Universidad del País Vasco Euskal Herriko Unibertsitatea
BILBOKO INGENIARITZA ESKOLA
ESCUELA DE INGENIERÍA DE BILBAO

GRADO EN INGENIERÍA EN TECNOLOGÍA INDUSTRIAL

TRABAJO FIN DE GRADO

REDES NEURONALES EN MATLAB

Alumno: Zumárraga Eguidazu, Ignacio

Director: Anza Aguirrezabala, Juan José

Curso: 2018-2019

Fecha: Bilbao, 17, junio, 2019

Resumen

Este proyecto se basa en el estudio de los algoritmos que conforman una red neuronal artificial. Para ello se estudia primero la teoría matemática que hay detrás, partiendo de los métodos clásicos de aproximación de funciones y evolucionando a su adaptación en las redes, mostrando especial atención al algoritmo de Backpropagation.

Posteriormente se estudia su aplicación en Matlab, adaptándolos para sacar el máximo beneficio a las capacidades matriciales propias del programa y a la programación orientada a objeto, con que se desarrollan los contenidos de la red neuronal. Ver los algoritmos aplicados y su funcionamiento en los ejemplos permite comprenderlos mejor, así como poder observar las ventajas y desventajas de los distintos métodos.

Abstract

The main objective of this project is the study of the algorithms that form an artificial neural network. To do that, we will study, first, the mathematical theory behind them, starting from the classical methods of function approximation and evolving to their adaptation to the networks, paying special attention to the Backpropagation.

After that, we will study their application in Matlab, adapting them to get maximum benefit to the capacities of the program and getting to program a Neural Network object, which will enable us to solve simple examples. Seeing the algorithms applied and working may help to get a better understanding, and also enables to see the advantages and disadvantages of the different methods.

Laburpena

Proiektu hau oinarritzen da sare neuronal artifiziala osatzen duten algoritmoen estudioan. Horretarako, lehenik teoria matematikoa ikasten da, funtzio-hurbilketa metodo klasikotik hasiz eta sareetara bere egokitzera eboluzionatuz, Backpropagation arreta berezi erakusten.

Geroago Matlab-en aplikazioa ikasten da, programaren ahalmenei onura handiena ateratzeko moldatuz eta Sare Neuronala objektu bat programatzea lortuz, adibide errazei aurre egiteko. Aplikatutako algoritmoak eta adibideetan funtzionamendua ikustea, baimentzen du haiek hobeto ulertzea, baita ere metodo desberdinen abantailak eta desabantailak ikusi ahal izatea.

Contenido

Tabla de Ilustraciones

Lista de Acrónimos

RN: red neuronal

IA: inteligencia artificial

c: vector de pesos

BP: algoritmo de backpropagation

p: datos de entrada

t: datos de salida (objetivo del entrenamiento)

Q: número de muestras

a^m: salida de la capa m (a^M: salida de la última capa)

d^m: número de neuronas en la capa m (d^M: número de neuronas de la última capa)

e: vector de errores de una muestra

e_k: error de una muestra debido a la salida de la neurona k-ésima de la última capa

BP: algoritmo de Backpropagation

NW: inicialización mediante el algoritmo de Nguyen-Widrow.

LM: método de Levenberg-Marquardt

RB: regularización bayesiana

ES: early stopping

FF: red feedforward.

$\otimes$: producto de Kronecker

$$A \otimes B = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \otimes \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} a_{11} \cdot \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} & a_{12} \cdot \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} \\ a_{21} \cdot \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} & a_{22} \cdot \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} \end{pmatrix}$$

.*: producto elemento a elemento.

$$A.*B = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} .* \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} a_{1,1} \cdot b_{1,1} & \cdots & a_{1,n} \cdot b_{1,n} \\ \vdots & \ddots & \vdots \\ a_{m,1} \cdot b_{m,1} & \cdots & a_{m,n} \cdot b_{m,n} \end{pmatrix}$$

Introducción

Este documento contiene la propuesta de una red neuronal (RN) con Matlab que permita al lector conocer los algoritmos básicos de las redes neuronales, así como su funcionamiento al aplicarlo en ejemplos sencillos.

Una red neuronal es un modelo de computación cuya estructura de capas se asemeja a la estructura interconectada de las neuronas en el cerebro, con capas de nodos conectados. Una red neuronal puede aprender de los datos, de manera que se puede entrenar para que reconozca patrones, clasifique datos y pronostique eventos futuros[1].

A partir del trabajo realizado, definiría una red neuronal como un conjunto de algoritmos que permiten sustituir a los métodos numéricos en la aproximación de funciones (y otros métodos semejantes), reduciendo el número de operaciones necesarias y permitiendo el manejo masivo de datos.

Primero se enmarcarán las redes en su contexto, por qué surgen y su futuro esperado. Además, se expondrán los objetivos de este trabajo (de carácter docente) y su alcance. También se explicarán los beneficios que puede aportar. A continuación, se explicarán los conceptos teóricos detrás de las redes neuronales, los conceptos generales de una red y los algoritmos que la conforman.

Por último, se explicarán las fases en las que se ha divido el proyecto; el trabajo realizado (el objeto de Matlab: Red Neuronal), centrándonos en cómo se han adaptado los algoritmos para su programación; y una aplicación práctica del trabajo, comprobando los resultados y analizando las ventajas e inconvenientes de los distintos métodos.

[1] Definición de Matlab.

Contexto

Las redes neuronales surgen entre 1940 y 1950, en un intento de simular el comportamiento del cerebro del humano. Frente a la programación estructurada (clásica), en la que se diseña un programa a seguir para llegar a un objetivo, surge la programación declarativa, en la que no se indica la forma de llegar hasta el objetivo. Con las RN se pretende lograr esto, al buscar que la red aprenda por sí misma a partir de la experiencia.

Un ejemplo sería la red AlphaZero, una red diseñada por DeepMind para jugar al ajedrez. A base de jugar partidas, sin tener ninguna base de datos sobre la que basar sus decisiones, ha ido aprendiendo (en tan solo unas horas) hasta superar a cualquier otro motor o jugador. En palabras de Leontxo García[1]: *'Las cuatro partidas de AlphaZero que he publicado en mi columna de El País son joyas equiparables con las más brillantes de 1.500 años de historia documentada.'*

Si bien su origen se remonta a mediados del siglo pasado, no ha sido hasta este siglo que han irrumpido en la industria. La capacidad técnica de la época lo hacía imposible, el tiempo requerido (debido al número de cálculos) era impracticable. En 1970, Seppo Linnainmaa, desarrolla el algoritmo de backpropagation (BP), esto, junto con la capacidad creciente de los ordenadores, extendió su uso y permitió su desarrollo, siendo actualmente una de las herramientas de programación más potentes.

La era de la información está caracterizada por la cantidad ingente de información generada y como está transformando el mundo. Se generan más de 2500 gigabytes diarios[2], cantidad que desborda por completo la capacidad de un cerebro humano de procesarlo. Si bien es cierto que nadie pretende (ni tiene acceso, ni le sería útil) abordar toda esa información, si puede ayudarnos a entender que una empresa tiene a su disposición una cantidad descomunal de datos, a partir de la cual poder sacar información.

Las técnicas convencionales de análisis de datos se han visto sobrepasadas, no pudiendo sacar el máximo provecho a este recurso (que se antoja uno de los más importantes en la actualidad), así que se están viendo desplazadas en favor de las RN, constituyendo un elemento diferenciador en muchas empresas (entre las que destacan Google o Facebook).

[1] Periodista especializado en ajedrez, en una entrevista a RTVE por Samuel A. Pilar. Recuperado de: http://www.rtve.es/noticias/20171219/leontxo-garcia-nadie-ni-humano-ni-maquina-habia-jugado-tan-bien-ajedrez-como-alphazero/1648606.shtml

[2] Europapress, portaltic, Madrid 23/03/2017.
Recuperado de https://www.europapress.es/portaltic /internet/noticia-cada-dia-generan-2500-millones-gb-datos-ibm-crea-plataforma-empresas-aprovechen-20170323162319.html.

En la actualidad, su capacidad de manejar datos y aprender a partir de ellos, están permitiendo su aplicación en los siguientes campos:

- Reconocimiento de patrones. La capacidad de una red para reconocer patrones las convierte en la herramienta ideal para el reconocimiento de imágenes o de texto. En Cabify, por ejemplo, han desarrollado una red capaz de reconocer el texto de las quejas de los clientes y clasificarlos para su posterior análisis.
- Predicción. La capacidad de las RN de simular comportamientos dinámicos permite la predicción de comportamientos del mercado financiero o la meteorología, entre otros.
- Modelizado. Una RN sencilla puede ajustar prácticamente cualquier función a partir de un conjunto de datos, de ahí que se pueda utilizar para modelizar sistemas físicos, elemento indispensable en ciencia o ingeniería.
- Clasificación. Otro uso de las RN es el de discernir a que grupo pertenece una muestra mediante comparación. Se puede utilizar, por ejemplo, en la biopsia de tumores[1], permitiendo un diagnostico más rápido.
- Control. La principal aplicación industrial de las RN, permitiendo sustituir a los controladores digitales clásicos, aplicando funciones más complejas o convirtiéndose en controladores adaptativos.
- Inteligencia Artificial (IA). La gran razón de ser de las RN, siendo la base fundamental de las IA. Aunque todavía es un campo en desarrollo, ya está demostrando la capacidad de superar[2] al ser humano en ciertos aspectos (sea AlphaZero ejemplo de ello).

[1]Qing Zhou, Zhiyong Zhou, Chunmiao Chen, Gouhua Fan, Guangqiang Chen, Haiyan Heng, Jiansong Ji. Grading of heptocellular carcinoma using 3D SE-DenseNet in dynamic enhanced MR images. Computers in Biology and Medicine, Volumen 107, Abril 2019, páginas 47-57.

[2] No pretendo insinuar que la IA fuerte (inteligencia real) sea posible, pero sí que se han logrado avances extraordinarios en áreas que antes se pensaban que pertenecían solo al ser humano.

Objetivo y Alcance

El objetivo principal de este trabajo es servir como un primer acercamiento a las redes neuronales, entendiendo los algoritmos básicos que las forman y su funcionamiento en ejemplos prácticos. Para lograrlo se buscará lograr los siguientes objetivos parciales:

- Programación de los algoritmos en Matlab. Utilizando Matlab se intentará aplicar los algoritmos estudiados en [1], de una forma sencilla y comprensible para el lector no iniciado, frente a la descripción más teórica del libro que puede resultar engorroso.
- Formación de una RN. Una vez se ha logrado una comprensión teórica de los algoritmos se pueden analizar sus ventajas y desventajas frente a los métodos más clásicos. Para ello, programaremos un objeto de Matlab que permita aplicar distintas variantes de las RN.

  Cabe remarcar que el objetivo no es sustituir la toolbox existente de Matlab, sino, facilitar su comprensión. En un primer momento la toolbox de Matlab puede resultar una 'caja negra', siendo difícil comprender que está ocurriendo (debido a que no proporciona las fuentes de los algoritmos básicos). Con nuestra red se pretende 'dar luz', haciéndola accesible y manipulable, pudiendo observar los algoritmos matemáticos y resultados en pasos intermedios.
- Aplicación práctica de la red. A partir de la red programada se podrán abordar ejemplos prácticos, comparando los resultados ofrecidos por los distintos algoritmos y observando las ventajas ofrecidas por esta herramienta frente a los métodos clásicos.

Además del objetivo principal, encontramos otro secundario, servir como base de futuros trabajos en este tema, ya que como se explica a continuación, se consideran las RN como una herramienta que será indispensable en el futuro de un ingeniero. Con este fin, la RN programada se hará con estrucuctura de objeto, permitiendo una mayor organización y hacer cambios sin alterar el trabajo previo. Además, también se buscará la mayor eficiencia posible, prestando especial énfasis en la capacidad matricial de Matlab.

Beneficios

Como ya se ha explicado en el apartado anterior el marco de este trabajo es docente y los beneficios serán, por tanto, de carácter educativo.

Si bien nadie duda de la necesidad de un ingeniero de tener una base de álgebra, cálculo diferencial o cálculo numérico, no es (o será) menor la necesidad de saber acerca de redes neuronales. La aplicación de las RN se está extendiendo rápidamente en los últimos años, abarcando campos muy variados: finanzas, meteorología, medicina… en la ingeniería su aplicación se extiende principalmente en robótica y control de procesos. Pero, en la medida que avanza la Industria 4.0, los datos cobran una mayor relevancia en los procesos industriales y las redes neuronales son, posiblemente, la mejor herramienta para su procesado.

