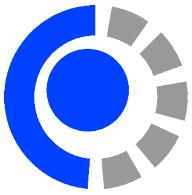

Universidad Nacional Del Comahue

Facultad de Informática

Sistemas Inteligentes

Trabajo práctico obligatorio
Redes neuronales

Alumnos:

LUSSO, Adriano Mauricio FAI-2908
adriano.lusso@est.fi.uncoma.edu.ar
TORRES, Bianca Antonella FAI-2991
bianca.torres@est.fi.uncoma.edu.ar

Profesores del curso:

Sandra Roger
Christian Gimenez

Abril 2024

Índice

1. Introducción

Las redes neuronales se han abierto paso en la ciencia de datos y la inteligencia artificial, siendo de gran utilidad en entornos académicos e industriales. Algunas de sus aplicaciones son el reconocimiento de patrones y la predicción de ciertos eventos.

A pesar de la gran perspectiva actual y a futuro que presentan, estos modelos cuentan con desafíos significativos. Entre estos, la necesidad de grandes datos para entrenamiento, riesgos tales como el *overfitting* y la compleja parametrización que requieren para obtener resultados útiles. En este contexto, este trabajo busca probar dos implementaciones diferentes de una red neuronal, con el fin de comparar sus resultados. Una de ellas, hecha a través de una implementación realizada en Python. La otra, siendo un modelo predefinido de red neuronal que proporciona el software WEKA [3]. Con esto, se busca lograr un entendimiento profundo de lo que involucra la creación y utilización de redes neuronales, así como el análisis de las diversas métricas que permiten caracterizar su rendimiento.

La metodología de este trabajo consiste en seguir un flujo de trabajo basado en cuatro etapas básicas: la preparación de los datos, la creación del modelo, los entrenamientos del modelo y, finalmente, su evaluación.

2. Estructura

En la Sección 3, se detallarán todos los conceptos asociados al marco teórico. Del mismo, se encontrará en la Subsección 3.1 la explicación del modelo de redes neuronales artificiales. En la Subsección 3.2, una introducción al software WEKA. Y, en la Subsección 3.3, una explicación de un formato de archivo que sera de utilidad para representar un data-set. En la Sección 4, se introduce el data-set utilizado. En la Sección 5, se profundiza sobre metodología llevada a cabo. En la Sección 6, se explica como fue la preparación general del data-set. En la Sección 7 y sus respectivas subsecciones, se definen las implementaciones de la red neuronal en WEKA y en el prototipo en Python, especificando las respectivas configuraciones llevadas a cabo. En la Sección 8, se explican los entrenamiento llevados a cabo para ambas implementaciones, para posteriormente analizar sus resultado en la Sección 9. Finalmente, en la Sección 10 y Sección 11 se discute y concluye, respectivamente, sobre los resultados encontrados.

3. Marco teórico

3.1. Redes neuronales

Las redes neuronales surgen como parte de los modelos conexionistas, uno de los paradigmas alternativos a la inteligencia artificial simbólica. Estos modelos se inspiran en el funcionamiento del cerebro, a través del modelado de neuronas individuales y sus conexiones. Formalmente, se las define como una estructura en

red de elementos de procesamiento simples, es decir, sus neuronas. Estas poseen un alto grado de interconectividad parametrizada, de forma que la información fluye en forma de cascada. Cada conexión entre neuronas cuenta con un peso asociado, que es un valor que indica la importancia de la información comunicada por esa conexión. El aprendizaje de información tiene lugar en la actualización de los pesos.

3.1.1. Elementos del modelo

El primer elemento, ya introducido, es la neurona. En la Figura 1, puede verse su estructura básica. Es la unidad básica de procesamiento, encargada de recibir, procesar y transmitir la información. Las neuronas tienen embebidas en su funcionamiento una función de activación. La misma recibe de entrada para el procesamiento una entrada neta y un sesgo o *bias*. La entrada neta se obtiene de promediar los datos provenientes de las conexiones de entrada de la neurona y sus respectivos pesos. El bias es un valor constante añadido a la entrada neta, que permite que la red neuronal tenga mayor flexibilidad. Haciendo uso de estas dos entradas, se utiliza la función de activación, que permite introducir no linealidad en el procesamiento de datos del modelo. Es decir, permite expresar relaciones entre variables con funciones de mayor complejidad que la dada por funciones lineales. En la Figura 3, pueden verse alguna de las funciones de activación más utilizadas.

