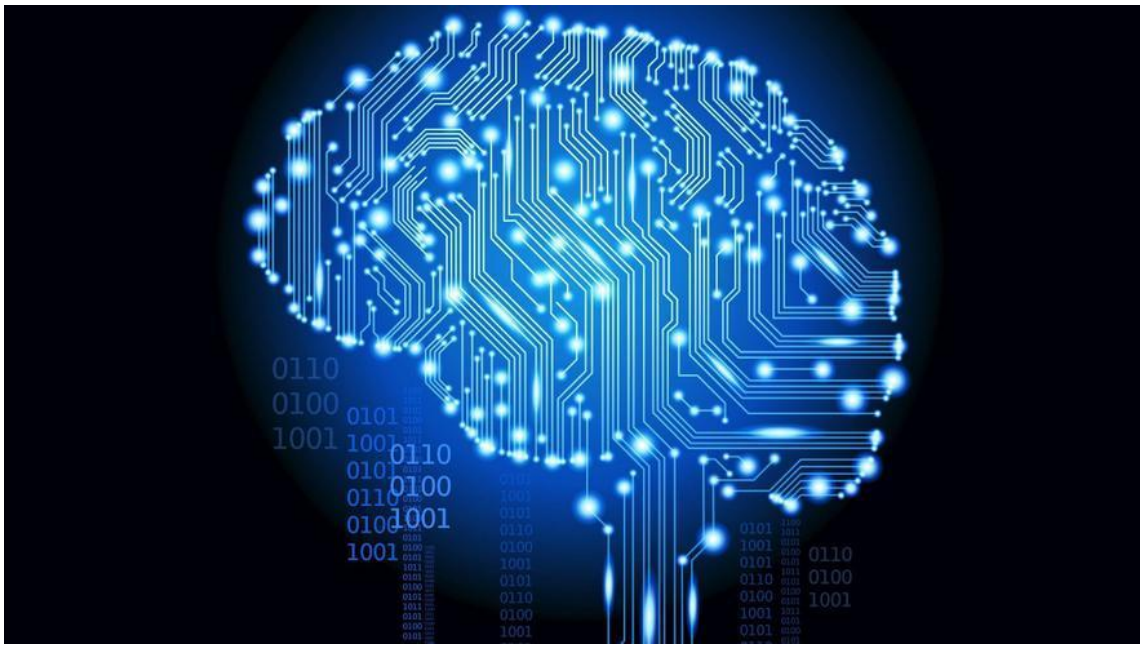
# PROYECTO FUNDAMENTOS DE APRENDIZAJE AUTOMÁTICO

**Análisis y Comparativa de Funciones de Activación en Modelos de Redes Neuronales**



***Trabajo realizado por:***

*Alicia Alonso Dubost, Berta Pelegrina Martínez y Laura Sánchez Garzón*

*Grado en Ingeniería Biomédica*

*Curso 2023/2024*

# Tabla de contenido

# 1. INTRODUCCIÓN

En la era actual de la inteligencia artificial, su desarrollo y expansión se manifiestan como un horizonte sin límites aparentes. Utilizada para abordar una variedad de problemas complejos, desde la automatización de procesos, hasta la capacidad de aprendizaje de las máquinas, la IA se ha convertido en uno de los paradigmas del siglo XXI.

Para entenderla en su conjunto, se ha de comprender cómo funcionan sus bases, y entre ellas se encuentran los algoritmos de entrenamiento. Existen infinidad de ellos, pero queda demostrado que una de las mejores estructuras de aprendizaje automático son las redes neuronales, destacando por ser una de las estructuras más efectivas, basadas en tratar de replicar el funcionamiento de las neuronas humanas, a la hora de procesar información y aprender patrones.

Este trabajo de investigación trata de analizar, comparar y evaluar diferentes funciones de activación para algunos de los algoritmos de redes neuronales más exitosos en cuanto a inteligencia artificial se refiere, con el objetivo de encontrar qué funciones de activación (o qué combinaciones) son las que generan mayor rendimiento y capacidad de aprendizaje.

## 1.1. Dataset, preprocesamiento y visualización de datos

El dataset escogido ha sido ***Heart Disease Dataset*** (heart.csv). Consta de doce variables:

- ***Age****: age of the patient [years]*
- ***Sex:*** *sex of the patient [M: Male, F: Female]*
- ***ChestPainType:*** *chest pain type [TA: Typical Angina, ATA: Atypical Angina, NAP: Non-Anginal Pain, ASY: Asymptomatic]*
- ***RestingBP:*** *resting blood pressure [mm Hg]*
- ***Cholesterol****: serum cholesterol [mm/dl]*
- ***FastingBS:*** *fasting blood sugar [1: if FastingBS > 120 mg/dl, 0: otherwise]*
- ***RestingECG:*** *resting electrocardiogram results [Normal: Normal, ST: having ST-T wave abnormality (T wave inversions and/or ST elevation or depression of > 0.05 mV), LVH: showing probable or definite left ventricular hypertrophy by Estes' criteria]*
- ***MaxHR:*** *maximum heart rate achieved [Numeric value between 60 and 202]*
- ***ExerciseAngina:*** *exercise-induced angina [Y: Yes, N: No]*
- ***Oldpeak:*** *oldpeak = ST [Numeric value measured in depression]*
- ***ST_Slope:*** *the slope of the peak exercise ST segment [Up: upsloping, Flat: flat, Down: downsloping]*
- ***HeartDisease:*** *output class [1: heart disease, 0: Normal]*

Inicialmente el dataset constaba de 918 instancias, (Dimensión del dataset: (918, 12)). En el preprocesamiento realizamos varias pruebas. En primer lugar, nos aseguramos de que no se presentan *missing values* y a continuación buscamos las posibles incoherencias. En este caso las variables categóricas presentan los valores correctos indicados anteriormente. Lo único que hemos tenido que trabajar son los atributos *RestingBP* y *Cholesterol*, puesto que toman en algunos valores 0 y no es coherente, pues no corresponde con los valores fisiológicos.

Al tener un dataset con 918 nos podemos permitir eliminar la única instancia que toma valor 0 para *RestingBP* y las 172 instancias de *Cholesterol*. En este caso, decidimos eliminarlas en vez de

realizar una imputación ya que preferimos un menor número de datos, pero una mayor precisión a la hora de decidir si una persona presenta un problema del corazón o no para evitar posibles problemas de aprendizaje.

Para una primera visualización, se aplica la matriz de correlación, que permite dar una idea general de cuánto de correlacionadas están las variables entre sí. El resultado nos da una idea de que no va a ser un problema fácilmente separable, pero hay cierta correlación de alguna manera entre columnas.

| | Age | RestingBP | Cholesterol | FastingBS | MaxHR | Oldpeak | HeartDisease |
|---|---|---|---|---|---|---|---|
| **Age** | 1.000000 | 0.259865 | 0.058758 | 0.241338 | -0.382112 | 0.286006 | 0.298617 |
| **RestingBP** | 0.259865 | 1.000000 | 0.095939 | 0.173765 | -0.125774 | 0.198575 | 0.173242 |
| **Cholesterol** | 0.058758 | 0.095939 | 1.000000 | 0.054012 | -0.019856 | 0.058488 | 0.103866 |
| **FastingBS** | 0.241338 | 0.173765 | 0.054012 | 1.000000 | -0.102710 | 0.055568 | 0.160594 |
| **MaxHR** | -0.382112 | -0.125774 | -0.019856 | -0.102710 | 1.000000 | -0.259533 | -0.377212 |
| **Oldpeak** | 0.286006 | 0.198575 | 0.058488 | 0.055568 | -0.259533 | 1.000000 | 0.495696 |
| **HeartDisease** | 0.298617 | 0.173242 | 0.103866 | 0.160594 | -0.377212 | 0.495696 | 1.000000 |

*Figura 1: Matriz de correlación del dataset mediante valores numéricos*

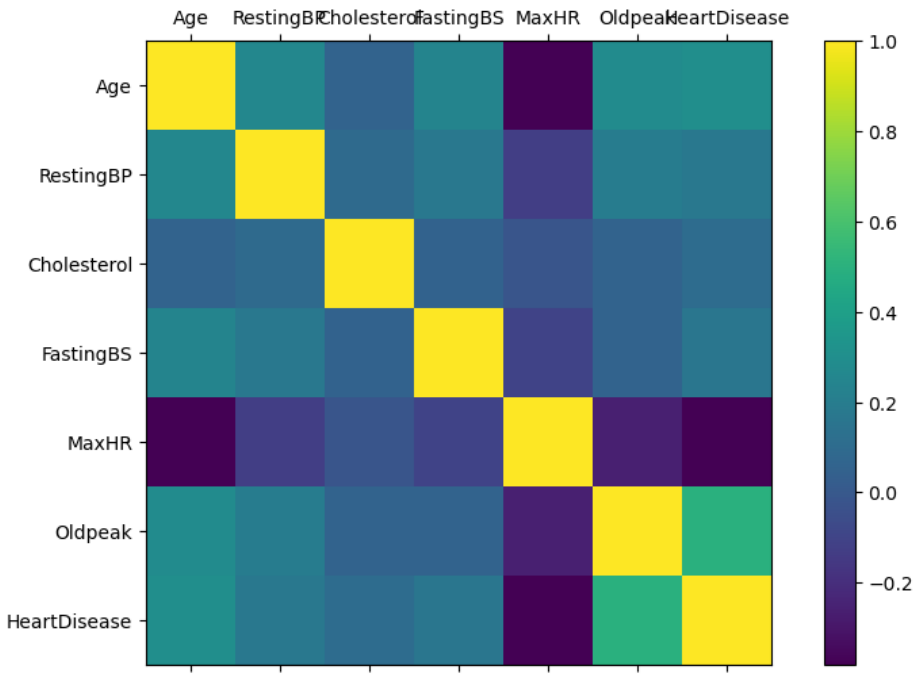



*Figura 2: Matriz de correlación del dataset mediante colores; colores más claros implican mayor correlación entre variables, y viceversa.*

Para una visualización algo más comprensible, se grafican los diagramas de dispersión en dos dimensiones. Concluimos que el dataset no es separable linealmente. En primer lugar, es complicado separar de manera lineal los gráficos de puntos. En segundo lugar, en la diagonal, podemos ver como el no presentar una enfermedad cardiaca (azul) y el presentarla (naranja) se superponen para todas las variables.

