UNIVERSITAT JAUME I

Grado en Matemática Computacional

Trabajo Final de Grado

Red neuronal para la elección de los parámetros de compresión óptimos

Autor:
Marta Aicart Pauner

Supervisor:
Francesc Alted Abad
Tutor académico:
Maria Victoria Ibañez Gual

Fecha de lectura: 20 de junio de 2022
Curso académico 2021/2022

Resumen

Este documento recopila los detalles y procedimientos más importantes del Trabajo Final de Grado, cuya aplicación práctica ha sido desarrollada durante la Estancia en Prácticas en la empresa ironArray.

Uno de los principales objetivos de los científicos a lo largo de la historia ha sido el diseño y construcción de máquinas capaces de realizar procesos con cierta inteligencia. En la actualidad, gracias a muchos años de investigación de profesionales repartidos por todo el mundo, las redes neuronales han alcanzado un gran nivel de madurez y se emplean en todo tipo de aplicaciones.

El objetivo de este Trabajo de Fin de Grado es la implementación de una red neuronal artificial capaz de predecir la mejor combinación de parámetros de compresión para un conjunto de datos dado. Posteriormente, esta red es usada para mejorar el rendimiento del compresor Blosc2 de ironArray. En primer lugar, se proporcionará una base teórica del aprendizaje automático, centrándose en las redes neuronales artificiales. A continuación, se presentará el modelo de red neuronal desarrollado junto con los resultados obtenidos. Después de este análisis, se compararán los objetivos con lo conseguido en la red. Finalmente, se expondrán las conclusiones y líneas futuras de trabajo, a partir de este Trabajo de Fin de Grado.

Palabras clave

Red neuronal, Retropropagación, Gradiente Descendiente.

Keywords

Neural network, Backpropagation, Gradient Descent.

Índice general

Capítulo 1

Introducción

1.1. Contexto y motivación del proyecto

Este documento recoge el Trabajo Final de Grado, cuyo objetivo principal es el estudio de la elección de los parámetros de compresión óptimos para un conjunto de datos mediante redes neuronales artificiales.

Con el paso del tiempo, para superar limitaciones físicas en el campo de la informática ha sido fundamental el uso de la compresión. Esta técnica minimiza el espacio ocupado por los archivos en disco duro y consecuentemente reduce el tiempo necesario para que sean descargados o transferidos. Esta reducción de espacio y tiempo implica un gran ahorro de costes. Por lo tanto, es fundamental el uso de librerías de rápida compresión como lo es C-Blosc2.

Actualmente, vivimos en un crecimiento tecnológico continuo y, entre estos avances cabe resaltar el logrado en el campo de la inteligencia artificial, concretamente el campo de las redes de neuronas artificiales o, comúnmente llamadas, redes neuronales. Las redes neuronales tratan de imitar el comportamiento de la red neuronal de un cerebro humano. En esta, millones de neuronas forman una red y a través de unas conexiones o sinapsis se comunican entre ellas para realizar distintas acciones. Sin embargo, las redes neuronales artificiales van más allá, ya que tratan de imitar este comportamiento y llevarlo a la computación para automatizar el aprendizaje. Esto supone una gran ventaja ya que no necesitan ser programadas explícitamente y destacan en ámbitos donde la programación convencional tiene recursos limitados para alcanzar soluciones a ciertos problemas.

Este proyecto nace motivado por el interés en comprender el funcionamiento de las redes neuronales y en la oportunidad de poder desarrollar un modelo de red neuronal propio. Este

tema es muy interesante por su gran y veloz desarrollo, y como consecuencia por su uso en una gran variedad de ámbitos tecnológicos. Además, existía la posibilidad de poder aplicar este modelo en una situación real, ya que era idóneo para optimizar el compresor de datos C-Blosc2, una librería que permite comprimir y descomprimir datos a una gran velocidad y sin pérdida de información.

Capítulo 2

Objetivos

El principal objetivo de este trabajo es el estudio y aplicación de las redes neuronales. Con esto se pretende desarrollar un modelo de red neuronal que elija en cada caso la mejor combinación de parámetros para la compresión del meta-compresor C-Blosc2 [8]. Concretamente, dado un dataset se pretende que la red indique cuales son los parámetros de compresión óptimos para maximizar su velocidad y ratio de compresión.

Capítulo 3

Desarrollo del TFG

3.1. Las redes neuronales

La red neuronal es un modelo computacional compuesto por distintos elementos procesales organizados en capas. Estos elementos se denominan neuronas, y a través de canales de comunicación unidireccionales envían información a otras unidades de la red.

La neurona es la unidad básica de procesamiento de la red neuronal. A cada neurona i se le asocia un valor, denominado variable de estado x_i. Tiene conexiones de entrada a través de las cuales recibe los valores de entrada. A estos valores se les aplica una determinada transformación y se genera un valor de salida.

Las neuronas se agrupan por capas, y si estas unidades tienen un conjunto común de entradas se denomina capa densa. Dos neuronas que se encuentren en una misma capa densa reciben la misma información de entrada de la capa anterior, y las salidas de cada capa son los inputs de la capa siguiente. La primera capa se denomina capa de entrada, y es aquella en la que las neuronas reciben los inputs, es decir, los datos a evaluar de cada elemento de la muestra (de cada observación); la última capa, es decir, aquella cuyas neuronas producen el output de la red, se denomina capa de salida; y las capas intermedias se denominan capas ocultas, en las cuales se realizan los cálculos internos de la red. Por lo tanto, la red neuronal consiste en una serie de capas en las que cada una recibe la información procesada por la capa anterior. En la figura 3.1 se puede apreciar la arquitectura descrita de una red neuronal.

Además, cada neurona está conectada con otras a través de unos enlaces, y a cada conexión (i, j) de las neuronas i y j se le asocia un peso $w_{ij} \in \mathbb{R}$. Este peso define con qué intensidad cada variable de entrada afecta a la neurona. De hecho, estos pesos son parámetros del modelo,

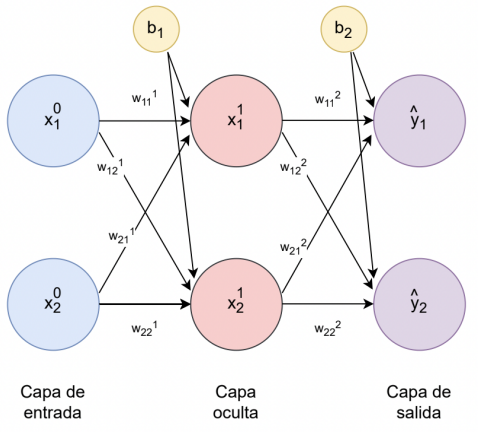

Figura 3.1: Arquitectura de una red neuronal con una única capa oculta

cuyos valores debe estimar la red neuronal.

Estos pesos se representan en forma de matriz, matriz de pesos W, y existe una para cada capa oculta. Su dimensión es n x m donde, el número de filas n es es el número de neuronas de la capa actual y el número de columnas m es el número de neuronas de la capa anterior. Por lo tanto, si W^L es la matriz de pesos de la capa oculta L, entonces el peso w_{ij}^L de esta matriz representa el peso de enlace entre la neurona i de la capa $L-1$ hacia la neurona j de la capa L.

En la primera capa de la red, la variable de estado x_i de cada neurona es un valor de entrada a la red, y en el resto de capas es el valor obtenido tras realizar una serie de cálculos a partir de los valores de las neuronas de las capas anteriores (Eq. 3.2).

Las variables de estado de las neuronas de la cada capa oculta se puede expresar vectorialmente como $X_L = [x_1^L, x_2^L, \ldots, x_m^L]$ para $L = 0, 1, .., N$, siendo L la capa oculta, N el número de capas (sin contar la capa de entrada), $L = 0$ la capa de entrada y m el número de neuronas de la capa L, pudiendo variar este número dependiendo de la capa en la que nos situemos.

Para la neurona i de una capa L distinta a la primera, se define la siguiente suma ponderada a partir de las variables de estado de la capa anterior:

$$z_i^L = Z_i^L(X^{L-1}) = \sum_{j=1}^{m_{L-1}} w_{ji}^L x_j^{L-1} + b_L \tag{3.1}$$

donde en cada capa, b_L (bias en inglés) es un nuevo parámetro a estimar. Los sesgos de una

red pueden expresarse en forma vectorial b, donde $b = [b_1, b_2, \ldots, b_N]$, siendo b_i el sesgo de la capa oculta i y siendo N el número de capas (sin contar la capa de entrada).

La red conecta múltiples neuronas de forma secuencial, pero en cada neurona se realiza un problema de regresión lineal. Por lo tanto, en la red se concatenan diferentes operaciones de regresión lineal. A la salida de la neurona, puede existir una función no lineal, que modifica el valor resultado o impone un límite que no se debe sobrepasar antes de propagarse a otra neurona. Esta función se conoce como función de activación, que utiliza la suma ponderada de la entrada anterior y la transforma una vez más como salida. Esta función añade deformaciones no lineales al valor de salida de la neurona, para así poder encadenar de forma efectiva la computación de varias unidades. Existen muchas funciones de activación, como por ejemplo la función sigmoide cuya ecuación es $a(z) = \frac{1}{1+e^{-z}}$, la tangente hiperbólica cuya ecuación es $a(z) = \frac{e^z - e^{-z}}{e^z - e^{-z}}$, etc, pero se va a explicar solo una de ellas, porque es la utilizada en la red neuronal creada que se verá posteriormente. Esta función es la Rectified Linear Unit (ReLU), y se define como

$$x_i^L = a(z_i^L) = max(0, z_i^L) = max(0, \sum_{j=1}^{m_{L-1}} w_{ji}^L x_j^{L-1} + b_L) \tag{3.2}$$

Esta función se comporta como una función lineal cuando el valor de entrada es positivo, y constante a cero cuando es negativo. Se trata de una función de activación no lineal, la cual ayuda a modelar funciones curvas o no triviales. En la figura 3.2 se puede apreciar la representación gráfica de diversas funciones de activación.

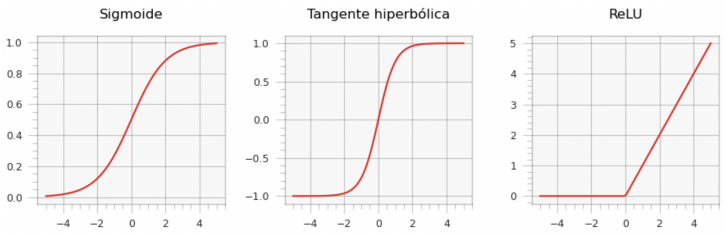

Figura 3.2: Ejemplos de funciones de activación

Por lo tanto, en cada neurona (menos en las neuronas de la capa de entrada) se realiza primero la suma ponderada de sus entradas expresada en la ecuación 3.1, y después a este valor se le aplica una función de activación, es decir, se realiza la composición de dos funciones expresada en la ecuación 3.2. En la figura 3.4 se puede apreciar el proceso de estos cálculos.

Una vez realizados todos los cálculos en las capas ocultas de la red, el resultado de la última capa oculta es utilizado como entrada en la capa de salida para realizar estos mismos cálculos, es decir, los definidos en la ecuación 3.2. En nuestro caso, crearemos una red de clasificación, es decir para clasificar cada observación entre J posibles clases disjuntas. Por lo tanto, la capa

de salida constará de J neuronas, y en cada neurona j se calcula un valor de salida $\hat{y}_j$, que en nuestro caso nos dará la probabilidad de que la observación que estamos analizando pertenezca a la j-ésima clase. En caso de que $J = 2$, la red tendrá una única neurona en la capa final.

En el caso particular de una red neuronal con 2 neuronas en la capa de entrada, nuestro conjunto de datos será $T = \{(x_1(i), x_2(i), y_i\}_{1=1,...,M}$. Llamaremos muestra u observación a una única fila de datos, la cual contiene unos inputs que son los que se introducen en el algoritmo y una salida, la etiqueta $((x_1(i), x_2(i), y_i), i \in \{1 = 1, \ldots, M\})$. Cada observación del conjunto de datos está relacionada con su etiqueta observada, que en la red de clasificación se refiere a la clase a la que pertenece la observación. Tanto a las etiquetas observadas como a los valores predichos por la red se les aplica una codificación denominada *one hot encoding*. Esta codificación consiste en transformar estas variables numéricas en vectores de probabilidad de longitud J, siendo J el número de clases. Estos vectores se expresan como $y_i = [y_{i1}, y_{i2}, \ldots, y_{iJ}]$ para las etiquetas observadas, con

$$\sum_{j=1}^{J} y_{ij} = 1, \quad i = 1, ..., M, \quad y_{ij} \in \{0, 1\} \tag{3.3}$$

donde todos los elementos del vector son 0, excepto uno, que tiene un 1 como valor, donde el 1 especifica la categoría del elemento.

Utilizaremos la notación $\hat{y}_i = [\hat{y}_{i1}, \hat{y}_{i2}, \ldots, \hat{y}_{iJ}]$ para los valores predichos por la red para la muestra i-ésima. Es decir, los valores del vector representan probabilidades y por lo tanto la suma de todos estos valores siempre debe ser 1, es decir:

$$\sum_{j=1}^{J} \hat{y}_{ij} = 1, \quad i = 1, ..., M, \quad \hat{y}_{ij} \geq 0 \tag{3.4}$$

Tanto las etiquetas observadas como los valores predichos se representan en matrices con M filas y J columnas, donde M es el número de observaciones, y J es el número de clases. Entonces, el elemento ij-ésimo de las matrices será la probabilidad de que el elemento correspondiente al input i pertenezca a la clase j-ésima. Por lo tanto, el valor y_{ij} es la probabilidad real de que ocurra el evento j-ésimo para la muestra i-ésima y el valor $\hat{y}_{ij}$ es la probabilidad predicha por la red de que ocurra el evento correspondiente para la muestra i-ésima.

Una de las formas de lograr que las salidas de cada capa sean no negativas es utilizar la función de activación ReLU definida anteriormente en la ecuación 3.2. Además, en la última capa utilizaremos la función de activación *softmax* (Ec. 3.5) que toma como entrada un vector de números reales $[x_i]_{i=1}^{J}$ y lo normaliza en una distribución de probabilidad que consta de J probabilidades proporcionales a las exponenciales de los números de entrada. En este caso, los valores que toma como entrada son los calculados en las neuronas de la capa de salida N, definidos en forma vectorial como $X^N = [x_{i1}^N, x_{i2}^N, \ldots, x_{iJ}^N]$.

Por lo tanto, softmax convierte estas salidas en probabilidades, siendo s(x_{ij}^N) la probabilidad de que la muestra i pertenezca a la clase j. Su fórmula es:

$$\hat{y}_{ij} = s(x_{ij}^N) = \frac{e^{x_{ij}^N}}{\sum_{j=1}^{J} e^{x_{ij}^N}}, \quad j = 1, ..J \tag{3.5}$$

donde J es el número de clases. En la figura 3.3 se puede observar el funcionamiento de esta función.

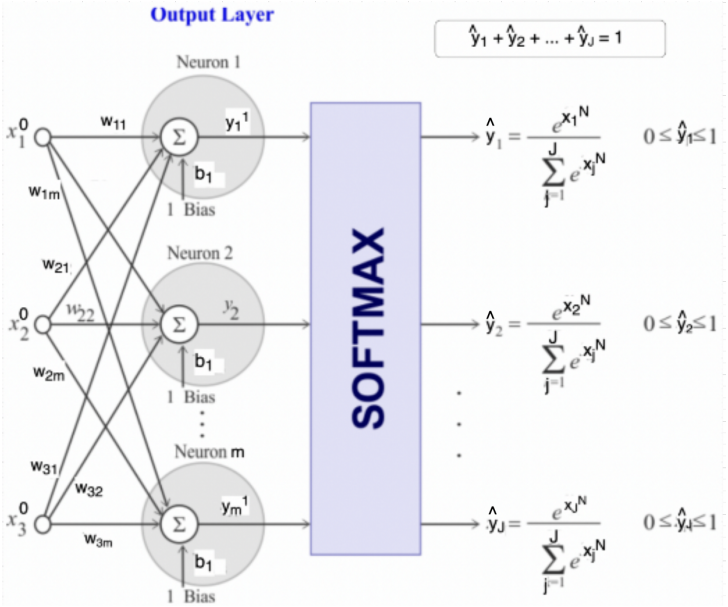

Figura 3.3: Función softmax

3.2. Entrenamiento de las redes neuronales

Originalmente partimos de un conjunto de datos, $T = \{(x_1(i), x_2(i), \cdots, x_m(i), y_i\}_{1=1,\cdots,M}$, en inglés dataset, donde para cada observación hemos recogido el valor de m variables ($\{x_1, x_2, \cdots, x_m\}$) a las que denominamos inputs, y la clase a la que pertenecen ($\{y\}$).

Entrenar una red neuronal es estimar el valor de los parámetros que forman parte de la red para lograr que la salida ($\{\hat{y}\}$) sea lo más parecida posible al valor real ($\{y\}$).

A la hora de entrenar, es importante dividir el dataset en dos subconjuntos de datos disjuntos. Uno de ellos es el conjunto de datos de entrenamiento, mientras que el otro es el de test. El

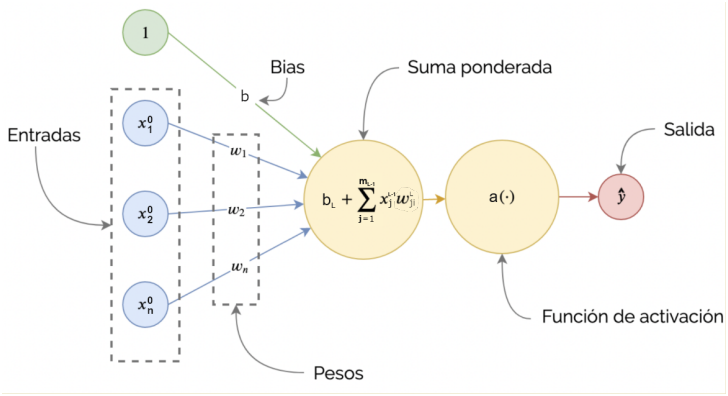

Figura 3.4: Arquitectura de una neurona

primero de ellos sirve como input de la red durante su entrenamiento, es decir, con el objetivo de estimar los pesos de las matrices W de la red de tal forma que se minimice el error de predicción. El segundo conjunto de datos, sirve para comprobar cómo de bien predice la red para unos datos nuevos no usados durante el entrenamiento. En el entrenamiento de la red debemos evaluar su rendimiento, y esta evaluación varía según el tipo de red. En nuestro caso, se trata de una red de clasificación, y por lo tanto, una forma de evaluar su rendimiento es calculando la frecuencia con la que la red clasifica correctamente los inputs, es decir, el porcentaje de número de aciertos sobre el total de elementos del conjunto de entrenamiento. A este parámetro se le denomina precisión o exactitud. Por otro lado, otra forma de evaluar su rendimiento es mediante la definición de una función de pérdida, que mide de una manera numérica la diferencia existente entre la salida ofrecida por la red y la salida correcta. Existen diversas funciones de pérdida, entre ellas el error lineal, el error cuadrático medio, entropía cruzada binaria, etc. No obstante, se va a explicar solo la que se aplica en la red neuronal creada que se presentará posteriormente.

Como se ha explicado previamente, la función de pérdida es una función que estima la desviación entre los valores reales y las predicciones hechas por la red. El objetivo de la red es minimizar esta función, es decir, reducir al mínimo la desviación entre el valor real y el valor predicho, mediante el ajuste de los parámetros de la red. En este caso, la función de pérdida utilizada es la Categorical Cross Entropy (Ec. 3.6), una función que rinde muy bien en redes de clasificación.

En cuanto a la optimización de la red, esta se lleva a cabo en dos etapas: la primera de ellas consiste en la aplicación del algoritmo de backpropagation, el cual calcula cómo afectan los valores de los parámetros al resultado de la función de pérdida, y en la segunda etapa se utiliza el algoritmo de optimización, el cual optimiza la red y cambia el valor de los parámetros a lo largo del entrenamiento para así ajustarlos. En nuestra aplicación utilizaremos como optimizador el método Adam. A continuación se explica con detalle la función de pérdida Categorical Cross

Entropy (Sec. 3.3) y el algoritmo Adam (Sec. 3.4.6).

3.3. Categorical Cross Entropy

La entropía cruzada categórica es una función de pérdida utilizada en una red neuronal para la clasificación multiclase, donde a cada muetra de los datos se le asigna una única etiqueta de clase. Esta función cuantifica la diferencia entre dos distribuciones de probabilidad discretas.

A partir de las etiquetas observadas en el conjunto de datos ($\{y_i\}_{i=1}^{J}$) y las predicciones obtenidas en la red ($\{\hat{y}_i\}_{i=1}^{J}$), con (Sec. 3.1):

$$y_i = [y_{i1}, y_{i2}, ..., y_{iJ}] \text{ tal que}$$
$$\sum_{j=1}^{J} y_{ij} = 1, \quad i = 1, \ldots, M, \quad y_{ij} \in \{0, 1\}$$

y

$$\hat{y}_i = [\hat{y}_{i1}, \hat{y}_{i2}, ..., \hat{y}_{iJ}] \text{ tal que}$$
$$\sum_{j=1}^{J} \hat{y}_{ij} = 1, \quad i = 1, \ldots, M, \quad \hat{y}_{ij} \geq 0,$$

la fórmula de la función Categorical Cross Entropy se define como:

$$C(y_i, \hat{y}_i) = -\sum_{i=1}^{M} \sum_{j=1}^{J} y_{ij} \cdot log\hat{y}_{ij} \tag{3.6}$$

donde M es el número de inputs (tamaño de la muestra), J el número de clases, $\hat{y}_{ij}$ es la probabilidad predicha por la red de que la muestra i-ésima pertenezca a la clase j-ésima y y_{ij} es la probabilidad real de que ocurra el evento correspondiente para la muestra i-ésima. Además, el signo menos asegura que la pérdida disminuya cuando las distribuciones se acercan más entre sí. Como $\hat{y}_{ij} \in [0, 1]$, con el fin de no calcular el logaritmo con argumento cero que no existe, realmente la biblioteca keras realiza internamente el $\log(\hat{y}_{ij} + \epsilon)$, siendo ϵ un valor muy pequeño del orden de 10^{-7}.

3.4. Adam

El algoritmo Adam [4], Adaptative Moment Estimation, es un algoritmo de optimización utilizado para estimar los pesos de la red de forma iterativa en función de los datos de entrenamiento. Su objetivo es ajustar los parámetros de la red θ (W y b) para minimizar la función de

pérdida, en nuestro caso la función Categorical Cross Entropy (Ec. 3.6). Para ello utiliza unos algoritmos de optimización, concretamente combina el algoritmo SGD con Momentum (Sec. 3.4.3) y el algoritmo RMSprop (Sec. 3.4.4) definidos en las próximos apartados. Así mismo, estos algoritmos de optimización, concretamente el Adam que es el que se va a usar en el modelo de red neuronal, utilizan el algoritmo del Backpropagation (Sec. 3.4.5) para calcular las derivadas necesarias en los algoritmos de optimización.

A continuación se va a explicar los algoritmos de optimización SGD con Momentum (Sec. 3.4.3) y RMSProp (Sec. 3.4.4) para posteriormente presentar el algoritmo Adam (Sec. 3.4.6) que combina ambos. Además, se va a explicar el algoritmo de optimización del gradiente descendente, GD, (Sec. 3.4.1) y el del gradiente descendiente estocástico, SGD, (Sec. 3.4.2) ya que SGD con Momentum es una optimización de sus predecesores DG y SGD. Y para concluir, se va a presentar el algoritmo Backpropagation (Sec. 3.4.5) ya que es el usado por estos algoritmos de optimización para calcular sus derivadas.

3.4.1. Gradiente Descendiente (GD)

El método del gradiente descendiente [1] es un algoritmo de optimización cuyo objetivo es converger hacia el valor mínimo de una función, $C(\theta)$, mediante un proceso iterativo.

