Implementación de Descenso por Gradiente Estocástico y Comparación de Funciones de Coste y Activación en Redes Neuronales Artificiales

Proyecto Optimización 1

Amelia Hoyos Vélez*[1], Valentina Vásquez George†[1] y Luisa Fernanda Ciro Restrepo‡[1]

[1] *Ingeniería Matemática, Universidad EAFIT, Medellín, Colombia*

May 21, 2024

1 Introducción

En numerosas ocasiones se ha observado la necesidad de determinar si un fenómeno pertenece a una categoría específica dentro de una lista de posibilidades; por ejemplo, determinar si un animal es una nutria, un delfín o una foca, o si un tumor es benigno o maligno. Esta tarea, conocida como clasificación, desempeña un papel fundamental en diversas disciplinas y su automatización ha demostrado ser de gran utilidad. En el ámbito de la inteligencia artificial, se han desarrollado diversos métodos para llevar a cabo esta tarea, entre los cuales las redes neuronales artificiales destacan.

Una red neuronal recibe valores numéricos como entradas $x_1, \ldots, x_n$, que es la información que se tiene respecto a cierto fenómeno, y produce unas salidas $y_1, \ldots, y_n$, que es el resultado de la clasificación, funcionando conceptualmente como una función vectorial matemática. Sin embargo, tiene una diferencia fundamental con respecto a las funciones convencionales. Mientras que estas últimas operan mediante la aplicación de reglas definidas, las neuronas artificiales tienen la capacidad de aprender y adaptarse a partir de los datos que se tiene.

*ahoyosv2@eafit.edu.co
†vvasquezg1@eafit.edu.co
‡lfciror@eafit.edu.co

Para comprender el funcionamiento de las redes neuronales, es esencial entender primero el proceso que lleva a cabo una neurona individual (Figura 1). Como se mencionó anteriormente, una neurona recibe un conjunto de entradas $x_1, \ldots, x_n$. Para cada una de estas entradas, existe un parámetro conocido como *peso*. Cada entrada se multiplica por su peso correspondiente y luego se suman todos estos términos. A esta suma se le agrega un parámetro adicional llamado *valor umbral*, resultando en lo que se conoce como la entrada ponderada, expresada por la fórmula:

$$z = \sum_{i=0}^{n} (w_i x_i) + b \tag{1}$$

Finalmente, esta entrada ponderada se somete a una función de activación f, cuyo resultado, $\hat{y}$, se determina por:

$$\hat{y} = f(z) \tag{2}$$

Cuando la tarea de una red neuronal es de clasificación, la función de activación suele generar valores en un rango entre cero y uno. Si el resultado de la función de activación es cercano a 1, la neurona está indicando que es altamente probable que el fenómeno representado por los valores de entrada pertenezca a la categoría deseada. Por el contrario, si el resultado es cercano a cero, la neurona nos está in-

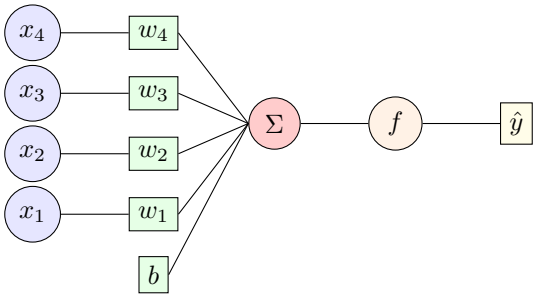

Figura 1: Diagrama de una neurona artificial

dicando que es poco probable que el fenómeno pertenezca a la categoría. En este contexto, el valor $\hat{y}$ que produce la función de activación representa de alguna manera una medida de probabilidad de que el fenómeno en cuestión pertenezca a la categoría objetivo.

En muchas ocasiones, las tareas de predicción no se realizan utilizando neuronas individuales, sino combinando varias capas de estas neuronas en un modelo (Figura 2). Este modelo suele estar compuesto por una capa de entrada, compuesta por tantas neuronas como valores de entrada se tenga, que está totalmente conectada a otra capa interna. Esta capa interna está compuesta por neuronas que toman todos los valores de la capa de entrada y actúan de forma independiente como se muestra en la Figura 1. Estas neuronas producen valores, llamados valores de activación, que a su vez son utilizados como valores de entrada para capas siguientes. Este proceso se repite cuantas veces sea necesario para modelar la complejidad de los datos. Las capas intermedias entre la capa de entrada y la capa de salida se denominan capas ocultas, ya que para un usuario que utiliza este modelo de predicción, estas capas son una *caja negra* cuyo funcionamiento interno no es directamente visible.

