Facultad de Ciencias

Implementación de modelos de Deep Learning en dispositivos de bajo coste

(Implementation of Deep Learning Models on Low-Cost Devices)

Trabajo de Fin de Máster

para acceder al

MÁSTER EN DATA SCIENCE

Autor: Luis Mier Bedia

Director\es: Adolfo Cobo García

José Julián Valdiande Gutiérrez

Julio - 2023

ÍNDICE

AGRADECIMIENTOS

Este trabajo se ha basado en los datos proporcionados por las empresas eDrónica, Tecnología para Vehículos No Tripulados, S.L., y Ecohydros, S.L.; y en medios aportados por los proyectos PID2019-107270RB-C21 (MCIN/ AEI /10.13039/501100011033) y TED2021-130378B-C21 (MCIN/ AEI /10.13039/501100011033 y NextGenerationEU/PRTR)

RESUMEN

En el presente documento se realiza un análisis exhaustivo del proceso de diseño, creación y prueba de modelos de redes neuronales utilizando la biblioteca Tensorflow. El objetivo principal es comprimir estos modelos utilizando diversos métodos con el fin de generar modelos compatibles con dispositivos de bajo coste y limitadas capacidades computacionales.

El análisis abarcará el flujo de trabajo completo, desde la evaluación de los datos proporcionados por eDrónica, provenientes de un sensor de medición de calidad de agua, hasta la implementación de los modelos de Deep Learning en dispositivos como la Raspberry PI 4 o un dispositivo basado en un sistema en chip (SoC) ESP32.

La implementación de estos modelos en dispositivos de esta naturaleza representa un proceso que combina el Deep Learning convencional con el Edge Computing. Esto posibilita la creación de un sistema capaz de realizar mediciones de calidad del agua y procesarlas para inferir la calidad del mencionado recurso.

ABSTRACT

This document presents a comprehensive analysis of the process of designing, creating, and testing neural network models using the Tensorflow library. The main objective is to compress these models using various methods in order to generate models that are compatible with low-cost devices and have limited computational capabilities.

The analysis covers the complete workflow, starting from the evaluation of data provided by eDrónica, which is collected from a water quality measurement sensor, to the implementation of Deep Learning models on devices such as the Raspberry PI 4 or a system-on-chip (SoC) based device like ESP32.

The implementation of these models on devices of this nature involves a combination of conventional Deep Learning techniques with Edge Computing. This enables the creation

of a system capable of both measuring water quality and processing the data to infer its overall quality.

INTRODUCCIÓN

La evaluación de la calidad del agua en cuerpos de agua afectados por la eutrofización o blooms de algas nocivas (HAB, Harmful Algal Blooms) requiere el conocimiento preciso de las concentraciones *(mg/l)* de nutrientes presentes. Sin embargo, el monitoreo continuo de nutrientes sigue siendo un desafío debido a las dificultades técnicas y los altos costos asociados con la toma de mediciones en tiempo real. Para abordar esta problemática, la espectrofotometría ultravioleta (UV) se presenta como una técnica eficiente y sencilla, especialmente en el caso del ion nitrato, que exhibe una fuerte absorción en el rango UV. La estimación del nitrato utilizando la segunda derivada de la absorbancia (SDA, Second Derivative of Absorbance) del espectro de absorción fue propuesta por Crumpton et al. (1992) [1]. Esta metodología permite mitigar las señales relacionadas con la absorbancia difusa causada por coloides y partículas en suspensión, a la vez que amplifica las señales de diferentes compuestos, generando picos en el espectro que corresponden a los puntos de máxima absorción.

Causse et al. (2017) [2] proponen un nuevo enfoque para mejorar la determinación de nitrato mediante el uso de la SDA del espectro UV. Su método optimiza la precisión y confiabilidad de la medición, permitiendo una mayor sensibilidad y eliminando las interferencias causadas por otros componentes presentes en la muestra. Al reducir el ruido y maximizar la señal de interés, este enfoque ofrece una herramienta más efectiva para la cuantificación precisa del nitrato en aguas afectadas por eutrofización o blooms de algas nocivas. En resumen, la aplicación de la espectrofotometría UV y el análisis de la segunda derivada de la absorbancia ofrecen una solución valiosa y eficiente para la estimación precisa de concentraciones *(mg/l)* de nitrato en aguas contaminadas, mejorando así la evaluación de la calidad del agua y facilitando la toma de decisiones relacionadas con su gestión y conservación.

OBJETIVO

El propósito de este proyecto es evaluar la viabilidad de un sistema de Edge Computing que posibilite la adquisición de datos de un sensor utilizado para medir la calidad del

agua, y su procesamiento a través de una red neuronal capaz de estimar los niveles de nitratos a partir de las lecturas de los diferentes canales del sensor.

Para lograr este objetivo, se llevará a cabo el entrenamiento de una red neuronal y se aplicarán diversas técnicas para adaptarla a dispositivos de baja capacidad computacional y bajo costo. El objetivo final es desarrollar un sistema de inferencia que pueda operar en el mismo entorno en el que se encuentra el sensor de medición de espectros, permitiendo una integración eficiente y efectiva de ambos sistemas.

CAPÍTULO 1: DATOS ORIGINALES Y ETL *(EXTRACT, TRANSFORM, LOAD)*

La memoria de este proyecto comienza con un análisis de los datos originales proporcionados por eDrónica.

Tal y como se comentó previamente en el contexto general, los datos analizados se obtienen a partir de las mediciones de una serie de sensores de calidad de agua. Dichos sensores se colocan en el fondo de depósitos de agua (pantanos, lagos...) y realizan una serie de lecturas sobre la cantidad de luz que es capaz de llegar hasta ellos. Esta luz se divide en diversos espectros y se toman medidas de cada uno de ellos individualmente.

En este caso contamos con un sensor TriOS ProPs CW. Se trata de un transmisómetro ultravioleta hiperespectral sumergible, de tamaño y peso reducidos. Este dispositivo cuenta con una lámpara de deuterio de baja intensidad (5 W) como fuente de radiación ultravioleta y un detector de silicio de 256 canales que permite medir la absorción de longitudes de onda comprendidas en el rango de 190 a 360 nm, Además de eso, cuenta con un paso óptico ajustable entre 10 y 60 mm (para el presente estudio de escogió un paso de 10 mm, lo cual permite obtener datos más precisos en un rango amplio de concentración de nitratos).

Figura 1: Sensor TriOS ProPs

Este dispositivo se encarga de realizar las mediciones y almacenar las lecturas que obtiene a lo largo del tiempo. El almacenamiento de estos datos se hace en una serie de archivos *.dat* que, además de los datos correspondientes a cada una de las franjas de medición (de las 256 con las que cuenta), contienen también datos sobre la configuración

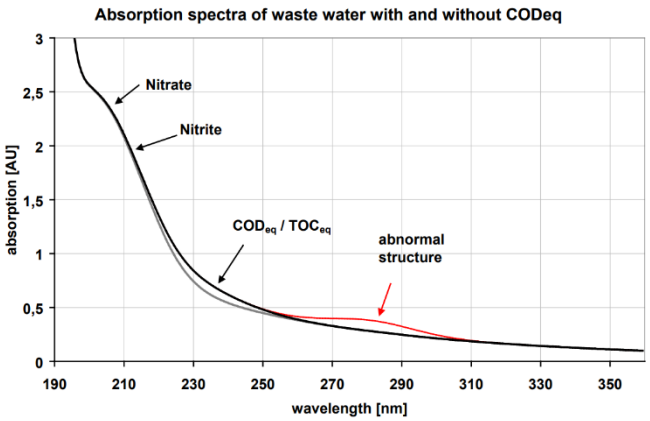


Figura 2:Ejemplo de curva de absorbancia

del dispositivo y las fechas en las cuales se han hecho las mediciones. Se puede ver un ejemplo de estas cabeceras en la Figura 3.

```
1   [Spectrum]
2   Version=1
3   IDData=VAL1_R1-1_2020-06-22_18-59-10_020
4   IDDevice=D16D
5   IDDataType=SPECTRUM
6   IDDataTypeSub1=RAW
7   DateTime=2020-06-22 18:59:10
8   [Attributes]
9   IntegrationTime=512
10  Unit1=$05 $00 Pixel
11  Unit2=$03 $05 Intensity counts
12  Unit3=$f0 $05 Error counts
13  Unit4=$f1 $00 Status
14  [END] of [Attributes]
```

Figura 3: Cabecera de los archivos generados por el sensor.

Además de eso, contiene también las mediciones que obtiene el sensor para cada uno de los 256 canales de medición que utiliza. Se puede ver un ejemplo de los datos, tanto al inicio como al final, en la Figura 4.

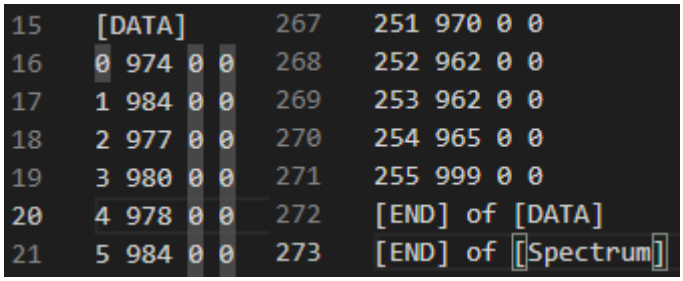


Figura 4: Inicio y fin de los datos del sensor

Tal y como se puede ver, contamos con 256 filas de datos, una por cada uno de los canales de silicio que conforman el sensor. Los datos que resultan relevantes para este estudio son los que se encuentran en la segunda columna, que son los que indican la medición que ha podido hacer el sensor.

