Implementación del Algoritmo de Back-Propagation Para su Aplicación en la Clasificación, Regresión e IA Generativa

Carlos Santana Esplá y
Ricardo Juan Cárdenes Pérez

Enero, 2024

Resumen

Presentamos en este trabajo las bases matemáticas para implementar el algoritmo de retropropagación, utilizado en aprendizaje profundo para el entrenamiento de modelos basados en redes neuronales, para una red neuronal fully-connected. También implementamos dicho algoritmo y desarrollamos diversos modelos, obteniendo una precisión superior a 92 % para la clasificación del conunto de datos MNIST, muy bajos errores en varios problemas de regresión relevantes, y excelentes resultados en modelos de reducción de ruido para dichas imagenes.

1. Introducción

Desde sus primeros pasos en la década de 1970 hasta su integración en arquitecturas neuronales más complejas en la actualidad, el algoritmo de retropropagación ha desempeñado un papel esencial en la evolución y mejora continua de las capacidades de las redes neuronales. Durante sus primeras instancias, el backpropagation sentó las bases para el desarrollo de sistemas de aprendizaje automático, proporcionando un mecanismo eficiente para ajustar los pesos y sesgos de las redes en función de los errores de predicción. A medida que la investigación avanzaba, este algoritmo se hizo crucial para superar desafíos en la captura de patrones complejos en conjuntos de datos de alta dimensionalidad.

Con el tiempo, el backpropagation ha evolucionado, y se ha adaptado para su implementación en arquitecturas neuronales más avanzadas, permitiendo la construcción de modelos más profundos y sofisticados, como se hará en este artículo. Su contribución se ha vuelto aún más significativa a medida que las aplicaciones de inteligencia artificial se expanden a campos como el reconocimiento de imágenes, el procesamiento de lenguaje natural y la conducción autónoma, donde la capacidad de las redes para discernir patrones sutiles y abordar problemas complejos es esencial. En este contexto, el backpropagation se destaca como una herramienta fundamental que ha allanado el camino para el desarrollo y despliegue exitoso de diversas aplicaciones de inteligencia artificial.

Inspirado por los principios del cálculo diferencial, este método comienza dada una inicialización aleatoria de pesos y sesgos de una red neuronal, y a través de la propagación hacia adelante, las entradas se procesan capa a capa, aplicando sobre ellas funciones de activación, generando predicciones. Luego, se evalúa la discrepancia entre estas predicciones y las salidas reales a través de una función de pérdida, lo cual se verá más en detalle en la sección 3. La retropropagación del error sigue, calculando las derivadas parciales de la función de pérdida respecto a los pesos y sesgos en cada capa, permitiendo ajustar los parámetros de la red mediante técnicas de optimización, como detallaremos más adelante.

2. Arquitectura de red Fully-Connected

Supongamos que tenemos una red neuronal FC compuesta por l capas. Cada una de las capas, ocultas o de salida, computará una combinación lineal con las salidas de la capa anterior, las cuales, para la capa k, se denotan con el vector $A^{[k-1]}$. Para ello, cada neurona i de la capa k contará con un vector de pesos $w_i = (w_{1i} \cdots w_{n(k-1),i}) \in \mathbb{R}^{n(k-1)}$, donde $n(k-1)$ representa el número de neuronas de la capa anterior, y un sesgo o bias $b_i^{[k]}$. Estos, en conjunto, actuarán como los coeficientes en dicha operación. A su salida, se le aplicará una no linealidad que será general para toda la capa, evitando que la red colapse en una única combinación lineal compuesta de las entradas. Así, tenemos que la salida de la neurona en cuestión para la capa arbitraria escogida no es más que $A_i^{[k]}$, donde

$$z_i^{[k]} = w_i^{[k]} \cdot A^{[k-1]} + b_i^{[k]} \tag{1}$$

$$A_i^{[k]} = g_k\left(z_i^{[k]}\right) \tag{2}$$

Siendo $g_k : \mathbb{R} \to \mathbb{R}$ la función de activación definida para dicha capa y $z_i^{[k]}$ la combinación lineal descrita. La notación utilizada para los pesos, algo que cobrará gran importancia en las siguientes secciones, es $w_{ji}^{[k]}$, donde los índices i y j representan la neurona de entrada en la capa k, y la de salida en la capa $k-1$, respectivamente.

Podemos calcular así para una muestra cualquiera $x \in \mathbb{R}^n$ su salida en la red como

$$A^{[l]} = g_l(w^{[l]} \cdot (g_{l-1}(w^{[l-1]} \cdot (\cdots(g_1(w^{[1]}x^t + b^{[1]}))) + b^{[l-1]})) + b^{[l]}) \tag{3}$$

Hemos definido así el feedfoward de nuestra red. Ahora bien, ¿cómo podemos optimizar esta red para que la salida se ajuste correctamente a la realidad de los datos?

3. Optimización de la red neuronal

Usaremos la función de error de Cross-Entropy, la cual resulta ser de las mejores opciones a la hora de desarrollar redes de clasificación, aunque otras como MSE nos traen al mismo resultado ajustando adecuadamente las constantes implicadas. La función se define, para una muestra $X \in \mathbb{R}^n$ y su etiqueta $Y \in \{0,1\}^C$, como

$$\mathcal{L}(X,Y) = -\sum_{i=1}^{C} y_i \cdot log(A_i^{[l]}) \tag{4}$$

donde $A_i^{[l]}$ es la salida de la red para la $i-$ésima clase de la muestra X. Ante la posibilidad de trabajar en un espacio muestral con múltiples clases, usamos la función softmax como función de activación para la capa de salida. Esto es, sea $z^{[l]}$ el vector que contiene los cálculos lineales realizados en dicha capa, la salida de la misma se define como

$$A^{[l]} = \frac{e^{z^{[l]}}}{\sum_{j=1}^{C} e^{z_j^{[l]}}} \tag{5}$$

Sustituyendo este vector de salida en la función de coste dada en la ecuacion (4), obtenemos que

$$\mathcal{L}(X,Y) = -\sum_{i=1}^{C} y_i \cdot log\left(\frac{e^{z_i^{[l]}}}{\sum_{j=1}^{C} e^{z_j^{[l]}}}\right) \tag{6}$$

Y al aplicar las propiedades básicas del logaritmo

$$\mathcal{L}(X,Y) = -\sum_{i=1}^{C} y_i \left(log\left(e^{z_i^{[l]}}\right) - log\left(\sum_{j=1}^{C} e^{z_j^{[l]}}\right)\right) \tag{7}$$

El algoritmo de retropropagación (Back-Propagation) se fundamenta en el cálculo de las derivadas del error con respecto a los diferentes pesos de la red. Estas derivadas se utilizan en técnicas de descenso del gradiente para actualizar los pesos, uno por uno, epoch tras epoch (cada ciclo de entrenamiento). Por consiguiente, procederemos a calcular las derivadas correspondientes a los pesos de la capa de salida. Para lograr esto, aplicaremos la regla de la cadena. Empezamos derivando la función de error con respecto a $z_i^{[k]}$, donde lo primero será aplicar la regla de la derivada para la suma de funciones. Así

$$\frac{\partial \mathcal{L}}{\partial z_i^{[l]}} = -\frac{\partial}{\partial z_i^{[l]}}\left(y_i \cdot log\left(e^{z_i^{[l]}}\right)\right) + \sum_{k=1}^{C} \frac{\partial}{\partial z_i^{[l]}}\left(y_k \cdot log\left(\sum_{j=1}^{C} e^{z_j^{[l]}}\right)\right) \tag{8}$$

Veáse que en el primer término de la derivada nos hemos quedado únicamente con el término del sumatorio original que depende del parámetro respecto al cual estamos derivando. Centrándonos en dicho término, vemos que

$$-\frac{\partial}{\partial z_i^{[l]}}\left(y_i \cdot log\left(e^{z_i^{[l]}}\right)\right) = -\frac{\partial}{\partial z_i^{[l]}}\left(y_i z_i^{[l]}\right) = -y_i \tag{9}$$

Por otra parte,

$$\sum_{k=1}^{C}\frac{\partial}{\partial z_i^{[l]}}\left(y_k \cdot log\left(\sum_{j=1}^{C} e^{z_j^{[l]}}\right)\right) = \sum_{k=1}^{C} y_k \frac{e^{z_i^{[l]}}}{\sum_{j=1}^{C} e^{z_j^{[l]}}} = \frac{e^{z_i^{[l]}}}{\sum_{j=1}^{C} e^{z_j^{[l]}}}\sum_{k=1}^{C} y_k \tag{10}$$

Hemos de tener en cuenta que las etiquetas de las muestras vienen dadas en codificación one-hot, y por tanto $\sum_k y_k = 1$. Bastaría así con aplicar esta propiedad junto con la igualdad (5) para obtener que

$$\sum_{k=1}^{C}\frac{\partial}{\partial z_i^{[l]}}\left(y_k \cdot log\left(\sum_{j=1}^{C} e^{z_j^{[l]}}\right)\right) = A_i^{[l]} \tag{11}$$

Podemos ahora sustituir los resultados (9) y (11) en la expresión (8), de forma que

$$\frac{\partial \mathcal{L}}{\partial z_i^{[l]}} = A_i^{[l]} - y_i \tag{12}$$

Expresado en forma matricial,

$$\frac{\partial \mathcal{L}}{\partial z^{[l]}} = A^{[l]} - Y \tag{13}$$

Teniendo esto claro, uno podría calcular las derivadas del error con respecto a los pesos de la última capa con tan solo aplicar la regla de la cadena, donde se tiene que, por definición de $z^{[l]}$,

$$\frac{\partial \mathcal{L}}{\partial w_{ji}^{[l]}} = \frac{\partial \mathcal{L}}{\partial z_i^{[l]}}\frac{\partial z_i^{[l]}}{\partial w_{ji}^{[l]}} \tag{14}$$

