.csv', ';', False)
test_labels = CSVDataLoader.load_labels('./resources/data/mnist/mnist_test_10K_labels.csv', ';', False)

train_samples_normalized = Normalization.z_score(train_samples)
test_samples_normalized = Normalization.z_score(test_samples)

multilayer_neural_network = MultilayerNeuralNetwork([[train_samples_normalized.shape[1]],
                                                     [20, ActivationFunctionsEnum.SIGMOID_FUNCTION],
                                                     [10, ActivationFunctionsEnum.SIGMOID_FUNCTION]])
multilayer_neural_network.train(train_samples_normalized, train_labels, LossFunctionsEnum.MSE_FUNCTION, 0.1, 20)

NeuralSystemPicker.save_neural_system('./resources/serialized_objects/multilayer_neural_network', multilayer_neural_
multilayer_neural_network = NeuralSystemPicker.load_neural_system('./resources/serialized_objects/multilayer_neural_

print(multilayer_neural_network.calculate_error_rate(test_samples_normalized, test_labels))
```

Figura 6.6. Documentación Jupyter Notebook, ejemplo de uso de la red.

[7] http://yann.lecun.com/exdb/mnist base de datos pública de dígitos manuscritos.

7. Conclusiones.

Echando la vista atrás al apartado 1.2 correspondiente a los objetivos del presente proyecto se puede concluir que se han desarrollado cada uno de los puntos mencionados en dicho apartado de forma satisfactoria:

- Creación de una memoria con el desarrollo del proyecto.
- Implementación de una librería escrita en *Python* que desarrolla los tres sistemas de clasificación propuestos: perceptrón, clasificador multiclase y redes neuronales profundas.
- Se ha generado una guía sobre el uso y la implementación de la librería en diferentes formatos e idiomas (castellano e inglés).
- Se ha creado un repositorio público en GitHub donde se aloja tanto la librería como la documentación.

El desarrollo de este proyecto me ha servido para afianzar conocimientos vistos a lo largo de la carrera relativos a la inteligencia artificial. Algunos conceptos, como las redes neuronales multicapa o el algoritmo de *Backpropagation,* se habían tratado de forma teórica sin profundizar en aspectos de implementación. Son estos conceptos los que se han tenido que profundizar para poder realizar el desarrollo de la librería.

La elección del lenguaje de programación para la implementación de la librería ha sido *Python*. Ya que en general a lo largo de la carrera el lenguaje de programación más utilizado es *Java,* consideraba que el trabajo final de grado podría ser un buen momento para aprender y familiarizarme con un lenguaje de programación nuevo.

Este proyecto también me ha servido para poner en práctica recursos y herramientas estudiadas en asignaturas como son *Ingeniería del software* y *Gestión de proyectos*, haciendo uso de un sistema de control de versiones como es *Git* y una herramienta de gestión como *Trello* para la planificación del proyecto.

8. Trabajos futuros.

Dada la naturaleza didáctica de la librería, sería interesante que esta pudiera servir a otros desarrolladores para iniciarse en el mundo de la inteligencia artificial y que puedan participar en el desarrollo y crecimiento de ésta. Es por ello por lo que se crea un repositorio público con una extensa documentación.

En esta librería se han desarrollado tres sistemas de clasificación. Algunos de estos sistemas, como el perceptrón o la red neuronal multicapa, son los elementos fundamentales de otros sistemas de clasificación que se podrían construir a partir de éstos, como el caso de las redes neuronales convolucionales. Así, haciendo uso de la mayoría de los elementos que componen la librería y con unas pocas implementaciones nuevas, este tipo de redes podrían formar parte de los sistemas que actualmente componen la librería.