Por un lado, tenemos los datos del sensor, y por otro lado (y pensando en el posterior proceso de entrenamiento de la red neuronal) se tienen datos de medidas de laboratorio sobre estos mismos datos. En otras condiciones, la inferencia de las cantidades de nitratos en el agua se realiza a partir de la siguiente fórmula:

$$A_\lambda = -log_{10}\left(\frac{I_\lambda / t}{I_{\lambda,B} / t_B}\right);$$

donde: A_λ es la absorbancia a una longitud de onda λ; ii) I_λ y t se corresponden con la intensidad (I) a una determinada λ y el tiempo total de integración en cada una de las muestras; y iii) $I_{\lambda,B}$ y t_B representan la intensidad y tiempo de integración para una muestra de agua desionizada (blanco).

A partir del cálculo de esta absorbancia se puede obtener una medida de las concentraciones de nitratos en cada muestra de agua medida, a partir de la siguiente fórmula:

$$SDA_\lambda = k * \frac{(A_{\lambda-h} + A_{\lambda+h} - 2 * A_\lambda)}{h^2}$$

donde A_λ es la absorbancia a una determinada longitud de onda λ, k es una constante arbitraria (k = *1000*) y h es el desfase derivativo o *lags* (h = 10 nm).

Este proceso se basa en la metodología general descrita por *Causse et al. (2016)*

Una vez se calculan las concentraciones de nitratos, se tiene una relación directa entre las medidas de los 256 canales del sensor y la presencia de nitratos en el agua.

El objetivo de este TFM es tomar estos datos como base de entrenamiento para una red neuronal, y verificar si es posible inferir las concentraciones de nitratos a partir de las mediciones del sensor sin necesidad de analizar las muestras en un laboratorio. Para ello, se hará un tratamiento de los datos de tal manera que se tenga un dataframe en el cual las columnas serán cada una de las 256 bandas de medición del sensor, con una columna extra que será la concentración de nitratos analizada para esa medición en concreto

Las filas serán todas y cada una de las mediciones disponibles al momento de la realización de este proyecto. Contamos al mismo tiempo con dos conjuntos de datos distintos, derivados de dos localizaciones distintas, por lo que el objetivo será combinarlos en un solo dataframe con el que se realizará el entrenamiento de la red neuronal.

En las siguientes líneas se da una explicación del proceso de ETL (Extracción, transformación y carga) llevado a cabo para organizar los datos de la forma deseada.

Debido a que los datos obtenidos del sensor vienen altamente estandarizados, es sencillo implementar una función que generalice a todos estos archivos y permita su transformación de forma eficaz.

El primer paso es acceder desde el entorno de programación a la carpeta en la cual se almacenan todos los archivos *.dat*:

```
foldername =
"./Files/data/props_mobile_data_input/Descarga_SD_R1_from1to9/"
dataframes = {}
```

A continuación, se crea un diccionario vacío llamado dataframes que se utilizará para almacenar los DataFrames resultantes. El siguiente paso en este proceso es el de acceder a cada uno de los archivos y realizar su tratamiento para obtener una estructura como la deseada anteriormente:

```
for filename in os.listdir(foldername):
    if filename.endswith(".dat"):
        filepath = os.path.join(foldername, filename)
        with open(filepath) as f:
            data_lines = []
            flag = False
            for line in f:
                if "[DATA]" in line:
                    flag = True
                    continue
                if flag:
                    if "[END]" in line:
                        break
                    data_lines.append(line)
```

Se itera sobre cada uno de los archivos, y para cada uno de ellos se seleccionan únicamente las líneas entre [DATA] y [END]. El objetivo de esto es desechar las líneas correspondientes a las cabeceras (como ya se vieron anteriormente) y quedarse únicamente con las líneas que contienen los datos que interesan de cara al análisis.

```
        data_values = []
        for line in data_lines:
            values = line.strip().split()[:2]
            data_values.append(values)

        df = pd.DataFrame(data_values, columns=["Pixel", "Intensity"])
```

En las líneas superiores se eliminan las dos últimas columnas del archivo de datos (ya que no contienen información relevante para el estudio) y las líneas seleccionadas se introducen en un dataframe cuyas columnas son Pixel (habrá 256 columnas, una por cada canal) y otra Intensity (la medición realizada por cada canal de medidas)

```
        with open(filepath) as f:
            datetime = None
            for line in f:
                if "DateTime" in line:
                    datetime = line.strip().split("=")[1]
                    break

        datetime_obj = pd.to_datetime(datetime, format="%Y/%m/%d",
dayfirst=True)
        df["DATE"] = datetime_obj.strftime("%d/%m/%Y")
```

```
        df["HOUR"] = datetime_obj.strftime("%H:%M:%S")

         # Pivoteo del dataframe
        df = pd.pivot_table(df, index=['DATE', 'HOUR'], columns='Pixel',
values='Intensity')
        df.columns = ['Pixel_' + str(col) for col in df.columns]
        df.reset_index(inplace=True)
        df['DATE'] = pd.to_datetime(df['DATE'], format="%d/%m/%Y",
dayfirst=True)

        # Obtener el nombre del archivo sin la extensión .dat
        df_name = os.path.splitext(os.path.basename(filepath))[0]

        # Obtener la parte del nombre que corresponde al código de
muestra (p. ej. VAL1_R1_1)
        cod_sample = "_".join(df_name.split("_")[:3])

        # Asignar el valor de COD_SAMPLE a todas las filas del dataframe
        df["COD_SAMPLE"] = cod_sample

        # asignamos el nombre del archivo al DataFrame
        df_name = os.path.splitext(os.path.basename(filepath))[0]
        dataframes[df_name] = df
```

Por último, se realizan diversos formateos correspondientes a campos de fechas y se pivota el dataframe. El objetivo de pivotar el dataframe es tener, tal y como se comentó previamente, una columna para cada pixel (ya que ahora mismo se tienen todos los píxeles en una sola columna) y en las filas las mediciones de intensidad.

Una vez que se completa este proceso, se tiene finalmente un diccionario que contiene un dataframe por cada medición. Para obtener el dataframe final con el cual se realiza el análisis y entrenamiento de la red neuronal es necesario concatenar los valores de dicho diccionario, para así obtener un dataframe de mayores dimensiones.

Se puede ver dicho dataframe en la Figura 5:

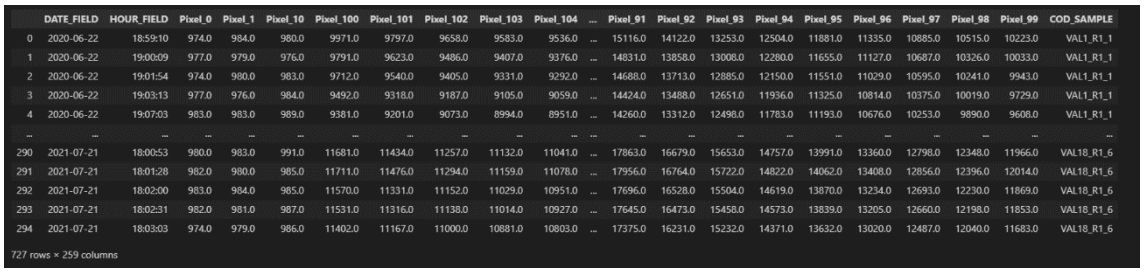
	DATE	HOUR	Pixel_0	Pixel_1	Pixel_10	Pixel_100	Pixel_101	Pixel_102	Pixel_103	Pixel_104	...	Pixel_91	Pixel_92	Pixel_93	Pixel_94	Pixel_95	Pixel_96	Pixel_97	Pixel_98	Pixel_99	COD_SAMPLE
0	2020-06-22	18:59:10	974.0	984.0	980.0	9971.0	9797.0	9658.0	9583.0	9536.0	...	15116.0	14122.0	13253.0	12504.0	11881.0	11335.0	10885.0	10515.0	10223.0	VAL1_R1_1
1	2020-06-22	19:00:09	977.0	979.0	976.0	9791.0	9623.0	9486.0	9407.0	9376.0	...	14831.0	13858.0	13008.0	12280.0	11655.0	11127.0	10687.0	10326.0	10033.0	VAL1_R1_1
2	2020-06-22	19:01:54	974.0	980.0	983.0	9712.0	9540.0	9405.0	9331.0	9292.0	...	14688.0	13713.0	12885.0	12150.0	11551.0	11029.0	10595.0	10241.0	9943.0	VAL1_R1_1
3	2020-06-22	19:03:13	977.0	976.0	984.0	9492.0	9318.0	9187.0	9105.0	9059.0	...	14424.0	13488.0	12651.0	11936.0	11325.0	10814.0	10375.0	10019.0	9729.0	VAL1_R1_1
4	2020-06-22	19:07:03	983.0	983.0	989.0	9381.0	9201.0	9073.0	8994.0	8951.0	...	14260.0	13312.0	12498.0	11783.0	11193.0	10676.0	10253.0	9890.0	9608.0	VAL1_R1_1
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
427	2020-10-22	01:03:49	981.0	992.0	1007.0	9112.0	8956.0	8850.0	8785.0	8747.0	...	13489.0	12636.0	11918.0	11259.0	10737.0	10281.0	9890.0	9566.0	9309.0	VAL9_R1_6
428	2020-10-22	01:04:25	989.0	984.0	994.0	9067.0	8919.0	8809.0	8736.0	8706.0	...	13410.0	12573.0	11850.0	11214.0	10675.0	10227.0	9841.0	9518.0	9269.0	VAL9_R1_6
429	2020-10-22	01:04:59	974.0	984.0	1004.0	9015.0	8876.0	8764.0	8705.0	8671.0	...	13348.0	12507.0	11784.0	11155.0	10631.0	10174.0	9785.0	9482.0	9224.0	VAL9_R1_6
430	2020-10-22	01:05:31	989.0	983.0	1000.0	8972.0	8826.0	8720.0	8647.0	8615.0	...	13257.0	12447.0	11724.0	11089.0	10559.0	10119.0	9732.0	9425.0	9170.0	VAL9_R1_6
431	2020-10-22	01:06:04	989.0	985.0	1005.0	8942.0	8806.0	8699.0	8631.0	8597.0	...	13226.0	12403.0	11700.0	11072.0	10538.0	10087.0	9712.0	9391.0	9155.0	VAL9_R1_6

432 rows × 259 columns

Figura 5: Dataframe para una de las localizaciones

Tal y como se comentó previamente, se cuenta con mediciones en dos localizaciones distintas, por lo que este proceso de ETL deberá ser repetido con los datos de la siguiente localización.

