MICROPROYECTO DATASETS

Cálculo de la corriente de excitación de un motor sincrónico

Melina Paula Gonzalez Rubiño- 2236748
melina.gonzalez@uao.edu.co

Juan Camilo Caicedo Cortes - 2190732
juan_c.caicedo@uao.edu.co

Juan Camilo León - 2190612
juan_camilo.leon@uao.edu.co

Redes neuronales artificiales y deep learning
Universidad Autónoma de Occidente
Facultad de Ingeniería
Septiembre - 2023

Resumen: El siguiente reporte describe el proceso de implementación del entrenamiento de una red MLP superficial en Tensorflow2-Keras que permite resolver el problema de regresión definido de acuerdo a nuestro dataset seleccionado, el cual posteriormente es emulado en Arduino como plataforma de implementación final.

Palabras claves: Entrenamiento, dataset, problema, red MLP superficial, Tensorflow2-Keras.

***Abstract*:** The following report describes the implementation process of the training of a shallow MLP network in Tensorflow2-Keras that allows to solve the regression problem defined according to our selected dataset, which is then emulated in Arduino as the final implementation platform.

Keywords: *Training; dataset; problem, shallow MLP network, Tensorflow2-Keras.*

I. INTRODUCCIÓN

En el campo del aprendizaje automático y la inteligencia artificial, el poder de las redes neuronales artificiales para resolver una amplia variedad de problemas es indiscutible. Las Redes MLP han demostrado ser una herramienta valiosa especialmente cuando se trata de problemas de regresión, donde el objetivo es predecir valores numéricos a partir de un conjunto de características. En este informe, seguiremos a detalle el proceso de entrenamiento de una Red MLP superficial utilizando TensorFlow2-Keras, una de las herramientas o bibliotecas más versátiles en el ámbito del aprendizaje profundo.

Nuestro objetivo principal consiste en abordar un problema de regresión específico, seleccionado cuidadosamente (dataset), y acondicionar a la red para que sea capaz de hacer predicciones precisas y útiles. Para ello, comenzaremos por comprender los fundamentos teóricos que tienen gran relevancia en el aprendizaje profundo y la regresión. Luego, presentaremos el dataset elegido, en donde describiremos el problema a solucionar, que servirá como base para nuestro proyecto, y explicaremos cómo se prepara y se divide en conjuntos de entrenamiento y prueba.

A lo largo de este informe, examinaremos los detalles de la arquitectura de la Red MLP, destacando la importancia de las capas ocultas, las funciones de activación en cuanto a la capacidad de la red para aprender relaciones no lineales en los datos y los aspectos claves del proceso de entrenamiento, como la elección de la función de pérdida adecuada y la aplicación de la retropropagación para ajustar los pesos y sesgos del modelo.

Además, se abordará la evaluación del rendimiento del modelo a través de métodos de regresión pertinentes, lo que nos permitirá medir su capacidad para hacer predicciones precisas en datos no vistos. También consideraremos la importancia de la experimentación y el ajuste de hiperparámetros para mejorar el rendimiento del modelo.

II. MARCO TEÓRICO

A. Regresión.

La regresión es una técnica de aprendizaje supervisado utilizada para predecir un valor numérico (variable continua) a partir de un conjunto de variables de entrada.

B. Redes neuronales artificiales (ANNs)

Las redes neuronales artificiales son modelos de aprendizaje automático inspirados en la estructura y funcionamiento del cerebro humano. Están compuestas

por capas de neuronas interconectadas, que procesan la información mediante la aplicación de funciones de activación.

C. Multilayer Perceptron (MLP)..

Un MLP es un tipo de red neuronal artificial que consta de al menos tres capas de entrada, una o varias capas ocultas y una capa de salida.
Las capas ocultas están formadas por neuronas que aplican funciones de activación a las entradas ponderadas para aprender representaciones no lineales de los datos.

D. Capas de neuronas.

Cada capa de neuronas en una MLP consta de un conjunto de neuronas o nodos que reciben entradas aplican una función de activación y producen salidas.

- La capa de entrada recibe las características del conjunto de datos (dataset).
- Las capas ocultas realizan transformaciones no lineales a las entradas.
- La capa de salida produce la predicción final.

E. Funciones de activación.

Las funciones de activación introducen no linealidades en la red y permiten que esta aprenda relaciones complejas en los datos. Algunas de estas funciones son: ReLU, Lineal, Tangente sigmoidal, Escalón, entre otras.