El estudio de las redes neuronales está, por el momento, destinado a los ingenieros informáticos, por su gran aplicación en IA, pero, como acabamos de decir, debería extenderse al resto de ingenierías. Por ello, el beneficio principal de este trabajo se basa en poder servir como punto de introducción, a partir del cual se pueda seguir trabajando y formándose.

Preliminares Matemáticos

Nos ocuparemos en esta sección de los aspectos matematicos que permitirán describir el funcionamiento de una red neuronal y su capacidad de aprender mediante entrenamiento cuando se dispone de un conjunto de datos experimentales en forma de entradas y salidas correspondientes al sistema estudiado mediante la RN.

Por su analogía con el problema de aproximación de funciones, que es una de las aplicaciones de las RN (base también para los problemas de clasificación y predicción), recordaremos inicialmente el problema de la aproximación lineal, para abordar a continuación la no lineal, propia de las RN, y describir los métodos de resolución más empleados.

Dado un conjunto discreto de datos

$$(x_1, y_1), \ldots, (x_n, y_n)$$

donde las abscisas $\{x_k, k = 1,..., n\}$ representan la variable independiente, y las ordenadas $\{y_k, k = 1,..., n\}$ una medida asociada. Interesa entonces determinar los parámetros c_i de una función $y = f(x, c_1, ..., c_m)$ que aproxime los datos

$$y_k \approx f(x_k, c_1, ..., c_m) \quad , \quad k = 1, ..., n \quad , \quad n > m \tag{1}$$

donde, en general, el número n de observaciones será mayor que el número de parámetros m.

Una RN proporciona la función $f(x_k, c_1, ..., c_m)$ a partir de la entrada x_k y establecido el valor de los parámetros **c**.

El sistema de ecuaciones (1) estará sobredeterminado y no tendrá solución exacta, lo cual indica que la función aproximante no pasa exactamente por los puntos $\{(x_k, y_k)(k = 1, ..., n)\}$. Como criterio de aproximación utilizaremos en este trabajo el ajuste mínimo-cuadrático.

El problema simple de aproximación lineal

En la aproximación lineal, dados n puntos $\{(x_k, y_k)(k=1,\ldots,n)\}$ y m funciones linealmente independientes $\{f_j(x)(j=1,\ldots,m)\}$, se trata de encontrar m coeficientes $\{c_i\}$ tales que la función f (x) definida como la combinación lineal:

$$f(x) = \sum_{i=1}^{m} c_i f_i(x)$$

minimice la suma de los cuadrados de los errores cometidos en cada punto:

$$E_c(c_1, c_2, \ldots, c_m) = \sum_{k=1}^{n} (y_k - f(x_k))^2 \tag{2}$$

Obtendremos así la solución aproximada del sistema sobredeterminado:

$$\underbrace{\begin{pmatrix} f_1(x_1) & f_2(x_1) & \cdots & f_m(x_1) \\ f_1(x_2) & f_2(x_2) & \cdots & f_m(x_2) \\ \vdots & \vdots & \ddots & \vdots \\ f_1(x_n) & f_2(x_n) & \cdots & f_m(x_n) \end{pmatrix}}_{A} \underbrace{\begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_m \end{pmatrix}}_{\mathbf{c}} \approx \underbrace{\begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{pmatrix}}_{\mathbf{b}} \equiv A\mathbf{c} \approx \mathbf{b} \tag{3}$$

donde las dimensiones de la matriz de coeficientes A y del vector **b** son n×m y n×1 respectivamente.

La solución aproximada **c** del sistema (3) que minimiza el error cuadrático (2) se obtiene resolviendo el sistema lineal de m ecuaciones con m incógnitas (c) (ver ref. [4]):

$$A^T A\mathbf{c} = A^T\mathbf{b} \tag{4}$$

que se conocen como ecuaciones normales de Gauss.

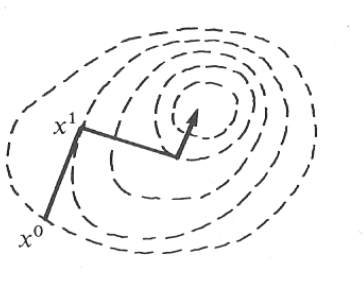
El problema general de aproximación no lineal

El problema general de aproximación es de mayor dificultad ya que presenta dos diferencias muy notables: por un lado hay que abordar la no linealidad de la aproximación (1) en los parámetros **c**; y por otro, el conjunto de datos o muestras podrá ser multidimensional.

En los problemas de optimización, hallar un mínimo de una función escalar objetivo multivariable $E_c(c_1,\ldots,c_n)$ requiere la anulación del gradiente:

$$\mathbf{g}(\mathbf{c}) = \nabla E_c(c_1,\ldots,c_m) = \begin{pmatrix} \dfrac{\partial E_c(c_1,\ldots,c_n)}{\partial c_1} \\ \vdots \\ \dfrac{\partial E_c(c_1,\ldots,c_n)}{\partial c_m} \end{pmatrix} \tag{5}$$

Para el caso particular de dos incógnitas: $(c_1, c_2) \rightarrow (x, y)$, la función objetivo $E_c(x, y)$ se puede representar mediante sus curvas de nivel $E_c(x, y) = C$ como se muestra en la figura, lo que permite presentar con sencillez algunos conceptos que son generalizables al caso n-dimensional.



1. Método del gradiente 2 variables

En particular, es bien sabido que el gradiente es perpendicular a las curvas de nivel, y coincide con la dirección de máxima pendiente. En la figura se muestra el proceso iterativo de búsqueda del mínimo conocido como método del gradiente y que se describe a continuación.

Método del gradiente

En este método también conocido como de "máxima pendiente" (steepest descent), el gradiente proporciona la dirección del descenso máximo de $E_c(\mathbf{c})$. El proceso iterativo que se muestra en la figura 1 queda descrito matemáticamente de la siguiente manera:

$$\text{A partir de } \mathbf{c}_0 \rightarrow \left\{ \begin{array}{l} \Delta\mathbf{c} = -\alpha\mathbf{g}(\mathbf{c}_i) \\ \mathbf{c}_{i+1} = \mathbf{c}_i + \Delta\mathbf{c} \end{array} \right\} \text{ , } i = 0,1,2,\ldots \text{ hasta que } \left\|\mathbf{g}(\mathbf{c}_{i+1})\right\| < tol \approx 0 \tag{6}$$

donde α es el valor escalar que maximiza el descenso de $E_c(\mathbf{c})$ en la dirección $-\mathbf{g}(\mathbf{c}_i)$. Es decir, el valor α para el que se produce la tangencia entre la dirección del gradiente y una curva de nivel. La ecuación de dicha semirrecta será en coordenadas paramétricas:

$$\mathbf{c} = \mathbf{c}_i - \alpha\mathbf{g}(\mathbf{c}_i) \quad , \quad \alpha \geq 0$$

Entonces, la función objetivo $E_c(\mathbf{c})$ pasa a ser unidimensional en α sobre la semirrecta:

$$E_c(\mathbf{c}_i - \alpha \mathbf{g}(\mathbf{c}_i)) \rightarrow E_c(\alpha)$$

y podremos aproximar el valor de α utilizando cualquiera de los algoritmos numéricos existentes para la optimización unidimensional. En particular, utilizaremos el método de la sección aurea (anexo I) que sobresale por su eficiencia computacional.

Este método tiene buena convergencia global, es decir, aunque $\mathbf{c}_0$ esté muy lejos del mínimo $\mathbf{c}^*$, el proceso suele converger con orden 1. Pero es computacionalmente costoso y tiene mala convergencia local, es decir, una vez cerca de $\mathbf{c}^*$, el avance hacia el mínimo puede ser muy lento e incluso puede parar lejos de $\mathbf{c}^*$ debido a los errores de redondeo. Se suele utilizar inicialmente para acercarse desde $\mathbf{c}_0$ hasta $\mathbf{c}^*$ lo suficiente como para arrancar otros métodos con mejor convergencia local como, por ejemplo, el método de Newton. Una mejora importante es el método del gradiente conjugado que es de orden 2.

Método del gradiente conjugado

Este método es muy similar al anterior salvo en la elección de direcciones conjugadas de avance:

$$\left\{\begin{array}{l}\text{A partir de } \mathbf{c}_0 \\ \text{con } \mathbf{d}_0 = \mathbf{g}(\mathbf{c}_0)\end{array}\right\} \rightarrow \left\{\begin{array}{l}\Delta\mathbf{c}_k = -\alpha\mathbf{d}_k \\ \left\{\begin{array}{l}\mathbf{d}_k = -\mathbf{g}(\mathbf{c}_k) + \beta\mathbf{d}_{k-1} \\ \beta = \dfrac{\mathbf{g}_k^T\mathbf{g}_k}{\mathbf{g}_{k-1}^T\mathbf{g}_{k-1}}\end{array}\right\} \\ \mathbf{c}_{k+1} = \mathbf{c}_k + \Delta\mathbf{c}\end{array}\right\}, \; k = 0,1,2,\ldots \text{ hasta que } \left\|\mathbf{g}(\mathbf{c}_{k+1})\right\| < tol \approx 0 \quad (7)$$

El método corrige la dirección de avance con el término $\beta \cdot \mathbf{d}_{k-1}$ y la elección de β es tal que la nueva dirección $\mathbf{d}_k$ es A-ortogonal a la anterior $\mathbf{d}_{k-1}$, es decir, se verifica: $\mathbf{d}_k \cdot A \cdot \mathbf{d}_{k-1} = 0$ donde A es la matriz hessiana de $E_c(\mathbf{c})$.

De esta forma se evita el problema del valle estrecho del método de la pendiente máxima, donde los gradientes sucesivos pueden ser muy parecidos salvo el signo, y la convergencia al mínimo muy lenta.

Es más, en el caso de función objetivo cuadrática en n dimensiones, el algoritmo converge a $\mathbf{c}^*$ en n iteraciones (2 en la figura anterior donde n=2).

Método de Newton

La anulación del gradiente (5) implica la resolución del sistema no lineal de ecuaciones:

$$\mathbf{g}(\mathbf{c}) = 0 \tag{8}$$

que podemos resolver mediante el siguiente proceso iterativo

$$\text{A partir de } \mathbf{c}_0 \rightarrow \left\{ \begin{array}{l} \Delta\mathbf{c}_i = -J(\mathbf{c}_i)^{-1}\mathbf{g}(\mathbf{c}_i) \\ \mathbf{c}_{i+1} = \mathbf{c}_i + \Delta\mathbf{c}_i \end{array} \right\},\ i = 0,1,2,\ldots \text{ hasta que } \left\|\mathbf{g}(\mathbf{c}_i)\right\| < tol \approx 0 \tag{9}$$

$$\text{donde} \qquad J(\mathbf{c}_i) = \frac{\partial \mathbf{g}(\mathbf{c}_i)}{\partial \mathbf{c}_i} = \begin{pmatrix} \frac{\partial g_1}{\partial c_1} & \cdots & \frac{\partial g_1}{\partial c_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial g_n}{\partial c_1} & \cdots & \frac{\partial g_n}{\partial c_n} \end{pmatrix} = \begin{pmatrix} \frac{\partial^2 F}{\partial c_1^2} & \cdots & \frac{\partial^2 F}{\partial c_1 \partial c_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 F}{\partial c_n \partial c_1} & \cdots & \frac{\partial^2 F}{\partial c_n^2} \end{pmatrix} \tag{10}$$

En este caso la matriz jacobiana es simétrica y se construye con las derivadas segundas de la función objetivo $E_c(\mathbf{c})$. Se denomina matriz Hessiana y será definida positiva (autovalores positivos) si el punto $\mathbf{c}^* = (c_1,\ldots,c_m)$ que verifica (8) es un mínimo.

Converge cuadráticamente, pero sólo si el vector inicial está suficientemente cerca de una solución. Si el gradiente $\mathbf{g}(\mathbf{c}) = \nabla E_c(\mathbf{c})$ no es conocido analíticamente, habrá que calcular numéricamente por diferencias finitas de la función objetivo, cada una de las m derivadas parciales, lo cual puede suponer un esfuerzo computacional importante, que se ve fuertemente acrecentado si también hay que evaluar cada una de las m^2 derivadas parciales segundas de la matriz Hessiana.

