

# CENTRO UNIVERSITARIO DE CIENCIAS EXACTAS E INGENIERÍAS



DEPARTAMENTO DE CIENCIAS COMPUTACIONALES  
LEVENBERG-MARQUARDT NEURAL NETWORK (LMNN)  
PROYECTO FINAL

**Estudiante:**

- Diego Andrés Preciado Rodríguez
- Manuel Sedano Luna
- Alejandro Sedano Luna

**Materia:** INTELIGENCIA ARTIFICIAL II

**Sección:** D02

**Horario:** 19:00P.M-20: 55P.M

**Docente:** JULIO ESTEBAN VALDEZ LÓPEZ

**Calendario:** 2023B

**Fecha y lugar:** GUADALAJARA, 22/11/2023

[https://github.com/dethres01/IA2\\_LMNN\\_2023](https://github.com/dethres01/IA2_LMNN_2023)

## INDICE

Introducción.....	3
¿Qué es el algoritmo Leverberg-Marquardt?.....	3
Formula general.....	3
Jacobiano.....	4
Matriz Hessiana.....	5
Error cuadrático medio.....	5
Pasos para la implementación.....	6
Paso 1:Definir las variables.....	6
Paso 2: Calcular: Si= -[H+ I]-1 $\nabla f(x_1)$ .....	6
Paso 3: Actualizar: $x_{i+1} = x_i + S_i$ .....	6
Paso 4: Comparar: Si $f(x_{i+1}) < f(x_i)$ .....	6
Paso 5: Termina ciclo iterativo usando:.....	7
Paso 6: Imprimir: $x_i + 1, f(x_i + 1)$ .....	7
Desarrollo.....	7
def set_canvas(self):.....	7
def train_jacobian(self):.....	8
plotting_jacob(self):.....	11
Código de la práctica:.....	11
<b>Capturas del programa en ejecución.....</b>	<b>19</b>
<b>Conclusión.....</b>	<b>23</b>
Bibliografías:.....	24
Ilustración 1 Despliegue del programa vista general	19
Ilustración 2 Inserción de los puntos por mouse	19
Ilustración 3 Entrenamiento de la red neuronal con los datos ingresados	20
Ilustración 4 Resultado obtenido del entrenamiento	20
Ilustración 5 Inserción de los puntos por mouse	21
Ilustración 6 Entrenamiento de la red neuronal con los datos ingresados usando levenberg marquardt	21
Ilustración 7 Resultado obtenido	22

# Introducción

## ¿Qué es el algoritmo Levenberg-Marquardt?

El algoritmo de Levenberg-Marquardt es un método de optimización que se utiliza para resolver problemas de mínimos cuadrados no lineales. Estos problemas son comunes en muchas áreas de la ciencia y la ingeniería, como el ajuste de curvas, la solución de sistemas de ecuaciones no lineales y la optimización de funciones.

Ya que es la combinación de dos métodos: el método de Gauss-Newton y el método de descenso de gradiente. El método de Gauss-Newton es rápido pero puede ser inestable, mientras que el método de descenso de gradiente es más lento pero más estable. El algoritmo de Levenberg-Marquardt intenta obtener lo mejor de ambos métodos, ajustando un parámetro ( $\lambda$ ) que determina cuánto confiar en cada método en cada paso del algoritmo.

En resumen, el algoritmo de Levenberg-Marquardt es útil cuando necesitamos encontrar una solución que minimice el error entre un conjunto de datos y un modelo matemático, especialmente cuando ese modelo no es lineal.

para poder entender y poder implementarlo este algoritmo es necesario sentar algunas bases

la fórmula del algoritmo de Levenberg-Marquardt está denotada de la siguiente manera =  $-(J^T J + \lambda I)^{-1} J^T e$

## Formula general

$$w_{k+1} = w_k - [J^T J + \mu I]^{-1} J^T e$$

Donde:

$w_k$  es el vector de parámetros en la iteración actual.

$w_{k+1}$  es el vector de parámetros en la próxima iteración.

$J$  Es el Jacobiano del modelo con respecto a los parámetros.

$\mu$  Es un escalar que determina cuánto confiar en cada método en cada paso del algoritmo, al inicio del algoritmo tiene un valor pequeño. El valor de  $\lambda$  es alterado durante cada iteración si se ha encontrado un valor más pequeño al evaluar la función objetivo. si el valor de la función objetivo es

reducido. lambda también se reduce. por el otro lado si la función objetivo aumenta lambda incrementa

$I$  es la matriz identidad.

$e$  Es el vector de errores (la diferencia entre los datos observados y los predichos por el modelo).

## Jacobiano

El jacobiano está descrito con la siguiente fórmula:

Sea  $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$  una función cuyas derivadas parciales de primer orden existen en todo  $\mathbb{R}^n$  y denotemos  $f_1, f_2, \dots, f_m$  a sus componentes escalares. Se define la matriz jacobiana de  $f$  en un punto  $x \in \mathbb{R}^n$  como:

$$J_f(x) = \begin{pmatrix} \frac{\partial f_1}{\partial x_1}(x) & \frac{\partial f_1}{\partial x_2}(x) & \dots & \frac{\partial f_1}{\partial x_n}(x) \\ \frac{\partial f_2}{\partial x_1}(x) & \frac{\partial f_2}{\partial x_2}(x) & \dots & \frac{\partial f_2}{\partial x_n}(x) \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1}(x) & \frac{\partial f_m}{\partial x_2}(x) & \dots & \frac{\partial f_m}{\partial x_n}(x) \end{pmatrix} = \begin{pmatrix} \nabla f_1(x) \\ \vdots \\ \nabla f_m(x) \end{pmatrix}$$

La matriz Jacobiana contiene las derivadas parciales de la función objetivo con respecto a los parámetros del modelo es como un mapa que nos dice cómo se inclina la función objetivo cuando cambiamos un poquito los parámetros del modelo. Es esencial para saber en qué dirección ajustar nuestros parámetros.

## Matriz Hessiana

$$\mathbf{H}f(x_0, y_0, \dots) = \begin{bmatrix} \frac{\partial^2 f}{\partial \mathbf{x}^2}(x_0, y_0, \dots) & \frac{\partial^2 f}{\partial \mathbf{x} \partial \mathbf{y}}(x_0, y_0, \dots) & \cdots \\ \frac{\partial^2 f}{\partial \mathbf{y} \partial \mathbf{x}}(x_0, y_0, \dots) & \frac{\partial^2 f}{\partial \mathbf{y}^2}(x_0, y_0, \dots) & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix}$$

The above Hessian is of the function  $f : \mathbb{R}^n \rightarrow \mathbb{R}$  where all second order partial derivatives of  $f$  exist and are continuous throughout its domain & the function is  $f(x_1, x_2, x_3, \dots, x_n)$

La matriz Hessiana completa puede ser costoso en términos computacionales, por lo que en el método de Levenberg-Marquardt se utiliza una aproximación de la matriz Hessiana ( $H$ ) que se calcula como

$$H = [J^T J] + \mu I$$

Donde  $I$  es una matriz identidad por lambda (Gradiente descendiente), esta matriz nos da una idea de la curvatura de la función objetivo en el espacio de parámetros, lo que ayuda a ajustar los parámetros de una manera más efectiva.

## Error cuadrático medio

Y luego está el error cuadrático medio, en la implementación lo calculamos de la siguiente manera:

$$\text{square\_error} = 1/N (D - Y)^2$$

que es básicamente una medida de qué tan mal nuestro modelo está haciendo sus predicciones en comparación con la realidad. En este algoritmo, tratamos de

minimizar la suma de los cuadrados de estos errores, lo que significa mejorar la concordancia entre lo que predice nuestro modelo y lo que realmente vemos.