Teniendo en cuenta la ecuación (1), vemos que para una capa $k \in \{1, ..., l\}$,

$$\frac{\partial z_i^{[k]}}{\partial w_{ji}^{[k]}} = \frac{\partial}{\partial w_{ji}^{[k]}}\left(w_i^{[k]} \cdot A^{[k-1]} + b^{[k]}\right) = \frac{\partial}{\partial w_{ji}^{[k]}}\left(w_{ji}^{[k]} A_j^{[k-1]}\right) = A_j^{[k-1]} \tag{15}$$

Concluimos así que para un peso $w_{ji}^{[l]}$, la deriva de la función de error con respecto a dicho peso no es más que

$$\frac{\partial \mathcal{L}}{\partial w_{ji}^{[l]}} = \frac{\partial \mathcal{L}}{\partial z_i^{[l]}} \frac{\partial z_i^{[l]}}{\partial w_{ji}^{[l]}} = \left(A_i^{[l]} - y_i \right) A_j^{[l-1]} \tag{16}$$

y, para toda capa $k \in \{1, ..., l\}$,

$$\frac{\partial \mathcal{L}}{\partial b_i^{[k]}} = \frac{\partial \mathcal{L}}{\partial z_i^{[k]}} \tag{17}$$

por lo que

$$\frac{\partial \mathcal{L}}{\partial b_i^{[l]}} = \left(A_i^{[l]} - y_i \right) \tag{18}$$

3.1. Derivadas para parámetros en capas ocultas

Veamos como se comportan las derivadas para la última capa oculta. Comenzaremos derivando con respecto a $z_i^{[l-1]}$ para luego aplicar la regla de la cadena y obtener la derivada respecto a cada uno de sus pesos. Teniendo en cuenta que estamos tratando de derivar la ecuación (6), que consiste en un sumatorio de C términos todos dependientes de $z_i^{[l-1]}$, por tratarse de una red FC, la derivada consistirá en un sumatorio, por la propiedad de la derivada de la suma de funciones, con las derivadas parciales con respecto a $z_i^{[l-1]}$ para cada uno de sus sumandos. Esto es,

$$\frac{\partial \mathcal{L}}{\partial z_i^{[l-1]}} = \sum_{n_l=1}^{C} \frac{\partial \mathcal{L}}{\partial A_{n_l}^{[l]}} \frac{\partial A_{n_l}^{[l]}}{\partial z_{n_l}^{[l]}} \frac{\partial z_{n_l}^{[l]}}{\partial A_i^{[l-1]}} \frac{\partial A_i^{[l-1]}}{\partial z_i^{[l-1]}} \tag{19}$$

Para simplificar los cálculos, supondremos que todas las capas ocultas de la red utilizan una función de activación sigmoide, $\sigma : \mathbb{R} \to \mathbb{R}$, la cual es derivable $\forall x \in \mathbb{R}$ y cumple que $\sigma'(x) = \sigma(x)(1 - \sigma(x))$. Gracias a esta suposición, sea $k \in \{1, ..., l-1\}$, sabemos que

$$A_i^{[k]} = \sigma(z_i^{[k]}) \tag{20}$$

Y, consecuentemente,

$$\frac{\partial A_i^{[k]}}{\partial z_i^{[k]}} = \sigma(z_i^{[k]}) \left(1 - \sigma(z_i^{[k]})\right) = A_i^{[k]} \left(1 - A_i^{[k]}\right) \tag{21}$$

Por tanto, al sustituir los resultados obtenidos en las ecuaciones (12) y (21) en la expresión (16), y observar que la derivada parcial de $z_{n_l}^{[l]}$ con respecto a $A_i^{[l-1]}$ no es más que su coeficiente $w_{i,n_l}^{[l]}$, obtenemos la siguiente expresión

$$\frac{\partial \mathcal{L}}{\partial z_i^{[l-1]}} = \sum_{n_l=1}^{C} \left(A_{n_l}^{[l]} - y_{n_l} \right) w_{i,n_l}^{[l]} A_i^{[l-1]} \left(1 - A_i^{[l-1]}\right) \tag{22}$$

Si ahora queremos la derivada con respecto a los pesos de una neurona i de la capa $l-1$, obtenemos, aplicando (14), que

$$\frac{\partial \mathcal{L}}{\partial w_{ji}^{[l-1]}} = \sum_{n_l=1}^{C} \left(A_{n_l}^{[l]} - y_{n_l}\right) w_{i,n_l}^{[l]} A_i^{[l-1]} \left(1 - A_i^{[l-1]}\right) A_j^{[l-2]} \tag{23}$$

El conjunto MNIST puede ser clasificado con un alto accuracy con una única capa oculta, por lo que ya tendríamos los cálculos necesarios para desarrollar el algoritmo. No obstante, para generalizar este algoritmo a cualquier red FC, necesitamos saber que ocurre con los pesos más allá de la penúltima capa.

Veamos así que sucede para la capa $l-2$. Para ello, hemos de tener en cuenta que, dado un peso $w_{ji}^{[l-2]}$, su valor únicamente pondera a la salida de la neurona j de la capa $l-3$, como entrada a la neurona i de la capa $l-2$. Ahora bien, al tratarse de una red neuronal FC, todas las neuronas de la capa $l-1$ estarán conectadas con todas las de la capa $l-2$. Por tanto, el peso $w_{ji}^{[l-2]}$ afectará a todas las salidas de esta $l-1$. Es por esta razón por la que aparece un nuevo sumatorio en la expresión, ya que todas las $z_i^{[l]}$ consisten en una suma ponderada de las distintas coordenadas del vector $A^{[l-1]}$, las cuales dependen todas de dicho peso y, por tanto, actúara la regla de la derivada para la suma de funciones sobre ellas. Esto es,

$$\frac{\partial}{\partial w_{ji}^{[l-2]}} \left(w_i^{[l]} \cdot A^{[l-1]}\right) = \sum_{n_{l-1}=1}^{n(l-1)} \frac{\partial}{\partial w_{ji}^{[l-2]}} \left(w_{n_{l-1},i}^{[l]} A_{n_{l-1}}^{[l-1]}\right) \tag{24}$$

Así, cobra más sentido el siguiente resultado

$$\frac{\partial \mathcal{L}}{\partial z_i^{[l-2]}} = \sum_{n_l=1}^{C} \frac{\partial \mathcal{L}}{\partial A_{n_l}^{[l]}} \frac{\partial A_{n_l}^{[l]}}{\partial z_{n_l}^{[l]}} \sum_{n_{l-1}=1}^{n(l-1)} \frac{\partial z_{n_l}^{[l]}}{\partial A_{n_{l-1}}^{[l-1]}} \frac{\partial A_{n_{l-1}}^{[l-1]}}{\partial z_{n_{l-1}}^{[l-1]}} \frac{\partial z_{n_{l-1}}^{[l-1]}}{\partial A_i^{[l-2]}} \frac{\partial A_i^{[l-2]}}{\partial z_i^{[l-2]}} \tag{25}$$

pues no es más que

$$\frac{\partial \mathcal{L}}{\partial z_i^{[l-2]}} = \sum_{n_l=1}^{C} \frac{\partial \mathcal{L}}{\partial A_{n_l}^{[l]}} \frac{\partial A_{n_l}^{[l]}}{\partial z_{n_l}^{[l]}} \sum_{n_{l-1}=1}^{n(l-1)} \frac{\partial}{\partial w_{ji}^{[l-2]}} \left(w_{n_{l-1},i}^{[l]} A_{n_{l-1}}^{[l-1]}\right) \tag{26}$$

3.2. Generalización de las derivadas

Uno podría seguir calculando las derivadas para capas inferiores, y observaría un patrón que se repite. Existirían $l-k$ sumatorios anidados en la expresión, uno por cada capa comprendida entre la capa k que se esté evaluando y la de salida, incluyendo a esta. Los $l-k-1$ sumatorios correspondientes a las capas $\{l, l-1, ..., k+2\}$ computarían la derivada del cálculo lineal que hace cada una de las neuronas de dicha capa con respecto a la salida de la capa anterior, multiplicado por su sumatorio. Esto, en conjunto, supone la derivada de la función de error con respecto al cálculo lineal realizado en la capa $k+1$, tras aplicar sucesivas veces la regla de la cadena. Es decir,

$$S_{l:(k+2)} = \sum_{j=1}^{C} \frac{\partial \mathcal{L}}{\partial A_j^{[l]}} \frac{\partial A_j^{[l]}}{\partial z_j^{[l]}} \sum_{n_1=1}^{n(l-1)} \frac{\partial z_j^{[l]}}{\partial A_{n_1}^{[l-1]}} \frac{\partial A_{n_1}^{[l-1]}}{\partial z_{n_1}^{[l-1]}} \cdots \sum_{n_{k+2}=1}^{n(k+2)} \frac{\partial z_{n_{k+3}}^{[k+3]}}{\partial A_{n_{k+2}}^{[k+2]}} \frac{\partial A_{n_{k+2}}^{[k+2]}}{\partial z_{n_{k+2}}^{[k+2]}} \tag{27}$$