La última capa en la red neuronal se conoce como la capa de salida. Esta capa tiene tantas neuronas como categorías se desean predecir. Cada neurona en la capa de salida representa una categoría específica. Para que un fenómeno se prediga como perteneciente a una categoría particular, la neurona correspondiente a esa categoría tendrá un valor de salida cercano a uno, lo que representa una alta probabilidad de pertenencia a dicha categoría. El resto de las neuronas de salida tendrán valores cercanos a cero, indicando una baja probabilidad de pertenencia a esas categorías.

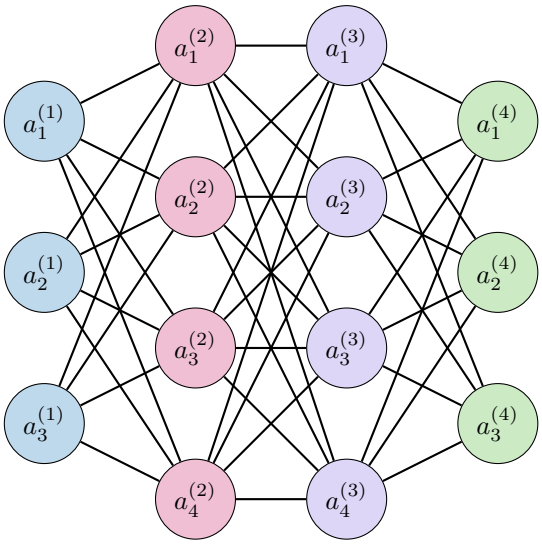

Figura 2: Arquitectura de una red neuronal

Cada elemento de la red neuronal se representa de la siguiente manera:

- El valor de activación de cada neurona se denota como $a_i^{(k)}$, donde k indica la capa y i la posición de la neurona dentro de la capa. En particular, $a_i^{(1)}$ corresponde a la entrada x_i y $a_i^{(n)}$ al valor de salida $\hat{y}_i$.

- Los pesos se representan como $w_{ji}^{(k)}$, donde el subíndice ji indica la conexión desde la neurona i en la capa $k-1$ hacia la neurona j en la capa k.

- Los valores umbral se denotan como $b_i^{(k)}$, representando el valor umbral correspondiente de la neurona i en la capa k.

Para simplificar las operaciones de la red neuronal, organizamos estos términos en vectores y matrices. Esta notación es equivalente a la presentada en las ecuaciones 1 y 2, pero se aplica a una capa completa de la red neuronal.

La operación general de una capa k se expresa mediante la siguiente fórmula de forma matricial:

$$\mathbf{a}^{(k)} = f\left(\mathbf{W}^{(k)}\mathbf{a}^{(k-1)} + \mathbf{b}^{(k)}\right) \tag{3}$$

Donde cada término se define como:

$$\mathbf{a}^{(k)} = \begin{pmatrix} a_1^{(k)} & a_2^{(k)} & \cdots & a_m^{(k)} \end{pmatrix}^\top$$

$$\mathbf{W}^{(k)} = \begin{pmatrix} w_{1,1}^{(k)} & w_{1,2}^{(k)} & \cdots & w_{1,n}^{(k)} \\ w_{2,1}^{(k)} & w_{2,2}^{(k)} & \cdots & w_{2,n}^{(k)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m,1}^{(k)} & w_{m,2}^{(k)} & \cdots & w_{m,n}^{(k)} \end{pmatrix}$$

$$\mathbf{b}^{(k)} = \begin{pmatrix} b_1^{(k)} & b_2^{(k)} & \cdots & b_m^{(k)} \end{pmatrix}^\top$$

Donde $\mathbf{a}^{(k)}$ es el vector de activaciones de la capa k de tamaño $m \times 1$, $\mathbf{a}^{(k-1)}$ es el vector de activaciones de la capa $k-1$ de tamaño $n \times 1$, $\mathbf{W}^{(k)}$ es la matriz de pesos de la capa k de tamaño $m \times n$, y $\mathbf{b}^{(k)}$ es el vector de valores umbral de la capa k de tamaño $m \times 1$. f es la función de activación que se aplica elemento por elemento al resultado de la suma ponderada.