F. Función de pérdida.

La función de pérdida mide la discrepancia entre las predicciones del modelo y los valores reales en el conjunto de entrenamiento. En problemas de regresión, una función de pérdida común es el error cuadrático medio (MSE), que penaliza las diferencias cuadráticas entre las predicciones y los valores reales.

G. Optimizadores.

Un optimizador se encarga de optimizar (actualizar) los parámetros de la red para reducir el error cometido por la misma. Existe variedad de optimizadores como por ejemplo: Gradiente Descendiente Estocástico (SGD), Adam, Adamax, Adadelta, Adagrad, RSMprop, Nadam, entre otros.

H. Backpropagation.

Es el algoritmo utilizado para entrenar una red neuronal. El backpropagation o retropropagación, ajusta los pesos y sesgos de las neuronas en función de la derivada de la función de pérdida con respecto a los parámetros del modelo.

I. Dataset

Se divide en un conjunto de entrenamiento y un conjunto de pruebas. El dataset o conjuntos de datos se utiliza para ajustar los pesos de la red, mientras que el conjunto de prueba se utiliza para evaluar el rendimiento del modelo.

J. Hiperparámetros.

Los hiperparámetros son configuraciones del modelo que se aprenden durante el entrenamiento, como el número de capas y neuronas, la tasa de aprendizaje y la función de activación. Ajustar correctamente los hiperparámetros es crucial para obtener un óptimo rendimiento del modelo.

K. Evaluación del modelo.

Las métricas de evaluación, como el MSE y el R^2, se utilizan para medir el rendimiento del modelo en el conjunto de prueba. También suelen utilizarse visualizaciones, tales como gráficos de dispersión, para comparar las predicciones y los valores reales.

L. Motor síncrono.

Es un tipo de motor eléctrico en el que la velocidad de rotación del rotor está sincronizada con la frecuencia de la corriente alterna suministrada al estator, lo que significa que, la velocidad de giro del rotor es constante y se mantiene en sincronía con la frecuencia de la corriente alterna suministrada.

III. DESCRIPCIÓN DEL PROBLEMA

En el mundo de la ingeniería eléctrica y la automatización industrial, los motores síncronos (SM) son ampliamente utilizados debido a su capacidad para operar con una velocidad constante y mantener una relación fija entre la frecuencia de la corriente alterna suministrada y la velocidad de rotación del motor. Esto los hace esenciales en aplicaciones donde se requiere una precisión extrema, como en la generación de energía, la industria manufacturera y el transporte ferroviario.
Sin embargo, para garantizar el funcionamiento óptimo y la eficiencia de estos motores síncronos, es fundamental tener una comprensión detallada de su rendimiento eléctrico y mecánico. Uno de los parámetros críticos a monitorear y controlar en un motor síncrono es la corriente de excitación, que determina la intensidad del campo magnético interno y, por lo tanto, influye en su capacidad para mantener una velocidad constante.
Para abordar esta importante problemática, se ha recopilado un conjunto de datos de motores síncronos a partir de experimentos en situaciones del mundo real. Este conjunto de datos es una valiosa fuente de

información que puede utilizarse para comprender mejor la relación entre la corriente de excitación y otros factores, como la carga, la temperatura y el voltaje suministrado. El objetivo central de este proyecto es desarrollar modelos sólidos y precisos que permitan estimar la corriente de excitación de los motores síncronos en función de estas variables.
La capacidad de estimar con precisión la corriente de excitación de un SM es esencial para el monitoreo en tiempo real de su rendimiento y para la implementación de estrategias de control avanzadas. Además, esta investigación puede tener un impacto significativo en la eficiencia energética y la confiabilidad de los sistemas eléctricos y mecánicos en los que se utilizan estos motores.
En este contexto, el proyecto se centrará en la construcción de modelos de aprendizaje automático y técnicas de análisis de datos que permitan abordar la tarea de estimar la corriente de excitación de los motores síncronos. La aplicación de estos modelos sólidos a datos experimentales reales representa un desafío apasionante que puede tener un impacto directo en la mejora de la eficiencia y la confiabilidad de sistemas industriales y de generación de energía que dependen de los motores síncronos.

IV. PLANTEAMIENTO DE LA SOLUCIÓN.

La solución será implementada en Python utilizando diversas librerías que permiten trabajar los datos a fin de modelar las redes neuronales pertinentes y sus correspondientes características y parámetros. Para esta problemática en particular debía plantearse una solución comparativa utilizando tres diferentes optimizadores y posteriormente escoger aquella con un mejor resultado.