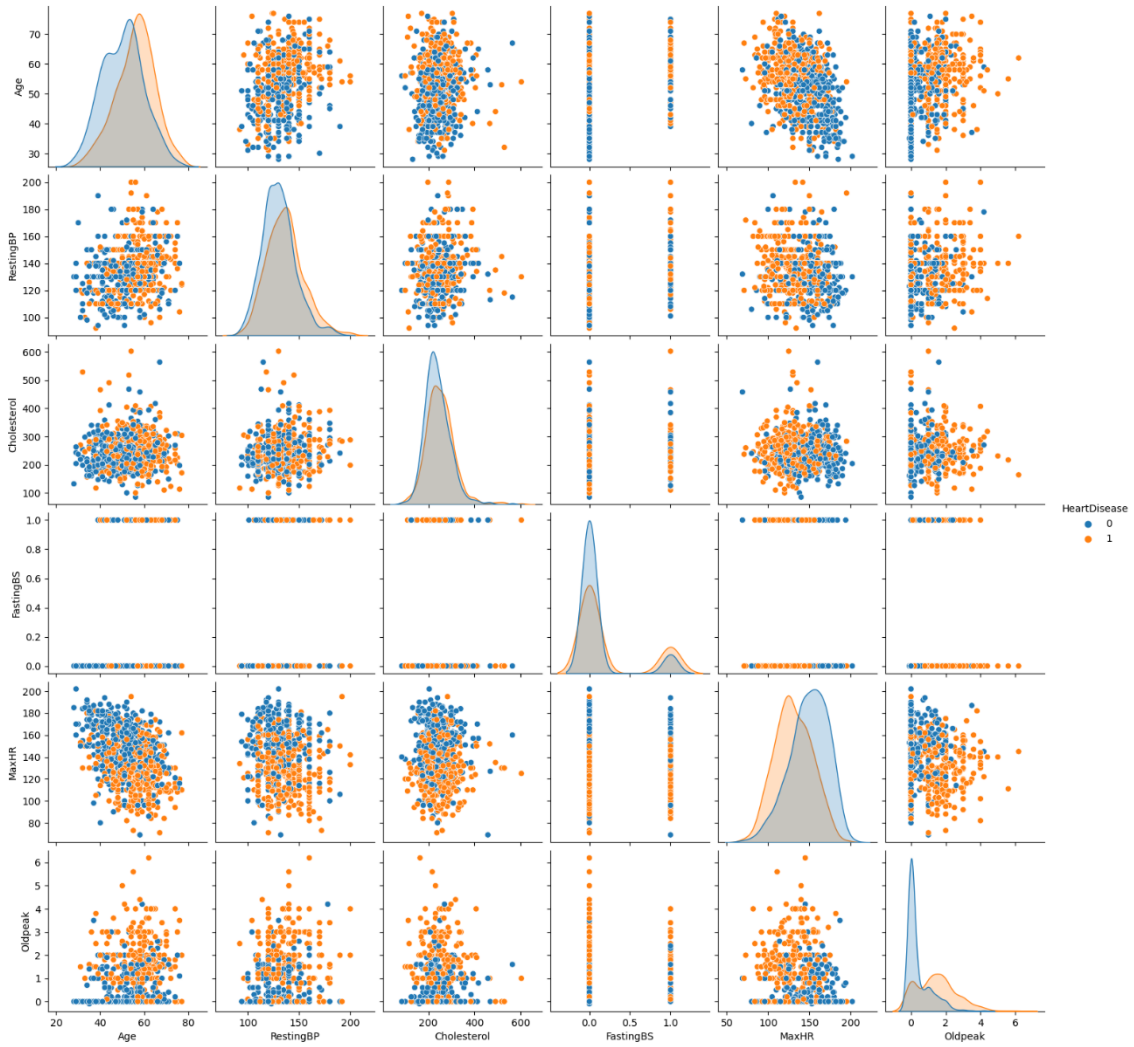



*Figura 3: visualización de la relación entre el target y cada variable. Resultados poco concluyentes.*

No obstante, hay casos que no son separables en 2D, pero en más dimensiones, lo son. Dibujamos los gráficos en tres dimensiones.

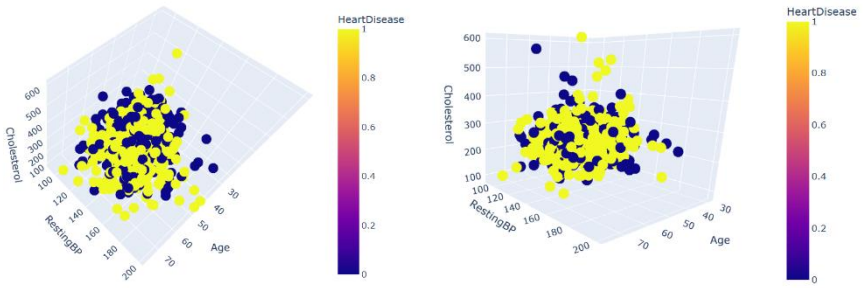



*Figura 4: Representación en 3D para visualizar la relación entre Cholesterol, RestingBP y Age.*

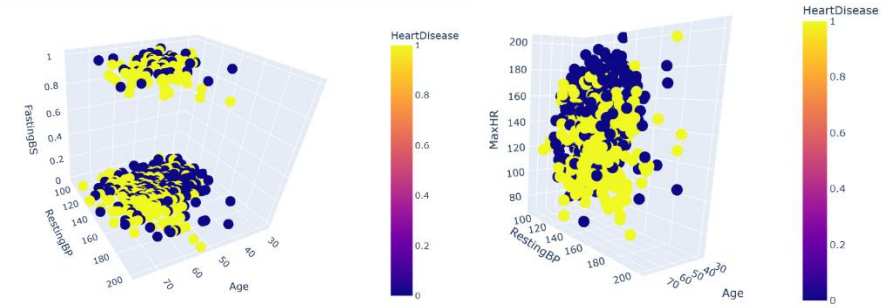



*Figura 5: Representación en 3D para visualizar la relación entre Fastings, RestingBP y Age y MaxHR, RestingBP y Age.*

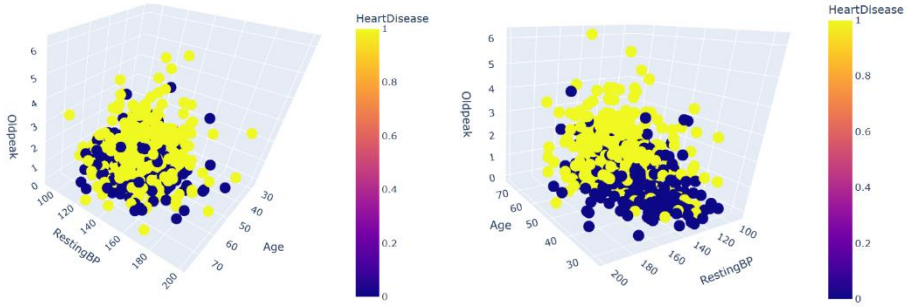



*Figura 6: Representación en 3D para visualizar la relación entre Oldpeak, RestingBP y Age.*

Tras este visualizado, vemos que no es un problema linealmente separable, al menos en pocas dimensiones.

## 1.2. Funciones de Activación

Se han estudiado para diferentes redes neuronales cuatro funciones de activación diferentes.

### 1.2.1. Función *ReLu* (Rectified Linear Unit)

Matemáticamente, queda representada por la función $f(x) = max(0, x)$ , es decir, si el valor de entrada $x$ es positivo, devuelve el mismo valor; si en cambio es negativo o cero, la función devuelve cero. Analíticamente se ve tal que:

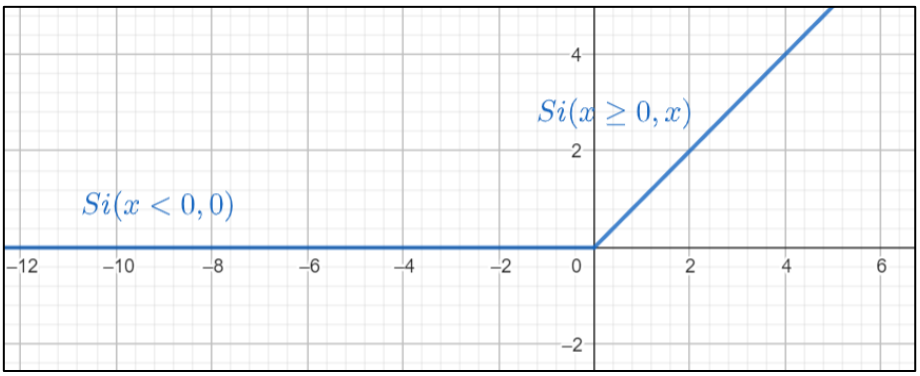



*Figura 7: Representación analítica de la función ReLu.*

La función *ReLu* destaca por su simplicidad y eficiencia computacional; "tomar el máximo" es simple y no requiere una función exponencial costosa.

Además, ayuda a mitigar el problema del desvanecimiento del gradiente al permitir que los gradientes fluyan sin cambiar en la parte positiva (observar su función analítica en la derivada); pero puede causar el efecto contrario y generar gradientes grandes durante el entrenamiento, afectando a la estabilidad del modelo.