2 Datos

Hemos descrito cómo funciona una red neuronal ya entrenada, es decir, una red que predice la categoría de un fenómeno correctamente la mayor parte de las veces. Sin embargo, aún no se ha explicado la ventaja de las redes neuronales frente a otros enfoques de modelado. Una de las principales ventajas radica en su capacidad para aprender patrones complejos y no lineales a partir de los datos disponibles.

Para lograr esto, se utiliza un enfoque de aprendizaje supervisado, en el que disponemos de datos de varios sucesos clasificados correctamente. Para cada suceso s, contamos con un conjunto de características $x_{1s}, ..., x_{ns}$ que describen el suceso, y su categoría correspondiente, llamada etiqueta, y_s. De esta manera, formamos el conjunto de datos D:

$$D = \{(x_{1s}, ..., x_{ns}, y_s) : s = 1, ..., S\}$$

Es decir, hemos invertido el problema. Ahora, los valores de entrada son parámetros del modelo y tenemos valores de salida deseados. Nuestro objetivo es encontrar los valores de $\mathbf{W}^{(k)}$ y $\mathbf{b}^{(k)}$ para cada capa del modelo que nos den las mejores predicciones.

2.1 Problema

Para llevar a cabo nuestra investigación, decidimos desarrollar la implementación de una red neuronal en Python. Optamos por abordar un problema simple de clasificación por tres razones principales. En primer lugar, la implementación de nuestra red neuronal, al no utilizar ninguna librería optimizada para el aprendizaje automático, no es del todo rápida. Además, teníamos recursos computacionales limitados a nuestra disposición. Por último, deseábamos utilizar una base de datos de fácil acceso para garantizar la reproducibilidad de los resultados.

El problema por el que nos decidimos es el de clasificar una imagen que representa un dígito del cero al nueve en alguna de las diez categorías. Cada imagen es de 8×8 bits, por lo que habrá 64 valores de entrada (ver Figura 3).

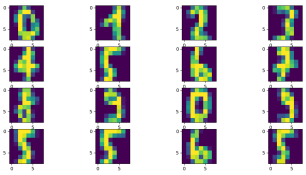

Figura 3: Imágenes de muestra

La base de datos *The Digit Dataset* la tomamos de *scikit-learn datasets* [1]. Tiene en total 1797 imágenes. Se dividió en los siguientes conjuntos de entrenamiento, prueba y validación (ver Tabla 1 y Figura 4):

	Test	Train	Validation
Datos	1078	360	359
Proporción	0.6	0.2	0.2

Tabla 1: Cantidad de datos en el conjunto de entrenamiento, prueba y validación

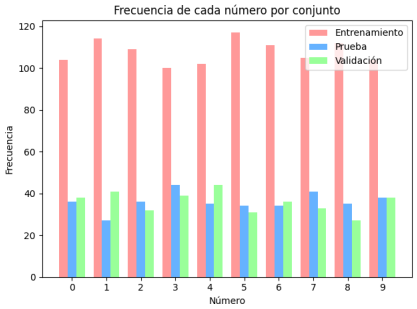

Figura 4: Frecuencia de cada número por conjunto

3 Planteamiento matemático

Ahora, ¿cómo denotamos matemáticamente el objetivo de tener buenas predicciones? Aquí es donde entra lo que se denomina la función de costo $J(\hat{\mathbf{y}})$ que va a depender de las predicciones hechas por el modelo. Nótese que $\hat{y}$ depende a su vez de los pesos y valores umbrales, por lo que también se denota como $J(\mathbf{W}, \mathbf{b})$. Las siguientes son funciones comunes de costo, para predicciones $\hat{y}_i$, valores objetivos y_i, y n número de categorías a predecir.

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

$$\text{CE} = -\frac{1}{n}\sum_{i=1}^{n}\left[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)\right]$$

Donde cada una de ellas representa:

- **MSE**: Error cuadrático medio, mide el error cuadrático medio de las diferencias entre las predicciones y los valores verdaderos.
- **CE**: Entropía cruzada, sirve como una forma de cuantificar la diferencia entre la distribución de probabilidad o modelo deseado y la distribución actual.

