Asignatura:

Inteligencia Artificial II

Título del documento:

Perceptrones y MLPs

Práctica 2 - Laboratorios 1, 2 y 3

Presentado por:

David Anastasio Enrique Alba

Luis Eduardo de Santiago Martínez

Cristina Espejo Fernández

Mario Martínez Vitutia

Carmen Moreno Udaondo

Francisco Javier Zaballa Gutiérrez

Índice de contenidos

Índice de figuras

Índice de tablas

Índice de ecuaciones

Resumen

En esta práctica exploramos el funcionamiento y las limitaciones del perceptrón simple, así como la capacidad de los perceptrones multicapa (MLP) para resolver problemas no linealmente separables como la función lógica XOR. Comenzamos implementando un perceptrón en PyTorch y verificamos su efectividad para resolver funciones AND y OR, observando sus limitaciones al abordar XOR. A continuación, desarrollamos un MLP configurable, evaluando el impacto de distintos hiperparámetros (número de neuronas ocultas, *learning rate* y funciones de activación) tanto ReLU como sigmoide.

Los resultados nos mostraron que la arquitectura del MLP con 8 neuronas ocultas y función ReLU era capaz de aprender correctamente la función XOR, alcanzando errores mínimos y generando fronteras de decisión no lineales. Sin embargo, también detectamos limitaciones de ReLU, como “la muerte de neuronas”. Al comparar con la función sigmoide, observamos que también alcanzaba buenos resultados con menos neuronas y menor número de iteraciones, aunque con fronteras más suaves y simétricas. Finalmente, aplicamos el MLP a un problema más complejo: la predicción de muertes en la serie Juego de Tronos. El modelo aprendió patrones útiles, aunque detectamos posibles problemas de overfitting derivados del tamaño reducido del dataset.

1 Introducción

1.1 Contexto general: Perceptrón y perceptrón multicapa

En el campo del aprendizaje automático supervisado, los **perceptrones** constituyen uno de los modelos más simples y fundamentales para la clasificación binaria. El perceptrón fue propuesto por Frank Rosenblatt en 1958 como un **modelo matemático** que simula el comportamiento de una **neurona** biológica [1]. Su funcionamiento se basa en recibir múltiples entradas, ponderarlas mediante pesos, sumar esos valores y aplicar un umbral (*bias*) a través de una función de activación. De esta forma, produce una salida binaria que clasifica los datos en una de dos clases.

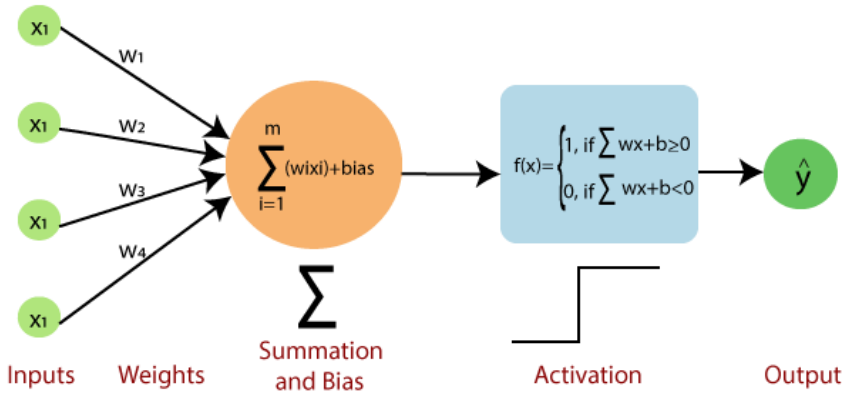

Ilustración 1. Estructura de un perceptrón simple.

El **aprendizaje** del perceptrón consiste en ajustar los pesos y el *bias* mediante un proceso iterativo, en el que se comparan las predicciones con las salidas esperadas y se corrigen los errores [1]. El algoritmo de aprendizaje del perceptrón **converge** siempre que los datos sean linealmente separables, es decir, cuando existe un hiperplano que divide perfectamente ambas clases [2].

Sin embargo, los perceptrones simples tienen limitaciones importantes. En concreto, no pueden resolver problemas **no linealmente separables**, como el clásico ejemplo de la función lógica XOR. Para superar estas limitaciones, se introdujeron las redes neuronales multicapa, también conocidas como **perceptrones multicapa (MLP)** [3]**.**

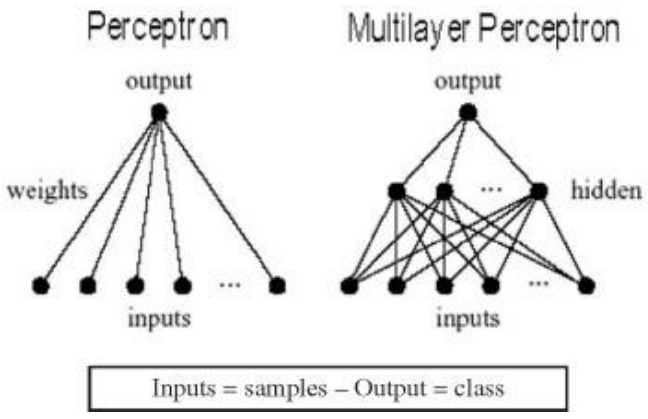

Ilustración 2. Perceptrón de una capa y perceptrón de múltiples capas.

Un **MLP** es una red neuronal artificial de tipo *feedforward*, es decir, la información fluye unidireccionalmente desde la entrada hasta la salida. Estos modelos están compuestos por al menos tres capas: una **capa de entrada**, una o más **capas ocultas** y una **capa de salida** [3]. Cada neurona aplica una función de activación no lineal, como la sigmoide, la ReLU o la tangente hiperbólica, lo que le da a los MLPs la capacidad de modelar relaciones complejas y no lineales entre los datos [4].

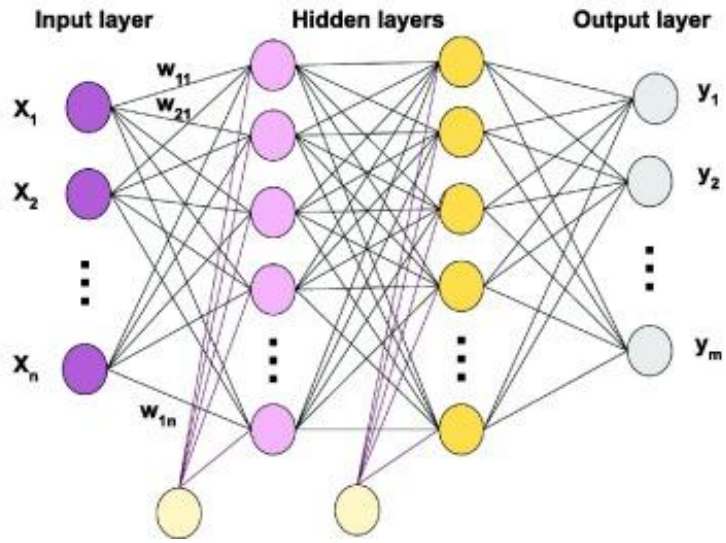

Ilustración 3. Topología de un MLP.

El **entrenamiento** de un MLP se lleva a cabo mediante técnicas más avanzadas que en el perceptrón clásico, como el **algoritmo de retropropagación del error** *(backpropagation)* o métodos de optimización como **Adam**, que ajustan los pesos a partir de los gradientes de la función de pérdida calculados sobre mini-lotes del conjunto de entrenamiento. Gracias a su estructura y flexibilidad, los MLP pueden aproximar cualquier función continua (teorema de aproximación universal), lo que los convierte en herramientas muy potentes para tareas de clasificación y regresión [5].

A continuación, resumimos las diferencias entre ambos modelos:

Aspecto	**Perceptrón**	**Perceptrón Multicapa**
Capas	Una sola capa (entrada y salida).	Al menos 3 capas (entrada, oculta y salida).
Problema tipo	Linealmente separable (clasificación binaria).	No linealmente separable, clasificación y regresión complejas.
Entrenamiento	Regla de aprendizaje simple.	*Backpropagation* y otros métodos más complejos (requiere funciones no lineales).
Limitaciones	Esta limitado a datos lineales (no resuelve el XOR).	Puede sobreajustar si no se controla la complejidad del modelo.

Tabla 1. Diferencias entre perceptrón y MLP.

1.2 Objetivo de la práctica

El objetivo de esta práctica es explorar e implementar modelos neuronales supervisados en PyTorch, comprendiendo las bases teóricas y prácticas del aprendizaje en perceptrones y redes neuronales multicapa (MLP). Para ellos, experimentamos con distintas arquitecturas, funciones de activación, *learning rates* y criterios de parada, utilizando funciones lógicas como AND y XOR como problemas base, así como un conjunto de datos más complejos sobre muertas en la serie *Juego de Tronos*.

La práctica se estructura en tres bloques o laboratorios:

- **Laboratorio 1:** Implementación y entrenamiento de un perceptrón simple para resolver las funciones lógicas AND y XOR, analizando la convergencia y sus limitaciones.

- **Laboratorio 2:** Diseño e implementación de un MLP para resolver el problema XOR, comparando distintas configuraciones de activación, *learning rates* y número de neuronas en la capa oculta.