En el caso de la red neuronal artificial implementada se ha desarrollado la función de coste del *error cuadrático medio* y la función de activación *sigmoide*. Otro de los posibles desarrollos sería implementar nuevas funciones de coste y funciones de activación que doten al sistema de mayor riqueza.

Seguido de cualquier desarrollo se debe completar siempre la documentación del *Jupyter Notebook*, ya que la ventaja de este proyecto reside precisamente en la documentación de éste.

Respecto a las líneas de desarrollo que no deben seguirse en este proyecto, serían relativas a la mejora de la eficiencia de la librería siempre y cuando éstas supongan una penalización en la legibilidad del proyecto.

9. Bibliografía.

[1] F. Rosenblatt, «The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain,» *Psychological Review,* vol. 65, nº 6, pp. 386-407, 1958.

[2] B. Smith y G. Linden, «Two Decades of Recommender Systems at Amazon.com,» *IEEE Internet Computing,* vol. 21, nº 3, pp. 12-18, 2017.

[3] W. E. Forum, «weforum,» 2019. [En línea]. Available: https://www.weforum.org/platforms/shaping-the-future-of-technology-governance-artificial-intelligence-and-machine-learning. [Último acceso: 22 04 2020].

[4] TensorFlow, «TensorFlow,» [En línea]. Available: https://www.tensorflow.org. [Último acceso: 03 06 2020].

[5] PyTorch, «PyTorch,» [En línea]. Available: https://pytorch.org. [Último acceso: 23 05 2020].

[6] GitHub, «GitHub,» 24 01 2019. [En línea]. Available: https://github.blog/2019-01-24-the-state-of-the-octoverse-machine-learning/. [Último acceso: 15 05 2020].

[7] I. Popova, «Light it,» 2020. [En línea]. Available: https://light-it.net/blog/top-10-python-libraries-for-machine-learning/. [Último acceso: 15 05 2020].

[8] C. D, «Medium,» 25 03 2020. [En línea]. Available: https://towardsdatascience.com/best-python-libraries-for-machine-learning-and-deep-learning-b0bd40c7e8c. [Último acceso: 15 05 2020].

[9] C. Williams, «Medium,» 03 09 2019. [En línea]. Available: https://medium.com/@cwillycs/committing-to-pytorch-by-someone-who-doesnt-know-a-ton-about-pytorch-fa222253cf2d. [Último acceso: 15 05 2020].

[10] W. McCulloch y W. Pitts, «A logical calculus of the ideas immanent in nervous activity,» *Bulletin of Mathematical Biophysics,* vol. 5, pp. 115-133, 1943.

[11] F. Rosenblatt, Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms, Nueva York: Spartan Books, 1962.

[12] M. Minsky y S. Papert, Perceptrons: An Introduction to Computational Geometry, Cambridge: Mit Press, 1969.

[13] D. Rumelhart, G. Hinton y R. Williams, «Learning representations by back-propagating errors,» *Nature,* vol. 323, pp. 533-536, 1986.

[14] G. Hinton y R. Salakhutdinov, «Reducing the Dimensionality of Data with Neural Networks,» *Science,* vol. 313, nº 5786, pp. 504-507, 2006.

[15] IBM, «IBM Knowledge Center,» [En línea]. Available: https://www.ibm.com/support/knowledgecenter/es/SS3RA7_sub/modeler_mainhelp_client_ddita/components/neuralnet/neuralnet_model.html. [Último acceso: 19 04 2020].

[16] J. R. Hilera González y V. J. Martínez Hernando, Redes neuronales artificiales: fundamentos, modelos y aplicaciones, RA-MA Editorial, 1994.

[17] A. Novikoff, «On convergence proofs for perceptrons,» *Proceedings of the Symposium on Mathematical Theory of Automata,* vol. XII, pp. 615-622, 1962.

[18] Stack Overflow, «Developer Survey Results,» 2020. [En línea]. Available: https://insights.stackoverflow.com/survey/2019#most-popular-technologies. [Último acceso: 10 05 2020].

[19] P. Bharadwaj, «Medium,» 01 2017. [En línea]. Available: https://towardsdatascience.com/perceptron-and-its-implementation-in-python-f87d6c7aa428. [Último acceso: 14 04 2020].

[20] J.-C. Loiseau, «Medium,» 11 Marzo 2019. [En línea]. Available: https://towardsdatascience.com/rosenblatts-perceptron-the-very-first-neural-network-37a3ec09038a. [Último acceso: 17 04 2020].