Partiendo de la descripción de nuestro dataset, tenemos que este consta de 5 atributos, los cuales poseen 557 datos cada uno. Los atributos son los siguientes:

- **Iy:** Corriente de carga.
- **PF**: Factor de potencia.
- **E:** Error, que proviene de la diferencia entre el factor de potencia deseado y el real.
- **DIf:** Variación de la corriente de excitación de la máquina síncrona.
- **If:** Corriente de excitación de la máquina síncrona.

Con estos datos se trabaja posteriormente normalizandolos (para que no se saturen las funciones de activación) y realizando las operaciones necesarias para trabajar con el modelo de red neuronal propuesto.

De acuerdo a lo anteriormente mencionado, se escogieron tres optimizadores para el entrenamiento de nuestro modelo, con el fin de compararlos y establecer cuál es el óptimo para la solución del problema. A continuación, se comparan los resultados de cada uno de los optimizadores.

Cabe resaltar que para los tres se utiliza el mismo código, lo que cambia en ellos es la implementación del optimizador y el número de épocas para el entrenamiento del modelo.

1. **Optimizador Adam (Adaptative Moment Estimation).**

- Librerías: Se incluyen las librerías para obtener el mejor desempeño en la implementación del código y poder trabajar con las herramientas necesarias (esto es dado que python no tiene "todo" embebido, sólo lo esencial para trabajar de manera sencilla. Si se desea trabajar con algoritmos más complejos, deben incluirse las herramientas que hacen esto posible, como sucede en todos los lenguajes de programación).

```
[ ]  import datetime, os
     import numpy as np
     import pandas as pd
     import tensorflow as tf
     import seaborn as sn
     import matplotlib.pyplot as plt

     from tensorflow.keras import layers
     from tensorflow.keras.models import Sequential
     from tensorflow.keras.layers import Dense
     from tensorflow.keras.utils import plot_model
     from tensorflow.keras.callbacks import TensorBoard
     from sklearn import preprocessing
     from sklearn import linear_model
     from sklearn.model_selection import train_test_split
```

Fig 1. Librerías a utilizar

- **Importación y lectura del dataset:** Se importa el dataset (en este caso el dataset se tienen en formato .csv, pero podría tenerse en otros formatos de base de datos, como por ejemplo .xlsx) y se muestran los primeros 10 datos de cada atributo de los 557 que posee cada uno a fin de conocer cómo están conformados los mismos.

```
[ ] data = pd.read_csv('/content/1synchronous-machine.csv')
    print(data.shape)
    data.head(10)
```

```
(557, 5)
```

	Iy	PF	e	dIf	If
0	3.0	0.66	0.34	0.383	1.563
1	3.0	0.68	0.32	0.372	1.552
2	3.0	0.70	0.30	0.360	1.540
3	3.0	0.72	0.28	0.338	1.518
4	3.0	0.74	0.26	0.317	1.497
5	3.0	0.76	0.24	0.301	1.481
6	3.0	0.78	0.22	0.290	1.470
7	3.0	0.80	0.20	0.280	1.460
8	3.0	0.82	0.18	0.250	1.430
9	3.0	0.84	0.16	0.221	1.401

Fig 2. Lectura del dataset

- **Descripción de los datos:** Se identifica la dispersión que existe entre los datos, los percentiles, la media, el mínimo y el máximo de cada uno de los atributos.

```
data.describe()
```

	Iy	PF	e	dIf	If
count	557.000000	557.000000	557.000000	557.000000	557.000000
mean	4.499820	0.825296	0.174704	0.350659	1.530659
std	0.896024	0.103925	0.103925	0.180566	0.180566
min	3.000000	0.650000	0.000000	0.037000	1.217000
25%	3.700000	0.740000	0.080000	0.189000	1.369000
50%	4.500000	0.820000	0.180000	0.345000	1.525000
75%	5.300000	0.920000	0.260000	0.486000	1.666000
max	6.000000	1.000000	0.350000	0.769000	1.949000

Fig 3. Descripción de los datos

- **Atributo de interés:** Se define el atributo de interés (salida), en nuestro caso es la corriente de excitación.

```
data['If']
```

```
0      1.563
1      1.552
2      1.540
3      1.518
4      1.497
       ...
552    1.322
553    1.331
554    1.340
555    1.340
556    1.340
Name: If, Length: 557, dtype: float64
```