- **Laboratorio 3:** Aplicación de un MLP para un problema real de clasificación, usando como dataset la predicción de muertes en la serie de *Juego de Tronos.*

Por último, presentamos unas conclusiones sobre lo que hemos aprendido y una sección de bibliografía con los recursos que hemos utilizado durante el desarrollo de la práctica.

2 Laboratorio 1 – Perceptrón AND y XOR

En este primer laboratorio implementamos un perceptrón simple para resolver funciones lógicas básicas. El objetivo era entrenar y evaluar un modelo sobre las compuertas lógicas AND, OR y XOR, y observar sus resultados, especialmente en lo relativo a la capacidad de aprendizaje y convergencia del modelo.

2.1 Implementación del perceptrón

El perceptrón que hemos implementado es un modelo clásico, propuesto por Rosenblatt en 1958, donde hay una sola neurona que recibe dos entradas, cada una con su correspondiente peso, y un sesgo (*bias*). La salida del modelo la obtenemos al aplicar una función de activación escalón sobre a combinación lineal de las entradas y sus pesos, más el sesgo. La activación se define como:

$$Y = \begin{cases} 1 \quad si \quad \sum_{i=1}^{n} w_i x_i + b \geq \theta \\ 0 \quad si \quad \sum_{i=1}^{n} w_i x_i + b < \theta \end{cases}$$

Ecuación 1. Función de activación.

donde w_i representa los pesos, x_i las entradas, b el bias, y θ el umbral de activación.

El proceso de entrenamiento del perceptrón se basa en una regla de aprendizaje supervisado que ajusta los pesos y el bias en función del error cometido en cada iteración. Dado un conjunto de datos etiquetado como $\{(x^{(i)}, y^{(i)})\}$, calculamos la salida del perceptrón y_r, y comparamos con la salida esperada y_d. Si existe un error, actualizamos los pesos con la siguiente regla:

$$w := w_i + \eta \cdot error \cdot x_i$$

$$b := b_i + \eta \cdot error$$

Ecuación 2. Regla de aprendizaje (perceptrón simple).

donde η es el *learning rate* y $error = y_d - y_r$. Este proceso lo repetimos hasta que el error promedio en una época/iteración sea cero (convergencia), o hasta alcanzar un número máximo de iteraciones. Este método de corrección nos garantiza la convergencia del algoritmo si el conjunto de datos es linealmente separable.

Para desarrollar la implementación, hemos definido una clase `Perceptron` que contiene todos los componentes necesarios para su entrenamiento, prueba y análisis. A continuación, explicamos en detalle las funciones principales que forman parte del modelo.

En primer lugar, la función `funcion_activacion(x)` es responsable de aplicar una función de activación escalón, que transforma la entrada neta del perceptrón en una salida binaria. En concreto, si el valor de la entrada neta super el umbral que hemos definido, la salida será 1 (neurona activada); en caso contrario, la salida será 0 (neurona desactivada). Este método refleja el comportamiento básico de una neurona artificial y permite tomar decisiones binarias en función de los valores ponderados de la entrada.

```
1. def funcion_activacion(self, x) -> float:
2.     return 1 if x >= self.umbral else 0
```

Ilustración 4. Implementación de la función de activación.

La función `calcular_salida(entradas)` es la que se encarga de procesar una muestra concreta y determinar la salida del perceptrón. Para realizarlo, calcula el producto escalar entre las entradas y sus respectivos pesos, suma el valor del sesgo (*bias*) y aplica la función de activación que hemos mencionado anteriormente. Esta función es fundamental tanto en la fase de entrenamiento como en la de prueba y evaluación, ya que representa la forma en que nuestro perceptrón toma decisiones.

```
1. def calcular_salida(self, entradas) -> float:
2.     suma_ponderada = np.dot(entradas, self.pesos) + self.bias
3.     return self.funcion_activacion(suma_ponderada)
```

Ilustración 5. Implementación de la función de cálculo de salida.

El método `entrenar(X, y)` constituye el núcleo del aprendizaje de nuestro modelo. Recibe como entrada el conjunto de entrenamiento (entradas `X` y salidas `y`) y ajusta los pesos y el bias del modelo mediante un algoritmo iterativo. En cada iteración, recorremos todas las muestras del conjunto, calculamos la salida del modelo, comparamos esta salida con la esperada, y en caso de haber error, actualizamos los pesos según la regla de aprendizaje. Detenemos el entrenamiento cuando el modelo logra clasificar correctamente todas las muestras o cuando alcana el número máximo de iteraciones. Durante este proceso, almacenamos el error medio de cada época, así como los detalles del entrenamiento paso a paso (pesos, bias, entradas, errores, etc).

```
1. # Actualizamos los pesos y el bias si hay error
2.     if error != 0:
3.         self.pesos += self.learning_rate * error * X[muestra]
4.         self.bias += self.learning_rate * error
```

Ilustración 6. Implementación de la regla de aprendizaje.

Una vez que hemos entrenado el modelo, la función `probar_mostrar(X, y)` nos permite comprobar el rendimiento del modelo. Para cada muestra del conjunto de entrada, mostramos por pantalla la predicción que ha realizado el perceptrón junto con la salida real. Esta tabla de resultados nos resulta muy útil para verificar si el modelo está clasificando correctamente los datos.

```
 1. def probar_y_mostrar(self, X, y) -> None:
 2.         print("\nResultados del perceptrón:")
 3.         print("---------------------------------")
 4.         print("| X1 | X2 | Target | Predicho  |")
 5.         print("---------------------------------")
 6.
 7.         for iteracion in range(len(X)):
 8.             prediction = self.calcular_salida(X[iteracion])
 9.             print(f"| {X[iteracion][0]}  | {X[iteracion][1]}  | {y[iteracion]}      |
{prediction}         |")
10.
11.         print("---------------------------------")
12.
```

Ilustración 7. Implementación de la prueba y visualización del entrenamiento.

El método de `calcular_ecuacion_hiperplano()` nos permite obtener la expresión matemática del hiperplano de decisión aprendido por nuestro modelo. Este hiperplano es el que separa las clases en el espacio de características y su ecuación deriva de la condición de activación:

$$w_1 \cdot x_i + w_2 \cdot x_2 + b - \theta = 0$$

Ecuación 3. Ecuación del hiperplano.

Para visualizar el hiperplano que hemos calculado con el método anterior, utilizamos la función `mostrar_frontera_decision(X, y)`, que construye una malla bidimensional con todos los posibles valores de entrada y evalúa la salida del perceptrón en cada uno de ellos. De esta forma, generamos un gráfico de colores que muestra cómo el modelo divide el espacio en zonas correspondientes a cada clase. Sobre esta visualización, representamos también los datos reales y el hiperplano de decisión en forma de línea discontinua, lo que permite comprobar visualmente si la separación aprendida es coherente con los datos.

Con el objetivo de facilitar el análisis del proceso de aprendizaje de nuestro perceptrón, hemos implementado la función `mostrar_errores()`, el cuál genera una gráfica que representa la evolución del error medio a lo largo de las iteraciones. Esta visualización nos permite identificar de forma intuitiva si el modelo ha conseguido converger y en cuántas épocas lo ha hecho.

Por otro lado, con la función `mostrar_detalles_entrenamiento()` proporcionamos un análisis más detallado del entrenamiento. Con este método visualizamos la evolución temporal de los parámetros del modelo (los pesos `w1` y `w2`, y el bias), así como el error medio por iteración. Para ello, generamos cuatro gráficas que permiten observar como se modifican los pesos y el sesgo a medida que el modelo aprende.

Finalmente, con la función `generar_tabla_entrenamiento()` construimos un `DataFrame` que resume todos los pasos realizados durante el entrenamiento. Para cada iteración y cada muestra, registramos las entradas, los pesos antes y después de la actualización, el valor predicho, el valor esperado y el error cometido. Esta tabla es extremadamente útil porque nos permite reconstruir y entender cada paso que ha dado el modelo hasta llegar a su solución final.

2.2 Función lógica AND

La puerta lógica AND es una de las operaciones lógicas más simples y conocidas. Su funcionamiento se basa en devolver una salida positiva únicamente cuando todas las entradas son 1. En cualquier otro caso, la salida es 0. Es decir, representa el concepto lógico de *"y"*: solo obtenemos un resultado verdadero si todas las condiciones lo son [7].

La tabla de verdad correspondiente a la función AND con dos entradas es la siguiente, Donde `X1` y `X2` representan entradas binarias, e `Y` es la salida:

X1	X2	Y
0	0	0
0	1	0
1	0	0
1	1	1

Tabla 2. Puerta lógica AND.

Desde un punto de vista geométrico, este problema puede representar en un plano de dos dimensiones, donde cada par `(X1, X2)` es un punto. En este espacio, la clase `Y = 1` se encuentra en la esquina superior derecha del plano (el punto `(1, 1)`), mientras que el resto de las combinaciones pertenecen a la clase `Y = 0`. Esta configuración permite que un perceptrón simple sea capaz de encontrar una línea recta (un hiperplano en dos dimensiones) que separe correctamente ambas clases. Por lo tanto, el problema AND es linealmente separable, lo que lo hace ideal para ser resuelto por nuestro modelo con una única neurona.