Para el caso que nos ocupa, donde la función objetivo es el error cuadrático, el método de Newton admite una simplificación importante:

$$E_c = \sum_{k=1}^{n} e_k^{\,2} \qquad \text{donde} \qquad e_k = y_k - f\left(x_k, c_1, \ldots, c_m\right) \tag{11}$$

Dicha simplificación nos conduce al método de Gauss-Newton donde no hay que evaluar las derivadas segundas, lo cual supondrá un ahorro computacional muy considerable.

Método de Gauss-Newton

Minimizar el error cuadrático (11) implica igualar su gradiente a cero, lo cual nos lleva al siguiente sistema no lineal de ecuaciones:

$$\mathbf{g}(\mathbf{c}) = \nabla E_c(\mathbf{c}) = \begin{pmatrix} \frac{\partial E_c}{\partial c_1} \\ \vdots \\ \frac{\partial E_c}{\partial c_n} \end{pmatrix} = 2\begin{pmatrix} \sum_{k=1}^{n} e_k(\mathbf{c})\frac{\partial e_k}{\partial c_1} \\ \vdots \\ \sum_{k=1}^{n} e_k(\mathbf{c})\frac{\partial e_k}{\partial c_n} \end{pmatrix} = 2\begin{pmatrix} e_1\frac{\partial e_1}{\partial c_1} + \ldots + e_n\frac{\partial e_n}{\partial c_1} \\ \vdots \\ e_1\frac{\partial e_1}{\partial c_n} + \ldots + e_n\frac{\partial e_n}{\partial c_n} \end{pmatrix} = 2\left(\frac{\partial \mathbf{e}(\mathbf{c})}{\partial \mathbf{c}}\right)^T \mathbf{e}(\mathbf{c}) = \begin{pmatrix} 0 \\ \vdots \\ 0 \end{pmatrix} \quad (n\times 1) \quad (12)$$

donde $\frac{\partial \mathbf{e}(\mathbf{c})}{\partial \mathbf{c}}$ es la matriz jacobiana de la función vectorial error $\mathbf{e}(\mathbf{c})$ respecto a la variable vectorial $\mathbf{c}$:

$$\frac{\partial \mathbf{e}(\mathbf{c})}{\partial \mathbf{c}} = \begin{pmatrix} \frac{\partial e_1(\mathbf{c})}{\partial c_1} & \cdots & \frac{\partial e_1(\mathbf{c})}{\partial c_m} \\ \vdots & \ddots & \vdots \\ \frac{\partial e_n(\mathbf{c})}{\partial c_1} & \cdots & \frac{\partial e_n(\mathbf{c})}{\partial c_m} \end{pmatrix} \rightarrow A(\mathbf{c}) \quad (n\times m) \qquad (13)$$

Eliminando el 2 del sistema (12), el vector gradiente y la matriz jacobiana serán:

$$\mathbf{g}(\mathbf{c}) = \nabla E_c(\mathbf{c}) = A^T(\mathbf{c})\cdot\mathbf{e}(\mathbf{c}) \quad (m\times 1)$$

$$J(\mathbf{c}) = \frac{\partial\left(A^T\mathbf{e}\right)}{\partial \mathbf{c}} = A^T\frac{\partial \mathbf{e}}{\partial \mathbf{c}} + \frac{\partial A^T}{\partial \mathbf{c}}\mathbf{e} = A^T(\mathbf{c})\cdot \le A(\mathbf{c}) + \sum_{k=1}^{n} e_k(\mathbf{c})\left(\frac{\partial^2 e_k(\mathbf{c})}{\partial \mathbf{c}^2}\right) \quad (m\times m) \qquad (14)$$

Pero si el modelo de ajuste es razonablemente bueno, las desviaciones e_k serán pequeñas y podrá despreciarse el segundo término de $J(\mathbf{c})$ en (14), es decir las derivadas segundas. Entonces, $J(\mathbf{c}) = A^T(\mathbf{c})\cdot A(\mathbf{c})$ y en cada iteración i del proceso de Newton (9), habrá que resolver el sistema lineal:

$$\left[A^T(\mathbf{c}_i)\cdot A(\mathbf{c}_i)\right]\Delta\mathbf{c}_i = -A^T(\mathbf{c}_i)\cdot\mathbf{e}(\mathbf{c}_i) \qquad (15)$$

apreciándose la similitud con las ecuaciones normales de Gauss (4), del caso lineal.

Debe tenerse en cuenta que la aproximación solo afecta al cálculo de la matriz jacobiana $J(\mathbf{c})$ y por tanto puede reducir ligeramente el orden de convergencia 2, aumentando ligeramente quizás, el número necesario de iteraciones, pero no la precisión del vector de incógnitas $\mathbf{c}$ obtenido.

Método de Levenberg-Marquardt

El orden de convergencia es lineal (1) en el método de la máxima pendiente y la parte final del proceso puede ser muy lenta e incluso imposible cuando la forma de las curvas de nivel sea excesivamente alargada (valle muy estrecho). Entonces el método avanza en pasos cortos transversales, adelante atrás, trasladándose muy lentamente en la dirección longitudinal hacia el mínimo. Por el contrario, el método de Newton solo converge localmente, pero entonces lo hace en forma cuadrática, es decir con gran velocidad.

En este sentido ambos métodos son complementarios, y en la solución del sistema de ecuaciones (5) en un problema de optimización de convergencia dificultosa, es recomendable iniciar el proceso iterativo con el método de máxima pendiente y cambiar al método de Newton cuando la solución se haya acercado lo suficiente.

Una alternativa más completa que integra ambas opciones es el método de Levenberg-Marquardt, que consiste en aunar las ventajas de ambos métodos, alternando entre uno y otro, en función de un parámetro μ, que al tomar valores grandes aproxima el método al del gradiente y al disminuir su valor al de Gauss-Newton.

$$\text{A partir de } \mathbf{c}_0 \rightarrow \left\{ \begin{array}{l} \Delta\mathbf{c} = -\left(J(\mathbf{c}_i) + \mu \cdot I_n\right)^{-1} \mathbf{g}(\mathbf{c}_i) \\ \mathbf{c}_{i+1} = \mathbf{c}_i + \Delta\mathbf{c} \end{array} \right\}, \; i = 0,1,2,\ldots \text{ hasta que } \left\|\mathbf{g}(\mathbf{c}_i)\right\| < tol \approx 0 \quad (16)$$

En las iteraciones en las que el valor de $E_c(\mathbf{c})$ aumenta, aumenta también el valor de μ (que se divide por determinado factor menor que la unidad a especificar). Al contrario, cuando $E_c(\mathbf{c})$ disminuye, también disminuye μ (multiplicando por el mismo factor).

El cálculo de estas soluciones implica una gran cantidad de operaciones (en especial en la predicción de comportamientos dinámicos, una de las aplicaciones de las RN), ejemplo de ello es el siguiente comentario[1] que en la predicción del retorno del cometa Halley (llevada a cabo por Clairaut, Lalande y Lepaute en 1748) Lalande escribió: *Durante seis meses, calculábamos de la mañana a la noche, a veces incluso en las comidas; como consecuencia de lo cual, contraje una enfermedad que cambió mi constitución para siempre. La ayuda prestada por Madame Lalande fue tal que, sin ella, no hubiésemos podido llevar a cabo una tarea que implicaba el cálculo de la distancia de Júpiter y Saturno al cometa, separadamente por cada grado, durante 150 años.* La evolución de la tecnología y de las matemáticas ha permitido que estas tareas se puedan llevar a cabo de una forma practicable, pero la creciente velocidad en la generación de datos y la búsqueda de precisiones mayores, hace que las matemáticas (la algoritmia en este caso) tenga que evolucionar también, dando lugar a las redes neuronales.

[1] C.W. Gear, R.D. Skeel, The development of ODE methods: A symbiosis between hardware and numerical analysis, A history of scientific computing, ed. By Stephen Nash, AC. Press 1990.

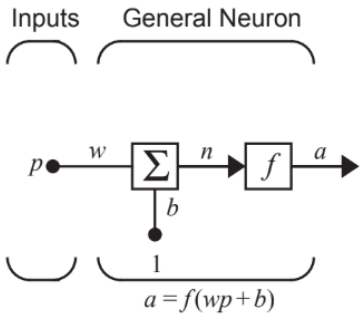
Redes neuronales

General

Las redes neuronales tratan de imitar al cerebro humano, simulando su sistema de aprendizaje. El cerebro está formado por neuronas que se interconectan entre sí y se envían impulsos, pudiendo activar las neuronas adyacentes. En las redes neuronales también existen unidades (neuronas) que se interconectan, como se explica a continuación:

Una neurona es una unidad de procesamiento con la siguiente estructura:

- Entrada (**p**). Es una señal que recibe del exterior o de otras neuronas, puede ser un escalar o un vector.
- Pesos[1] (***w***). Las entradas son ponderadas según la importancia que tiene una señal para la neurona. En el símil biológico, si una neurona se encarga de reconocer los olores, la información proveniente de los ojos será inútil (w=0), en cambio la proveniente de la nariz será muy importante. Será un vector fila de dimensión R, donde R es la dimensión de la entrada.
- Sesgos (b). Similar a los pesos, pero su entrada está siempre activa (vale 1).
- Sumador.
- Función de transferencia (f). Transforma la entrada a la neurona y determina la zona de activación de la neurona.
- Salida (a). Salida de la neurona, que se enviará a otra neurona o será una salida del sistema. Como se puede observar en la siguiente figura, vendrá dada por $a = f(\mathbf{w} \cdot \mathbf{p} + b)$.



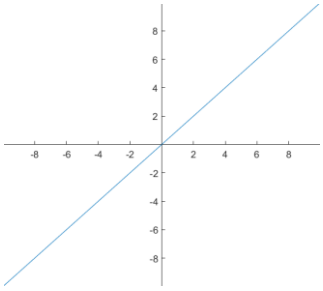
2. Neurona Artificial

[1] Cuando no haya posibilidad de confusión se utilizará 'pesos' para hacer referencia a pesos y sesgos.

La función de transferencia (f) no es más que una función matemática, que vendrá determinado, en parte, por el objetivo de la red. En este trabajo haremos uso de la función sigmoide, la tangente hiperbólica y la función lineal. Otras funciones comunes vienen recogidas en la tabla I del anexo I.

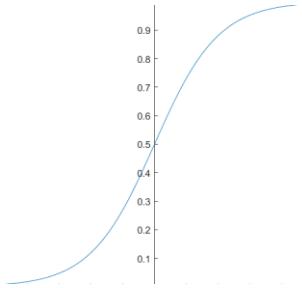
- Función lineal. Asigna a la salida el valor de la entrada, se usa principalmente en la capa de salida.
- Función sigmoidea. Esta función permite transformar el intervalo (-∞,∞) en el intervalo (0,1). Además, tiene la ventaja de ser diferenciable.
- Tangente hiperbólica. Otra función sigmoidea, similar a la anterior, pero que transforma el intervalo (-∞,∞) en (-1,1).

$$f(x) = x$$



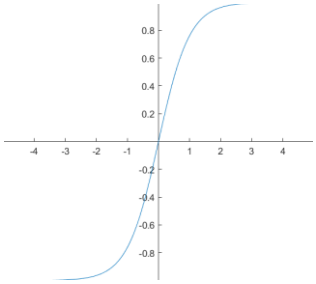
3. Función lineal

$$f(x) = \frac{1}{1+e^{-x}}$$



4. Función sigmoidea

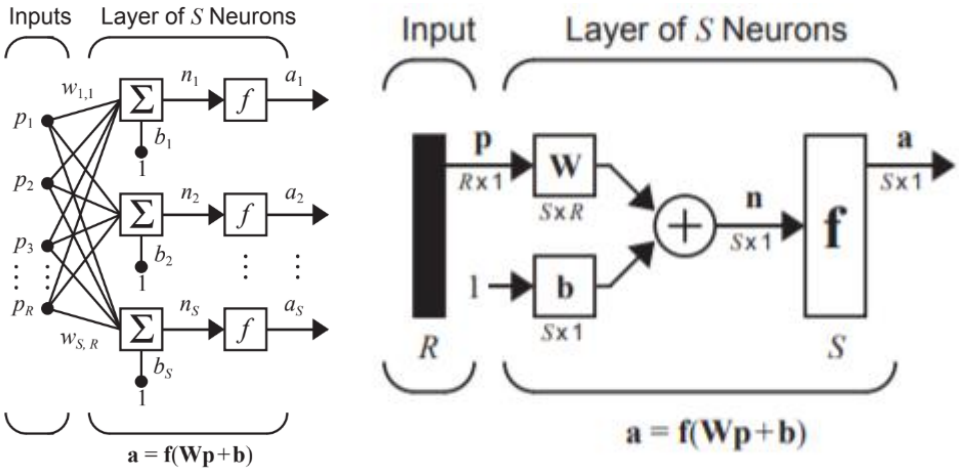
$$f(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$