Fig 4. Definición del atributo de interés

- **Normalización de los datos:** Esta se realiza con el objetivo de ajustar las escalas de las variables, haciendo que todas se encuentren en una magnitud similar o se encuentren dentro del mismo rango. En este caso el rango adoptado es entre -1 y 1. Esto se hace, como se dijo anteriormente, con el fin de que las funciones de activación no se saturen, dado que varias de ellas trabajan en este rango (o entre 0 y 1). Este es el caso de las funciones escalón, tangente sigmoidal y ReLU.

```
def normalizar(x,xmax,xmin,ymax,ymin):
    m = (ymax-ymin)/(xmax-xmin)
    b = ymin - m*xmin
    y = m*x + b

    return y

max = np.max(data).values
min = np.min(data).values

dataNorm = normalizar(data,max,min,1,-1)
dataNorm.head(10)
#print(data)
```

Fig 5. Función de normalización.

A continuación se muestran los diez primeros datos, nuevamente, pero en esta ocasión ya normalizados, para verificar que este proceso haya sido exitoso.

	Iy	PF	e	dIf	If
0	-1.0	-0.942857	0.942857	-0.054645	-0.054645
1	-1.0	-0.828571	0.828571	-0.084699	-0.084699
2	-1.0	-0.714286	0.714286	-0.117486	-0.117486
3	-1.0	-0.600000	0.600000	-0.177596	-0.177596
4	-1.0	-0.485714	0.485714	-0.234973	-0.234973
5	-1.0	-0.371429	0.371429	-0.278689	-0.278689
6	-1.0	-0.257143	0.257143	-0.308743	-0.308743
7	-1.0	-0.142857	0.142857	-0.336066	-0.336066
8	-1.0	-0.028571	0.028571	-0.418033	-0.418033
9	-1.0	0.085714	-0.085714	-0.497268	-0.497268

Fig 6. Datos normalizados.

- **Evaluación de correlación entre los atributos:** Se calcula el coeficiente de correlación que existe entre los atributos o variables. Estos

valores se visualizan luego mediante gráficos de dispersión e histogramas, que permiten, de una manera más "cómoda" evaluar la correlación de los datos a partir de ver la ubicación de un atributo respecto de otro.

```
corr=dataNorm.corr()
print(corr)
```

```
            Iy        PF         e        dIf        If
Iy    1.000000 -0.041574  0.041574  0.424945  0.424945
PF   -0.041574  1.000000 -1.000000 -0.861013 -0.861013
e     0.041574 -1.000000  1.000000  0.861013  0.861013
dIf   0.424945 -0.861013  0.861013  1.000000  1.000000
If    0.424945 -0.861013  0.861013  1.000000  1.000000
```

Fig 7. Coeficientes de correlación

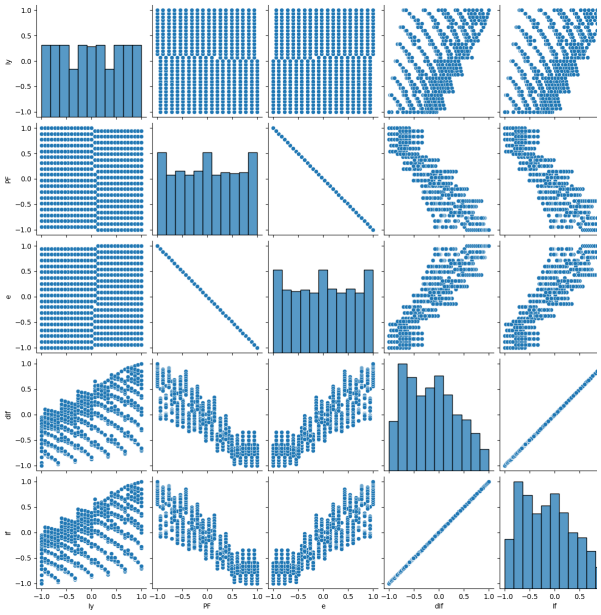

Fig 8. Histogramas, mapas de dispersión y de correlación entre las variables.

- **División de los datos en conjuntos de entrenamiento y prueba:** En este paso se dividieron el 80% de los datos para el conjunto de entrenamiento y el 20% para el conjunto de prueba. Se hace con el fin de poder testear a la red con otros datos que no sean los de entrenamiento y con ello verificar si "se aprendió" los datos y quedó sobreentrenada (overfitting) o si fue entrenada de manera correcta.

```
seed = 42

x_train,x_test,y_train,y_test = train_test_split(inputData, target, test_size=0.2, random_state=seed)

print(x_train.shape)
print(y_train.shape)
print(x_test.shape)
print(y_test.shape)
```

```
(445, 4)
(445,)
(112, 4)
(112,)
```

Fig 9. División de los datos en conjuntos de datos.