Evaluamos el modelo de perceptrón utilizando distintas combinaciones de *learning rates* y umbrales de activación. En concreto, probamos con los *learning rates* $\eta \in \{0.01, 0.1, 0.5, 0.9\}$ y los valores umbral $\theta \in \{0.0,\ 0.5, 1.0\}$. En todos los casos, el modelo fue capaz de converger con éxito, clasificando correctamente los cuatro patrones definidos por la tabla de verdad de la función AND.

La mejor configuración que obtuvimos fue un *learning rate* $\eta = 0.5$ y un umbral $\theta = 1.0$, logrando la convergencia en tan solo una iteración, lo cual demuestra que es eficiente. Durante este proceso, ajustamos los pesos y el sesgo hasta encontrar una configuración que permitiera separar el único punto positivo `(1, 1)` del resto del plano.

Iteración	Entradas		Pesos iniciales		Yr	Yd	Error	Pesos finales	
	X1	X2	W1	W2				W1	W2
1	0	0	0.3483	0.6296	0	0	0	0.3483	0.6296
1	0	1	0.3483	0.6296	1	0	-1	0.3483	0.1296
1	1	0	0.3483	0.1296	0	0	0	0.3483	0.1296
1	1	1	0.3483	0.1296	0	1	1	0.8483	0.6296
2	0	0	0.8483	0.6296	0	0	0	0.8483	0.6296
2	0	1	0.8483	0.6296	1	0	-1	0.8483	0.1296
2	1	0	0.8483	0.1296	0	0	0	0.8483	0.1296
2	1	1	0.8483	0.1296	0	1	1	1.3483	0.6296
3	0	0	1.3483	0.6296	0	0	0	1.3483	0.6296
3	0	1	1.3483	0.6296	1	0	-1	1.3483	0.1296
3	1	0	1.3483	0.1296	1	0	-1	0.8483	0.1296
3	1	1	0.8483	0.1296	0	1	1	1.3483	0.6296
4	0	0	1.3483	0.6296	0	0	0	1.3483	0.6296
4	0	1	1.3483	0.6296	0	0	0	1.3483	0.6296
4	1	0	1.3483	0.6296	1	0	-1	0.8483	0.6296
4	1	1	0.8483	0.6296	0	1	1	1.3483	1.1296
5	0	0	1.3483	1.1296	0	0	0	1.3483	1.1296
5	0	1	1.3483	1.1296	1	0	-1	1.3483	0.6296
5	1	0	1.3483	0.6296	0	0	0	1.3483	0.6296
5	1	1	1.3483	0.6296	1	1	0	1.3483	0.6296
6	0	0	1.3483	0.6296	0	0	0	1.3483	0.6296
6	0	1	1.3483	0.6296	0	0	0	1.3483	0.6296
6	1	0	1.3483	0.6296	0	0	0	1.3483	0.6296
6	1	1	1.3483	0.6296	1	1	0	1.3483	0.6296

Tabla 3. Tabla detalla de los resultados de entrenamiento (Puerta AND).

El hiperplano generado por el perceptrón en este caso, lo podemos expresar de la siguiente forma:

$$1.3483 \cdot x_1 + 0.6296 \cdot x_2 - 1.5612 = 0$$

Ecuación 4. Ecuación del hiperplano (Puerta AND).

Esta ecuación define una línea recta que divide el espacio de entrada en dos regiones, clasificando correctamente los datos: todos los puntos distintos de `(1, 1)` quedan en una región (etiquetada con salida 0), mientras que el punto `(1, 1)` queda en la región opuesta (salida 1).

Durante el entrenamiento, registramos los errores medios por iteración, así como la evolución de los pesos y del *bias*. En la siguiente figura, podemos observar cómo el error medio disminuye progresivamente hasta alcanzar el valor 0 en la sexta iteración, momento en el cual el modelo ha aprendido correctamente la función AND:

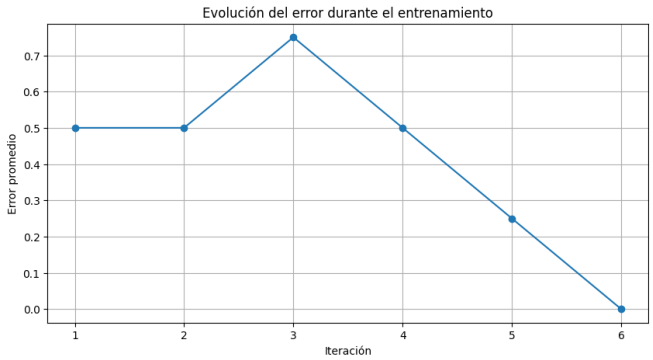

Ilustración 8. Evolución del error durante el entrenamiento (Puerta AND).

En la figura 9 mostramos con mayor detalle cómo evolucionan los pesos `w1`, `w2` y el *bias* a lo largo del entrenamiento. Podemos observar cómo, en cada paso de actualización, el modelo ajusta sus parámetros de forma incremental, siguiendo la regla de aprendizaje del perceptrón. Esta evolución refleja el proceso mediante el cual el modelo minimiza el error y mejorar su capacidad de predicción:

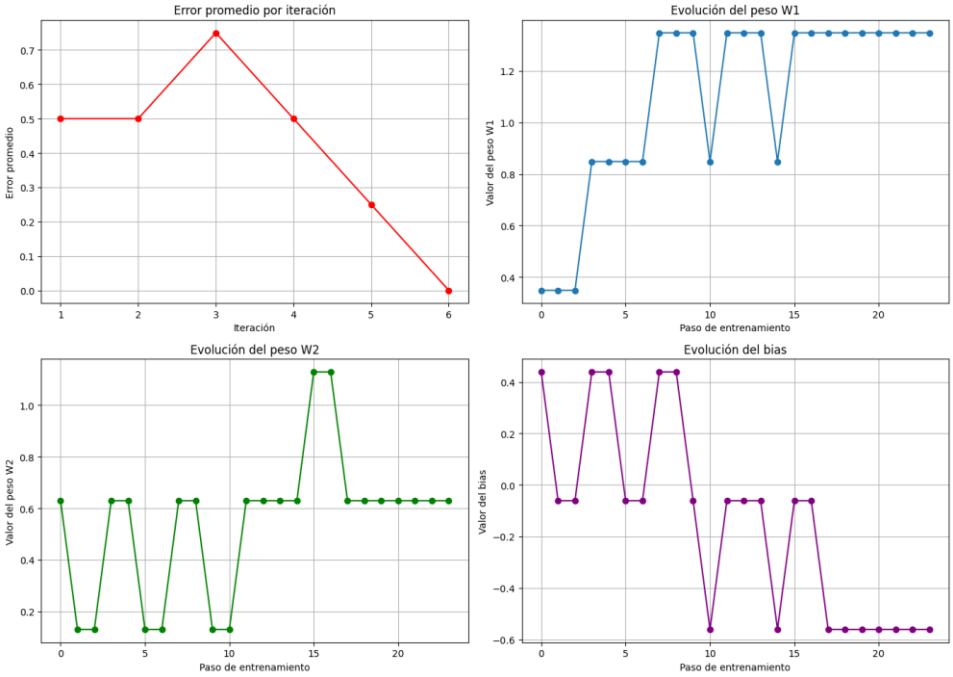

Ilustración 9. Evolución de los parámetros durante el entrenamiento (Puerta AND).

Por último, en la figura 10 representamos la frontera de decisión que ha aprendido nuestro modelo. En esta visualización, los puntos rojos corresponden a la clase 0 y el punto azul al único caso de salida 1. El hiperplano generado (línea discontinua verde) separa correctamente ambos tipos de clase, resolviendo con éxito el problema de clasificación:

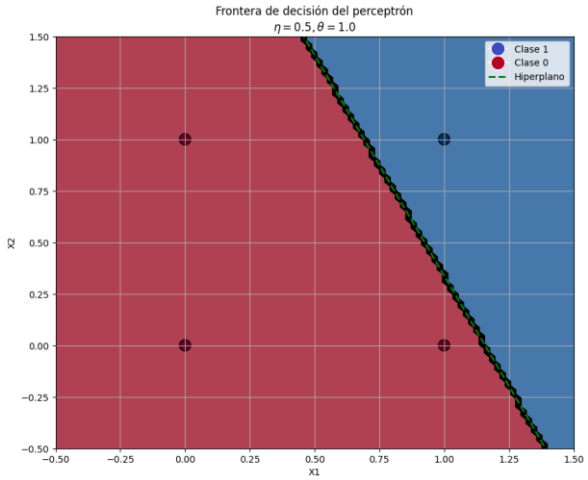

Ilustración 10. Frontera de decisión del perceptrón (Puerta AND).

2.3 Función lógica OR

La puerta lógica OR es otra operación booleana fundamental. A diferencia de la puerta AND, la OR devuelve una salida positiva (1) cuando al menos una de sus entradas es 1. Es decir, representa el concepto de *"o inclusiva*": basta con que una condición sea verdadera para que la salida lo sea también [8].

La tabla de verdad que corresponde a la función OR con dos datos de entrada es:

X1	X2	Y
0	0	0
0	1	1
1	0	1
1	1	1

Tabla 4. Puerta lógica OR.