Más adelante veremos cual de estas funciones de costo se desempeña mejor para nuestro problema en específico. Por otro lado, es pertinente conocer las distintas funciones de activación que probaremos.

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

$$\text{ReLU}(x) = \max(0, x)$$

$$\text{Tanh}(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

- **Función Logística Sigmoide**: Produce una salida en un rango entre cero y uno.
- **ReLU**: Eficiente computacionalmente. Cero para entradas negativas, lineal para entradas positivas.
- **Tangente Hiperbólica**: Produce una salida en un rango entre menos uno y uno.

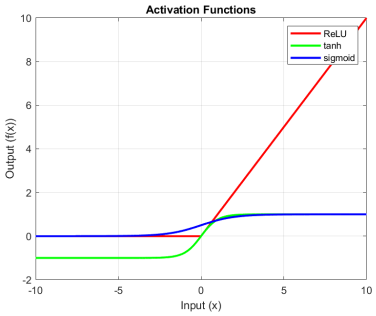

Figura 5: Frecuencia de cada número por conjunto

Podemos plantear el siguiente problema de optimización.

Siendo las variables de decisión los pesos $w_{ji}^{(k)}$, los valores umbrales $b_i^{(k)}$, los valores de activación $a_{is}^{(k)}$ para cada dato de entrenamiento, y las predicciones $\hat{y_{is}}$ para cada dato de entrenamiento con la notación descrita anteriormente, trataremos de minimizar la función de coste:

$$\sum_{s=1}^{S} J(\hat{\mathbf{y}}_s)$$

Es decir, nuestro problema es minimizar el error de todos los datos de prueba, sujeto a las restricciones de que, para cada ejemplo de entrenamiento s y para cada característica de los ejemplos de entrenamiento r, las activaciones de la primera capa son iguales a las características de entrada:

$$a_{rs}^{(1)} = x_{rs}$$

Para cada ejemplo de entrenamiento s, capa oculta k, y neurona i en la capa, siendo n_k la cantidad de neuronas en una capa,

$$a_{is}^{(k)} = f\left(\sum_{m=1}^{n_{k-1}} a_{ms}^{(k-1)} \cdot w_{im}^{(k)} + b_i^{(k)}\right)$$

Para cada ejemplo de entrenamiento s y predicción de salida j,

$$\hat{y}_{js} = f\left(\sum_{m=1}^{n_{L-1}} a_{ms}^{(L-1)} \cdot w_{jm}^{(L)} + b_j^{(L)}\right)$$

Siendo L la última capa.

Este problema de optimización busca ajustar los pesos y valores umbrales de la red neuronal para minimizar la función de pérdida $J(\hat{y})$ sobre los datos de entrenamiento. Nótese que el problema no es lineal. Aunque la función de coste y la función de activación bien pudieran ser lineales, las restricciones contienen la multiplicación de las propias variables de decisión, ya que cada valor de activación es conjuntamente proporcional a los valores de activación previos y a sus pesos. Por lo tanto, no se puede realizar un enfoque de optimización lineal. Por el contrario, se han desarrollado técnicas para el entrenamiento de redes neuronales, entre ellas el método por descenso de gradiente estocástico.

El método del gradiente descendente busca hallar un mínimo local en la función de costo ajustando gradualmente los valores de los pesos y los umbrales en la dirección de máximo decremento. Se sabe que el gradiente de una función indica la dirección de máximo incremento en su espacio dimensional correspondiente, por lo que una manera de encontrar un mínimo es avanzar en pequeños pasos en la dirección opuesta al gradiente hasta que se encuentre un punto suficientemente cercano al mínimo.