## Pasos para la implementación

para la correcta implementación del algoritmo es vital el seguir una serie de pasos para su correcto funcionamiento los cuales están denotados por las siguientes instrucciones

### Paso 1:Definir las variables

- lambda\_factor = 0.1
- eta = 0.1 -> aprendizaje rate (learning rate)
- epoch = 1500 -> número de épocas (iteraciones)
- n\_epoch = 0 -> número de épocas (iteraciones) actuales (para gui)
- min\_error = 0.01 -> error mínimo para parar el entrenamiento
- n\_neurons = 6 -> número de neuronas en la capa oculta (hidden layer)
- w\_hidden = np.matrix(np.random.rand(n\_neurons, 3)) -> pesos de la capa oculta (hidden layer)
- w\_output = np.random.rand(n\_neurons+1) -> pesos de la capa de salida (output layer) +1 por el bias (sesgo)
- X = [] -> puntos de entrada (input) (x,y)
- d = [] -> salida deseada (desired output) (0,1) (1,0) (0,0) (1,1)
- errors = [] -> errores de cada época (iteración) (para gui)
- fig\_e, ax\_e = plt.subplots() -> figura y ejes para el error
- fig, ax = plt.subplots() -> figura y ejes para el gráfico
- canvas = None -> canvas para el gráfico
- canvas\_e = None -> canvas para el error

**Paso 2: Calcular:**  $S_i = - [H + \lambda I]^{-1} \nabla f(x_i)$

**Paso 3: Actualizar:**  $x_{i+1} = x_i + S_i$

**Paso 4: Comparar:** Si  $f(x_{i+1}) < f(x_i)$

Entonces  $\lambda_{i+1} = 0.9\lambda_i$

Sino  $\lambda_{i+1} = 1.1\lambda_i$

**Paso 5: Termina ciclo iterativo usando:**

Si  $|f(x_{i+1}) - f(x_i)| > \epsilon_1$  o  $\|\nabla f(x_i)\| > \epsilon_2$

Entonces vuelve al paso 2

Sino ve al paso 6

**Paso 6: Imprimir:**  $x_{i+1}, f(x_{i+1})$

## Desarrollo

Lo que se realizó fue tratar de implementar una red neuronal multicapa utilizando el algoritmo Levenberg-Marquardt para clasificación binaria. La red neuronal tiene dos entradas y una salida, lo que la hace adecuada para tareas de clasificación binaria. La función sigmoidea se utiliza como función de activación y el error cuadrático medio se emplea como función de pérdida

**def set\_canvas(self):**

esta función es responsable de configurar y mostrar la interfaz gráfica para visualizar el proceso de entrenamiento del algoritmo de Levenberg-Marquardt

Creamos una nueva ventana principal para después asignar un título a la ventana a continuación Establecemos el tamaño de la ventana, Adicionalmente agregamos una imagen de fondo a la ventana creamos y colocamos un lienzo para el gráfico principal en la ventana para después conectar un evento de clic de ratón para agregar puntos al gráfico y continuamos creando y colocando un lienzo para visualizar el error en la ventana. En el área de los botones creamos tres botones el primero para iniciar el entrenamiento normal el segundo para iniciar el entrenamiento jacobian y el último para reiniciar y limpiar el lienzo finalmente Mostramos la información sobre la época actual y el error.

**def train\_jacobian(self):**

Para poder realizar el entrenamiento con el método de Levenberg-Marquardt lo primero que tocó hacer fue declarar el número de muestras (filas) el cual es igual al número de puntos de entrada (input); en seguida declaramos el tamaño de la matriz jacobiana usando el tamaño de los pesos de la capa oculta (hidden layer) más el tamaño de los pesos de la capa de salida (output layer) para con esto declarar la matriz jacobiana e inicializarla con ceros. El resultante lo obtendremos

del producto punto del (número de puntos de entrada por el tamaño de la matriz jacobiana) además fue necesario usar banderas para el error (para gui)

Continuando con el entrenamiento necesitamos la obtención de los puntos de entrada (input), bias + x + y para después poder calcular la salida de la capa oculta (hidden layer) y la capa de salida (output layer)

Para controlar la cantidad de corrección aplicada en cada iteración. declaramos el factor lambda para el método de newton en 0.1 porque es un buen valor para empezar.

Usamos un condicional para realizar una parada brusca en una época o error y poder hacer todo el proceso para la propagación hacia adelante. Dentro del ciclo nos Fue necesario calcular la salida de la capa oculta y la salida de la capa de salida usando los pesos actuales de la red neuronal, calculamos el error por medio de resultado deseado menos el resultado obtenido con los pesos actuales de la red neuronal

A continuación llenamos la matriz jacobiana y para esto es necesario recorrer todos los puntos de entrada usando un ciclo For. Dentro de este ciclo lo queharemos parte desde el punto en que En redes neuronales la jacobiana es la derivada parcial del error/output sobre todos los inputs para llenar esta matriz será necesario el usar la regla de la cadena, para esto declaramos las variables dv\_dy\_dl\_dy para calcular el error relativo sobre los pesos de acuerdo con que tanto UH participan en el output por así decirlo. DL\_dy es la derivada de la función de pérdida con respecto a la salida, dy\_dv es la derivada de la función de activación con respecto a la salida ( $dy = f'(v)$ ), dv\_dw es la derivada de la salida con respecto a los pesos adicionalmente usamos la variable dl\_dw para definir la derivada de la función de pérdida con respecto a los pesos y a la función de activación por la derivada de la función de activación con respecto a la producción por la derivada de la producción con respecto a los pesos

con estas derivadas podemos llenar la matriz jacobiana para la capa de salida (output layer) que es igual a la derivada de la función de pérdida con respecto a los pesos de la capa de salida (output layer)

para continuar con el entrenamiento nos fue necesario declarar dv/dh que es la derivada de la salida con respecto a la capa oculta que es igual a los pesos de la capa de salida (output Layer) transpuestos esto lo hacemos con el fin de ayudar a calcular la matriz jacobiana para la capa oculta

seguido de esto usamos un ciclo For para entrar en cada neurona y actualizar todos los datos de la matriz jacobiana además de cada peso en las neuronas de la capa oculta declaramos dh/dvh que nos ayudará a definir la derivada de la función de activación relativa a la capa oculta

continuamente declaramos dvh/de la cual nos ayudará a definir la derivada de la capa oculta relativa a los pesos esta es la derivada de la función de pérdida con respecto a los pesos

posteriormente declaramos dla/dw la cual nos ayudará a definir la derivada de la función de pérdida con respecto a la función de activación por la derivada de la función de activación con respecto a la salida por la derivada de la salida con respecto a la capa oculta por la derivada de la capa oculta con respecto a los pesos

finalmente lo último que haremos en este ciclo for es asignar a jacobian la derivada de la función de pérdida con respecto a los pesos de la capa oculta (hidden Layer)

una vez completa la matriz jacobiana continuamos con la fórmula que define el método de Levenberg-Marquardt es por eso por lo que necesitamos fuera del ciclo for generar la transpuesta de la matriz jacobiana  $jacobian\_T = jacobian.T$  ( $j \times m$ ) -> ( $m \times j$ ) junto con la matriz identidad de tamaño ( $m \times m$ ) donde  $m$  es el número de pesos en la matriz jacobiana ( $jacobian\_T$ ) así como el gradiente descendiente a partir del factor Lambda por la matriz identidad ( $m \times m$ )

así como declarar la matriz hessiana a partir del producto punto de la matriz jacobiana transpuesta ( $jacobian\_T$ ) y la matriz jacobiana ( $jacobian$ ) más el gradiente descendiente ( $gradient\_descent$ ) ( $m \times j$ ) \* ( $j \times m$ ) + ( $m \times m$ ) = ( $m \times m$ )

generamos la inversa de la matriz hessiana de ( $m \times m$ ), usamos la función linear algebra de numpy para calcular la inversa de la matriz hessiana ( $hessian$ )

es importante mencionar que usamos la jacobiana para llenar la matriz hessiana ya que el implementar la hessian es muy lento y algo complicado

finalmente declaramos delta para actualizar los pesos de la red neuronal a partir del producto punto de la inversa de la matriz hessiana ( $hessian\_inv$ ) y la matriz jacobiana transpuesta ( $jacobian\_T$ ) y el error ( $errors$ ) transpuesto ( $T$ ) ( $m \times m$ ) \* ( $m \times j$ ) \* ( $j \times 1$ ) = ( $m \times 1$ )ya una vez hecho esto convertimos delta a un array de una dimensión ( $m \times 1$ ) -> ( $m \times 1$ ) para poder actualizar los pesos de la red neuronal , seguido de esto calculamos el error cuadrático medio (average of the square of the errors) para saber si el entrenamiento debe parar junto a el error nos apoyamos de un contador para actualizar los pesos de la red neuronal

usamos un ciclo for para la actualización de cada peso en la capa de salida (output layer); dentro de este for actualizamos los pesos de la capa de salida (output layer) según el método de newton = peso -  $\delta_{1d}[i]$  y aumentamos el contador para actualizar los pesos de la red neuronal

implementamos otro for para cada neurona en la capa oculta (hidden layer) adentro añadimos otro for para cada peso en la capa oculta (hidden layer) todo

esto con el fin de actualizar los pesos de la capa oculta (hidden layer) según el método de newton = peso - delta\_1d[caunter] de igual forma aumentamos el contador para actualizar los pesos de la red neuronal

afuera del ciclo for calculamos la salida de la capa oculta y la salida de la capa de salida con los nuevos pesos de la red neuronal con la función cal\_hidden en esta función lo que hacemos es declaramos la salida de la capa oculta (hidden layer) a partir de la función de activación y los puntos de entrada (input) (x, y) y los pesos de la capa oculta (hidden layer)  $hidden\_y = f(x \cdot w)$  <-  $y = f(v)$   $hidden\_layer = f(input \cdot weights)$ , declaramos la salida de la capa oculta en una matriz de una dimensión (hidden\_x) a partir de la salida de la capa oculta (hidden layer)  $hidden\_x = [1, hidden\_y]$  <- bias + hidden\_y finalmente en esta función declaramos "y" a partir de la función de activación y la salida de la capa oculta (hidden layer) y los pesos de la capa de salida (output layer)  $y = f(hidden\_x \cdot w)$  <-  $y = f(v)$   $output\_layer = f(hidden\_layer \cdot weights)$