El sumatorio de la capa $k+1$ calcula la derivada de cada $z_{n_{k+2}}^{[k+2]}$ con respecto al $z_i^{[k]}$ que utiliza el peso $w_{ji}^{[k]}$ que nos intersea, que se reduce a lo siguiente

$$\frac{\partial \mathcal{L}}{\partial z_i^{[k]}} = S_{l:(k+2)} \cdot \sum_{n_{k+1}=1}^{n(k+1)} \frac{\partial z_{n_{k+2}}^{[k+2]}}{\partial A_{n_{k+1}}^{[k+1]}} \frac{\partial A_{n_{k+1}}^{[k+1]}}{\partial z_{n_{k+1}}^{[k+1]}} \frac{\partial z_{n_{k+1}}^{[k+1]}}{\partial A_i^{[k]}} \frac{\partial A_i^{[k]}}{\partial z_i^{[k]}} \tag{28}$$

Finalmente, sea la k la capa en la que se encuentra un peso respecto al cual queremos calcular la derivada, tenemos que

$$\frac{\partial \mathcal{L}}{\partial z_i^{[k]}} = \sum_{j=1}^{C} \frac{\partial \mathcal{L}}{\partial A_j^{[l]}} \frac{\partial A_j^{[l]}}{\partial z_j^{[l]}} \sum_{n_1=1}^{n(l-1)} \left(\cdots \sum_{n_{k+1}=1}^{n(k+1)} \frac{\partial z_{n_{k+2}}^{[k+2]}}{\partial A_{n_{k+1}}^{[k+1]}} \frac{\partial A_{n_{k+1}}^{[k+1]}}{\partial z_{n_{k+1}}^{[k+1]}} \frac{\partial z_{n_{k+1}}^{[k+1]}}{\partial A_i^{[k]}} \frac{\partial A_i^{[k]}}{\partial z_i^{[k]}} \right) \tag{29}$$

Y, consecuentemente, la derivada del error con respecto a dicho peso no es más que

$$\frac{\partial \mathcal{L}}{\partial w_{ji}^{[k]}} = \sum_{j=1}^{C} \frac{\partial \mathcal{L}}{\partial A_j^{[l]}} \frac{\partial A_j^{[l]}}{\partial z_j^{[l]}} \sum_{n_1=1}^{n(l-1)} \left(\cdots \sum_{n_{k+1}=1}^{n(k+1)} \frac{\partial z_{n_{k+2}}^{[k+2]}}{\partial A_{n_{k+1}}^{[k+1]}} \frac{\partial A_{n_{k+1}}^{[k+1]}}{\partial z_{n_{k+1}}^{[k+1]}} \frac{\partial z_{n_{k+1}}^{[k+1]}}{\partial A_i^{[k]}} \frac{\partial A_i^{[k]}}{\partial z_i^{[k]}} A_j^{[k-1]} \right) \tag{30}$$

4. Algoritmo de Back-Propagation

El algoritmo de retropropagación es fundamental en el aprendizaje supervisado de las redes neuronales, pues desencadena un proceso esencial para la optimización de los parámetros de la red. Al abordar la complejidad inherente de la optimización, la retropropagación calcula las derivadas parciales de la función de pérdida con respecto a cada peso y sesgo en la red neuronal. Este cálculo preciso de las derivadas se lleva a cabo, generalmente, mediante el uso de la notación matricial, un enfoque que simplifica la representación y manipulación de las operaciones matemáticas implicadas, ya desarrolladas en la sección 3.

4.1. Notación matricial para el Back-Propagation

A la hora de implementar estas derivadas para que puedan calcularse de forma sencilla en un algoritmo de aprendizaje, se hace impresindible buscar una manera de emcapsular los cálculos en alguna estructura que nos permita reducir la notación planteada. Para los pesos de la primera capa, por ejemplo, cuyas derivadas vienen expresadas de forma genérica en la expresión (16), uno tenría que tener en cuenta las ixj combinaciones de subíndices posibles, a fin de calcular todas las derivadas necesarias para la corrección de errores en la capa de salida. Esto podría resolverse cómodamente haciendo uso del producto de matrices.

Por ser la matriz de pesos de la capa de salida, $W^{[l]}$, de dimensión $n(l)$ x $n(l-1)$, tenemos que $\frac{\partial \mathcal{L}}{\partial W^{[l]}}$, que no es más que la derivada parcial del error con respecto a dicha matriz, es también una matriz de orden $n(l)$ x $n(l-1)$. En ella, cada entrada d_{ji} nos indica la derivada del error con respecto al peso $w_{ji}^{[l]}$.

$$\frac{\partial \mathcal{L}}{\partial W^{[l]}} = \begin{pmatrix} \frac{\partial \mathcal{L}}{\partial w_{11}^{[l]}} & \frac{\partial \mathcal{L}}{\partial w_{12}^{[l]}} & \cdots & \frac{\partial \mathcal{L}}{\partial w_{1,n(l-1)}^{[l]}} \\ \frac{\partial \mathcal{L}}{\partial w_{21}^{[l]}} & \frac{\partial \mathcal{L}}{\partial w_{22}^{[l]}} & \cdots & \frac{\partial \mathcal{L}}{\partial w_{2,n(l-1)}^{[l]}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial \mathcal{L}}{\partial w_{n(l),1}^{[l]}} & \frac{\partial \mathcal{L}}{\partial w_{n(l),2}^{[l]}} & \cdots & \frac{\partial \mathcal{L}}{\partial w_{n(l),n(l-1)}^{[l]}} \end{pmatrix} \tag{31}$$

Aplicando las propiedades de matrices, y apoyándonos en la ecuación (16), podemos expresar dicha derivada en la siguiente forma matricial

$$\frac{\partial \mathcal{L}}{\partial W^{[l]}} = \begin{pmatrix} \frac{\partial \mathcal{L}}{\partial z_1^{[l]}} \\ \frac{\partial \mathcal{L}}{\partial z_2^{[l]}} \\ \vdots \\ \frac{\partial \mathcal{L}}{\partial z_{n(l)}^{[l]}} \end{pmatrix} \cdot \begin{pmatrix} A_1^{[l-1]} & A_2^{[l-1]} & \cdots & A_{n(l-1)}^{[l-1]} \end{pmatrix} = \frac{\partial \mathcal{L}}{\partial z^{[l]}} \cdot (A^{[l-1]})^t \tag{32}$$

Hemos conseguido no solo reducir la notación, sino simplificar la implementación del cálculo, lo cual se hace imprescindible para capas más profundas. Veamos que ocurre para la penúltima capa del modelo. Recordemos la expresión de la derivada del error respecto a sus pesos, dada en la ecuación (23)

$$\frac{\partial \mathcal{L}}{\partial w_{ji}^{[l-1]}} = \sum_{n_l=1}^{C} \left(A_{n_l}^{[l]} - y_{n_l}\right) w_{i,n_l}^{[l]} A_i^{[l-1]} \left(1 - A_i^{[l-1]}\right) A_j^{[l-2]}$$

Podemos reordenar el segundo miembro, extrayendo del sumatorio aquellos términos que no dependen de su índice. Así,

$$\frac{\partial \mathcal{L}}{\partial w_{ji}^{[l-1]}} = A_i^{[l-1]} \left(1 - A_i^{[l-1]}\right) A_j^{[l-2]} \sum_{n_l=1}^{C} \left(A_{n_l}^{[l]} - y_{n_l}\right) w_{i,n_l}^{[l]} \tag{33}$$

Y nos bastaría con hacer uso del producto matricial para llegar al siguiente resultado

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial w_{ji}^{[l-1]}} &= A_i^{[l-1]} \left(1 - A_i^{[l-1]}\right) A_j^{[l-2]} w_i^{[l]} \left(A^{[l]} - y\right) \\ &= A_i^{[l-1]} \left(1 - A_i^{[l-1]}\right) w_i^{[l]} \left(A^{[l]} - y\right) A_j^{[l-2]} \end{aligned} \tag{34}$$

Aplicando la misma estrategia seguida para los pesos de la primera capa, obtenemos la siguiente forma matricial para la derivada con respecto a los pesos de la capa $l-1$

$$\frac{\partial \mathcal{L}}{\partial W^{[l-1]}} = \left(\left(A^{[l-1]} \odot \left(1 - A^{[l-1]}\right)\right) \odot \left(W^{[l]}\right) \left(A^{[l]} - y\right)\right) \left(A^{[l-2]}\right)^t \tag{35}$$

y

$$\frac{\partial \mathcal{L}}{\partial b^{[l-1]}} = \left(\left(A^{[l-1]} \odot \left(1 - A^{[l-1]}\right)\right) \odot \left(W^{[l]}\right) \left(A^{[l]} - y\right)\right) \tag{36}$$