5. Tangente hiperbólica

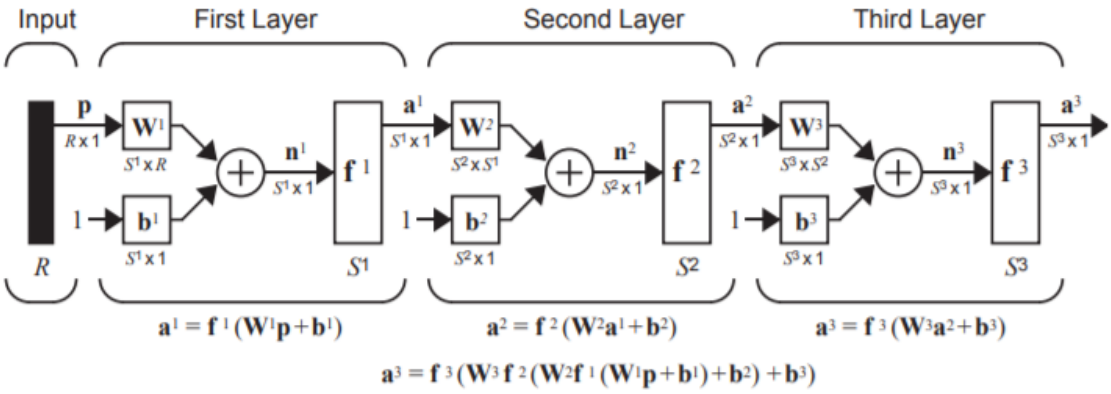
Al igual que en el cerebro, el poder de las neuronas se basa en las conexiones, así, el concepto de neurona induce el de capa de neuronas. En el esquema siguiente se muestra una capa con *d* neuronas y una entrada de dimensión R. Su funcionamiento es el de las neuronas trabajando independientemente. El vector fila ***w***, ahora será una matriz W_{dxR}, donde el elemento w_{ij} será el peso correspondiente a la entrada en la posición j y a la neurona i-ésima. La salida a, a su vez, se transforma en un vector de dimensión d (aunque en la figura consta como S).

La segunda parte de la figura es una representación alternativa de la capa que resulta más útil y utilizaremos en el futuro.



6. Capa Neuronal

El siguiente paso es la evolución a red neuronal multicapa, las distintas capas de neuronas se interconectan, haciendo que la salida de una capa sea la entrada de la siguiente. Los superíndices indican a que capa pertenecen. Así para la capa *m*, d^m es el numero de neuronas, W^m es la matriz de pesos, f^m la función de transferencia y a^m el vector de salida de dimensión d^m. Mención especial requiere la nomenclatura de la última capa que define la salida de la red, y donde d^M es el numero de neuronas, W^M es la matriz de pesos, f^M la función de transferencia y $\mathbf{a}^M = \mathbf{a}$ el vector de salida de la red de dimensión d^M.



7. Red neuronal feedforward

Aunque la capacidad de estos sistemas diste mucho de sus homónimas biológicas (en un cerebro cada neurona tiene alrededor de 7000 conexiones), nos permiten adaptar prácticamente cualquier función. Sin embargo, aumentar la capacidad de la red a base de aumentar las conexiones o el número de neuronas aumenta considerablemente el coste computacional, así uno de los primeros problemas a afrontar es determinar la estructura de la red.

El número de neuronas de la capa de entrada y la de salida viene determinado por las necesidades del problema, pero el número de neuronas por capa y el número de capas están sujetos a la experiencia del usuario, aunque más adelante veremos un método que nos permite conocer el número de pesos utilizados, permitiendo corregir la estructura inicial.

La definición de una red neuronal con su arquitectura: capas y neuronas, y los valores correspondientes de pesos y sesgos (vector $\mathbf{c}$), permite calcular de manera directa la salida $\mathbf{a}(\mathbf{c})$ para la entrada $\mathbf{p}$ (en la figura 7 podemos encontrar el caso particular de una red con 3 capas):

$$\mathbf{a}(\mathbf{c}) = f^M(W^M \cdot f(\ldots f(W^1 \cdot \mathbf{p} + \mathbf{b}^1)) + \mathbf{b}^M) \tag{17}$$

El siguiente problema a resolver es el entrenamiento de la red (determinar los valores de los pesos para que las salidas $\mathbf{a}(\mathbf{c})$ aproximen lo mejor posible la información experimental disponible). La dificultad de este proceso hizo que las redes neuronales no se utilizaran hasta el desarrollo del algoritmo de backpropagation en 1986 y que luego explicamos.

Entrenamiento

El problema central de las RN radica en hallar los valores de los pesos que hagan que la red reproduzca la función deseada. Existen varios grupos en los que clasificar los distintos algoritmos de entrenamiento (o aprendizaje), este trabajo se centrará en el aprendizaje supervisado y también se comentará el aprendizaje no supervisado:

- Aprendizaje supervisado. En estos algoritmos se parte de un conjunto de datos de entrenamiento de los que se conoce la entrada $\left(\mathbf{p} \rightarrow \left[\begin{Bmatrix} p_1 \\ \vdots \\ p_R \end{Bmatrix}_1, \ldots, \begin{Bmatrix} p_1 \\ \vdots \\ p_R \end{Bmatrix}_Q\right]\right)$ y el objetivo $\left(\mathbf{t} \rightarrow \left[\begin{Bmatrix} t_1 \\ \vdots \\ t_{d^M} \end{Bmatrix}_1, \ldots, \begin{Bmatrix} t_1 \\ \vdots \\ t_{d^M} \end{Bmatrix}_Q\right]\right)$ (p. ej.: un grupo de personas del que se conoce su edad, altura, sexo… y también su peso), con los que la red buscará simular el comportamiento del objeto de estudio.

  El algoritmo básico es el Backpropagation, que permite calcular el vector gradiente respecto de los pesos y que se explica a continuación. La característica principal de estos métodos es que para aprender se debe conocer a priori el resultado en un conjunto suficiente de casos o muestras.

- Aprendizaje no supervisado. En estos algoritmos el conjunto de entrenamiento solo dispone de entradas y es la red la que, con la experiencia, altera sus parámetros hasta dar con valores de salida óptimos (en base a parámetros estadísticos). La capacidad de las redes es mayor con estos métodos, pero su complejidad obliga a tener estructuras más simples.

  El algoritmo básico es el algoritmo de Hebb, que consiste en reforzar el peso entre dos neuronas conectadas si al activarse[1] una (aumentar su valor) se activa la otra, de lo contrario el valor del peso se atenúa. Un ejemplo biológico sencillo podría ser una neurona encargada de decidir si te gusta un jersey, que recibe información del ojo y de la oreja. Al ver el jersey el ojo envía una señal (se activa) y se decide sobre el jersey (se activa), esa conexión se fortalecerá. Sin embargo, al oírse unas escaleras caerse, el oído enviará una señal, pero no afectará a la decisión sobe el jersey, por lo que la conexión se debilita.

Entrenamiento Supervisado

La descripción de los métodos anteriores nos lleva con naturalidad a quedarnos con el gradiente conjugado, y Levenberg-Marquardt que integra el método del gradiente y el de Gauss-Newton. Son los más eficientes y no precisan el cálculo de la matriz Hessiana, es decir, de las derivadas segundas.

Para el cálculo de las primeras derivadas que conforman el gradiente y la matriz jacobiana de los errores, siempre se puede recurrir al método de las diferencias finitas. Sin embargo, los requerimientos de eficiencia computacional aconsejan explorar la derivación analítica, lo cual en el caso de redes neuronales se plasma en algoritmos regresivos a partir de la última capa que reciben el nombre de "backpropagation". Ambos tipos de métodos se describen en los siguientes apartados.

Se define el error cometido para una muestra q como: $\mathbf{e}(\mathbf{c})\big|_q = \{\mathbf{a}(\mathbf{c}) - \mathbf{t}\}_q$, es decir, la diferencia entre el valor estimado por la red y el valor real (objetivo). La capacidad de Matlab de trabajar con todas las muestras a la vez permite disponer simultáneamente todos los errores para todas las muestras (Q) en el siguiente vector columna:

$$\mathbf{e}(\mathbf{c}) = \left\{ \left\{ e_1, \ldots, e_{d^M} \right\}_1, \ldots, \left\{ e_1, \ldots, e_{d^M} \right\}_q, \ldots, \left\{ e_1, \ldots, e_{d^M} \right\}_Q \right\}^T \tag{18}$$

[1] En el caso de las neuronas biológicas su comportamiento es discreto (a base de impulsos), por lo que se activan en momentos dados y permaneces inactivas el resto del tiempo. En el caso de las RN, funcionan de forma continuada, pudiendo dar salidas nulas.

y calcular fácilmente el error cuadrático $E_c(\mathbf{c}) = \sum_{k=1}^{n} e_k{}^2$ mediante la suma de sus cuadrados, y también sus derivadas, tal y como requieren los métodos de entrenamiento.

Diferencias finitas

A partir de (18) es sencillo aunque costoso computacionalmente, obtener mediante diferencias finitas el vector gradiente del error cuadrático $E_c(\mathbf{c})$ y la matriz jacobiana del vector error $\mathbf{e}(\mathbf{c})$, en ambos casos respecto a los pesos $\mathbf{c}$.

Cálculo del gradiente

Los métodos del gradiente y del gradiente conjugado requieren el cálculo del gradiente del error cuadrático:

$$E_c(\mathbf{c}) = \sum_{q=1}^{Q} \left\{ \sum_{k=1}^{d^M} e_k^2 \right\}_q \quad \text{donde} \quad e_k = t_k - a_k(\mathbf{c}) \tag{19}$$

respecto a las incógnitas $\mathbf{c}$: $\mathbf{g}(\mathbf{c}) = \dfrac{\partial E_c(\mathbf{c})}{\partial \mathbf{c}}$

Por tanto, el vector $\mathbf{g}(\mathbf{c}) = \dfrac{\partial E_c(\mathbf{c})}{\partial \mathbf{c}}$ tendrá una única columna y tantas filas como número (*nx*) de pesos incógnita existan en la RN. Se podrá aproximar mediante diferencias finitas de la siguiente forma:

$$g_k = \frac{\partial E_c}{\partial c_k} = \frac{E_c(c_1,\ldots,c_k+h,\ldots,c_m) - E_c(c_1,\ldots,c_k,\ldots,c_m)}{h} \quad \text{donde} \quad \begin{cases} k = 1,\ldots,m = nx \\ h << 1 \end{cases} \tag{20}$$

En Matlab es muy sencillo y eficiente calcular simultáneamente los errores para todas las muestras y sumar sus cuadrados para obtener el error cuadrático (19), pero el cálculo del error perturbado $(h << 1)$ habrá que repetirlo para cada una de las variables incógnita, en un bucle for.