El segundo elemento, son las capas de neuronas. Estas pueden ser capas de entrada, ocultas y de salida. La capa de entrada es la primera de una red neuronal, encargada de recibir las señales de entrada al algoritmo. Las capas ocultas son capas intermedias entre las de entrada y de salida. Extraen patrones relevantes de los datos, haciendo uso de las conexiones entre neuronas. La capa de salida es la ultima capa de la red, que produce la salida de la misma. En la Figura 2, se observa la estructura general de una red neuronal.

El tercer elemento, también ya introducido previamente, son las conexiones entre neuronas. Cada conexión tiene un peso asociado, que permite ajustar la influencia entre neuronas. Es decir, caracteriza la importancia del dato transportado por esa conexión. Los pesos son un elemento fundamental ya que son los parámetros que se ajustan durante el entrenamiento del modelo. Es decir, este elemento es el que caracteriza la simulación de aprendizaje.

El cuarto y quinto elemento son, respectivamente, la función de perdida y el algoritmo de optimización. A la hora de entrenar una red neuronal, se realizan n iteraciones, donde se ajustan los pesos de la misma. A estas iteraciones se las denomina **épocas**. En cada época, se realiza una ejecución de la red neuronal con una entrada, y se calcula la función de perdida con la salida obtenida. Esta función muestra la diferencia entre el resultado y el verdadero valor que debería haber inferido la red neuronal. La función de perdida es utilizada como la función objetivo del algoritmo de optimización, que busca minimizar la diferencia entre el valor inferido de la red neuronal y el valor real a partir de la modificación de los pesos.

Además del ajuste de pesos, el algoritmo de optimización tiene implementada

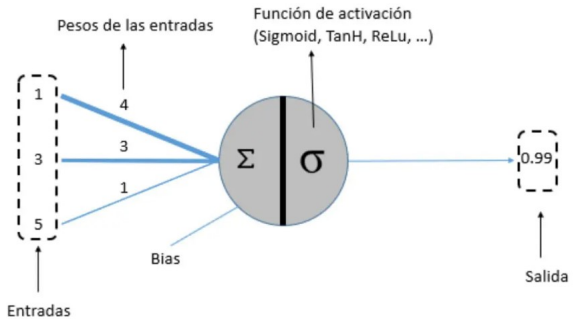

Figura 1: Modelo básico de una neurona artificial.

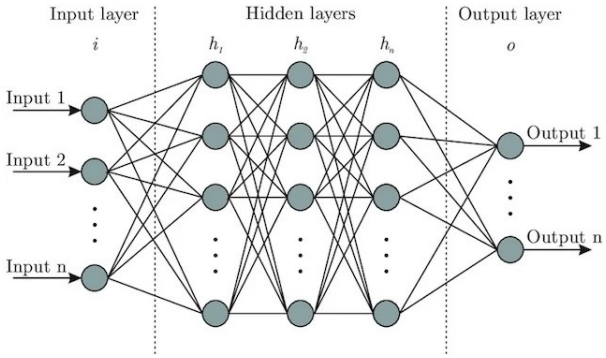

Figura 2: Modelo básico de una red neuronal artificial.

una técnica denominada **retro-propagación**. Esta técnica calcula el gradiente de la función de perdida con respecto a cada peso en la red, propagando desde la ultima capa hacia la primera el error calculado, con el fin de que el ajuste de los pesos de las conexiones minimice el error.

3.2. Software WEKA

El *Waikato Environment for Knowledge Analysis* (WEKA) [10] es un conjunto de librerias de clases Java que implementa métodos de aprendizaje automático y de minería de datos. Además, brinda herramientas para pre-procesamiento de datos y análisis estadísticos de resultados.

Su principal categoría de algoritmos de aprendizaje es la de **clasificadores**. Estos definen un conjuntos de reglas que permiten el modelado de los datos. Otras de sus categorías son las de reglas de asociación y clustering de datos.