La magnitud de estos pasos está determinada por un hiperparámetro del modelo α llamado tasa de aprendizaje. Si este es muy bajo, el algoritmo puede tardar mucho en encontrar un mínimo. Si es muy alto, la solución puede no converger. Es importante notar que este algoritmo no siempre nos lleva a un mínimo global, sino que nos conduce a un mínimo local.

$$
\begin{aligned}
&\text{repetir } N \text{ veces } \{ \\
&\quad w_{ji}^{(k)} := w_{ji}^{(k)} - \alpha \tfrac{\partial}{\partial w_{ji}^{(k)}} J(\mathbf{W}, \mathbf{b})) \\
&\quad b_{i}^{(k)} := b_{i}^{(k)} - \alpha \tfrac{\partial}{\partial b_{i}^{(k)}} J(\mathbf{W}, \mathbf{b})) \\
&\}
\end{aligned}
\tag{4}
$$

Cada peso y umbral se actualiza restando α multiplicado por la derivada de la función de costo con respecto a ese parámetro. Esta derivada indica la rapidez con la que cambia la función de costo cuando se modifica ese parámetro. Si el cambio es grande, se restará una cantidad proporcionalmente mayor; si es pequeño, se restará menos.

Computar el gradiente en cada paso para una función de costo que dependa de todos los datos de entrenamiento es computacionalmente demandante. Aquí es donde entra en juego la parte estocástica del algoritmo. En lugar de utilizar todo el conjunto de datos de entrenamiento en cada iteración para actualizar los pesos y valores umbrales, como se hace en el gradiente descendente tradicional, el gradiente descendente estocástico selecciona aleatoriamente una pequeña muestra de datos en cada iteración.

Esta selección aleatoria de subconjuntos permite que el algoritmo sea más rápido y eficiente computacionalmente, además de introducir algo de aleatoriedad al modelo, lo que posibilita una exploración más exhaustiva del espacio de soluciones. Esto puede evitar que el algoritmo caiga en un mínimo local subóptimo.

Ahora surge otro problema: ¿cómo se calculan estos gradientes? Recordemos que las predicciones dependen de pesos que se multiplican con los valores de activación de la capa anterior, y a su vez, estos valores de activación dependen de sus propios pesos y umbrales. Calcular este gradiente a mano sería tedioso debido a la cantidad de reglas de la cadena necesarias para encontrarlo. Por lo tanto, se ha desarrollado una manera de encontrar este gradiente para aplicar el algoritmo de gradiente descendente. Este algoritmo se llama *backpropagation* o *retropropagación* y funciona de la siguiente manera.

1. Inicializar los pesos y valores umbrales de la red. En el primer paso, se establecen aleatoriamente.
2. Presentar el primer dato de entrenamiento de nuestro subconjunto elegido a la red y propagarlo hasta obtener una salida.
3. Calcular un error comparando la salida real con la salida deseada.
4. Propagar la señal de error de vuelta a través de la red y obtener un gradiente para cada peso y valor umbral.
5. Repetir los pasos del 2 al 3 con el siguiente vector de entrada
6. Calcular el promedio de los gradientes.
7. Ajustar los pesos y valores umbrales para minimizar el error total con el algoritmo de gradiente descendiente estocástico.

Explicaremos cada uno de estos pasos. En primer lugar, se inicializan los pesos y valores umbrales desde una distribución gaussiana estándar. Algunas funciones de activación, como la función sigmoide, tienen un problema de saturación, lo que significa que producen valores de salida cercanos a los extremos de su rango (0 o 1) para una amplia gama de valores de entrada. Esto puede resultar en gradientes cercanos a cero durante la retropropagación y dificultar el entrenamiento de la red neuronal. Para abordar este problema, se utiliza comúnmente una técnica de inicialización de pesos llamada Xavier, que divide este valor aleatorio por la raíz cuadrada del número de neuronas de la capa anterior. Si no se hiciera, el valor de uno de estos pesos estaría en el 95% de las veces en un rango de $[-2, 2]$, lo que haría que el propio valor de activación estuviera muy cerca de cero o uno, haciendo que el gradiente desaparezca o explote.

Probaremos si esta técnica realmente funciona en nuestro estudio.