La velocidad de convergencia del algoritmo dependerá de un parámetro α al que llamaremos tasa de aprendizaje o tamaño de paso. Una tasa demasiado grande puede impedir que se alcance la solución óptima, mientras que un valor demasiado pequeño implica que el algoritmo necesitará muchos pasos para alcanzar el óptimo. En la figura 3.5 se puede observar el efecto que tienen diferentes tasas de aprendizaje en una determinada función $C(\theta)$.

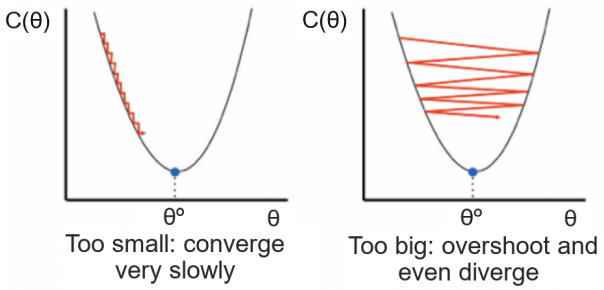

Figura 3.5: Efecto de diferentes tasas de aprendizaje

En el primer paso, el algoritmo presentado en 1 inicializa el parámetro (o vector de parámetros) θ aleatoriamente como θ_0. A continuación se calcula el gradiente de la función respecto a θ y se evalua en θ_0 ($g_1 \leftarrow \nabla_\theta C(\theta_0)$), posteriormente se actualiza el parámetro $\theta_1 \leftarrow \theta_0 - \alpha \cdot g_1$.

El algoritmo comprueba si se ha alcanzado la convergencia, es decir, si la diferencia entre el valor de los parámetros actualizados y los parámetros de la iteración anterior no superan el valor de tolerancia definido. Si es así, se ha alcanzado la convergencia y hemos llegado a un óptimo (que puede ser global o local), pero en caso contrario el algoritmo prosigue, calculando $g_t \leftarrow \nabla_\theta C(\theta_{t-1})$ y actualizando $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot g_t$.

Algorithm 1 Gradiente descendiente (GD)

Require: α : *Tasa de aprendizaje o tamaño de paso*
Require: $C(\theta)$ *función objetivo con vector de parámetros* θ
Require: θ_0 : *vector inicial de parámetros*
  $t \leftarrow 0$
  while θ_t *not converged* **do**
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta C(\theta_{t-1})$ *(cálculo de los gradientes de la función de pérdida en la iteración actual t)*
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot g_t$ *(actualización de los parámetros)*
  end while
  return θ_t

GD en Redes Neuronales

La función a optimizar cuando trabajamos con redes neuronales es una función de pérdida, $C(\theta)$, que en nuestro caso es la Categorical Cross Entropy (Ec. 3.6), que no se evalúa solamente sobre una observación sino sobre un conjunto de observaciones (conjunto de entrenamiento) de tamaño M. En este caso, denotamos por $C_i(\theta)$ a la función de pérdida correspondiente a la i-ésima observación y la función a minimizar es:

$$C(\theta) = \sum_{i=1}^{M} C_i(\theta)$$

3.4.2. Gradiente Descendiente Estocástico (SGD)

El descenso de gradiente estocástico [2] es un método iterativo para optimizar una función objetivo, en nuestro caso, la función de pérdida Categorical Cross Entropy 3.6. Se trata de una optimización del algoritmo previamente definido GD 3.4.1, ya que la diferencia entre estos dos algoritmos es que en el algoritmo GD el número de muestras de entrenamiento coincide con todo el conjunto de entrenamiento, mientras que en SGD se puede elegir la cantidad de muestras a usar del conjunto de entrenamiento, las cuales son seleccionadas aleatoriamente. Aunque SGD utiliza una sola muestra como unidad de entrenamiento para entrenar más rápido, no es eficiente en conjuntos de datos muy grandes.

3.4.3. Gradiente Descendiente Estocástico con Momentum (SGD con Momentum)

Una optimización del SGD es el SGD con Momentum [7]. Este algoritmo, en vez de tomar el gradiente en el punto en particular y desplazarse en esa dirección, toma un promedio ponderado de los gradientes previos, que denotará por m_t. De esta forma, las oscilaciones en una determinada dirección se cancelan en gran medida y por lo tanto se converge mucho más rápido al valor que minimiza la función.

El algoritmo 2 inicializa aleatoriamente θ a θ_0 e inicializa la tasa de aprendizaje o tamaño de paso α y la tasa de decaimiento exponencial β, cuyos valores habituales asignados son $\alpha = 0{,}001$ y $\beta = 0{,}9$. Además, el promedio ponderado m_0 es inicializado a 0.

En cada iteración primero se calcula el gradiente mediante la derivada parcial de la función de coste C respecto de θ ($g_t \leftarrow \nabla_\theta C_t(\theta_{t-1})$). A continuación se actualizan los promedios ponderados $m_t \leftarrow \beta \cdot m_{t-1} + (1-\beta) \cdot g_t$ y posteriormente se actualizan los parámetros $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot m_t$. A continuación se comprueba si el algoritmo ha alcanzado la convergencia, es decir, si la diferencia entre los parámetros actualizados y los parámetros de la iteración anterior no superan el valor de tolerancia definido. Si es así, se ha alcanzado la convergencia y hemos llegado a un óptimo (que puede ser global o local), pero en caso contrario el algoritmo prosigue, calculando $g_t \leftarrow \nabla_\theta C(\theta_{t-1})$ y actualizando $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot m_t$.

Algorithm 2 Gradiente Descendiente Estocástico con Momentum (SGD con Momentum)

Require: α : *Tasa de aprendizaje o tamaño de paso*
Require: β : *Tasa de decaimiento exponencial*
Require: $C(\theta)$ *función objetivo con parámetros* θ
Require: θ_0 : *vector inicial de parámetros*
  $m_0 \leftarrow 0$ *(Inicializar los promedios ponderados)*
  $t \leftarrow 0$
  while θ_t *not converged* **do**
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta C(\theta_{t-1})$ *(cálculo de los gradientes de la función de pérdida en la iteración actual t)*
    $m_t \leftarrow \beta \cdot m_{t-1} + (1-\beta) \cdot g_t$ *(actualización de los promedios ponderados)*
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot m_t$ *(actualización de los parámetros)*
  end while
  return θ_t

Además, se puede aproximar el número de elementos previos tomados en el promedio ponderado: 1/(1 - β).

3.4.4. Propagación de raíz cuadrática media (RMSProp)

RMSProp [3] también hace uso de un promedio ponderado y del gradiente de la función de pérdida respecto los parámetros, pero usa otras propiedades matemáticas, lo que permite converger más rápido al mínimo de la función. Este algoritmo trata de normalizar el gradiente con la raíz cuadrada del valor medio de los cuadrados. El objetivo es conseguir que la componente que tiene una razón de cambio mayor, avance más lento; y la que tiene razón de cambio menor avance más rápido. En este caso, se toma el promedio ponderado del cuadrado de las razones de cambio.

Algorithm 3 Propagación de raíz cuadrada media (RMSProp)

Require: α : *Tasa de aprendizaje o tamaño de paso*
Require: β : *Tasa de decaimiento exponencial*
Require: $C(\theta)$ *función objetivo con parámetros* θ
Require: θ_0 : *vector inicial de parámetros*
  $v_0 \leftarrow 0$ *(Inicializar los promedios ponderados)*
  $t \leftarrow 0$
  while θ_t *not converged* **do**
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta C(\theta_{t-1})$ *(cálculo de los gradientes de la función de pérdida en la iteración actual t)*
    $v_t \leftarrow \beta \cdot v_{t-1} + (1 - \beta) \cdot g_t^2$ *(actualización de los promedios ponderados)*
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \frac{g_t}{\sqrt{v_t + \epsilon}}$ *(actualización de los parámetros)*
  end while
  return θ_t

El algoritmo 3 inicializa aleatoriamente θ a θ_0, inicializa la tasa de aprendizaje α y la tasa de decaimiento exponencial β, cuyos valores habituales asignados son $\alpha = 0{,}001$ y $\beta = 0{,}999$. Además, el promedio ponderado m_0 es inicializado a 0. En cada iteración primero se calcula el gradiente mediante la derivada parcial de la función de coste C respecto de θ ($g_t \leftarrow \nabla_\theta C(\theta_{t-1})$). A continuación se actualizan los promedios ponderados $v_t \leftarrow \beta \cdot v_{t-1} + (1-\beta) \cdot g_t^2$ y posteriormente se actualizan los parámetros $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \frac{g_t}{\sqrt{v_t + \epsilon}}$. A continuación se comprueba si el algoritmo ha alcanzado la convergencia, es decir, si la diferencia entre los parámetros actualizados y los parámetros de la iteración anterior no superan el valor de tolerancia definido. Si es así, se ha alcanzado la convergencia y hemos llegado a un óptimo (que puede ser global o local), pero en caso contrario el algoritmo prosigue, calculando $g_t \leftarrow \nabla_\theta C(\theta_{t-1})$ y actualizando $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \frac{g_t}{\sqrt{v_t + \epsilon}}$.

3.4.5. Backpropagation

El backpropagation [6] es un método de cálculo del gradiente utilizado en algoritmos de aprendizaje supervisado con el fin de entrenar redes neuronales. Este método usa un ciclo de propagación dividido en dos fases. En la primera de ellas, se aplica un patrón de entrada a la red y este se propaga desde la primera hasta la última a través de las capas ocultas. La salida de la red se compara con la salida esperada y se calcula un error para cada una de las salidas, y en nuestros caso el error está definido por la función de pérdida definida en la ecuación 3.6. En la segunda fase, las salidas del error se propagan hacia atrás desde la última capa hacia la capa oculta anterior. No obstante, las neuronas de la capa oculta sólo reciben una fracción de la señal total del error, basándose aproximadamente en la contribución relativa que haya aportado cada neurona a la salida original. Este proceso se repite, capa por capa, hasta que todas las neuronas de la red hayan recibido una señal de error que describa su contribución relativa al error total.

Por lo tanto, el backpropagation es el método utilizado para calcular las derivadas parciales de cada parámetro de la red con respecto al coste. Posteriormente, la optimización por la cual se va a minimizar el coste de la red se realiza mediante el algoritmo de descenso del gradiente. Este método usa el vector gradiente, un vector con las derivadas parciales de los parámetros de la red respecto al coste, y estas derivadas parciales son las calculadas con el backpropagation.

Veamos ahora cómo se calculan estas derivadas:

El objetivo es calcular cómo varía el coste (la función de pérdida) C ante un cambio del parámetro. En la red hay dos tipos de parámetros: los pesos (w) y el sesgo (b) y estos son inicializados de manera aleatoria. En el proceso de optimización necesitaremos calcular la derivada parcial respecto w ($\frac{\partial C}{\partial w}$) y la derivada parcial respecto b ($\frac{\partial C}{\partial b}$).

En primer lugar se calcula la derivada de los parámetros de la última capa. El superíndice indica el número de la capa a la que corresponde el parámetro. Suponiendo que N es el número total de capas de la red, recordemos que:

- $a()$ es la función de activación, siendo $a^N()$ esta función aplicada en las neuronas de la última capa (N), y $a^{N-1}()$ aplicada en la capa anterior ($N-1$).
- $Z^N = W^N X^{N-1} + b^N = W^N a^{N-1}(Z^{N-1}) + b^N$, es resultado de la suma ponderada de la última capa, donde W^N es la matriz de pesos involucrada en el cálculo de la última capa y X^{N-1} el conjunto de las variables de estado de la capa anterior.
- $X^N = a(Z^{N-1})$
- $\hat{y}_{ij} = s(x_{ij}^N)$ (Eq. 3.5), pero para simplificar la notación en la explicación de como funciona el backpropagation vamos a suponer que podemos tomar $s = identidad$, es decir: $\hat{y}_{ij} = x_{ij}^N$

- $C()$ es la función de coste o de pérdida definida en la ecuación 3.6.

Como $Error = C(y, \hat{y}) = C(y, a(Z^N))$ y $Z^N = W^N a^{N-1}(Z^{N-1}) + b^N$, entonces $Error = C(a(W^N a(Z^{N-1}) + b^N))$ y aplicando la regla de la cadena obtenemos las derivadas parciales de la última capa, que se describen como:

$$\frac{\partial C}{\partial w^N} = \frac{\partial C}{\partial a^N} * \frac{\partial a^N}{\partial z^N} * \frac{\partial z^N}{\partial w^N}$$

$$\frac{\partial C}{\partial b^N} = \frac{\partial C}{\partial a^N} * \frac{\partial a^N}{\partial z^N} * \frac{\partial z^N}{\partial b^N}$$

$\frac{\partial C}{\partial a^N}$ es la derivada de la función de coste respecto de la función de activación, es decir, la derivada de la función de coste respecto al output de la red neuronal.

$\frac{\partial a^N}{\partial z^N}$ es la derivada de la función de activación respecto a la suma ponderada de la neurona, es decir, la derivada de la función de activación.

$\frac{\partial z^N}{\partial w^N}$ y $\frac{\partial z^N}{\partial b^N}$ son las derivadas de la suma ponderada al variar los pesos o el sesgo respectivamente. Por lo tanto, $\frac{\partial z^N}{\partial b^N} = 1$ ya que b es una variable independiente, y $\frac{\partial z^N}{\partial w^N} = a_i^{N-1}$ es el valor de entrada a la neurona.

Además, $\frac{\partial C}{\partial z^N} = \frac{\partial C}{\partial a^N} * \frac{\partial a^N}{\partial z^N}$ representa cómo varía el error en función de la suma ponderada, es decir, en qué grado se modifica el error al realizar un cambio en la suma de la neurona. Si la derivada es grande, significa que ante un pequeño cambio en el valor de la neurona, este se verá reflejado en el resultado final. No obstante, si la derivada es pequeña significa que el valor de la suma no afectará al valor de la red. Entonces, esta derivada indica qué responsabilidad tiene la neurona en el resultado final y por tanto en el error. Además, recibe el nombre de error imputado a la neurona, y se representa como ς^N.

Por lo tanto, $\frac{\partial C}{\partial w^N} = \varsigma^N \; a_i^{N-1}$ y $\frac{\partial C}{\partial b^N} = \varsigma^N$.

De esta forma, hemos calculado el error para cada una de las neuronas de la última capa (ς^N) y el error para cada una de las derivadas parciales ($\frac{\partial C}{\partial w^N}$ y $\frac{\partial C}{\partial b^N}$). No obstante, todavía se necesita seguir derivando para poder calcular el resto de derivadas de la red. Para ello, realizamos el mismo cálculo para la capa anterior, la capa $N-1$.

Como Error $= C(a^N(Z^N(Z^{N-1})))$ y $Z^{N-1} = W^N a^{N-1}(W^{N-1}a^{N-2} + b^{N-1}) + b^N$, entonces $Error = C(a^N(Z^N(W^N a^{N-1}(W^{N-1}a^{N-2} + b^{N-1}) + b^N)))$, y aplicando la regla de la cadena obtenemos las derivadas parciales de la penúltima capa se describen como:

$$\frac{\partial C}{\partial w^{N-1}} = \frac{\partial C}{\partial a^N} * \frac{\partial a^N}{\partial z^N} * \frac{\partial z^N}{\partial a^{N-1}} * \frac{\partial aN-1}{\partial z^{N-1}} * \frac{\partial z^{N-1}}{\partial w^{N-1}}$$

$\frac{\partial C}{\partial b^{N-1}} = \frac{\partial C}{\partial a^N} * \frac{\partial a^N}{\partial z^N} * \frac{\partial z^N}{\partial a^{N-1}} * \frac{\partial aN-1}{\partial z^{N-1}} * \frac{\partial z^{N-1}}{\partial b^{N-1}}$

donde $\frac{\partial C}{\partial a^N} \frac{\partial a^N}{\partial z^N} = \varsigma^N$ y por lo tanto $\frac{\partial C}{\partial a^N} * \frac{\partial a^N}{\partial z^N} * \frac{\partial z^N}{\partial a^{N-1}} * \frac{\partial a^{N-1}}{\partial z^{N-1}} = \frac{\partial C}{\partial z^{N-1}} = \varsigma^{N-1}$. Además, $\frac{\partial zN-1}{\partial w^{N-1}} = a^{N-2}$ y $\frac{\partial z^{N-1}}{\partial b^{N-1}} = 1$.

De esta forma, tenemos calculadas:

- El error de la última capa: $\varsigma^N = \frac{\partial C}{\partial a^N} * \frac{\partial a^N}{\partial z^N}$
- El error de la capa anterior: $\varsigma^{N-1} = W^N \varsigma^N * \frac{\partial a^{N-1}}{\partial z^{N-1}}$
- Las derivadas de la capa usando el error: $\frac{\partial C}{\partial b^{N-1}} = \varsigma^{N-1}$ y $\frac{\partial C}{\partial w^{N-1}} = \varsigma^{N-1} * a^{N-2}$

Y así sucesivamente recorriendo todas las capas de la red. Por lo tanto, con estas cuatro expresiones, con un único pase se pueden calcular todos los errores y derivadas parciales de la red.

3.4.6. Algoritmo Adam

En el algoritmo Adam (4) se mantiene una tasa de aprendizaje para cada parámetro de la red, es decir, el tamaño de paso del gradiente descendiente estocástico se ajusta a medida que se desarrolla el aprendizaje.

Los parámetros de configuración de Adam son:

- α: se conoce como tasa de aprendizaje o tamaño de paso. La proporción en la que se actualizan las ponderaciones. Los valores más grandes dan como resultado un aprendizaje inicial más rápido antes de que se actualice la tasa. Los valores más pequeños ralentizan el aprendizaje durante el entrenamiento
- β_1: tasa de caída exponencial para las estimaciones del primer momento. Suele ser de 0,9.
- β_2: tasa de caída exponencial para las estimaciones del segundo momento. Suele ser de 0,999.
- ϵ: número muy pequeño para evitar cualquier división por cero en la implementación (por ejemplo, 10^{-9}).

Como hemos dicho anteriormente, Adam combina SGD Momentum con RMSProp, y su algoritmo se define como:

Algorithm 4 Adam

Require: α : *Tasa de aprendizaje o tamaño de paso*
Require: β_1 , $\beta_2 \in [0, 1)$: *Tasas de decaimiento exponenciales para los momentos estimados*
Require: $C(\theta)$ *función objetivo con parámetros* θ
Require: θ_0 : *vector inicial de parámetros*
  $m_0 \leftarrow 0$ *(Inicializar el primer momento)*
  $v_0 \leftarrow 0$ *(Inicializar el segundo momento)*
  $t \leftarrow 0$
  while θ_t *not converged* **do**
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta C_t(\theta_{t-1})$ *(cálculo de los gradientes de la función de pérdida en la iteración actual t)*
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ *(actualización de los primeros momentos estimados)*
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ *(actualización de los segundos momentos estimados)*
    $\hat{m}_t \leftarrow \frac{m_t}{1-\beta_1^t}$ *(cálculo de la corrección de los primeros momentos)*
    $\hat{v}_t \leftarrow \frac{v_t}{1-\beta_2^t}$ *(cálculo de la corrección de los segundos momentos)*
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t}+\epsilon}$ *(actualización de los parámetros)*
  end while
  return θ_t

Como podemos observar el algoritmo Adam combina SGD con Momentum y RMSProp. Como se ha explicado anteriormente, m_t y v_t son los promedios ponderados que se calculaban en los algorítmos SGD con Momentum y RMSProp respectivamente. m_t y v_t representan los dos momentos, m_t es el que modela la media de los gradientes a lo largo del tiempo y v_t hace lo mismo pero modelando la varianza; y lo mismo ocurre respectivamente con el sesgo. En este algoritmo, en las primeras iteraciones tiene un arranque lento, es decir, en las actualizaciones de los parámetros de las primeras iteraciones se tiene muy poco en cuenta el valor de los promedios ponderados. Para solventar este problema en cada iteración se realiza una corrección de los momentos antes de actualizar los parámetros.

Concretamente, el algoritmo calcula un promedio móvil exponencial del gradiente y el gradiente al cuadrado, y los parámetros β_1 y β_2 controlan las tasas de caída de estos promedios móviles.

El algoritmo 4 se inicia tomando unos valores aleatorios de los parámetros θ (W y b), que al ser los iniciales los denominamos como θ_0. Además, inicializa la tasa de aprendizaje o tamaño de paso α y las tasas de decaimiento exponenciales β_1 y β_2. Una buena configuración determinada para este algoritmo es definir α como 0.001, β_1 como 0.9 y β_2 como 0.999. Además, los primeros y segundos momentos m_0 y v_0 son inicializados a 0. En cada iteración primero se calcula el gradiente mediante la derivada parcial de la función de coste C respecto de los parámetros θ ($g_t \leftarrow \nabla_\theta C_t(\theta_{t-1})$) resultantes de la iteración anterior. A continuación se actualizan los primeros y segundos momentos mediante las expresiónes $m_t \leftarrow \beta 1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ y

$v_t \leftarrow \beta 2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ y posteriormente se calcula la corrección de los primeros y segundos momentos mediante las expresiones $\hat{m}_t \leftarrow \frac{m_t}{1-\beta_1^t}$ y $\hat{v}_t \leftarrow \frac{v_t}{1-\beta_2^t}$. Después, se actualizan los parámetros de esta iteración mediante la expresión $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t + \epsilon}}$. A continuación se comprueba si el algoritmo ha alcanzado la convergencia, es decir, si la diferencia entre los parámetros actualizados y los parámetros de la iteración anterior superan el valor de tolerancia definido. Si es así, el algoritmo prosigue, pero en caso contrario esto implica que ha alcanzado la convergencia y termina.

3.4.7. Ejemplo

Una vez visto como funciona Adam, veamos sobre un caso particular cómo este algoritmo calcula las derivadas de los parámetros de una red para ajustarlos con el fin de minimizar una función de pérdida. Concretamente, se han realizado los cálculos de estas derivadas para una sola observación, suponiendo que la función a minimizar es $C(\theta) = C_i(\theta)$, pero el desarrollo sería análogo en el caso de $C(\theta) = \frac{1}{M}\sum_{i=1}^{N} C_i(\theta)$. En este ejemplo, la arquitectura de la red consiste en una capa de entrada, una capa oculta densa y una capa de salida, cada una de ellas con dos neuronas. Además, la función de activación usada en la capa oculta y en la capa de salida es distinta; en la primera de ellas se usa la función ReLU (ec. 3.2) y en la segunda de ellas se usa la función softmax (ec. 3.5). Podemos observar esta red en la imagen 3.6.