- **Arquitectura del modelo y optimizadores:** Se establece la arquitectura del modelo, en donde se define la función de pérdida (loss), que se calculará por medio del error medio al cuadrado, y el parámetro de aprendizaje (learning rate) con un valor de 3e-3 en este caso. Para la primera capa, la capa oculta, se establece una capa densa de 8 neuronas con su función de activación (ReLu), tiene cuatro entradas (los cuatro atributos del modelo) y luego una capa de salida con una única neurona (la corriente de excitación) con una función de activación lineal. Los optimizadores son:

- **Optimizador Adam:** Para este optimizador se establece un número de épocas (epochs) igual a 250.

Crear la arquitectura del modelo

```
[ ]  input_dim = x_train.shape[1]
     print(input_dim)
     num_clases = 1
     lr = 3e-3
     loss = 'mse'#'mean_squared_error'
```

```
[ ]  def model():
       model = Sequential()
       model.add(Dense(8, input_dim = input_dim, activation='relu'))

       model.add(Dense(num_clases, activation='linear'))

       model.summary()

       opt = tf.keras.optimizers.Adam(learning_rate=lr)

       model.compile(loss = loss, optimizer = opt)

       return model
```

Fig 10. Arquitectura del modelo con optimizador Adam

- **Optimizador RMSprop:** Para este optimizador se estableció un número de épocas igual que al optimizador anterior (Optimizador Adam).

Crear la arquitectura del modelo

```
[ ]  input_dim = x_train.shape[1]

     num_clases = 1
     lr = 3e-3
     loss = 'mse'#'mean_squared_error'
```

```
def model():
  model = Sequential()
  model.add(Dense(8, input_dim = input_dim, activation='relu'))
  model.add(Dense(num_clases, activation='linear'))

  model.summary()

  opt = tf.keras.optimizers.RMSprop(learning_rate=lr)

  model.compile(loss = loss, optimizer = opt)

  return model
```

Fig 11. Arquitectura del modelo con optimizador RMSprop.

- **Optimizador SGD:** Para este optimizador se eligió un número de épocas mucho mayor (500) que al de los dos optimizadores anteriores.

Crear la arquitectura del modelo

```
[ ]  input_dim = x_train.shape[1]

     num_clases = 1
     lr = 3e-3
     loss = 'mse'#'mean_squared_error'
```

```
[ ]  def model():
       model = Sequential()
       model.add(Dense(8, input_dim = input_dim, activation='relu'))
       model.add(Dense(num_clases, activation='linear'))

       model.summary()

       opt = tf.keras.optimizers.SGD(learning_rate=lr)

       model.compile(loss = loss, optimizer = opt)

       return model
```

Fig 12. Arquitectura del modelo con optimizador SGD.

V. RESULTADOS.

Dado lo anterior y teniendo la arquitectura de los modelos con cada uno de los optimizadores, se obtienen los siguientes resultados, los cuales se pueden visualizar en las gráficas de épocas vs función de pérdida (epochs vs loss).

- **Optimizador Adam.**

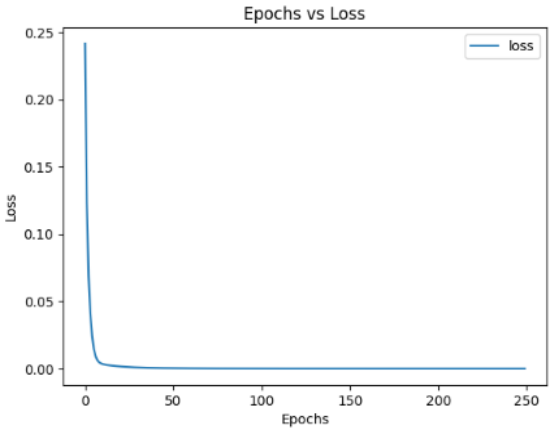

Fig 13. Gráfica epochs vs loss optimizador Adam

Puede verse que la función de pérdida decae rápidamente y se hace asintótica a cero para una cantidad de épocas menor a 50.