Una vez hecho el cálculo de los pesos de la capa de salida y oculta continuamos calculando el error (deseado - obtenido) con los nuevos pesos de la red neuronal

calculamos el error cuadrático medio (average of the square of the errors) para saber si el entrenamiento debe parar con los nuevos pesos de la red neuronal

si el error cuadrático medio (average of the square of the errors) con los nuevos pesos de la red neuronal es mayor que el error cuadrático medio (average of the square of the errors) con los pesos anteriores de la red neuronal asignaremos el error cuadrático medio con los nuevos pesos de la red neuronal al error antiguo

proseguimos aumentando el factor Lambda para el método de newton para que el entrenamiento sea más preciso

si el error cuadrático medio con los nuevos pesos de la red neuronal es menor que el error cuadrático medio con los pesos anteriores de la red neuronal entonces disminuimos el factor lambda para el método de newton para que el entrenamiento sea más rápido

usamos otro condicional si para comprobar si el error cuadrático medio es mayor que el error mínimo para parar el entrenamiento convirtiendo error a verdadero

si el error cuadrático medio es menor que el error mínimo para parar el entrenamiento convertimos el error a falso después d esto agregamos el error cuadrático medio a la lista de errores

finalmente imprimimos los resultados con la ayuda de la función plotting\_jacob para con esto

graficamos los puntos de entrada (input) (x,y) y la salida de la capa oculta (hidden layer) y la salida de la capa de salida (output layer) mostrando cómo clasifica y disminuye el error finalmente nos queda aumentar el número de épocas

(iteraciones) actuales y disminuir el número de épocas (iteraciones) hasta llegar a la respuesta deseada con el error mínimo declarado

### **plotting\_jacob(self):**

Esta función se encarga de trazar un gráfico interactivo

Lo primero que hacemos es limpiar el gráfico con self.ax.cla la cual borra cualquier contenido previo del gráfico para prepararse para la nueva visualización. después calculamos las salidas de la red con la siguiente línea `_, _, y = self.calc_hidden` la cual lo que hace es con la función `calc_hidden` donde devuelve el cálculo de los pesos de la capa de salida y de la capa oculta para después regresando a la función `plotear` los puntos según las salidas (for i in range(len(np.array(y).flatten()))) para después iterar sobre las salidas de la red y plotea puntos rojos o azules dependiendo de si el valor de la salida es mayor o igual a 0.5.

Después esto creamos un meshgrid para visualizar el espacio de entrada (`x_v, y_v, meshX, meshY`): Esto lo utilizamos para crear una visualización del espacio de entrada. A continuación calculamos las predicciones de la red para el meshgrid (for i in range(len(meshX))) donde utilizamos la red neuronal para obtener las predicciones en el espacio de entrada definido por el meshgrid.

Adicionalmente necesitábamos visualizar el espacio de entrada y las predicciones con (self.ax.contourf): aquí utilizamos `contourf` para crear un gráfico de contorno y relleno que muestra cómo la red clasifica diferentes regiones del espacio de entrada. Después de esto actualizamos la visualización con (self.canvas.draw(), self.canvas\_e.draw()) para refrescar la visualización en la interfaz gráfica. Y finalmente actualiza la información sobre la época y el error con (self.label\_n\_epoch.config, self.label\_error.config): Mostrando la información sobre la época actual y el error en etiquetas de la interfaz gráfica.

## **Código de la práctica:**

```

import matplotlib.pyplot as plt
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
from tkinter import *
import numpy as np
import threading

# Multi layer neural network

Alejandro-Sedano, hace 21 minutos | 3 authors (Hirane and others)
class LMNN: Alejandro-Sedano, hace 18 minutos • modificación en la posición botones

    def __init__(self, eta=0.1, epoch=1500, min_error=0.01, n_neurons=6):

        self.lr = eta #aprendizaje rate (learning rate)
        self.epoch = epoch #numero de epochas (iteraciones)
        self.n_epoch = 0 #numero de epochas (iteraciones) actuales (para gui)
        self.min_error = min_error #error minimo para parar el entrenamiento
        self.n_neurons = n_neurons #numero de neuronas en la capa oculta (hidden layer)
        # init weights
        self.w_hidden = np.matrix(np.random.rand(self.n_neurons, 3)) #pesos de la capa oculta (hidden layer)
        self.w_output = np.random.rand(self.n_neurons+1) #pesos de la capa de salida (output layer) +1 por el bias (sesgo)
        # X is the points, d is the desired output
        # keep track of errors
        self.X = [] #puntos de entrada (input) (x,y)
        self.d = [] #salida deseada (desired output) (0,1) (1,0) (0,0) (1,1)
        self.errors = [] #errores de cada epocha (iteracion) (para gui)
        # gui stuff
        self.fig_e, self.ax_e = plt.subplots() #figura y ejes para el error
        self.fig, self.ax = plt.subplots() #figura y ejes para el grafico
        self.canvas = None #canvas para el grafico
        self.canvas_e = None #canvas para el error

    def set_canvas(self):
        # set window
        mainwindow = Toplevel() #ventana principal
        mainwindow.wm_title("Levenberg-Marquardt") #titulo de la ventana
        mainwindow.geometry("1080x720") #tamano de la ventana
        #add image to window
        img = PhotoImage(file="bg.png") #imagen de fondo
        img_label = Label(mainwindow, image=img, bg='white') #label para la imagen de fondo
        img_label.place(x=0, y=0, relwidth=1, relheight=1) #posicion de la imagen de fondo

        # add canvas

        self.canvas = FigureCanvasTkAgg(self.fig, master=mainwindow)
        self.canvas.get_tk_widget().place(x=520, y=120, width=580, height=580) #posicion del grafico en la ventana principal
        self.fig.canvas.mpl_connect('button_press_event', self.set_dots) #evento para agregar puntos al grafico con el mouse

```

```

# error canvas

self.canvas_e = FigureCanvasTkAgg(self.fig_e, master=mainwindow) #canvas para el error
self.canvas_e.get_tk_widget().place(x=20, y=70, width=300, height=200)
execute_button = Button(mainwindow, text="Train Function", command=lambda: threading.Thread(target=self.train).start())
execute_button.place(x=10, y=350)
jacobian_button = Button(mainwindow, text="Train Jacobian", command=lambda: threading.Thread(target=self.train_jacobian).start())
jacobian_button.place(x=10, y=380)
title = Label(mainwindow, text="Levenberg-Marquardt", font=("Helvetica", 16))
title.place(x=400, y=10)
# reset button
reset_button = Button(mainwindow, text="Reset", command=lambda: self.clear_data(), width=11)
reset_button.place(x=10, y=410)
self.set_axis()
self.label_n_epoch = Label(mainwindow, text="Epoch: 0")
self.label_n_epoch.place(x=10, y=300)
self.label_error = Label(mainwindow, text="Error: 0")
self.label_error.place(x=10, y=320)

mainwindow.mainloop()

# Funcion para resetear la interfaz
def clear_data(self):
    # Clear existing data
    self.X = []
    self.d = []
    self.errors = []
    self.n_epoch = 0
    self.epoch = 1500
    self.n_neurons = 6
    self.w_hidden = np.matrix(np.random.rand(self.n_neurons, 3))
    self.w_output = np.random.rand(self.n_neurons+1)
    self.label_n_epoch.config(text="Epoch: 0")
    self.label_error.config(text="Error: 0")
    self.ax.cla()
    self.ax_e.cla()
    self.set_axis()
    self.canvas.draw()
    self.canvas_e.draw()
    self.fig.canvas.mpl_connect('button_press_event', self.set_dots)
    self.fig_e.canvas.mpl_connect('button_press_event', self.set_dots)
    self.canvas.draw()
    self.canvas_e.draw()
    #self.label_n_epoch.config(text="Epoch: "+str(self.n_epoch))
    #self.label_error.config(text="Error: "+str(self.errors[-1]))