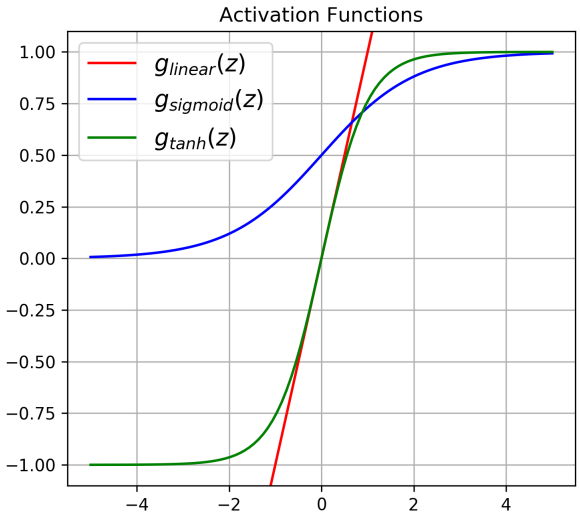

Figura 3: Gráfico de algunas de las funciones de activación más conocidas.

3.3. Attribute-relation file format

El *attribute-relation file format* (ARFF) [9], es un formato estandarizado para representar datos en función de su nombre y de su tipo, y es utilizado por WEKA para incorporar data-sets en sus diferentes algoritmos. A continuación, se visualiza un pseudocódigo para ejemplificar su sintaxis:

```
@relation vehiculos

@attribute cantidadRuedas {1,2,3,4}
@attribute tieneMotor Boolean
@attribute maxVelocidadAlcanzada Numeric

@data
4,true,400
2,false,30
...
```

En el mismo, primero se declara el nombre de la relación a modelar. Luego se definen los atributos de la relación, especificando su nombre y su dominio. Para el primer atributo, vemos un tipo de datos nominal, donde se especifica cada elemento del dominio. Para los otros dos atributos, se utilizan tipos de datos predefinidos. Finalmente, se especifica un listado de tuplas de la relación. Cada fila es una tupla. Los valores de los atributos se especifican en el mismo orden en el estos fueron declarados y se usa el carácter 'coma' como separador.

4. Data-set utilizado

Para las pruebas desarrolladas, se utilizó un data-set de animales, donde se especifica su nombre, ciertas características físicas y su tipo [4]. Cuenta con 101 instancias caracterizadas por los siguientes atributos:

1. animal name: el nombre del animal, único por cada instancia
2. hair: si tiene pelo
3. feathers: si tiene plumas
4. eggs: si pone huevos
5. milk: si da leche
6. airborne: si puede volar
7. aquatic: si es acuático
8. predator: si es un depredador
9. toothed: si tiene dientes
10. backbone: si es vertebrado
11. breathes: si respira
12. venomous: si es venenoso
13. fins: si tiene aletas
14. legs: cuantas piernas tiene
15. tail: si tiene cola
16. domestic: si es domestico
17. catsize: si tiene el tamaño de un gato
18. type: el tipo del animal

El mismo se se encuentra en el archivo `zoo.data`, y su descripción detallada en `zoo.names`.