De forma análoga, es posible representar matricialmente las derivadas para el resto de capas ocultas. Así, uno podría apreciar que, una vez calculada la derivada $\frac{\partial \mathcal{L}}{\partial z^{[k]}}$ para una capa k, nos bastaría con antender la ecuación (30),

$$\frac{\partial \mathcal{L}}{\partial w_{ji}^{[k]}} = \sum_{j=1}^{C} \frac{\partial \mathcal{L}}{\partial A_j^{[l]}} \frac{\partial A_j^{[l]}}{\partial z_j^{[l]}} \sum_{n_1=1}^{n(l-1)} \left(\cdots \sum_{n_{k+1}=1}^{n(k+1)} \frac{\partial z_{n_{k+2}}^{[k+2]}}{\partial A_{n_{k+1}}^{[k+1]}} \frac{\partial A_{n_{k+1}}^{[k+1]}}{\partial z_{n_{k+1}}^{[k+1]}} \frac{\partial z_{n_{k+1}}^{[k+1]}}{\partial A_i^{[k]}} \frac{\partial A_i^{[k]}}{\partial z_i^{[k]}} A_j^{[k-1]} \right) \tag{37}$$

para ver que, al aplicar nuevamente la regla de la cadena,

$$\frac{\partial \mathcal{L}}{\partial z^{[k-1]}} = \begin{pmatrix} \frac{\partial \mathcal{L}}{\partial z_1^{[k-1]}} \\ \frac{\partial \mathcal{L}}{\partial z_2^{[k-1]}} \\ \vdots \\ \frac{\partial \mathcal{L}}{\partial z_{n(l)}^{[k-1]}} \end{pmatrix} = \begin{pmatrix} \sum_{n_k=1}^{n(k)} \frac{\partial \mathcal{L}}{\partial z_{n_k}^{[k]}} \frac{\partial z_{n_k}^{[k]}}{\partial A_1^{[k-1]}} \frac{\partial A_1^{[k-1]}}{\partial z_1^{[k-1]}} \\ \sum_{n_k=1}^{n(k)} \frac{\partial \mathcal{L}}{\partial z_{n_k}^{[k]}} \frac{\partial z_{n_k}^{[k]}}{\partial A_2^{[k-1]}} \frac{\partial A_2^{[k-1]}}{\partial z_2^{[k-1]}} \\ \vdots \\ \sum_{n_k=1}^{n(k)} \frac{\partial \mathcal{L}}{\partial z_{n_k}^{[k]}} \frac{\partial z_{n_k}^{[k]}}{\partial A_{n(k-1)}^{[k-1]}} \frac{\partial A_{n(k-1)}^{[k-1]}}{\partial z_{n(k-1)}^{[k-1]}} \end{pmatrix} \Rightarrow \tag{38}$$

$$\Rightarrow \frac{\partial \mathcal{L}}{\partial z^{[k-1]}} = \left(\frac{\partial z^{[k]}}{\partial A^{[k-1]}} \odot \frac{\partial A^{[k-1]}}{\partial z^{[k-1]}}\right) \cdot \frac{\partial \mathcal{L}}{\partial z^{[k]}} \Rightarrow \tag{39}$$

$$\Rightarrow \frac{\partial \mathcal{L}}{\partial z^{[k-1]}} = \left(\left(W^{[k]}\right)^t \odot \frac{\partial A^{[k-1]}}{\partial z^{[k-1]}}\right) \cdot \frac{\partial \mathcal{L}}{\partial z^{[k]}} \tag{40}$$

Y, por tanto,

$$\frac{\partial \mathcal{L}}{\partial W^{[k-1]}} = \frac{\partial \mathcal{L}}{\partial z^{[k-1]}} \cdot \frac{\partial z^{[k-1]}}{\partial W^{[k-1]}} \tag{41}$$

Hemos obtenido, por tanto, una manera iterativa de obtener las derivadas para una capa $k-1$ a partir de los resultados obtenidos en la capa k. Dicho método iterativo de presenta formalmente en la siguiente sección, de forma algorítmica.

4.2. Implementación del algoritmo

Implementaremos a continuación el algoritmo que se acaba de describir para una red neuronal de $l-1$ capas ocultas y una de salida, aunque con apenas una capa ocualta resulta ser suficiente para la clasificación de regiones convexas en el espacio. Dicho algoritmo , de ahora en adelante llamado back-propagation, únicamente necesitará calcular las derivadas dadas por las ecuaciones (32), (33), (38) y (39). Diseñaremos el algoritmo de tal forma que únicamente calcule las derivadas, sin realizar ni una modificación sobre la red. La ventaja principal que nos ofrece esta implementación es que uno podría aplicar cualquiera de los métodos basados en descenso por el gradiente que existan, ya sea bien el descenso por el gradiente estocástisco, SGD, o el mismo Adam. En la implementación realizada en Python, se presentan varias alternativas, cuyos rendimientos se mostrarán más adelante.

Algorithm 1: Algoritmo de Back-Propagation

Data: Muestras y etiqueta para entrenar $X \in \mathbb{R}^n$, $Y \in \{0,1\}^C$

Result: Derivadas $\frac{\partial \mathcal{L}}{\partial W^{[l]}}, \frac{\partial \mathcal{L}}{\partial b^{[l]}}, ..., \frac{\partial \mathcal{L}}{\partial W^{[1]}}, \frac{\partial \mathcal{L}}{\partial b^{[1]}}$ en (X, Y)

$A^{[l]} \leftarrow$ Resultado de la red para la muestra X ;

$\frac{\partial \mathcal{L}}{\partial z^{[l]}} \leftarrow A^{[l]} - Y$;

$\frac{\partial \mathcal{L}}{\partial b^{[l]}} \leftarrow \frac{\partial \mathcal{L}}{\partial b^{[l]}}$;

$\frac{\partial \mathcal{L}}{\partial W^{[l]}} \leftarrow \frac{\partial \mathcal{L}}{\partial z^{[l]}} \cdot \left(A^{[l-1]}\right)^t$;

for $k \in \{l-1, ..., 1\}$ **do**

  $\frac{\partial \mathcal{L}}{\partial z^{[k]}} \leftarrow \frac{\partial \mathcal{L}}{\partial z^{[k]}} \odot \left(\frac{\partial z^{[k]}}{\partial A^{[k-1]}} \cdot \frac{\partial A^{[k-1]}}{\partial z^{[k-1]}}\right)$;

  $\frac{\partial \mathcal{L}}{\partial b^{[k]}} \leftarrow \frac{\partial \mathcal{L}}{\partial z^{[k]}}$;

  $\frac{\partial \mathcal{L}}{\partial W^{[k]}} \leftarrow \frac{\partial \mathcal{L}}{\partial z^{[k]}} \cdot \left(A^{[k-1]}\right)^t$;

return *Lista con las derivadas* $\frac{\partial \mathcal{L}}{\partial W^{[l]}}, \frac{\partial \mathcal{L}}{\partial b^{[l]}}, ..., \frac{\partial \mathcal{L}}{\partial W^{[1]}}, \frac{\partial \mathcal{L}}{\partial b^{[1]}}$

Ahora bien, este algoritmo es únicamente válido para redes de clasificación. Si por ejemplo quisiera ajustarse para que pueda adaptarse a problemas de optimización de otro tipo de redes, como por ejemplo los autoencoders, tenemos el resultado genérico dado en el algoritmo 2, que generaliza la derivada del error.

5. Implementación de la red neuronal en Python

En la implementación en Python hemos decidido seguir el paradigma de programación orientada a objetos con el objetivo de desarrollar un código que siga los principios de código limpio, de forma que facilite su mejora para la siguiente entrega. En su arquitectura, la clase ‘SequentialLayer‘ encapsula los componentes esenciales de una capa en la red neuronal. Sus atributos comprenden el número de neuronas de dicha capa (units), las matrices de pesos (weights) y los sesgos de cada neurona (bias), así como funciones de activación y sus deriva-

Algorithm 2: Algoritmo de Back-Propagation

Data: Muestras y etiqueta para entrenar $X \in \mathbb{R}^n$, $Y \in \mathbb{R}^m$

Result: Derivadas $\frac{\partial \mathcal{L}}{\partial W^{[l]}}$, $\frac{\partial \mathcal{L}}{\partial b^{[l]}}$, ..., $\frac{\partial \mathcal{L}}{\partial W^{[1]}}$, $\frac{\partial \mathcal{L}}{\partial b^{[1]}}$ en (X, Y)

$\frac{\partial \mathcal{L}}{\partial z^{[l]}} \leftarrow \frac{\partial \mathcal{L}}{\partial A^{[l]}} \cdot \frac{\partial A^{[l]}}{\partial z^{[l]}}$;

$\frac{\partial \mathcal{L}}{\partial b^{[l]}} \leftarrow \frac{\partial \mathcal{L}}{\partial b^{[l]}}$;

$\frac{\partial \mathcal{L}}{\partial W^{[l]}} \leftarrow \frac{\partial \mathcal{L}}{\partial z^{[l]}} \cdot \left(A^{[l-1]}\right)^t$;

for $k \in \{l-1, ..., 1\}$ **do**

  $\frac{\partial \mathcal{L}}{\partial z^{[k]}} \leftarrow \frac{\partial \mathcal{L}}{\partial z^{[k]}} \odot \left(\frac{\partial z^{[k]}}{\partial A^{[k-1]}} \cdot \frac{\partial A^{[k-1]}}{\partial z^{[k-1]}}\right)$;

  $\frac{\partial \mathcal{L}}{\partial b^{[k]}} \leftarrow \frac{\partial \mathcal{L}}{\partial z^{[k]}}$;

  $\frac{\partial \mathcal{L}}{\partial W^{[k]}} \leftarrow \frac{\partial \mathcal{L}}{\partial z^{[k]}} \cdot \left(A^{[k-1]}\right)^t$;

return *Lista con las derivadas* $\frac{\partial \mathcal{L}}{\partial W^{[l]}}, \frac{\partial \mathcal{L}}{\partial b^{[l]}}, ..., \frac{\partial \mathcal{L}}{\partial W^{[1]}}, \frac{\partial \mathcal{L}}{\partial b^{[1]}}$

das (activation y activation derivative). Estos elementos definen la estructura y comportamiento de cada capa.