Cálculo de la matriz jacobiana del error

El método de Levenberg-Marquardt requiere el cálculo de la matriz jacobiana del error respecto a las incógnitas $\mathbf{c}$: $j = \dfrac{\partial \mathbf{e}(\mathbf{c})}{\partial \mathbf{c}}$ donde el vector error $\mathbf{e}(\mathbf{c})$ se organiza según lo indicado en (18), en una única columna adosando sucesivamente los errores correspondientes a cada una de las muestras:

$$\mathbf{e}(\mathbf{c}) = \left\{ \left\{ e_1,\ldots,e_{d^M} \right\}_1,\ldots,\left\{ e_1,\ldots,e_{d^M} \right\}_q,\ldots,\left\{ e_1,\ldots,e_{d^M} \right\}_Q \right\}^T$$

Por tanto, la matriz $j = \frac{\partial \mathbf{e}(\mathbf{c})}{\partial \mathbf{c}}$ tendrá $d^M \cdot Q$ filas y tantas columnas como número (nx) de pesos incógnita existan. Se podrá aproximar mediante diferencias finitas de la siguiente forma:

$$j_{ik} = \frac{\partial e_i}{\partial c_k} = \frac{e_i(c_1,...,c_k+h,...,c_n) - R_i(c_1,...,c_k,...,c_n)}{h} \quad \text{donde} \begin{cases} i = 1,...,d^M \cdot Q \\ k = 1,...,m = nx \\ h << 1 \end{cases} \qquad (21)$$

En Matlab se puede automatizar el cálculo de la matriz j (por columnas) de manera eficiente mediante el cálculo simultáneo para todas las muestras, del vector error $\mathbf{e}(\mathbf{c})$ y de los sucesivos vectores error perturbados:

$$j_k = \frac{\partial \mathbf{e}(\mathbf{c})}{\partial c_k} = \frac{\mathbf{e}(c_1,...,c_k+h,...,c_m) - \mathbf{e}(c_1,...,c_k,...,c_m)}{h} \quad k = 1,...,m = nx$$

pero el cálculo del error perturbado habrá que repetirlo para cada una de las columnas en un bucle for.

Backpropagation

Los requisitos de eficiencia computacional aconsejan deducir las derivadas analíticamente, ya que la derivación numérica por diferencias finitas puede ser muy costosa cuando se eleva el número de incógnitas **c**, lo cual es común al trabajar con redes neuronales. No obstante, el cálculo analítico del gradiente y de la matriz jacobiana del vector error (18) no es sencillo ni inmediato, ya que la mayor parte de las derivadas lo son con respecto a pesos de capas internas, no directamente relacionadas con el vector error definido en la última capa. Deberemos considerar la regla de la cadena para la derivación e implementar en Matlab los algoritmos resultantes sin bucles for, todo lo cual resulta complejo y laborioso, pero muy ventajoso computacionalmente frente a la alternativa de diferencias finitas descrita en el apartado anterior.

Cálculo del gradiente

El método del gradiente y el método del gradiente conjugado requieren el cálculo del gradiente del error cuadrático del conjunto de muestras con respecto a las incógnitas (pesos y sesgos) de la red:

$$\nabla E_c = \nabla \left[\sum_{q=1}^{Q} \left\{ \sum_{k=1}^{d^M} e_k^2 \right\}_q \right] = \sum_{q=1}^{Q} \nabla \left\{ \sum_{k=1}^{d^M} e_k^2 \right\}_q$$

donde d^M es el número de neuronas de la última capa M, número que determina la dimensión de la salida a de la red y por tanto el número de errores $\{e_k\}_q = \{t_k - a_k\}_q$, $k = 1,\ldots,d^M$ a contabilizar para una única muestra q.

Desarrollaremos en primer lugar la expresión del gradiente de una única muestra q:

$$\nabla\{E_c\}_q = \nabla\left\{\sum_{k=1}^{d^M} e_k^2\right\}_q$$

y mostraremos a continuación como obtener en Matlab el gradiente de todas las muestras simultáneamente sin bucles for. Finalmente, sumaremos los gradientes $\nabla\{E_c\}_q$ de las muestras para obtener el gradiente ∇E_c del conjunto de muestras.

La regla de la cadena para una capa m

Bastará, con mostrar el cálculo del gradiente de una muestra q para una capa m. Comenzaremos por la parte más compleja que son las derivadas respecto a las incógnitas peso W^m:

$\nabla_w^m\{E_c\}_q = \nabla_w^m\left\{\sum_{k=1}^{d^M} e_k^2\right\}_q$ para una muestra $\{\mathbf{p},\mathbf{t}\}_q$ respecto a los pesos $W^m = \begin{pmatrix} w_{1,1} & \cdots & w_{1,d^{m-1}} \\ \vdots & \ddots & \vdots \\ w_{d^m,1} & \cdots & w_{d^m,d^{m-1}} \end{pmatrix}^m$ resultando en un vector columna de dimensiones $\left(d^m \cdot d^{m-1}\,\mathrm{x}\,1\right)_q$.

Derivamos primero manteniendo la estructura matricial de W^m para obtener una matriz de dimensiones $\left(d^m\,\mathrm{x}\,d^{m-1}\right)_q$, que posteriormente reformaremos adosando sus columnas en un único vector.

$$\left\{\frac{\partial E_c}{\partial w_{ij}^m}\right\}_q = \left\{\begin{pmatrix} \frac{\partial}{\partial w_{1,1}} & \cdots & \frac{\partial}{\partial w_{1,d^{m-1}}} \\ \vdots & \ddots & \vdots \\ \frac{\partial}{\partial w_{d^m,1}} & \cdots & \frac{\partial}{\partial w_{d^m,d^{m-1}}} \end{pmatrix}^m E_c\right\}_q$$

$$\left\{\frac{\partial E_c}{\partial w_{ij}^m}\right\}_q = \left\{\frac{\partial E_c}{\partial n_i^m}\frac{\partial n_i^m}{\partial w_{ij}^m},\ i=1,\ldots,d^m,\ j=1,\ldots,d^{m-1}\right\}_q = \left\{\begin{pmatrix} \frac{\partial E_c}{\partial n_1^m}\frac{\partial n_1^m}{\partial w_{11}^m} & \cdots & \frac{\partial E_c}{\partial n_1^m}\frac{\partial n_1^m}{\partial w_{1,d^{m-1}}} \\ \vdots & \ddots & \vdots \\ \frac{\partial E_c}{\partial n_{d^m}^m}\frac{\partial n_{d^m}^m}{\partial w_{d^m,1}^m} & \cdots & \frac{\partial E_c}{\partial n_{d^m}^m}\frac{\partial n_{d^m}^m}{\partial w_{d^m,d^{m-1}}} \end{pmatrix}\right\}_q \quad (22)$$

Recordando que:

$$\left\{\mathbf{n}^m\right\}_q = \left\{W^m\mathbf{a}^{m-1}+\mathbf{b}^m\right\}_q = \left\{\begin{pmatrix} w_{1,1} & \cdots & w_{1,d^{m-1}} \\ \vdots & \ddots & \vdots \\ w_{d^m,1} & \cdots & w_{d^m,d^{m-1}} \end{pmatrix}^m \begin{pmatrix} a_1 \\ \vdots \\ a_{d^{m-1}} \end{pmatrix}^{m-1} + \begin{pmatrix} b_1 \\ \vdots \\ b_{d^m} \end{pmatrix}^m\right\}_q \quad (23)$$

deducimos:

$$\begin{pmatrix} \frac{\partial n_1^m}{\partial w_{11}^m} = a_1^m & \cdots & \frac{\partial n_{d^m}^m}{\partial w_{1,d^{m-1}}^m} = a_{d^{m-1}}^{m-1} \\ \vdots & \ddots & \vdots \\ \frac{\partial n_{d^m}^m}{\partial w_{d^m,1}^m} = a_1^m & \cdots & \frac{\partial n_{d^m}^m}{\partial w_{d^m,d^{m-1}}^m} = a_{d^{m-1}}^{m-1} \end{pmatrix}_q \quad (24)$$

Entonces, llamando vector sensibilidad $\left\{\mathbf{s}^m\right\}_q$ al vector $\left\{\frac{\partial E_c}{\partial \mathbf{n}^m}\right\}_q$, tendremos:

$$\left\{\frac{\partial E_c}{\partial w_{ij}^m}\right\}_q = \left\{\left(\frac{\partial E_c}{\partial \mathbf{n}^m}a_1^{m-1}, \frac{\partial E_c}{\partial \mathbf{n}^m}a_2^{m-1}, \ldots, \frac{\partial E_c}{\partial \mathbf{n}^m}a_{d^{m-1}}^{m-1}\right)\right\}_q = \left\{\left(\mathbf{s}^m a_1^{m-1}, \mathbf{s}^m a_2^{m-1}, \ldots, \mathbf{s}^m a_{d^{m-1}}^{m-1}\right)\right\}_q \quad (25)$$

Finalmente pasamos de la matriz (25) al vector columna gradiente, simplemente adosando las columnas de (25) en un único vector:

$$\left\{\nabla_w^m E_c\right\}_q = \left\{\begin{pmatrix} \mathbf{s}^m a_1^{m-1} \\ \vdots \\ \mathbf{s}^m a_{d^{m-1}}^{m-1} \end{pmatrix}\right\}_q \quad (26)$$

Lo realizado para una muestra $\left\{\mathbf{p},\mathbf{t}\right\}_q$, es válido para todas las muestras a la vez, ya que el lenguaje matricial de Matlab permite operar simultáneamente con todos los valores de

$$\mathbf{p} \rightarrow \left[\left\{\mathbf{p}\right\}_1, \ldots, \left\{\mathbf{p}\right\}_Q\right] \rightarrow \left(R \text{ x } 1 \cdot Q\right)$$

Entonces podremos obtener simultáneamente todos los valores de cálculo de la primera capa:

$$\mathbf{n}^1 \rightarrow \left[\left\{ \mathbf{n}^1 \right\}_1, ..., \left\{ \mathbf{n}^1 \right\}_Q \right] \rightarrow \left(d^1 \, \mathrm{x} \, 1 \cdot Q \right)$$

mediante: $$\mathbf{n}^1 = W^1 \mathbf{p} + \mathbf{b}^1 = \left\{ \begin{pmatrix} w^1_{1,1} & \cdots & w^1_{1,R} \\ \vdots & \ddots & \vdots \\ w^1_{d^1,1} & \cdots & w^1_{d^1,R} \end{pmatrix} \begin{pmatrix} p_1 \\ \vdots \\ p_R \end{pmatrix}_q + \begin{pmatrix} b_1 \\ \vdots \\ b_{d^1} \end{pmatrix} \right\}_{q=1,...,Q}$$

para obtener todas las salidas de la primera capa

$$\mathbf{a}^1 = \left[\left\{ \mathbf{a}^1 \right\}_1, ..., \left\{ \mathbf{a}^1 \right\}_Q \right] \rightarrow \left(d^1 \, \mathrm{x} \, 1 \cdot Q \right) \quad \text{mediante} \quad \left\{ \mathbf{a}^1 \right\}_q = \left\{ f^1(\mathbf{n}^1) \right\}_q$$

y progresar sucesivamente hasta cualquier capa m:

$$\mathbf{a}^m = f^m(\mathbf{n}^m) = f^m(W^m \mathbf{a}^{m-1} + \mathbf{b}^m) \rightarrow (d^m \, \mathrm{x} \, 1 \cdot Q)$$

La expresión (26) del gradiente se puede generalizar a todas las muestras:

$$\left\{ \nabla^m_w E_c \right\} \rightarrow \left\{ \begin{pmatrix} \mathbf{s}^m a_1^{m-1} \\ \vdots \\ \mathbf{s}^m a_{d^{m-1}}^{-1} \end{pmatrix}_1, ..., \begin{pmatrix} \mathbf{s}^m a_1^{m-1} \\ \vdots \\ \mathbf{s}^m a_{d^{m-1}}^{-1} \end{pmatrix}_Q \right\} (d^m \cdot d^{m-1} \, \mathrm{x} \, Q) \qquad (27)$$

y calcular eficientemente como se describe en el apartado "Implementación en Matlab". Bastará sumar las filas de (27) para obtener el vector gradiente de todas las muestras:

$$\nabla^m_w E_c = \sum_{q=1}^{Q} \nabla^m_w \left\{ \sum_{k=1}^{d^M} e_k^2 \right\}_q$$

Lo mismo, pero de manera más sencilla ocurre con las derivadas respecto a los sesgos (**b**), ya que de (23) se deduce que la matriz $\begin{pmatrix} \frac{\partial n^m_1}{\partial b^m_1} = 1 & \cdots & \frac{\partial n^m_{d^m}}{\partial b^m_1} = 0 \\ \vdots & \ddots & \vdots \\ \frac{\partial n^m_{d^m}}{\partial b^m_{d^m}} = 0 & \cdots & \frac{\partial n^m_{d^m}}{\partial b^m_{d^m}} = 1 \end{pmatrix}_q$ es la matriz identidad y por tanto:

$$\left\{ \nabla^m_b E_c \right\} \rightarrow \left\{ \left(\mathbf{s}^m \right)_1, ..., \left(\mathbf{s}^m \right)_Q \right\} (d^m \, \mathrm{x} \, Q) \qquad (28)$$

y sumando las filas de (28): $$\nabla^m_b E_c = \sum_{q=1}^{Q} \nabla^m_b \left\{ \sum_{k=1}^{d^M} e_k^2 \right\}_q$$

Resuelta la capa genérica m, organizaremos el vector gradiente completo con las derivadas respecto a los pesos de todas las capas, en el siguiente vector columna:

$$\nabla E_c = \left(\left[\nabla_b^1 E_c, \nabla_w^1 E_c \right], ..., \left[\nabla_b^m E_c, \nabla_w^m E_c \right], ..., \left[\nabla_b^M E_c, \nabla_w^M E_c \right] \right)^T \rightarrow (nx \,\mathrm{x}\, 1)$$

donde *nx* es el número total de incógnitas.