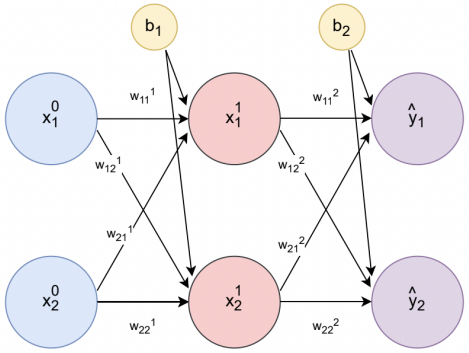

Figura 3.6: Arquitectura de la red del ejemplo práctico

- $x_1^1 = a(z_1^1) = a(x_1^0 \cdot w_{11}^1 + x_2^0 \cdot w_{21}^1 + b_1) = max(0, x_1^0 \cdot w_{11}^1 + x_1^0 \cdot w_{21}^1 + b_1)$
- $x_2^1 = a(z_2^1) = a(x_1^0 \cdot w_{12}^1 + x_2^0 \cdot w_{22}^1 + b_1) = max(0, x_1^0 \cdot w_{12}^1 + x_2^0 \cdot w_{22}^1 + b_1)$
- $\hat{y}_1 = s(z_1^2) = s(x_1^2 \cdot w_{11}^2 + x_2^2 \cdot w_{21}^2 + b_2)$
- $\hat{y}_1 = s(z_1^2) = \frac{e^{z_1^2}}{e^{z_1^2} + e^{z_2^2}}$

- $\hat{y}_2 = s(z_2^2) = s(x_1^2 \cdot w_{12}^2 + x_2^2 \cdot w_{22}^2 + b_2)$
- $\hat{y}_2 = s(z_2^2) = \frac{e^{z_2^2}}{e^{z_1^2} + e^{z_2^2}}$
- $C(y_i, \hat{y}_i) = -\sum_{i=1}^{J} y_i \cdot log\hat{y}_i = y_1 \cdot log\hat{y}_1 + y_2 \cdot log\hat{y}_2$

- CÁLCULO DE LAS DERIVADAS PARCIALES DE LA FUNCIÓN DE COSTE C RESPECTO A LOS PARÁMETROS DE LA ÚLTIMA CAPA (b_2, w_{11}^2, w_{12}^2, w_{21}^2 y w_{22}^2):

Aplicamos las siguientes fórmulas ya definidas en la sección 3.4.5: $\frac{\partial C}{\partial w^N} = \frac{\partial C}{\partial a^N} \cdot \frac{\partial a^N}{\partial z^N} \cdot \frac{\partial z^N}{\partial w^N}$ y $\frac{\partial C}{\partial b^N} = \frac{\partial C}{\partial a^N} \cdot \frac{\partial a^N}{\partial z^N} \cdot \frac{\partial z^N}{\partial b^N}$

- $\frac{\partial z^N}{\partial w^N}$
  - $\frac{\partial z_1^2}{\partial w_{11}^2} = x_1^2$
  - $\frac{\partial z_1^2}{\partial w_{21}^2} = x_1^2$
  - $\frac{\partial z_2^2}{\partial w_{12}^2} = x_1^2$
  - $\frac{\partial z_2^2}{\partial w_{22}^2} = x_1^2$
- $\frac{\partial z^N}{\partial b^N}$
  - $\frac{\partial z_1^2}{\partial b^2} = 1$
  - $\frac{\partial z_2^2}{\partial b^2} = 1$
- $\frac{\partial a^N}{\partial z^N}$
  - $\frac{\partial \hat{y}_1}{\partial z_1^2} = \frac{e^{z_1^2} \cdot (2 \cdot e^{z_1^2} + e^{z_2^2})}{(e^{z_1^2} + e^{z_2^2})^2}$
  - $\frac{\partial \hat{y}_2}{\partial z_2^2} = \frac{e^{z_2^2} \cdot (2 \cdot e^{z_2^2} + e^{z_1^2})}{(e^{z_1^2} + e^{z_2^2})^2}$
- $\frac{\partial C}{\partial a^N}$
  - $\frac{\partial C}{\partial \hat{y}_1} = \frac{y_1}{\hat{y}_1}$
  - $\frac{\partial C}{\partial \hat{y}_2} = \frac{y_2}{\hat{y}_2}$

Por lo tanto, juntando las derivadas parciales, sustituyendo y expresando en función de x y w obtenemos:

- $\frac{\partial C}{\partial w_{11}^2} = \frac{\partial C}{\partial \hat{y}_1} \cdot \frac{\partial \hat{y}_1}{\partial z_1^2} \cdot \frac{\partial z_1^2}{\partial w_{11}^2} = y_1 \cdot x_1^2 \cdot \frac{2 \cdot e^{x_1^2 \cdot w_{11}^2 + x_2^2 \cdot w_{21}^2 + b_2} + e^{x_1^2 \cdot w_{12}^2 + x_2^2 \cdot w_{22}^2 + b_2}}{e^{x_1^2 \cdot w_{11}^2 + x_2^2 \cdot w_{21}^2 + b_2} + e^{x_1^2 \cdot w_{12}^2 + x_2^2 \cdot w_{22}^2 + b_2}}$
- $\frac{\partial C}{\partial w_{12}^2} = \frac{\partial C}{\partial \hat{y}_2} \cdot \frac{\partial \hat{y}_2}{\partial z_2^2} \cdot \frac{\partial z_2^2}{\partial w_{12}^2} = y_2 \cdot x_1^2 \cdot \frac{2 \cdot e^{x_1^2 \cdot w_{12}^2 + x_2^2 \cdot w_{22}^2 + b_2} + e^{x_1^2 \cdot w_{11}^2 + x_2^2 \cdot w_{21}^2 + b_2}}{e^{x_1^2 \cdot w_{11}^2 + x_2^2 \cdot w_{21}^2 + b_2} + e^{x_1^2 \cdot w_{12}^2 + x_2^2 \cdot w_{22}^2 + b_2}}$
- $\frac{\partial C}{\partial w_{21}^2} = \frac{\partial C}{\partial \hat{y}_1} \cdot \frac{\partial \hat{y}_1}{\partial z_1^2} \cdot \frac{\partial z_1^2}{\partial w_{21}^2} = y_1 \cdot x_2^1 \cdot \frac{2 \cdot e^{x_1^2 \cdot w_{11}^2 + x_2^2 \cdot w_{21}^2 + b_2} + e^{x_1^2 \cdot w_{12}^2 + x_2^2 \cdot w_{22}^2 + b_2}}{e^{x_1^2 \cdot w_{11}^2 + x_2^2 \cdot w_{21}^2 + b_2} + e^{x_1^2 \cdot w_{12}^2 + x_2^2 \cdot w_{22}^2 + b_2}}$
- $\frac{\partial C}{\partial w_{22}^2} = \frac{\partial C}{\partial \hat{y}_2} \cdot \frac{\partial \hat{y}_2}{\partial z_2^2} \cdot \frac{\partial z_2^2}{\partial w_{22}^2} = y_2 \cdot x_2^1 \cdot \frac{2 \cdot e^{x_1^2 \cdot w_{12}^2 + x_2^2 \cdot w_{22}^2 + b_2} + e^{x_1^2 \cdot w_{11}^2 + x_2^2 \cdot w_{21}^2 + b_2}}{e^{x_1^2 \cdot w_{11}^2 + x_2^2 \cdot w_{21}^2 + b_2} + e^{x_1^2 \cdot w_{12}^2 + x_2^2 \cdot w_{22}^2 + b_2}}$
- $\frac{\partial C}{\partial b_2} = \frac{\partial C}{\partial \hat{y}_1} \cdot \frac{\partial \hat{y}_1}{\partial z_1^2} \cdot \frac{\partial z_1^2}{\partial b_2} + \frac{\partial C}{\partial \hat{y}_2} \cdot \frac{\partial \hat{y}_2}{\partial z_2^2} \cdot \frac{\partial z_2^2}{\partial b_2} = y_1 \cdot \frac{2 \cdot e^{x_1^2 \cdot w_{11}^2 + x_2^2 \cdot w_{21}^2 + b_2} + e^{x_1^2 \cdot w_{12}^2 + x_2^2 \cdot w_{22}^2 + b_2}}{e^{x_1^2 \cdot w_{11}^2 + x_2^2 \cdot w_{21}^2 + b_2} + e^{x_1^2 \cdot w_{12}^2 + x_2^2 \cdot w_{22}^2 + b_2}} + y_2 \cdot \frac{2 \cdot e^{x_1^2 \cdot w_{12}^2 + x_2^2 \cdot w_{22}^2 + b_2} + e^{x_1^2 \cdot w_{11}^2 + x_2^2 \cdot w_{21}^2 + b_2}}{e^{x_1^2 \cdot w_{11}^2 + x_2^2 \cdot w_{21}^2 + b_2} + e^{x_1^2 \cdot w_{12}^2 + x_2^2 \cdot w_{22}^2 + b_2}}$

- CÁLCULO DE LAS DERIVADAS PARCIALES DE LA FUNCIÓN DE COSTE C RESPECTO A LOS PARÁMETROS DE LA CAPA OCULTA (b_2, w_{11}^2, w_{12}^2, w_{21}^2 y w_{22}^2):

Aplicamos las siguientes fórmulas ya definidas en la sección 3.4.5:
$\frac{\partial C}{\partial w^{N-1}} = \frac{\partial C}{\partial a^N} \cdot \frac{\partial a^N}{\partial z^N} \cdot \frac{\partial z^N}{\partial a^{N-1}} \cdot \frac{\partial aN-1}{\partial z^{N-1}} \cdot \frac{\partial z^{N-1}}{\partial w^{N-1}}$
y
$\frac{\partial C}{\partial b^{N-1}} = \frac{\partial C}{\partial a^N} \cdot \frac{\partial a^N}{\partial z^N} \cdot \frac{\partial z^N}{\partial a^{N-1}} * \frac{\partial aN-1}{\partial z^{N-1}} \cdot \frac{\partial z^{N-1}}{\partial b^{N-1}}$

- $\frac{\partial z^{N-1}}{\partial w^{N-1}}$
  - $\frac{\partial z_1^1}{\partial w_{11}^1} = x_1^0$
  - $\frac{\partial z_1^1}{\partial w_{21}^1} = x_2^0$
  - $\frac{\partial z_2^1}{\partial w_{12}^1} = x_1^0$
  - $\frac{\partial z_2^1}{\partial w_{22}^1} = x_2^0$
- $\frac{\partial a^{N-1}}{\partial z^{N-1}}$
  - $\frac{\partial x_1^1}{\partial z_1^1} = 0$ si $z_1^1 = 0$, $\frac{\partial x_1^1}{\partial z_1^1} = 1$ si $z_1^1 \neq 0$
  - $\frac{\partial x_2^1}{\partial z_2^1} = 0$ si $z_2^1 = 0$, $\frac{\partial x_2^1}{\partial z_2^1} = 1$ si $z_2^1 \neq 0$
- $\frac{\partial z^N}{\partial a^{N-1}}$
  - $\frac{\partial z_1^2}{\partial x_1^1} = w_{11}^2$

- $\frac{\partial z_1^2}{\partial x_2^1} = w_{21}^2$
- $\frac{\partial z_2^2}{\partial x_1^1} = w_{12}^2$
- $\frac{\partial z_2^2}{\partial x_2^1} = w_{22}^2$

- $\frac{\partial a^N}{\partial z^N}$
  - $\frac{\partial \hat{y}_1}{\partial z_1^2} = \frac{e^{z_1^2} \cdot (2 \cdot e^{z_1^2} + e^{z_2^2})}{(e^{z_1^2} + e^{z_2^2 2})}$
  - $\frac{\partial \hat{y}_2}{\partial z_2^2} = \frac{e^{z_2^2} \cdot (2 \cdot e^{z_2^2} + e^{z_1^2})}{(e^{z_1^2} + e^{z_2^2 2})}$
- $\frac{\partial C}{\partial a^N}$
  - $\frac{\partial C}{\partial \hat{y}_1} = \frac{y_1}{\hat{y}_1}$
  - $\frac{\partial C}{\partial \hat{y}_2} = \frac{y_2}{\hat{y}_2}$

Para hacer los siguientes cálculos suponemos que $z_1^1 \neq 0$ y $z_2^1 \neq 0$, ya que en caso contrario las derivadas serían 0. Por lo tanto, juntando las derivadas parciales, sustituyendo y expresando en función de x y w obtenemos:

- $\frac{\partial C}{\partial w_{11}^1} = \frac{\partial C}{\partial \hat{y}_1} \cdot \frac{\partial \hat{y}_1}{\partial z_1^2} \cdot \frac{\partial z_1^2}{\partial x_1^1} \cdot \frac{\partial x_1^1}{\partial z_1^1} \cdot \frac{\partial z_1^1}{\partial w_{11}^1} + \frac{\partial C}{\partial \hat{y}_2} \cdot \frac{\partial \hat{y}_2}{\partial z_2^2} \cdot \frac{\partial z_2^2}{\partial x_1^1} \cdot \frac{\partial x_1^1}{\partial z_1^1} \cdot \frac{\partial z_1^1}{\partial w_{11}^1} = x_1^0 \cdot [y_1 \cdot w_{11}^2 \cdot \frac{2 \cdot e^{z_1^2} + e^{z_2^2}}{(e^{z_1^2} + e^{z_2^2 2})} + y_2 \cdot w_{12}^2 \cdot \frac{2 \cdot e^{z_2^2} + e^{z_1^2}}{(e^{z_1^2} + e^{z_2^2 2})}]$
- $\frac{\partial C}{\partial w_{12}^1} = \frac{\partial C}{\partial \hat{y}_1} \cdot \frac{\partial \hat{y}_1}{\partial z_1^2} \cdot \frac{\partial z_1^2}{\partial x_2^1} \cdot \frac{\partial x_2^1}{\partial z_2^1} \cdot \frac{\partial z_2^1}{\partial w_{12}^1} + \frac{\partial C}{\partial \hat{y}_2} \cdot \frac{\partial \hat{y}_2}{\partial z_2^2} \cdot \frac{\partial z_2^2}{\partial x_2^1} \cdot \frac{\partial x_2^1}{\partial z_2^1} \cdot \frac{\partial z_2^1}{\partial w_{12}^1} = x_1^0 \cdot [y_1 \cdot w_{21}^2 \cdot \frac{2 \cdot e^{z_1^2} + e^{z_2^2}}{(e^{z_1^2} + e^{z_2^2 2})} + y_2 \cdot w_{22}^2 \cdot \frac{2 \cdot e^{z_2^2} + e^{z_1^2}}{(e^{z_1^2} + e^{z_2^2 2})}]$
- $\frac{\partial C}{\partial w_{21}^1} = \frac{\partial C}{\partial \hat{y}_1} \cdot \frac{\partial \hat{y}_1}{\partial z_1^2} \cdot \frac{\partial z_1^2}{\partial x_1^1} \cdot \frac{\partial x_1^1}{\partial z_1^1} \cdot \frac{\partial z_1^1}{\partial w_{21}^1} + \frac{\partial C}{\partial \hat{y}_2} \cdot \frac{\partial \hat{y}_2}{\partial z_2^2} \cdot \frac{\partial z_2^2}{\partial x_1^1} \cdot \frac{\partial x_1^1}{\partial z_1^1} \cdot \frac{\partial z_1^1}{\partial w_{21}^1} = x_2^0 \cdot [y_1 \cdot w_{11}^2 \cdot \frac{2 \cdot e^{z_1^2} + e^{z_2^2}}{(e^{z_1^2} + e^{z_2^2 2})} + y_2 \cdot w_{12}^2 \cdot \frac{2 \cdot e^{z_2^2} + e^{z_1^2}}{(e^{z_1^2} + e^{z_2^2 2})}]$
- $\frac{\partial C}{\partial w_{22}^1} = \frac{\partial C}{\partial \hat{y}_1} \cdot \frac{\partial \hat{y}_1}{\partial z_1^2} \cdot \frac{\partial z_1^2}{\partial x_2^1} \cdot \frac{\partial x_2^1}{\partial z_2^1} \cdot \frac{\partial z_2^1}{\partial w_{22}^1} + \frac{\partial C}{\partial \hat{y}_2} \cdot \frac{\partial \hat{y}_2}{\partial z_2^2} \cdot \frac{\partial z_2^2}{\partial x_2^1} \cdot \frac{\partial x_2^1}{\partial z_2^1} \cdot \frac{\partial z_2^1}{\partial w_{22}^1} = x_2^0 \cdot [y_1 \cdot w_{21}^2 \cdot \frac{2 \cdot e^{z_1^2} + e^{z_2^2}}{(e^{z_1^2} + e^{z_2^2 2})} + y_2 \cdot w_{22}^2 \cdot \frac{2 \cdot e^{z_2^2} + e^{z_1^2}}{(e^{z_1^2} + e^{z_2^2 2})}]$
- $\frac{\partial C}{\partial b_1} = \frac{\partial C}{\partial \hat{y}_1} \cdot \frac{\partial \hat{y}_1}{\partial z_1^2} \cdot \frac{\partial z_1^2}{\partial x_1^1} \cdot \frac{\partial x_1^1}{\partial z_1^1} + \frac{\partial C}{\partial \hat{y}_1} \cdot \frac{\partial \hat{y}_1}{\partial z_1^2} \cdot \frac{\partial z_1^2}{\partial x_2^1} \cdot \frac{\partial x_2^1}{\partial z_2^1} + \frac{\partial C}{\partial \hat{y}_2} \cdot \frac{\partial \hat{y}_2}{\partial z_2^2} + \frac{\partial C}{\partial \hat{y}_2} \cdot \frac{\partial \hat{y}_2}{\partial z_2^2} \cdot \frac{\partial z_2^2}{\partial x_1^1} \cdot \frac{\partial x_2^1}{\partial z_2^1} = y_1 \cdot w_{11}^2 \cdot \frac{2 \cdot e^{z_1^2} + e^{z_2^2}}{(e^{z_1^2} + e^{z_2^2 2})} + y_1 \cdot w_{11}^2 \cdot \frac{2 \cdot e^{z_1^2} + e^{z_2^2}}{(e^{z_1^2} + e^{z_2^2 2})} + y_2 \cdot w_{12}^2 \cdot \frac{2 \cdot e^{z_2^2} + e^{z_1^2}}{(e^{z_1^2} + e^{z_2^2 2})} + y_2 \cdot w_{22}^2 \cdot \frac{2 \cdot e^{z_2^2} + e^{z_1^2}}{(e^{z_1^2} + e^{z_2^2 2})}$

Capítulo 4

Resultados

Una vez estudiados los fundamentos teóricos de las redes neuronales más sencillas, vamos a aplicarlos de forma práctica en un desafío propuesto por la empresa donde he desarrollado mis prácticas curriculares.

Esta empresa es IronArray y su actividad consiste en proporcionar herramientas para llevar a cabo una compresión y descompresión de datos rápida y eficaz, adaptándose a las necesidades de rendimiento y almacenamiento del cliente. Una de las principales bibliotecas desarrolladas por el equipo es C-Blosc2 [8], un compresor y metacompresor al mismo tiempo. Entre los distintos métodos de compresión usados por la empresa, encontramos tres parámetros a ajustar.

El primero de ellos es el *filtro*, un programa que reordena los datos sin modificar su tamaño, de manera que el tamaño inicial y final sean iguales. Un filtro consta de codificador y decodificador. Partimos de que un dato es un conjunto o bloque de bytes de cierto tamaño. Por lo tanto, los filtros reorganizan los bytes o bits del bloque de datos a comprimir, acumulando a la izquierda los bytes o bits que son más significativos de cada dato y los bytes o bits menos significativos a la derecha. En una representación binaria (ceros y unos) donde cada número representa un byte, los bytes más significativos son los que tienen el mayor valor y están situados más a la izquiera, mientras que los menos significativos son los que tienen el menor valor y se situan más a la derecha. De entre los filtros que proporciona el compresor C-Blosc2, vamos a distinguir entre shuffle, el cual reordena los datos a nivel de byte; y bitshuffle, que los reordena a nivel de bit. En la figura 4.1 se puede apreciar el esquema del filtro shuffle, siendo el del bitshuffle el mismo pero en vez de tratar con bytes lo hace con bits. El segundo parámetro es el códec, un algoritmo capaz de comprimir y descomprimir un flujo de datos con el objetivo de reducir su tamaño y permitir una transmisión de datos más rápida. Disponemos de cinco códecs distintos: blosclz, zlib, zstd, lz4 i lz4hc. El codificador de filtro se aplica antes del compresor de códec (o codificador de códec) para que los datos sean más fáciles de comprimir y el decodificador

de filtro se usa después del descompresor de códec (o decodificador de códec) para restaurar la disposición original de los datos. El tercero de ellos es el splitmode, el cual especifica si se va a dividir los datos en splits o bloques (conjuntos de splits). Si el splitmode es split, significa que divide el dataset en splits, pero si el splitmode es nosplit, entonces se divide en bloques.

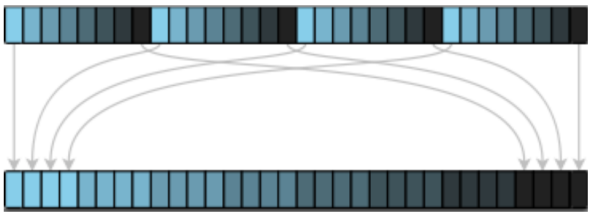

Figura 4.1: Esquema del filtro shuffle, en la parte superior aparecen 4 datos de 8 bytes, mientras que en la parte inferior aparecen los bytes filtrados y reordenados.

Una de las variables a definir es el favour, es decir, el parámetro en el que el usuario decide si quiere maximizar el ratio de compresión (max_cratio), la velocidad de compresión (max_speed) o equilibrio entre los dos (balance). Dependiendo del favour seleccionado, se le aplicará un determinado peso a la función de puntuación de categorías definida previamente. Por lo tanto, para cada favour se ha creado un modelo de red neuronal concreto.

Por experiencia previa, sabemos que cuando se quiere maximizar el ratio de compresión, los códecs más efectivos son zlib, zstd y lz4hc; cuando lo que se quiere maximizar es la velocidad de compresión, son idóneos los códecs blosclz y lz4; y finalmente para optimizar el modo balance, los mejores resultados los proporcionan los códecs blosclz y zstd, el primero de ellos en modo split, y el segundo en modo nosplit. Por lo tanto, dada esta información se ha decidido fijar uno de los filtros disponibles, concretamente el shuffle, y lo mismo en los dos primeros tipos de balance con unos de los splitmode disponibles, el split. En las siguientes tablas se muestra las combinaciones de parámetros (categorías) entre las cuales cada red neuronal deberá clasificar.

Favour
max_cratio
max_speed
balance

Cuadro 4.1: Tipos de favour según objetivo a optimizar

Además del compresor con estos parámetros, ironArray dispone de un programa llamado entropy_probe, el cual dado un dataset calcula su entropía, es decir, el grado de desorden de los datos de ese dataset. Con el objetivo de obtener información más detallada sobre los datos comprimidos, he realizado unas implementaciones sobre él. Como consecuencia, al programa se le pasa un dataset y su función es hacer una estimación del ratio y velocidad de compresión para un determinado filtro, aunque en los modelos de red neuronal creados se ha decidido

Filtro	Códec	Splitmode	Identificador de categoría	Categoría
shuffle	zlib	split	0	zlib-shuffle-split
shuffle	zstd	split	1	zstd-shuffle-split
shuffle	lz4hc	split	2	lz4hc-shuffle-split

Cuadro 4.2: Categorías de la red neuronal para optimizar el cratio (max_cratio)

Filtro	Códec	Splitmode	Identificador de categoría	Categoría
shuffle	blosclz	split	0	blosclz-shuffle-split
shuffle	lz4	split	1	lz4-shuffle-split

Cuadro 4.3: Categorías de la red neuronal para optimizar el speed (max_speed)

fijar como filtro el shuffle. Este estimador funciona siempre con el splitmode en modo split. Posteriormente, estos ratios y velocidades serán los inputs del modelo de red neuronal de multi clasificación creado.

El objetivo es optimizar estos parámetros de compresión para cada dataset que se vaya a comprimir. Para ello, se ha creado un modelo de red neuronal que dados el ratio y velocidad de compresión de unos datos, predice cuál es la mejor combinación que se le puede aplicar. Concretamente, se ha creado un modelo de red neuronal para cada objetivo posible de favour; uno para optimizar el ratio de compresión, otro para la velocidad de compresión y otro para optimizar el balance. Los inputs de las redes son los ratios y velocidades de compresión de entropía, y los outputs son las combinaciones óptimas de parámetros, a las que denominamos categorías. Por lo tanto, estos modelos consisten en redes de multi clasificación.

4.1. Obtención de los datos

Para entrenar la red neuronal necesitamos unos inputs. En este caso, estos son el ratio y la velocidad de compresión. El ratio de compresión es la relación entre el tamaño del dato comprimido y el tamaño del dato sin comprimir. En cuanto a la velocidad de compresión, esta se refiere al tiempo (Bytes/segundo) que se tarda en comprimir los datos.

Como se ha introducido anteriormente, con el fin de obtener estos inputs de los datos he implementado unas funciones en el programa en lenguaje C denominado entropy_probe.c, el cual ha sido proporcionado por ironArray. Como consecuencia, al ejecutar el programa para un determinado dataset, se obtiene un documento csv donde para cada porción del dataset (split), obtenemos información sobre su ratio de compresión y velocidad de compresión, entre otros. Esta información se obtiene por una parte para los filtros y codecs seleccionados, y por otra parte, si se especifica en las configuraciones de ejecución del programa, para un códec especial

Filtro	Códec	Splitmode	Identificador de categoría	Categoría
shuffle	blosclz	split	0	blosclz-shuffle-split
shuffle	lz4	nosplit	1	lz4-shuffle-nosplit

Cuadro 4.4: Categorías de la red neuronal para optimizar el balance

desarrollado por ironArray. Este códec se denomina entropy_mode. Su finalidad es medir la entropía del dataset, es decir, el grado de ordenación de los datos dentro del dataset. Además, en este caso sirve como estimador de ratio y velocidad de compresión en la red neuronal. En los modelos de red neuronal creados, se ha utilizado tres datasets. El primero de ellos contiene información sobre precipitaciones, el segundo sobre temperaturas y el tercero sobre el viento. Para la creación de las redes neuronales se ha utilizado un dataset resultado de la unión de los tres nombrados.