Por otra parte, la clase 'NeuralNetwork' administra la conexión y operación de estas capas. Su método 'set(layer)' agrega capas a la red. La función 'feedforward(X, n)' realiza la propagación hacia adelante, aplicando sucesivamente las funciones de activación de cada capa hasta la capa n (o la última capa por defecto). El método 'backpropagation(X, y, lr)' implementa la retropropagación para actualizar los pesos y sesgos de la red.

5.1. Optimizadores

En esta implementación se incluye una serie de funciones encargadas única y exclusivamente de optimizar nuestro modelo, haciendo uso del backpropagation que ofrece la clase 'NeuralNetwork', y que aplican algunos de los métodos de optimización ya vistos, como son: descenso del gradiente, descenso del gradiente con momento, y adam. Su estructura algorítmica podría resumirse en lo siguiente.

Algorithm 3: Algoritmo base para optimizadores

Data: Una instancia de la clase NeuralNetwork, modelo, un conjunto de entrenamiento, $L = \{(x, y) : x \in \mathbb{R}^n, y \in \{0,1\}^C\}$, un factor de aprendizaje, α, y un número de épocas, epochs

for *cada epoch* $\in [1, epochs]$ **do**

  for *cada* $(x, y) \in L$ **do**

    AlgoritmoDeOptimizacion(modelo,x,y,α);

    $\alpha \leftarrow$ ActualizacionLearningRate(α);

6. Experimentos

Procedemos a continuación a evaluar el rendimiento del código implementado, utilizando dos datasets hitos en el mundo del aprendizaje automático.

6.1. Conjuntos de datos para la evaluación

- El conjunto de datos Iris contiene muestras de tres especies de flores Iris: Setosa, Versicolor y Virginica. Cada muestra está compuesta por cuatro características numéricas:
  - **Longitud** del sépalo (en centímetros)
  - **Anchura** del sépalo (en centímetros)
  - **Longitud** del pétalo (en centímetros)
  - **Anchura** del pétalo (en centímetros)
- El conjunto de datos MNIST consiste en imágenes de dígitos escritos a mano, del 0 al 9. Cada imagen es de tamaño 28x28 píxeles y, para su procesamiento en un RN, se necesita aplanar cada imagen en un vector de 784 elementos.
- El conjunto de datos 'load_breast_cancer' de 'scikit-learn' contiene información sobre características extraídas de imágenes digitalizadas de biopsias mamarias. Cada muestra tiene 30 atributos numéricos que describen diversas características de los núcleos celulares presentes en las imágenes, como el radio, la textura, la simetría, la suavidad y otros atributos relacionados con la forma y tamaño de las células. La variable a predecir en este conjunto de datos corresponde a la naturaleza del tumor: benigno o maligno, representado por los valores 0 y 1 respectivamente. Es un conjunto de datos comúnmente utilizado para tareas de clasificación binaria, donde se intenta predecir si un tumor mamario es maligno o no en función de estas características derivadas de las imágenes de biopsias.

6.2. Evaluación del rendimiento

En el primer experimento realizado, se entrenaron tres modelos idénticos de una red neuronal utilizando el conjunto de datos Iris. Cada modelo consta de 4 entradas, una capa oculta de 10 neuronas con una función de activación sigmoide y una capa de salida de clasificación con 3 neuronas correspondientes a las diferentes especies de flores Iris. Los modelos se entrenaron con tres optimizadores distintos: el descenso del gradiente clásico, el descenso del gradiente con momento y el optimizador Adam. Durante el entrenamiento, se registraron las medidas de precisión (accuracy) obtenidas para evaluar el desempeño de cada modelo. Se llevaron a cabo múltiples iteraciones para ajustar los pesos y minimizar la función de pérdida con el objetivo de mejorar la capacidad predictiva de los modelos. Los resultados obtenidos proporcionan información sobre la eficacia y la convergencia de los modelos bajo cada optimizador, lo que permite comparar y analizar su desempeño en la tarea de clasificación de las especies de flores Iris.

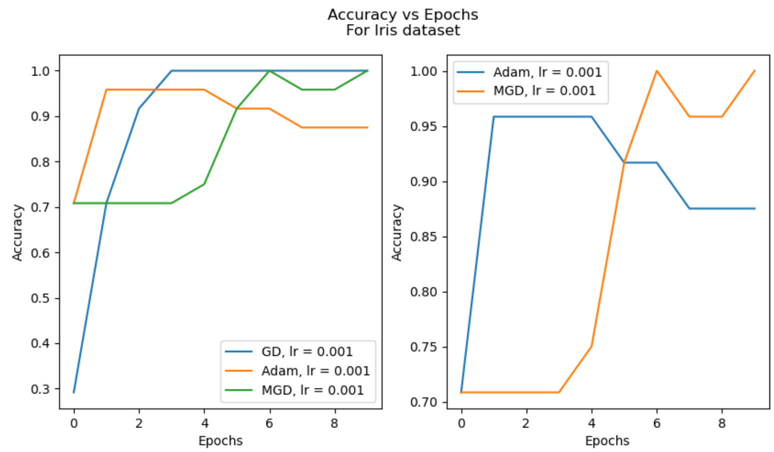

Figura 1: Accuracy en la clasificación del dataset Iris.

Presentamos, además, los resultados obtenidos para una red de 748 entradas, una capa oculta de 100 neuronas con función de activación sigmoide, y 10 salidas, con el objetivo de clasificar el dataset MNIST.

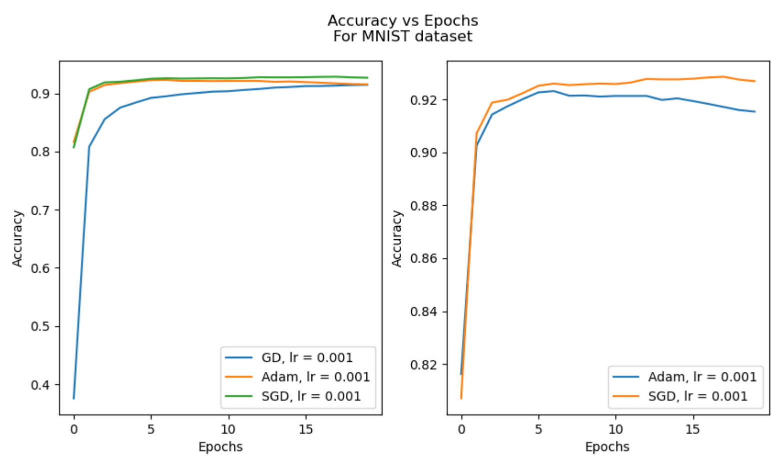

Figura 2: Accuracy en la clasificación del dataset MNIST.

Podemos observar como, para ambos modelos y conjuntos de datos, los optimizadores adam y descenso del gradiente con momento alcanzan un mayor

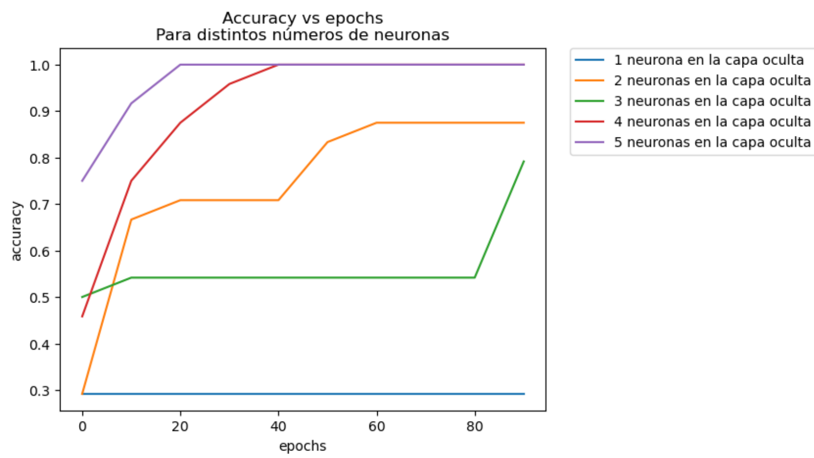

Figura 3: Accuracy para Iris con distintos número de neuronas.

accuracy en una menor cantidad de tiempo, algo que se apreciaría más notablemente al tomar una frecuencia de muestreo mayor en las distinta épocas. No obstante, el algoritmo del descenso por el gradiente clásico también consigue llegar al resultado deseado. Podemos además ver como varía el accuracy a medida que vamos aumentando el número de neuronas en la capa oculta de nuestra red. Lo probamos principalmente para el dataset Iris, obteniendo el siguiente gráfico.

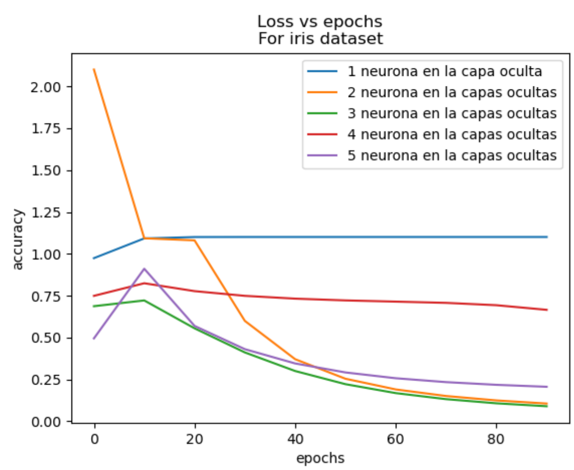

Figura 4: Error para Iris con distintos número de neuronas.