En este caso, la única combinación que produce una salida negativa (0) es cuando ambas entradas son 0. Todas las demás dan como resultado un 1. Si representamos estos puntos en un plano, veremos que el punto `(0, 0)` se encuentra separado del resto. Esto significa que también es un problema linealmente separable, por lo que el perceptrón simple debe ser capaz de resolverlo correctamente.

Al igual que en la prueba anterior con la puerta AND, entrenamos el perceptrón sobre la función OR utilizando diferentes combinaciones de *learning rates* y valores de umbral. Los resultados siguieron siendo positivos: en todas las configuraciones, el modelo logró aprender correctamente la función, clasificando adecuadamente los cuatro patrones.

La mejor configuración que alcanzamos fue un *learning rate* $\eta = 0.1$ y un umbral $\theta = 0.5$, logrando la convergencia en solo una iteración, lo que nos evidencia tanto la simplicidad del problema como la eficiencia del algoritmo para resolverlo.

Iteración	Entradas		Pesos iniciales		Yr	Yd	Error	Pesos finales	
	X1	X2	W1	W2				W1	W2
1	0	0	0.7532	0.1451	0	0	0	0.7532	0.1451
1	0	1	0.7532	0.1451	0	1	1	0.7532	0.2451
1	1	0	0.7532	0.2451	0	1	1	0.8532	0.2451
1	1	1	0.8532	0.2451	1	1	0	0.8532	0.2451
2	0	0	0.8532	0.2451	0	0	0	0.8532	0.2451
2	0	1	0.8532	0.2451	0	1	1	0.8532	0.3451
2	1	0	0.8532	0.3451	1	1	0	0.8532	0.3451
2	1	1	0.8532	0.3451	1	1	0	0.8532	0.3451
3	0	0	0.8532	0.3451	0	0	0	0.8532	0.3451
3	0	1	0.8532	0.3451	0	1	1	0.8532	0.4451
3	1	0	0.8532	0.4451	1	1	0	0.8532	0.4451
3	1	1	0.8532	0.4451	1	1	0	0.8532	0.4451
4	0	0	0.8532	0.4451	0	0	0	0.8532	0.4451
4	0	1	0.8532	0.4451	0	1	1	0.8532	0.5451
4	1	0	0.8532	0.5451	1	1	0	0.8532	0.5451
4	1	1	0.8532	0.5451	1	1	0	0.8532	0.5451
5	0	0	0.8532	0.5451	0	0	0	0.8532	0.5451
5	0	1	0.8532	0.5451	1	1	0	0.8532	0.5451
5	1	0	0.8532	0.5451	1	1	0	0.8532	0.5451
5	1	1	0.8532	0.5451	1	1	0	0.8532	0.5451

Tabla 5. Tabla detalla de los resultados de entrenamiento (Puerta OR).

La frontera de decisión generada por el modelo para este caso puede expresarse con la siguiente ecuación del hiperplano:

$$0.8532 \cdot x_1 + 0.5451 \cdot x_2 - 0.4553 = 0$$

Ecuación 5. Ecuación del hiperplano (Puerta OR).

Esta ecuación divide el espacio de entrada en dos regiones: por un lado, el punto `(0, 0)` correspondiente a la clase 0, y por el otro, el resto de los puntos que pertenecen a la clase 1.

Durante el proceso de entrenamiento, hemos recigo métricas clave como la evolución del error medio por iteración, y la evolución de los pesos y del bias a lo largo del tiempo. En la Figura 11 podemos observar que el error disminuye progresivamente hasta alcanzar 0 en la iteración 5, confirmando que el modelo ha aprendido correctamente el patrón:

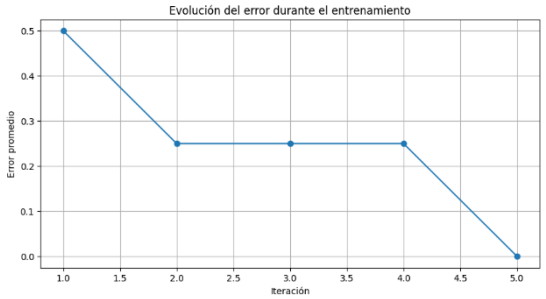

Ilustración 11. Evolución del error durante el entrenamiento (Puerta OR).

En la figura 12, ilustramos cómo varían los parámetros del modelo a medida que aprende. Se puede ver cómo los pesos `w1` y `w2`, así como el sesgo, se van ajustando para reducir el error. Esto Vuelve a refleja de forma directa la regla de aprendizaje del perceptrón, que modifica los pesos únicamente cuando se produce un error:

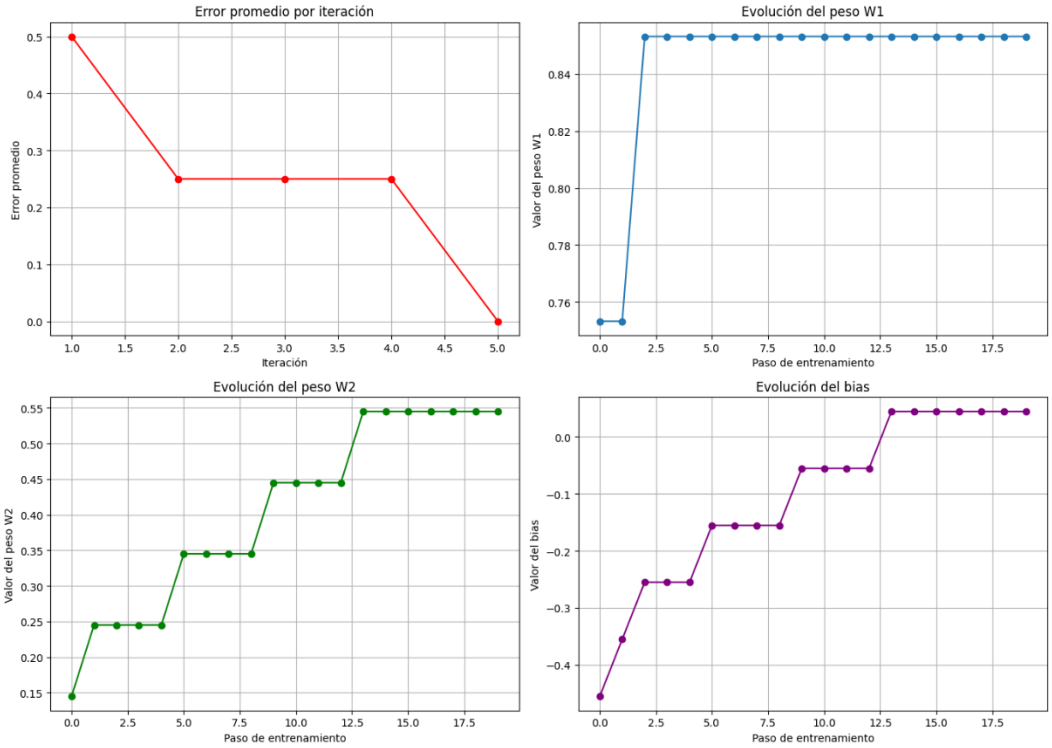

Ilustración 12. Evolución de los parámetros durante el entrenamiento (Puerta OR).

Finalmente, en la figura 13 presentamos la frontera de decisión aprendida por nuestro perceptrón. El hiperplano separa claramente el punto negativo (`0, 0`) de los positivos, con una línea verde discontinua que actúa como límite de decisión:

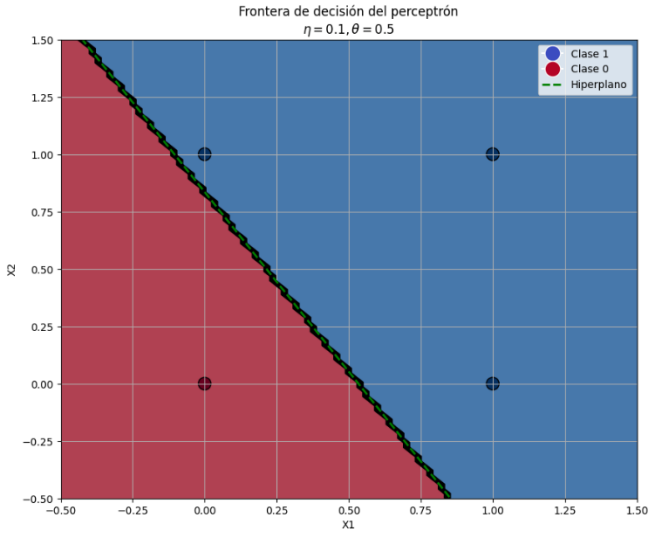

Ilustración 13. Frontera de decisión del perceptrón (Puerta OR).

La prueba con la función lógica OR no era necesaria, pero la hemos realizado con el fin de confirma el correcto funcionamiento de nuestro modelo sobre datos linealmente separables. Como era de esperar, el perceptrón fue capaz de encontrar rápidamente una solución precisa, adaptando sus pesos y bias para crear un hiperplano. Esto refuerza el hecho de que el perceptrón simple es adecuado para resolver problemas de clasificación binaria cuando las clases pueden separarse linealmente.

2.4 Función lógica XOR

Tras comprobar que el perceptrón funcionaba correctamente con compuertas lógicas linealmente separables como AND y OR, decidimos poner a prueba sus límites con la función XOR. Esta puerta, a diferencia de las anteriores, devuelve 1 solo cuando una de las dos entradas es 1, pero no ambas. Su tabla de verdad es la siguiente:

X1	X2	Y
0	0	0
0	1	1
1	0	1
1	1	0

Tabla 6. Puerta lógica XOR.