En base a estos pesos y valores umbrales iniciales, propagamos una primera señal de entrada (véase la Ecuación 3) y obtenemos un error en base la función de costo escogida. Definiremos un nuevo término llamado $\delta_j^k = \frac{\partial C}{\partial z_j^k}$ para cada capa k y neurona j de la capa. Para la última capa K, este será igual a:

$$\delta_j^K = \frac{\partial C}{\partial a_j^K} f'(z_j^K). \tag{5}$$

El término de la izquierda mide cuánto cambia la función de costo respecto al valor de activación j, y el término de la derecha mide qué tan rápido cambia la función de activación con la entrada z. Esto se reescribe de forma matricial como:

$$\delta^k = \nabla_a C \odot f'(z^k). \tag{6}$$

Ahora propagamos el error a través de la red. Para cada capa k:

$$\delta^k = ((w^{k+1})^T \delta^{k+1}) \odot f'(z^k). \tag{7}$$

Por último, estos términos δ^k nos ayudan a calcular el gradiente de cada peso y valor umbral:

$$\frac{\partial C}{\partial b_j^k} = \delta_j^k, \tag{8}$$

$$\frac{\partial C}{\partial w_{ji}^k} = a_i^{k-1} \delta_j^k. \tag{9}$$

Estos valores se suman para cada dato de entrenamiento en el subconjunto escogido en el gradiente descendente estocástico y se dividen por el número de ellos, obteniendo un promedio del cambio necesario para cada peso y valor umbral. Ahora tenemos los gradientes que nos ayudarán a desarrollar el algoritmo de descenso por gradiente. Este proceso se repite cuantas veces sea necesario para encontrar un valor mínimo satisfactorio.

La demostración de estas ecuaciones se basa en la regla de la cadena (Véase [2]).

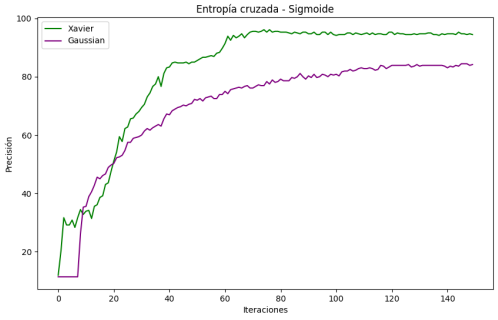

Figura 6: Precisión de la función de coste de entropía cruzada con la función de activación sigmoide

4 Modelo de solución

La implementación de la red neuronal usada para evaluar las distintas funciones de coste y activación está basada en [3]. La red neuronal implementada es de dos capas internas con 32 y 16 neuronas en cada una. Los valores de la tasa de aprendizaje se ajustaron para obtener los mejores resultados posibles en cada situación. La tasa se varió para valores de 1, 0.1, 0.01, 0.001, 0.0001, y 0.00001.

Para la función de entropía cruzada junto con la función de activación sigmoide (Figura 6), vemos que alcanza la mayor precisión con el conjunto de prueba de todos los modelos y converge más rápido cuando se aplica la inicialización Xavier. Nótese que, aunque tienen valores distintos, ambas alcanzan valores de precisión altos. La inicialización Xavier evita que las neuronas se sobresaturen, lo que les permite aprender más rápido y mejor. Para la función de activación tangente hiperbólica junto con la función de coste de entropía cruzada(Figura 7), el resultado es parecido: se alcanza una mayor precisión con la inicialización Xavier. Sin embargo, la precisión se reduce un poco. Cuando se inicializa de forma normal, la solución no supera el límite del 10%, que es la precisión que un generador aleatorio tendría. La función tangente hiperbólica

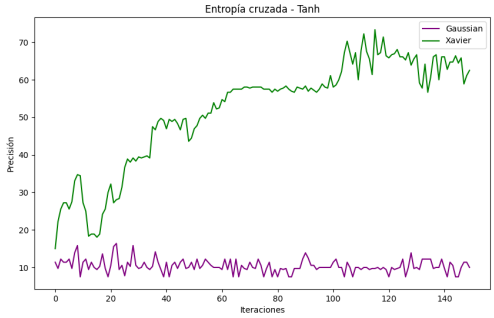

Figura 7: Precisión de la función de coste de entropía cruzada con la función de activación tangente hiperbólica

puede sufrir de problemas de saturación donde las entradas grandes resultan en gradientes muy pequeños, dificultando el aprendizaje. Para la función de activación ReLu con la función de entropía cruzada (Figura 8), notamos una baja en precisión aún mayor. Por sí sola, no tiene un buen comportamiento. Esto puede deberse a la naturaleza de la ReLu, que puede llevar a neuronas muertas durante el entrenamiento, es decir que ciertas neuronas dejen de activarse, especialmente con una mala inicialización de pesos.