Por el otro lado, tiende a activar solo un subconjunto de unidades en una capa, ya que cualquier entrada negativa se mapea a cero, promoviendo gran eficiencia en muchos casos; aunque puede suponer a su vez el problema de neuronas muertas, conocido como "*dying ReLU problem*", que nunca se activan y, por lo tanto, no aprenden ninguna información. Se han propuesto variaciones de *ReLU*, como *Leaky ReLU* o *Parametric ReLU*, para abordar este problema manteniendo una pequeña pendiente para valores negativos.

Su derivada, $f'(x) = 0, x < 0;\ 1, x > 0$ no satura para valores positivos, lo que mitiga el problema del desvanecimiento del gradiente, que puede ocurrir durante el entrenamiento cuando las derivadas son muy pequeñas. Además, es computacionalmente eficiente, ya que simplemente activa todos los valores positivos y establece a cero los valores negativos.

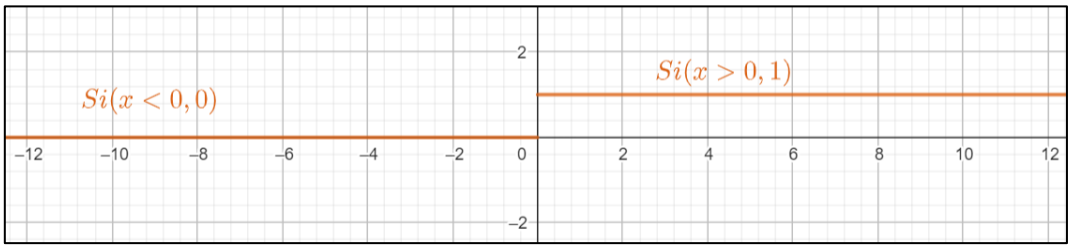



*Figura 8: Representación analítica de la función ReLu derivada.*

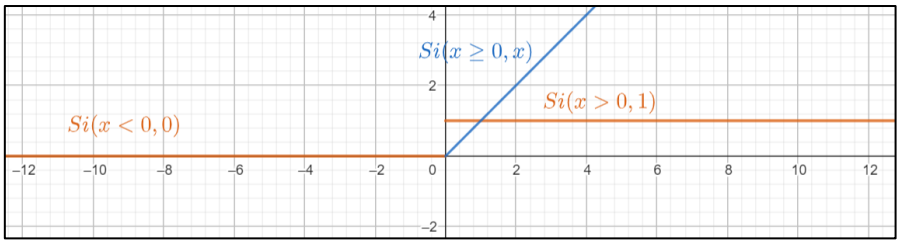



*Figura 9: Comparativa entre ReLu y ReLu'.*

Es una función que, debido a sus propiedades no lineales, es capaz de mitigar el desvanecimiento del gradiente; y por su eficiencia computacional y derivada simple, es útil en perceptrones multicapa, retropropagación y redes convolucionales. En cambio, no se utiliza comúnmente para redes LSTM, ya que está diseñada para abordar los desafíos particulares asociados con el aprendizaje en secuencias de datos a largo plazo, y la *ReLU* puede tener limitaciones en la capacidad para manejar dependencias a largo plazo.

Es por eso, que, en este proyecto de investigación, se ha utilizado como función de activación fija para hallar la arquitectura de cada algoritmo de redes (excepto para la SLTM).

### 1.2.2. Función tanh (Tangente Hiperbólica )

La Tangente Hiperbólica, $f(x) = \frac{e^{2x}-1}{e^{2x}+1}$ , es una función de activación simétrica en torno al origen y continua que tiene una salida en el intervalo [-1, 1] (mirar Figura 10). Esto significa que, para entradas negativas, la salida es negativa, y para entradas positivas, la salida es positiva.

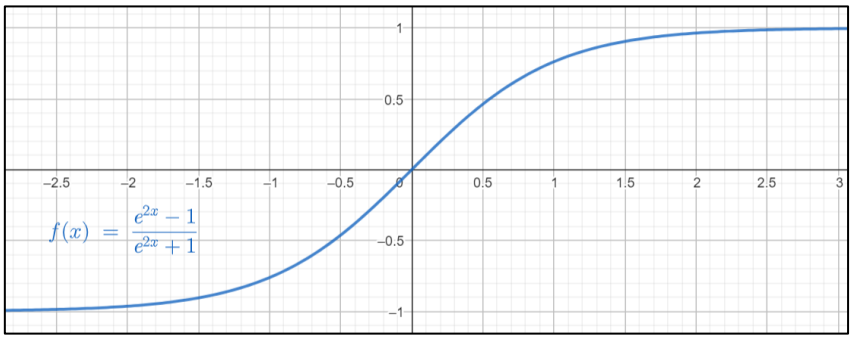



*Figura 10: Representación analítica de la función tanh.*

Esto la convierte en una opción ideal para redes neuronales que requieren una salida balanceada, y, además, al ser una función no lineal, permite a las redes neuronales modelar relaciones no lineales entre los datos. La función de activación se aplica a cada neurona de la capa oculta para determinar si la neurona debe «disparar» o no, enviando el resultado a la siguiente capa de neuronas, y así sucesivamente hasta que se obtiene la salida final.

Aunque ha perdido popularidad en los últimos años debido a la creciente popularidad de otras funciones de activación, sigue siendo una opción efectiva para mejorar el rendimiento de las redes neuronales y aprender representaciones más complejas de los datos. Es una de las funciones de activación con mayor precisión en las RNN; y su derivada, $f'(x) = 1 - tanh^2(x)$ , es utilizada durante el entrenamiento de redes neuronales mediante el algoritmo de retropropagación.

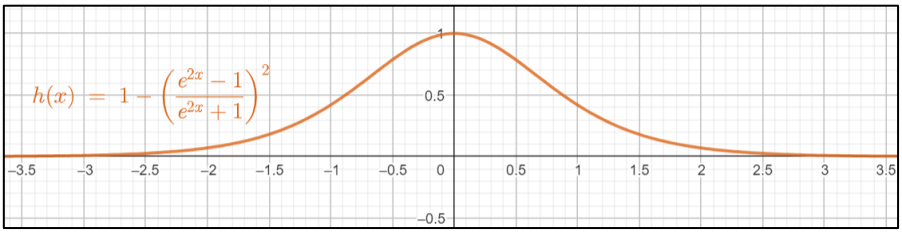



*Figura 11: Representación analítica de la función tanh derivada.*

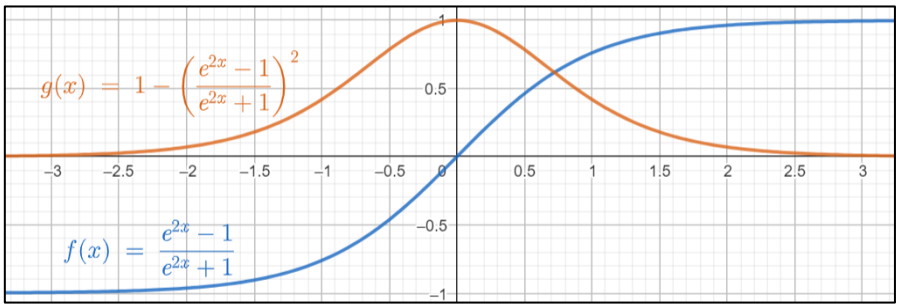



*Figura 12: Comparativa entre tanh y tanh'*

### 1.2.3. Función Sigmoide

Matemáticamente, $f(x) = \frac{1}{1+e^{-x}}$, esta función produce y acota valores en el rango de (0, 1). La salida se acerca a 0 para entradas muy negativas y se acerca a 1 para entradas muy positivas. A diferencia de la tangente hiperbólica, que produce valores en el rango de [-1, 1], la función sigmoide tiene un rango de (0, 1) y, por tanto, no es simétrica alrededor del origen. Gráficamente:

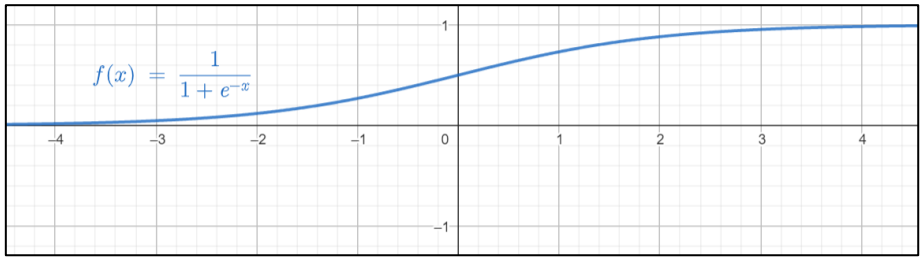



*Figura 13: Representación analítica de la función sigmoide.*

La función sigmoide se utiliza comúnmente en capas de salida de modelos de clasificación binaria, donde la tarea consiste en asignar una etiqueta a una de dos clases. Es por eso, que, en este trabajo, dado que la salida de nuestro *target* es binaria, se ha utilizado como función fija en la capa de salida.

Actualmente, es una función que se ha quedado obsoleta, y suele ser sustituida por *ReLu*, debido a su capacidad para mitigar el problema del desvanecimiento del gradiente, y llegar a una convergencia más rápida durante el entrenamiento. Aun así, su derivada, $f(x)(1-f(x))$, es particularmente útil en la retropropagación, porque tiene una forma simple y permite un cálculo eficiente del gradiente.