A simple vista, la salida de esta puerta parece lógica y coherente. Sin embargo, no existe una línea recta que separe las clases de forma correcta en el espacio bidimensional. Esto convierte al problema de la puerta XOR en un caso no linealmente separable, lo cual es clave para entender los resultados obtenidos.

Utilizando exactamente el mismo procedimiento anterior, entrenamos el perceptrón con distintas combinaciones de *learning rate* y umbrales. En este caso, los resultados fueren diferentes: ninguna configuración fue capaz de lograr la convergencia. En todos los casos, el modelo fallaba en clasificar correctamente al menos uno de los cuatro patrones.

Esto podemos observarlo claramente en los resultados de las predicciones. Por ejemplo, con un *learning rate* $\eta = 0.5$ y un umbral $\theta = 0.5$, el modelo predice correctamente $(0, 1) \rightarrow 1$ y $(1, 1) \rightarrow 0$, pero falla con $(1, 0) \rightarrow 1$ y $(0, 0) \rightarrow 0$, confundiendo el patrón clave de esta función.

```
1. === Learning Rate: 0.5, umbral: 0.5 ===
2.
3. Resultados del perceptrón:
4. --------------------------------
5. | X1 | X2 | Target | Predecido |
6. --------------------------------
7. | 0  | 0  | 0      | 1         |
8. | 0  | 1  | 1      | 1         |
9. | 1  | 0  | 1      | 0         |
10. | 1  | 1  | 0      | 0         |
11. --------------------------------
```

Ilustración 14. Ejemplo de los resultados finales erróneos perceptrón simple (Puerta XOR).

Incluso tras múltiples iteraciones e intentos, el modelo era incapaz de encontrar una combinación de pesos y bias que separase correctamente las clases. Finalmente, el algoritmo no logró converger, confirmándonos de forma empírica una de las limitaciones teóricas del perceptrón simple: no puede resolver problemas no linealmente sepárales.

3 Laboratorio 2 – Perceptrón multicapa XOR

Dado que el perceptrón simple no es capaz de resolver problemas que no son linealmente separables, como es el caso de la función XOR, hemos implementado un perceptrón multicapa o MLP (*Multi-Layer Perceptron*) para solucionar esta limitación. En la primera parte de esta sección, nos centramos en una red con función de activación ReLU en la capa oculta, aunque posteriormente analizaremos también los resultados obtenidos con la función sigmoide. Para ello, desarrollamos una clase en PyTorch que nos ha permitido definir con flexibilidad la arquitectura del modelo, el número de neuronas en la capa oculta, el número máximo de iteraciones, el tipo de función de activación y el *learning rate.*

La arquitectura que hemos implementado consta de una capa de entrada de tamaño 2 (corresponde a las dos variables de entrada de la función XOR), una capa oculta configurable y una capa de salida con una única neurona. La función de activación que usamos en la salida es siempre la sigmoide, esto lo decidimos así ya que su rango de salida entre 0 y 1 se adapta bien al problema de clasificación binaria. Nuestra clase de `Perceptron` también permite seleccionar como función de activación de la capa oculta ReLU o sigmoide.

El proceso de aprendizaje lo realizamos mediante el algoritmo Adam, un optimizador basado en el gradiente descendente adaptativo. Este optimizador inicializa los pesos W y sesgos b de la red aleatoriamente, además de los vectores de momento $m_0 = 0$ y $v_0 = 0$. A partir de estos momentos, y utilizando los hiperparámetros de Adam ($\alpha, \beta_1, \beta_2, \epsilon$), actualizamos los parámetros de la red en cada iteración. En cada época, la red realiza una propagación hacia delante calculando la salida de capa oculta con la función ReLU o sigmoide, y posteriormente la salida final con la función sigmoide. A continuación, calculamos el error mediante la función de pérdida de error cuadrático medio (MSE), definida como [9]:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_{pred,i} - y_i)^2$$

Ecuación 6. Ecuación error cuadrático medio.

Posteriormente, realizamos la retropropagación (*backpropagation*) calculando los gradientes del error respecto a los parámetros de la red. Estos gradientes los utilizamos en las siguientes actualizaciones, donde Adam ajusta los pesos mediante el uso de los momentos corregidos:

$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$$

$$v_t = \beta_2 \cdot v + (1 - \beta_2) \cdot {g_t}^2$$

$$\hat{m}_t = \frac{m_t}{1 - {\beta_1}^t}$$

$$\hat{v}_t = \frac{v_t}{1 - {\beta_2}^t}$$

$$W_t = W_{t-1} - \alpha \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

Ecuación 7. Ajuste con Adam en cada iteración.

Repetimos este proceso durante un número máximo de épocas o hasta que alcanzamos un umbral de error mínimo, aplicando además una estrategia de *early stopping* si el error no mejora durante varias iteraciones consecutivas.

En cuanto a la implementación, el entrenamiento lo llevamos a cabo mediante el siguiente fragmento de código, donde podemos ver cómo realizamos la propagación, retropropagación y actualización de pesos utilizando Adam:

```
for epoca in range(self.max_iteraciones):

    #1. Fast Forward
    salida = self.forward(X)
    error = self.mse(salida, y)
    error_valor = error.item()
    self.errores.append(error_valor)

    # 2. Backpropagation
    # Limpiamos los gradientes para no acumularlos
    optimizador.zero_grad()

    # Calculamos los gradientes
    error.backward()

    # Actualizamos los pesos
    optimizador.step()
```

Ilustración 15. Implementación del entrenamiento del MLP.

3.1 Función de activación ReLU

Para abordar el problema de clasificación de la función XOR, comenzamos evaluando el rendimiento de nuestro MLP utilizando como función de activación en la capa oculta la función ReLU. ReLU (*Rectified Linear Unit*) es una función ampliamente utilizada en redes neuronales debido a su simplicidad computacional y a sus propiedades favorables para el aprendizaje profundo. La función ReLU se define matemáticamente como [10]:

$$f(x) = max(0, x)$$

Ecuación 8. Función ReLU

Esta función anula las activaciones negativas y deja pasar sin modificación las activaciones positivas. Su principal ventaja respecto a funciones como la sigmoide o la tangente hiperbólica es que no satura en el rango positivo, lo que reduce drásticamente el problema del desvanecimiento del gradiente y permite una convergencia más rápida en redes profundas. Además, al ser lineal por tramos, mantiene la no linealidad necesaria para que la red aprenda funciones complejas, pero con un coste computacional bajo.

Para evaluar el rendimiento del modelo, variamos dos hiperparámetros fundamentales: el número de neuronas ocultas (2, 4 y 8) y el *learning rate* (0.01, 0.1 y 0.5). Entrenamos un total de nueve modelos con distintas combinaciones de estos valores. A continuación, mostramos una tabla resumen con el número de épocas necesarias, el error final y la salida predicha para cada modelo:

Neuronas capa oculta	Épocas	Salida	Error
2	431	[0.662, 0.662, 0.662, 0.056]	0.167481
2	17	[0.483, 0.483, 0.483, 0.483]	0.250702
2	54	[0.000, 0.649, 0.649, 0.649]	0.166852
4	500	[0.066, 0.910, 0.943, 0.058]	0.004793
4	25	[0.500, 0.498, 0.493, 0.491]	0.250034
4	39	[0.518, 0.518, 0.518, 0.518]	0.250459
8	500	[0.028, 0.978, 0.975, 0.023]	0.000604
8	183	[0.007, 0.995, 0.978, 0.005]	0.000141
8	32	[0.504, 0.504, 0.504, 0.504]	0.250212

Tabla 7. Tabla de resultados de los modelos (Función activación ReLU).

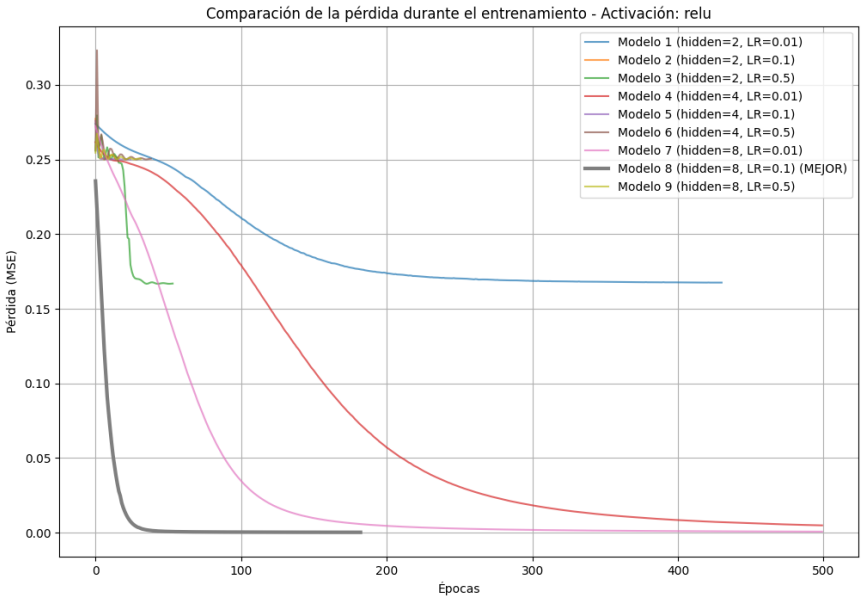

Ilustración 16. Evolución del error de los diferentes modelos (Función activación ReLU).