Podemos ver, además, la disminución del error para los distintas número de neuronas en dicha capa, donde se ha tomado la función de error por defecto: Cross-Entropy.

Por último, vemos que ocurre para la clasificación del Breast Cancer dataset, esta vez mostrando los resultados en el espacio ROC. Para esto, hemos desarrollado 16 modelos, los cuales tienen todos 3 capas ocultas de 20, 10 y 5 neuronas, respectivamente. Lo que se ha hecho así ha sido variar las funciones de activación que actúan sobre las capaz de los diversos modelos, además de recurrir a distintos optimizadorer: descenso del gradiente, descenso del gradiente con momento y Adam. Ver tablas y matrices en el apéndice (TA1, FA1).

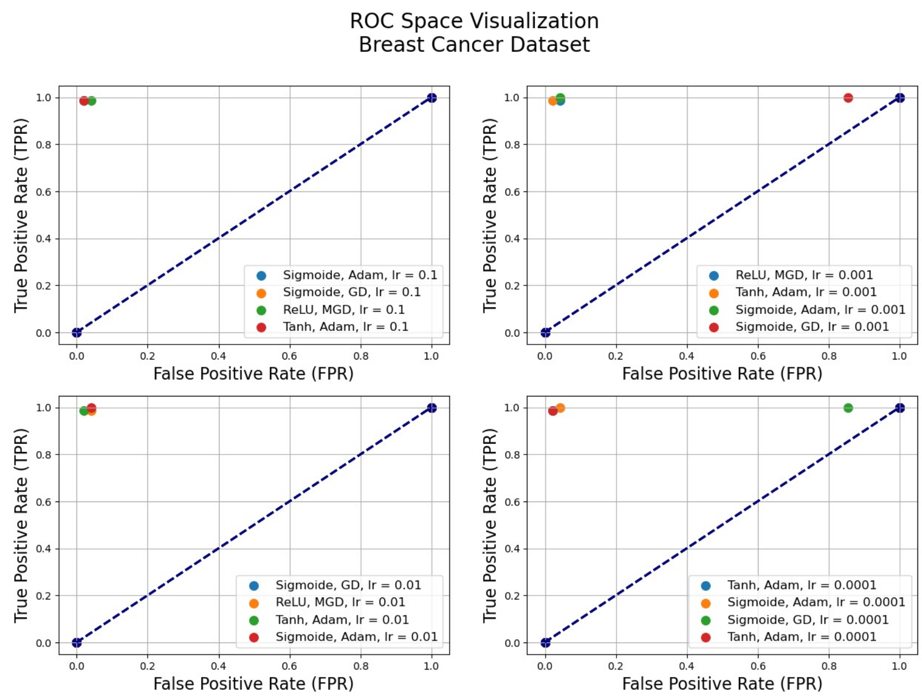

Figura 5: Espacio ROC para modelos de clasificación de Cáncer de Mama

7. Introducción a la IA generativa

En esta investigación, hemos abordado la implementación de redes neuronales, un componente central en el desarrollo de tecnologías vanguardistas como Chat-GPT, fundamentado en transformers. Aunque nuestro énfasis se ha centrado en la clasificación, el enfoque predominante en Inteligencia Artificial (IA) radica en la creación de contenido inédito. La IA generativa, opuesta a la clasificación, se dedica a generar datos originales utilizando arquitecturas como las Redes Generativas Adversarias (GAN) y las Redes Neuronales Recurrentes (RNN). En contraste con el enfoque de clasificación, la IA generativa emplea funciones de error alternativas al cross-entropy, como el Error Cuadrático Medio

(MSE). Esta función evalúa las discrepancias entre los valores predichos y reales, esencial en regresiones continuas que permiten pronosticar valores numéricos en lugar de categorizar. La diferenciación entre enfoques pone de relieve el papel clave de la IA generativa en la creación de nuevos datos y su uso en contextos diversos, comparado con la tarea tradicional de clasificación.

7.1. Conjuntos de datos para la evaluación

- El conjunto de datos **Quality of Wine** contiene información sobre diversas características químicas de muestras de vino tinto y blanco. Estas características incluyen, entre otras, la acidez fija, volatilidad de ácidos, ácido cítrico, contenido de azúcar residual, cloruros y contenido de alcohol. Estos datos pueden ser relevantes para modelos de regresión, donde se busca predecir la calidad del vino, una variable objetivo, utilizando un conjunto de características mencionadas.
- **Life Expectancy**, recopilado por la OMS, proporciona datos sobre la esperanza de vida y factores de salud de 193 países entre 2000 y 2015. Se tomaron en cuenta factores relevantes de salud y economía. Entre sus variables, destacan la esperanza de vida, además características clave como la mortalidad infantil, el PIB, los gastos en salud, etc.

.

7.2. Problemas de regresión

La aplicación de redes neuronales en problemas de regresión es esencial para la aproximación de funciones no lineales. En su forma más simple, un modelo de regresión lineal busca establecer una relación lineal entre las variables de entrada $x_1, x_2, \ldots, x_n$ y la variable de salida y mediante la fórmula:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_n x_n + \varepsilon$$

Aquí, y representa la variable dependiente, $x_1, x_2, \ldots, x_n$ son las variables independientes, $\beta_0, \beta_1, \ldots, \beta_n$ son los coeficientes y ε es el término de error. El objetivo es minimizar la suma de los cuadrados de las diferencias entre las observaciones reales y las predicciones del modelo, lo que se formaliza a través del Error Cuadrático Medio (MSE):

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

Donde y_i son los valores reales, $\hat{y}_i$ son las predicciones del modelo para cada observación, y n es el número total. De esta manera, uno podría pensar en una red neuronal como un interpolador, capaz de adaptarse a cualquier función. A continuación, mostramos la curva de aprendizaje de un modelo con tres capas ocultas de 20 neuronas cada una y una única capa de salida, la cual tiene como objetivo servir de regresor a un problema relacionado con la predicción de la calidad del vino tinto en función de una serie de características del mismo. Para ello, se han utilizado funciones de activación sigmoide en todas las capas salvo en la de salida, en la cual se ha aplicado una mera función identidad, pues los valores de la regresión pueden variar en todo el conjunto de los números reales.

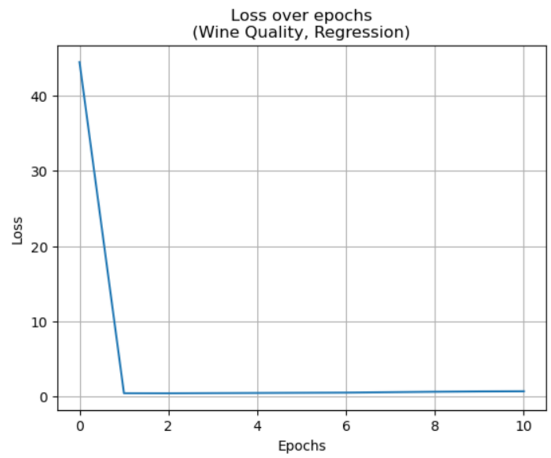

Figura 6: Error para la regresión del dataset del vino.

Como podemos apreciar, en solo una época el modelo es capaz de adaptarse a los datos a la perfección. Ahora, probaremos un modelo que entraña una mayor complejidad, y se basa en la regresión sobre la esperanza de vida de personas en función de ciertas constantes. Se tratará de un modelo que dispone de 19 variables de entrada, y dispondrá de una única capa oculta.

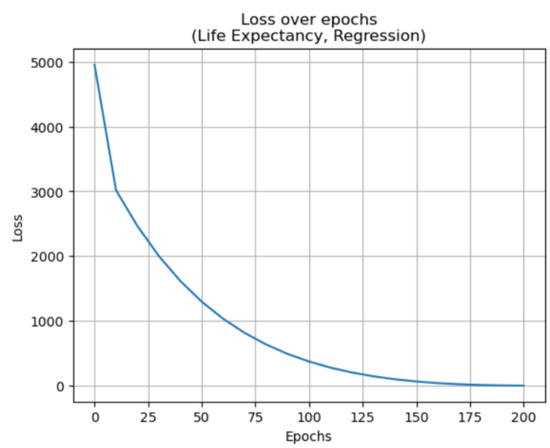

Figura 7: Error para la regresión del dataset de life exp.

El modelo parece ajustarse perfectamente a los datos de entrenamiento. Ahora bien, podríamos mejorar esto incrementanto la complejidad del modelo. Veamos: probaremos para dos modelos más, uno con dos capas ocultas densas con 10 neuronas cada uno y otro con tres capas ocultas, cuyo número de neuronas irá decrementando secuencialmente. Al realizar los entrenamientos, obtenemos los resultados mostrados en la Figura 8.

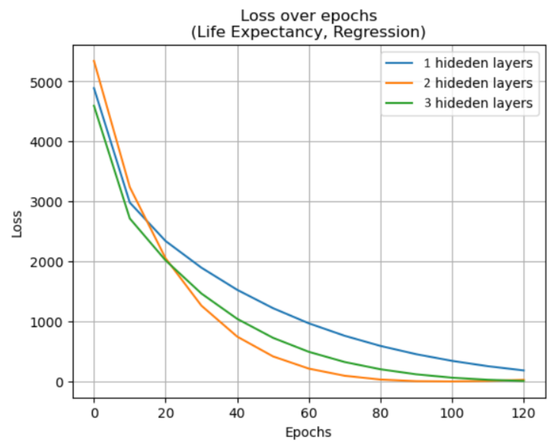

Figura 8: Error para la regresión con distintas capas.