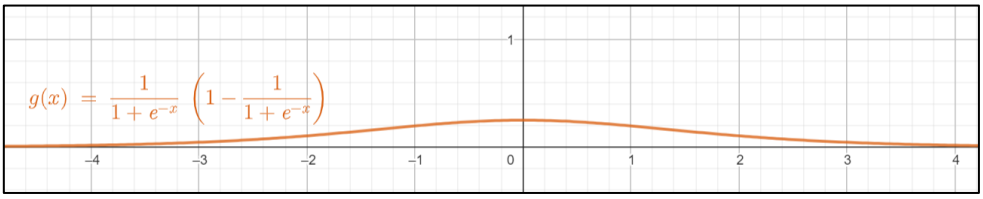



*Figura 14: Representación analítica de la función sigmoide derivada.*

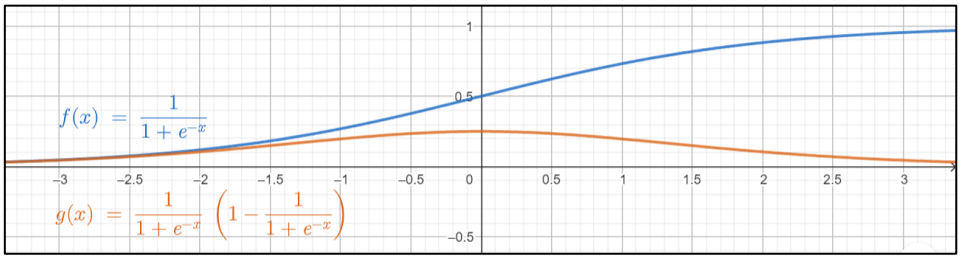



*Figura 15: Comparativa entre sigmoide y sigmoide'.*

### 1.2.4. Función de activación lineal

Se trata de una función simple. Matemáticamente, $f(x) = x$ , es decir, tiene una tasa de cambio constante y siempre igual a 1 para cualquier valor de x. Gráficamente:

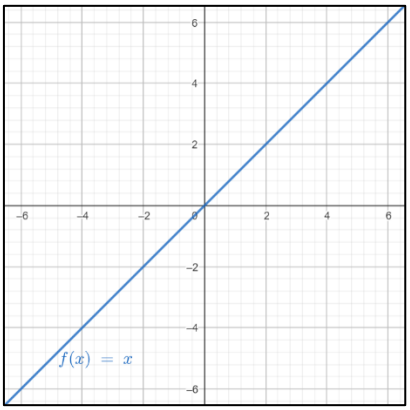



*Figura 16: Representación analítica de la función lineal.*

Es una función que se utiliza particularmente en la regresión, y en especial, en la capa de salida, porque predice un valor continuo en lugar de asignar a una clase como en la clasificación. Sin embargo, en las capas intermedias de una red neuronal, la función de activación lineal no se utiliza comúnmente, dado que es limitada al aprender patrones complejos y no lineales. Aun así, se ha considerado interesante aplicar esta función a cada modelo de redes neuronales tratados en este proyecto, por ver la comparativa entre funciones de activación.

Su derivada, siempre igual a 1 para cualquier valor de $x$, en el contexto de la retropropagación facilita el cálculo del gradiente en comparación con algunas funciones de activación no lineales más complejas.

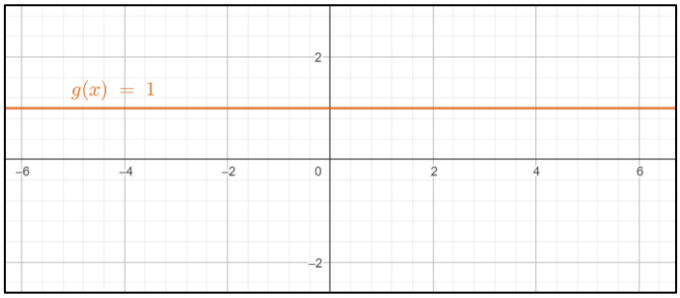



*Figura 17: Representación analítica de la función lineal derivada.*

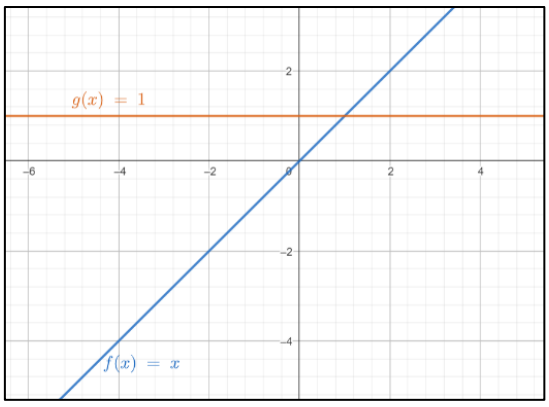



*Figura 18: Comparativa entre lineal y lineal'.*

# 2. ANÁLISIS Y COMPARATIVA EN LOS DISTINTOS ALGORITMOS

El objetivo de este trabajo es estudiar los efectos de las funciones de activación en el aprendizaje de nuestro dataset usando distintos algoritmos. Para ello, se ha planteado lo siguiente para cada uno de los algoritmos elegidos:

1. Definir la estructura de la red neuronal: número de capas ocultas y número de neuronas por capa oculta. Esto se ha hecho usando la función de activación mejor a priori para cada algoritmo teniendo en cuenta el problema propuesto con nuestro dataset.
2. Elegir un error objetivo que sirva para determinar cuándo damos el entrenamiento de nuestra red por finalizado.
3. Calcular el número de épocas y el tiempo transcurrido hasta llegar a dicho error haciendo uso de cada una de las funciones de activación elegidas.

### Backpropagation

La retropropagación del error, también conocida como la regla delta generalizada es un algoritmo de entrenamiento para redes *feedforward* multicapa. En este proyecto vamos a trabajar con modelos *Feedforward* como el perceptrón multicapa, además de con otras redes como RNN y CNN, por ello, utilizaremos este algoritmo muchas ocasiones.

Para ello utilizamos el método *fit()* que nos ofrece *TensorFlow*; se utiliza para entrenar modelos de aprendizaje profundo, de tal manera que la retropropagación se realiza de manera automática durante el proceso de entrenamiento. Durante cada época, la función *fit()* realiza iteraciones a través de los lotes de datos de entrenamiento. En cada iteración, los gradientes de la función de pérdida con respecto a los parámetros del modelo se calculan automáticamente utilizando la retropropagación.

## 2.1. Multi-layer Perceptron

El Perceptrón Multicapa (MLP, por sus siglas en inglés) es una red neuronal artificial que se utiliza para el aprendizaje supervisado en el campo del aprendizaje automático. El MLP está compuesto por múltiples capas de neuronas interconectadas, en las que las salidas de las neuronas de una capa se convierten en entradas para la siguiente capa. La primera capa se llama capa de entrada, la última capa se llama capa de salida y las capas intermedias se llaman capas ocultas.

El MLP es capaz de realizar tareas de clasificación y regresión, y puede ser utilizado para problemas en los que la relación entre las entradas y las salidas no es lineal. El algoritmo de entrenamiento utilizado por el MLP se basa en la propagación hacia atrás (*backpropagation*), que ajusta los pesos de las conexiones de la red para minimizar el error de predicción entre las salidas producidas por la red y las salidas deseadas.

El MLP es uno de los modelos de red neuronal más utilizados en el aprendizaje automático debido a su capacidad para modelar relaciones no lineales y su capacidad de generalización. Sin embargo, el entrenamiento de un MLP puede ser computacionalmente costoso y puede requerir un gran conjunto de datos de entrenamiento para evitar el sobreajuste (*overfitting*). Además, la elección adecuada de la arquitectura de la red, incluyendo el número de capas y el número de neuronas en cada capa, es un desafío importante en la construcción de un MLP eficaz para un problema dado.

A pesar de estos desafíos, el MLP ha sido utilizado con éxito en una variedad de aplicaciones de aprendizaje automático, como la detección de fraudes en tarjetas de crédito, el análisis de sentimientos en redes sociales y la identificación de objetos en imágenes. Además, el MLP ha sido utilizado como una base para el desarrollo de modelos de red neuronal más complejos, como las redes neuronales convolucionales (CNN) y las redes neuronales recurrentes (RNN).

Para este proyecto se ha trabajado con dos modelos de MLP, en el primero generamos un modelo secuencial, que es una pila lineal de capas, mediante *Sequential()* y después se le añaden capas mediante *add*. En el segundo modelo, importamos *tensorflow* y hacemos uso de *layers* para incluir directamente las capas densas al generar el modelo. Además en el primer modelo se utiliza únicamente el método *fit()* que utiliza un optimizador por defecto, mientras que en el segundo se le indica explícitamente.

En lo relacionado con los pesos, para el modelo 1 *Keras* inicializa por pesos por defecto, en cambio, en el segundo es *TensorFlow* el que se encarga, de tal manera que la inicialización de estos podría ser diferente, afectando a los resultados.

## Selección de la red neuronal

### *Una capa oculta de 15 neuronas*

Para la selección del número de capas ocultas y del número de neuronas en cada una de ellas hemos empezado con una red neuronal de una capa oculta. Hemos observado que el mejor valor se obtiene con 15 neuronas. A continuación, se muestran las gráficas de *accuracy* y loss del conjunto de entrenamiento y del conjunto de validación.

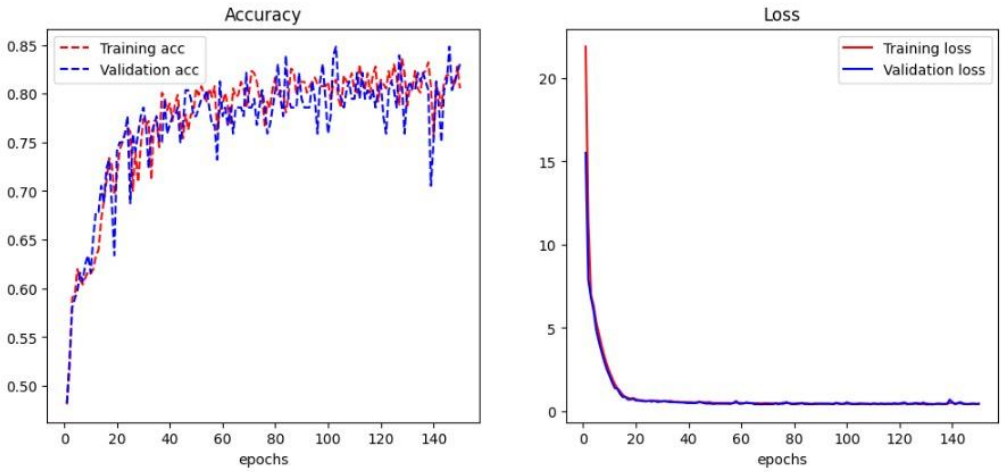



*Figura 19: Gráficas que muestran el accuracy y el loss, al diseñar una red MLP, de una capa oculta, con 15 neuronas.*

No parece haber sobreentrenamiento: los valores de las métricas se estabilizan tanto para el conjunto de entrenamiento como el de validación.