Una vez se hace, se obtiene un dataframe en donde se conjuntan las mediciones del sensor en ambas localizaciones. Se puede ver este dataframe en la Figura 6:



	DATE_FIELD	HOUR_FIELD	Pixel_0	Pixel_1	Pixel_10	Pixel_100	Pixel_101	Pixel_102	Pixel_103	Pixel_104	...	Pixel_91	Pixel_92	Pixel_93	Pixel_94	Pixel_95	Pixel_96	Pixel_97	Pixel_98	Pixel_99	COD_SAMPLE
0	2020-06-22	18:59:10	974.0	984.0	980.0	9971.0	9797.0	9658.0	9583.0	9536.0	...	15116.0	14122.0	13253.0	12504.0	11881.0	11335.0	10885.0	10515.0	10223.0	VAL1_R1_1
1	2020-06-22	19:00:09	977.0	979.0	976.0	9791.0	9623.0	9486.0	9407.0	9376.0	...	14831.0	13858.0	13008.0	12280.0	11655.0	11127.0	10687.0	10326.0	10033.0	VAL1_R1_1
2	2020-06-22	19:01:54	974.0	980.0	983.0	9712.0	9540.0	9405.0	9331.0	9292.0	...	14688.0	13713.0	12885.0	12150.0	11551.0	11029.0	10595.0	10241.0	9943.0	VAL1_R1_1
3	2020-06-22	19:03:13	977.0	976.0	984.0	9492.0	9318.0	9187.0	9105.0	9059.0	...	14424.0	13488.0	12651.0	11936.0	11325.0	10814.0	10375.0	10019.0	9729.0	VAL1_R1_1
4	2020-06-22	19:07:03	983.0	983.0	989.0	9381.0	9201.0	9073.0	8994.0	8951.0	...	14260.0	13312.0	12498.0	11783.0	11193.0	10676.0	10253.0	9890.0	9608.0	VAL1_R1_1
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
290	2021-07-21	18:00:53	980.0	983.0	991.0	11681.0	11434.0	11257.0	11132.0	11041.0	...	17863.0	16679.0	15653.0	14757.0	13991.0	13360.0	12798.0	12348.0	11966.0	VAL18_R1_6
291	2021-07-21	18:01:28	982.0	980.0	985.0	11711.0	11476.0	11294.0	11159.0	11078.0	...	17956.0	16764.0	15722.0	14822.0	14062.0	13408.0	12856.0	12396.0	12014.0	VAL18_R1_6
292	2021-07-21	18:02:00	983.0	984.0	985.0	11570.0	11331.0	11152.0	11029.0	10951.0	...	17696.0	16528.0	15504.0	14619.0	13870.0	13234.0	12693.0	12230.0	11869.0	VAL18_R1_6
293	2021-07-21	18:02:31	982.0	981.0	987.0	11531.0	11316.0	11138.0	11014.0	10927.0	...	17645.0	16473.0	15458.0	14573.0	13839.0	13205.0	12660.0	12198.0	11853.0	VAL18_R1_6
294	2021-07-21	18:03:03	974.0	979.0	986.0	11402.0	11167.0	11000.0	10881.0	10803.0	...	17375.0	16231.0	15232.0	14371.0	13632.0	13020.0	12487.0	12040.0	11683.0	VAL18_R1_6

727 rows × 259 columns

Figura 6: Dataframe final con todas las mediciones del sensor

El siguiente paso es transformar los datos de las mediciones en laboratorio, y asignarlos a las mediciones correspondientes del sensor. En las siguientes líneas se puede ver el proceso seguido.

```
data_medicionesR1 =
pd.read_csv('./Files/data/props_mobile_data_output/R1_curated_TUKEY_ECOH_
IPRO.csv', delimiter = ';')

data_medicionesR1 = data_medicionesR1[['DATE', 'HOUR', 'COD_SAMPLE ',
'N_NO3']]

data_medicionesR1 = data_medicionesR1.rename(columns={'COD_SAMPLE ':
'COD_SAMPLE', 'DATE': 'DATE_LAB', 'HOUR': 'HOUR_LAB'})
```

```
data_medicionesR1["DATE_LAB"] =
pd.to_datetime(data_medicionesR1["DATE_LAB"], format="%d/%m/%Y", dayfirst
= True)

df_mediciones_lab = data_medicionesR1
```

Se comienza abriendo el archivo que contiene dichas medidas de laboratorio. Luego, se seleccionan únicamente las columnas que resultan de interés para el estudio y se les asigna el nombre adecuado. Es necesario también formatear de forma adecuada los campos de fecha, tal y como se hizo en los pasos anteriores.

Una vez se finaliza este proceso, el último paso es combinar el dataframe de las medidas del sensor, con este último dataframe de medidas en laboratorio. Para ello, se hace un Join entre ambos dataframes a través del campo COD_SAMPLE, que permite saber (tanto en un dataframe como en el otro) qué medida de laboratorio se corresponde a qué medida del sensor.

```
df_final = df_mediciones_lab.merge(df_mediciones_campo, left_on =
'COD_SAMPLE', right_on = 'COD_SAMPLE' , how = 'inner')

df_final_train = df_final.drop(['DATE_LAB', 'HOUR_LAB', 'COD_SAMPLE',
'DATE_FIELD', 'HOUR_FIELD'], axis = 1)

df_final_train['N_NO3'] = df_final_train['N_NO3'].str.replace(',',
'.').astype(float)
```

Finalmente, el dataframe **df_final_train** ya contiene toda la información necesaria, y en el formato deseado desde el inicio. Se puede ver la estructura de dicho dataframe en la Figura 7.

	N_NO3	Pixel_0	Pixel_1	Pixel_10	Pixel_100	Pixel_101	Pixel_102	Pixel_103	Pixel_104	Pixel_105	...	Pixel_90	Pixel_91	Pixel_92	Pixel_93	Pixel_94	Pixel_95	Pixel_96	Pixel_97	Pixel_98	Pixel_99
0	4.27	974.0	984.0	980.0	9971.0	9797.0	9658.0	9583.0	9536.0	9523.0	...	16272.0	15116.0	14122.0	13253.0	12504.0	11881.0	11335.0	10885.0	10515.0	10223.0
1	4.27	977.0	979.0	976.0	9791.0	9623.0	9486.0	9407.0	9376.0	9366.0	...	15969.0	14831.0	13858.0	13008.0	12280.0	11655.0	11127.0	10687.0	10326.0	10033.0
2	4.27	974.0	980.0	983.0	9712.0	9540.0	9405.0	9331.0	9292.0	9291.0	...	15799.0	14688.0	13713.0	12885.0	12150.0	11551.0	11029.0	10595.0	10241.0	9943.0
3	4.27	977.0	976.0	984.0	9492.0	9318.0	9187.0	9105.0	9059.0	9055.0	...	15543.0	14424.0	13488.0	12651.0	11936.0	11325.0	10814.0	10375.0	10019.0	9729.0
4	4.27	983.0	983.0	989.0	9381.0	9201.0	9073.0	8994.0	8951.0	8932.0	...	15343.0	14260.0	13312.0	12498.0	11783.0	11193.0	10676.0	10253.0	9890.0	9608.0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
620	7.68	980.0	983.0	991.0	11681.0	11434.0	11257.0	11132.0	11041.0	10991.0	...	19237.0	17863.0	16679.0	15653.0	14757.0	13991.0	13360.0	12798.0	12348.0	11966.0
621	7.68	982.0	980.0	985.0	11711.0	11476.0	11294.0	11159.0	11078.0	11034.0	...	19336.0	17956.0	16764.0	15722.0	14822.0	14062.0	13408.0	12856.0	12396.0	12014.0
622	7.68	983.0	984.0	985.0	11570.0	11331.0	11152.0	11029.0	10951.0	10899.0	...	19059.0	17696.0	16528.0	15504.0	14619.0	13870.0	13234.0	12693.0	12230.0	11869.0
623	7.68	982.0	981.0	987.0	11531.0	11316.0	11138.0	11014.0	10927.0	10893.0	...	18985.0	17645.0	16473.0	15458.0	14573.0	13839.0	13205.0	12660.0	12198.0	11853.0
624	7.68	974.0	979.0	986.0	11402.0	11167.0	11000.0	10881.0	10803.0	10754.0	...	18694.0	17375.0	16231.0	15232.0	14371.0	13632.0	13020.0	12487.0	12040.0	11683.0

625 rows × 257 columns

Figura 7: Dataframe final con el que se procede al entrenamiento de la red neuronal

Tal y como se ve, se cuenta con una columna correspondiente a las medidas de concentraciones de nitratos, y toda una serie de columnas correspondientes a las medidas de cada uno de los canales del sensor. De este modo, se tiene en cada fila la medida de nitratos correspondiente a las medidas del sensor en ese instante.

El dataframe final cuenta con 3494 filas y 257 columnas, de las cuales las 256 correspondientes a los píxeles serán los predictores y la columna N_NO3 será el valor a predecir.

De este modo, se tienen los datos en el formato deseado y se puede dar por concluido el proceso de ETL.