Uno podría comprobar como al aumentarse en una capa al modelo, este parece entrenarse más rápidamente. No obstante, incrementar excesivamente su complejidad daría lugar a over-fitting. De hecho, esto comienza a ocurrir para el último modelo presentado con tres capas ocultas. Además, lo que se ha hecho en la gráfica anterior es dejar el error a cero una vez en el entrenamiento se alcanza dicho valor de rendimiento, pero eso no es lo que ocurre realmente.

Como puede apreciar el lector, basta con fijarnos en las últimas épocas para ver como una vez alcanzado un mínimo error, este empieza a aumentar. Esto se debe a los errores de sesgo intrínsecos al modelo y, en consecuencia, al overfitting. El primer modelo, el cual entraña una menor complejidad, es el único en el que parece no ocurrir este fenómeno.

7.3. AutoEncoders

Los autoencoders son una clase de redes neuronales que se utilizan principalmente para tareas de aprendizaje no supervisado, especialmente en la compresión de datos y la extracción de características relevantes. Su arquitectura consta de una red neuronal dividida en dos partes: un codificador, encargado de transformar los datos de entrada en una representación codificada de menor dimensión, y un decodificador que intenta reconstruir la entrada original a partir de esta representación reducida. Esta capacidad de compresión y reconstrucción hace que los autoencoders sean valiosos en diversas aplicaciones,

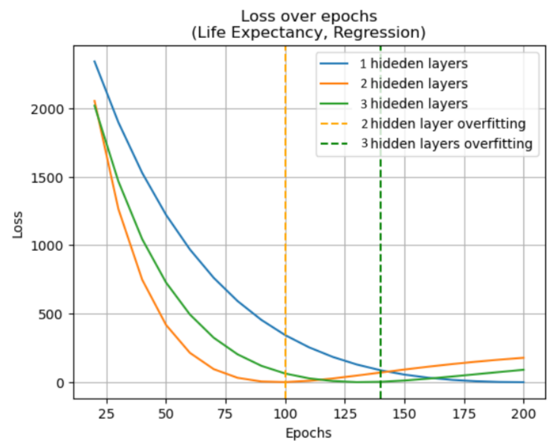

Figura 9: Overfitting en la regresión.

desde la reducción de dimensionalidad en conjuntos de datos complejos hasta la generación de nuevas muestras o la eliminación de ruido en imágenes, señales o datos en general. Su versatilidad y capacidad para aprender representaciones significativas de los datos, incluso en ausencia de etiquetas, los convierten en herramientas fundamentales en campos como el procesamiento de imágenes, la detección de anomalías, la recomendación de contenido y la síntesis de datos.

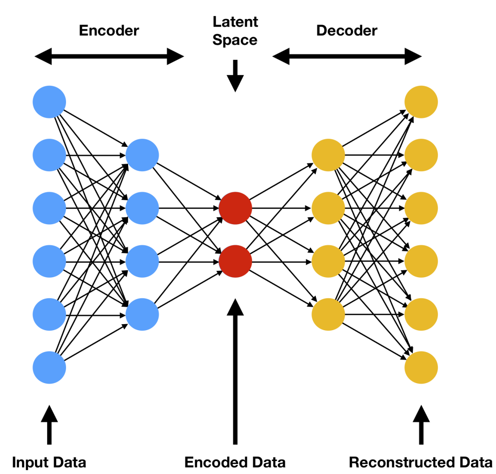

Figura 10: Arquitectura del autoencoder

Matemáticamente, los autoencoders buscan aprender una función $f(x)$ que mapea los datos de entrada x a sí mismos, pasando por una representación intermedia de menor dimensión h. Al espacio al que pertenecen estas representaciones lo llamamos espacio latente. Este proceso se puede dividir en dos etapas: la fase de codificación y la fase de decodificación. En la fase de codificación, la red neuronal busca encontrar una representación h mediante la transformación de los datos de entrada x a través de una función $h = \text{Codificador}(x)$. Posteriormente, en la fase de decodificación, el objetivo es reconstruir la entrada original x a partir de la representación h utilizando una función $r = \text{Decodificador}(h)$, buscando minimizar la pérdida entre la entrada reconstruida y la entrada original.

Para este proyecto, nuestro objetivo ha sido desarrollar un autoencoder para el conjunto de datos mnist, y con tan solo 200 épocas, hemos conseguido el increíble resultado que se muestra en la figura 10.

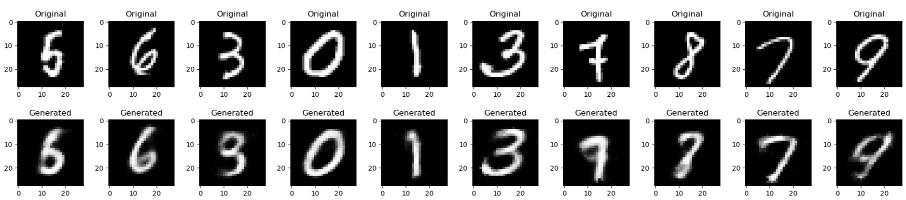

Figura 11: Reconstrucción del MNIST con AE

Esto se ha conseguido para un autoencoder que cuenta con apenas 9 neuronas para su espacio latente. En sí, la arquitectura consta de 4 capas, de 784, 128, 9, 128 y 784 neuronas cada una de ellas, incluyéndo las de entrada y las de salida. Una clara prueba de que el modelo ha entrenado correctamente la vemos al comparar con las imágenes generadas con en la primera época de todas, donde los resultados dejan bastante que desear (ver figura 12). En el repositorio de GitHub se pueda ver el proceso de entrenamiento completo mediante un conjunto de imágenes generadas para cada una de las épocas. Claramente, en el código fuente puede ver más en detalle como se ha implementado esto en código.

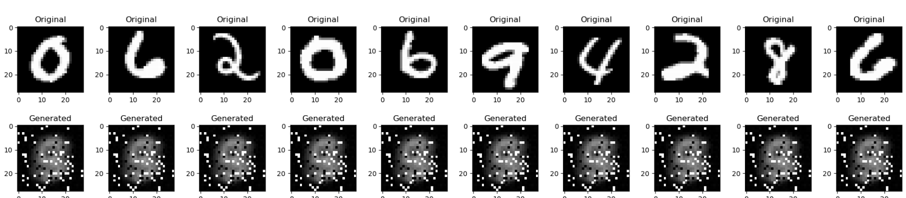

Figura 12: Reconstrucción del MNIST con AE, epoch 0

7.4. Reducción de ruido en imágenes

La reducción de ruido es crucial en la preservación de la calidad y la utilidad de los datos en diversas áreas, como el procesamiento de imágenes, señales y datos en general. Los Autoencoders (AE) son herramientas poderosas para abordar este desafío al aprender a reconstruir datos limpios a partir de versiones ruidosas. Utilizan una arquitectura de red neuronal para comprimir la información, luego reconstruyen la entrada original, filtrando y eliminando el ruido en el proceso. Específicamente en el procesamiento de imágenes, los AE pueden restaurar detalles dañados o afectados por ruido, mejorando significativamente la calidad visual. Esta capacidad de los AE para recuperar información relevante a partir de datos corruptos o ruidosos los convierte en una solución efectiva para tareas donde la integridad y la precisión de los datos son esenciales, brindando una herramienta valiosa para la mejora de datos en diversos dominios.