### *Dos capas ocultas de 15 y 10 neuronas*

Repetimos el proceso con dos capas ocultas de 15 y 10 neuronas respectivamente para ver si conseguimos que nuestro modelo mejore.

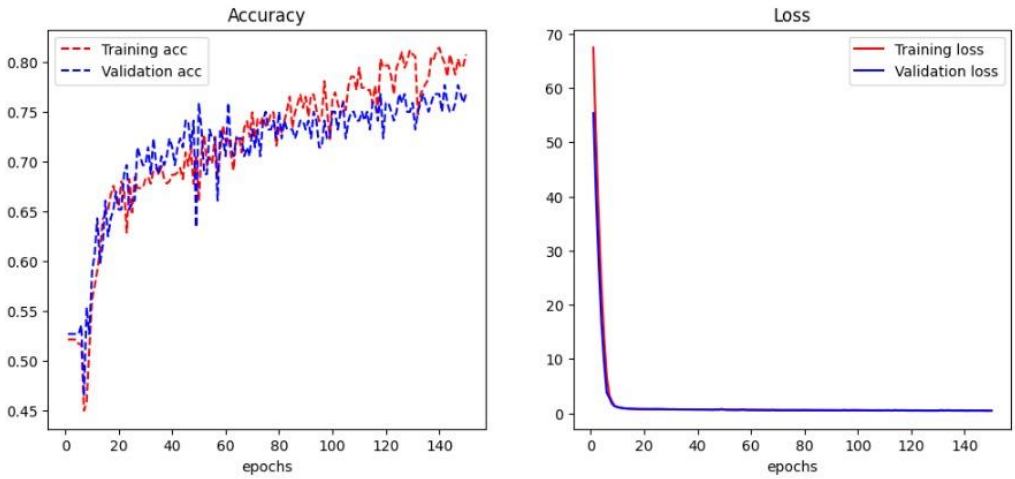



*Figura 20: Gráficas que muestran el accuracy y el loss, al diseñar una red MLP, con dos capas oculta, con 15 y 10 neuronas respectivamente.*

En este caso podemos ver que parece que los valores de nuestras métricas no convergen y que podríamos estar sobreentrenando. Por ello, decidimos usar la estructura de red de una única capa oculta de 15 neuronas.

## Resultados

Como se ha demostrado en el apartado anterior, una red neuronal de dos capas ocultas de 15 y 10 neuronas es suficiente y apropiada para resolver este problema. Por ello, hemos comparado las diferentes funciones de activación en dicha estructura de red. Además, es importante tener en cuenta que, como se ha explicado anteriormente, la función de activación de la capa de salida es siempre la sigmoide binaria dado que tenemos un problema de clasificación binario.

Por otra parte, analizando los resultados de *accuracy* y *val_loss* obtenidos con el análisis previo, hemos determinado que el error objetivo no puede ser menor que 0,4 para este algoritmo. Es un error alto que se debe a la complejidad del problema. Si se quisiera realizar un buen aprendizaje de este conjunto de datos sería interesante probar con otros algoritmos más precisos dado que no se ha conseguido ejecutar nuestro algoritmo para un error menor.

En la siguiente tabla se muestran el número de épocas y el tiempo de ejecución del perceptrón multicapa para cada una de las funciones de activación.

| | Error= 0,4 | |
|---|---|---|
| | Épocas | Tiempo (s) |
| ReLu | 175 | 20,3 |
| Sigmoide binaria | 403 | 43,1 |
| Tanh | 1580 | 215,8 |
| Lineal | 49 | 12,3 |

*Tabla 1: Comparativa de número de épocas y tiempo que requiere cada función de activación para resolver el problema con un 40% de error, con el algoritmo de MLP.*

### ReLU (Rectified Linear Unit):

La función *ReLU* es conocida por su capacidad para abordar el problema de la desaparición del gradiente, permitiendo que las unidades de la red se activen de manera más eficiente. Los resultados indican que *ReLU* logra una convergencia rápida en un número relativamente bajo de épocas, señalando una adaptación eficaz del modelo. Además, el tiempo de ejecución es notablemente eficiente, sugiriendo un rendimiento óptimo en el aprendizaje dado.

### Sigmoide binaria:

La función sigmoide binaria, comúnmente utilizada en problemas de clasificación binaria, muestra un mayor número de épocas requeridas para la convergencia en comparación con *ReLU*. Esto era esperable dado que la función sigmoide binaria, al solo tener valores de 0 a 1, al tener un rango tan pequeño se dificulta la actualización de pesos.

### Tanh (Tangente Hiperbólica):

La función *tanh*, similar a la sigmoide pero con un rango de salida ampliado, muestra un rendimiento que requiere un mayor número de épocas en comparación con *ReLu*. Aunque *tanh* puede gestionar mejor los valores negativos, estos resultados sugieren que, en este caso, la convergencia es más lenta. Esto puede deberse al problema del gradiente desvaneciente que significa que durante el entrenamiento de redes neuronales profundas, los gradientes pueden volverse muy pequeños, dificultando la actualización efectiva de los pesos de la red. Por otra parte, puede deberse a la elección inicial de los pesos y de los hiperparámetros como la tasa de aprendizaje. Además, el tiempo de ejecución más prolongado indica que *tanh* puede ser menos eficiente para la tarea específica evaluada.

### Lineal:

La función lineal muestra un rendimiento sorprendentemente eficiente en términos de épocas y tiempo de ejecución. La simplicidad de la función puede estar contribuyendo a una rápida adaptación del modelo a la tarea dada. Esto puede deberse a que, a diferencia de las funciones de activación no lineales como *tanh* o *sigmoide*, la función lineal no sufre de problemas de saturación del gradiente. Esto significa que los gradientes no tienden a volverse extremadamente pequeños durante el retroceso y, por lo tanto, la actualización de los pesos puede ser más efectiva. Otro motivo es que nuestros datos, aunque no sean separables linealmente, sí presentan cierta linealidad.

## 2.2. Multi-layer Perceptron mediante tensorflow

### Selección de la red neuronal

A la hora de elegir es importante evitar el sobreajuste, para ello hay que seleccionar cuidadosamente el ancho y la profundidad de red, ya que la probabilidad de sufrir *overfitting* aumenta cuanto mayor sean. Utilizamos, como se ha mencionado, la función de activación *ReLu* para obtener el mejor modelo.

#### *Una única capa oculta con 10 neuronas*

Comenzamos probando una capa oculta. Vamos variando el número de neuronas hasta dar con el mejor modelo, en este caso de 10 neuronas. Con menos neuronas el modelo no terminaba de aprender y con más neuronas se llegaba a presentar cierto *overfitting*. A continuación, se muestran las gráficas de *accuracy* y *loss* del entrenamiento para la estructura con 10 neuronas en la capa oculta (dividido a su vez en un conjunto de entrenamiento y otro de validación). Vemos que los resultados son aceptables, aunque no excepcionales: el *accuracy* no es muy elevado y el *loss* es bastante elevado, alcanzando poco más de 0,33. Además, al igual que en el caso anterior, por mucho que se aumenten las épocas los resultados no mejoran.

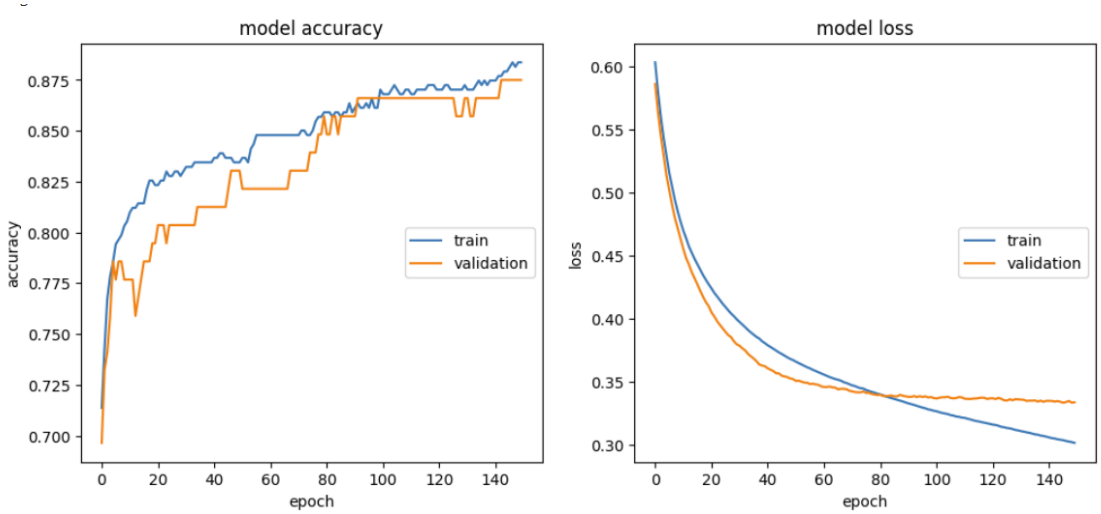



*Figura 21: Gráficas que muestran el accuracy y el loss, al diseñar una red con perceptrón multicapa, de una capa oculta, con 10 neuronas.*

#### *Dos capas ocultas con 7 y 5 neuronas*

Realizamos el mismo proceso con dos capas ocultas, llegando a que el mejor modelo para este caso requiere de una primera capa oculta con 7 neuronas y otra con 5.