En el siguiente capítulo se verá el proceso de entrenamiento y testeo de una red neuronal usando estos datos como conjunto de entrenamiento.

CAPÍTULO 2: ENTRENAMIENTO DE RED NEURONAL

Como ya se vio en el capítulo anterior, en este punto ya se cuenta con un conjunto de datos apto para el entrenamiento de diversos modelos de predicción. En el caso de este proyecto, se opta por una red neuronal.

Dicha red se construirá utilizando Tensorflow y Keras como frameworks de trabajo, y se alimentará con el conjunto de datos obtenido en el capítulo anterior con el objetivo de conocer su rendimiento de cara a ser posteriormente convertido con Tensorflow Lite, e implementado en un dispositivo de bajo coste.

El primer paso en este proceso es el de dividir el conjunto de datos en dos subconjuntos; uno para entrenamiento y otro para validación. En las siguientes líneas podemos ver el proceso, en el cual se asigna un 80% para entrenamiento, y el 20% restante para validación. Además, se le indica que la variable a predecir será la correspondiente a la columna N_NO3:

```
train_dataset = df_final_train.sample(frac=0.8,random_state=0)
test_dataset = df_final_train.drop(train_dataset.index)

train_labels = train_dataset.pop('N_NO3')
test_labels = test_dataset.pop('N_NO3')
```

Una vez hecha esta división, se normalizan los datos tanto de test como de train:

```
def norm(x):
  return (x - train_stats['mean']) / train_stats['std']

normed_train_data = norm(train_dataset)
normed_test_data = norm(test_dataset)
```

El siguiente paso en este proceso es el de especificar el modelo que se quiere usar. Para ello, se tendrá que definir la arquitectura de la red, el número de capas, número de neuronas por capa, funciones de activación de cada capa...

En las siguientes líneas de código se puede ver el código destinado a la creación de la red neuronal en la que se basará todo el proceso posterior de análisis:

```
  model = keras.Sequential([
    layers.Dense(32, activation='relu',
input_shape=[len(train_dataset.keys())]),
    layers.Dense(64, activation='relu'),
    layers.Dense(64, activation='relu'),
    layers.Dense(32, activation='relu'),
    layers.Dense(1)
  ]
```

A continuación, se hace una pequeña explicación de las propiedades de esta red neuronal.

El modelo de Keras que se muestra es una red neuronal secuencial, construida utilizando la API secuencial de Keras. Esta API permite crear modelos de manera secuencial, donde las capas se apilan una encima de otra.

El modelo se define utilizando la función keras.Sequential([]), donde se pasan las capas del modelo como una lista de argumentos. A continuación, se describen las capas utilizadas en el modelo:

- Capa densa (Dense) con 32 neuronas: Esta capa utiliza la función de activación ReLU (Rectified Linear Unit) y tiene una forma de entrada definida por la longitud del conjunto de datos de entrenamiento (len(train_dataset.keys())). La función de activación ReLU proporciona una salida no lineal al aplicar la función max(0, x) a cada elemento de la capa. Esta capa es la capa de entrada del modelo.
- Capa densa (Dense) con 64 neuronas: Al igual que la capa anterior, utiliza la función de activación ReLU. Esta capa no requiere especificar una forma de entrada, ya que automáticamente toma la salida de la capa anterior como entrada.
- Capa densa (Dense) con 64 neuronas: También utiliza la función de activación ReLU y se conecta a la capa anterior.
- Capa densa (Dense) con 32 neuronas: Utiliza la función de activación ReLU y se conecta a la capa anterior.
- Capa densa (Dense) con 1 neurona: Esta es la capa de salida del modelo y no se especifica ninguna función de activación. Al no definir una función de activación, se espera que esta capa produzca una salida lineal.

En resumen, este modelo de red neuronal cuenta con cinco capas, de las cuales las cuatro primeras cuentan con una función de activación de tipo ReLU. La última capa contiene solo una neurona, que será la que devuelva el valor inferido a partir de las entradas de la red.

Este modelo se utilizará para realizar tareas de regresión, donde se busca predecir un valor numérico continuo, tal y como se requiere en el presente estudio.

Una vez se tiene el modelo definido, es momento de entrenarlo con los datos de entrenamiento anteriormente especificados. Para ello, se llama al parámetro .fit del modelo y se le pasan los valores de entrenamiento, tal y como se puede ver en las siguientes líneas:

```
EPOCHS = 1000

history = model.fit(
  normed_train_data, train_labels,
  epochs=EPOCHS, validation_split = 0.2, verbose=0,
  callbacks=[PrintDot()])
```

En este caso, se ajusta el número inicial de épocas en 1000, pero a través del Early Stopping se logra que el entrenamiento pare antes de que se produzca el fenómeno de sobreajuste.

El sobreajuste (o sobreentrenamiento) es un fenómeno común en el campo del aprendizaje automático, incluyendo las redes neuronales, donde un modelo es capaz de ajustarse muy bien a los datos de entrenamiento, pero no generaliza bien a nuevos datos no vistos. En otras palabras, el modelo se "memoriza" los ejemplos de entrenamiento en lugar de capturar los patrones subyacentes que se aplican a datos no vistos.

El sobreajuste ocurre cuando la capacidad del modelo es demasiado alta en relación con la cantidad y calidad de los datos de entrenamiento disponibles. Esto puede suceder cuando la red neuronal es muy grande y tiene muchos parámetros, lo que le permite capturar incluso el ruido y las fluctuaciones aleatorias presentes en los datos de entrenamiento. A medida que el modelo se ajusta cada vez más a esos detalles específicos

de los datos de entrenamiento, se vuelve menos capaz de generalizar a nuevas muestras.[3]

Conociendo qué es el sobreajuste y por qué se produce, se aplica Early Stopping al entrenamiento de la red neuronal con el objetivo de limitar el número de épocas que dicho entrenamiento aplica. Con esto se logra cortar el proceso antes de que este fenómeno aparezca. A través del parámetro Patience se le indica cuán permisivo tiene que ser en este proceso; cuanto menor sea el parámetro, antes cortará el proceso de entrenamiento en cuanto detecte signos de sobreajuste, y, por el contrario, cuanto mayor sea más tardará en cortar dicho proceso, dejando más libertad para que el modelo presente un ligero sobreajuste, pero sin llegar a cumplir con el número original de épocas establecido.

Una vez se tiene el modelo entrenado sobre el conjunto de entrenamiento, es hora de verificar su comportamiento sobre el conjunto de test.

Para hacer dicha verificación se aplica el argumento .predict sobre el modelo y se le pasan los datos sobre los que se quiere realizar dicha predicción, que en este caso son los datos de test:

```
test_predictions = model.predict(normed_test_data).flatten()
```

Y para verificar los resultados que arroja dicha predicción, se calcula el parámetro R2.

El parámetro R2, también conocido como coeficiente de determinación, es una medida estadística utilizada para evaluar la calidad de un modelo de regresión en redes neuronales y en otros métodos de regresión. El coeficiente de determinación proporciona información sobre qué tan bien se ajustan los valores predichos por el modelo a los valores reales de la variable dependiente.

Formalmente, el coeficiente de determinación R2 se define como la proporción de la varianza total de la variable dependiente que es explicada por el modelo de regresión. Toma valores entre 0 y 1, donde un valor más cercano a 1 indica que el modelo es capaz de explicar una mayor parte de la variabilidad en los datos observados. [4]

En este caso, el valor de R2 arrojado por el modelo es:

$$R^2_{ORIGINAL} = 0.92$$

Como se puede ver, el valor de R^2 inicial es bastante alto. Se puede ver este resultado gráficamente en la Figura 8:

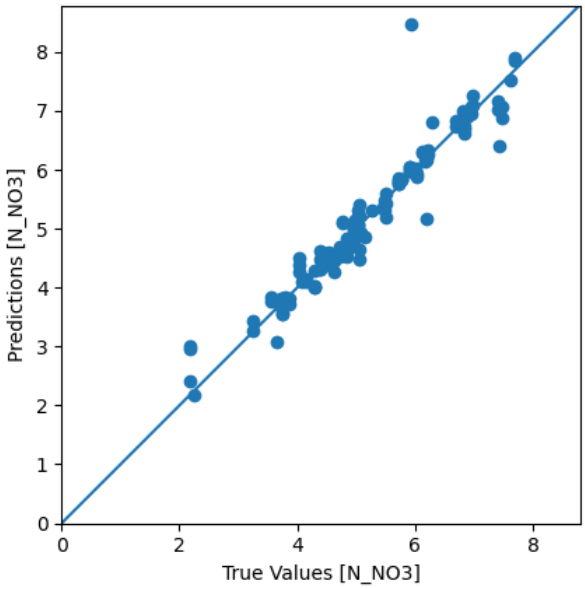


Figura 8: Puntos de datos y regresión propuesta por el modelo

El siguiente paso comprenderá la aplicación de los distintos métodos de poda y reducción de tamaño que se han seguido, con el objetivo de crear un modelo de menor tamaño que sea capaz de funcionar con un rendimiento similar, pero en dispositivos con altas limitaciones de memoria y capacidad de procesamiento.

CAPÍTULO 3: PRUNNING Y MODELOS TF LITE

Como se ha visto en el capítulo anterior, el modelo creado presenta un rendimiento bastante satisfactorio, pero dado que el objetivo de este proyecto es el de implementar un modelo en un dispositivo de bajo coste, con capacidades de cómputo limitadas, lo que se busca es crear una versión de dicho modelo mucho más compacta y liviana, computacionalmente hablando.

Para ello, se aplican distintas técnicas que se verán a lo largo del presente capítulo, incluyendo la poda de pesos en la red neuronal, la compresión en modelo de TensorFlow Lite, la conversión a 16 Bits y la cuantización. El primer paso a aplicar es el de la poda, o pruning, de los pesos.