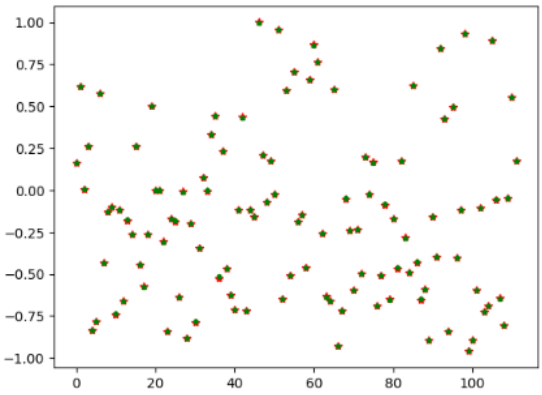

Fig 14. Datos de salida deseados vs datos de salida entrenados optimizador Adam.

En concordancia con lo obtenido en la función de pérdida, se ve cómo los datos obtenidos luego de evaluar la red neuronal con los datos de testeo coinciden exactamente con los datos deseados.

```
(112, 1)
(112, 1)
Coeficiente:  1.0006682563710436
```

Fig 15. Coeficiente de correlación para el optimizador Adam.

Como era de esperarse dados los resultados previos, el coeficiente de correlación del modelo se encuentra en un valor igual a uno, lo que justifica el hecho de que coincidan los valores deseados con los obtenidos y el error tienda a cero.

- **Optimizador RMSprop.**

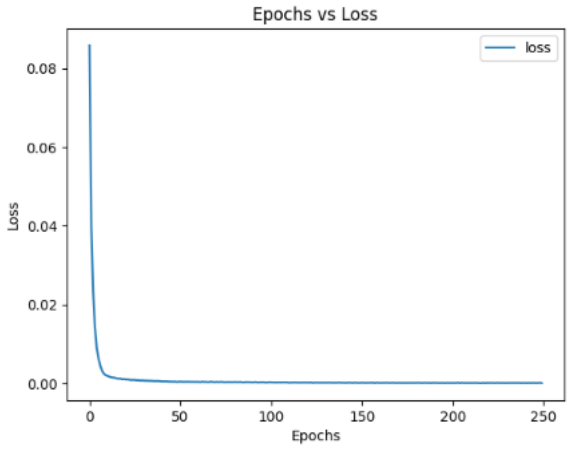

Fig 16. Gráfica epochs vs loss optimizador RMSprop.

En el caso de este optimizador, nuevamente la función de pérdida tiende a cero para un valor pequeño de épocas (menor a 50).

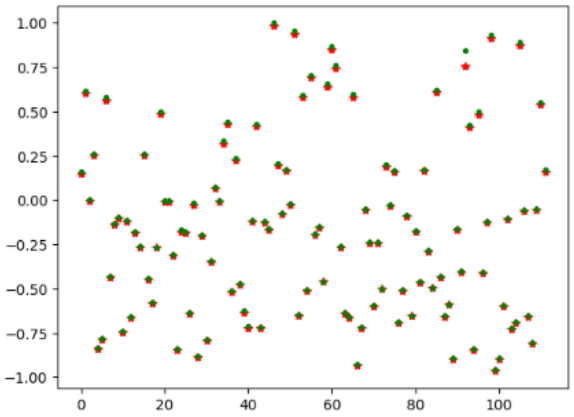

Fig 17. Datos de salida deseados vs datos de salida entrenados optimizador RMSprop.

Al momento de graficar la salida obtenida vs. la salida deseada puede verse un pequeño desplazamiento de algunos datos obtenidos respecto de los deseados, pero en gran parte todos coinciden.

```
(112, 1)
(112, 1)
Coeficiente:  0.9910923417708076
```

Fig 18. Coeficiente de correlación para el optimizador RMSprop.

El desplazamiento mencionado anteriormente se explica en su coeficiente de correlación, que en este caso es de 0.99, pero esto no implica que el optimizador no esté siendo el adecuado o que la red haya sido "mal entrenada".

- **Optimizador SGD.**

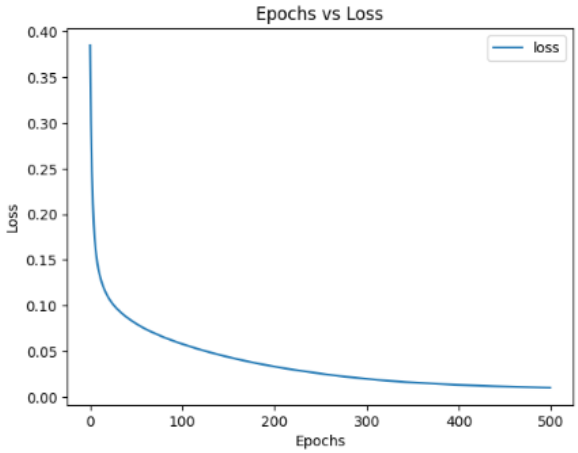

Fig 19. Gráfica epochs vs loss optimizador SGD.