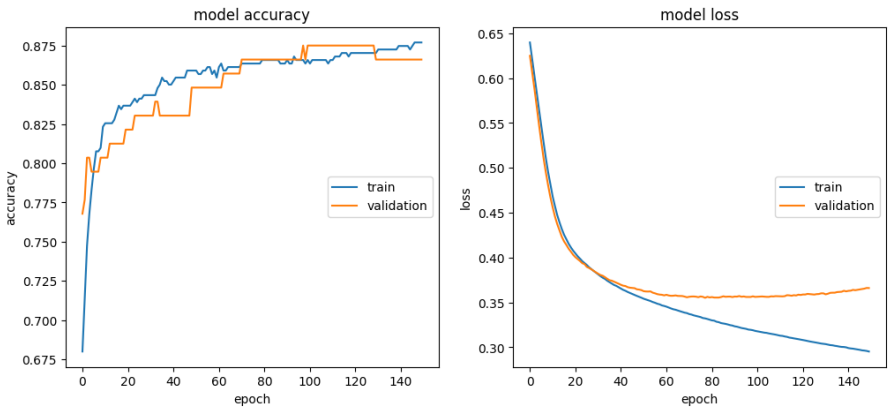



*Figura 22: Gráficas que muestran el accuracy y el loss, al diseñar una red con backpropagation, con dos capas ocultas, con con 7 y 5 neuronas respectivamente.*

### *Tres capas ocultas con 9, 3 y 2 neuronas*

Con tres capas ocultas los resultados en general son nefastos. En este caso se muestran las gráficas para una estructura con una primera capa oculta de 9 neuronas, una segunda 3 y una última de 2 neuronas.

Se aprecia un claro sobreentrenamiento en las gráficas. En la de *accuracy* el modelo de entrenamiento no se estabiliza, sino que continúa subiendo mientras que el conjunto de validación se mantiene. En la de *loss* se aprecia como el conjunto de entrenamiento disminuye mientras que el de validación tiende a aumentar.

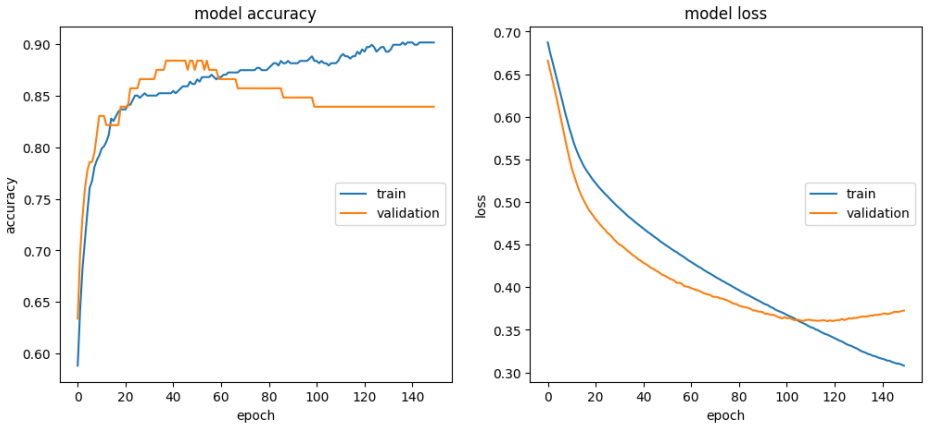



*Figura 23: Gráficas que muestran el accuracy y el loss, al diseñar una red con backpropagation, con tres capas ocultas, con 9, 3 y 2 neuronas respectivamente.*

## Resultados

Simplemente con la visualización de las gráficas podemos ver que la arquitectura de nuestra red es idónea con una única capa, ya que, al añadir más, sobreentrenamos a nuestro modelo.

Tras observar las tres estructuras, nos decantamos la única capa oculta de 10 neuronas, aplicando a esta las diferentes funciones de activación. Siguiendo la explicación anterior, al ser una clasificación binaria, la función de activación de la capa de salida será en todos los casos la sigmoide binaria.

A diferencia del otro perceptrón multicapa, en este caso se puede reducir el error, probando además de con un error de 0,4 otros valores algo inferiores, aunque no menores de 0,3. A pesar de ser menor que para el anterior modelo, este sigue presentando un error alto.

En la siguiente tabla se muestran el número de épocas y el tiempo de ejecución del perceptrón multicapa para cada una de las funciones de activación y errores de 0,4 y 0,35.

Durante la retropropagación, se utiliza el gradiente de la función de activación con respecto a su entrada para actualizar los pesos de la red y minimizar la función de pérdida.

| | Error= 0,4 | | Error= 0,35 | |
|---|---|---|---|---|
| | Épocas | Tiempo (s) | Épocas | Tiempo (s) |
| ReLu | 26 | 3,13 | 66 | 6,66 |
| Sigmoide binaria | 43 | 5,60 | 458 | 41,73 |
| Tanh | 23 | 3,37 | 44 | 5,19 |
| Lineal | 34 | 3,63 | 146 | 13,14 |

*Tabla 2: Comparativa de número de épocas y tiempo que requiere cada función de activación para resolver el problema con un 40 y 35% de error, con el algoritmo de MLP con Tensorflow.*

### ReLU (Rectified Linear Unit):

En este caso *ReLu* presenta buenos resultados ya que ayuda a mitigar el problema de desvanecimiento del gradiente. Se presenta una convergencia rápida en un número relativamente bajo de épocas, señalando una adaptación eficaz del modelo.

### Sigmoide binaria:

La función sigmoide binaria, comúnmente utilizada en problemas de clasificación binaria, muestra un mayor número de épocas requeridas para la convergencia en comparación con las demás, siendo coherente con lo esperado, pues se anulan ciertos pesos cuando la función toma valor 0, entorpeciendo el entrenamiento.

### Tanh (Tangente Hiperbólica):

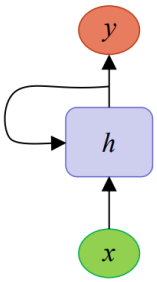
La función *tanh*, similar a la sigmoide, pero con un rango de salida ampliado, muestra un rendimiento que requiere un mayor número de épocas en comparación con *ReLU*. Aunque *tanh* puede gestionar mejor los valores negativos, estos resultados sugieren que, en este caso, la convergencia es más lenta. Además, el tiempo de ejecución más prolongado indica que *tanh* puede ser menos eficiente para la tarea específica evaluada.

### Lineal:

Como se ha mencionado, esta función de activación no es especialmente eficaz en problemas con patrones complejos y no lineales, como es el caso. Por tanto, es lógico que sea de las que peor desempeño presenta.

Se puede concluir que los resultados sugieren que la función de activación de la tangente hiperbólica destaca como la más eficiente en términos de convergencia y tiempo de ejecución para la tarea específica.

## 2.3. Long-Short Term Memory (LSTM)




Es un tipo de red neuronal artificial, de la familia de las RNN (*Recurrent Neural Networks*), que utiliza datos secuenciales o datos de series temporales.

Las RNN tienen conexiones recurrentes, en ciclo cerrado, desde una capa a sí misma o a otra capa anterior, de forma que la información fluye de forma recurrente en estos ciclos cerrados, en vez de propagarse solamente hacia adelante.

Por este diseño las redes recurrentes pueden procesar mejor entradas secuenciales, y la información puede persistir introduciendo bucles en el diagrama de la red, por lo que, básicamente, pueden «recordar» estados previos y utilizar esta información para decidir cuál será el siguiente. Mientras que las redes neuronales profundas tradicionales asumen que las entradas y salida son independientes entre sí, la salida de las redes neuronales recurrentes depende de los elementos anteriores dentro de la secuencia.

Las LSTM son un tipo especial de redes recurrentes. Lo que caracteriza a este tipo modelo, es que, mientras las redes recurrentes estándar pueden modelar dependencias a corto plazo, las LSTM pueden aprender dependencias largas, por lo que se podría decir que tienen una "memoria" a más largo plazo.

### Selección de la red neuronal

#### *Una capa oculta de 2 neuronas*

Utilizamos la función *tanh* como función de referencia, para estimar el número de capas y neuronas que se van a utilizar en cada una de las funciones de activación.

En este caso, tras varias pruebas, se encontró que los mejores resultados se obtenían utilizando 2 neuronas y una sola capa oculta. Los resultados son similares a los otros modelos, obteniendo un error (*loss*) del 43.4% y un *accuracy* del 69.41% en el conjunto de entrenamiento. En el conjunto de entrenamiento, obtenemos 49.73% de *loss* y 69.36% de *accuracy*.

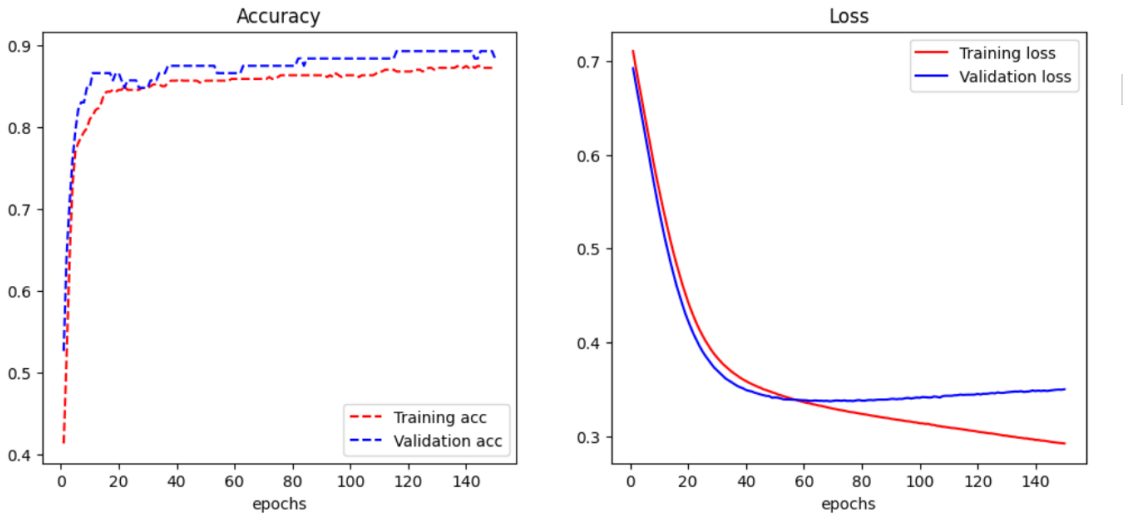



*Figura 24: Gráficas que muestran el accuracy y el loss, al diseñar una red LSTM, de una capa oculta, con 2 neuronas.*

No sobreentrena. Quizás hubiera sido mejor terminar unas pocas épocas antes.