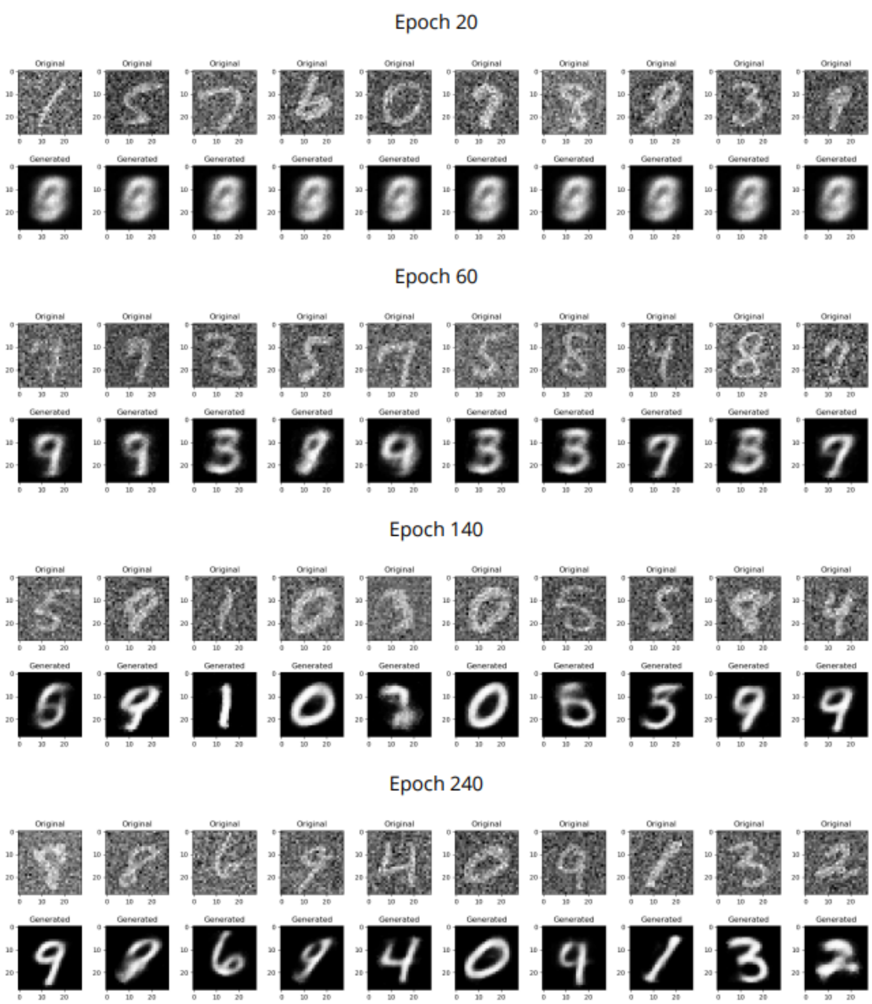

Figura 13: Reduccióon de ruido del MNIST con AE

En la figura 13, presentamos los resultados obtenidos para un modelo de reducción de ruido para el dataset MNIST para imágenes de validación, es decir, imágenes con las que el modelo no ha sido entrenado. La arquitecutra consiste en un autoencoder de tres capas ocultas, con 128, 9 y 128 neuronas, respectivamente. Para el entrenamiento, desarrollamos una función que le añadía ruido gausiano a las imágenes, y luego medíamos el error comparando los resultados con las imágenes reales del dataset. Se utilizó un learning rate de 0.01 y un descenso del gradiente clásico.

Que el modelo funcione tan bien nos informa de la precisión del código desarrollado. El principal cambio con respecto a la entrega anterior es la posibilidad de desarrollar modelos con un mayor número de capas ocultas, pues la implementación previa tan solo estaba pensada para una de estas capas. Así, un autoencoder como los desarrollados ahora funcionando de esta manera indica que los pesos para cada una de las capas ocultas están corrigiéndose debidamente, por lo que estamos realmente satisfechos con nuestro trabajo.

8. Conclusión

Como se ha podido apreciar a lo largo de este trabajo, el algoritmo de backpropagation es una herramienta indispensable a la hora de optimizar ciertos modelos en el ámbito del aprendizaje automático. La Inteligencia Artificial (IA) ha experimentado un impulso significativo gracias al avance de la retropropagación, un pilar fundamental en el funcionamiento de las redes neuronales que hemos implementado, y nos han permitido obtener una precisión de hasta el 100 % en la predicción de flores Iris, o un 92 % en la clasificación de imágenes de números manuscritos. Este algoritmo desempeña un papel crucial al optimizar modelos no solo en el ámbito de la clasificación, o incluso de la regresión, sino también en la innovadora área de los autoencoders. La capacidad de ajustar con precisión los parámetros de la red mediante la regla de la cadena ha llevado a un aumento sustancial en la capacidad predictiva y en la generación de representaciones significativas a partir de los datos.

Además, la aplicación de este algoritmo ha permitido el desarrollo de la IA generativa, la cual ha abierto nuevos horizontes en la creación de contenido original. El surgimiento de redes generativas como las Redes Generativas Adversarias (GAN) y las Redes Neuronales Recurrentes (RNN) ha revolucionado la capacidad de las máquinas para crear imágenes, música y texto auténticos. Estos modelos han trascendido la mera clasificación, adentrándose en la generación de información novedosa y abriendo posibilidades en la creatividad artificial. Este avance no solo ha mejorado la precisión de los modelos, sino que también ha ampliado el alcance de la IA, explorando nuevos dominios en la producción de contenido genuino.

Future Work

Para la siguiente entrega, el objetivo principal será el paso del back-propagation clásico a una implementación basada en la autodiferenciación, donde hemos comenzado a dar los primeros pasos en la branch 'convolutional-layers', del repositorio de nuestro proyecto. Además, queremos implementar capas convolutivas y, así, mejorar la capacidad de los autoencoders ya desarrollados o, incluso, ser capaces de distinguir dígitos manuscritos incluso estando rotados.

Referencias

[1] Cárdenes Pérez, Ricardo J. (2023) Neural Network Python Implementation
https://github.com/ricardocardn/NeuralNetworkImplementation

[2] Santana Esplá, Carlos (2023) Neural Network Matlab Implementation
https://github.com/carlos-esplaa/Backpropagation

[3] Life Expectancy Dataset, Kaggle
https://www.kaggle.com/datasets/kumarajarshi/life-expectancy-who

[4] Wine Quality Dataset, Kaggle
https://www.kaggle.com/datasets/yasserh/wine-quality-dataset

[5] Javier García (2016) Redes Neuronales
https://www.youtube.com/watch?v=XVmbDxZ-HLI&list=PLAnA8FVrBl8AWkZmbswwWiF8a_52dQ3JQ&index=4

[6] Goodfellow, Ian and Bengio, Yoshua and Courville, Aaron (2016) Deep Learning
MIT press

APÉNDICE

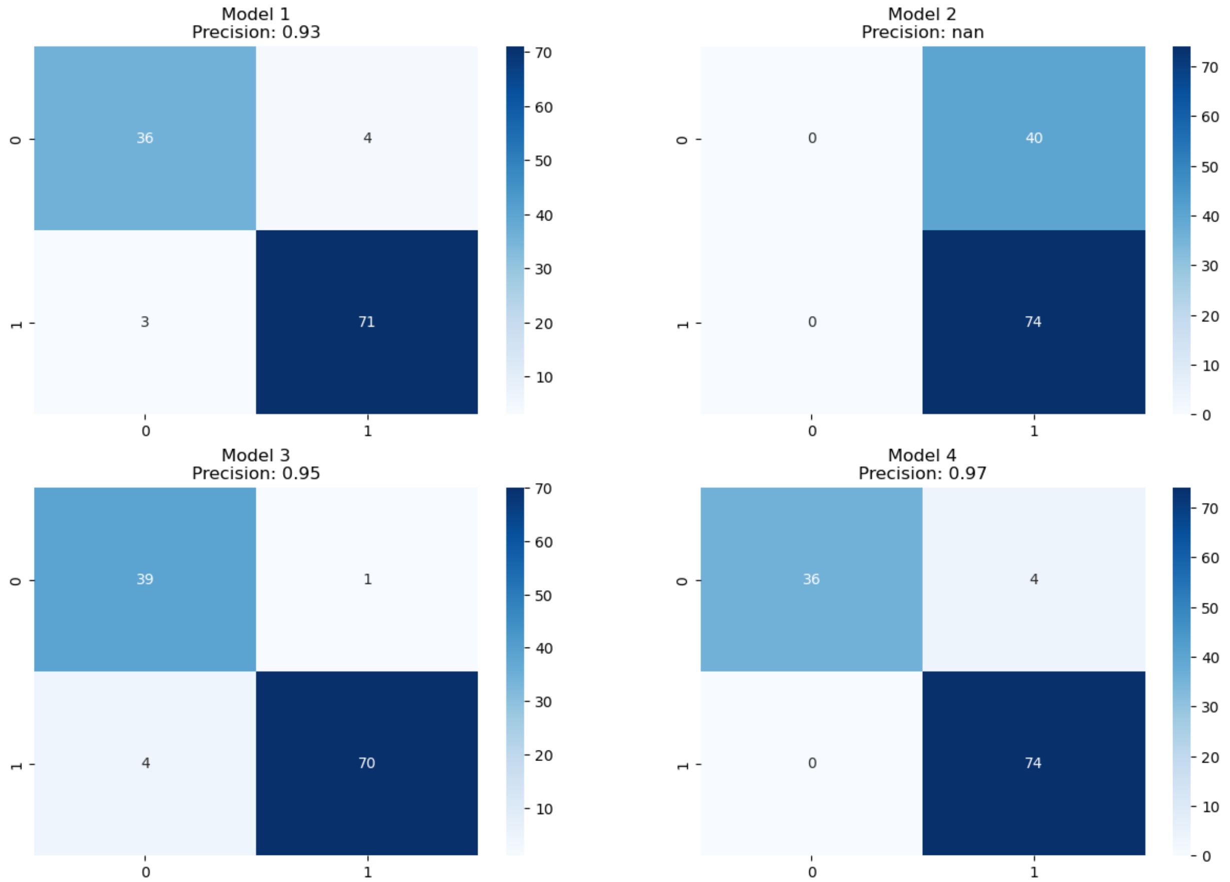

Figura A1: Matrices de confusión para modelos con lr=0.01

para la detección de cáncer de mama

Modelo 1: Sigmoide, Descenso del Gradiente
Modelo 2: ReLU, Descenso del Gradiente con momento
Modelo 3: Tangente Hiperbólica, Adam
Modelo 4: Sigmoide, Adam

Modelo	Accuracy	Precision	TPR	AUC
Modelo1	0.94	0.93	1	0.93
Modelo2	0.65	1	0	0.5
Modelo3	0.96	0.95	1	0.96
Modelo4	0.96	0.97	1	0.95

Tabla A1: Tabla de resultados para modelos con lr=0.01

para la detección de cáncer de mama

Modelo 1: Sigmoide, Descenso del Gradiente
Modelo 2: ReLU, Descenso del Gradiente con momento
Modelo 3: Tangente Hiperbólica, Adam
Modelo 4: Sigmoide, Adam

(*) Nota al lector: El propio lector puede obtener las medidas de rendimiento para distintos learning rate e incluso arquitecturas modificando el notebook de test que se incluye en el repositorio del código fuente [1], dado en las referencias de este artículo.