Para el optimizador SGD, aún con más épocas de entrenamiento que para los otros dos optimizadores, puede verse que la función de pérdida tiende a cero recién para las 500 épocas. Esto se traduce luego en los resultados obtenidos al momento de comparar los datos producto del entrenamiento de la red y los deseados.

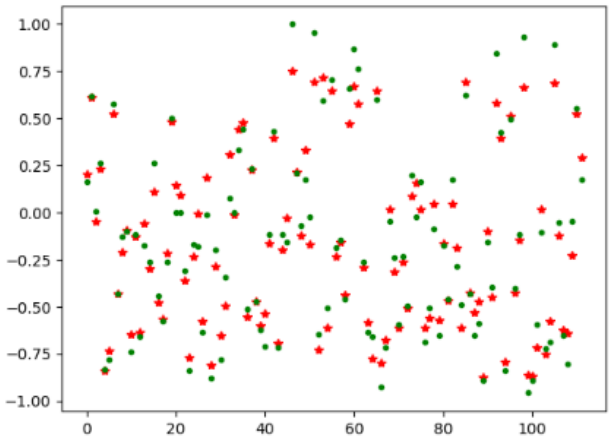

Fig 20. Datos de salida deseados vs datos de salida entrenados optimizador SGD.

Como se observa, los datos obtenidos luego del entrenamiento de la red se encuentran alejados en gran medida respecto de los datos deseados, lo que implica que este optimizador necesita más épocas o una arquitectura de red distinta para poder conseguir los datos deseados.

```
(112, 1)
(112, 1)
Coeficiente:  0.9113645361560606
```

Fig 21. Coeficiente de correlación para el optimizador SGD.

Lo dicho anteriormente se refleja en el coeficiente de correlación, mucho menor a los dos anteriores, igual a

0.91. No es un mal coeficiente pero no es el idóneo dados los resultados obtenidos con los optimizadores Adam y RMSprop.

VI. CONCLUSIONES.

De acuerdo a los resultados obtenidos durante la implementación de este miniproyecto, se puede llegar a concluir lo siguiente:

- La elección del dataset de la máquina síncrona fue fundamental para garantizar la calidad de los resultados obtenidos en nuestro proyecto.
- Al comparar los tres optimizadores (Adam, RMSprop y SGD), en la arquitectura del modelo, pudimos evaluar su rendimiento en términos de convergencia y eficacia en la optimización de los parámetros del modelo.
- Según nuestros resultados, el optimizador Adam se destacó como el más eficaz entre los tres evaluados, puesto que demostró una convergencia más rápida y un mejor ajuste a los datos, lo que indica que es una elección sólida para este problema en específico de aprendizaje automático. Cabe resaltar que aunque el optimizador Adam demostró ser el más adecuado para este proyecto en particular, es importante considerar que la elección del optimizador puede depender totalmente de la naturaleza específica del problema y de la arquitectura del modelo, así como de la cantidad de épocas de entrenamiento y de los valores de los hiper parámetros.

Articulando todo lo anteriormente mencionado, se hace pertinente concluir de manera general que este proyecto demostró la gran relevancia que tienen ciertos aspectos técnicos y las decisiones de diseño para la resolución de problemas puntuales de aprendizaje automático, debido a que, no existe una solución única para todos los problemas, y cada proyecto puede requerir un enfoque personalizado. Además se verificó la utilidad del mismo al implementarlo en plataformas de hardware como arduino. A continuación se detalla un poco más sobre este proceso. Para ello se utilizó el simulador de arduino integrado en la página Tinkercad. Cabe aclarar que para no saturar el arduino se trabajó sólo con diez datos para cada uno de los atributos (tanto entradas como salidas).

```
#include <math.h>

// Número de neuronas en cada capa
const int num_val = 10;
const int num_inputs = 4;
const int num_hidden = 8;
const int num_output = 1;