Este vector es el que utilizaremos como vector gradiente $\mathbf{g}(\mathbf{x})$ en las iteraciones (6) y (7) de los métodos del gradiente y del gradiente conjugado.

Fórmula recursiva para el vector de sensibilidad

Las expresiones (27) y (28) requieren el cálculo previo del vector de sensibilidad:

$$\left\{ \mathbf{s}^m \right\}_q = \left\{ \frac{\partial E_c}{\partial \mathbf{n}^m} \right\}_q \rightarrow \left\{ \begin{array}{l} s_1^m = \dfrac{\partial E_c}{\partial n_1^m} = \dfrac{\partial E_c}{\partial n_1^{m+1}} \dfrac{\partial n_1^{m+1}}{\partial n_1^m} + \dfrac{\partial E_c}{\partial n_2^{m+1}} \dfrac{\partial n_2^{m+1}}{\partial n_1^m} + ... + \dfrac{\partial E_c}{\partial n_{d^{m+1}}^{m+1}} \dfrac{\partial n_{d^{m+1}}^{m+1}}{\partial n_1^m} \\ \vdots \\ s_{d^m}^m = \dfrac{\partial E_c}{\partial n_{d^m}^m} = \dfrac{\partial E_c}{\partial n_1^{m+1}} \dfrac{\partial n_1^{m+1}}{\partial n_{d^m}^m} + \dfrac{\partial E_c}{\partial n_2^{m+1}} \dfrac{\partial n_2^{m+1}}{\partial n_{d^m}^m} + ... + \dfrac{\partial E_c}{\partial n_{d^{m+1}}^{m+1}} \dfrac{\partial n_{d^{m+1}}^{m+1}}{\partial n_{d^m}^m} \end{array} \right\}_q \tag{29}$$

$$\left\{ \begin{pmatrix} s_1^m \\ \vdots \\ s_{d^m}^m \end{pmatrix} = \begin{pmatrix} \dfrac{\partial n_1^{m+1}}{\partial n_1^m} & \cdots & \dfrac{\partial n_{d^{m+1}}^{m+1}}{\partial n_1^m} \\ \vdots & \ddots & \vdots \\ \dfrac{\partial n_1^{m+1}}{\partial n_{d^m}^m} & \cdots & \dfrac{\partial n_{d^{m+1}}^{m+1}}{\partial n_{d^m}^m} \end{pmatrix} \begin{pmatrix} \dfrac{\partial E_c}{\partial n_1^{m+1}} \\ \vdots \\ \dfrac{\partial E_c}{\partial n_{d^{m+1}}^{m+1}} \end{pmatrix} \right\}_q \rightarrow \left\{ \mathbf{s}^m = \left(\frac{\partial \mathbf{n}^{m+1}}{\partial \mathbf{n}^m} \right)^T \frac{\partial E_c}{\partial \mathbf{n}^{m+1}} = \left(\frac{\partial \mathbf{n}^{m+1}}{\partial \mathbf{n}^m} \right)^T \mathbf{s}^{m+1} \right\}_q \tag{30}$$

La expresión (30) es una fórmula recurrente para obtener la sensibilidad a falta de concretar la matriz jacobiana $\left(\dfrac{\partial \mathbf{n}^{m+1}}{\partial \mathbf{n}^m} \right)$ y la sensibilidad de la última capa $\mathbf{s}^M$. Para ello, teniendo en cuenta que:

$$\frac{\partial n_i^{m+1}}{\partial n_j^m} = \frac{\partial \sum_{l=1}^{d^m} w_{il}^{m+1} a_l^m + b_i}{\partial n_j^m} = w_{ij}^{m+1} \frac{\partial a_j^m}{\partial n_j^m} = w_{ij}^{m+1} \frac{\partial f^m(n_j^m)}{\partial n_j^m} = w_{ij}^{m+1} \dot{f}^m(n_j^m)$$

$$\left\{ \frac{\partial \mathbf{n}^{m+1}}{\partial \mathbf{n}^m} \right\}_q = \left\{ \begin{array}{l} \left[\dfrac{\partial n_1^{m+1}}{\partial \mathbf{n}^m} \right]^T = \left[w_{11}^{m+1} \dot{f}^m(n_1^m),\ w_{12}^{m+1} \dot{f}^m(n_2^m), ..., w_{1,d^m}^{m+1} \dot{f}^m(n_{d^m}^m) \right] \\ \vdots \\ \left[\dfrac{\partial n_{d^{m+1}}^{m+1}}{\partial \mathbf{n}^m} \right]^T = \left[w_{d^{m+1},1}^{m+1} \dot{f}^m(n_1^m),\ w_{d^{m+1},2}^{m+1} \dot{f}^m(n_2^m), ..., w_{d^{m+1},d^m}^{m+1} \dot{f}^m(n_{d^m}^m) \right] \end{array} \right\}_q \tag{31}$$

La expresión (31) se puede escribir de manera equivalente:

$$\left(\frac{\partial \mathbf{n}^{m+1}}{\partial \mathbf{n}^{m}}\right)_q = \left\{ \begin{pmatrix} w_{1,1} & \cdots & w_{1,d^m} \\ \vdots & \ddots & \vdots \\ w_{d^{m+1},1} & \cdots & w_{d^{m+1},d^m} \end{pmatrix}^{m+1} \begin{pmatrix} \dot{f}^m(n_1^m) & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \dot{f}^m(n_{d^m}^m) \end{pmatrix} \right\}_q = \left\{ W^{m+1} \dot{\mathbf{F}}^m(\mathbf{n}^m) \right\}_q \quad (32)$$

ya que el producto de una matriz A por una matriz diagonal a su derecha equivale a multiplicar cada columna de A por el término diagonal correspondiente.

Entonces la fórmula recurrente para una muestra será:

$$\left\{ \mathbf{s}^m = \left(\frac{\partial \mathbf{n}^{m+1}}{\partial \mathbf{n}^{m}}\right)^T \mathbf{s}^{m+1} = \dot{\mathbf{F}}^m(\mathbf{n}^m)\left(W^{m+1}\right)^T \mathbf{s}^{m+1} \right\}_q \quad (33)$$

Definiendo la sensibilidad de todas las muestras:

$$\mathbf{s}^m = \left[\left\{\mathbf{s}^m\right\}_1, \ldots, \left\{\mathbf{s}^m\right\}_Q \right] \rightarrow (d^m \text{ x} 1 \cdot Q) \quad (34)$$

la fórmula regresiva (33) se extiende a todas las muestras:

$$\mathbf{s}^m = \dot{\mathbf{F}}^m(\mathbf{n}^m)\left(W^{m+1}\right)^T \mathbf{s}^{m+1} \quad (35)$$

La fórmula (35) se podrá ejecutar simultáneamente para todas las muestras mediante la multiplicación matricial elemento a elemento descrita en el apartado "Implementación en Matlab".

Cálculo del vector de sensibilidad de la última capa

La fórmula recurrente (35) requiere concretar la sensibilidad $\mathbf{s}^M$ de la última capa:

$$\left\{\mathbf{s}^M\right\}_q = \left\{\frac{\partial E_c}{\partial \mathbf{n}^M}\right\}_q = \left\{ \begin{matrix} \dfrac{\partial}{\partial n_1^M} \displaystyle\sum_{k=1}^{d^M} (t_k - a_k)^2 \\ \vdots \\ \dfrac{\partial}{\partial n_{d^M}^M} \displaystyle\sum_{k=1}^{d^M} (t_k - a_k)^2 \end{matrix} \right\}_q$$

Teniendo en cuenta que:

$$\left\{\mathbf{a}\right\}_q = \left\{ f^M(\mathbf{n}^M) \right\}_q = \left\{ \begin{matrix} a_1 = f^M\left(n_1^M\right) \\ \vdots \\ a_{d^M} = f^M\left(n_{d^M}^M\right) \end{matrix} \right\}_q \rightarrow \left\{ \frac{\partial a_k}{\partial n_j^M} = \begin{cases} \dfrac{\partial a_j}{\partial n_j^M} = \dot{f}^M\left(n_j^M\right) & \text{si } j = k \\ 0 & \text{si } j \neq k \end{cases} \right\}_q$$

Las componentes del vector sensibilidad serán:

$$\left\{\frac{\partial}{\partial n_j^M}\sum_{k=1}^{d^M}(t_k-a_k)^2\right\}_q=\left\{-2\sum_{k=1}^{d^M}(t_k-a_k)\frac{\partial a_k}{\partial n_j^M}\right\}_q=\left\{-2(t_j-a_j)\frac{\partial a_j}{\partial n_j^M}\right\}_q=\left\{-2(t_j-a_j)\,\dot f^M(n_j^M)\right\}_q$$

Entonces:
$$\left\{\mathbf{s}^M\right\}_q=\left\{-2\begin{pmatrix}(t_1-a_1)\cdot\dot f^M(n_1^M)\\ \vdots\\ \left(t_{d^M}-a_{d^M}\right)\cdot\dot f^M(n_{d^M}^M)\end{pmatrix}\right\}_q \tag{36}$$

Definiendo la sensibilidad para todas las muestras:

$$\mathbf{s}^M=\left[\left\{\mathbf{s}^M\right\}_1,\ldots,\left\{\mathbf{s}^M\right\}_Q\right]$$

podemos extender el cálculo (36) a todas las muestras de manera simultánea, si se utiliza la multiplicación elemento a elemento como se indica en el apartado "Implementación en Matlab".

Cálculo de la matriz jacobiana del error

El método de Gauss-Newton y su mejora en el método de Levenberg-Marquardt requieren el cálculo de la matriz jacobiana: $j=\dfrac{\partial\mathbf{e}}{\partial\mathbf{x}}\begin{cases}\mathbf{e} & \text{es el vector error}\\ \mathbf{x} & \text{son las incógnitas (pesos y sesgos)}\end{cases}$

Para una única muestra, las dimensiones de $\left\{j=\dfrac{\partial\mathbf{e}}{\partial\mathbf{x}}\right\}_q$ serán $\left(d^M\,\mathrm{x}\,nx\right)$, donde nx es el número total de pesos.

Para Q muestras, las dimensiones de $j=\begin{bmatrix}\left\{\dfrac{\partial\mathbf{e}}{\partial\mathbf{x}}\right\}_1\\ \vdots\\ \left\{\dfrac{\partial\mathbf{e}}{\partial\mathbf{x}}\right\}_Q\end{bmatrix}$ serán $\left(d^M\cdot Q\ \mathrm{x}\ nx\right)$.

Analizamos en primer lugar, la aportación a j de las derivadas de los pesos de la capa m:

$$j_w^m=\frac{\partial\mathbf{e}}{\partial w_{ij}^m}\begin{cases}\mathbf{e} & \to(\mathrm{d}^M\cdot Q\,x1)\\ w_{ij}^m & \to(d^m\cdot d^{m-1}\,\mathrm{x}\,1)\end{cases}\to\left(\mathrm{d}^M\cdot\ Q\ \mathrm{x}\ d^m\cdot d^{m-1}\right) \tag{37}$$

Trabajaremos primero con una muestra y luego haremos la extensión a Q muestras.