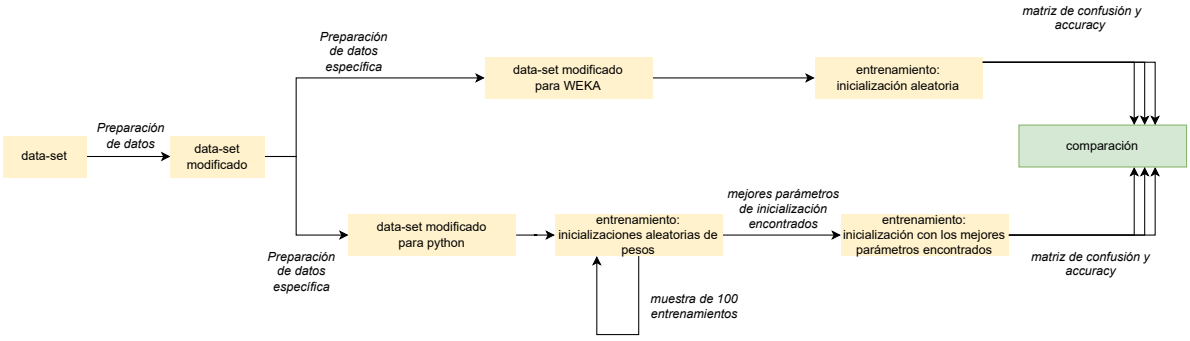

Figura 4: Flujo de trabajo del proyecto explicado en este informe

5. Metodología

La metodología se visualiza como un flujo de trabajo sencillo en la Sección 6. Como ya se mencionó en la Sección 1, esta fuertemente inspirada en cuatro pasos básicos: la preparación de los datos, la creación del modelo, el entrenamiento y su evaluación. Particularmente, en el flujo de trabajo planteado se respeta la etapa de evaluación final de resultados. Pero, las primeras tres etapas, cuentan con una variación. En primer lugar, la preparación de los datos se divide en dos subetapas: la preparación general y las preparaciones específicas. La preparación general, descrita en la Sección 6, realiza una estandarización en el data-set que es común a todos los entrenamiento que se harán. En cambio, las preparaciones específicas varían de acuerdo a si el entrenamiento es en WEKA o en el prototipo implementando en Python. Estas preparaciones específicas serán detalladas en las respectivas Subsección 7.1 y Subsección 7.2. Estas diferencias en la preparación específica de datos generan una bifurcación en el flujo de trabajo. También se ven diferencias en las etapas de entrenamiento del modelo previsto. Las mismas serán especificadas en las Subsección 8.1 y Subsección 8.2.

6. Preparación general del data-set

De los atributos mencionados en Sección 4, el nombre del animal será descartado, ya que no aporta información relevante para la detección de patrones. De los atributos 2 al 13 y del 15 al 17, son de tipo booleano. El atributo 14 puede tomar valores del conjunto $\{0, 2, 4, 6, 8\}$. El atributo 18 puede tomar valores naturales del rango $[1, 7]$. Para la red neuronal, se utilizarán los atributos del 2 al 17 como datos de entrada (denominados como conjunto X), y el atributo 18 será el que debe inferir la red neuronal (denominado como conjunto y).

Haciendo uso de las modificaciones planteadas, se creará un archivo ARFF llamado `zoo_modified.arff`.

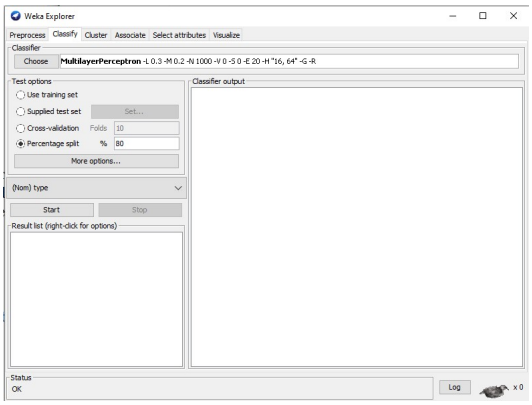

Figura 5: Configuración en WEKA de los conjuntos de entrenamiento y prueba

7. Detalles de la implementación

Esta sección explica la preparación específica de datos para WEKA y para el prototipo en Python, así como la implementación de este último. Toda la implementación se encuentra en el mismo directorio en el que este informe fue entregado.

7.1. Implementación en WEKA

Se decidió utilizar `MultilayerPerceptron` que, como su nombre indica, es el módulo de WEKA que implementa un perceptrón multicapa. Sus neuronas tienen implementadas una función de activación sigmoide [1]. Se le configuró un ratio de aprendizaje de 0,01, y 1000 épocas de entrenamiento, además de permitir la normalización de los datos. Se dividieron los datos en un 80 % de datos de entrenamiento y un 20 % de prueba usando el el módulo de división automática que incluye WEKA. El mismo realiza la división entre conjunto de prueba y entrenamiento de manera aleatoria con el uso de una semilla (*seed*). En la Figura 5 y Figura 6 se puede apreciar las configuraciones previamente explicadas.

7.2. Prototipo implementado en Python

Para la implementación en código se utilizó el lenguaje de programación Python, además de las librerías Numpy [7] y Pandas [5] para un sencillo manejo de grandes estructuras de datos.