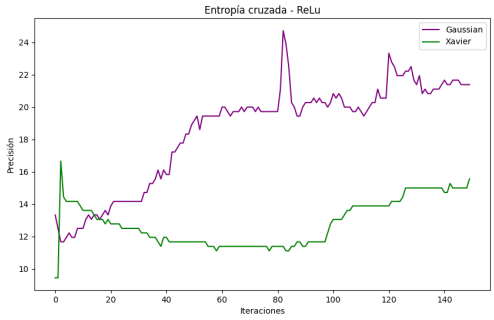

Figura 8: Precisión de la función de coste de entropía cruzada con la función de activación ReLu

Ahora, para la función de coste cuadrática

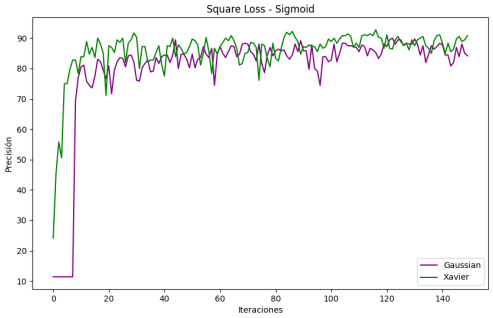

Figura 9: Precisión de la función de coste de error cuadrático con la función de activación sigmoide

y la función sigmoide (Figura 9), vemos que ambos modelos alcanzan valores altos, con el modelo de la inicialización de Xavier obteniendo resultados un poco mejores. La función de coste cuadrática es menos adecuada para clasificación en comparación con la entropía cruzada, pero aún así produce buenos resultados cuando la activación es sigmoide. Para la función de activación tangente hiperbólica con la función de coste cuadrática (Figura 10), vemos que los resultados son pésimos. No fue posible que el modelo de inicialización gaussiana convergiera para ningún valor de α. La combinación de una mala inicialización y una función de coste menos adecuada conduce a un aprendizaje deficiente y a una falta de convergencia. Para la función de activación ReLu con la función de coste cuadrática (Figura 11), incluso vemos mejores resultados para la inicialización gaussiana. Sin embargo los resultados siguen sin ser los mejores.

5 Conclusión

Concluimos que para nuestro problema de clasificación, la función de coste de entropía cruzada junto con la función sigmoide da mejores resultados, especialmente si se inicializan los pesos con el método Xavier para que

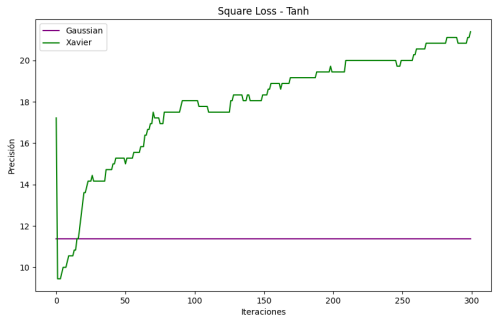

Figura 10: Precisión de la función de coste de error cuadrático con la función de activación tangente hiperbólica

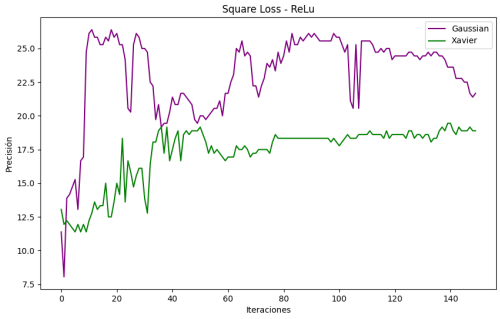

Figura 11: Precisión de la función de coste de error cuadrático con la función de activación ReLu

no se sobresaturen las neuronas. En la tabla se puede ver un resumen de la precisión de los resultados con el conjunto de validación.