A continuación se muestra las gráficas donde aparecen las nubes de puntos para cada clase, es decir, el ratio y velocidad de compresión para cada muestra de los tres datasets.

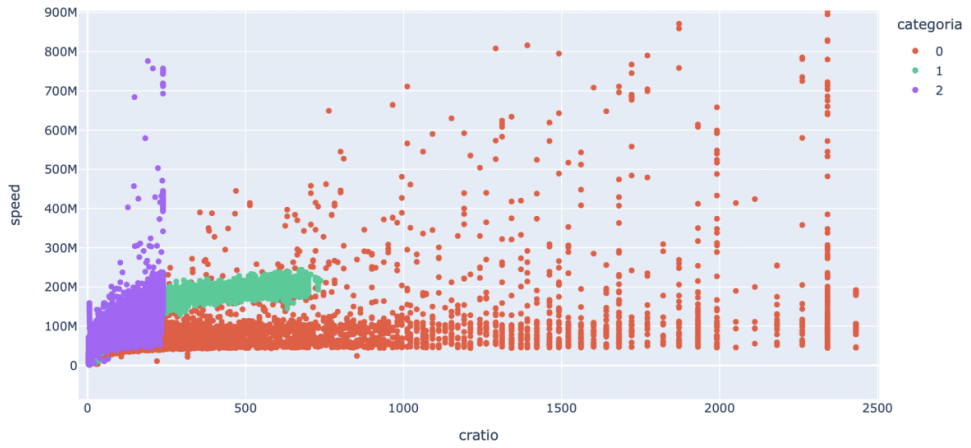

Figura 4.2: Mediciones del dataset para el favour en modo max_cratio.

En la figura 4.2 podemos observar las mediciones de ratio y velocidad de compresión para los datos del dataset global, y lo mismo en las figuras 4.3 y 4.4; diferenciándose entre ellas por el modo de favour seleccionado.

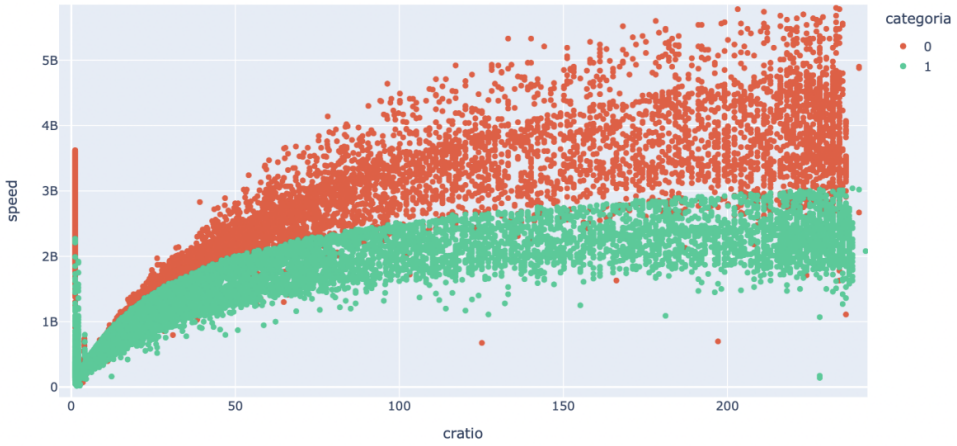

Figura 4.3: Mediciones del dataset para el favour en modo max_speed.

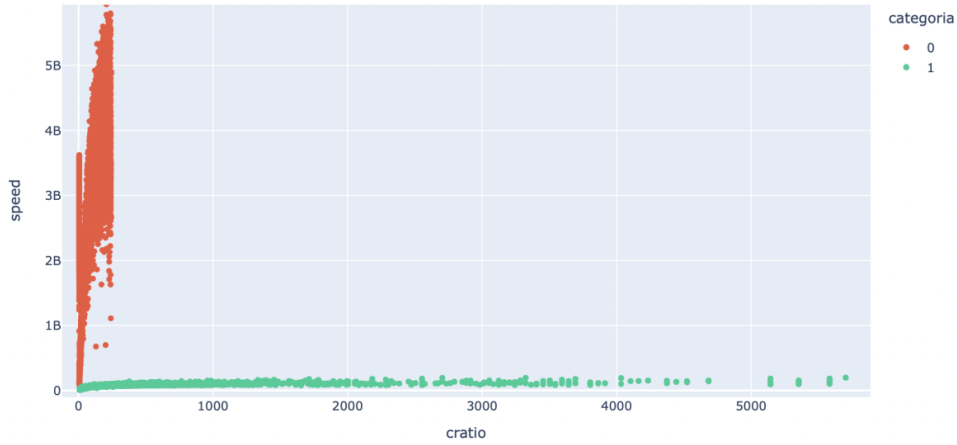

Figura 4.4: Mediciones del dataset para el favour en modo balance.

4.2. Preparación y manipulación de los datos

Una vez obtenidos los archivos csv con la información sobre los datasets, la siguiente tarea a realizar es la preparación de estos datos para convertirlos en los inputs definitivos de los modelos de red neuronal.

Esta preparación se ha llevado a cabo en el Jupyter Notebook, una aplicación web libre para crear y compartir documentos que contienen bloques de código con sus correspondientes salidas

por pantalla, y al mismo tiempo bloques de texto potenciados con el lenguaje Markdown, de tal forma que soporta LaTex y HTML. Es muy útil en el análisis de datos ya que permite llevar un registro de las técnicas que se han empleado, además de que permite generar informes en HTML y pdf. Además, para realizar ciertas operaciones de manejo de datos se han utilizado las librerías pandas, matplotlib, pyplot, plotly y numpy.

El objetivo de la red neuronal es que, dado un dataset, clasifique cual es la combinación óptima de filtro y códec para cada split. De esta manera, una vez se tienen las predicciones para todos los splits que forman el dataset, se aplicaría a todo el dataset la combinación de parámetros de compresión que la red haya predicho con mayor frecuencia.

Como he descrito anteriormente, los inputs de la red son el ratio y velocidad de compresión de entropía. A cada par de estos inputs de un determinado split se le debe asignar una etiqueta, que en las redes neuronales de multi clasificación se refiere a la categoría óptima a la que pertenece ese par de inputs.

Por otra parte, en los datasets puede haber un tipo especial de valor, los denominados special values. Estos valores son conjuntos de bits que, al ser reordenados por alguno de los dos filtros, se convierten en agrupaciones de un mismo número. Como consecuencia, su compresión es prácticamente inmediata y por lo tanto producen unos valores de ratio y velocidad de compresión extremadamente elevados. Por consiguiente, se ha decidido eliminar estos valores del conjunto de datos a utilizar como input de la red neuronal.

Una vez realizado este filtrado, se han agrupado los datos según el filtro usado. A continuación se ha procedido a calcular cuál es la categoría que produce mejores resultados para cada split. Para ello se ha definido una función de puntuación (score), la cual se trata de maximizar. Esta función consiste en sumar el ratio y velocidad de compresión normalizados:

$$S(cratio_i, speed_i) = theta * \frac{cratio_i - mean\_cratios}{std\_cratios} + (1 - theta) * \frac{speed_i - mean\_speeds}{std\_speeds} \quad (4.1)$$

para i = 1, 2, ..., N siendo N el número de splits del dataset. El peso theta dependerá del favour seleccionado.

Una vez calculado el score de las categorías de cada filtro, seleccionamos la mayor de ellas ya que a mayor ratio y velocidad, mejor es la compresión. Esta categoría seleccionada como la mejor pasa a ser la etiqueta que se le introducirá a la red, la cual estará relacionada con el ratio y velocidad de compresión del entropy_mode.

4.3. Construcción del modelo de red neuronal

Una vez preparados los inputs de la red, el siguiente paso es crear y construir la red. Para ello he utilizado las librerías keras y tensorflow.

4.3.1. Arquitectura de la red

Las redes neuronales creadas están formadas por una capa de entrada, una capa oculta y otra capa de salida. La capa de entrada dispone de dos neuronas, una para cada input; la capa oculta tiene 2 neuronas, y finalmente la capa de salida tiene cuatro neuronas, ya que la clasificación se realiza entre cuatro categorías. En cuanto a las funciones de optimización, en todas sus capas menos la de salida se ha usado la función ReLu (3.2). Además, en la capa de salida la función de activación utilizada es la función softmax (3.5).

4.3.2. Compilación de la red

Una vez creada la arquitectura de las redes, el siguiente paso es compilarlas. Para ello hay que definir tres parámetros: optimizador, función de pérdida y métricas. El optimizador es el mecanismo por el cual el modelo se actualiza basándose en los datos de entrenamiento que ve, con el objetivo de mejorar su rendimiento. La función de pérdida expresa cómo el modelo será capaz de medir su rendimiento en los datos de entrenamiento, y por lo tanto cómo será capaz de dirigirse en la dirección correcta.

En este caso, el optimizador utilizado es Adam, el cual ha sido descrito anteriormente. La función de pérdida empleada es la Categorical Cross Entropy, que también ha sido explicada en el capítulo 3.3. Finalmente, en cuanto a las métricas a monitorizar durante el entrenamiento, existen diversas pero la escogida es la precisión (accuracy), es decir, la fracción de datos que han sido clasificados correctamente.

4.3.3. Adecuación del modelo (fitting)

Tras haber construido y compilado las redes, el último paso es la adecuación. En este paso introducimos los inputs y etiquetas de entrenamiento y de test, especificando también las épocas que queremos que se realicen en la red. Ahora ya se puede proceder al entrenamiento de las redes neuronales.

4.3.4. Resultados de la red

Una vez entrenadas, las redes neuronales podemos observar su rendimiento y su función de pérdida representadas mediante gráficas. Como ya se ha explicado, la precisión es la fracción de datos que han sido clasificados correctamente, por lo tanto su valor varía entre 0 y 1, y si la red funciona correctamente se espera que vaya aumentando, ya que a mayor precisión, significa que la red predice mejor. En cuanto a la función de pérdida, como se ha explicado en el capítulo 4.3.3, indica cómo de bien la red clasifica correctamente la entrada y por lo tanto se quiere minimizar. Entonces se espera que este valor vaya descendiendo conforme avanzan las épocas de entrenamiento, con el objetivo de aproximarse a un valor lo más cercano a 0. A la hora de entrenar la red, se ha indicado que se reserve un 10 % de los datos para test. Por lo tanto, en las gráficas hay información sobre el entrenamiento (en color rojo) y sobre el test (en color verde).

A continuación se va a mostrar estas gráficas para los tres modelos de redes neuronales. En las gráficas situadas a la izquierda se representa la precisión de entrenamiento y de test, y en las de la derecha el valor de la función de pérdida de entrenamiento y de test.

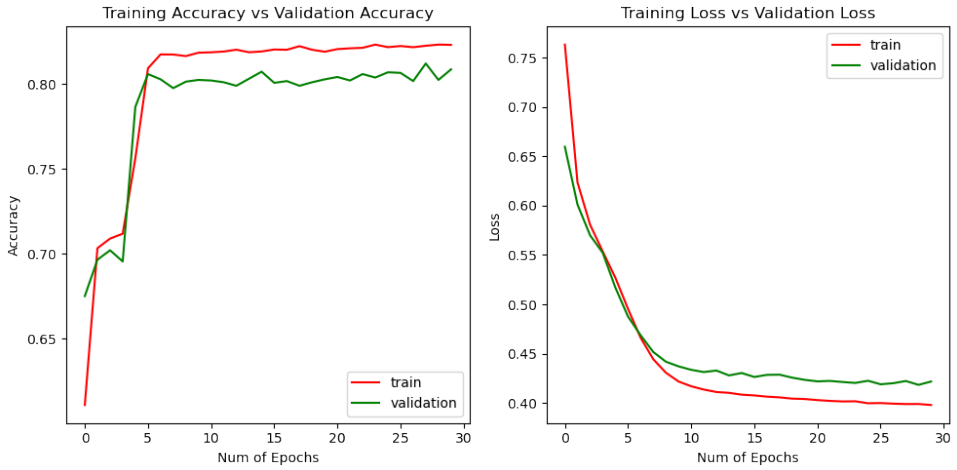

Figura 4.5: Gráficas de precisión y función de pérdida para el favour en modo max_cratio.

Como podemos observar en la figura 4.5, tanto la precisión de entrenamiento como la de test aumenta hasta estabilizarse en un 80 %. Estos valores indican una gran precisión de la red. En cuanto a la función de pérdida, tanto la pérdida de entrenamiento como la de test disminuye hasta estabilizarse en 0.4. Estos valores de pérdida son muy buenos e indican que la red predice muy bien.

Como podemos observar en la figura 4.6, tanto la precisión de entrenamiento como la de test aumenta hasta estabilizarse en un 70 %. Estos valores indican una gran precisión de la red.

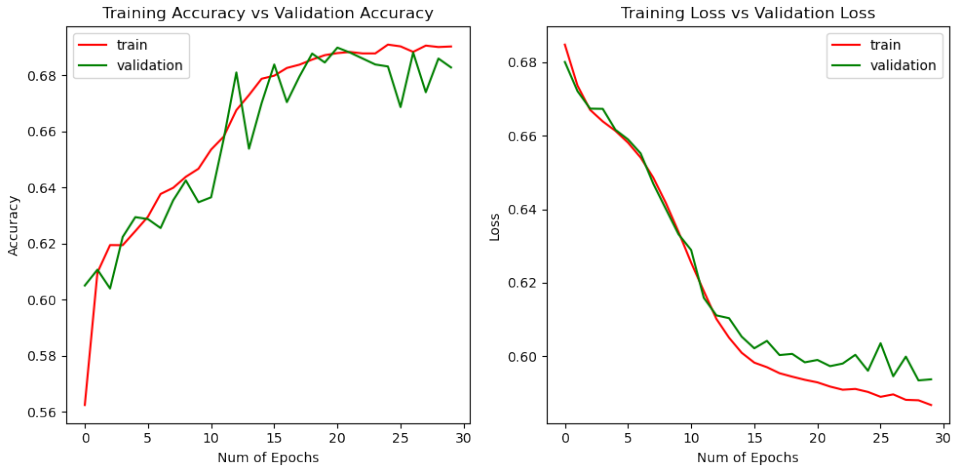

Figura 4.6: Gráficas de precisión y función de pérdida para el favour en modo max_speed.

En cuanto a la función de pérdida, tanto la pérdida de entrenamiento como la de test disminuye hasta estabilizarse en 0.6. Estos valores de pérdida son muy buenos e indican que la red predice muy bien.

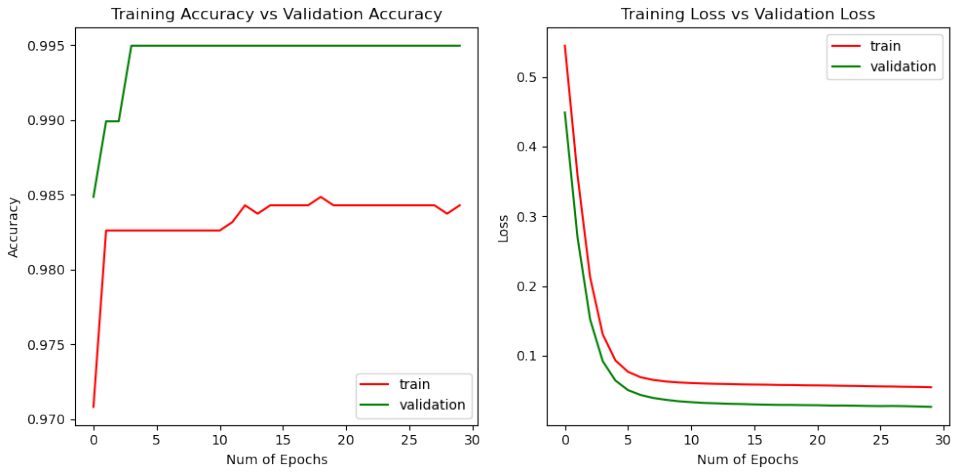

Figura 4.7: Gráficas de precisión y función de pérdida para el favour en modo balance.

Como podemos observar en la figura 4.7, tanto la precisión de entrenamiento como la de test aumenta hasta estabilizarse en prácticamente un 100 %. Estos valores indican una gran precisión de la red. En cuanto a la función de pérdida, tanto la pérdida de entrenamiento como la de test disminuye hasta estabilizarse en prácticamente 0. Estos valores de pérdida son muy

buenos e indican que la red predice muy bien.

4.3.5. Estudio de las observaciones mal clasificadas

Una vez mostrados los resultados predichos por la red, podemos comparar estas predicciones con sus categorías correctas. En las siguientes imágenes podemos observar, para cada uno de los tres modelos de red neuronal, en primer lugar las predicciones de la red para todas las categorías (imágenes 4.8, 4.12 y 4.15) y en segundo lugar una comparación de cada categoría predicha con los valores predichos erróneamente de esa categoría, incluyendo un razonamiento de a qué se deben esos errores (imágenes 4.9, 4.10 y 4.11 para el modelo de red con el favour max_cratio, imágenes 4.13 y 4.13 para el modelo de red con el favour max_speed y imágenes 4.16 y 4.16 para el modelo de red con el favour balance).

En las tablas 4.2, 4.3 y 4.4 podemos observar las equivalencias de los números de las gráficas con sus categorías correspondientes. Además, las categorías que aparecen el las gráficas con un número negativo, representan la categoría con número positivo pero predichas erróneamente por la red, es decir, los errores de predicción de la red para esa categoría específica.

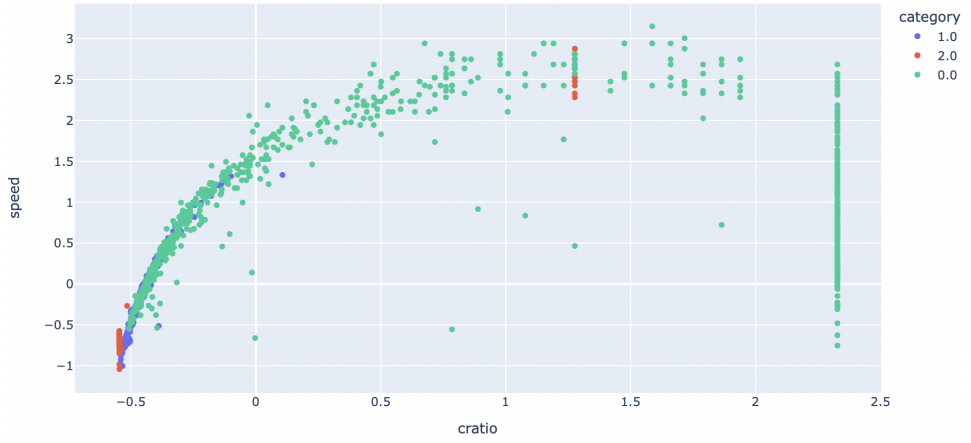

Figura 4.8: Gráfica de predicciones de la red para el favour en modo max_cratio.

En las figuras 4.18, 4.19 y 4.20 podemos observar una tabla cruzada, una técnica para el estudio de la relación existente entre dos variables categóricas, en nuestro caso entre las categorías predichas y las reales (etiquetas). En el eje horizontal se encuentran las categorias predichas por la red para cada observación del dataset, y en el eje vertical las etiquetas asociadas a esas mismas observaciones. Esta tabla cruzada se puede expresar con una matriz P de J filas y J columnas, $P = [[p_{1,1}, p_{1,2}, .., p_{1,J}], [p_{2,1}, p_{2,2}, .., p_{2,J}], ..., [p_{J,1}, p_{J,2}, .., p_{J,J}]]$. Cada elemento $p_{i,j}$ representa el número de veces que la red ha predicho la categoría j cuando realmente su

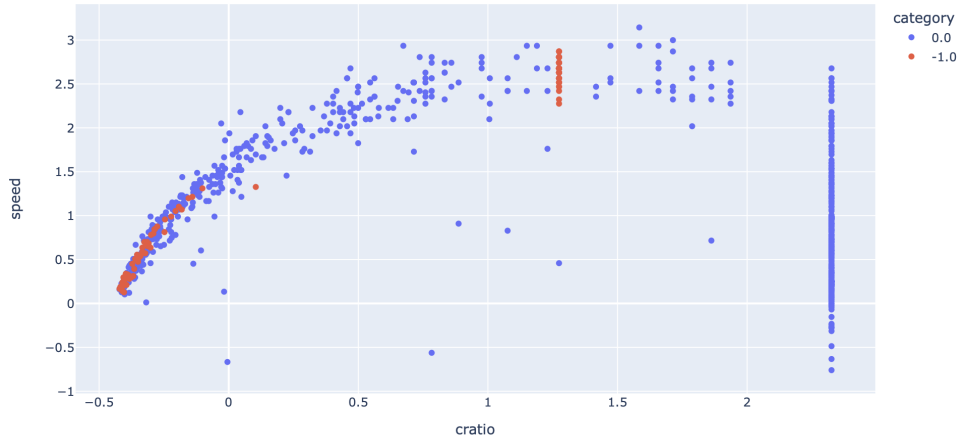

Figura 4.9: Gráfica de predicciones de la red para el favour en modo max_cratio y la categoría 0.

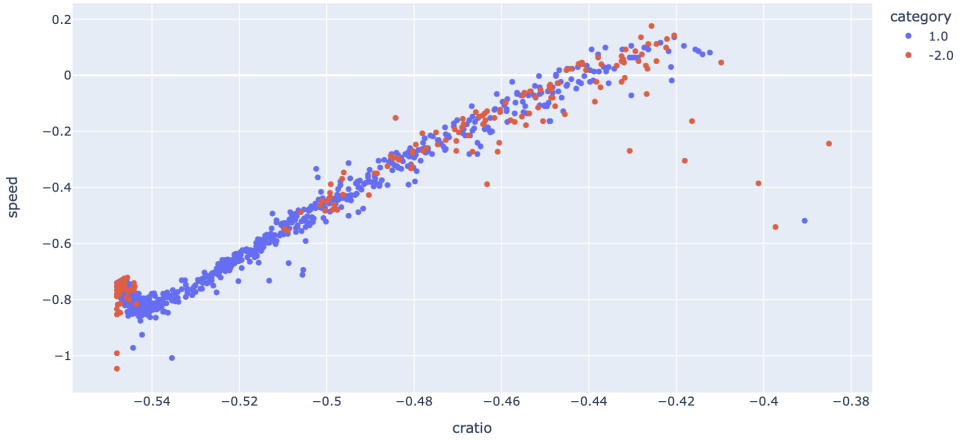

Figura 4.10: Gráfica de predicciones de la red para el favour en modo max_cratio y la categoría 1.

categoría real o etiqueta es la clase i, donde i, $j \in 0, 1, .., J-1$, ya que J es el número de clases. En cada figura se representa la tabla cruzada de un modelo de red neuronal donde se optimiza cada uno de los tipos de favour.

Una vez observados los errores de las predicciones de la red para cada categoría en cada favour, el siguiente paso es analizar estos errores y estudiar por qué se han producido. Este trabajo queda planteado como línea futura de investigación.