Para empezar, dado los conjuntos X e Y definidos en Sección 4, se deben subdivir en X_{train}, X_{test}, y_{train} e y_{test}. Como los nombres de los subconjuntos indican, estos son los respectivos subconjuntos de entrenamiento y prueba. Esta subdivisión se hizo de manera de que los conjuntos sean lo suficientemente representativos en sus valores, tanto en X como en y. Para esto, se utilizó la función

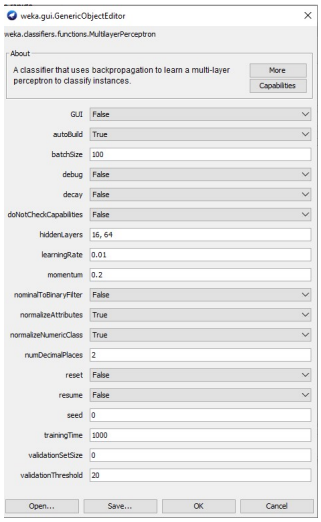

Figura 6: Configuración en WEKA de ratio de aprendizaje y otros parámetros

`train_test_split(X, y, test_size=0.2, stratify=y)`. La misma divide los conjuntos X e y en X_{train}, X_{test}, y_{train} e y_{test} aleatoriamente, respetando que el 80 % del data-set forme parte del conjunto de entrenamiento. El parámetro $stratify$ es de gran importancia, ya que garantiza que, en los conjuntos, haya una alta representabilidad de datos en función de los valores existentes en y. Los conjuntos generados pueden ser encontrados en los archivos `X_train.csv`, `X_test.csv`, `y_train.csv` e `y_test.csv` respectivamente. Finalmente, los datos fueron normalizados utilizando un método fuertemente inspirado en la normalización min-max [8], simplificando su rango de valores a reales en el rango $[0, 1]$.

Luego, se definió un perceptrón multicapa en una clase Python con el mismo nombre, cuya implementación se ve en la Subsección 12.1. En esta, se cuenta con una capa de entrada, una de salida y una única capa oculta. Tanto sus pesos como sus sesgos pueden ser inicializados de manera aleatoria o de manera personalizada por el usuario. La red neuronal implementa una función sigmoide [6] para todas sus neuronas, y como optimizador, el algoritmo de descenso de gradiente [2]. Para el optimizador, se definió también la función sigmoide derivada, que permite el cálculo del gradiente. Ambas funciones mencionadas pueden verse en la Subsección 12.2.

8. Experimentos

Esta sección explica las etapas de entrenamiento de las respectivas bifuraciones del flujo de trabajo. Es decir, para el software WEKA y para el prototipo en Python.

8.1. Entrenamiento en WEKA

Gracias a la facilidad de uso para el usuario, el entrenamiento en WEKA se realiza únicamente presionando el botón `START`. Luego de esto, WEKA proporciona a través de un reporte amigable tanto el modelo entrenado como diferentes métricas para evaluación de su rendimiento.

8.2. Entrenamiento en Python

El entrenamiento en Python consistió en dos experimentos. El primero, haciendo uso de inicializaciones aleatorias de pesos y sesgos. El segundo, tomando aquellos parámetros que lograban la mejor optimización, y realizando un segundo entrenamiento con los mismos.

8.2.1. Primer experimento: inicialización aleatoria de pesos y sesgos

En principio, se decidió hacer múltiples entrenamientos de la red neuronal, sin especificar los pesos y sesgos de inicialización. De esta manera, estos serian inicializados aleatoriamente. Para esto, se configuró 1000 épocas de entrenamiento y un ratio de aprendizaje de 0,01. Tras un muestreo de 100 entrenamientos, y su consecuente evaluación de su métrica de exactitud (*accuracy*), se encontró una inicialización cuyo entrenamiento resultaba en un rendimiento aceptable para los esperados en el marco de este trabajo. Los mismos fueron guardados en los siguientes archivos:

- `initialized_weights_input_hidden.csv`
- `initialized_weights_hidden_output.csv`
- `initialized_weights_bias_hiden.csv`
- `initialized_weights_bias_output.csv`