```

Fig 22. Definición de variables

Luego de ello se declaran e inicializan las matrices de pesos y bias tanto de la capa oculta como de la capa de salida, y posteriormente la matriz con los valores de entrada a la red neuronal.

```
// Valores de entrada (conocidos)
float inputs[num_val][num_inputs] = {
  {-1.0, -0.942857, 0.942857, -0.054645},
  {-1.0, -0.828571, 0.828571, -0.084699},
  {-1.0, -0.714286, 0.714286, -0.117486},
  {-1.0, -0.600000, 0.600000, -0.177596},
  {-1.0, -0.485714, 0.485714, -0.234973},
  {-1.0, -0.371429, 0.371429, -0.278689},
  {-1.0, -0.257143, 0.257143, -0.308743},
  {-1.0, -0.142857, 0.142857, -0.336066},
  {-1.0, -0.028571, 0.028571,-0.418033},
  {-1.0, 0.085714, -0.085714, -0.497268}
};
```

Fig 23. Declaración e inicialización de la matriz de entrada.

Se elaboran las funciones de activación para poder implementarlas luego:

```
// Función de activación ReLU
float relu(float x) {
  if (x < 0) {
    return 0.0;
  } else {
    return x;
  }
}

// Función de activación lineal (para la capa de salida)
float linear(float x) {
  return x;
}
```

Fig 24. Elaboración de la función de activación.

Arduino funciona con dos principales “bloques” de trabajo: la función setup y la función loop. La primera define todas aquellas acciones y configuraciones del hardware que sólo deben ejecutarse o producirse una vez. En este caso se inicializa el monitor serie en 9600 baudios.

```
void setup() {
  Serial.begin(9600);
}
```

Fig 25. Función setup

Luego, la función loop contiene todo el código que deba repetirse indefinidamente (o hasta que una condición de

corte lo detenga). En este caso se implementa la red neuronal dentro de este bloque.

```
void loop() {
  float hidden[num_hidden];
  float output[num_output];
  for (int sample = 0; sample < num_val; sample++) {
    // Propagación hacia adelante para cada conjunto de entrada

    for (int i = 0; i < num_hidden; i++) {
      hidden[i] = 0;
      for (int j = 0; j < num_inputs; j++) {
        hidden[i] += (inputs[sample][j] * weights_to_hidden[j][i])+bias_hidden[i];
      }
      hidden[i] = relu(hidden[i]);
    }

    for (int i = 0; i < num_output; i++) {
      output[i] = 0;
      for (int j = 0; j < num_hidden; j++) {
        output[i] += (hidden[j] * weights_output[j][i])+bias_output;
      }
      output[i] = linear(output[i]);
    }
    for (int i = 0; i < num_output; i++) {
      Serial.print(output[i]);
      Serial.print("\t");
    }
  }
  Serial.print("\n\n");
  delay(1000);
}
```

Fig 26. Función loop

Finalmente el monitor serie imprime las salidas obtenidas para la corriente de excitación para cada una de las entradas ingresadas:

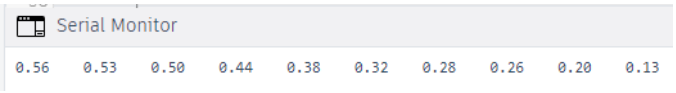

Fig 27. Monitor serie

VII. ENLACES DE ACCESO AL CÓDIGO

- Modelo con optimizador Adam: https://colab.research.google.com/drive/1f1ceN49P_erXaXgBvs_CYkbdPcY8TK7e?usp=sharing
- Modelo con optimizador RMSprop: https://colab.research.google.com/drive/1fUUhTMfLCQv1btk3pFECH8BOVKva4zuB?usp=sharing
- Modelo con optimizador SGD: https://colab.research.google.com/drive/1uZboUqcnCZOTZvR2TlKYIjJHBQNWyhps?usp=sharing
- Implementación en arduino: https://www.tinkercad.com/things/9bNx1r9ss6w-shiny-snaget-jarv/editel?sharecode=BqZcQqoNivXUZeNGYgS40V8nXQBlJlVk3hImVUlBU4Q

VIII. REFERENCIAS.

[1] Synchronous Machine Data Set. (2021). UCI Machine Learning Repository. Available on: https://doi.org/10.24432/C5W32R.

[2] R. Heymsfeld. A neural network for arduino. Hobbizine. Available on: http://robotics.hobbizine.com/arduinoann.html.

[3] Xabier Maestre Betolaza. (2020/2021). Implementación de redes neuronales en plataformas hardware para su aplicación en ingeniería eléctrica. Escuela de Ingeniería de Bilbao. Available on: https://addi.ehu.es/bitstream/handle/10810/54026/TFM_XabierMaestreBetolaza.pdf?sequence=1.