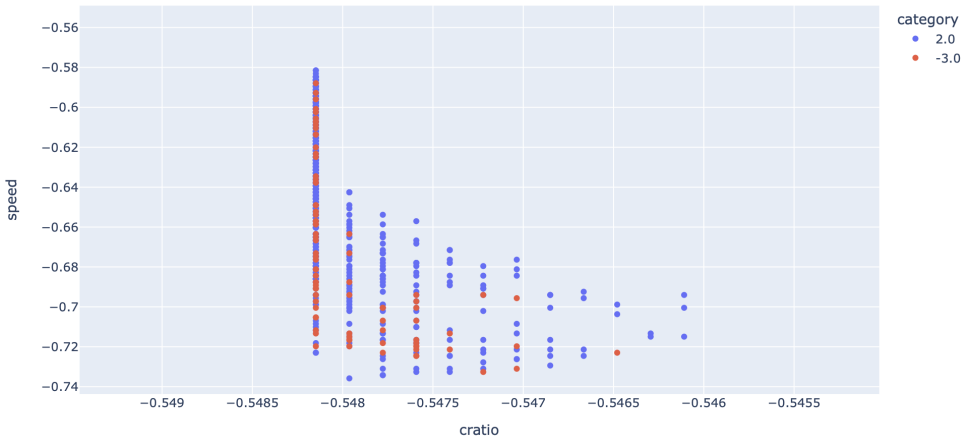

Figura 4.11: Gráfica de predicciones de la red para el favour en modo max_cratio y la categoría 2.

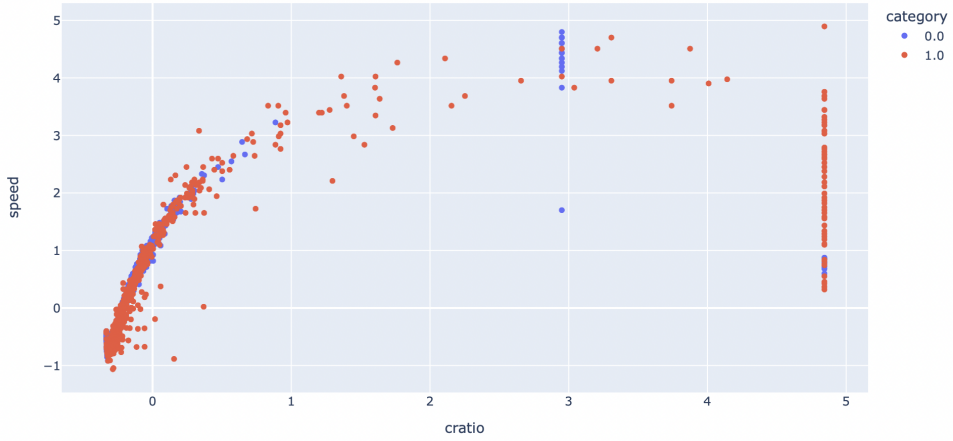

Figura 4.12: Gráfica de predicciones de la red para el favour en modo max_speed.

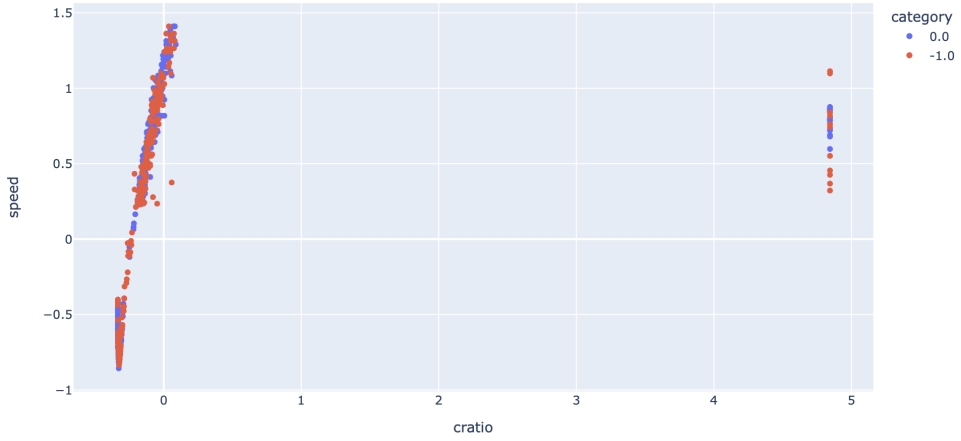

Figura 4.13: Gráfica de predicciones de la red para el favour en modo max_speed y la categoría 0.

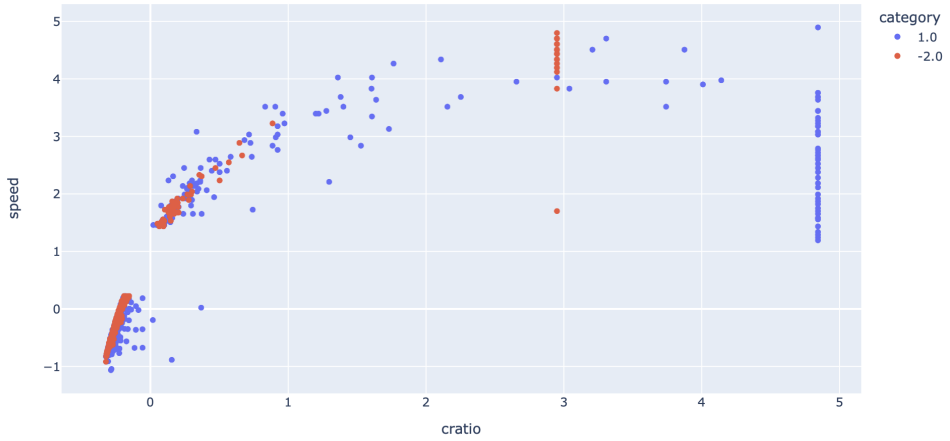

Figura 4.14: Gráfica de predicciones de la red para el favour en modo max_speed y la categoría 1.

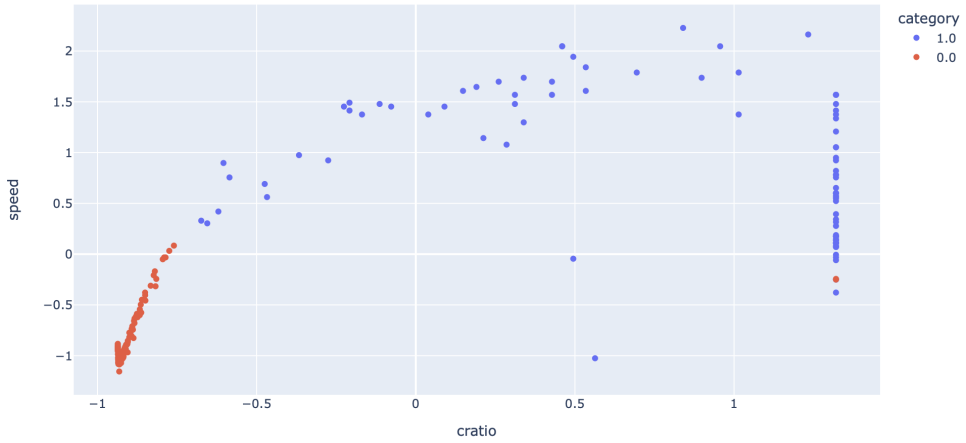

Figura 4.15: Gráfica de predicciones de la red para el favour en modo balance.

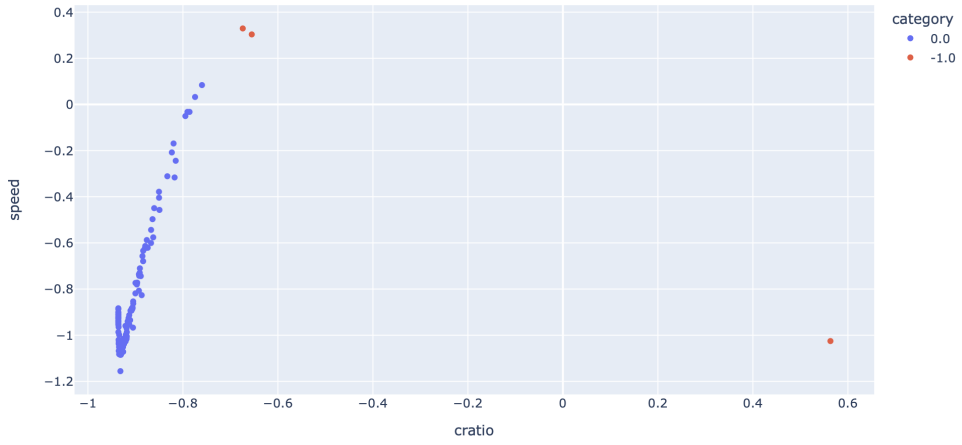

Figura 4.16: Gráfica de predicciones de la red para el favour en modo balance y la categoría 0.

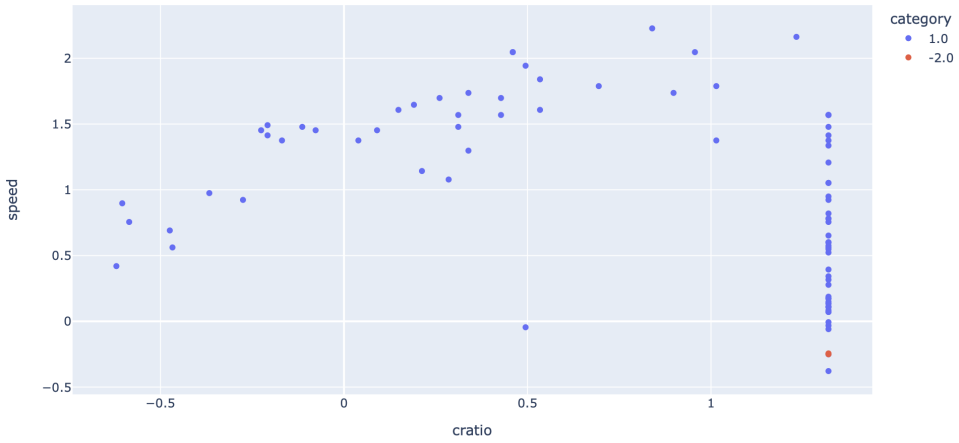

Figura 4.17: Gráfica de predicciones de la red para el favour en modo balance y la categoría 1.

col_0	0	1	2
row_0			
0	782	57	53
1	123	834	52
2	0	84	901

Figura 4.18: Tabla cruzada para el favour en modo max_cratio.

col_0	0	1
row_0		
0	929	379
1	511	1012

Figura 4.19: Tabla cruzada para el favour en modo max_speed.

col_0	0	1
row_0		
0	107	3
1	2	86

Figura 4.20: Tabla cruzada para el favour en modo balance.

Capítulo 5

Conclusiones

El objetivo principal del proyecto era desarrollar una red neuronal artificial que fuese capaz de determinar la combinación de parámetros de compresión óptima dado un dataset determinado.

Para ello se ha desarrollado un proyecto de investigación que ha implicado desde la generación de los datos, pasando por la manipulación de estos, hasta la construcción de los modelos de redes neuronales artificiales y la evaluación de su rendimiento. Para poder llevar a cabo estas tareas ha sido necesario un estudio teórico previo de las redes neuronales de clasificación, así como su aplicación práctica.

Bibliografía

[1] Augustin Cauchy. Méthode générale pour la résolution des systemes d'équations simultanées. *Comp. Rend. Sci. Paris, :*, 25:536–538, 1847. `https://cs.uwaterloo.ca/~y328yu/classics/cauchy-en.pdf`.

[2] Robert Mansel Gower, Nicolas Loizou, Xun Qian, Alibek Sailanbayev, Egor Shulgin, and Peter Richtárik. Sgd: General analysis and improved rates. In *International Conference on Machine Learning*, pages 5200–5209. PMLR, 2019.

[3] Geoffrey Hinton. Lecture 6e rmsprop: Divide the gradient by a running average of its recent magnitude. `http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf`.

[4] B Lei P. Diederik, J. Kingma. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings. San Diego, USA, 7-9 Mayo.* `https://arxiv.org/pdf/1412.6980.pdf`.

[5] Marta Aicart Pauner. Dl_compression. `https://github.com/MartaAicart/DL_Compression.git`.

[6] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[7] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[8] The Blosc Development Team. C-blosc2: A fast, compressed and persistent data store library for c. `https://c-blosc2.readthedocs.io/en/latest/`.

Anexo A

Generador de datos

A.1. Código entropy_probe.c

En este anexo se presenta el código realizado en C que genera un documento csv donde para cada porción del dataset, se obtiene información sobre su ratio y velocidad de compresión, entre otros. El fichero del código se llama **entropy_probe.c** que incluye el programa principal y las funciones auxiliares que utiliza.

```
/*
  Copyright (C) 2021  The Blosc Developers <blosc@blosc.org>
  https://blosc.org
  License: BSD 3-Clause (see LICENSE.txt)

  Example program demonstrating use of the Blosc entropy probe codec from C
    code.

  To compile this program:

  $ gcc entropy_probe.c -o entropy_probe -lblosc2

  To run:

  $ ./entropy_probe

  To run in entropy mode:

  $ ./entropy_probe -e output.csv
*/

#include <stdio.h>
```

```
#include "blosc2.h"

#define ENTROPY_PROBE_ID 244

#define KB  1024.
#define MB  (1024*KB)
#define GB  (1024*MB)

#define MAX_COPY 32U
#define MAX_DISTANCE 8191
#define MAX_FARDISTANCE (65535 + MAX_DISTANCE - 1)

// The hash length (1 << HASH_LOG2) can be tuned for performance (12 -> 15)
#define HASH_LOG2 (12U)
//#define HASH_LOG2 (13U)
//#define HASH_LOG2 (14U)

#define HASH_FUNCTION(v, s, h) {      \
  v = (s * 2654435761U) >> (32U - h); \
}

#define BLOSCLZ_READU16(p) *((const uint16_t*)(p))
#define BLOSCLZ_READU32(p) *((const uint32_t*)(p))

#define LITERAL2(ip, anchor, copy) {                    \
  oc++; anchor++;                                       \
  ip = anchor;                                          \
  copy++;                                               \
  if (copy == MAX_COPY) {                               \
    copy = 0;                                           \
    oc++;                                               \
  }                                                     \
}


static uint8_t *get_run(uint8_t *ip, const uint8_t *ip_bound, const uint8_t *
    ref) {
  uint8_t x = ip[-1];
  int64_t value, value2;
  /* Broadcast the value for every byte in a 64-bit register */
  memset(&value, x, 8);
  /* safe because the outer check against ip limit */
  while (ip < (ip_bound - sizeof(int64_t))) {
    value2 = ((int64_t *) ref)[0];
    if (value != value2) {
      /* Return the byte that starts to differ */
      while (*ref++ == x) ip++;
      return ip;
    } else {
      ip += 8;
      ref += 8;
    }
```

```
  }
  /* Look into the remainder */
  while ((ip < ip_bound) && (*ref++ == x)) ip++;
  return ip;
}


static uint8_t *get_match(uint8_t *ip, const uint8_t *ip_bound, const uint8_t *
    ref) {
  while (ip < (ip_bound - sizeof(int64_t))) {
    if (*(int64_t *) ref != *(int64_t *) ip) {
      /* Return the byte that starts to differ */
      while (*ref++ == *ip++) {}
      return ip;
    } else {
      ip += sizeof(int64_t);
      ref += sizeof(int64_t);
    }
  }
  /* Look into the remainder */
  while ((ip < ip_bound) && (*ref++ == *ip++)) {}
  return ip;
}


static uint8_t *get_run_or_match(uint8_t *ip, uint8_t *ip_bound, const uint8_t
    *ref, bool run) {
  if (run) {
    ip = get_run(ip, ip_bound, ref);
  } else {
    ip = get_match(ip, ip_bound, ref);
  }

  return ip;
}


// Get a guess for the compressed size of a buffer
static float get_cratio(const uint8_t *ibase, int maxlen, int minlen, int
    ipshift) {
  const uint8_t *ip = ibase;
  int32_t oc = 0;
  const uint16_t hashlen = (1U << (uint8_t) HASH_LOG2);
  uint16_t htab[1U << (uint8_t) HASH_LOG2];
  uint32_t hval;
  uint32_t seq;
  uint8_t copy;
  // Make a tradeoff between testing too much and too little
  uint16_t limit = (maxlen > hashlen) ? hashlen : maxlen;
  const uint8_t *ip_bound = ibase + limit - 1;
  const uint8_t *ip_limit = ibase + limit - 12;