Entropía Cruzada	**Sigmoide**	**Tanh**	**ReLU**
Xavier	96.39%	67.22%	19.44%
Gaussiana	85.83%	8.89%	10.56%

Tabla 2: Rendimiento de la Entropía Cruzada de las Funciones de Activación

Error Cuadrático Medio	**Sigmoide**	**Tanh**	**ReLU**
Xavier	91.67%	25.28%	19.17%
Gaussiana	82.50%	9.17%	22.50%

Tabla 3: Rendimiento del Error Cuadrático Medio de las Funciones de Activación

Para el modelo con la función sigmoide y la función de coste de entropía cruzada, implementamos modelos equivalentes en las librerías de Python *scikit-learn* (*MLPClassifier*) y *TensorFlow*, obteniendo una precisión de 0.72263 y de 0.94999 en cada caso.

Además, como se pudo notar, los modelos eran altamente sensibles a la inicialización de los pesos y valores umbrales. Sin embargo, cuando un modelo funciona bien, siempre convergerá a una precisión similar independientemente de la semilla con la que se inicializa. En el caso de nuestro mejor modelo, que utiliza la función de coste de entropía cruzada, función de activación sigmoide e inicialización Xavier, probamos 100 semillas distintas para la inicialización. Esto se hizo para comprobar que, aun cuando este modelo tiene un factor aleatorio, converge a una precisión alta.

Si un modelo es sensible a la semilla aleatoria, diríamos que no es un buen modelo. Véase en la figura 12 cómo aprendieron los distintos modelos. Nótese que, aunque aprenden de manera distinta, algunos más lentamente que otros, al final siempre alcanzan la misma precisión. En esta prueba se obtuvo un resultado

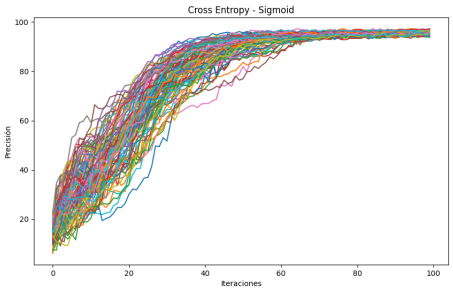

Figura 12: Precisión del modelo variando la semilla

promedio de 95.875% de precisión, con un intervalo de confianza de [95.7352, 96.0148], lo cual indica poca variabilidad. Esto nos permite estar seguros de que nuestro modelo no es altamente sensible a la aleatoriedad. Un buen modelo no debería serlo.

De estos resultados podemos extraer dos conclusiones importantes: en primer lugar, para problemas de clasificación, la combinación de la función de coste de entropía cruzada y la función sigmoide muestra ser altamente efectiva para mejorar la precisión del modelo. En segundo lugar, es crucial destacar la sensibilidad de los modelos a la inicialización de los pesos; una inicialización inapropiada puede resultar en valores extremadamente grandes, lo que potencialmente causa la inactivación de neuronas y su consiguiente falta de contribución al proceso de aprendizaje. Por último, aunque puede parecer que las funciones ReLu y Tanh no funcionan bien para este problema por si solas, han demostrado a través de los años que su fuerza está al combinarlas con otras. De esta manera se pueden modelar fenómenos más complejos con una mayor precisión que la función sigmoide.

6 Implementación

Para ver la implementación descrita aquí, ir al siguiente repositorio de GitHub: Implementacion. En él se encuentra el código de la red neuronal *neural-network.py*, la implementación en *scikit-learn* y *tensorflow* para propósitos de comparación, un archivo para la visualización de la base de datos *show.py* y otro para la carga de datos *data.py*, además del archivo principal *main.py*. Las librerías necesarias están en *requirements.txt*. Por último, se han guardado los pesos y valores umbrales de la mejor solución, cada capa en un archivo *.npy*, para facilitar el uso de esta red neuronal.

References

[1] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine Learning in Python . *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[2] Hao Li. Proofs for the four fundamental equations of the backpropagation and algorithms in feedforward neural networks. 10 2023.

[3] Li Deng. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

[4] M. W. Gardner and S. R. Dorling. Artificial neural networks (the multilayer perceptron)—a review of applications in the atmospheric sciences. *Atmospheric Environment*, 32(14–15):2627–2636, 1998.

[5] G. Sanderson. Neural networks: The basics of neural networks, and the math behind how they learn, 2017.

[6] M. Nielsen. *Neural Networks and Deep Learning*. 2019.

[7] *Neural networks for pattern recognition*. The Clarendon Press, Oxford University Press, New York, 1995.

[8] I. Neutelings. Neural networks, 2022.