8.2.2. Segundo experimento: inicialización de pesos y sesgos óptimos

En este segundo experimento, se decidió usar la misma configuración de la red neuronal que la usada para el de la sección Subsección 8.2.1, con la diferencia de que no se inicializaría los pesos y sesgos aleatoriamente. Esta vez, se utilizarían aquellos que mejor fueron optimizados. El código de este experimento se encuentra en `nn_python.ipynb`, y se puede ejecutar abriendo el notebook y activando todas sus celdas desde un entorno local, o desde Google Colab. En la Figura 7, se ve la celda que ejecuta el experimento en sí. Esta celda hace uso de las celdas previas, donde se definen las funciones relevantes.

```
X_train,X_test,y_train,y_test = cargar_train_test()
X_train,X_test,y_train,y_test = preparar_datos(X_train,X_test,y_train,y_test)
weights_input_hidden,weights_hidden_output,bias_hidden,bias_output = importar_pesos_bias()

input_size = 16#X_train.shape[1]
hidden_size = 64
output_size = 7

perceptron = PerceptronMulticapa(input_size, hidden_size, output_size,weights_input_hidden,
                                 weights_hidden_output,bias_hidden,bias_output)
# esta inicializacion de abajo fue usada previamente. Aca, no se pasan por parametros los pesos y bias.
#Por lo tanto, se inicializan aleatoriamente.
# perceptron = PerceptronMulticapa(input_size, hidden_size, output_size)

perceptron.entrenar(X_train, y_train, epocas=1000, ratio_aprendizaje=0.01,debug=True)

predicciones = perceptron.predecir(X_test)
```

Figura 7: Celda de ejecución del segundo experimento en la implementación en código Python

9. Análisis de resultados

Sobre el experimento en WEKA, se logró de forma consistente un accuracy de 60 %, cuya matriz de confusión asociada es

$$\begin{pmatrix} 8 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 & 0 & 0 \end{pmatrix} \tag{1}$$

donde un elemento en (i, j) indica que fue clasificado como de categoría i siendo realmente de categoría j, tal que $i, j \in [1, 7]$. En los archivos `weka_nn_result` y `weka_nn.model` se encuentran el log de resultado y los parametros optimizados respectivamente.

Respecto al prototipo en Python, los resultados del entrenamiento con inicialización aleatoria eran mucho menos consistentes. Los mismos podían resultar en un accuracy desde el 4 % hasta el 47 %. Una vez que se inicializaba el modelo con los parámetros que lograban optimizar hacia un 47 %, el accuracy se mantenía. Su matriz de densidad es:

$$\begin{pmatrix} 8 & 0 & 0 & 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 \end{pmatrix} \tag{2}$$

10. Discusiones

En principio, se ve una mejora importante en términos de accuracy del modelo WEKA sobre el modelo en código. Esto es algo esperable, ya que es probable que WEKA realice otro tipo de procesamiento y técnicas más complejas que no fueron implementadas en el código. Es interesante marcar la gran diferencia en estabilidad de optimización en los modelos. Como se mencionó en la Sección 9, el modelo en código tenía alta variabilidad en los accuracy conseguidos en diferentes optimizaciones a comparación del modelo WEKA, incluso aunque ambos tuvieran una inicialización aleatoria de conjuntos de prueba y entrenamiento. No se cree que sea un error en la representabilidad de los datos en la creación del prototipo codificado, ya que se usó la función `train_test_split` con el parámetro `stratify` configurado.

11. Conclusiones

Se realizó una comparación exitosa de el perceptrón multicapa implementado en WEKA contra una implementación más simple del mismo modelo en código Python. Se compararon los resultados y se detectaron posibles causas a ciertos problemas de rendimiento en la implementación en Python. Como futura mejora, se plantea mejorar la implementación en Python para permitir múltiples capas ocultas, y realizar mayor investigación sobre que otro procesamiento puede estar realizando WEKA para la inicialización aleatoria de parámetros.

El desarrollo de este trabajo fue de vital importancia a la hora de profundizar, de parte de los integrantes del mismo, en como se desarrolla un flujo de trabajo para aprendizaje automático. Además, permitió comprender los conceptos esenciales que caracterizan al modelo de redes neuronales.