```

```
  // Initialize the hash table to distances of 0
  memset(htab, 0, hashlen * sizeof(uint16_t));

  /* we start with literal copy */
  copy = 4;
  oc += 5;

  /* main loop */
  while (ip < ip_limit) {
    const uint8_t *ref;
    unsigned distance;
    const uint8_t *anchor = ip;    /* comparison starting-point */

    /* find potential match */
    seq = BLOSCLZ_READU32(ip);
    HASH_FUNCTION(hval, seq, HASH_LOG2)
    ref = ibase + htab[hval];

    /* calculate distance to the match */
    distance = (unsigned int) (anchor - ref);

    /* update hash table */
    htab[hval] = (uint16_t) (anchor - ibase);

    if (distance == 0 || (distance >= MAX_FARDISTANCE)) {
      LITERAL2(ip, anchor, copy)
      continue;
    }

    /* is this a match? check the first 4 bytes */
    if (BLOSCLZ_READU32(ref) == BLOSCLZ_READU32(ip)) {
      ref += 4;
    } else {
      /* no luck, copy as a literal */
      LITERAL2(ip, anchor, copy)
      continue;
    }

    /* last matched byte */
    ip = anchor + 4;

    /* distance is biased */
    distance--;

    /* get runs or matches; zero distance means a run */
    ip = get_run_or_match((uint8_t*)ip, (uint8_t*)ip_bound, ref, !distance);

    ip -= ipshift;
    int32_t len = (int32_t) (ip - anchor);
    if (len < minlen) {
      LITERAL2(ip, anchor, copy)
      continue;
```

```
    }

    /* if we haven't copied anything, adjust the output counter */
    if (!copy)
      oc--;
    /* reset literal counter */
    copy = 0;

    /* encode the match */
    if (distance < MAX_DISTANCE) {
      if (len >= 7) {
        oc += ((len - 7) / 255) + 1;
      }
      oc += 2;
    } else {
      /* far away, but not yet in the another galaxy... */
      if (len >= 7) {
        oc += ((len - 7) / 255) + 1;
      }
      oc += 4;
    }

    /* update the hash at match boundary */
    seq = BLOSCLZ_READU32(ip);
    HASH_FUNCTION(hval, seq, HASH_LOG2)
    htab[hval] = (uint16_t) (ip++ - ibase);
    ip++;
    /* assuming literal copy */
    oc++;
  }

  float ic = (float) (ip - ibase);
  return ic / (float) oc;
}


int entropy_probe(const uint8_t *input, int32_t input_len,
                  uint8_t *output, int32_t output_len,
                  uint8_t meta,
                  blosc2_cparams *cparams, const void *chunk) {
  if (cparams->splitmode != BLOSC_ALWAYS_SPLIT) {
    BLOSC_TRACE_ERROR("Entropy probe can only be used in SPLIT mode. Aborting
    !");
    return BLOSC2_ERROR_CODEC_PARAM;
  }
  // Get the cratio.  minlen and ipshift are decent defaults, but one can try
    with (4, 4) or (3, 4) or (4, 3)
  float cratio = get_cratio(input, input_len, 3, 3);
  int cbytes = (int) ((float)input_len / cratio);
  if (cbytes > input_len) {
    cbytes = input_len;
  }
```

```
224   return cbytes;
225 }
226
227
228 // This function receives an instrumented chunk having nstreams
229 int extr_data(FILE *csv_file, int nchunk, uint8_t *chunk, int nstreams, int
      category) {
230   blosc2_instr *instr_data = (blosc2_instr *) chunk;
231
232   for (int nstream = 0; nstream < nstreams; nstream++) {
233     float cratio = instr_data->cratio;
234     float cspeed = instr_data->cspeed;
235     bool special_val = instr_data->flags[0];
236     if (!special_val) {
237       // Fill csv file
238       int special_vals = 0;
239       fprintf(csv_file, "%.3g, %.3g, %d, %d, %d\n", cratio, cspeed,
      special_vals, nchunk, category);
240       printf("Chunk %d, block %d: cratio %.3g, speed %.3g\n", nchunk, nstream,
      cratio, cspeed);
241     } else {
242         // Fill csv file
243         int special_vals = 1;
244         fprintf(csv_file, "%.3g, %.3g, %d, %d, %d\n", cratio, cspeed,
      special_vals, nchunk, category);
245         printf("Chunk %d, block %d: cratio %.3g, speed %.3g\n", nchunk, nstream
      , cratio, cspeed);
246     }
247     instr_data++;
248   }
249
250   return 0;
251 }
252
253 //Categorize the data according to the combination of filter-codec-splitmode
254 int categorize_data(char codec, char filter, char splitmode) {
255     printf(" AAAAAAAAAAAAAAAAAAAAAAAAAAAAA el nombre del codec es: %c\n", codec
      );
256
257     int category;
258     if (codec == BLOSC_BLOSCLZ && filter == BLOSC_NOFILTER && splitmode ==
      BLOSC_ALWAYS_SPLIT) {
259         category = 0;
260     } else if (codec == BLOSC_BLOSCLZ && filter == BLOSC_SHUFFLE && splitmode
      == BLOSC_ALWAYS_SPLIT) {
261         category = 1;
262     } else if (codec == BLOSC_BLOSCLZ && filter == BLOSC_BITSHUFFLE &&
      splitmode == BLOSC_ALWAYS_SPLIT) {
263         category = 2;
264     } else if (codec == BLOSC_LZ4 && filter == BLOSC_NOFILTER && splitmode ==
      BLOSC_ALWAYS_SPLIT) {
265         category = 3;
```

```
266     } else if (codec == BLOSC_LZ4 && filter == BLOSC_SHUFFLE && splitmode ==
    BLOSC_ALWAYS_SPLIT) {
267         category = 4;
268     } else if (codec == BLOSC_LZ4 && filter == BLOSC_BITSHUFFLE && splitmode ==
     BLOSC_ALWAYS_SPLIT) {
269         category = 5;
270     } else if (codec == BLOSC_LZ4HC && filter == BLOSC_NOFILTER && splitmode ==
     BLOSC_ALWAYS_SPLIT) {
271         category = 6;
272     } else if (codec == BLOSC_LZ4HC && filter == BLOSC_SHUFFLE && splitmode ==
    BLOSC_ALWAYS_SPLIT) {
273         category = 7;
274     } else if (codec == BLOSC_LZ4HC && filter == BLOSC_BITSHUFFLE && splitmode
    == BLOSC_ALWAYS_SPLIT) {
275         category = 8;
276     } else if (codec == BLOSC_ZLIB && filter == BLOSC_NOFILTER && splitmode ==
    BLOSC_ALWAYS_SPLIT) {
277         category = 9;
278     } else if (codec == BLOSC_ZLIB && filter == BLOSC_SHUFFLE && splitmode ==
    BLOSC_ALWAYS_SPLIT) {
279         category = 10;
280     } else if (codec == BLOSC_ZLIB && filter == BLOSC_BITSHUFFLE && splitmode
    == BLOSC_ALWAYS_SPLIT) {
281         category = 11;
282     } else if (codec == BLOSC_ZSTD && filter == BLOSC_NOFILTER && splitmode ==
    BLOSC_ALWAYS_SPLIT) {
283         category = 12;
284     } else if (codec == BLOSC_ZSTD && filter == BLOSC_SHUFFLE && splitmode ==
    BLOSC_ALWAYS_SPLIT) {
285         category = 13;
286     } else if (codec == BLOSC_ZSTD && filter == BLOSC_BITSHUFFLE && splitmode
    == BLOSC_ALWAYS_SPLIT) {
287         category = 14;
288 
289     } else if (codec == BLOSC_BLOSCLZ && filter == BLOSC_NOFILTER && splitmode
    == BLOSC_NEVER_SPLIT) {
290         category = 15;
291     } else if (codec == BLOSC_BLOSCLZ && filter == BLOSC_SHUFFLE && splitmode
    == BLOSC_NEVER_SPLIT) {
292         category = 16;
293     } else if (codec == BLOSC_BLOSCLZ && filter == BLOSC_BITSHUFFLE &&
    splitmode == BLOSC_NEVER_SPLIT) {
294         category = 17;
295     } else if (codec == BLOSC_LZ4 && filter == BLOSC_NOFILTER && splitmode ==
    BLOSC_NEVER_SPLIT) {
296         category = 18;
297     } else if (codec == BLOSC_LZ4 && filter == BLOSC_SHUFFLE && splitmode ==
    BLOSC_NEVER_SPLIT) {
298         category = 19;
299     } else if (codec == BLOSC_LZ4 && filter == BLOSC_BITSHUFFLE && splitmode ==
     BLOSC_NEVER_SPLIT) {
300         category = 20;
```

```
301     } else if (codec == BLOSC_LZ4HC && filter == BLOSC_NOFILTER && splitmode ==
         BLOSC_NEVER_SPLIT) {
302         category = 21;
303     } else if (codec == BLOSC_LZ4HC && filter == BLOSC_SHUFFLE && splitmode ==
        BLOSC_NEVER_SPLIT) {
304         category = 22;
305     } else if (codec == BLOSC_LZ4HC && filter == BLOSC_BITSHUFFLE && splitmode
        == BLOSC_NEVER_SPLIT) {
306         category = 23;
307     } else if (codec == BLOSC_ZLIB && filter == BLOSC_NOFILTER && splitmode ==
        BLOSC_NEVER_SPLIT) {
308         category = 24;
309     } else if (codec == BLOSC_ZLIB && filter == BLOSC_SHUFFLE && splitmode ==
        BLOSC_NEVER_SPLIT) {
310         category = 25;
311     } else if (codec == BLOSC_ZLIB && filter == BLOSC_BITSHUFFLE && splitmode
        == BLOSC_NEVER_SPLIT) {
312         category = 26;
313     } else if (codec == BLOSC_ZSTD && filter == BLOSC_NOFILTER && splitmode ==
        BLOSC_NEVER_SPLIT) {
314         category = 27;
315     } else if (codec == BLOSC_ZSTD && filter == BLOSC_SHUFFLE && splitmode ==
        BLOSC_NEVER_SPLIT) {
316         category = 28;
317     } else if (codec == BLOSC_ZSTD && filter == BLOSC_BITSHUFFLE && splitmode
        == BLOSC_NEVER_SPLIT) {
318         category = 29;
319     }
320 
321     return category;
322 }
323 
324 void print_compress_info(void) {
325   char *name = NULL, *version = NULL;
326   int ret;
327 
328   printf("Blosc version: %s (%s)\n", BLOSC_VERSION_STRING, BLOSC_VERSION_DATE);
329 
330   printf("List of supported compressors in this build: %s\n",
331          blosc_list_compressors());
332 
333   printf("Supported compression libraries:\n");
334   ret = blosc_get_complib_info("blosclz", &name, &version);
335   if (ret >= 0) printf("  %s: %s\n", name, version);
336   free(name);
337   free(version);
338   ret = blosc_get_complib_info("lz4", &name, &version);
339   if (ret >= 0) printf("  %s: %s\n", name, version);
340   free(name);
341   free(version);
342   ret = blosc_get_complib_info("zlib", &name, &version);
343   if (ret >= 0) printf("  %s: %s\n", name, version);
```

```
  free(name);
  free(version);
  ret = blosc_get_complib_info("zstd", &name, &version);
  if (ret >= 0) printf("  %s: %s\n", name, version);
  free(name);
  free(version);
}


int main(int argc, char *argv[]) {
  char usage[256];
  char csv_filename[256];
  char data_filename[256];
  bool entropy_probe_mode = false;

  print_compress_info();

  strcpy(usage, "Usage: entropy_probe [-e] data_filename");

  if (argc < 2) {
    printf("%s\n", usage);
    exit(1);
  }

  if (argc >= 3) {
    if (strcmp("-e", argv[1]) != 0) {
      printf("%s\n", usage);
      exit(1);
    }
    strcpy(data_filename, argv[2]);
    entropy_probe_mode = true;
  }
  else {
    strcpy(data_filename, argv[1]);
  }
  printf("fitxer %s\n", data_filename);
  blosc2_cparams cparams = BLOSC2_CPARAMS_DEFAULTS;
  cparams.instr_codec = true;
  cparams.blocksize = 256 * (int)KB;
  cparams.splitmode = BLOSC_ALWAYS_SPLIT;

  if (entropy_probe_mode) {
    // The entropy probe detector is meant to always be used in SPLIT mode
    cparams.splitmode = BLOSC_ALWAYS_SPLIT;
  }
  cparams.typesize = sizeof(float);

  if (entropy_probe_mode) {
    blosc2_codec udcodec;
    udcodec.compcode = ENTROPY_PROBE_ID;
    udcodec.compver = 1;
    udcodec.complib = 1;
```

```
  udcodec.compname = "entropy_probe";
  udcodec.encoder = entropy_probe;
  udcodec.decoder = NULL;
  blosc2_register_codec(&udcodec);
  cparams.compcode = ENTROPY_PROBE_ID;
} //else {
//  cparams.compcode = BLOSC_BLOSCLZ;
//}

blosc2_dparams dparams = BLOSC2_DPARAMS_DEFAULTS;

blosc2_schunk *schunk = blosc2_schunk_open(data_filename);
if (schunk == NULL) {
  printf("Cannot open the data file\n");
  exit(1);
}
uint8_t *chunk = malloc(schunk->chunksize);
uint8_t *chunk2 = malloc(schunk->chunksize);
printf("nchunks in dataset: %d\n", schunk->nchunks);

if (!entropy_probe_mode) {
    // Loop over different filters and codecs
    int v_codecs[] = {0, 1, 2, 4, 5};
    int v_splits[] = {1, 2};

    for (int ncodec = 0; ncodec < sizeof(v_codecs); ++ncodec) {
        for (int nfilter = 0; nfilter <= BLOSC_BITSHUFFLE; ++nfilter) {
            for (int nsplit = 0; nsplit < BLOSC_NEVER_SPLIT; ++nsplit) {

                cparams.splitmode = v_splits[nsplit];
                cparams.compcode = v_codecs[ncodec];
                cparams.filters[BLOSC2_MAX_FILTERS - 1] = nfilter;
                blosc2_context *cctx = blosc2_create_cctx(cparams);
                blosc2_context *dctx = blosc2_create_dctx(dparams);

                const char *compname;
                switch (cparams.compcode) {
                    case BLOSC_BLOSCLZ:
                        compname = BLOSC_BLOSCLZ_COMPNAME;
                        break;
                    case BLOSC_LZ4:
                        compname = BLOSC_LZ4_COMPNAME;
                        break;
                    case BLOSC_LZ4HC:
                        compname = BLOSC_LZ4HC_COMPNAME;
                        break;
                    case BLOSC_ZLIB:
                        compname = BLOSC_ZLIB_COMPNAME;
                        break;
                    case BLOSC_ZSTD:
                        compname = BLOSC_ZSTD_COMPNAME;
                        break;
```

```
                    case ENTROPY_PROBE_ID:
                        compname = "entropy";
                        break;
                    default:
                        printf("Unsupported codec!");
                        exit(1);
                }

                char *sfilter;
                switch (nfilter) {
                    case BLOSC_NOFILTER:
                        sfilter = "nofilter";
                        break;
                    case BLOSC_SHUFFLE:
                        sfilter = "shuffle";
                        break;
                    case BLOSC_BITSHUFFLE:
                        sfilter = "bitshuffle";
                        break;
                    default:
                        printf("Unsupported filter!");
                        exit(1);
                }

                char *ssplit;
                switch (cparams.splitmode) {
                    case BLOSC_ALWAYS_SPLIT:
                        ssplit = "split";
                        break;
                    case BLOSC_NEVER_SPLIT:
                        ssplit = "nosplit";
                        break;

                    default:
                        printf("Unsupported splitmode!");
                        exit(1);
                }

                // Create csv file
                sprintf(csv_filename, "%s-%s-%s.csv", compname, sfilter,
    ssplit);
                printf("Codec : %s\n", compname);
                printf("Filter : %s\n", sfilter);
                printf("Splitmode: %s\n", ssplit);

                FILE *csv_file = fopen(csv_filename, "w");
                if (csv_file == NULL) {
                    printf("Error creating the file\n");
                    return -1;
                }

                int category;
```

```
499                 if (strcmp(compname, BLOSC_BLOSCLZ_COMPNAME) == 0 && strcmp(
    sfilter, "nofilter") == 0 && strcmp(ssplit, "split") == 0) {
500                     category = 0;
501                 } else if (strcmp(compname, BLOSC_BLOSCLZ_COMPNAME) == 0 &&
    strcmp(sfilter, "shuffle") == 0 && strcmp(ssplit, "split") == 0) {
502                     category = 1;
503                 } else if (strcmp(compname, BLOSC_BLOSCLZ_COMPNAME) == 0 &&
    strcmp(sfilter, "bitshuffle") == 0 && strcmp(ssplit, "split") == 0) {
504                     category = 2;
505                 } else if (strcmp(compname, BLOSC_LZ4_COMPNAME) == 0 &&
    strcmp(sfilter, "nofilter") == 0 && strcmp(ssplit, "split") == 0) {
506                     category = 3;
507                 } else if (strcmp(compname, BLOSC_LZ4_COMPNAME) == 0 &&
    strcmp(sfilter, "shuffle") == 0 && strcmp(ssplit, "split") == 0) {
508                     category = 4;
509                 } else if (strcmp(compname, BLOSC_LZ4_COMPNAME) == 0 &&
    strcmp(sfilter, "bitshuffle") == 0 && strcmp(ssplit, "split") == 0) {
510                     category = 5;
511                 } else if (strcmp(compname, BLOSC_LZ4HC_COMPNAME) == 0 &&
    strcmp(sfilter, "nofilter") == 0 && strcmp(ssplit, "split") == 0) {
512                     category = 6;
513                 } else if (strcmp(compname, BLOSC_LZ4HC_COMPNAME) == 0 &&
    strcmp(sfilter, "shuffle") == 0 && strcmp(ssplit, "split") == 0) {
514                     category = 7;
515                 } else if (strcmp(compname, BLOSC_LZ4HC_COMPNAME) == 0 &&
    strcmp(sfilter, "bitshuffle") == 0 && strcmp(ssplit, "split") == 0) {
516                     category = 8;
517                 } else if (strcmp(compname, BLOSC_ZLIB_COMPNAME) == 0 &&
    strcmp(sfilter, "nofilter") == 0 && strcmp(ssplit, "split") == 0) {
518                     category = 9;
519                 } else if (strcmp(compname, BLOSC_ZLIB_COMPNAME) == 0 &&
    strcmp(sfilter, "shuffle") == 0 && strcmp(ssplit, "split") == 00) {
520                     category = 10;
521                 } else if (strcmp(compname, BLOSC_ZLIB_COMPNAME) == 0 &&
    strcmp(sfilter, "bitshuffle") == 0 && strcmp(ssplit, "split") == 0) {
522                     category = 11;
523                 } else if (strcmp(compname, BLOSC_ZSTD_COMPNAME) == 0 &&
    strcmp(sfilter, "nofilter") == 0 && strcmp(ssplit, "split") == 0) {
524                     category = 12;
525                 } else if (strcmp(compname, BLOSC_ZSTD_COMPNAME) == 0 &&
    strcmp(sfilter, "shuffle") == 0 && strcmp(ssplit, "split") == 0) {
526                     category = 13;
527                 } else if (strcmp(compname, BLOSC_ZSTD_COMPNAME) == 0 &&
    strcmp(sfilter, "bitshuffle") == 0 && strcmp(ssplit, "split") == 0) {
528                     category = 14;
529 
530                 } else if (strcmp(compname, BLOSC_BLOSCLZ_COMPNAME) == 0 &&
    strcmp(sfilter, "nofilter") == 0 && strcmp(ssplit, "nosplit") == 0) {
531                     category = 15;
532                 } else if (strcmp(compname, BLOSC_BLOSCLZ_COMPNAME) == 0 &&
    strcmp(sfilter, "shuffle") == 0 && strcmp(ssplit, "nosplit") == 0) {
533                     category = 16;
```

```
534                 } else if (strcmp(compname, BLOSC_BLOSCLZ_COMPNAME) == 0 &&
    strcmp(sfilter, "bitshuffle") == 0 && strcmp(ssplit, "nosplit") == 0) {
535                     category = 17;
536                 } else if (strcmp(compname, BLOSC_LZ4_COMPNAME) == 0 &&
    strcmp(sfilter, "nofilter") == 0 && strcmp(ssplit, "nosplit") == 0) {
537                     category = 18;
538                 } else if (strcmp(compname, BLOSC_LZ4_COMPNAME) == 0 &&
    strcmp(sfilter, "shuffle") == 0 && strcmp(ssplit, "nosplit") == 0) {
539                     category = 19;
540                 } else if (strcmp(compname, BLOSC_LZ4_COMPNAME) == 0 &&
    strcmp(sfilter, "bitshuffle") == 0 && strcmp(ssplit, "nosplit") == 0) {
541                     category = 20;
542                 } else if (strcmp(compname, BLOSC_LZ4HC_COMPNAME) == 0 &&
    strcmp(sfilter, "nofilter") == 0 && strcmp(ssplit, "nosplit") == 0) {
543                     category = 21;
544                 } else if (strcmp(compname, BLOSC_LZ4HC_COMPNAME) == 0 &&
    strcmp(sfilter, "shuffle") == 0 && strcmp(ssplit, "nosplit") == 0) {
545                     category = 22;
546                 } else if (strcmp(compname, BLOSC_LZ4HC_COMPNAME) == 0 &&
    strcmp(sfilter, "bitshuffle") == 0 && strcmp(ssplit, "nosplit") == 0) {
547                     category = 23;
548                 } else if (strcmp(compname, BLOSC_ZLIB_COMPNAME) == 0 &&
    strcmp(sfilter, "nofilter") == 0 && strcmp(ssplit, "nosplit") == 0) {
549                     category = 24;
550                 } else if (strcmp(compname, BLOSC_ZLIB_COMPNAME) == 0 &&
    strcmp(sfilter, "shuffle") == 0 && strcmp(ssplit, "nosplit") == 0) {
551                     category = 25;
552                 } else if (strcmp(compname, BLOSC_ZLIB_COMPNAME) == 0 &&
    strcmp(sfilter, "bitshuffle") == 0 && strcmp(ssplit, "nosplit") == 0) {
553                     category = 26;
554                 } else if (strcmp(compname, BLOSC_ZSTD_COMPNAME) == 0 &&
    strcmp(sfilter, "nofilter") == 0 && strcmp(ssplit, "nosplit") == 00) {
555                     category = 27;
556                 } else if (strcmp(compname, BLOSC_ZSTD_COMPNAME) == 0 &&
    strcmp(sfilter, "shuffle") == 0 && strcmp(ssplit, "nosplit") == 0) {
557                     category = 28;
558                 } else if (strcmp(compname, BLOSC_ZSTD_COMPNAME) == 0 &&
    strcmp(sfilter, "bitshuffle") == 0 && strcmp(ssplit, "nosplit") == 0) {
559                     category = 29;
560                 }
561
562                 fprintf(csv_file, "cratio, speed, special_vals, nchunk,
    category\n");
563                 for (int nchunk = 0; nchunk < schunk->nchunks; nchunk++) {
564                     printf("decompressing chunk # %d (out of %d)\n", nchunk,
    schunk->nchunks);
565                     int dsize = blosc2_schunk_decompress_chunk(schunk, nchunk
    , chunk, schunk->chunksize);
566                     if (dsize < 0) {
567                         printf("Error decompressing chunk in schunk.  Error
    code: %d\n", dsize);
568                         return dsize;
```

```
                    }
                    int csize = blosc2_compress_ctx(cctx, chunk, dsize,
  chunk2, schunk->chunksize);
                    if (csize < 0) {
                        printf("Error compressing chunk.  Error code: %d\n",
  csize);
                        return csize;
                    }
                    int dsize2 = blosc2_decompress_ctx(dctx, chunk2, csize,
  chunk, dsize);
                    if (dsize2 < 0) {
                        printf("Error decompressing chunk.  Error code: %d\n
  ", dsize2);
                        return dsize2;
                    }

                    int nstreams = dsize2 / (int) sizeof(blosc2_instr);
                    printf("Chunk %d data with %d streams:\n", nchunk,
  nstreams);

                    int categoria = categorize_data(compname, *sfilter, *
  ssplit);
                    extr_data(csv_file, nchunk, chunk, nstreams, category);
                }

                fclose(csv_file);
            }
        }
    }

} else {
    // Loop over different filters
    for (int nfilter = 0; nfilter <= BLOSC_BITSHUFFLE; ++nfilter) {
        cparams.filters[BLOSC2_MAX_FILTERS - 1] = nfilter;
        blosc2_context *cctx = blosc2_create_cctx(cparams);
        blosc2_context *dctx = blosc2_create_dctx(dparams);

        const char *compname;
        switch (cparams.compcode) {
            case BLOSC_BLOSCLZ:
                compname = BLOSC_BLOSCLZ_COMPNAME;
                break;
            case BLOSC_LZ4:
                compname = BLOSC_LZ4_COMPNAME;
                break;
            case BLOSC_LZ4HC:
                compname = BLOSC_LZ4HC_COMPNAME;
                break;
            case BLOSC_ZLIB:
                compname = BLOSC_ZLIB_COMPNAME;
                break;
            case BLOSC_ZSTD:
```

```
                compname = BLOSC_ZSTD_COMPNAME;
                break;
            case ENTROPY_PROBE_ID:
                compname = "entropy";
                break;
            default:
                printf("Unsupported codec!");
                exit(1);
        }

        char *sfilter;
        switch (nfilter) {
            case BLOSC_NOFILTER:
                sfilter = "nofilter";
                break;
            case BLOSC_SHUFFLE:
                sfilter = "shuffle";
                break;
            case BLOSC_BITSHUFFLE:
                sfilter = "bitshuffle";
                break;
            default:
                printf("Unsupported filter!");
                exit(1);
        }

        // Create csv file
        sprintf(csv_filename, "%s-%s.csv", compname, sfilter);
        FILE *csv_file = fopen(csv_filename, "w");
        if (csv_file == NULL) {
            printf("Error creating the file\n");
            return -1;
        }

        fprintf(csv_file, "cratio, speed, special_vals, nchunk, category\n");
        for (int nchunk = 0; nchunk < schunk->nchunks; nchunk++) {
            printf("decompressing chunk # %d (out of %d)\n", nchunk, schunk->
    nchunks);
            int dsize = blosc2_schunk_decompress_chunk(schunk, nchunk, chunk,
     schunk->chunksize);
            if (dsize < 0) {
                printf("Error decompressing chunk in schunk.  Error code: %d\
    n", dsize);
                return dsize;
            }
            int csize = blosc2_compress_ctx(cctx, chunk, dsize, chunk2,
    schunk->chunksize);
            if (csize < 0) {
                printf("Error compressing chunk.  Error code: %d\n", csize);
                return csize;
            }
```

```
662                 int dsize2 = blosc2_decompress_ctx(dctx, chunk2, csize, chunk,
    dsize);
663                 if (dsize2 < 0) {
664                     printf("Error decompressing chunk.  Error code: %d\n", dsize2
    );
665                     return dsize2;
666                 }
667 
668                 int nstreams = dsize2 / (int) sizeof(blosc2_instr);
669                 printf("Chunk %d data with %d streams:\n", nchunk, nstreams);
670 
671                 int category = -1;
672                 extr_data(csv_file, nchunk, chunk, nstreams, category);
673             }
674 
675             fclose(csv_file);
676 
677         }
678     }
679 
680     /* Free resources */
681     blosc2_schunk_free(schunk);
682     free(chunk);
683     free(chunk2);
684     printf("Success!\n");
685 
686     return 0;
687 }
```

Anexo B

Análisis de datos

Los siguientes códigos de las secciones B.1, B.2 y B.3 estan publicados en mi repositorio de Git Hub [5].

B.1. Código Neural-Network-CRATIO.ipynb

En este anexo se presenta el código realizado en Jupyter Notebook que prepara y manipula los datos, para finalmente construir y evaluar el modelo de red neuronal donde se optimiza el favour de max_cratio.El fichero del código se llama **Neural-Network-CRATIO.ipynb**.

```
#!/usr/bin/env python
# coding: utf-8

# In[128]:


import pandas as pd
import matplotlib.pyplot as plt
import plotly.express as px
import numpy as np


# In[129]:


from pathlib import Path
path = Path('../data_inputs_temp/')

```

```
# In[130]:


ficheros = ['entropy-shuffle', 'zstd-shuffle-split', 'zlib-shuffle-split', '
    lz4hc-shuffle-split']

ficheros_path = [str(path) + '/' + fichero + '.csv.gz' for fichero in ficheros]
dataframes1 = {}
nsplits = [55296] * 4

for i in range(0, len(ficheros)):
    dataframes1[ficheros[i]] = pd.read_csv(ficheros_path[i], delimiter=',')
    codec_filter = np.repeat(ficheros[i], nsplits[i])
    dataframes1[ficheros[i]]['codec_filter'] = codec_filter
    if i == 0:
        categoria = np.repeat(str(30+i), nsplits[i])
    else:
        categoria = np.repeat(str(i-1), nsplits[i])
    dataframes1[ficheros[i]]['categoria'] = categoria

    dataframes1[ficheros[i]].rename(columns = {'cratio':'cratio', ' speed':'
    speed', ' special_vals':'special_vals', ' nchunk':'nchunk', ' category':'
    category'}, inplace = True)


# In[131]:


from pathlib import Path
path = Path('../data_inputs')


# In[132]:


ficheros = ['entropy-shuffle', 'zstd-shuffle-split', 'zlib-shuffle-split', '
    lz4hc-shuffle-split']

ficheros_path = [str(path) + '/' + fichero + '.csv.gz' for fichero in ficheros]
dataframes2 = {}
nsplits = [55296] * 4

for i in range(0, len(ficheros)):
    dataframes2[ficheros[i]] = pd.read_csv(ficheros_path[i], delimiter=',')
    codec_filter = np.repeat(ficheros[i], nsplits[i])
    dataframes2[ficheros[i]]['codec_filter'] = codec_filter
    if i == 0:
        categoria = np.repeat(str(30+i), nsplits[i])
    else:
        categoria = np.repeat(str(i-1), nsplits[i])
    dataframes2[ficheros[i]]['categoria'] = categoria

```

```
    dataframes2[ficheros[i]].rename(columns = {'cratio':'cratio', ' speed':'speed', ' special_vals':'special_vals', ' nchunk':'nchunk', ' category':'category'}, inplace = True)


# In[133]:


from pathlib import Path
path = Path('../data_inputs_wind/')


# In[134]:


ficheros = ['entropy-shuffle', 'zstd-shuffle-split', 'zlib-shuffle-split', 'lz4hc-shuffle-split']

ficheros_path = [str(path) + '/' + fichero + '.csv.gz' for fichero in ficheros]
dataframes3 = {}
nsplits = [55296] * 4

for i in range(0, len(ficheros)):
    dataframes3[ficheros[i]] = pd.read_csv(ficheros_path[i], delimiter=',')
    codec_filter = np.repeat(ficheros[i], nsplits[i])
    dataframes3[ficheros[i]]['codec_filter'] = codec_filter
    if i == 0:
        categoria = np.repeat(str(30+i), nsplits[i])
    else:
        categoria = np.repeat(str(i-1), nsplits[i])
    dataframes3[ficheros[i]]['categoria'] = categoria

    dataframes3[ficheros[i]].rename(columns = {'cratio':'cratio', ' speed':'speed', ' special_vals':'special_vals', ' nchunk':'nchunk', ' category':'category'}, inplace = True)


# In[135]:


df_fig = pd.DataFrame()
for i in range(0, len(ficheros)):
    df_fig = pd.concat([df_fig, dataframes1[ficheros[i]],
                        dataframes2[ficheros[i]], dataframes3[ficheros[i]]],
    axis = 0)


# In[136]:


df_fig = df_fig[df_fig['special_vals'] == 0]