Desarrollaremos primero la matriz traspuesta $\left\{j_w^m\right\}_q^T$ que reconvertiremos al final a $\left\{j_w^m\right\}_q$

$$\left\{j_w^m\right\}_q^T = \left\{\frac{\partial e_1}{\partial w_{ij}^m},\ldots,\frac{\partial e_{d^M}}{\partial w_{ij}^m}\right\}_q$$

Cada columna es un vector gradiente que trataremos igual que en el apartado anterior, con la única diferencia que antes la función escalar era el error cuadrático con un doble sumatorio sobre todos los errores elevados al cuadrado y ahora es simplemente una función error, pero repetida en columnas para todas las neuronas $\left(\mathrm{d}^M\right)$ de la última capa. Entonces, de acuerdo con (22), (24) y (25):

$$\left\{j_w^m\right\}_q^T = \left\{\frac{\partial e_1}{\partial w_{ij}^m},\ldots,\frac{\partial e_{d^M}}{\partial w_{ij}^m}\right\}_q = \left\{\begin{array}{c}\left(\frac{\partial e_1}{\partial \mathbf{n}^m},\ldots,\frac{\partial e_{d^M}}{\partial \mathbf{n}^m}\right)a_1^{m-1}\\ \vdots \\ \left(\frac{\partial e_1}{\partial \mathbf{n}^m},\ldots,\frac{\partial e_{d^M}}{\partial \mathbf{n}^m}\right)a_{d^{m-1}}^{m-1}\end{array}\right\}_q = \left\{\begin{array}{c}\mathbf{S}^m a_1^{m-1}\\ \mathbf{S}^m a_2^{m-1}\\ \vdots \\ \mathbf{S}^m a_{d^{m-1}}^{m-1}\end{array}\right\}_q \quad (38)$$

donde ahora la sensibilidad es una matriz de d^M columnas porque hay d^M errores:

$$\left\{\mathbf{S}^m\right\}_q = \left\{\left(\frac{\partial e_1}{\partial \mathbf{n}^m},\ldots,\frac{\partial e_{d^M}}{\partial \mathbf{n}^m}\right)\right\}_q \quad (39)$$

La expresión (38) también se puede generalizar y calcular simultáneamente para todas las muestras, en la forma que posteriormente veremos en el apartado "Implementación en Matlab":

$$\left\{j_w^m\right\}_q^T \rightarrow \left\{j_w^m\right\}^T = \left\{\left\{\begin{array}{c}\mathbf{S}^m a_1^{m-1}\\ \mathbf{S}^m a_2^{m-1}\\ \vdots \\ \mathbf{S}^m a_{d^{m-1}}^{m-1}\end{array}\right\}_1,\ldots,\left\{\begin{array}{c}\mathbf{S}^m a_1^{m-1}\\ \mathbf{S}^m a_2^{m-1}\\ \vdots \\ \mathbf{S}^m a_{d^{m-1}}^{m-1}\end{array}\right\}_Q\right\} \left(d^m \cdot d^{m-1}\,\mathrm{x}\,d^M \cdot Q\right) \quad (40)$$

Obtenida la matriz $\left\{j_w^m\right\}^T$ la traspondremos para obtener $j_w^m \rightarrow (d^M \cdot Q\,\mathrm{x}\,d^m \cdot d^{m-1})$ que es lo que buscamos.

Lo mismo, pero de manera más sencilla ocurre con las derivadas respecto a los sesgos (**b**), ya que de (23) se deduce que la matriz $\left(\begin{array}{ccc}\frac{\partial n_1^m}{\partial b_1^m}=1 & \cdots & \frac{\partial n_{d^m}^m}{\partial b_1^m}=0\\ \vdots & \ddots & \vdots \\ \frac{\partial n_{d^m}^m}{\partial b_{d^m}^m}=0 & \cdots & \frac{\partial n_{d^m}^m}{\partial b_{d^m}^m}=1\end{array}\right)_q$ es la matriz identidad y por tanto:

$$\left\{j_b^m\right\}=\left\{\begin{Bmatrix}\mathbf{S}^m\\ \mathbf{S}^m\\ \vdots\\ \mathbf{S}^m\end{Bmatrix}_1,\ldots,\begin{Bmatrix}\mathbf{S}^m\\ \mathbf{S}^m\\ \vdots\\ \mathbf{S}^m\end{Bmatrix}_Q\right\}^T\rightarrow(\,d^M\cdot Q\,\mathrm{x}\,d^m)$$

Resuelta la capa genérica m, organizaremos el vector gradiente completo con las derivadas respecto a los pesos y sesgos de todas las capas, en el siguiente vector columna:

$$j^m=\left(\left[\,j_b^1,j_w^1\right],\ldots,\left[\,j_b^m,j_w^m\right],\ldots,\left[\,j_b^M,j_w^M\right]\right)^T\rightarrow\left(d^M\cdot Q\,\mathrm{x}\,nx\right)$$

donde nx es el número total de pesos. Obsérvese, que a diferencia del gradiente no ha hecho falta sumar en muestras debido a que las derivadas se toman de los errores y no de la suma de los errores cuadráticos. No obstante, sí aparecen en j^m las derivadas de los errores de todas las muestras, adosadas verticalmente por filas.

Fórmula recursiva para la matriz de sensibilidad

Para hacer lo anterior hay que precalcular la matriz de sensibilidad, $\left\{\mathbf{S}^m\right\}_q=\left\{\left(\frac{\partial e_1}{\partial \mathbf{n}^m},\ldots,\frac{\partial e_{d^M}}{\partial \mathbf{n}^m}\right)\right\}_q$ que también se extiende a todas las muestras:

$$\mathbf{S}^m=\left[\left\{\mathbf{S}^m\right\}_1,\ldots,\left\{\mathbf{S}^m\right\}_Q\right]\rightarrow(d^m\,\mathrm{x}\,d^M\cdot Q)\tag{41}$$

Las dimensiones de $\left\{\mathbf{S}^m\right\}_q$ son ahora $\left(d^m\,\mathrm{x}\,d^M\right)$ y las de $\mathbf{S}^m$ pasan a ser $\left(d^m\,\mathrm{x}\,d^M\cdot Q\right)$. La diferencia entre las matrices de sensibilidad (39) y (41) y las definidas para el gradiente en (29) y (34) solo afectan al número de columnas, y en consecuencia también es aplicable la fórmula regresiva (35) que ahora toma la forma:

$$\mathbf{S}^m=\dot{\bar{\mathbf{F}}}^m(\mathbf{n}^m)\left(W^{m+1}\right)^T\mathbf{S}^{m+1}\tag{42}$$

ya que definida en (32), $\dot{\mathbf{F}}^m(\mathbf{n}^m)$ se repite ahora d^M veces, tantas como errores, y define:

$$\dot{\bar{\mathbf{F}}}^m(\mathbf{n}^m)\rightarrow\left[\dot{\mathbf{F}}^m(\mathbf{n}^m),\ldots,\dot{\mathbf{F}}^m(\mathbf{n}^m)\right]_{d^M}\tag{43}$$

Se podrá ejecutar la fórmula (42) simultáneamente para todas las muestras, mediante la multiplicación matricial elemento a elemento descrita en el apartado "Implementación en Matlab".

Las fórmulas recurrentes (35) y (42) son similares para gradiente y matriz jacobiana. La diferencia solo se manifiesta en la última capa que es la que da inicio a las iteraciones recursivas.

Cálculo de la matriz de sensibilidad de la última capa

Finalmente, la fórmula recurrente (42) requiere concretar la sensibilidad $\mathbf{S}^M$ de la última capa:

$$\left\{\mathbf{S}^M\right\}_q = \left\{\left(\frac{\partial e_1}{\partial \mathbf{n}^M},\ldots,\frac{\partial e_{d^M}}{\partial \mathbf{n}^M}\right)\right\}_q = \left\{\frac{\partial}{\partial \mathbf{n}^M}\left(\left(t_1 - a_1^M\right),\ldots,\left(t_{d^M} - a_{d^M}^M\right)\right)\right\}_q =$$

$$= \left\{\begin{pmatrix} \left(-\frac{\partial a_1^M}{\partial n_1^M}\right) & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \left(-\frac{\partial a_{d^M}^M}{\partial n_{d^M}^M}\right) \end{pmatrix}\right\}_q = \left\{\begin{pmatrix} \dot{f}^M(n_1^M) & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \dot{f}^M(n_{d^M}^M) \end{pmatrix}\right\}_q = \left\{\dot{\mathbf{F}}^M(\mathbf{n}^M)\right\}_q$$

Y para todas las muestras, la extensión también es inmediata si se utiliza la multiplicación elemento a elemento, como se indica en el apartado "Implementación en Matlab", para obtener:

$$\mathbf{S}^M = \left[\left\{\mathbf{S}^M\right\}_1,\ldots,\left\{\mathbf{S}^M\right\}_Q\right] = \left[\left\{\dot{\mathbf{F}}^M(\mathbf{n}^M)\right\}_1,\ldots,\left\{\dot{\mathbf{F}}^M(\mathbf{n}^M)\right\}_Q\right] \rightarrow \left(d^M \text{ x } d^M \cdot Q\right) \qquad (44)$$

Implementación en Matlab

Bucle for en todas las muestras es oneroso en Matlab que es un intérprete y conviene sustituirlo siempre que es posible por una multiplicación matricial del tipo elemento a elemento, lo cual puede implicar ahorros muy considerables de tiempo computacional.

En Matlab la multiplicación elemento a elemento de dos matrices se expresa con (.*), y se define a continuación: Sean $A_{(m \text{ x } n)}$ y $B_{(m \text{ x } n)}$ dos matrices, entonces el producto elemento a elemento de A y B es otra matriz $C_{(m \text{ x } n)}$, formada al multiplicar cada elemento de A por cada elemento de B:

$$C = A.*B = \begin{pmatrix} a_{1,1}\cdot b_{1,1} & \cdots & a_{1,n}\cdot b_{1,n} \\ \vdots & \ddots & \vdots \\ a_{m,1}\cdot b_{m,1} & \cdots & a_{m,n}\cdot b_{m,n} \end{pmatrix}$$

Como todas las operaciones matriciales de Matlab, se realiza en lenguaje máquina muy eficiente y conlleva importantes ahorros computacionales frente a la alternativa de la programación con bucles.

Otra operación matricial muy eficiente de Matlab es el producto tensorial de Kronecker de dos matrices que se expresa con $(\otimes)$ y se define a continuación:

Sean $A_{(m \text{ x } n)}$ y $B_{(\text{p x q})}$ matrices, entonces el producto tensorial de A y B es otra matriz $C_{(m\cdot p \text{ x } n\cdot q)}$ mayor formada al multiplicar B por cada elemento de A:

$$C = A \otimes B = \begin{pmatrix} a_{1,1} \cdot B & \cdots & a_{1,n} \cdot B \\ \vdots & \ddots & \vdots \\ a_{m,1} \cdot B & \cdots & a_{m,n} \cdot B \end{pmatrix}$$

En el cálculo del gradiente

Para la expresión (27): $$\left\{\nabla_w^m E_c\right\} \rightarrow \left\{ \begin{pmatrix} \mathbf{s}^m a_1^{m-1} \\ \vdots \\ \mathbf{s}^m a_{d^{m-1}}^{-1} \end{pmatrix}_1, ..., \begin{pmatrix} \mathbf{s}^m a_1^{m-1} \\ \vdots \\ \mathbf{s}^m a_{d^{m-1}}^{-1} \end{pmatrix}_Q \right\} (d^m \cdot d^{m-1} \text{ x } Q)$$

Debemos recordar que: $$\begin{cases} \mathbf{a}^{m-1} = \left[\left\{\mathbf{a}^{m-1}\right\}_1, ..., \left\{\mathbf{a}^{m-1}\right\}_Q \right] \rightarrow (d^{m-1} \text{ x } 1 \cdot Q) \\ \mathbf{s}^m = \left[\left\{\mathbf{s}^m\right\}_1, ..., \left\{\mathbf{s}^m\right\}_Q \right] \quad \rightarrow (d^m \text{ x} 1 \cdot Q) \end{cases}$$

Por tanto, tanto el vector $\mathbf{a}^{m-1}$ como la matriz $\mathbf{s}^m$ ya contienen la información de todas las muestras. Sin embargo, antes de multiplicarlos elemento a elemento (.*) hay que darles forma para que la operación sea coherente dimensionalmente y refleje fielmente (27). Para ello la herramienta matricial eficiente es el producto de Kroneker.