```

```

# set dots on canvas and add to X and d arrays for training later
def set_dots(self, event):

    ix, iy = event.xdata, event.ydata #posicion del mouse en el grafico
    self.X.append((ix, iy)) #agregamos el punto a la lista de puntos de entrada (input)
    if event.button == 1: #si el boton izquierdo del mouse es presionado
        self.d.append(1) #agregamos el punto a la lista de salidas deseadas (desired output)
        self.ax.plot(ix, iy, marker='o', color='red') #graficamos el punto en el grafico
        #self.ax.plot(ix, iy, '.r')
    elif event.button == 3:
        self.d.append(0) #agregamos el punto a la lista de salidas deseadas (desired output)
        self.ax.plot(ix, iy, marker='x', color='blue')
        #self.ax.plot(ix, iy, '.b')
    self.canvas.draw()

def activation(self, x, w):

    a = 1
    v = np.dot(x, w)
    f = 1/(1+np.exp(-a*v))
    return f

def fd_activation(self, Y):

    a = 1 #factor de activacion
    f = a*Y*(1-Y) #derivada de la funcion de activacion (sigmoid)
    return f

def calc_hidden(self):

    print("X.shape", self.X.shape, "w_hidden.shape", np.transpose(self.w_hidden).shape)
    hidden_y = self.activation(self.X, np.transpose(self.w_hidden)) # y = f(X*w) <- y = f(v)
    # wX+b <-
    hidden_x = np.c_[np.ones(len(hidden_y)), hidden_y] # x = [1, y] <- bias + y
    y = self.activation(hidden_x, np.array(self.w_output).flatten()) # y = f(x*w) <- y = f(v)
    return hidden_y, hidden_x, y

def train(self):

    #we are going to try to calculate the jacobian matrix while training
    num_samples = len(self.X) # number of samples (rows)
    jacobian_size = self.w_hidden.size + self.w_output.size # size of the jacobian matrix (number of weights) + 1 for the bias
    print("jacobian size", jacobian_size)
    jacobian = np.zeros((num_samples, jacobian_size))
    # going for a hard stop on epoch or error
    error = True #error flag (para gui)

    self.X = np.matrix(self.X) #puntos de entrada (input) (x,y)
    # weights already initialized
    while self.epoch and error:

        error = False
        #forward propagation
        hidden_y, hidden_x, y = self.calc_hidden() #calculamos la salida de la capa oculta y la salida de la capa de salida
        # calculate error
        errors = np.array(self.d) - y #calculamos el error (deseado - obtenido)
        # calculate delta
        delta_output = [] #delta de la capa de salida
        #back propagation
        for i in range(len(hidden_x)): #para cada punto de entrada (input)
            dy = self.fd_activation(np.array(y).flatten()[i]) #calculamos la derivada de la funcion de activacion de la capa de salida
            delta_output.append(dy*np.array(errors).flatten()[i]) #this is the gradient of the loss function with respect to the output

        self.w_output = self.w_output + np.dot(hidden_x[i], self.lr*delta_output[-1]) #actualizamos los pesos de la capa de salida (output layer) lr*delta_output[-1] es el ultimo
        for i in range(len(self.X)):
            for j in range(len(self.w_hidden)):
                dy = self.fd_activation(hidden_y[i, j])
                #derivada para delta
                hidden_delta = np.array(self.w_output).flatten()[i]*np.array(delta_output).flatten()[i]*dy # delta = w*delta_output*dy
                #sumamos deltas siguiendo porque es hidden
                #actualizamos valores
                self.w_hidden[j] = self.w_hidden[j] + np.dot(self.X[i], self.lr*hidden_delta)

        # check error
        square_error = np.average(np.power(errors, 2))

        if square_error > self.min_error:
            error = True
        self.errors.append(square_error)

        hidden_y, hidden_x, y = self.calc_hidden()
        self.plotting(Y)
        self.epoch -= 1
        self.n_epoch += 1

def train_jacobian(self):

    num_samples = len(self.X) # numero de muestras (filas) es igual al numero de puntos de entrada (input)
    jacobian_size = self.w_hidden.size + self.w_output.size # el tamaño de la matriz jacobiana es igual al tamaño de los pesos de la capa oculta (hidden layer) + el tamaño de los pesos de la capa de salida (output layer)
    jacobian = np.zeros((num_samples, jacobian_size)) # la matriz jacobiana es una matriz de ceros de tamaño (numero de puntos de entrada por el tamaño de la matriz jacobiana)

```

```

error = True
self.X = np.matrix(self.X) #puntos de entrada (input) (x,y)
lambda_factor = 0.1 #factor lambda para el metodo de newton
while self.epoch and error:
    error = False
    #forward propagation
    hidden_y, hidden_x, y = self.calc_hidden() #calculamos la salida de la capa oculta y la salida de la capa de salida
    # calculate error
    errors = np.array(self.d) - y #calculamos el error (deseado - obtenido)
    print("errors", errors)

    for n in range(num_samples):
        # jacobian matrix filling
        #dl_dy = -2 * (self.d[n]-np.array(y).flatten())[n] # this is dl/dy which is the derivative of the loss function with respect to output
        dl_dy = -1
        dy_dv = self.fd_activation(np.array(y).flatten())[n] # this is dy/dv which is the derivative of the activation function with respect to the output
        dv_dw = hidden_x[n] # this is dv/dw which is the derivative of the output with respect to the weights
        dl_dw = dl_dy*dy_dv*dv_dw# this is the derivative of the loss function with respect to the weights

        # jacobian matrix
        jacobian[n, :self.w_output.size] = dl_dw.flatten()
        #hidden layer
        # we need dl_dy, dy_dv

        dv_dh = np.transpose(self.w_output) # this is dw/dh which is the derivative of the output with respect to the hidden layer
        # for each hidden neuron
        for i in range(self.n_neurons):

            dh_dvh = self.fd_activation(hidden_y[n,i])
            vh_dw = self.X[n]

            dl_dvh_hidden = dl_dy*dy_dv*dv_dh[i]*dh_dvh*dvh_dw
            jacobian[n, self.w_output.size+i*self.w_hidden.shape[1]:self.w_output.size+(i+1)*self.w_hidden.shape[1]] = dl_dvh_hidden.flatten()

        jacobian_T = np.transpose(jacobian)

        identity = np.identity(jacobian_T.shape[0])

        gradient_descent = lambda_factor*identity

        hessian = np.dot(jacobian_T, jacobian) + gradient_descent

        hessian_inv = np.linalg.inv(hessian)

```

```

        delta = np.dot(np.dot(hessian_inv, jacobian_T), errors.T)
        delta_1d = np.array(delta).flatten()
        #print("delta1d", delta_1d.shape)

        temp_w_output = self.w_output.copy()
        temp_w_hidden = self.w_hidden.copy()

        square_error = np.average(np.power(errors, 2))

        counter = 0

        for i in range(self.w_output.size):

            self.w_output[i] = self.w_output[i] - delta_1d[i]

            counter += 1
        for i in range(self.n_neurons):

            for j in range(self.X.shape[1]):

                self.w_hidden[i,j] = self.w_hidden[i,j] - delta_1d[counter]

                counter += 1
        hidden_y, hidden_x, y_new = self.calc_hidden()
        errors = np.array(self.d) - y_new
        square_error_new = np.average(np.power(errors, 2))
        if square_error_new > square_error:
            square_error = square_error_new
            lambda_factor *= 1.1
        else:
            lambda_factor *= 0.9

        if square_error > self.min_error:
            error = True
        else:
            error = False
        self.errors.append(square_error)

        self.plotting_jacob()
        self.n_epoch += 1
        self.epoch -= 1

    def plotting_jacob(self):

```

```

class LMNN:      Alejandro-Sedano, hace 18 minutos • modificacion en la posicion botones
def plotting_jacob(self):
    self.ax.cla()
    _, _, y = self.calc_hidden()

    for i in range(len(np.array(y).flatten())):
        if np.array(y).flatten()[i] >= 0.5:
            self.ax.plot(self.X[i,1], self.X[i,2],marker='o', color='red')
            #self.ax.plot(self.X[i,1], self.X[i,2],'.r')
        else:
            self.ax.plot(self.X[i,1], self.X[i,2],marker='x', color='blue')
            #self.ax.plot(self.X[i,1], self.X[i,2],'.b')

    x_v = np.linspace(-6, 6, 36) #valores de x para el meshgrid
    y_v = np.linspace(-6, 6, 36)
    meshX, meshY = np.meshgrid(x_v, y_v)
    meshZ = []
    for i in range(len(meshX)):
        xc = np.transpose([meshX[i], meshY[i]])
        xc = np.c_[np.ones(len(xc)), xc]
        hidden_y = self.activation(xc, np.transpose(self.w_hidden))
        hidden_x = np.c_[np.ones(len(hidden_y)), hidden_y]
        yc = self.activation(hidden_x, np.array(self.w_output).flatten())
        meshZ.append(np.array(yc).flatten())

    mapping = plt.get_cmap('viridis')
    self.ax.contourf(meshX, meshY, meshZ, cmap=mapping)
    self.ax_e.cla()
    self.ax_e.plot(self.errors, c='r')
    self.ax_e.set_xticklabels([])

    self.set_axis()

    self.canvas.draw()
    self.canvas_e.draw()
    self.label_n_epoch.config(text="Epoch: "+str(self.n_epoch))
    self.label_error.config(text="Error: "+str(self.errors[-1]))


def set_axis(self):

    aX = [-6,6]
    aY = [-6,6]
    self.ax.grid('on')
    self.ax_e.grid('on')
    zeros = np.zeros(2)
    self.ax.plot(aX, zeros, c='k')
    self.ax.plot(zeros, aY, c='k') # k