Los resultados nos muestran que el modelo con 8 neuronas en la capa oculta y *learning rate* de 0.1 fue el que mejor se comportó, alcanzando un error mínimo de 0.000141 y clasificando correctamente todos los patrones de la puerta XOR. Además, la convergencia fue rápida y estable, tal y como podemos observar en la siguiente gráfica de evolución del error del mejor modelo, donde la pérdida disminuye bruscamente en las primeras épocas y luego se estabiliza cercano a cero.

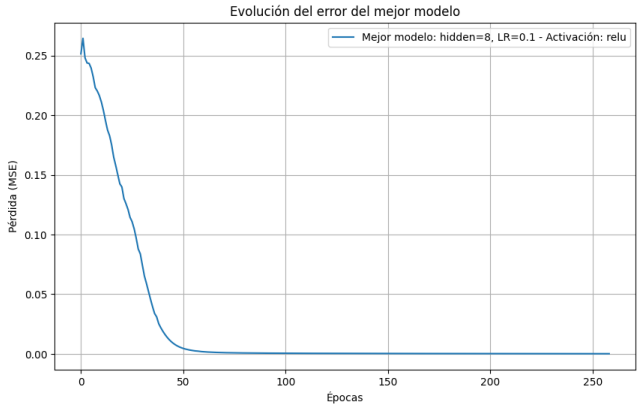

Ilustración 17. Evolución del error del mejor modelo (Función activación ReLU).

La representación de la frontera de decisión que genera este modelo, la cual podemos observar en la siguiente ilustración, confirma que nuestro modelo ha aprendido correctamente la lógica XOR. Podemos observar cómo se han creado regiones no lineales que separan adecuadamente los puntos de entrada que deben clasificarse como clase 0 y clase 1, lo cual valida visualmente la capacidad de nuestro MLP con ReLU para resolver problemas linealmente no separables.

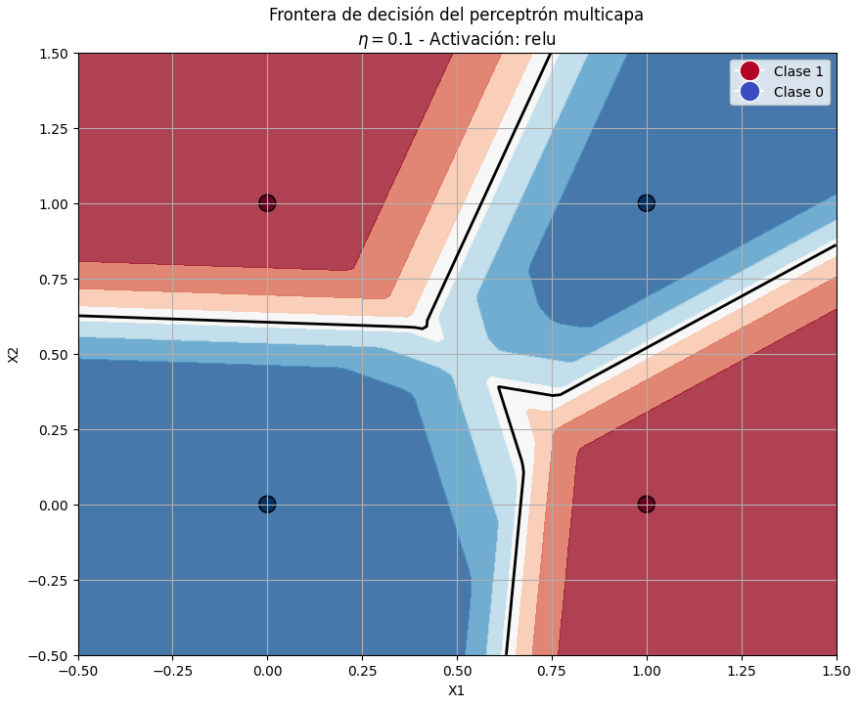

Ilustración 18. Frontera de decisión del MLP con ReLU (Puerta XOR).

Esta implementación nos demuestra que la combinación de la función ReLU en la capa oculta y el algoritmo Adam como método de optimización es una estrategia eficaz para entrenar redes neuronales en tareas de clasificación no lineal. En particular, el modelo que hemos entrenado ha logrado clasificar correctamente cuatro patrones de la función XOR, y alcanzó un error cuadrático medio muy bajo, lo que indica que cuenta con una buena capacidad de generalización y aprendizaje. Sin embargo, aunque los resultados obtenidos son notablemente buenos, no pueden considerar completamente perfectos.

La función ReLU tiene varias ventajas, como su simplicidad computacional y su capacidad para mitigar el problema del desvanecimiento del gradiente. No obstante, también tiene limitaciones importantes. Una de ellas se trata de un fenómeno que se produce cuando ciertas neuronas dejan de activarse permanentemente (es decir, su salida es siempre cero), lo que puede dificultar la capacidad de nuestro modelo para aprender representaciones ricas. Esto ocurre cuando los valores de entrada a una neurona caen consistentemente en la región negativa de la función, y sus pesos no se actualización durante el entrenamiento. Además, en problemas muy sensibles a los pequeños cambios en los datos, la no linealidad abrupta de ReLU puede generar superficies de pérdida menos suaves, complicando la convergencia hacia mínimos óptimos [11].

En la próxima sección, vamos a abordar los resultados obtenidos utilizando la función de activación sigmoide, con el fin de comprar su desempeño con respecto a ReLU y analizar si ofrece mayor estabilidad o ventajas adicionales en este tipo de tareas.

3.2 Función de activación sigmoide

Con el objetivo de comparar el rendimiento y la robustez frente a ReLU, repetimos las mismas pruebas utilizando la función de activación sigmoide en la capa oculta. Esta función es capaz de representar transiciones suaves entre clases, produciendo valores en el rango [0, 1], lo que puede favorecer la estabilidad en tareas binarias. Esta función se define matemáticamente como [12]:

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

Ecuación 9. Función Sigmoide

Los resultados que hemos obtenido muestran también una buena capacidad de aprendizaje, aunque con diferencias notables en cuanto al número de iteraciones y la forma de la frontera de decisión. El mejor modelo con sigmoide utilizó 4 neuronas en la capa oculta y un learning rate de 0.5, alcanzando un error de 0.000115 en tan solo 137 iteraciones. La evolución del error fue más regular y la convergencia más rápida que en muchos de los modelos con ReLU. A continuación, presentamos la tabla resumen de todos los modelos evaluados con sigmoide:

Neuronas capa oculta	Épocas	Salida	Error
2	500	[0.083, 0.488, 0.924, 0.508]	0.133209
2	330	[0.016, 0.666, 0.666, 0.667]	0.166825
2	74	[0.003, 0.663, 0.663, 0.663]	0.166718
4	21	[0.479, 0.483, 0.477, 0.481]	0.250346
4	330	[0.012, 0.986, 0.987, 0.016]	0.000190
4	137	[0.007, 0.991, 0.990, 0.015]	0.000115
8	18	[0.506, 0.510, 0.524, 0.524]	0.249623
8	211	[0.009, 0.989, 0.987, 0.015]	0.000150
8	216	[0.002, 0.988, 0.994, 0.020]	0.000148

Tabla 8. Tabla de resultados de los modelos (Función activación Sigmoide).

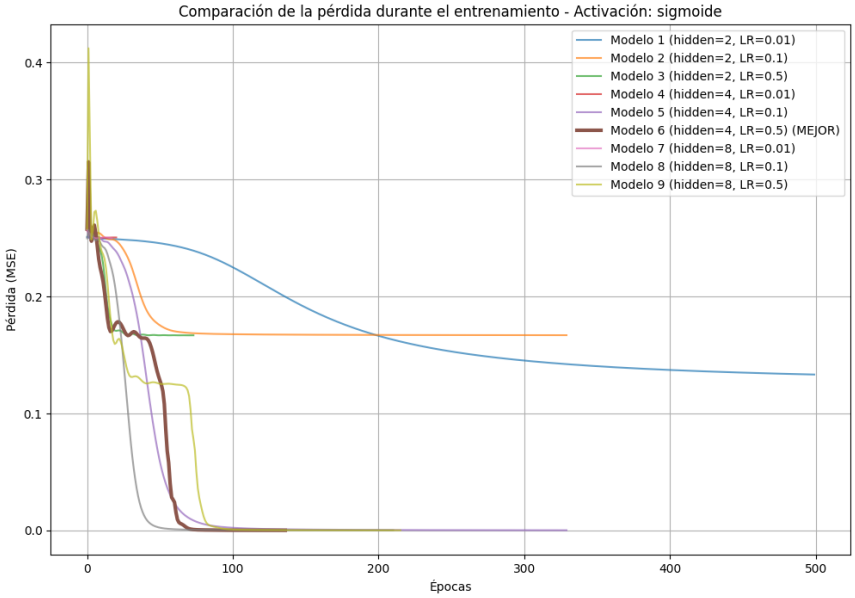

Ilustración 19. Evolución del error de los diferentes modelos (Función activación Sigmoide).