3.1 Poda o pruning de los pesos del modelo

El método de poda, o "pruning" en inglés, es una técnica utilizada en redes neuronales para reducir el tamaño o la complejidad del modelo mediante la eliminación selectiva de conexiones o unidades neuronales menos relevantes. El objetivo de la poda es mejorar la eficiencia y la generalización del modelo al eliminar la redundancia y sobreparametrización innecesaria. [5]

De este modo, se consigue también reducir el peso del modelo en memoria, así como la cantidad de recursos que necesita para realizar sus predicciones. Este es un primer paso para lograr un modelo más reducido y más apto para dispositivos limitados.

Relativo al modelo, para aplicar este proceso de poda se utilizan una serie de parámetros importados de la librería de Model optimization de TensorFlow. Concretamente. El parámetro *sparsity* permite indicar qué porcentaje de los pesos menos relevantes se quiere eliminar del modelo durante el proceso de poda:

```
pruning_params = {"pruning_schedule" :
tfmot.sparsity.keras.ConstantSparsity(target_sparsity = 0.80, begin_step
= 200, end_step = -1)}
```

En este caso, se le indica al modelo que se quieren eliminar el 80% de los pesos menos relevantes, tanto neuronas como sinapsis entre neuronas. De esta manera lo que se consigue es un modelo que mantiene sus ítems más relevantes, de cara a mantener un rendimiento aceptable, pero es capaz de discriminar aquellos que no aportan un valor suficiente al modelo. El resultado de este proceso, es un modelo que, idealmente, mantendrá un rendimiento similar al original, pero con una menor carga computacional.

El proceso de entrenamiento una vez completado este paso es similar a lo previamente visto con el modelo original. Se le indica un mismo número de épocas original y se le asigna también un parámetro de Early Stopping con el fin de evitar el sobreajuste. Sobre este modelo se vuelven a aportar los datos de entrenamiento durante su compilación, y posteriormente se le pasan los datos de test con el objetivo de validar los resultados.

En este caso, la regresión propuesta por el modelo es la que se puede ver en la Figura 9:

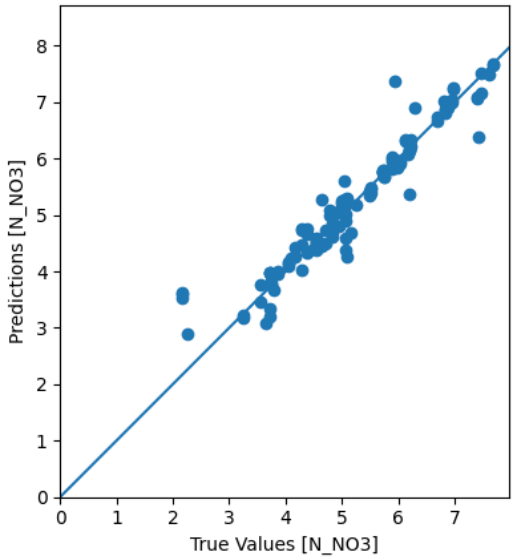


Figura 9: Regresión propuesta por el modelo con poda

Y el valor de R^2 obtenido en este caso es de:

$$R^2_{\text{PODADO}} = 0.9126$$

Tal y como se puede ver, el coeficiente de determinación en el modelo podado es inferior al del modelo original. Esto es un comportamiento esperado, porque lo que se está haciendo al fin y al cabo es limitar el modelo, retirándole algunos de sus parámetros,

El siguiente paso en el proceso de compresión y reducción de los modelos es el de crear modelos de TF Lite a partir de los dos modelos originales, tanto el podado como el no podado.

3.2 TensorFlow Lite

TensorFlow Lite [6] es una biblioteca de software desarrollada por Google que se utiliza para implementar modelos de redes neuronales en dispositivos móviles y dispositivos con recursos limitados. Proporciona una plataforma eficiente para ejecutar modelos de aprendizaje automático en dispositivos con capacidades computacionales más limitadas, como teléfonos móviles, dispositivos IoT (Internet de las cosas) y sistemas embebidos.

La principal función de TensorFlow Lite es la optimización de modelos de redes neuronales para su despliegue en dispositivos móviles y otros sistemas con restricciones de recursos. Esta optimización tiene como objetivo reducir el tamaño del modelo y mejorar la velocidad de inferencia, permitiendo así una ejecución más eficiente en dispositivos con recursos limitados, como memoria y capacidad de procesamiento.

TensorFlow Lite logra la optimización del modelo a través de varios enfoques:

- Conversión de modelos: TensorFlow Lite permite convertir modelos entrenados en TensorFlow (una biblioteca más completa de aprendizaje automático) a un formato más eficiente para su uso en dispositivos móviles. Durante esta conversión, se aplican técnicas de compresión, eliminación de capas innecesarias y cuantización de precisión (reducción del número de bits utilizados para representar los valores del modelo), lo que disminuye el tamaño del modelo y mejora la eficiencia computacional.
- Ejecución optimizada: TensorFlow Lite proporciona un intérprete de modelos que está diseñado para funcionar de manera eficiente en dispositivos con recursos

limitados. Este intérprete está optimizado para aprovechar las características específicas de los procesadores móviles y las arquitecturas de hardware disponibles en estos dispositivos. Además, TensorFlow Lite admite la aceleración de hardware, como el uso de unidades de procesamiento de gráficos (GPU) o unidades de procesamiento de redes neuronales (NPU), cuando están disponibles en el dispositivo.

- Soporte para múltiples lenguajes y plataformas: TensorFlow Lite ofrece soporte para diferentes lenguajes de programación, como Java, C++, Python y Swift, lo que facilita la integración de modelos en aplicaciones móviles y sistemas embebidos. Además, es compatible con varias plataformas, incluyendo Android, iOS, Linux y microcontroladores, lo que amplía su alcance y aplicabilidad.

El proceso para crear un modelo de TF Lite es el siguiente:

1. Se crea en un primer lugar el nombre que va a recibir dicho modelo en forma de string de Python.
2. Se crea una instancia del método TFLiteConverter, incluída en la librería de TF Lite y se le pasa el modelo original que se quiere convertir. En este caso se convertirán tanto el modelo sin poda como el modelo con poda.
3. Se guarda el modelo creado y se le da el nombre especificado en el primer paso.

El código para este proceso sería el siguiente:

```
TF_LITE_MODEL_FILE_NAME = "tf_lite_model.tflite"

''' Creamos el modelo de TF Lite a partir del modelo podado de Keras'''

converter = tf.lite.TFLiteConverter.from_keras_model(model_for_pruning)
tflite_model = converter.convert()

tflite_model_name = TF_LITE_MODEL_FILE_NAME
open(tflite_model_name, "wb").write(tflite_model)
```

De este modo, se tiene ya en el archivo *tf_lite_model.tflite* el modelo de TF Lite creado a partir del modelo podado de TF.

Una vez que se tiene el modelo de TF Lite creado, se han seguido otro paso adicional en el proceso de reducción y compresión del modelo; aplicar flotantes de 16 Bits.

La aplicación de flotantes de 16 Bits lo que hace es restringir con respecto al uso de valores enteros de 32 Bits, con lo cual se logra un espacio menor en memoria por parte del modelo a la hora de realizar las inferencias.

Para lograr esto el proceso es similar al anterior, pero con la diferencia de que ahora se le indica expresamente que convierta todos los valores del modelo a flotantes de 16 Bits mediante la siguiente sentencia:

```
tf_lite_converter.target_spec.supported_types = [tf.float16]
```

Por tanto, la conversion completa a un modelo de TF Lite con flotantes de 16 Bits sería la siguiente:

```
TF_LITE_SIZE_MODEL_FILE_NAME_16BITS = "tf_lite_model_16bit.tflite"
TF_LITE_SIZE_MODEL_FILE_NAME = "tf_lite_model.tflite"

''' Convertimos el modelo de TF Lite a TF Lite con flotantes de 16 bits,
buscando como objetivo la reducción en espacio'''

tf_lite_converter =
tf.lite.TFLiteConverter.from_keras_model(model_for_pruning)
tf_lite_converter.optimizations = [tf.lite.Optimize.DEFAULT]
tf_lite_converter.target_spec.supported_types = [tf.float16]
tflite_model = tf_lite_converter.convert()

tflite_model_name = TF_LITE_SIZE_MODEL_FILE_NAME_16BITS
open(tflite_model_name, "wb").write(tflite_model)
```

Si se compara el tamaño del modelo de TF Lite original que se creó en un primer modelo, con el modelo de TF Lite con flotantes de 16 Bits, se tiene que el modelo de 16 Bits es aproximadamente un 50% menor que el modelo de TF Lite original.

Una vez que se tienen ambos modelos creados, y al igual que con los modelos originales, se puede comprobar su rendimiento utilizando los mismos datos de validación utilizados en los modelos previos.