```

```

class LMNN:      Alejandro-Sedano, hace 18 minutos • modificación en la posición botones
    def set_axis(self):
        self.ax.plot(zeros, aY, c='k') # k
        plt.xlim(-6, 6)
        plt.ylim(-6, 6)

    def plotting(self,y):
        self.ax.cla()

        for i in range(len(np.array(y).flatten())):
            if np.array(y).flatten()[i] >= 0.5:
                #self.ax.plot(self.X[i,1], marker='o', color='red')
                self.ax.plot(self.X[i,1], self.X[i,2], '.r')
            else:
                #self.ax.plot(self.X[i,1], self.X[i,2], marker='o', color='blue')
                self.ax.plot(self.X[i,1], self.X[i,2], '.b')

            x_v = np.linspace(-6, 6, 36) #valores de x para el meshgrid (,36) es el numero de puntos en el meshgrid
            y_v = np.linspace(-6, 6, 36)
            # meshgrid
            meshX, meshY = np.meshgrid(x_v, y_v)
            meshZ = []
            for i in range(len(meshX)):
                xc = np.transpose([meshX[i], meshY[i]])
                xc = np.c_[np.ones(len(xc)), xc]
                hidden_y = self.activation(xc, np.transpose(self.w_hidden))
                hidden_x = np.c_[np.ones(len(hidden_y)), hidden_y]
                yc = self.activation(hidden_x, np.array(self.w_output).flatten())
                meshZ.append(np.array(yc).flatten())
            mapping = plt.get_cmap('viridis')
            self.ax.contourf(meshX, meshY, meshZ, cmap=mapping)
            self.ax_e.cla()
            self.ax_e.plot(self.errors, c='r')
            self.ax_e.set_xticklabels([])

            self.set_axis()

            self.canvas.draw()
            self.canvas_e.draw()
            self.label_n_epoch.config(text="Epoch: "+str(self.n_epoch))
            self.label_error.config(text="Error: "+str(self.errors[-1]))

import matplotlib.pyplot as plt

from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg

from tkinter import *

import numpy as np

import threading

# Multi layer neural network

```

```

class LMNN:

    def __init__(self, eta=0.1, epoch=1500, min_error=0.01,
n_neurons=6):

        self.lr = eta #aprendizaje rate (learning rate)

        self.epoch = epoch #numero de epochas (iteraciones)

        self.n_epoch = 0 #numero de epochas (iteraciones) actuales
(para gui)

        self.min_error = min_error #error minimo para parar el
entrenamiento

        self.n_neurons = n_neurons #numero de neuronas en la capa
oculta (hidden layer)

        # init weights

        self.w_hidden = np.matrix(np.random.rand(self.n_neurons, 3))
#pesos de la capa oculta (hidden layer)

        self.w_output = np.random.rand(self.n_neurons+1) #pesos de la
capa de salida (output layer) +1 por el bias (sesgo)

        # X is the points, d is the desired output

        # keep track of errors

        self.X = [] #puntos de entrada (input) (x,y)

        self.d = [] #salida deseada (desired output) (0,1) (1,0) (0,0)
(1,1)

        self.errors = [] #errores de cada epoca (iteracion) (para gui)

        # gui stuff

        self.fig_e, self.ax_e = plt.subplots() #figura y ejes para el
error

```

```

        self.fig, self.ax = plt.subplots() #figura y ejes para el
grafico

        self.canvas = None #canvas para el grafico

        self.canvas_e = None #canvas para el error


def set_canvas(self):

    # set window

    mainwindow = Toplevel() #ventana principal

    mainwindow.wm_title("Levenberg-Marquardt") #titulo de la
ventana

    mainwindow.geometry("1080x720") #tamano de la ventana

    # add image to window

    img = PhotoImage(file="img/bg2.png") #imagen de fondo

    img_label = Label(mainwindow, image=img, bg='white') #label
para la imagen de fondo

    img_label.place(x=0, y=0, relwidth=1, relheight=1) #posicion
de la imagen de fondo


    # add canvas

    self.canvas = FigureCanvasTkAgg(self.fig, master=mainwindow)

    self.canvas.get_tk_widget().place(x=520, y=120, width=580,
height=580) #posicion del grafico en la ventana principal

    self.fig.canvas.mpl_connect('button_press_event',
self.set_dots) #evento para agregar puntos al grafico con el mouse

```

```

# error canvas

                self.canvas_e = FigureCanvasTkAgg(self.fig_e,
master=mainwindow) #canvas para el error

                self.canvas_e.get_tk_widget().place(x=20, y=70, width=300,
height=200)

                execute_button = Button(mainwindow, text="Train
Function", command=lambda: threading.Thread(target=self.train).start())

                execute_button.place(x=10, y=350)

                jacobian_button = Button(mainwindow, text="Train
Jacobian", command=lambda:
threading.Thread(target=self.train_jacobian).start())

                jacobian_button.place(x=10, y=380)

                title = Label(mainwindow, text="Levenberg-Marquardt",
font=("Helvetica", 16))

                title.place(x=400, y=10)

                # reset button

                reset_button = Button(mainwindow, text="Reset",
command=lambda: self.clear_data(), width=11)

                reset_button.place(x=10, y=410)

                self.set_axis()

                self.label_n_epoch = Label(mainwindow, text="Epoch: 0")

                self.label_n_epoch.place(x=10, y=300)

                self.label_error = Label(mainwindow, text="Error: 0")

                self.label_error.place(x=10, y=320)

```

```

mainwindow.mainloop()

# Funcion para resetear la interfaz

def clear_data(self):

    # Clear existing data

    self.X = []

    self.d = []

    self.errors = []

    self.n_epoch = 0

    self.epoch = 1500

    self.n_neurons = 6

    self.w_hidden = np.matrix(np.random.rand(self.n_neurons, 3))

    self.w_output = np.random.rand(self.n_neurons+1)

    self.label_n_epoch.config(text="Epoch: 0")

    self.label_error.config(text="Error: 0")

    self.ax.cla()

    self.ax_e.cla()

    self.set_axis()

    self.canvas.draw()

    self.canvas_e.draw()

        self.fig.canvas.mpl_connect('button_press_event',
self.set_dots)

        self.fig_e.canvas.mpl_connect('button_press_event',
self.set_dots)

```

```

        self.canvas.draw()

        self.canvas_e.draw()

        #self.label_n_epoch.config(text="Epoch: "+str(self.n_epoch))

        #self.label_error.config(text="Error: "+str(self.errors[-1]))

# set dots on canvas and add to X and d arrays for training later

def set_dots(self, event):

    ix, iy = event.xdata, event.ydata #posicion del mouse en el
grafico

    self.X.append((1, ix, iy)) #agregamos el punto a la lista de
puntos de entrada (input)

    if event.button == 1: #si el boton izquierdo del mouse es
presionado

        self.d.append(1) #agregamos el punto a la lista de salidas
deseadas (desired output)

        self.ax.plot(ix, iy, marker='o', color='red') #graficamos
el punto en el grafico

        #self.ax.plot(ix, iy, '.r')

    elif event.button == 3:

        self.d.append(0) #agregamos el punto a la lista de salidas
deseadas (desired output)

        self.ax.plot(ix, iy, marker='x', color='blue')

        #self.ax.plot(ix, iy, '.b')

    self.canvas.draw()

```

```

def activation(self, x, w):

    a = 1

    v = np.dot(x, w)

    f = 1/(1+np.exp(-a*v))

    return f


def fd_activation(self, Y):

    a = 1 #factor de activacion

    f = a*Y*(1-Y) #derivada de la funcion de activacion (sigmoid)

    return f


def calc_hidden(self):

    #print("X.shape", self.X.shape, "w_hidden.shape",
    np.transpose(self.w_hidden).shape)

    hidden_y = self.activation(self.X,
    np.transpose(self.w_hidden)) # y = f(X*w) <- y = f(v)

    # wx+b <-

    hidden_x = np.c_[np.ones(len(hidden_y)), hidden_y] # x = [1,
    y] <- bias + y

    y = self.activation(hidden_x,
    np.array(self.w_output).flatten()) # y = f(x*w) <- y = f(v)