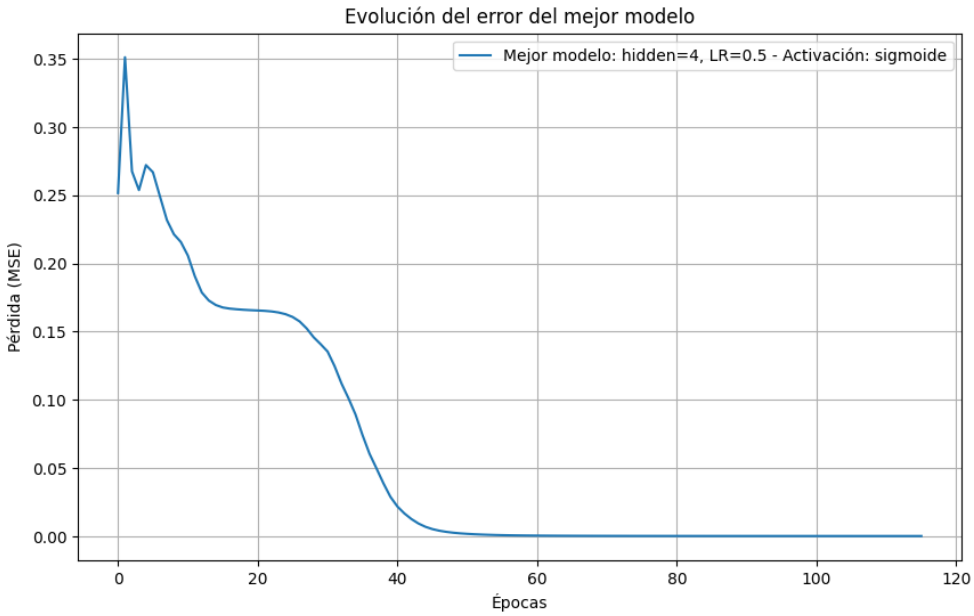

Ilustración 20. Evolución del error del mejor modelo (Función activación Sigmoide).

En la siguiente ilustación, podemos ver la frontera de decisión de nuestro mejor modelo con sigmoide, que, aunque también resuelve correctamente la clasificación, presenta una forma más suave y simétrica que la obtenida con ReLU, lo que nos indica una transición más gradual entre clases. Este tipo de fronteras puede ser beneficioso en contextos donde se requiere mayor regularidad o robustez ante el ruido.

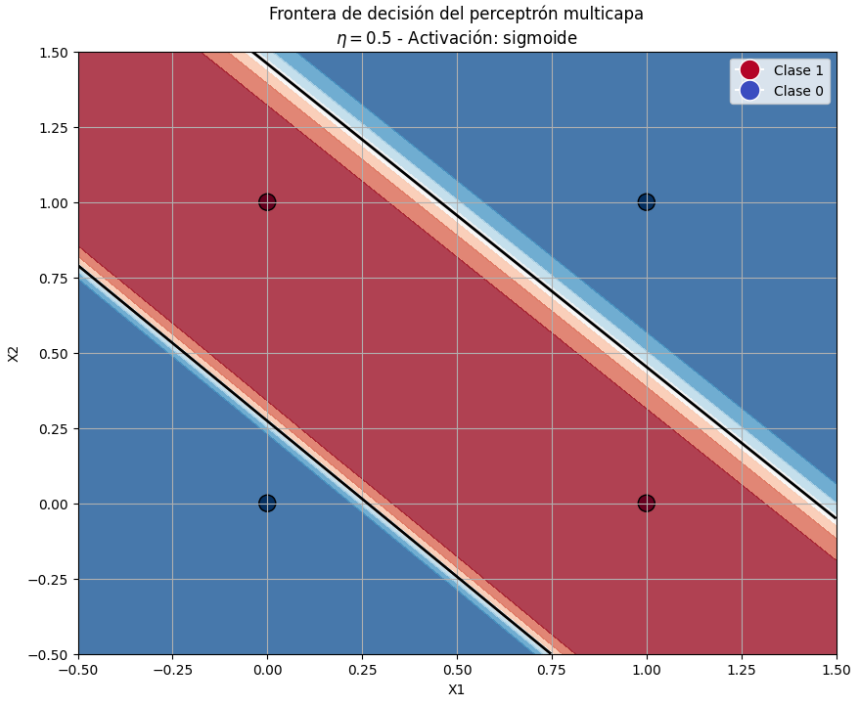

Ilustración 21. Frontera de decisión del MLP con Sigmoide (Puerta XOR).

En conclusión, ambos modelos han sido capaces de resolver el problema XOR con un error muy bajo. ReLU nos ha permitido encontrar una solución más expresiva con un mayor número de neuronas y una frontera más compleja, mientras que con la función sigmoide obtuvimos resultados comparables con menos neuronas y en menos iteración. Sin embargo, la elección de la función de activación depende del contexto: ReLU es preferible por eficiencia en problemas grandes o profundos, mientras que la sigmoide puede ser más estable en redes pequeñas o datos con simetría clara, como hemos visto en la práctica. Esta comparativa nos ha evidenciado cómo el diseño de la arquitectura y la selección de hiperparámetros impactan directamente en la capacidad de aprendizaje de las redes neuronales.

4 Laboratorio 3 – Perceptrón multicapa *Juego de Tronos*

4.1 Cuestión 1

En esta sección, describiremos la arquitectura de la red neuronal que utilizamos en nuestro modelo, así como los parámetros y funciones empleadas durante el entrenamiento.

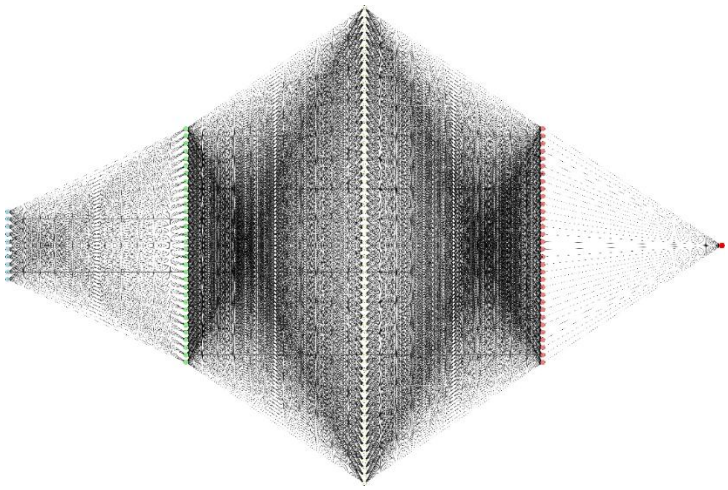

Ilustración 22. Arquitectura del modelo

1. **Optimizador:** Usamos el optimizador Adam, que es un algoritmo bastante eficiente y adaptativo, ajustando automáticamente las tasas de aprendizaje de cada parámetro. Establecimos un learning rate (tasa de aprendizaje) de 0.001, que nos permitió un ajuste controlado y preciso de los pesos a lo largo del entrenamiento.

2. **Función de Error:** Para calcular el error entre las predicciones y los valores reales, decidimos utilizar la función de error Error Cuadrático Medio (MSE). Este es un método común en tareas de regresión, que calcula la diferencia entre las predicciones y los valores reales, eleva al cuadrado esas diferencias y las promedia.

3. **Arquitectura de la Red:** Después de múltiples pruebas con diferentes arquitecturas, número de capas y neuronas, la que mejor resultado nos dio fue la siguiente distribución:

    - *Capas ocultas:* La red tiene un total de tres capas ocultas:

        - *Capa oculta 1 y Capa oculta 3:* Ambas tienen 32 neuronas.

        - *Capa oculta 2:* Esta capa tiene 64 neuronas, lo que le permite aprender características más complejas.

4. **Funciones de Activación:** En las capas ocultas, usamos la función de activación ReLU (Rectified Linear Unit). Esta función es muy eficaz porque ayuda a evitar el problema del desvanecimiento del gradiente y acelera el proceso de entrenamiento. En la capa de salida, optamos por la función Sigmoid, que limita la salida del modelo entre 0 y 1, lo cual es útil cuando se requiere una salida normalizada, como en problemas de clasificación binaria.

5. **Errores Finales:** Durante el proceso de entrenamiento, evaluamos el rendimiento del modelo tanto en el conjunto de entrenamiento como en el de validación:

   - La mejor pérdida en entrenamiento que conseguimos fue de 0.0555, lo que indica que el modelo logró un buen ajuste con los datos de entrenamiento.

   - La mejor pérdida en validación fue de 0.4472, lo que muestra que el modelo aún tiene margen de mejora en cuanto a la generalización a nuevos datos.

   - El mejor MSE en validación fue de 0.0639, lo que refleja el error cuadrático medio alcanzado en el conjunto de validación.

4.2 Cuestión 2

Una vez tenemos el modelo entrenado y produciendo unos resultados decentes, podemos comenzar a evaluar su capacidad de predicción con un nuevo dataset, el “got_predict”. Dados unos personajes, observaremos cuál de ellos es el que más probabilidades tiene de morir según nuestro modelo.