Para ello primero hay que realizar unos pasos de configuración que en los modelos de Tensorflow originales no eran necesarios, como por ejemplo:

- Modificar las dimensiones de los tensores de entrada y salida para ajustarlas a las dimensiones de los datos de entrada y salida.

```
interpreter_16BITS.resize_tensor_input(input_details[0]['index'],
(125,256))
interpreter_16BITS.resize_tensor_input(output_details[0]['index'], (1,
1))
interpreter_16BITS.allocate_tensors()
```

- Darle los datos de entrada convertidos a flotantes y realizar una llamada al compilador de TF Lite:

```
interpreter_16BITS.set_tensor(input_details[0]['index'],
normed_test_data)
interpreter_16BITS.invoke()
```

- Finalmente, obtener las predicciones del modelo a través del tensor de salida del mismo:

```
tflite_model_predictions =
interpreter_16BITS.get_tensor(output_details[0]['index'])
```

En este punto se tienen ya las predicciones, en este caso, del modelo de TF Lite de 16 Bits. Se hace a continuación una comparativa del rendimiento de ambos modelos de TF Lite:

	Valor de R^2
Modelo original	0.92
Modelo original podado	0.91
Modelo TF Lite	0.91
Modelo TF Lite podado	0.91
Modelo TF Lite 16 Bits	0.91
Modelo TF Lite podado 16 Bits	0.91

Tabla 1: Comparación entre todos los modelos estudiados

Tal y como se puede ver, cuanto mayor es el nivel de reducción en el tamaño de los modelos, más bajo es el coeficiente de determinación. Sin embargo, la caída en dicho coeficiente es muy baja, prácticamente irrelevante teniendo en cuenta los datos con los que se está trabajando en este proyecto (por cantidad)

Sabiendo que el rendimiento de los modelos reducidos se mantiene estable con respecto a los modelos originales, se puede continuar con el estudio de su viabilidad en dispositivos de bajo coste. El siguiente paso en este proyecto es el de cargar estos modelos en distintos dispositivos y probar su funcionamiento en un entorno real.

CAPÍTULO 4: RASPBERRY PI

El primer dispositivo en el cual se van a cargar estos modelos es una Raspberry PI 4.

4.1 Raspberry PI 4

Raspberry Pi 4 es un ordenador de placa única (SBC, por sus siglas en inglés) desarrollado por la Fundación Raspberry Pi [7]. Se trata de la cuarta generación de la serie Raspberry Pi, y ha sido diseñada como una herramienta versátil y de bajo coste que promueve la educación en informática y la experimentación con proyectos electrónicos.

Algunas de las características principales de la Raspberry Pi 4 son:

- Procesador: La Raspberry Pi 4 cuenta con un procesador Broadcom BCM2711 de cuatro núcleos Cortex-A72, que funciona a una velocidad de hasta 1.5 GHz.
- Memoria RAM: Está disponible en diferentes variantes de memoria RAM, incluyendo opciones de 2 GB, 4 GB y 8 GB.
- Conectividad: La Raspberry Pi 4 ofrece diversas opciones de conectividad, incluyendo dos puertos USB 3.0, dos puertos USB 2.0, un conector Ethernet, una ranura para tarjeta microSD, una conexión HDMI para conectar a monitores o televisores, así como también puertos GPIO (General Purpose Input/Output) para la conexión de periféricos y proyectos electrónicos.
- Gráficos: La Raspberry Pi 4 incorpora un procesador gráfico VideoCore VI, que proporciona un rendimiento gráfico mejorado en comparación con las generaciones anteriores.
- Sistema Operativo: La Raspberry Pi 4 es compatible con una variedad de sistemas operativos, incluyendo Raspbian (una distribución basada en Linux especialmente diseñada para Raspberry Pi), Ubuntu, Windows 10 IoT Core y otros sistemas operativos basados en Linux. En este caso se cuenta con Debian 12 como sistema operativo.

En resumen, se puede decir que la Raspberry PI es un mini-ordenador, con capacidades mucho más limitadas que un PC de sobremesa estándar, pero a medio camino entre este y el siguiente dispositivo que se probará en este proyecto.

La carga de los modelos en la Raspberry PI es simple, ya que al contar con un SO integrado, cuenta también con un sistema de gestión de archivos similar al que se puede encontrar en Windows, por lo que la carga de los modelos se limita a introducirlos con un pendrive y arrastrarlos hasta la carpeta deseada dentro de la RPI.

4.2 Ejecución del modelo en Raspberry PI 4

Una vez se tienen dichos archivos cargados en la RPI, se debe crear un pequeño script de Python en el cual se abren cada uno de los modelos usando el intérprete de Tensoflow Lite (el mismo intérprete que se utilizó anteriormente para crear y probar los modelos en PC). Se puede ver dicho script en la Figura 10:

```python
import pandas as pd
import tensorflow as tf
import numpy as np

TF_LITE_MODEL = "./tf_lite_model_quant_noprune.tflite"

interpreter = tf.lite.Interpreter(model_path = TF_LITE_MODEL)

normed_test_data = pd.read_csv("./df_pruebas.csv")

normed_test_data = normed_test_data.drop("N_NO3", axis = 1)

normed_test_data = np.float32(normed_test_data)

nrows = normed_test_data.shape[0]
ncols = normed_test_data.shape[1]

input_details = interpreter.get_input_details()
output_details = interpreter.get_output_details()

interpreter.resize_tensor_input(input_details[0]['index'], normed_test_data.shape)
interpreter.resize_tensor_input(output_details[0]['index'], (1, 1))
interpreter.allocate_tensors()
input_details = interpreter.get_input_details()
output_details = interpreter.get_output_details()

interpreter.set_tensor(input_details[0]['index'], normed_test_data)
interpreter.invoke()
tflite_model_predictions = interpreter.get_tensor(output_details[0]['index'])

print("Forma de los resultados de la predicción", tflite_model_predictions.shape)

print(f"Las predicciones son {tflite_model_predictions}")

resultados = pd.DataFrame(tflite_model_predictions)

resultados.to_csv(f"resultados_tflite_quant_noprune.csv", index = False)

```

Figura 10: Script de prueba creado para RPI

Si bien en la captura se ve que se está trabajando en Visual Studio Code, para la creación del script se utilizó el entorno de desarrollo que viene integrado en la propia RPI, llamado Geanny.

En este script se realiza la carga del modelo de TF Lite, se le pasan los datos de entrada en forma de dataframe normalizado (tal y como se hizo con los modelos originales) y se llama al intérprete tomando como valores de entrada dicho dataframe.

El dataframe que se usa como entrada al modelo es el 20% restante de los datos originales que no se utilizaron en el entrenamiento y validación del modelo, por lo que su formato es exactamente el mismo que se tenía en el proceso de entrenamiento.

Una vez se llama al intérprete y se le pasan los datos de entrada, el modelo puede ejecutarse y arrojar predicciones.

Los resultados de dichas predicciones se guardan en distintos archivos CSV (un archivo por cada modelo testeado), y se pueden comparar con los resultados del modelo correspondiente ejecutado en el PC.

Para ello, por un lado se guardan las predicciones obtenidas en la RPI y por otro las predicciones obtenidas en el PC. Se crean dos dataframes y se almacenan en ellos todos los valores obtenidos. Para comparar ambos resultados, se opta por calcular las correlaciones que presentan ambos conjuntos de datos.

Por un lado se usará la correlación de Spearman y por otro la correlación de Pearson.

La correlación de Spearman y la correlación de Pearson son dos medidas estadísticas utilizadas para evaluar la relación entre dos conjuntos de datos. Aunque ambas medidas cuantifican la relación entre las variables, difieren en su enfoque y supuestos subyacentes.

La correlación de Spearman se basa en el orden de los datos, lo que la hace adecuada para capturar relaciones monotónicas sin importar si son lineales o no. Esta medida clasifica los valores de las variables en rangos y luego evalúa la concordancia del orden entre las dos variables. Es robusta ante la presencia de valores atípicos y no asume una distribución específica de los datos. Por lo tanto, la correlación de Spearman es útil cuando la relación

entre las variables no es necesariamente lineal y cuando los datos pueden contener valores atípicos.

Por otro lado, la correlación de Pearson se enfoca en la relación lineal entre los datos y asume que las variables siguen una distribución normal. Esta medida evalúa cómo se distribuyen los puntos de datos alrededor de una línea recta en un diagrama de dispersión. Es sensible a las desviaciones de la linealidad y puede verse afectada por valores atípicos y por violaciones de los supuestos de normalidad. La correlación de Pearson es más apropiada cuando se espera una relación lineal entre las variables y cuando los datos se ajustan a una distribución normal.

Se pueden calcular ambas correlaciones para los conjuntos de datos presentes gracias a la librearía de análisis estadístico de SciPy.

Los valores de ambas correlaciones para las predicciones del mismo modelo de TF Lite ejecutado en PC y ejecutado en RPI son las siguientes:

CORRELACIÓN SPEARMAN	**P SPEARMAN**	**CORRELACIÓN PEARSON**	**P PEARSON**
0.99	≈ 0.0	0.99	≈ 0.0

Tabla 2: Correlaciones entre datos PC y datos RPI

Se puede observar que, efectivamente, existe una fuerte correlación en ambos casos entre los valores que se obtienen ejecutando el modelo en un PC convencional y los valores que se obtienen ejecutando ese mismo modelo en una RPI, usando en ambos casos el mismo conjunto de datos.

Se puede decir de este modo que el hecho de ejecutar el modelo de TF Lite en una Raspberry PI no ha afectado en gran medida a su rendimiento, y la precisión que se obtuvo durante su desarrollo y validación se ha transferido sin problemas al nuevo entorno de ejecución.

De este modo, y a la vista de los resultados, se da por concluido el análisis sobre la viabilidad de la Raspberry PI 4 como entorno de producción para un modelo de Tensorflow Lite, viendo que efectivamente es posible implementar una red neuronal que funcione de forma satisfactoria y con unos resultados igualmente satisfactorios.

En el siguiente capítulo se realizará un análisis de dicha viabilidad en un dispositivo aún más limitado en memoria y capacidad de cómputo; un M5Stack FIRE basado en un SoC ESP32.

CAPÍTULO 5: ESP32

El siguiente dispositivo en el cual resulta interesante probar los diversos modelos de Tensorflow Lite es el ESP32.

5.1 M5Stack FIRE

El M5Stack FIRE [8] es un dispositivo electrónico programable que combina un microcontrolador ESP32, una pantalla a color, un teclado, altavoces, conectividad WiFi y Bluetooth, y una batería recargable, todo integrado en un solo dispositivo compacto. Está diseñado para facilitar el desarrollo de aplicaciones y proyectos de IoT (Internet de las cosas) y ofrece una plataforma de desarrollo versátil y de fácil uso. Se puede ver dicho dispositivo en la Figura 11.