```

```

    return hidden_y, hidden_x, y

def train(self):

    #we are going to try to calculate the jacobian matrix while
    training

    num_samples = len(self.X) # number of samples (rows)

    jacobian_size = self.w_hidden.size + self.w_output.size # size
    of the jacobian matrix (number of weights) + 1 for the bias

    print("jacobian size", jacobian_size)

    jacobian = np.zeros((num_samples, jacobian_size))

    # going for a hard stop on epoch or error

    error = True #error flag (para gui)

    self.X = np.matrix(self.X) #puntos de entrada (input) (x,y)

    # weights already initialized

    while self.epoch and error:

        error = False

        #forward propagation

        hidden_y, hidden_x, y = self.calc_hidden() #calculamos la
        salida de la capa oculta y la salida de la capa de salida

        # calculate error

        errors = np.array(self.d) - y #calculamos el error
        (deseado - obtenido)

        # calculate delta

        delta_output = [] #delta de la capa de salida

```

```

#back propagation

    for i in range(len(hidden_x)): #para cada punto de entrada
(input)

        dy = self.fd_activation(np.array(y).flatten())[i]
#calculamos la derivada de la funcion de activacion de la capa de
salida

        delta_output.append(dy*np.array(errors).flatten()[i])
#this is the gradient of the loss function with respect to the output

#actualizamos pesos

        self.w_output = self.w_output + np.dot(hidden_x[i],
self.lr*delta_output[-1]) #actualizamos los pesos de la capa de salida
(output layer) lr*delta_output[-1] es el ultimo elemento de la lista
delta_output

        for i in range(len(self.X)):

            for j in range(len(self.w_hidden)):

                dy = self.fd_activation(hidden_y[i, j])
#derivada para delta

                hidden_delta = np.array(self.w_output).flatten()[j+1]*np.array(delta_output).flatten()
[i]*dy # delta = w*delta_output*dy

                #usamos delta siguiente porque es hidden

                #actualizamos valores

                self.w_hidden[j] = self.w_hidden[j] +
np.dot(self.X[i], self.lr*hidden_delta)

                # check error

square_error = np.average(np.power(errors, 2))

```

```

        if square_error > self.min_error:
            error = True
            self.errors.append(square_error)

hidden_y, hidden_x, y = self.calc_hidden()
self.plotting(y)
self.epoch -= 1
self.n_epoch += 1

def train_jacobian(self):
    num_samples = len(self.X) # numero de muestras (filas) es igual al numero de puntos de entrada (input)
    jacobian_size = self.w_hidden.size + self.w_output.size # el tamaño de la matriz jacobiana es igual al tamaño de los pesos de la capa oculta (hidden layer) + el tamaño de los pesos de la capa de salida (output layer)
    jacobian = np.zeros((num_samples, jacobian_size)) # la matriz jacobiana es una matriz de ceros de tamaño (numero de puntos de entrada por el tamaño de la matriz jacobiana)
    error = True

```

```

self.X = np.matrix(self.X) #puntos de entrada (input) (x,y)

lambda_factor = 0.1 #factor lambda para el metodo de newton

while self.epoch and error:

    error = False

    #forward propagation

        hidden_y, hidden_x, y = self.calc_hidden() #calculamos la
salida de la capa oculta y la salida de la capa de salida

    # calculate error

        errors = np.array(self.d) - y #calculamos el error
(deseado - obtenido)

        print("errors", errors)

    for n in range(num_samples):

        # jacobian matrix filling

            #dL_dy = -2 * (self.d[n]-np.array(y).flatten()[n]) #
this is dL/dy which is the derivative of the loss function with
respect to output

            dL_dy = -1

            dy_dv = self.fd_activation(np.array(y).flatten()[n]) #
this is dy/dv which is the derivative of the activation function with
respect to the output

            dv_dw = hidden_x[n] # this is dv/dw which is the
derivative of the output with respect to the weights

            dL_dw = dL_dy*dy_dv*dv_dw# this is the derivative of
the loss function with respect to the weights

        # jacobian matrix

```

```

jacobian[n, :self.w_output.size] = dL_dw.flatten()

#hidden layer

# we need dL_dy, dy_dv

dv_dh = np.transpose(self.w_output) # this is dv/dh
which is the derivative of the output with respect to the hidden layer

# for each hidden neuron

for i in range(self.n_neurons):

    dh_dvh = self.fd_activation(hidden_y[n,i])

    dvh_dw = self.X[n]

    dL_dw_hidden = dL_dy*dy_dv*dv_dh[i]*dh_dvh*dvh_dw

    jacobian[n,
    self.w_output.size+i*self.w_hidden.shape[1]:self.w_output.size+(i+1)*self.w_hidden.shape[1]] = dL_dw_hidden.flatten()

jacobian_T = np.transpose(jacobian)

identity = np.identity(jacobian_T.shape[0])

gradient_descent = lambda_factor*identity

```

```

hessian = np.dot(jacobian_T, jacobian) + gradient_descent

hessian_inv = np.linalg.inv(hessian)

delta = np.dot(np.dot(hessian_inv, jacobian_T), errors.T)

delta_1d = np.array(delta).flatten()

#print("delta1d", delta_1d.shape)

temp_w_output = self.w_output.copy()

temp_w_hidden = self.w_hidden.copy()

square_error = np.average(np.power(errors, 2))

counter = 0

for i in range(self.w_output.size):

    self.w_output[i] = self.w_output[i] - delta_1d[i]

    counter += 1

```

```

        for i in range(self.n_neurons):

            for j in range(self.X.shape[1]):

                self.w_hidden[i,j] = self.w_hidden[i,j] -
delta_1d[counter]

            counter += 1

hidden_y, hidden_x, y_new = self.calc_hidden()

errors = np.array(self.d) - y_new

square_error_new = np.average(np.power(errors, 2))

if square_error_new > square_error:

    square_error = square_error_new

    lambda_factor *= 1.1

else:

    lambda_factor *= 0.9

if square_error > self.min_error:

    error = True

else:

    error = False

self.errors.append(square_error)

self.plotting_jacob()

```

```

        self.n_epoch += 1

        self.epoch -= 1


def plotting_jacob(self):
    self.ax.cla()

    _, _, y = self.calc_hidden()

    for i in range(len(np.array(y).flatten())):
        if np.array(y).flatten()[i] >= 0.5:
            self.ax.plot(self.X[i,1], self.X[i,2],marker='o',
color='red')
            #self.ax.plot(self.X[i,1], self.X[i,2],'.r')

        else:
            self.ax.plot(self.X[i,1], self.X[i,2],marker='x',
color='blue')
            #self.ax.plot(self.X[i,1], self.X[i,2], '.b')

    x_v = np.linspace(-6, 6, 36) #valores de x para el meshgrid
    y_v = np.linspace(-6, 6, 36)
    meshX, meshY = np.meshgrid(x_v, y_v)
    meshZ = []
    for i in range(len(meshX)):
        xc = np.transpose([meshX[i], meshY[i]])

```

```

xc = np.c_[np.ones(len(xc)), xc]
hidden_y = self.activation(xc,
np.transpose(self.w_hidden))

hidden_x = np.c_[np.ones(len(hidden_y)), hidden_y]
yc = self.activation(hidden_x,
np.array(self.w_output).flatten())

meshZ.append(np.array(yc).flatten())


mapping = plt.get_cmap('viridis')

self.ax.contourf(meshX, meshY, meshZ, cmap=mapping)

self.ax_e.cla()

self.ax_e.plot(self.errors, c='r')

self.ax_e.set_xticklabels([])

self.set_axis()

self.canvas.draw()

self.canvas_e.draw()

self.label_n_epoch.config(text="Epoch: "+str(self.n_epoch))

self.label_error.config(text="Error: "+str(self.errors[-1]))


def set_axis(self):

```

```

aX = [-6,6]

aY = [-6,6]

self.ax.grid('on')

self.ax_e.grid('on')

zeros = np.zeros(2)

self.ax.plot(aX, zeros, c='k')

self.ax.plot(zeros, aY, c='k') # k

plt.xlim(-6, 6)

plt.ylim(-6, 6)

def plotting(self,y):

    self.ax cla()

    for i in range(len(np.array(y).flatten())):
        if np.array(y).flatten()[i] >= 0.5:
            #self.ax.plot(self.X[i,1], marker='o', color='red')
            self.ax.plot(self.X[i,1], self.X[i,2], '.r')
        else:
            #self.ax.plot(self.X[i,1], self.X[i,2], marker='o',
            color='blue')
            self.ax.plot(self.X[i,1], self.X[i,2], '.b')