```

```
114
115 # In[137]:
116
117
118 fig = px.scatter(df_fig, "cratio", "speed", color = "categoria")
119 fig.show()
120
121
122 # 0: zstd
123 #
124 # 1: zlib
125 #
126 # 2: lz4hc
127 #
128 # 30: entropy
129
130 # # Dataframe filter: SHUFFLE
131
132 # In[138]:
133
134
135 df_entropy = pd.DataFrame()
136 df_entropy = pd.concat([df_entropy, dataframes1[ficheros[0]], dataframes2[
    ficheros[0]],
137                         dataframes3[ficheros[0]]], axis = 0)
138
139 df_entropy.rename(columns = {'cratio':'cratio31', 'speed':'speed31', '
    special_vals':'special_vals31'}, inplace = True)
140 df_entropy = df_entropy.drop(['nchunk', 'category', 'codec_filter', 'categoria
    '], axis = 1)
141
142
143 # In[139]:
144
145
146 df_zstd = pd.DataFrame()
147 df_zstd = pd.concat([df_zstd, dataframes1[ficheros[1]], dataframes2[ficheros
    [1]],
148                         dataframes3[ficheros[1]]], axis = 0)
149
150 df_zstd.rename(columns = {'cratio':'cratio0', 'speed':'speed0', 'special_vals
    ':'special_vals0'}, inplace = True)
151 df_zstd = df_zstd.drop(['nchunk', 'category', 'codec_filter', 'categoria'],
    axis = 1)
152
153
154 # In[140]:
155
156
157 df_zlib = pd.DataFrame()
158 df_zlib = pd.concat([df_zlib, dataframes1[ficheros[2]], dataframes2[ficheros
    [2]],
```

```
                        dataframes3[ficheros[2]]], axis = 0)

df_zlib.rename(columns = {'cratio':'cratio1', 'speed':'speed1', 'special_vals
    ':'special_vals1'}, inplace = True)
df_zlib = df_zlib.drop(['nchunk', 'category', 'codec_filter', 'categoria'],
    axis = 1)


# In[141]:


df_lz4hc = pd.DataFrame()
df_lz4hc = pd.concat([df_lz4hc, dataframes1[ficheros[3]], dataframes2[ficheros
    [3]],
                        dataframes3[ficheros[3]]], axis = 0)

df_lz4hc.rename(columns = {'cratio':'cratio2', 'speed':'speed2', 'special_vals
    ':'special_vals2'}, inplace = True)
df_lz4hc = df_lz4hc.drop(['nchunk', 'category', 'codec_filter', 'categoria'],
    axis = 1)


# In[142]:


df_shuffle = pd.DataFrame()
df_shuffle = pd.concat([df_shuffle, df_zstd, df_zlib, df_lz4hc, df_entropy],
    axis = 1)


# Eliminar special_value (que lo sea en alguna de las 4 categor as)

# In[143]:


def delete_final(special_vals0, special_vals1, special_vals2, special_vals31):
    if (special_vals0 == 1 or special_vals1 == 1 or special_vals2 == 1 or
    special_vals31 == 1):
        return 1
    else:
        return 0


# In[144]:


df_shuffle["delete_f"] = df_shuffle.apply(lambda x: delete_final(x[2], x[5], x
    [8], x[11]), axis=1)


# In[145]:

```

```
203
204 df_shuffle = df_shuffle[df_shuffle['delete_f'] == 0]
205
206
207 # In[146]:
208
209
210 df_shuffle = df_shuffle.drop(['delete_f'], axis = 1)
211
212
213 # In[147]:
214
215
216 df_input_x = df_shuffle.iloc[:, [9, 10]]
217
218
219 # C lculo del SCORE
220
221 # In[148]:
222
223
224 df_shuffle = df_shuffle.drop(['cratio31', 'speed31', 'special_vals31', '
        special_vals0', 'special_vals1', 'special_vals2'], axis = 1)
225
226
227 # In[149]:
228
229
230 dfcratios = pd.DataFrame()
231 for i in range(0, 3):
232     dfcratios = pd.concat([dfcratios, df_shuffle.iloc[:, 2*i]], axis =0)
233
234
235 # In[150]:
236
237
238 desv_cratio = dfcratios.std()[0]
239 desv_cratio
240
241
242 # In[151]:
243
244
245 mean_cratio = dfcratios.mean()[0]
246 mean_cratio
247
248
249 # In[152]:
250
251
252 dfspeeds = pd.DataFrame()
253 for i in range(0, 3):
```

```
    dfspeeds = pd.concat([dfspeeds, df_shuffle.iloc[:, 2*i+1]], axis =0)


# In[153]:


mean_speed = dfspeeds.mean()[0]
mean_speed


# In[154]:


desv_speed = dfspeeds.std()[0]
desv_speed


# In[155]:


theta = 0.96

cratio0 = df_shuffle["cratio0"]
speed0 = df_shuffle["speed0"]
ncratio0 = (cratio0 - mean_cratio) / desv_cratio
nspeed0 = (speed0 - mean_speed) / desv_speed
df_shuffle.insert(2, 0, theta*ncratio0 + (1-theta)*nspeed0)

cratio1 = df_shuffle["cratio1"]
speed1 = df_shuffle["speed1"]
ncratio1 = (cratio1 - mean_cratio) / desv_cratio
nspeed1 = (speed1 - mean_speed) / desv_speed
df_shuffle.insert(5, 1, theta*ncratio1 + (1-theta)*nspeed1)

cratio2 = df_shuffle["cratio2"]
speed2 = df_shuffle["speed2"]
ncratio2 = (cratio2 - mean_cratio) / desv_cratio
nspeed2 = (speed2 - mean_speed) / desv_speed
df_shuffle.insert(8, 2, theta*ncratio2 + (1-theta)*nspeed2)


# In[156]:


df_pos = df_shuffle
df_pos = df_pos.drop(['cratio0', 'speed0', 'cratio1', 'speed1', 'cratio2', '
    speed2'], axis = 1)


# In[157]:


```

```
best_categ_shuffle = df_pos.idxmax(axis=1)
best_categ_shuffle = pd.DataFrame(best_categ_shuffle, columns=['best_categ'])


# In[158]:


categorias = best_categ_shuffle['best_categ'].unique()


# In[159]:


unique, counts = np.unique(best_categ_shuffle, return_counts=True)
result = np.column_stack((unique, counts))
result


# In[160]:


df_t = pd.DataFrame()
df_t = pd.concat([df_t, df_input_x['cratio31']], axis = 1)
df_t = pd.concat([df_t, df_input_x['speed31']], axis = 1)
df_t = pd.concat([df_t, best_categ_shuffle], axis = 1)


# Balancear

# In[161]:


df_blc = pd.DataFrame()
for i in categorias:
    df_i = df_t[df_t['best_categ'] == i]
    df_i = df_i.head(n=9618)
    df_blc = pd.concat([df_blc, df_i], axis = 0)


# In[162]:


dfinput_y = pd.DataFrame()
dfinput_y = pd.concat([dfinput_y, df_blc['best_categ']], axis = 1)


# In[163]:


dfinput_x = df_blc.drop(['best_categ'], axis = 1)


```

```
# Normalizar inputs

# In[164]:


dfcratios = pd.DataFrame()
dfcratios = pd.concat([dfcratios, dfinput_x.iloc[:, 0]], axis =0)


# In[165]:


desv_cratio = dfcratios.std()[0]
desv_cratio


# In[166]:


mean_cratio = dfcratios.mean()[0]
mean_cratio


# In[167]:


dfspeeds = pd.DataFrame()
dfspeeds = pd.concat([dfspeeds, dfinput_x.iloc[:, 1]], axis =0)


# In[168]:


mean_speed = dfspeeds.mean()[0]
mean_speed


# In[169]:


desv_speed = dfspeeds.std()[0]
desv_speed


# In[170]:


cratio = dfinput_x["cratio31"]
speed = dfinput_x["speed31"]
ncratio = (cratio - mean_cratio) / desv_cratio
nspeed = (speed - mean_speed) / desv_speed

```

```
dfinput_x_norm = pd.DataFrame()
dfinput_x_norm.insert(0, "ncratio", ncratio)
dfinput_x_norm.insert(1, "nspeed", nspeed)


# In[171]:


dfinput_x = dfinput_x_norm


# In[172]:


dfinput_x.shape


# In[173]:


dfinput_y.shape


# # RED NEURONAL (3 categor as)

# In[174]:


from sklearn.model_selection import train_test_split

# Split the data
x_train, x_test, y_train, y_test = train_test_split(dfinput_x, dfinput_y,
    test_size=0.1, shuffle= True)


# In[175]:


x_train.shape


# In[176]:


x_test.shape


# In[177]:


y_test_nocod = y_test.copy()

```

```

# In[178]:


from keras.utils import to_categorical
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)


# In[179]:


from keras import models, layers
import tensorflow

tensorflow.random.set_seed(100)

model = models.Sequential()
model.add(layers.Dense(2, activation='relu', input_shape=(2,)))
model.add(layers.Dense(2, activation='relu'))
model.add(layers.Dense(3, activation='softmax'))


# In[180]:


model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['acc'])


# In[181]:


history = model.fit(x_train, y_train, epochs = 60, validation_data = (x_test,
    y_test))


# In[182]:


_, ax = plt.subplots(1, 2, figsize=(10, 5), dpi=100)
ax[0].plot(history.history['acc'], 'r')
ax[0].plot(history.history['val_acc'], 'g')
ax[0].set_xlabel("Num of Epochs")
ax[0].set_ylabel("Accuracy")
ax[0].set_title("Training Accuracy vs Validation Accuracy")
ax[0].legend(['train', 'validation'])

ax[1].plot(history.history['loss'], 'r')
ax[1].plot(history.history['val_loss'], 'g')
ax[1].set_xlabel("Num of Epochs")
```

```
ax[1].set_ylabel("Loss")
ax[1].set_title("Training Loss vs Validation Loss")
ax[1].legend(['train', 'validation'])

plt.tight_layout()


# In[183]:


y_hat = model.predict(x=x_test)


# In[184]:


y_pred_label = np.argmax(y_hat, axis = 1)
y_true_label = np.argmax(y_test, axis = 1)


# In[185]:


unique, counts = np.unique(y_pred_label, return_counts=True)
result = np.column_stack((unique, counts))
result


# In[186]:


unique, counts = np.unique(y_true_label, return_counts=True)
result = np.column_stack((unique, counts))
result


# In[187]:


tabla = pd.crosstab(y_pred_label, y_true_label)
tabla


# # An lisis resultados

# In[188]:


df_correctos = pd.DataFrame(columns=['cratio', 'speed', 'category'])


# In[189]:
```

```
563
564
565 for i in range (0, len(y_pred_label)):
566     cratio_act = x_test.iloc[i, 0]
567     speed_act = x_test.iloc[i, 1]
568     categ_act = y_test_nocod.iloc[i, 0]
569
570     df_correctos = df_correctos.append({'cratio': cratio_act, 'speed':
        speed_act, 'category': categ_act}, ignore_index=True)
571
572
573
574 # In[190]:
575
576
577 fig = px.scatter(df_correctos, "cratio", "speed", color = "category")
578 fig.show()
579
580
581 # GR FICAS POR CATEGOR A (Y SU ERROR)
582
583 # In[191]:
584
585
586 df_0 = pd.DataFrame(columns=['cratio', 'speed', 'category'])
587
588
589 # In[192]:
590
591
592 for i in range (0, len(y_pred_label)):
593     cratio_act = x_test.iloc[i, 0]
594     speed_act = x_test.iloc[i, 1]
595     categ_act = y_test_nocod.iloc[i, 0]
596     if y_pred_label[i] == y_true_label[i] and y_pred_label[i] == 0.0:
597         df_0 = df_0.append({'cratio': cratio_act, 'speed': speed_act, 'category
        ': 0}, ignore_index=True)
598
599 for i in range (0, len(y_pred_label)):
600     cratio_act = x_test.iloc[i, 0]
601     speed_act = x_test.iloc[i, 1]
602     if y_pred_label[i] != y_true_label[i] and y_true_label[i] == 0.0:
603         df_0 = df_0.append({'cratio': cratio_act, 'speed': speed_act, 'category
        ': -1}, ignore_index=True)
604
605
606 # In[193]:
607
608
609 fig = px.scatter(df_0, "cratio", "speed", color = "category")
610 fig.show()
611
```

```
612
613 # In[194]:
614
615
616 df_1 = pd.DataFrame(columns=['cratio', 'speed', 'category'])
617
618
619 # In[195]:
620
621
622 for i in range (0, len(y_pred_label)):
623     cratio_act = x_test.iloc[i, 0]
624     speed_act = x_test.iloc[i, 1]
625     categ_act = y_test_nocod.iloc[i, 0]
626     if y_pred_label[i] == y_true_label[i] and y_pred_label[i] == 1.0:
627         df_1 = df_1.append({'cratio': cratio_act, 'speed': speed_act, 'category
    ': 1}, ignore_index=True)
628
629 for i in range (0, len(y_pred_label)):
630     cratio_act = x_test.iloc[i, 0]
631     speed_act = x_test.iloc[i, 1]
632     if y_pred_label[i] != y_true_label[i] and y_true_label[i] == 1.0:
633         df_1 = df_1.append({'cratio': cratio_act, 'speed': speed_act, 'category
    ': -2}, ignore_index=True)
634
635
636 # In[196]:
637
638
639 fig = px.scatter(df_1, "cratio", "speed", color = "category")
640 fig.show()
641
642
643 # In[197]:
644
645
646 df_2 = pd.DataFrame(columns=['cratio', 'speed', 'category'])
647
648
649 # In[198]:
650
651
652 for i in range (0, len(y_pred_label)):
653     cratio_act = x_test.iloc[i, 0]
654     speed_act = x_test.iloc[i, 1]
655     categ_act = y_test_nocod.iloc[i, 0]
656     if y_pred_label[i] == y_true_label[i] and y_pred_label[i] == 2.0:
657         df_2 = df_2.append({'cratio': cratio_act, 'speed': speed_act, 'category
    ': 2}, ignore_index=True)
658
659 for i in range (0, len(y_pred_label)):
660     cratio_act = x_test.iloc[i, 0]
```

```
    speed_act = x_test.iloc[i, 1]
    if y_pred_label[i] != y_true_label[i] and y_true_label[i] == 2.0:
        df_2 = df_2.append({'cratio': cratio_act, 'speed': speed_act, 'category
    ': -3}, ignore_index=True)


# In[199]:


fig = px.scatter(df_2, "cratio", "speed", color = "category")
fig.show()
```

B.2. Código Neural-Network-SPEED.ipynb

En este anexo se presenta el código realizado en Jupyter Notebook que prepara y manipula los datos, para finalmente construir y evaluar el modelo de red neuronal donde se optimiza el favour de max_speed.El fichero del código se llama **Neural-Network-SPEED.ipynb**.

```
#!/usr/bin/env python
# coding: utf-8

# In[120]:


import pandas as pd
import matplotlib.pyplot as plt
import plotly.express as px
import numpy as np


# In[121]:


from pathlib import Path
path = Path('../data_inputs_temp/')


# In[122]:


ficheros = ['entropy-shuffle', 'blosclz-shuffle-split', 'lz4-shuffle-split']

ficheros_path = [str(path) + '/' + fichero + '.csv.gz' for fichero in ficheros]
dataframes1 = {}
nsplits = [55296] * 3

for i in range(0, len(ficheros)):
    dataframes1[ficheros[i]] = pd.read_csv(ficheros_path[i], delimiter=',')
```

```
    codec_filter = np.repeat(ficheros[i], nsplits[i])
    dataframes1[ficheros[i]]['codec_filter'] = codec_filter
    if i == 0:
        categoria = np.repeat(str(30+i), nsplits[i])
    else:
        categoria = np.repeat(str(i-1), nsplits[i])
    dataframes1[ficheros[i]]['categoria'] = categoria

    dataframes1[ficheros[i]].rename(columns = {'cratio':'cratio', ' speed':'
    speed', ' special_vals':'special_vals', ' nchunk':'nchunk', ' category':'
    category'}, inplace = True)


# In[123]:


from pathlib import Path
path = Path('../data_inputs')


# In[124]:


ficheros = ['entropy-shuffle', 'blosclz-shuffle-split', 'lz4-shuffle-split']

ficheros_path = [str(path) + '/' + fichero + '.csv.gz' for fichero in ficheros]
dataframes2 = {}
nsplits = [55296] * 3

for i in range(0, len(ficheros)):
    dataframes2[ficheros[i]] = pd.read_csv(ficheros_path[i], delimiter=',')
    codec_filter = np.repeat(ficheros[i], nsplits[i])
    dataframes2[ficheros[i]]['codec_filter'] = codec_filter
    if i == 0:
        categoria = np.repeat(str(30+i), nsplits[i])
    else:
        categoria = np.repeat(str(i-1), nsplits[i])
    dataframes2[ficheros[i]]['categoria'] = categoria

    dataframes2[ficheros[i]].rename(columns = {'cratio':'cratio', ' speed':'
    speed', ' special_vals':'special_vals', ' nchunk':'nchunk', ' category':'
    category'}, inplace = True)


# In[125]:


from pathlib import Path
path = Path('../data_inputs_wind/')


# In[126]:
```

```
79
80
81 ficheros = ['entropy-shuffle', 'blosclz-shuffle-split', 'lz4-shuffle-split']
82
83 ficheros_path = [str(path) + '/' + fichero + '.csv.gz' for fichero in ficheros]
84 dataframes3 = {}
85 nsplits = [55296] * 3
86
87 for i in range(0, len(ficheros)):
88     dataframes3[ficheros[i]] = pd.read_csv(ficheros_path[i], delimiter=',')
89     codec_filter = np.repeat(ficheros[i], nsplits[i])
90     dataframes3[ficheros[i]]['codec_filter'] = codec_filter
91     if i == 0:
92         categoria = np.repeat(str(30+i), nsplits[i])
93     else:
94         categoria = np.repeat(str(i-1), nsplits[i])
95     dataframes3[ficheros[i]]['categoria'] = categoria
96
97     dataframes3[ficheros[i]].rename(columns = {'cratio':'cratio', ' speed':'
    speed', ' special_vals':'special_vals', ' nchunk':'nchunk', ' category':'
    category'}, inplace = True)
98
99
100 # In[127]:
101
102
103 df_fig = pd.DataFrame()
104 for i in range(0, len(ficheros)):
105     df_fig = pd.concat([df_fig, dataframes1[ficheros[i]],
106                         dataframes2[ficheros[i]], dataframes3[ficheros[i]]],
    axis = 0)
107
108
109 # In[128]:
110
111
112 df_fig = df_fig[df_fig['special_vals'] == 0]
113
114
115 # In[129]:
116
117
118 fig = px.scatter(df_fig, "cratio", "speed", color = "categoria")
119 fig.show()
120
121
122 # 0: blosclz
123 #
124 # 1: lz4
125 #
126 # 30: entropy
127
```

```
# # Dataframe filter: SHUFFLE

# In[130]:


df_entropy = pd.DataFrame()
df_entropy = pd.concat([df_entropy, dataframes1[ficheros[0]], dataframes2[
    ficheros[0]],
                        dataframes3[ficheros[0]]], axis = 0)

df_entropy.rename(columns = {'cratio':'cratio31', 'speed':'speed31', '
    special_vals':'special_vals31'}, inplace = True)
df_entropy = df_entropy.drop(['nchunk', 'category', 'codec_filter', 'categoria
    '], axis = 1)


# In[131]:


df_blosclz = pd.DataFrame()
df_blosclz = pd.concat([df_blosclz, dataframes1[ficheros[1]], dataframes2[
    ficheros[1]],
                        dataframes3[ficheros[1]]], axis = 0)

df_blosclz.rename(columns = {'cratio':'cratio0', 'speed':'speed0', '
    special_vals':'special_vals0'}, inplace = True)
df_blosclz = df_blosclz.drop(['nchunk', 'category', 'codec_filter', 'categoria
    '], axis = 1)


# In[132]:


df_lz4 = pd.DataFrame()
df_lz4 = pd.concat([df_lz4, dataframes1[ficheros[2]], dataframes2[ficheros[2]],
                         dataframes3[ficheros[2]]], axis = 0)

df_lz4.rename(columns = {'cratio':'cratio1', 'speed':'speed1', 'special_vals':'
    special_vals1'}, inplace = True)
df_lz4 = df_lz4.drop(['nchunk', 'category', 'codec_filter', 'categoria'], axis
    = 1)


# In[133]:


df_shuffle = pd.DataFrame()
df_shuffle = pd.concat([df_shuffle, df_blosclz, df_lz4, df_entropy], axis = 1)


# Eliminar special_value (que lo sea en alguna de las 3 categor as)

```

```
# In[134]:


def delete_final(special_vals0, special_vals1, special_vals31):
    if (special_vals0 == 1 or special_vals1 == 1 or special_vals31 == 1):
        return 1
    else:
        return 0


# In[135]:


df_shuffle["delete_f"] = df_shuffle.apply(lambda x: delete_final(x[2], x[5], x
    [8]), axis=1)


# In[136]:


df_shuffle = df_shuffle[df_shuffle['delete_f'] == 0]


# In[137]:


df_shuffle = df_shuffle.drop(['delete_f'], axis = 1)


# In[138]:


df_input_x = df_shuffle.iloc[:, [6, 7]]


# C lculo del SCORE

# In[139]:


df_shuffle = df_shuffle.drop(['cratio31', 'speed31', 'special_vals31', '
    special_vals0', 'special_vals1'], axis = 1)


# In[140]:


dfcratios = pd.DataFrame()
for i in range(0, 2):
    dfcratios = pd.concat([dfcratios, df_shuffle.iloc[:, 2*i]], axis =0)


```

```
# In[141]:


desv_cratio = dfcratios.std()[0]
desv_cratio


# In[142]:


mean_cratio = dfcratios.mean()[0]
mean_cratio


# In[143]:


dfspeeds = pd.DataFrame()
for i in range(0, 2):
    dfspeeds = pd.concat([dfspeeds, df_shuffle.iloc[:, 2*i+1]], axis =0)


# In[144]:


mean_speed = dfspeeds.mean()[0]
mean_speed


# In[145]:


desv_speed = dfspeeds.std()[0]
desv_speed


# In[146]:


theta = 0.1

cratio0 = df_shuffle["cratio0"]
speed0 = df_shuffle["speed0"]
ncratio0 = (cratio0 - mean_cratio) / desv_cratio
nspeed0 = (speed0 - mean_speed) / desv_speed
df_shuffle.insert(2, 0, theta*ncratio0 + (1-theta)*nspeed0)

cratio1 = df_shuffle["cratio1"]
speed1 = df_shuffle["speed1"]
ncratio1 = (cratio1 - mean_cratio) / desv_cratio
nspeed1 = (speed1 - mean_speed) / desv_speed
df_shuffle.insert(5, 1, theta*ncratio1 + (1-theta)*nspeed1)
```

```
274
275
276 # In[147]:
277
278
279 df_pos = df_shuffle
280 df_pos = df_pos.drop(['cratio0', 'speed0', 'cratio1', 'speed1'], axis = 1)
281
282
283 # In[148]:
284
285
286 best_categ_shuffle = df_pos.idxmax(axis=1)
287 best_categ_shuffle = pd.DataFrame(best_categ_shuffle, columns=['best_categ'])
288
289
290 # In[149]:
291
292
293 categorias = best_categ_shuffle['best_categ'].unique()
294
295
296 # In[150]:
297
298
299 unique, counts = np.unique(best_categ_shuffle, return_counts=True)
300 result = np.column_stack((unique, counts))
301 result
302
303
304 # In[151]:
305
306
307 df_t = pd.DataFrame()
308 df_t = pd.concat([df_t, df_input_x['cratio31']], axis = 1)
309 df_t = pd.concat([df_t, df_input_x['speed31']], axis = 1)
310 df_t = pd.concat([df_t, best_categ_shuffle], axis = 1)
311
312
313 # Balancear
314
315 # In[152]:
316
317
318 df_blc = pd.DataFrame()
319 for i in categorias:
320     df_i = df_t[df_t['best_categ'] == i]
321     df_i = df_i.head(n=14154)
322     df_blc = pd.concat([df_blc, df_i], axis = 0)
323
324
325 # In[153]:
```

```
326
327
328 dfinput_y = pd.DataFrame()
329 dfinput_y = pd.concat([dfinput_y, df_blc['best_categ']], axis = 1)
330 dfinput_y.shape
331
332
333 # In[154]:
334
335
336 dfinput_x = df_blc.drop(['best_categ'], axis = 1)
337 dfinput_x.shape
338
339
340 # Normalizar inputs
341
342 # In[155]:
343
344
345 dfcratios = pd.DataFrame()
346 dfcratios = pd.concat([dfcratios, dfinput_x.iloc[:, 0]], axis =0)
347
348
349 # In[156]:
350
351
352 desv_cratio = dfcratios.std()[0]
353 desv_cratio
354
355
356 # In[157]:
357
358
359 mean_cratio = dfcratios.mean()[0]
360 mean_cratio
361
362
363 # In[158]:
364
365
366 dfspeeds = pd.DataFrame()
367 dfspeeds = pd.concat([dfspeeds, dfinput_x.iloc[:, 1]], axis =0)
368
369
370 # In[159]:
371
372
373 mean_speed = dfspeeds.mean()[0]
374 mean_speed
375
376
377 # In[160]:
```

```
378
379
380 desv_speed = dfspeeds.std()[0]
381 desv_speed
382
383
384 # In[161]:
385
386
387 cratio = dfinput_x["cratio31"]
388 speed = dfinput_x["speed31"]
389 ncratio = (cratio - mean_cratio) / desv_cratio
390 nspeed = (speed - mean_speed) / desv_speed
391
392 dfinput_x_norm = pd.DataFrame()
393 dfinput_x_norm.insert(0, "ncratio", ncratio)
394 dfinput_x_norm.insert(1, "nspeed", nspeed)
395
396
397 # In[162]:
398
399
400 dfinput_x = dfinput_x_norm
401
402
403 # In[163]:
404
405
406 dfinput_x.shape
407
408
409 # In[164]:
410
411
412 dfinput_y.shape
413
414
415 # # RED NEURONAL (2 categor as)
416
417 # In[165]:
418
419
420 from sklearn.model_selection import train_test_split
421
422 # Split the data
423 x_train, x_test, y_train, y_test = train_test_split(dfinput_x, dfinput_y,
        test_size=0.1, shuffle= True)
424
425
426 # In[166]:
427
428
```

```
x_train.shape


# In[167]:


x_test.shape


# In[168]:


y_test_nocod = y_test.copy()


# In[169]:


from keras.utils import to_categorical
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)


# In[170]:


from keras import models, layers
import tensorflow

tensorflow.random.set_seed(100)

model = models.Sequential()
model.add(layers.Dense(2, activation='relu', input_shape=(2,)))
model.add(layers.Dense(2, activation='relu'))
model.add(layers.Dense(2, activation='softmax'))


# In[171]:


model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['acc'])


# In[172]:


history = model.fit(x_train, y_train, epochs = 40, validation_data = (x_test,
    y_test))