12. Anexos

12.1. Código Python de Perceptrón Multicapa

```
class PerceptronMulticapa:
    def __init__(self, input_size, hidden_size, output_size,weights_input_hidden=None,
                 weights_hidden_output=None,bias_hidden=None,bias_output=None):
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size

        # Inicializacion de pesos y bias
        # Estos parametros iran cambiando a medida que se optimizen
        if weights_input_hidden is None:
            self.weights_input_hidden = np.random.rand(input_size, hidden_size)
        else:
            self.weights_input_hidden = weights_input_hidden
```

```
    if weights_hidden_output is None:
        self.weights_hidden_output = np.random.rand(hidden_size, output_size)
    else:
        self.weights_hidden_output = weights_hidden_output

    if bias_hidden is None:
        self.bias_hidden = np.random.rand(1, hidden_size)
    else:
        self.bias_hidden = bias_hidden

    if bias_output is None:
            self.bias_output = np.random.rand(1, output_size)
    else:
        self.bias_output = bias_output


    # Se guardan las inicializaciones de pesos y bias en atributos diferentes
    self.initialized_weights_input_hidden = self.weights_input_hidden.copy()
    self.initialized_weights_hidden_output = self.weights_hidden_output.copy()
    self.initialized_bias_hidden = self.bias_hidden.copy()
    self.initialized_bias_output = self.bias_output.copy()


def pensar(self, X):
    self.hidden_input = np.dot(X, self.weights_input_hidden) + self.bias_hidden
    self.hidden_output = sigmoide(self.hidden_input)
    self.output = np.dot(self.hidden_output, self.weights_hidden_output)
    + self.bias_output
    return self.output

def retropropagacion(self, X, y, output, ratio_aprendizaje):
    # Retropropagacion
    error = y - output
    d_output = error
    d_hidden = np.dot(d_output, self.weights_hidden_output.T)
    * sigmoide_derivada(self.hidden_output)

    # Actualizacion de pesos y bias
    self.weights_hidden_output += np.dot(self.hidden_output.T, d_output)
    * ratio_aprendizaje
    self.weights_input_hidden += np.dot(X.T, d_hidden) * ratio_aprendizaje
    self.bias_output += np.sum(d_output, axis=0, keepdims=True)
    * ratio_aprendizaje
    self.bias_hidden += np.sum(d_hidden, axis=0, keepdims=True)
    * ratio_aprendizaje
```

```
def predecir(self, X):
    return self.pensar(X)

def entrenar(self, X, y, epocas, ratio_aprendizaje,debug=False,
intervalo_de_epoca = 100):
    for epoca in range(epocas):
        output = self.pensar(X)

        self.retropropagacion(X, y, output, ratio_aprendizaje)

        perdida = np.mean(np.square(y - output))
        if debug and epoca % intervalo_de_epoca == 0:
            print(f'Epoca {epoca}, Perdida: {perdida:.4f}')
```

12.2. Código Python de la función sigmoide y su derivada

```
# Funcion de activacion sigmoide
def sigmoide(x):
    x = np.clip(x, -500, 500)
    return 1 / (1 + np.exp(-x))

# Derivada de la funcion de activacion sigmoide
def sigmoide_derivada(x):
    return x * (1 - x)
```

Referencias

[1] Mahmoud Abdel-Sattar, Abdulwahed M Aboukarima, and Bandar M Alnahdi. Application of artificial neural network and support vector regression in predicting mass of ber fruits (ziziphus mauritiana lamk.) based on fruit axial dimensions. *Plos one*, 16(1):e0245228, 2021.

[2] Shun-ichi Amari. Backpropagation and stochastic gradient descent method. *Neurocomputing*, 5(4-5):185–196, 1993.

[3] Remco R Bouckaert, Eibe Frank, Mark Hall, Richard Kirkby, Peter Reutemann, Alex Seewald, David Scuse, et al. Weka manual for version 3-6-2. *The University of Waikato*, 2010.

[4] Richard Forsyth. Zoo. UCI Machine Learning Repository, 1990. DOI: https://doi.org/10.24432/C5R59V.

[5] Wes McKinney et al. Data structures for statistical computing in python. In *Proceedings of the 9th Python in Science Conference*, volume 445, pages 51–56. Austin, TX, 2010.

[6] Sridhar Narayan. The generalized sigmoid activation function: Competitive supervised learning. *Information sciences*, 99(1-2):69–82, 1997.

[7] Travis Oliphant. NumPy: A guide to NumPy. USA: Trelgol Publishing, 2006–.

[8] SGOPAL Patro and Kishore Kumar Sahu. Normalization: A preprocessing stage. *arXiv preprint arXiv:1503.06462*, 2015.

[9] Hannes Tribus. Static code features for a machine learning based inspection: An approach for c, 2010.

[10] Ian H Witten, Eibe Frank, Leonard E Trigg, Mark A Hall, Geoffrey Holmes, and Sally Jo Cunningham. Weka: Practical machine learning tools and techniques with java implementations. 1999.