```

```

x_v = np.linspace(-6, 6, 36) #valores de x para el meshgrid
(,36) es el numero de puntos en el meshgrid

y_v = np.linspace(-6, 6, 36)

# meshgrid

meshX, meshY = np.meshgrid(x_v, y_v)

meshZ = []

for i in range(len(meshX)):

    xc = np.transpose([meshX[i], meshY[i]])

    xc = np.c_[np.ones(len(xc)), xc]

    hidden_y = self.activation(xc,
np.transpose(self.w_hidden))

    hidden_x = np.c_[np.ones(len(hidden_y)), hidden_y]

    yc = self.activation(hidden_x,
np.array(self.w_output).flatten())

    meshZ.append(np.array(yc).flatten())

mapping = plt.get_cmap('viridis')

self.ax.contourf(meshX, meshY, meshZ, cmap=mapping)

self.ax_e.clear()

self.ax_e.plot(self.errors, c='r')

self.ax_e.set_xticklabels([])

self.set_axis()

```

```
    self.canvas.draw()

    self.canvas_e.draw()

    self.label_n_epoch.config(text="Epoch: "+str(self.n_epoch))

    self.label_error.config(text="Error: "+str(self.errors[-1]))


if __name__ == "__main__":
    lmnn = LMNN()

    lmnn.set_canvas()

    #mln.train_jacobian()
```

## Capturas del programa en ejecución.

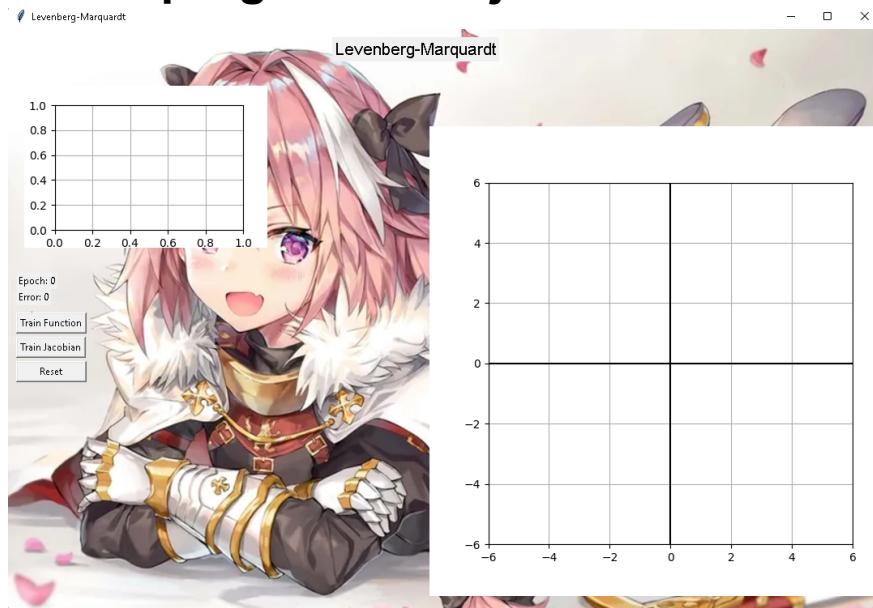


Ilustración 1 Despliegue del programa vista general

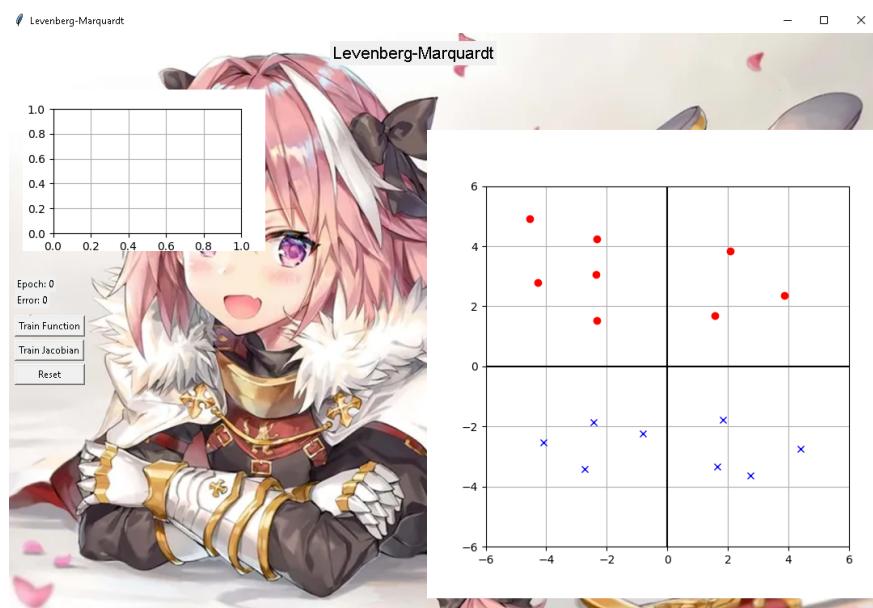
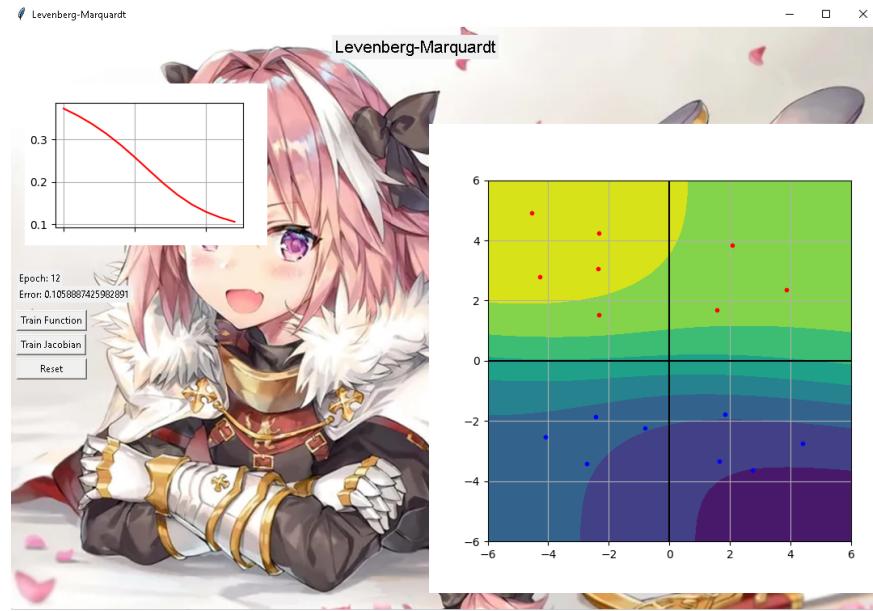
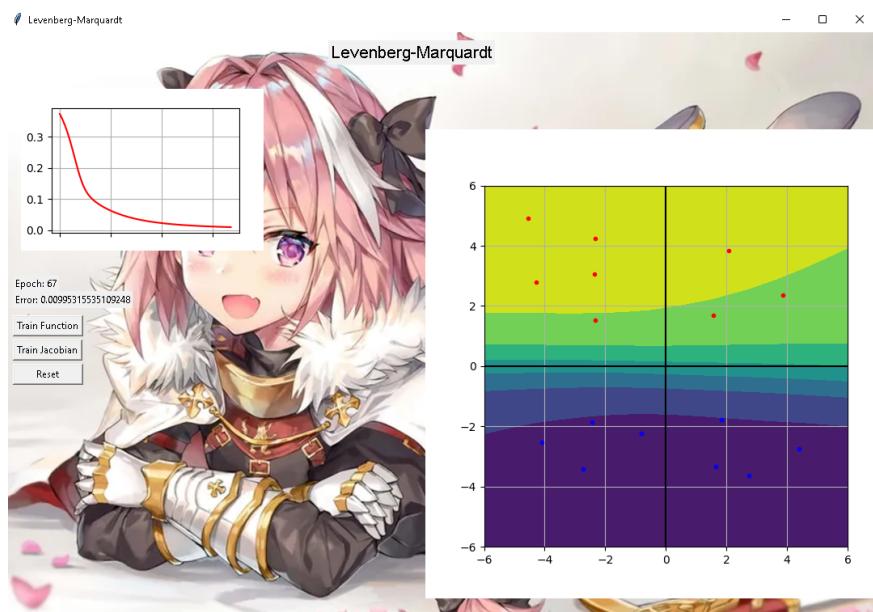