### *Dos capa ocultas, una neurona en cada capas*

Probamos con 2 capas ocultas. Tras probar varias versiones, los mejores resultados se obtuvieron con el mínimo de neuronas por capa, y aún así se puede distinguir un claro sobreentrenamiento:

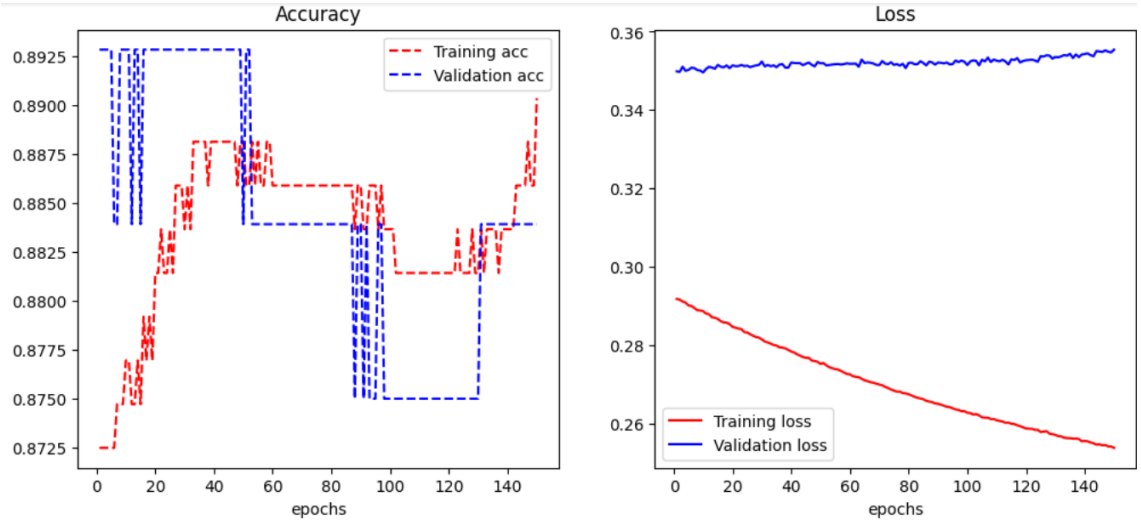



*Figura 24: Gráficas que muestran el accuracy y el loss, al diseñar una red LSTM, de dos capas ocultas, con una neurona en cada capa.*

Por tanto, claramente la arquitectura ideal para trabajar en este caso con el modelo LSTM es una capa oculta, y dos neuronas en dicha capa. La capa de salida, como siempre, se compondrá de una neurona cuya función de activación es la sigmoide binaria.

Ahora, sabiendo la estructura neuronal, se comparan las épocas y tiempos de las diferentes funciones de activación.

## Resultados

Los resultados son muy acordes a los que se esperaba teóricamente. Este es de los pocos modelos en los que se espera que la función *ReLu* funcione peor, y sin embargo la tangente hiperbólica y la sigmoide son funciones que optimizan dicho modelo.

Para continuar con la homogeneidad del estudio, se trató primero de llegar al error del 40%, y el modelo funcionó rápidamente, y en general, con pocas épocas, se logró que éste aprendiese. Cabe recalcar que es un modelo que internamente está compuesto por bucles que almacenan información y juegan con esta capacidad de “memoria” para llegar al resultado, y por ello, aún con pocas neuronas se consiguen buenos resultados.

Al haber obtenido resultados tan rápido, se intentó encontrar errores más bajos. Hasta un 0.37 se hizo de manera ligera; cabe destacar que la función con peores resultados es precisamente *ReLu*. En cambio, 0,36 parece ser el menor error que este modelo puede abordar para resolver este problema de clasificación del dataset dado (ver tabla 3).

| | Error = 0,4 | | Error = 0,37 | | Error = 0,36 | |
|---|---|---|---|---|---|---|
| | Épocas | Tiempo (s) | Épocas | Tiempo (s) | Épocas | Tiempo (s) |
| ReLu | 24 | 4,90 | 421 | 55,80 | 10000 | 1524,20 |
| Sigmoide binaria | 131 | 16,40 | 378 | 43,50 | 311 | 35,20 |
| Tanh | 44 | 0,30 | 44 | 0,80 | 55 | 0,94 |
| Lineal | 19 | 5,90 | 55 | 8,50 | 44 | 10,30 |

*Tabla 3: Comparativa de número de épocas y tiempo que requiere cada función de activación para resolver el problema con un 40, 37 y 36% de error, con el algoritmo de LSTM.*

### *Tanh* (Tangente Hiperbólica):

Teóricamente, y ahora en la práctica, queda demostrado que, para este tipo de redes neuronales, funciona óptimamente. Por una parte, dado que el rango de salida queda centrado en cero le resulta beneficioso para el aprendizaje al modelo LSTM, siendo capaz de gestionar la información en la memoria a largo plazo dentro de unos valores concretos. Especialmente funciona bien porque en la retropropagación, se mitiga el desvanecimiento del gradiente, al acortar el rango entre 0 y 1 (por la derivada de la *tanh*), y mitiga la saturación, manteniendo los gradientes más fuertes en rangos más amplios. De hecho, cabe recordar que, al crear una estructura con dos capas, el modelo sobreentrenaba.

La función *tanh* es adecuada para el manejo de la memoria a largo plazo en las LSTM. Al tener un rango amplio y una capacidad de representar tanto valores positivos como negativos, la *tanh* permite a las celdas LSTM almacenar y actualizar información a lo largo del tiempo.

### Sigmoide binaria:

Con un funcionamiento similar a la tangente hiperbólica, la sigmoide produce un rango de salidas entre 0 y 1, lo que es útil para modelar probabilidades o para representar estados de activación de celdas LSTM; pudiendo decidir qué información retener (valores cercanos a 1) y qué información desechar (*forget gate*, valores cercanos a 0) en su estado interno.

Además, al igual que la función *tanh*, la derivada, con valores en el rango (0, 1), hace que a la hora de retropropagar, se mitigue el problema del desvanecimiento del gradiente.

En comparación con la *tanh*, satura a valores extremos, y por ello tiene algo de peor rendimiento, pero igualmente, al bajar el error, *tanh* y sigmoide fueron las dos únicas que no tuvieron problemas al encontrar dicho error.

### *ReLU* (Rectified Linear Unit):

Teóricamente, la *ReLu* funciona bien para la mayoría de modelos, pero no para las redes RNN, y ha quedado demostrado. Las LSTM están diseñadas para manejar dependencias a largo plazo en secuencias temporales, y sin embargo, la *ReLU* puede tener problemas con el desvanecimiento del gradiente, ya que, durante la retropropagación, si la derivada de la ReLU es cero, no se propagará ningún gradiente, lo que dificulta la actualización de los pesos. Pero funciona especialmente mal por el problema de la inactivación de neuronas; partiendo de que de por sí este modelo tenía pocas neuronas, si además alguna ha sido inactivada, no ha contribuido al

aprendizaje, y, por tanto, ha habido una pérdida de retención de información a largo plazo. Además, la *ReLU* puede ser menos adecuada para reinicializar el estado interno de la memoria a largo plazo en comparación con funciones de activación como la tangente hiperbólica, que tiene un rango simétrico alrededor de cero.

### Lineal:

Tal y como era de esperar, el rendimiento de esta función de activación es bueno para errores fáciles de llegar, pero al bajar, supone problemas para hallarlos. Por una parte, la función lineal no introduce no linealidades en el modelo, y dado que las LSTM están diseñadas para manejar dependencias a largo plazo y aprender patrones complejos en secuencias temporales, la linealidad puede ser insuficiente para capturar estas relaciones no lineales. Por el otro lado, al no tener efecto en el desvanecimiento del gradiente, en arquitecturas con varias capas, se irá perdiendo información poco a poco. Al tratarse de una red que depende de la memoria a largo plazo, una función que impide discriminar datos y es invariante a transformaciones afines hace que la efectividad de esta sea pequeña.

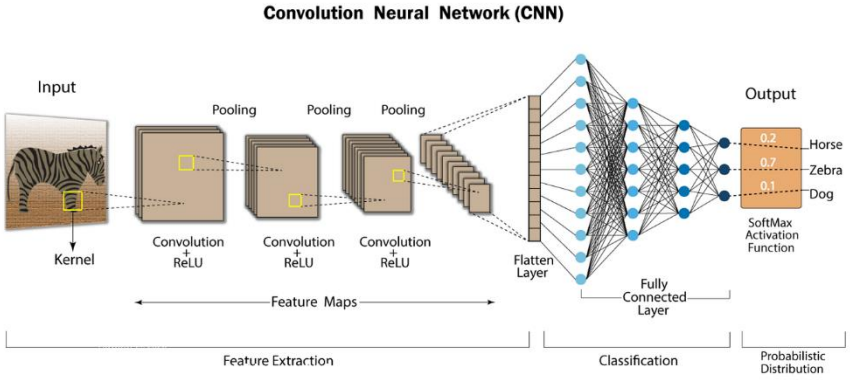
## 2.4. CNN

Una Red Neural Convolucional (CNN, por sus siglas en inglés, *Convolutional Neural Network*) es un tipo de arquitectura de red neuronal profunda diseñada específicamente para procesar datos estructurados en mallas, como imágenes.

Se ha tratado de aplicar a nuestro dataset.