Figura 11: Kit M5Stack FIRE usado en este proyecto

En términos formales, el M5Stack FIRE se puede describir como un microcontrolador basado en el ESP32, que es un sistema en chip (SoC) de bajo consumo energético con procesador de doble núcleo, conectividad WiFi y Bluetooth integrada, y capacidades de procesamiento y comunicación avanzadas. El M5Stack FIRE también incorpora una pantalla TFT a color de alta resolución, un teclado con botones de función programables, altavoces y una batería recargable para la alimentación portátil.

Además, el M5Stack FIRE es compatible con el entorno de desarrollo de Arduino, lo que facilita la programación y el desarrollo de software para el dispositivo. Aunque en este

caso, y con el objetivo de unificar el desarrollo en una única plataforma, se utilizará Visual Studio Code junto con Platform IO.

Platform IO es un framework de desarrollo de código abierto que se puede integrar en Visual Studio Code a través de la instalación de un plugin. Ofrece un entorno orientado al desarrollo de proyecto IoT y permite la interacción con diversos tipos de placas; desde Arduino o Raspberry Pi, hasta microcontroladores basados en ESP32 o ESP8266.

En este caso, se hará uso de dicho framework para adaptar y cargar el modelo de TF Lite hasta el dispositivo FIRE.

5.2 Conversión del modelo TF Lite. De Python a C++

El primer paso en el proceso de carga y utilización del modelo de TF Lite en el dispositivo FIRE es el de convertir dicho modelo a un lenguaje en el cual se pueda procesar dentro de dicho dispositivo. El lenguaje escogido es C++, que es en el cual se basa el desarrollo de Arduino (y del cual se heredarán algunas características)

Para realizar dicha conversión se ha usado una plantilla facilitada por el usuario Atomic14 en Github [9] que incluye varios procesos:

- En primer lugar, un ejemplo de cómo crear, entrenar y convertir un modelo de Keras a un modelo de TF Lite. Este punto no resulta de interés ya que dicho modelo ya ha sido creado y convertido con éxito en fases anteriores del desarrollo.
- Un conversor que permite transferir el modelo de TF Lite a C++ con solo un par de líneas de código.

Dicha conversión se hace ejecutando el siguiente comando en una terminal de Bash dentro del proyecto de Platform IO, en el cual ya se ha copiado previamente el modelo de TF Lite que se desea convertir:

```
xxd -i converted_model.tflite > firmware/src/model_data.cc
```

Este comando permite convertir directamente el modelo en un modelo de C++, aunque aún así se requieren un par de pasos extra en la configuración del proyecto, ya que la

lógica que gobierna los datos de entrada y salida de la red neuronal debe ser ajustada manualmente para cada caso.

En la Figura 12 se puede ver un fragmento del código que ha sido necesario implementar para poder operar con la red neuronal convertida a C++:

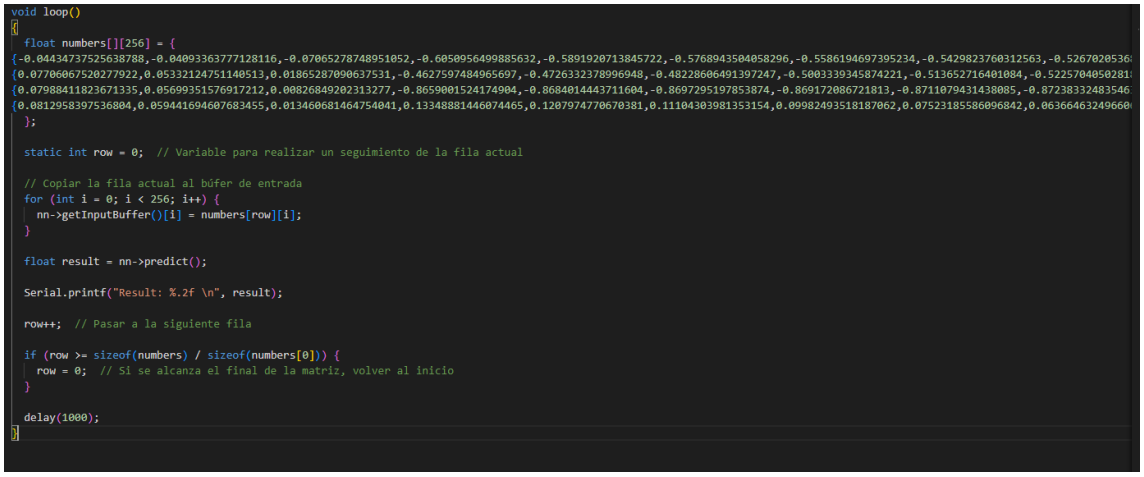


Figura 12: Código C++ relativo a la red neuronal en el ESP32

Siguiendo el código de inicio a fin, se tienen las siguientes características:

- Inicialmente se define una matriz de datos de entrada. Esta matriz tiene tres filas y 256 columnas. Lo que se ha buscado con esto es que dicha matriz tenga la misma forma que los datasets que se han usado previamente para entrenar y validar el modelo, que, a su vez, será la misma forma en la cual llegarán los datos del sensor en un hipotético proceso de predicción en campo.
- Se definen las entradas al buffer de la red neuronal. Tal y como se vio en capítulos anteriores, la entrada a la red neuronal consta de 256 valores distintos, que se corresponden con los 256 canales de medición del sensor. En este código, se ajusta la entrada para que tenga esa misma forma, y se le pasan cada una de las filas de la matriz en cada una de las iteraciones del bucle.
- Se llama a la red neuronal y se extrae la predicción que arroja para los datos insertados previamente. Dichos datos se imprimen dentro de la terminal de Visual

Code Studio a través del puerto serial al que se encuentra conectado el dispositivo FIRE.

- Se verifica si se ha alcanzado o no el final de la matriz de los datos de entrada. Si no se ha alcanzado, se procede con la siguiente fila, y si sí se ha alcanzado se repite el proceso desde el inicio.

En líneas generales, el procedimiento es similar al que se pudo ver en capítulos anteriores, con la diferencia de que en este caso se está trabajando con lenguaje C++ que es de más bajo nivel que Python, por lo que algunas tareas que en Python son triviales, en C++ es necesario explicitarlas con mayor detalle.

Además de este código, que es el que se encarga de procesar los datos de entrada y llamar a la red neuronal, se tiene el propio código de la red neuronal, que se puede ver en la Figura 13.

```
NeuralNetwork::NeuralNetwork()
{
    error_reporter = new tflite::MicroErrorReporter();

    model = tflite::GetModel(converted_model_tflite);
    if (model->version() != TFLITE_SCHEMA_VERSION)
    {
        TF_LITE_REPORT_ERROR(error_reporter, "Model provided is schema version %d not equal to supported version %d.",
                             model->version(), TFLITE_SCHEMA_VERSION);
        return;
    }
    // This pulls in the operators implementations we need
    resolver = new tflite::MicroMutableOpResolver<10>();
    resolver->AddFullyConnected();
    resolver->AddMul();
    resolver->AddAdd();
    resolver->AddLogistic();
    resolver->AddReshape();
    resolver->AddQuantize();
    resolver->AddDequantize();

    tensor_arena = (uint8_t *)malloc(kArenaSize);
    if (!tensor_arena)
    {
        TF_LITE_REPORT_ERROR(error_reporter, "Could not allocate arena");
        return;
    }

    // Build an interpreter to run the model with.
    interpreter = new tflite::MicroInterpreter(
        model, *resolver, tensor_arena, kArenaSize, error_reporter);

    // Allocate memory from the tensor_arena for the model's tensors.
    TfLiteStatus allocate_status = interpreter->AllocateTensors();
    if (allocate_status != kTfLiteOk)
    {
        TF_LITE_REPORT_ERROR(error_reporter, "AllocateTensors() failed");
        return;
    }

    size_t used_bytes = interpreter->arena_used_bytes();
    TF_LITE_REPORT_ERROR(error_reporter, "Used bytes %d\n", used_bytes);

    // Obtain pointers to the model's input and output tensors.
    input = interpreter->input(0);
    output = interpreter->output(0);
}
```

Figura 13: Red neuronal en C++

Esta porción de código se genera automáticamente a partir del modelo de TF Lite que se haya decidido convertir, por lo que no requiere ningún tipo de modificación. Se trata de

una traducción del modelo de Tensorflow original en Python, a un modelo de TF Lite en C++.

Por último, se pueden ver los resultados que arroja esta red neuronal una vez se compila y carga el código en el dispositivo. El proceso de compilado y carga se realiza de forma sencilla gracias al framework con el cual se está trabajando, ya que permite una interacción sencilla e intuitiva entre el entorno de desarrollo y el dispositivo FIRE. Para realizar este proceso es tan sencillo como hacer click sobre el botón Upload, encontrado en la cinta inferior de Visual Studio Code (señalado con un círculo en la Figura 14):

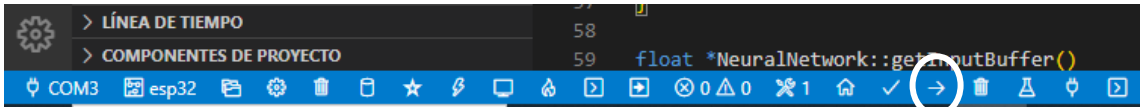


Figura 14: Compilación y carga de la red en el ESP32

Se puede ver también en la parte izquierda de la cinta que se está haciendo referencia a un puerto COM3. Este puerto es el puerto USB en el cual se tiene conectado el dispositivo al PC, por lo que se debe seleccionar dicho puerto antes de realizar la carga. Se debe seleccionar también el modelo de placa que se está usando, tal y como se puede ver a la derecha del selector de puertos.