Realizamos los mismos pasos introductorios para transformar y adaptar el dataset como ya se hizo con el dataset de entrenamiento, y comenzamos la evaluación:

Tras el proceso de evaluación, podemos observar los siguientes resultados:

Personaje	Probabilidad_Vivir	Probabilidad_Morir
Tommen Baratheon	0.015961	0.984039
Daenerys Targaryen	0.100902	0.899098
Roland Crakehall (Kingsguard)	0.456576	0.543424
Othell Yarwyck	0.554014	0.445986
Coldhands	0.710090	0.289910

Error
0.005039
0.053902
0.302424
0.079986
0.073090

Ilustración 23. Resultados de la evaluación del modelo

Basándonos únicamente en los resultados, el personaje Tommen Baratheon se trata del candidato más probable a morir, en un 98.4% de los casos.

Si comparamos la predicción del alive con esta de validación, obtenemos un escaso error de 0.005, que se trata de un valor mucho más bajo del que nos salió en el entrenamiento (0,4472).

Este error tan minúsculo puede indicar, sin embargo, que el modelo ha podido sobreaprenderse un personaje del dataset de entrenamiento muy similar a Tommen, dándose así un caso de overfitting.

Aunque esto debería ser un aspecto negativo del modelo, puede deberse a que el dataset es demasiado pequeño, o puede ser que ese elemento se haya repetido muchas veces ocasionando que la precisión de evaluación de este individuo en concreto se vea muy potenciada.

4.3 Cuestión 3

Si queremos forzar al modelo a elegir otro personaje con más probabilidades de morir, debemos conocer en profundidad aquellos atributos que aumentan dicha probabilidad.

Para ello, crearemos una matriz de correlaciones basada en el campo alive que nos mostrará con claridad lo que buscamos. Cuanto menor sea la correlación con alive, más influirá ese atributo en los chances de muerte del personaje. Los resultados son los siguientes:

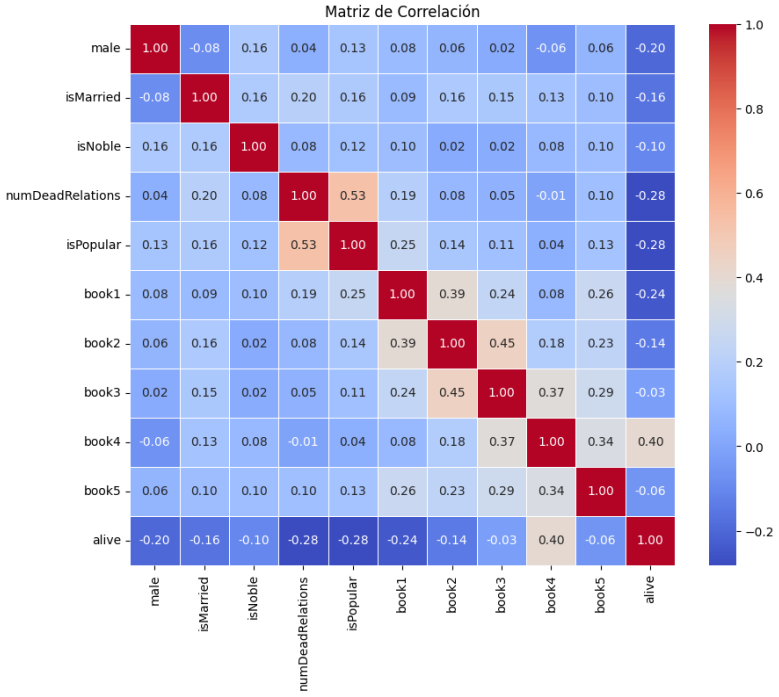

Ilustración 24. Matriz de correlación de los atributos con el campo alive

Observando los resultados, se puede concluir que todos aquellos valores negativos influyen en el aumento de las probabilidades de muerte, mientras que los positivos influyen en la probabilidad de supervivencia.

Para este caso, se optó por aumentar únicamente los cuatro primeros atributos que influyen en la muerte del personaje (isPopular, numDeadRelations, book1, male) y disminuir solamente uno de los positivos (book4) para observar que le ocurriría a un nuevo personaje al que hemos llamado Allan Nyom que recoge el aumento y decremento de los atributos anteriormente mencionados.

Después de inicializar a nuestro conejillo de indias Nyom, evaluamos el modelo, y observamos los siguientes resultados:

Personaje	Probabilidad_Vivir	Probabilidad_Morir
Allan Nyom	0.000965	0.999035
Tommen Baratheon	0.015961	0.984039
Daenerys Targaryen	0.100902	0.899098
Roland Crakehall (Kingsguard)	0.456576	0.543424
Othell Yarwyck	0.554014	0.445986
Coldhands	0.710090	0.289910

Error
0.000965
0.005039
0.053902
0.302424
0.079986
0.073090

Ilustración 25. Resultados de evaluación tras añadir al candidato con los atributos de muerte potenciados.

Se puede ver como Nyom ha sucedido a Tommen como el personaje con la mayor probabilidad de morir, con un 0.999. El error (0.000965), más bajo del que se obtuvo en la primera validación, nos indica que sucede el mismo escenario que con el dataset de got_predict, por lo que se aconsejaría utilizar un dataset mucho más grande para intentar evitar el posible overfitting que se ha generado.

5 Conclusiones

Durante el desarrollo de esta práctica hemos podido comprobar los fundamentos teóricos que explican el funcionamiento de los perceptrones y perceptrones multicapa. En primer lugar, analizamos el comportamiento de un perceptrón simple sobre funciones lógicas linealmente separables como AND y OR, verificando que el modelo convergía rápidamente con un número reducido de iteraciones. Sin embargo, al enfrentarse a un problema como XOR, que no es linealmente separable, el perceptrón simple era incapaz de encontrar una solución válida, lo que nos demostró sus limitaciones estructurales.

Para superar esta barrera, implementamos un MLP y analizamos su comportamiento sobre el mismo problema. En esta fase de la práctica observamos cómo el número de neuronas ocultas, la tasa de aprendizaje y la elección de la función de activación afectan significativamente al aprendizaje. Tanto ReLU como sigmoide permitieron resolver el problema XOR, pero presentaban características diferenciadas: ReLU nos ofrecía una frontera de decisión más expresiva y una convergencia eficiente, aunque con riesgos como la desactivación permanente de neuronas; mientras que la sigmoide nos ofrecía una convergencia más estable y con menos neuronas, pero podía verse afectada por el desvanecimiento del gradiente en redes profundas.

Por último, al aplicar estos conocimientos a un caso más complejo como la predicción de muertes en Juego de Tronos, observamos que los MLP pueden adaptarse a tareas reales, aunque también es crucial prestar atención a la calidad y tamaño del dataset para evitar el sobreajuste. Esta práctica nos ha permitido no solo profundizar en la teoría del aprendizaje automático, sino también adquirir experiencia práctica en el diseño, implementación y evaluación de modelos neuronales en PyTorch, entendiendo cómo ajustar arquitecturas y parámetros para lograr una buena generalización.

6 Referencias

[1] F. Rosenblatt, «The perceptron: A probabilistic model for information storage and organization in the brain,» *Psychological Review,* vol. 65, nº 6, pp. 386-408, 1958.

[2] A. B. J. Novikoff, «On convergence proofs on perceptrons. In Symposium on the mathematical theory of automata,» *Proceedings of the Symposium on the Mathematical Theory of Automata,* vol. 12, pp. 615-622, 1962.

[3] «Perceptron - an overview | ScienceDirect Topics,» [En línea]. Available: https://www.sciencedirect.com/topics/nursing-and-health-professions/perceptron.

[4] I. B. Y. &. C. A. Goodfellow, Deep learning, MIT Press, 2016.

[5] «Multilayer Perceptron - an overview | ScienceDirect Topics,» [En línea]. Available: https://www.sciencedirect.com/topics/computer-science/multilayer-perceptron.

[6] DataCamp, «Multilayer perceptrons in Machine Learning: A comprehensive guide,» 28 Julio 2024. [En línea]. Available: https://www.datacamp.com/tutorial/multilayer-perceptrons-in-machine-learning.

[7] Geeks for geeks, «AND Gate - GeeksForGeeks,» 18 Diciembre 2024. [En línea]. Available: https://www.geeksforgeeks.org/and-gate/.

[8] Geeks for geeks, «OR Gate - GeeksForGeeks,» 29 Noviembre 2024. [En línea]. Available: https://www.geeksforgeeks.org/or-gate/.

[9] Geeks for geeks, «Mean Squared Error,» 3 Marzo 2025. [En línea]. Available: https://www.geeksforgeeks.org/mean-squared-error/.

[10] J. Brownlee, «A Gentle Introduction to the Rectified Linear Unit (ReLU),» 20 Agosto 2020. [En línea]. Available: https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/.

[11] «What is the "dying ReLu" problem in neural networks?,» 10 Febrero 2024. [En línea]. Available: https://www.geeksforgeeks.org/what-is-the-dying-relu-problem-in-neural-networks/.

[12] «A Gentle Introduction To Sigmoid Function,» MachineLearningMastery.com, 24 Agosto 2021. [En línea]. Available: https://machinelearningmastery.com/a-gentle-introduction-to-sigmoid-function/.