Al estar diseñado para imágenes normalmente consta de una tercera dimensión que se identifica con los canales *rgb*, realizamos un cambio de forma (*reshape*) para ajustar los datos al formato de entrada esperado por una red neuronal convolucional (CNN).

No obstante, es un modelo poco adecuado, siendo mejor trabajar con RNN o FNN.




### Selección de la red neuronal

#### *Una capa oculta de 2 neuronas*

Además de las correspondientes capas convolucionales y de agrupación introducimos una capa oculta de 2 neuronas.

| Layer (type) | Output Shape | Param # |
|---|---|---|
| conv1d_12 (Conv1D) | (None, 9, 32) | 128 |
| max_pooling1d_12 (MaxPooling1D) | (None, 4, 32) | 0 |
| flatten_12 (Flatten) | (None, 128) | 0 |
| dense_30 (Dense) | (None, 2) | 258 |
| dense_31 (Dense) | (None, 1) | 3 |

Total params: 389 (1.52 KB)
Trainable params: 389 (1.52 KB)
Non-trainable params: 0 (0.00 Byte)

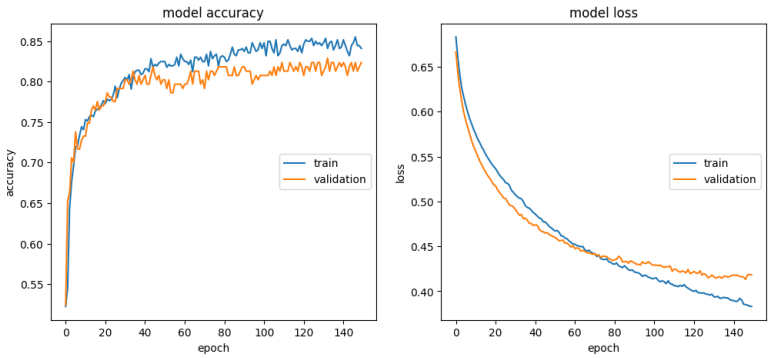



*Figura 25: Gráficas que muestran el accuracy y el loss, al diseñar una red CNN, de una capa oculta, con 4 neuronas.*

### *Dos capas ocultas, una de 4 y otra de 2 neuronas*

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv1d_13 (Conv1D)          (None, 9, 32)             128

 max_pooling1d_13 (MaxPooli  (None, 4, 32)             0
 ng1D)

 flatten_13 (Flatten)        (None, 128)               0

 dense_32 (Dense)            (None, 4)                 516

 dense_33 (Dense)            (None, 2)                 10

 dense_34 (Dense)            (None, 1)                 3

=================================================================
Total params: 657 (2.57 KB)
Trainable params: 657 (2.57 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

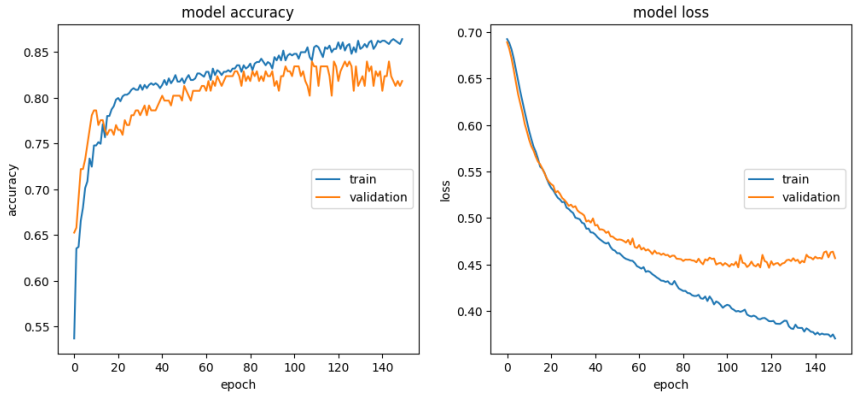



*Figura 26: Gráficas que muestran el accuracy y el loss, al diseñar una red CNN, de dos capa oculta, con 4 y 2 neuronas respectivamente.*

Ambos modelos son poco adecuados, pero con una capa oculta de dos neuronas tanto el conjunto de entrenamiento como el de test parecen estabilizarse. Además, en el modelo con dos capas ocultas podemos ver un poco de *overfitting*.

A diferencia de los otros modelos, al no estar pensado para *databases* de este estilo, sino para mallas, con imágenes, el error que se alcanza es bastante peor que en los casos anteriores, por lo que nos hemo visto obligadas a aumentar el error a 0,448.

## Resultados

Los resultados, como era de esperar, son mediocres, pero concuerdan con lo esperado.

| | Error = 0,448 | |
|---|---|---|
| | Épocas | Tiempo (s) |
| ReLu | 40 | 4,1 |
| Sigmoide binaria | 482 | 44,8 |
| Tanh | 194 | 22,8 |
| Lineal | 107 | 60,8 |

*Tabla 4: Comparativa de número de épocas y tiempo que requiere cada función de activación para resolver el problema con un 40% de error, con el algoritmo de CNN.*

### *ReLU* (Rectified Linear Unit):

Destaca la función *ReLu* al aportar no linealidad, favoreciendo el aprendizaje en redes complejas, como es el caso. Debido a su simplicidad, eficiencia y a los buenos resultados que muestra en el descenso por gradiente de la retropropagación *ReLu*, es una opción sólida para la mayoría de las capas ocultas de CNN.

Al ser una función no saturante y simple, suele permitir entrenamientos más rápidos. En este caso, la red convergió en solo 40 épocas y requirió un tiempo relativamente corto de 4.14 segundos.

### *Tanh* (Tangente Hiperbólica):

La función *tanh*, similar a la sigmoide pero con salidas en el rango (-1, 1), también puede sufrir de saturación, especialmente para entradas grandes o pequeñas. Las 194 épocas y el tiempo de 22.79 segundos sugieren una convergencia más rápida en comparación con la sigmoide binaria, pero aún más lenta que *ReLU*.

### Sigmoide binaria:

Con un funcionamiento similar a la tangente hiperbólica, la sigmoide produce un rango de salidas entre 0 y 1. La función sigmoide binaria, comúnmente utilizada en la capa de salida para problemas de clasificación binaria, es más propensa a problemas de saturación, lo que puede ralentizar el entrenamiento. El hecho de que requiera 482 épocas y 44.75 segundos sugiere que la convergencia es más lenta.

### Lineal:

La función lineal (también conocida como identidad) puede ser más lenta en converger debido a su naturaleza lineal. Puede no ser la opción óptima para capas ocultas debido a la falta de no linealidad.

# 3. CONCLUSIONES

Tras aplicar varios algoritmos a este problema de clasificación, nos encontramos con que es un problema muy difícil de separar, y los resultados no superan por mucho el 40% de error.

El algoritmo que mejor funcionó fue el perceptrón multi-capa, de la mano de *Tensorflow*, que lograron resolver el problema con un 35% de error. Concretamente en este caso, la función de activación que mejor funcionó fue la tangente hiperbólica, seguida por poco de *ReLu*. Se trata de funciones que pueden resolver problemas no lineales, y es cierto que *ReLu* por lo general es rápida, pero a largo plazo puede dar problemas. Además, las derivadas de estas funciones son derivadas simples pero efectivas (la *ReLu* mitiga el problema del desvanecimiento del gradiente, y la *tanh* aporta precisión entre capas).

Interesantemente, tanto para el algoritmo LSTM, como el MLP básico, la función que mejor resolvió el problema fue la linear (para el 40% de error). Entendemos que el motivo principal es que, aunque sea un problema de clasificación no lineal, la estructura de la red hizo que se lograra a lo largo de varias dimensiones, separar bien las variables, y por ello la función lineal funcionó bien. Si se observa la tabla 6, se verá que a pesar de que en este caso la lineal resolviese el problema con pocas épocas, sigue siendo una función muy simple y poco eficiente (tarda mucho en ser ejecutada por época). Es por ello, que, en el caso de la LSTM, es interesante observar que, aunque se resuelva con la *tanh* con más épocas, es muchísimo más rápida que con la lineal. Además, el LSTM sí logró bajar el error hasta el 36%, pero el MLP básico no fue capaz.

Por último, como era de esperar, el problema no fue resuelto con buenos resultados en el caso de la CNN, debido a que es un algoritmo que normalmente se aplica a imágenes.

| | Perceptrón 1 ε=0,4 | | Perceptrón 2 ε=0,4 | | LSTM ε=0,4 | | CNN ε=0,448 | |
|---|---|---|---|---|---|---|---|---|
| | Épocas | Tiempo (s) | Épocas | Tiempo (s) | Épocas | Tiempo (s) | Épocas | Tiempo (s) |
| ReLu | 175 | 20,3 | 26 | 3,13 | 24 | 4,90 | 40 | 4,1 |
| Sigmoide binaria | 403 | 43,1 | 43 | 5,60 | 131 | 16,40 | 482 | 44,8 |
| Tanh | 1580 | 215,8 | 23 | 3,37 | 44 | 0,30 | 194 | 22,8 |
| Lineal | 49 | 12,3 | 34 | 3,63 | 19 | 5,90 | 107 | 60,8 |

*Tabla 5: Comparativa de número de épocas y tiempo que requiere cada algoritmo para llegar al 40% de error.*

| | Perceptrón 1 | Perceptrón 2 | LSTM | CNN |
|---|---|---|---|---|
| ReLu | 0,12 | 0,12 | 0,20 | 0,10 |
| Sigmoide binaria | 0,11 | 0,13 | 0,13 | 0,09 |
| Tanh | 0,14 | 0,15 | 0,01 | 0,12 |
| Lineal | 0,25 | 0,11 | 0,31 | 0,57 |

*Tabla 6: Comparativa tiempos que requiere cada función de activación por época.*

En esta última tabla podemos ver que la función lineal es la menos eficiente computacionalmente hablando para casi todos los algoritmos y que la eficiencia del resto de funciones de activación es mejor y parecido para todas las funciones de activación y algortimos.