```

```
# In[173]:


_, ax = plt.subplots(1, 2, figsize=(10, 5), dpi=100)
ax[0].plot(history.history['acc'], 'r')
ax[0].plot(history.history['val_acc'], 'g')
ax[0].set_xlabel("Num of Epochs")
ax[0].set_ylabel("Accuracy")
ax[0].set_title("Training Accuracy vs Validation Accuracy")
ax[0].legend(['train', 'validation'])

ax[1].plot(history.history['loss'], 'r')
ax[1].plot(history.history['val_loss'], 'g')
ax[1].set_xlabel("Num of Epochs")
ax[1].set_ylabel("Loss")
ax[1].set_title("Training Loss vs Validation Loss")
ax[1].legend(['train', 'validation'])

plt.tight_layout()


# In[174]:


y_hat = model.predict(x=x_test)


# In[175]:


y_pred_label = np.argmax(y_hat, axis = 1)
y_true_label = np.argmax(y_test, axis = 1)


# In[176]:


unique, counts = np.unique(y_pred_label, return_counts=True)
result = np.column_stack((unique, counts))
result


# In[177]:


unique, counts = np.unique(y_true_label, return_counts=True)
result = np.column_stack((unique, counts))
result


# In[178]:

```

```
532
533 tabla = pd.crosstab(y_pred_label, y_true_label)
534 tabla
535
536
537 # # An lisis resultados
538
539 # In[179]:
540
541
542 df_correctos = pd.DataFrame(columns=['cratio', 'speed', 'category'])
543
544
545 # In[180]:
546
547
548 for i in range (0, len(y_pred_label)):
549     cratio_act = x_test.iloc[i, 0]
550     speed_act = x_test.iloc[i, 1]
551     categ_act = y_test_nocod.iloc[i, 0]
552
553     df_correctos = df_correctos.append({'cratio': cratio_act, 'speed':
        speed_act, 'category': categ_act}, ignore_index=True)
554
555
556
557 # In[181]:
558
559
560 fig = px.scatter(df_correctos, "cratio", "speed", color = "category")
561 fig.show()
562
563
564 # GR FICAS POR CATEGOR A (Y SU ERROR)
565
566 # In[182]:
567
568
569 df_0 = pd.DataFrame(columns=['cratio', 'speed', 'category'])
570
571
572 # In[183]:
573
574
575 for i in range (0, len(y_pred_label)):
576     cratio_act = x_test.iloc[i, 0]
577     speed_act = x_test.iloc[i, 1]
578     categ_act = y_test_nocod.iloc[i, 0]
579     if y_pred_label[i] == y_true_label[i] and y_pred_label[i] == 0.0:
580         df_0 = df_0.append({'cratio': cratio_act, 'speed': speed_act, 'category
        ': 0}, ignore_index=True)
581
```

```
for i in range (0, len(y_pred_label)):
    cratio_act = x_test.iloc[i, 0]
    speed_act = x_test.iloc[i, 1]
    if y_pred_label[i] != y_true_label[i] and y_true_label[i] == 0.0:
        df_0 = df_0.append({'cratio': cratio_act, 'speed': speed_act, 'category': -1}, ignore_index=True)


# In[184]:


fig = px.scatter(df_0, "cratio", "speed", color = "category")
fig.show()


# In[185]:


df_1 = pd.DataFrame(columns=['cratio', 'speed', 'category'])


# In[186]:


for i in range (0, len(y_pred_label)):
    cratio_act = x_test.iloc[i, 0]
    speed_act = x_test.iloc[i, 1]
    categ_act = y_test_nocod.iloc[i, 0]
    if y_pred_label[i] == y_true_label[i] and y_pred_label[i] == 1.0:
        df_1 = df_1.append({'cratio': cratio_act, 'speed': speed_act, 'category': 1}, ignore_index=True)

for i in range (0, len(y_pred_label)):
    cratio_act = x_test.iloc[i, 0]
    speed_act = x_test.iloc[i, 1]
    if y_pred_label[i] != y_true_label[i] and y_true_label[i] == 1.0:
        df_1 = df_1.append({'cratio': cratio_act, 'speed': speed_act, 'category': -2}, ignore_index=True)


# In[187]:


fig = px.scatter(df_1, "cratio", "speed", color = "category")
fig.show()
```

B.3. Código Neural-Network-BALANCE.ipynb

En este anexo se presenta el código realizado en Jupyter Notebook que prepara y manipula los datos, para finalmente construir y evaluar el modelo de red neuronal donde se optimiza el favour de balance.El fichero del código se llama **Neural-Network-BALANCE.ipynb**.

```
#!/usr/bin/env python
# coding: utf-8

# In[241]:


import pandas as pd
import matplotlib.pyplot as plt
import plotly.express as px
import numpy as np


# In[242]:


from pathlib import Path
path = Path('../data_inputs_temp/')


# In[243]:


ficheros = ['entropy-shuffle', 'blosclz-shuffle-split', 'zstd-shuffle-nosplit']

ficheros_path = [str(path) + '/' + fichero + '.csv.gz' for fichero in ficheros]
dataframes1 = {}
nsplits = [55296] * 2 + [13824]

for i in range(0, len(ficheros)):
    dataframes1[ficheros[i]] = pd.read_csv(ficheros_path[i], delimiter=',')
    codec_filter = np.repeat(ficheros[i], nsplits[i])
    dataframes1[ficheros[i]]['codec_filter'] = codec_filter
    if i == 0:
        categoria = np.repeat(str(30+i), nsplits[i])
    else:
        categoria = np.repeat(str(i-1), nsplits[i])
    dataframes1[ficheros[i]]['categoria'] = categoria

    dataframes1[ficheros[i]].rename(columns = {'cratio':'cratio', ' speed':'
    speed', ' special_vals':'special_vals', ' nchunk':'nchunk', ' category':'
    category'}, inplace = True)


# In[244]:

```

```

from pathlib import Path
path = Path('../data_inputs')


# In[245]:


ficheros = ['entropy-shuffle', 'blosclz-shuffle-split', 'zstd-shuffle-nosplit']

ficheros_path = [str(path) + '/' + fichero + '.csv.gz' for fichero in ficheros]
dataframes2 = {}
nsplits = [55296] * 2 + [13824]

for i in range(0, len(ficheros)):
    dataframes2[ficheros[i]] = pd.read_csv(ficheros_path[i], delimiter=',')
    codec_filter = np.repeat(ficheros[i], nsplits[i])
    dataframes2[ficheros[i]]['codec_filter'] = codec_filter
    if i == 0:
        categoria = np.repeat(str(30+i), nsplits[i])
    else:
        categoria = np.repeat(str(i-1), nsplits[i])
    dataframes2[ficheros[i]]['categoria'] = categoria

    dataframes2[ficheros[i]].rename(columns = {'cratio':'cratio', ' speed':'
    speed', ' special_vals':'special_vals', ' nchunk':'nchunk', ' category':'
    category'}, inplace = True)


# In[246]:


from pathlib import Path
path = Path('../data_inputs_wind/')


# In[247]:


ficheros = ['entropy-shuffle', 'blosclz-shuffle-split', 'zstd-shuffle-nosplit']

ficheros_path = [str(path) + '/' + fichero + '.csv.gz' for fichero in ficheros]
dataframes3 = {}
nsplits = [55296] * 2 + [13824]

for i in range(0, len(ficheros)):
    dataframes3[ficheros[i]] = pd.read_csv(ficheros_path[i], delimiter=',')
    codec_filter = np.repeat(ficheros[i], nsplits[i])
    dataframes3[ficheros[i]]['codec_filter'] = codec_filter
    if i == 0:
        categoria = np.repeat(str(30+i), nsplits[i])
    else:
```

```
        categoria = np.repeat(str(i-1), nsplits[i])
    dataframes3[ficheros[i]]['categoria'] = categoria

    dataframes3[ficheros[i]].rename(columns = {'cratio':'cratio', ' speed':'
    speed', ' special_vals':'special_vals', ' nchunk':'nchunk', ' category':'
    category'}, inplace = True)


# In[248]:


55296/13824


# In[249]:


dataframes1[ficheros[2]] = dataframes1[ficheros[2]].loc[np.repeat(dataframes1[
    ficheros[2]].index.values, 4)]
dataframes1[ficheros[2]].reset_index(drop=True)

dataframes2[ficheros[2]] = dataframes2[ficheros[2]].loc[np.repeat(dataframes2[
    ficheros[2]].index.values, 4)]
dataframes2[ficheros[2]].reset_index(drop=True)

dataframes3[ficheros[2]] = dataframes3[ficheros[2]].loc[np.repeat(dataframes3[
    ficheros[2]].index.values, 4)]
dataframes3[ficheros[2]].reset_index(drop=True)


# In[250]:


df_fig = pd.DataFrame()
for i in range(0, len(ficheros)):
    df_fig = pd.concat([df_fig, dataframes1[ficheros[i]],
                        dataframes2[ficheros[i]], dataframes3[ficheros[i]]],
    axis = 0)


# In[251]:


df_fig = df_fig[df_fig['special_vals'] == 0]


# In[252]:


fig = px.scatter(df_fig, "cratio", "speed", color = "categoria")
fig.show()

```

```
140
141 # 0: blosclz-split
142 #
143 # 1: zstd-nosplit
144 #
145 # 30: entropy
146
147 # # Dataframe filter: SHUFFLE
148
149 # In[253]:
150
151
152 df_entropy = pd.DataFrame()
153 df_entropy = pd.concat([df_entropy, dataframes1[ficheros[0]], dataframes2[
    ficheros[0]],
154                         dataframes3[ficheros[0]]], axis = 0)
155
156 df_entropy.rename(columns = {'cratio':'cratio31', 'speed':'speed31', '
    special_vals':'special_vals31'}, inplace = True)
157 df_entropy = df_entropy.drop(['nchunk', 'category', 'codec_filter', 'categoria
    '], axis = 1)
158
159 df_entropy = df_entropy.reset_index(drop=True)
160
161
162 # In[254]:
163
164
165 df_blosclz = pd.DataFrame()
166 df_blosclz = pd.concat([df_blosclz, dataframes1[ficheros[1]], dataframes2[
    ficheros[1]],
167                         dataframes3[ficheros[1]]], axis = 0)
168
169 df_blosclz.rename(columns = {'cratio':'cratio0', 'speed':'speed0', '
    special_vals':'special_vals0'}, inplace = True)
170 df_blosclz = df_blosclz.drop(['nchunk', 'category', 'codec_filter', 'categoria
    '], axis = 1)
171
172 df_blosclz = df_blosclz.reset_index(drop=True)
173
174
175 # In[255]:
176
177
178 df_zstd = pd.DataFrame()
179 df_zstd = pd.concat([df_zstd, dataframes1[ficheros[2]], dataframes2[ficheros
    [2]],
180                          dataframes3[ficheros[2]]], axis = 0)
181
182 df_zstd.rename(columns = {'cratio':'cratio1', 'speed':'speed1', 'special_vals
    ':'special_vals1'}, inplace = True)
```

```
df_zstd = df_zstd.drop(['nchunk', 'category', 'codec_filter', 'categoria'],
    axis = 1)

df_zstd = df_zstd.reset_index(drop=True)


# In[256]:


df_shuffle = pd.DataFrame()
df_shuffle = pd.concat([df_shuffle, df_blosclz, df_zstd, df_entropy], axis = 1)


# Eliminar special_value (que lo sea en alguna de las 3 categor as)

# In[257]:


def delete_final(special_vals0, special_vals1, special_vals31):
    if (special_vals0 == 1 or special_vals1 == 1 or special_vals31 == 1):
        return 1
    else:
        return 0


# In[258]:


df_shuffle["delete_f"] = df_shuffle.apply(lambda x: delete_final(x[2], x[5], x
    [8]), axis=1)


# In[259]:


df_shuffle = df_shuffle[df_shuffle['delete_f'] == 0]


# In[260]:


df_shuffle = df_shuffle.drop(['delete_f'], axis = 1)


# In[261]:


df_input_x = df_shuffle.iloc[:, [6, 7]]


# C lculo del SCORE

```

```
# In[262]:


df_shuffle = df_shuffle.drop(['cratio31', 'speed31', 'special_vals31', '
    special_vals0', 'special_vals1'], axis = 1)


# In[263]:


dfcratios = pd.DataFrame()
for i in range(0, 2):
    dfcratios = pd.concat([dfcratios, df_shuffle.iloc[:, 2*i]], axis =0)


# In[264]:


desv_cratio = dfcratios.std()[0]
desv_cratio


# In[265]:


mean_cratio = dfcratios.mean()[0]
mean_cratio


# In[266]:


dfspeeds = pd.DataFrame()
for i in range(0, 2):
    dfspeeds = pd.concat([dfspeeds, df_shuffle.iloc[:, 2*i+1]], axis =0)


# In[267]:


mean_speed = dfspeeds.mean()[0]
mean_speed


# In[268]:


desv_speed = dfspeeds.std()[0]
desv_speed


# In[269]:
```

```
284
285
286 theta = 0.5
287
288 cratio0 = df_shuffle["cratio0"]
289 speed0 = df_shuffle["speed0"]
290 ncratio0 = (cratio0 - mean_cratio) / desv_cratio
291 nspeed0 = (speed0 - mean_speed) / desv_speed
292 df_shuffle.insert(2, 0, theta*ncratio0 + (1-theta)*nspeed0)
293
294 cratio1 = df_shuffle["cratio1"]
295 speed1 = df_shuffle["speed1"]
296 ncratio1 = (cratio1 - mean_cratio) / desv_cratio
297 nspeed1 = (speed1 - mean_speed) / desv_speed
298 df_shuffle.insert(5, 1, theta*ncratio1 + (1-theta)*nspeed1)
299
300
301 # In[270]:
302
303
304 df_pos = df_shuffle
305 df_pos = df_pos.drop(['cratio0', 'speed0', 'cratio1', 'speed1'], axis = 1)
306
307
308 # In[271]:
309
310
311 best_categ_shuffle = df_pos.idxmax(axis=1)
312 best_categ_shuffle = pd.DataFrame(best_categ_shuffle, columns=['best_categ'])
313
314
315 # In[272]:
316
317
318 categorias = best_categ_shuffle['best_categ'].unique()
319
320
321 # In[273]:
322
323
324 unique, counts = np.unique(best_categ_shuffle, return_counts=True)
325 result = np.column_stack((unique, counts))
326 result
327
328
329 # In[274]:
330
331
332 df_t = pd.DataFrame()
333 df_t = pd.concat([df_t, df_input_x['cratio31']], axis = 1)
334 df_t = pd.concat([df_t, df_input_x['speed31']], axis = 1)
335 df_t = pd.concat([df_t, best_categ_shuffle], axis = 1)
```

```
336
337
338 # Balancear
339
340 # In[275]:
341
342
343 df_blc = pd.DataFrame()
344 for i in categorias:
345     df_i = df_t[df_t['best_categ'] == i]
346     df_i = df_i.head(n=990)
347     df_blc = pd.concat([df_blc, df_i], axis = 0)
348
349
350 # In[276]:
351
352
353 dfinput_y = pd.DataFrame()
354 dfinput_y = pd.concat([dfinput_y, df_blc['best_categ']], axis = 1)
355 dfinput_y.shape
356
357
358 # In[277]:
359
360
361 dfinput_x = df_blc.drop(['best_categ'], axis = 1)
362 dfinput_x.shape
363
364
365 # Normalizar inputs
366
367 # In[278]:
368
369
370 dfcratios = pd.DataFrame()
371 dfcratios = pd.concat([dfcratios, dfinput_x.iloc[:, 0]], axis =0)
372
373
374 # In[279]:
375
376
377 desv_cratio = dfcratios.std()[0]
378 desv_cratio
379
380
381 # In[280]:
382
383
384 mean_cratio = dfcratios.mean()[0]
385 mean_cratio
386
387
```

```
# In[281]:


dfspeeds = pd.DataFrame()
dfspeeds = pd.concat([dfspeeds, dfinput_x.iloc[:, 1]], axis =0)


# In[282]:


mean_speed = dfspeeds.mean()[0]
mean_speed


# In[283]:


desv_speed = dfspeeds.std()[0]
desv_speed


# In[284]:


cratio = dfinput_x["cratio31"]
speed = dfinput_x["speed31"]
ncratio = (cratio - mean_cratio) / desv_cratio
nspeed = (speed - mean_speed) / desv_speed

dfinput_x_norm = pd.DataFrame()
dfinput_x_norm.insert(0, "ncratio", ncratio)
dfinput_x_norm.insert(1, "nspeed", nspeed)


# In[285]:


dfinput_x = dfinput_x_norm


# In[286]:


dfinput_x.shape


# In[287]:


dfinput_y.shape


```

```
# # RED NEURONAL (2 categor as)

# In[288]:


from sklearn.model_selection import train_test_split

# Split the data
x_train, x_test, y_train, y_test = train_test_split(dfinput_x, dfinput_y,
    test_size=0.1, shuffle= True)


# In[289]:


x_train.shape


# In[290]:


x_test.shape


# In[291]:


y_test_nocod = y_test.copy()


# In[292]:


from keras.utils import to_categorical
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)


# In[293]:


from keras import models, layers
import tensorflow

tensorflow.random.set_seed(100)

model = models.Sequential()
model.add(layers.Dense(2, activation='relu', input_shape=(2,)))
model.add(layers.Dense(2, activation='relu'))
model.add(layers.Dense(2, activation='softmax'))


```

```
# In[294]:


model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['acc'])


# In[295]:


history = model.fit(x_train, y_train, epochs = 50, validation_data = (x_test,
    y_test))


# In[296]:


_, ax = plt.subplots(1, 2, figsize=(10, 5), dpi=100)
ax[0].plot(history.history['acc'], 'r')
ax[0].plot(history.history['val_acc'], 'g')
ax[0].set_xlabel("Num of Epochs")
ax[0].set_ylabel("Accuracy")
ax[0].set_title("Training Accuracy vs Validation Accuracy")
ax[0].legend(['train', 'validation'])

ax[1].plot(history.history['loss'], 'r')
ax[1].plot(history.history['val_loss'], 'g')
ax[1].set_xlabel("Num of Epochs")
ax[1].set_ylabel("Loss")
ax[1].set_title("Training Loss vs Validation Loss")
ax[1].legend(['train', 'validation'])

plt.tight_layout()


# In[297]:


y_hat = model.predict(x=x_test)


# In[298]:


y_pred_label = np.argmax(y_hat, axis = 1)
y_true_label = np.argmax(y_test, axis = 1)


# In[299]:


```

```
unique, counts = np.unique(y_pred_label, return_counts=True)
result = np.column_stack((unique, counts))
result


# In[300]:


unique, counts = np.unique(y_true_label, return_counts=True)
result = np.column_stack((unique, counts))
result


# In[301]:


tabla = pd.crosstab(y_pred_label, y_true_label)
tabla


# # An lisis resultados

# In[302]:


df_correctos = pd.DataFrame(columns=['cratio', 'speed', 'category'])


# In[303]:


for i in range (0, len(y_pred_label)):
    cratio_act = x_test.iloc[i, 0]
    speed_act = x_test.iloc[i, 1]
    categ_act = y_test_nocod.iloc[i, 0]

    df_correctos = df_correctos.append({'cratio': cratio_act, 'speed':
    speed_act, 'category': categ_act}, ignore_index=True)



# In[304]:


fig = px.scatter(df_correctos, "cratio", "speed", color = "category")
fig.show()


# GR FICAS POR CATEGOR A (Y SU ERROR)

# In[305]:

```

```
593
594 df_0 = pd.DataFrame(columns=['cratio', 'speed', 'category'])
595
596
597 # In[306]:
598
599
600 for i in range (0, len(y_pred_label)):
601     cratio_act = x_test.iloc[i, 0]
602     speed_act = x_test.iloc[i, 1]
603     categ_act = y_test_nocod.iloc[i, 0]
604     if y_pred_label[i] == y_true_label[i] and y_pred_label[i] == 0.0:
605         df_0 = df_0.append({'cratio': cratio_act, 'speed': speed_act, 'category
    ': 0}, ignore_index=True)
606
607 for i in range (0, len(y_pred_label)):
608     cratio_act = x_test.iloc[i, 0]
609     speed_act = x_test.iloc[i, 1]
610     if y_pred_label[i] != y_true_label[i] and y_true_label[i] == 0.0:
611         df_0 = df_0.append({'cratio': cratio_act, 'speed': speed_act, 'category
    ': -1}, ignore_index=True)
612
613
614 # In[307]:
615
616
617 fig = px.scatter(df_0, "cratio", "speed", color = "category")
618 fig.show()
619
620
621 # In[308]:
622
623
624 df_1 = pd.DataFrame(columns=['cratio', 'speed', 'category'])
625
626
627 # In[309]:
628
629
630 for i in range (0, len(y_pred_label)):
631     cratio_act = x_test.iloc[i, 0]
632     speed_act = x_test.iloc[i, 1]
633     categ_act = y_test_nocod.iloc[i, 0]
634     if y_pred_label[i] == y_true_label[i] and y_pred_label[i] == 1.0:
635         df_1 = df_1.append({'cratio': cratio_act, 'speed': speed_act, 'category
    ': 1}, ignore_index=True)
636
637 for i in range (0, len(y_pred_label)):
638     cratio_act = x_test.iloc[i, 0]
639     speed_act = x_test.iloc[i, 1]
640     if y_pred_label[i] != y_true_label[i] and y_true_label[i] == 1.0:
```

```
641         df_1 = df_1.append({'cratio': cratio_act, 'speed': speed_act, 'category
    ': -2}, ignore_index=True)
642
643
644 # In[310]:
645
646
647 fig = px.scatter(df_1, "cratio", "speed", color = "category")
648 fig.show()
```