Ilustración 2 Inserción de los puntos por mouse



*Ilustración 3 Entrenamiento de la red neuronal con los datos ingresados*



*Ilustración 4 Resultado obtenido*

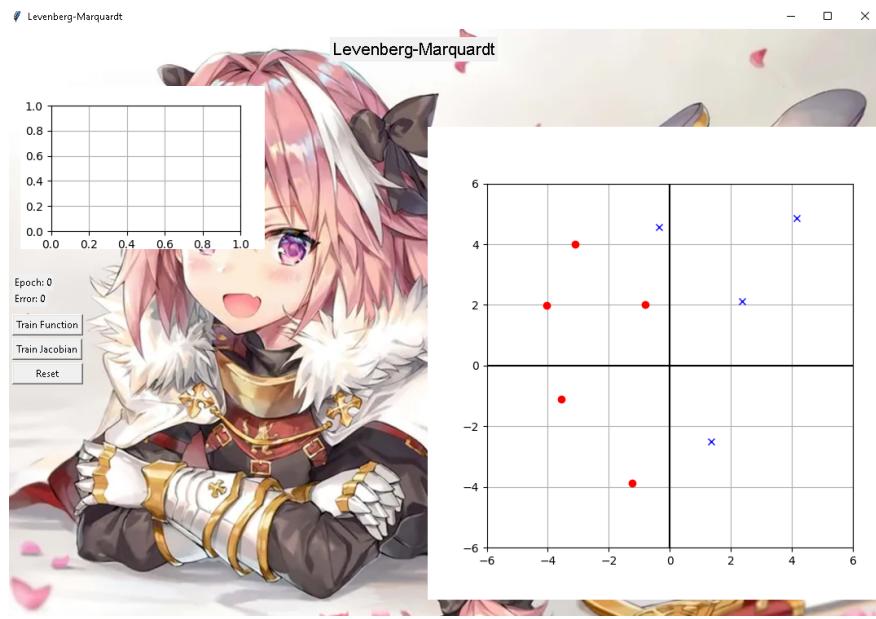


Ilustración 5 Inserción de puntos con el mouse

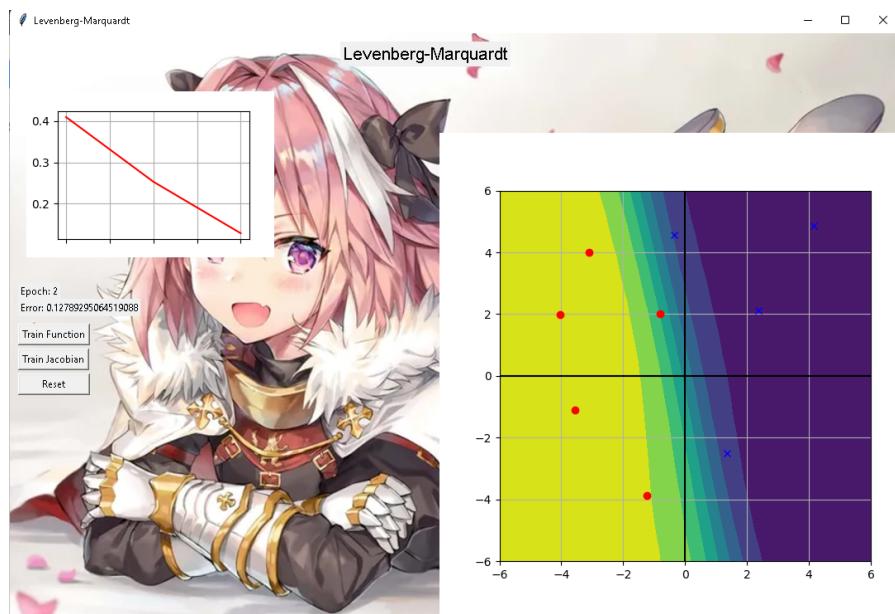


Ilustración 6 Inserción de nueva búsqueda con el entrenamiento jacobiano

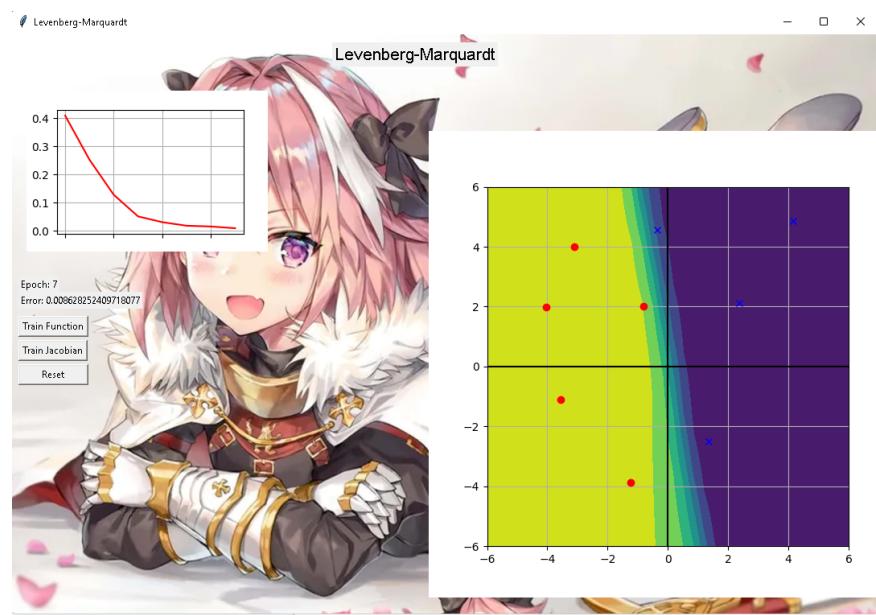


Ilustración 7 resultados obtenidos con el entrenamiento jacobiano

## Conclusión

El algoritmo de Levenberg-Marquardt sirve de mucho para resolver problemas relacionados a mínimos cuadrados no lineales ya que es la combinación de dos algoritmos más que son el descenso de gradiente y el gauss newton que combinados hacen que el número de épocas se reduzca drásticamente por la forma que se entrena usando lambda como variable que cambia de método dependiendo que tan cerca esté del error a su vez el uso de la matriz jacobiana para realizar esta tarea ya con ella se logra entrenar con respecto del número de datos que se le introduzca pero busca la mejor solución el implementarlo en python con una interfaz gráfica para comparar el tiempo de ejecución entre redes neuronales multicapa usando como base adaline y multicapa con el algoritmo de levenberg marquardt se puede notar una gran diferencia que demuestra el manejo de datos entre los dos algoritmos.

## Bibliografías:

- Engineering Educator Academy. (2021, 29 agosto). *Levenberg-Marquardt Algorithm* [Vídeo]. YouTube. <https://www.youtube.com/watch?v=2ToL9zUR8ZI>
- orvizar TV. (2021, 1 febrero). *LEVENBERG MARQUARDT | Optimización multidimensional* [Vídeo]. YouTube. <https://www.youtube.com/watch?v=U3UC499eOLw>
- colaboradores de Wikipedia. (2023b, octubre 21). *Matriz y determinante jacobianos*. Wikipedia, la encyclopedia libre. [https://es.wikipedia.org/wiki/Matriz\\_y\\_determinante\\_jacobianos](https://es.wikipedia.org/wiki/Matriz_y_determinante_jacobianos)
- Wikipedia contributors. (2023, 16 junio). *Levenberg–Marquardt Algorithm*. Wikipedia. [https://en.wikipedia.org/wiki/Levenberg%20%93Marquardt\\_algorithm](https://en.wikipedia.org/wiki/Levenberg%20%93Marquardt_algorithm)
- Wolfram Research, Inc. (s. f.). *Levenberg-Marquardt Method -- from Wolfram MathWorld*. <https://mathworld.wolfram.com/Levenberg-MarquardtMethod.html>
- Lopes, V. V., Rangel, C. M., & Novais, A. Q. (2011). Fractional-order transfer functions applied to the modeling of hydrogen PEM fuel cells. En *Computer-aided chemical engineering* (pp. 1748-1752). <https://doi.org/10.1016/b978-0-444-54298-4.50128-8>
- The Hessian*. (s. f.). khanacademy. <https://www.khanacademy.org/math/multivariable-calculus/applications-of-multivariable-derivatives/quadratic-approximations/a/the-hessian>
- Wikipedia contributors. (2023b, noviembre 8). *Hessian Matrix*. Wikipedia. [https://en.wikipedia.org/wiki/Hessian\\_matrix](https://en.wikipedia.org/wiki/Hessian_matrix)
- Hessian Matrix* | Brilliant Math & Science Wiki. (s. f.). <https://brilliant.org/wiki/hessian-matrix/>
- Wikipedia contributors. (2023c, noviembre 16). *Jacobian matrix and determinant*. Wikipedia. [https://en.wikipedia.org/wiki/Jacobian\\_matrix\\_and\\_determinant](https://en.wikipedia.org/wiki/Jacobian_matrix_and_determinant)
- Educative. (s. f.). *Educative Answers - trusted answers to developer questions*. <https://www.educative.io/answers/what-is-the-jacobian-matrix>