En este caso, se trabaja con una placa basada en un controlador ESP32 genérico. Hay diversas opciones para modelos específicos, pero en este caso al trabajar con un dispositivo basado en un ESP32, con seleccionar una placa ESP32 genérica es suficiente.

Una vez se tienen todos los parámetros de comunicaciones ajustados se realiza el compilado y la carga del código dentro del dispositivo. Dicho proceso toma unos segundos, y cuando finaliza se muestra el siguiente mensaje en la terminal, indicando que todo el proceso se ha cumplido de forma satisfactoria, tal y como se puede ver en la Figura 15:

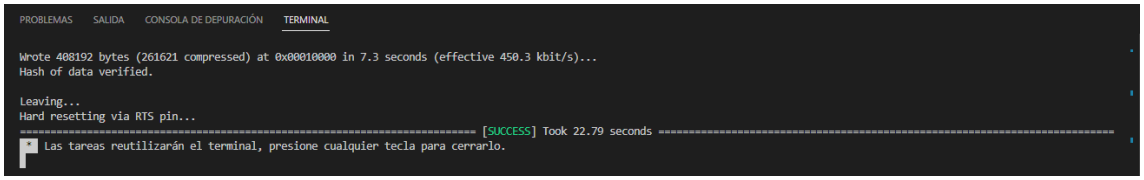


Figura 15: Proceso de compilado y carga completado

5.3 Extracción de resultados y comparación

Una vez se ha cargado el programa en el dispositivo, para monitorizar el comportamiento del mismo se abre una instancia del monitor serial.

El monitor serial se utiliza en distintos IDEs (Integrated Development Environment) para leer la comunicación que existe entre la placa conectada y el PC, y en este caso en el monitor serial se mostrarán los valores que ha inferido la red neuronal cargada en el dispositivo a partir de los valores de entrada que se le especificaron en la matriz inicial.

Los valores que se pueden ver en el monitor serial son los mostrados en la Figura 16:

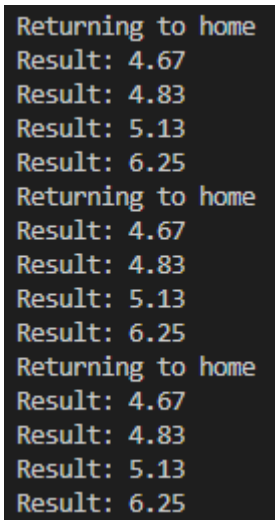


Figura 16: Datos retornados por el monitor serial del ESP32

Tal y como se puede ver, se tienen los mismos cuatro valores repetidos múltiples veces, con un texto que indica que se ha llegado al final de la matriz de los datos de entrada y que se retorna al inicio.

Los valores se repiten porque, en esta primera prueba, se le han insertado únicamente cuatro filas de datos a la matriz de entrada. Tampoco se pueden cargar muchas filas debido a que la memoria del dispositivo es limitada, y en las pruebas en las cuales se le ha tratado de cargar 20-30 filas, ha arrojado error de desbordamiento de memoria.

Esto no supone un problema de cara a la operación del sistema en un hipotético entorno de producción, ya que el sensor va a mandar al dispositivo las filas de datos de una en una. Por cada fila entrante, el dispositivo realizará una inferencia, y para la siguiente fila

realizará otra, y así sucesivamente, pero teniendo como entrada siempre una única fila de datos.

Sabiendo esto, se puede verificar si la inferencia realizada por el modelo cargado en la Raspberry PI es similar o igual a la inferencia realizada por el modelo cargado en el ESP32. Para comprobar esto, tomamos la primera fila de datos en el archivo de pruebas que se extrajo del dataset de entrenamiento (como se vio en el primer capítulo, se usó un 80% de los datos totales para entrenar, y el 20% restante se reservó para realizar este tipo de pruebas)

Si se cargan las primeras filas de datos en la Raspberry Pi se obtiene los siguientes valores (de concentración de nitratos inferida por el modelo):

	RESULTADOS TF LITE EN PC	**RESULTADOS RPI**	**RESULTADOS ESP32**
FILA 1	4.38	4.56	4.67
FILA 2	4.82	4.80	4.83
FILA 3	5.46	5.43	5.13
FILA 4	5.05	5.09	5.67
FILA 5	5.05	4.87	5.28

Tabla 3: Comparación de valores de concentración de nitratos (mg/l) entre los distintos entornos de ejecución

Tal y como se puede ver, la red que se ejecuta en la Raspberry PI aporta mayor nivel de precisión y una cantidad mayor de bits en los resultados. Por otro lado, la red alojada en el ESP32, al ser mucho más limitada en términos de memoria y capacidad de computación, retorna valores más simples y con peor precisión.

Si se comparan los resultados de ambas redes con la red original ejecutada en el PC, se ve que claramente los resultados aportados por la Raspberry PI son más fieles y cercanos

a los aportados por el ESP32. De nuevo, se debe tener en cuenta la capacidad de cómputo entre ambos dispositivos y en cómo afecta dicha capacidad al comportamiento general de la red alojada en ambos sistemas.

En este caso, no se realiza una medición de las correlaciones entre todos los resultados debido a que, como se comentó anteriormente, resulta complicado realizar la carga de todo el dataset de testeo por culpa de las limitaciones de memoria que presenta el dispositivo.

A la vista de estos resultados, se da cabida a varias conclusiones y posibles líneas futuras de mejora e investigación, que se comentarán el siguiente capítulo.

CAPÍTULO 6: CONCLUSIONES Y FUTURAS LÍNEAS DE TRABAJO

A lo largo del este documento se ha hecho un estudio de las posibilidades que ofrece Tensorflow para implementar sus modelos en dispositivos de bajo coste.

Se creó una red neuronal de la forma clásica en Keras y se la entrenó con los datos disponibles, tras lo cual se transformó esta red neuronal en un modelo de Tensorflow Lite siguiendo distintos procedimientos y optimizaciones (podado y cuantización el flotantes de 16 Bits)

Estos modelos de TF Lite se verificaron en el entorno de desarrolló y se comprobó que su funcionamiento y precisión eran satisfactorios, tras lo cual se decidió implementarlos en dos dispositivos de bajo coste y poder computacional. El primero de ellos, la Raspberry PI, cuenta con una capacidad de cómputo intermedia y una memoria limitada (aunque extensible mediante tarjetas SD), y se comprobó que todos y cada uno de los modelos TF Lite generados previamente funcionaban de forma satisfactoria una vez adaptados.

El segundo dispositivo, un M5Stack FIRE basado en un SoC ESP32, se encuentra un paso por debajo de la Raspberry. No cuenta con un sistema operativo ni con una interfaz visual, por lo que la comunicación se realiza mediante monitor serial a través del PC. Para este dispositivo fue necesario convertir la red a C++ y cargarla a través de un framework específico.

En general, se puede afirmar que los resultados son satisfactorios. En el caso de la Raspberry, la implementación fue sencilla y directa, con alguna limitación debido a su sistema operativo (Debian 12), pero en el caso del ESP32 la implementación fue mucho más compleja. La falta de interfaz, la conversión de la red a otro lenguaje, la poca información disponible en Internet, y la complejidad propia de un sistema ESP32, hicieron que su implementación fuese mucho más laboriosa que la de la Raspberry PI, por lo que, independientemente de los resultados arrojados (que son igualmente satisfactorios), el hecho de haber conseguido implementar todo el sistema en dicho dispositivo se puede considerar un gran comienzo de cara a futuras investigaciones o líneas de trabajo.

Sobre las posibles futuras líneas de trabajo en este ámbito, sería muy interesante profundizar más aún en la implementación en el ESP32. Es un dispositivo que, pese a su limitada potencia y memoria, ofrece multitud de opciones de desarrollo de cara a proyectos IoT.

Resultaría muy interesante también profundizar en la conversión de la red a lenguaje C++, y en la optimización de dicha red desde el origen sabiendo que el objetivo será implementarla en un dispositivo de este estilo.

En líneas generales, este proyecto abre un abanico de posibilidades referentes al uso de sistemas de Deep Learning y al desarrollo del Edge Computing en dispositivos de bajo coste y capacidad de computación.

REFERENCIAS

[1] Causse, J., Thomas, O., Jung, A. V., & Thomas, M. (2017). Direct DOC and nitrate determination in water using dual pathlength and second derivative UV spectrophotometry. *Water Research, 108*, 312-319.

[2] Crumpton, W. G., Isenhart, T. M., & Mitchell, P. D. (1992). Nitrate and organic N analysis with 2nd-derivative spectroscopy. *Limnology and Oceanography, 37*, 907-913.

[3] Deeplizard. (s.f.). Understanding Overfitting in Neural Networks. *Deeplizard*. Recuperado de [https://deeplizard.com/learn/video/DEMmkFC6IGM]

[4] Investopedia. (2023). R-Squared: Definition, Calculation Formula, Uses, and Limitations. *Investopedia*. Recuperado de [https://www.investopedia.com/terms/r/r-squared.asp]

[5] TowardsDataScience. (2020). Pruning Neural Networks. TowardsDataScience. Recuperado de [https://towardsdatascience.com/pruning-neural-networks-1bb3ab5791f9]

[6] TensorFlow. (s.f.). TensorFlow Lite. TensorFlow. Recuperado de [https://www.tensorflow.org/lite/guide?hl=es-419]

[7] Raspberry Pi Foundation. (s.f.). Recuperado de [https://www.raspberrypi.org]

[8] M5Stack. (s.f.). M5Stack FIRE official documentation. *M5Stack Documentation*. Recuperado de [https://docs.m5stack.com/en/core/fire]

[9] TensorFlow-Lite and Platform.io. (2021). tensorflow-lite-esp32. *GitHub*. Recuperado de [https://github.com/atomic14/tensorflow-lite-esp32]