Por un lado, debemos reflejar que hay d^{m-1} bloques $\mathbf{s}^m$ adosados en la dirección vertical, que podemos componer de la siguiente manera:

$$\left[\begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix}_{d^{m-1}} \otimes \mathbf{s}^m \right] = \begin{pmatrix} \mathbf{s}^m \\ \vdots \\ \mathbf{s}^m \end{pmatrix}_{d^{m-1}} \rightarrow \left(d^m \cdot d^{m-1} \text{ x} 1 \cdot Q\right)$$

Por otro lado, cada vector de sensibilidad $\left\{\mathbf{s}^m\right\}_q \rightarrow (d^m \text{ x} 1)$ debe multiplicarse por el correspondiente elemento de $\left\{\mathbf{a}^{m-1}\right\}_q$. Por ejemplo, el vector k de (27) deberá hacerlo por $\left\{a_k^{m-1}\right\}_q$. Para habilitar el posterior producto elemento a elemento, generamos un vector constante con las dimensiones de $\left\{\mathbf{s}^m\right\}_q$: $\left\{a_k^{m-1} \cdot \mathbf{1}_{d^m}\right\}_q = \left\{a_k^{m-1}\right\}_q \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix}_{d^m} \rightarrow (d^m \text{ x} 1)$, donde $\mathbf{1}_{d^m}$ es un vector columna con d^m valores unidad.

Esta operación puede generalizarse para el vector $\mathbf{a}^{m-1}$ completo y con todas las muestras, mediante el siguiente producto de Kroneker.

$$\left[\mathbf{a}^{m-1}\otimes\mathbf{1}_{d^m}\right]=\left[\begin{bmatrix}\mathbf{1}_{d^m}a_1^{m-1}\\ \vdots\\ \mathbf{1}_{d^m}a_{d^{m-1}}^{m-1}\end{bmatrix}_1,\ldots,\begin{bmatrix}\mathbf{1}_{d^m}a_1^{m-1}\\ \vdots\\ \mathbf{1}_{d^m}a_{d^{m-1}}^{m-1}\end{bmatrix}_Q\right]\rightarrow\left(d^{m-1}\cdot d^m\,\mathrm{x}\,1\cdot Q\right)$$

Ya estamos en condiciones de realizar la multiplicación elemento a elemento, para obtener

(27): $$\left\{\nabla_w^m E_c\right\}=\left[\begin{pmatrix}1\\ \vdots\\ 1\end{pmatrix}_{d^{m-1}}\otimes\mathbf{s}^m\right].*\left[\mathbf{a}^{m-1}\otimes\begin{pmatrix}1\\ \vdots\\ 1\end{pmatrix}_{d^m}\right]\rightarrow(d^{m-1}\cdot d^m\,\mathrm{x}\,Q).*(d^{m-1}\cdot d^m\,\mathrm{x}\,Q)$$

En el cálculo de la sensibilidad del gradiente

Para la expresión (35): $$\mathbf{s}^m=\dot{\mathbf{F}}^m(\mathbf{n}^m)\left(W^{m+1}\right)^T\mathbf{s}^{m+1}\rightarrow(d^m\,\mathrm{x}\,1\cdot Q)$$

tendremos en cuenta que el producto de una matriz A multiplicada a su izquierda por una matriz diagonal, equivale a multiplicar cada fila de A por el término diagonal correspondiente.

Como el producto $\left(W^{m+1}\right)^T\cdot\mathbf{s}^{m+1}$ es un vector columna, la expresión (35) se puede evaluar directamente elemento a elemento (.*) para una muestra q:

$$\left\{\mathbf{s}^m\right\}_q=\left\{\left[\begin{pmatrix}\dot{f}(n_1)\\ \vdots\\ \dot{f}(n_{d^m})\end{pmatrix}^m\right].*\left[\left(W^{m+1}\right)^T\mathbf{s}^{m+1}\right]\right\}_q\rightarrow\left(d^m\,\mathrm{x}\,1\right).*\left[\left(d^m\,\mathrm{x}\,d^{m+1}\right)*\left(d^{m+1}\,\mathrm{x}\,1\right)\right]=\left(d^m\,\mathrm{x}\,1\right)$$

y también para todas las muestras:

$$\mathbf{s}^m=\left[\begin{pmatrix}\dot{f}^m(n_1)\\ \vdots\\ \dot{f}^m(n_{d^m})\end{pmatrix}\right].*\left[\left(W^{m+1}\right)^T\mathbf{s}^{m+1}\right]\rightarrow\left(d^m\,\mathrm{x}\,Q\right).*\left(d^m\,\mathrm{x}\,Q\right)\qquad(45)$$

ya que $\dot{\mathbf{F}}(\mathbf{n}^m)$ y $\mathbf{s}^{m+1}$ ya están extendidas a todas las muestras.

En el cálculo de la sensibilidad del gradiente de la última capa

Para la expresión (36): $$\left\{\mathbf{s}^M\right\}_q=\left\{-2\begin{pmatrix}(t_1-a_1)\cdot\dot{f}^M(n_1^M)\\ \vdots\\ \left(t_{d^M}-a_{d^M}\right)\cdot\dot{f}^M(n_{d^M}^M)\end{pmatrix}\right\}_q$$

La extensión a todas las muestras es inmediata:

$$\mathbf{s}^M = \left[\left\{\mathbf{s}^M\right\}_1,\ldots,\left\{\mathbf{s}^M\right\}_Q\right] = -2(\mathbf{t}-\mathbf{a}).*\begin{pmatrix}\dot{f}^M(n_1^M)\\ \vdots \\ \dot{f}^M(n_{d^M}^M)\end{pmatrix} \rightarrow \left(d^M \text{ x } 1\cdot Q\right).*\left(d^M \text{ x } 1\cdot Q\right)$$

ya que $\mathbf{t}$, $\mathbf{a}$ y $\dot{\mathbf{F}}^M(\mathbf{n}^M)$ ya están extendidas a todas las muestras.

En el cálculo de la matriz jacobiana del error

Para la expresión (40): $$\left\{j_w^m\right\}^T = \left\{\begin{Bmatrix}\mathbf{S}^m a_1^{m-1}\\ \mathbf{S}^m a_2^{m-1}\\ \vdots \\ \mathbf{S}^m a_{d^{m-1}}^{m-1}\end{Bmatrix}_1,\ldots,\begin{Bmatrix}\mathbf{S}^m a_1^{m-1}\\ \mathbf{S}^m a_2^{m-1}\\ \vdots \\ \mathbf{S}^m a_{d^{m-1}}^{m-1}\end{Bmatrix}_Q\right\} (d^m\cdot d^{m-1} \text{ x } d^M\cdot Q)$$

Debemos recordar que: $$\begin{cases}\mathbf{a}^{m-1} = \left[\left\{\mathbf{a}^{m-1}\right\}_1,\ldots,\left\{\mathbf{a}^{m-1}\right\}_Q\right] \rightarrow (d^{m-1} \text{ x } 1\cdot Q)\\ \mathbf{S}^m = \left[\left\{\mathbf{S}^m\right\}_1,\ldots,\left\{\mathbf{S}^m\right\}_Q\right] \quad \rightarrow (d^m \text{ x } d^M\cdot Q)\end{cases}$$

Al igual que sucedía en el cálculo del gradiente, tanto el vector $\mathbf{a}^{m-1}$ como la matriz $\mathbf{S}^m$ ya contienen la información de todas las muestras. Sin embargo, antes de multiplicarlos elemento a elemento (.*) hay que darles forma con el producto de Kroneker para obtener (40).

Por un lado, debemos reflejar que hay d^{m-1} bloques $\mathbf{S}^m$ adosados en la dirección vertical, y debemos repetir la matriz de sensibilidad de la siguiente manera:

$$\left[\begin{pmatrix}1\\ \vdots \\ 1\end{pmatrix}_{d^{m-1}} \otimes \mathbf{S}^m\right] = \begin{pmatrix}\mathbf{S}^m\\ \vdots \\ \mathbf{S}^m\end{pmatrix}_{d^{m-1}} \rightarrow \left(d^m\cdot d^{m-1} \text{ x } d^M\cdot Q\right) \qquad (46)$$

Por otro lado, cada bloque de sensibilidad $\left\{\mathbf{S}^m\right\}_q \rightarrow (d^m \text{ x } d^M)$ debe multiplicarse por el correspondiente elemento de $\left\{\mathbf{a}^{m-1}\right\}_q$. Por ejemplo, el bloque k de (40) deberá hacerlo por $\left\{a_k^{m-1}\right\}_q$. Para habilitar el posterior producto elemento a elemento, generamos el siguiente bloque constante con las dimensiones de $\left\{\mathbf{S}^m\right\}_q$:

$$\left\{a_k^{m-1}\cdot\mathbf{1}_{\left(d^m \text{x } d^M\right)}\right\}_q = \left\{a_k^{m-1}\right\}_q \begin{pmatrix}1_{1,1} & \cdots & 1_{1,d^M}\\ \vdots & \ddots & \vdots \\ 1_{d^m,1} & \cdots & 1_{d^m,d^M}\end{pmatrix} \rightarrow (d^m \text{ x } d^M)$$

Bloque que puede generalizarse para el vector $\mathbf{a}^{m-1}$ completo y con todas las muestras, de la siguiente manera:

$$\left[\mathbf{a}^{m-1}\otimes\mathbf{1}_{\left(d^m \mathrm{x}\, d^M\right)}\right]=\left[\left\{\begin{matrix} a_1^{m-1}\cdot\mathbf{1}_{\left(d^m \mathrm{x}\, d^M\right)} \\ \vdots \\ a_{d^{m-1}}^{m-1}\cdot\mathbf{1}_{\left(d^m \mathrm{x}\, d^M\right)} \end{matrix}\right\}_1,\ldots,\left\{\begin{matrix} a_1^{m-1}\cdot\mathbf{1}_{\left(d^m \mathrm{x}\, d^M\right)} \\ \vdots \\ a_{d^{m-1}}^{m-1}\cdot\mathbf{1}_{\left(d^m \mathrm{x}\, d^M\right)} \end{matrix}\right\}_Q\right]\rightarrow\left(d^{m-1}\cdot d^m \mathrm{x}\, d^M\cdot Q\right) \qquad (47)$$

Finalmente estamos en condiciones de realizar la multiplicación elemento a elemento, para obtener (40):

$$\left\{j_w^m\right\}^T=\left[\begin{pmatrix}1\\ \vdots \\ 1\end{pmatrix}_{d^{m-1}}\otimes\mathbf{S}^m\right].*\left[\mathbf{a}^{m-1}\otimes\mathbf{1}_{\left(d^M \mathrm{x}\, d^m\right)}\right]\rightarrow\left(d^{m-1}\cdot d^m \mathrm{x}\, d^M\cdot Q\right).*\left(\left[d^{m-1}\cdot d^m \mathrm{x}\; d^M\cdot Q\right]\right) \qquad (48)$$

Trasponiendo, obtenemos la matriz jacobiana buscada: $j_w^m\rightarrow\left(d^M\cdot Q \mathrm{x}\, d^{m-1}\cdot d^m\right)$ cuyas dimensiones coinciden con (37), como debe ser.

En el cálculo de la sensibilidad de la matriz jacobiana del error

Para la expresión (42): $\mathbf{S}^m=\dot{\tilde{\mathbf{F}}}^m(\mathbf{n}^m)\left(W^{m+1}\right)^T\mathbf{S}^{m+1}$, *con* $\dot{\tilde{\mathbf{F}}}^m(\mathbf{n}^m)\rightarrow\left[\dot{\mathbf{F}}^m(\mathbf{n}^m),\ldots,\dot{\mathbf{F}}^m(\mathbf{n}^m)\right]_{d^M}$.

Teniendo en cuenta que $\left(\dot{f}^M(n_1),\cdots,\dot{f}^M(n_{d^M})\right)$ ya están extendidas a todas las muestras:

$$\dot{\tilde{\mathbf{F}}}^m(\mathbf{n}^m)\rightarrow\left[\begin{pmatrix}\dot{f}^M(n_1)\\ \vdots \\ \dot{f}^M(n_{d^m})\end{pmatrix}\otimes(1,\ldots,1)_{d^M}\right]=$$

$$=\left[\left\{\begin{pmatrix}\dot{f}^M(n_1),\ldots,\dot{f}^M(n_1)\\ \vdots \\ \dot{f}^M(n_{d^m}),\ldots,\dot{f}^M(n_{d^m})\end{pmatrix}_{d^M}\right\}_1,\ldots,\left\{\begin{pmatrix}\dot{f}^M(n_1),\ldots,\dot{f}^M(n_1)\\ \vdots \\ \dot{f}^M(n_{d^m}),\ldots,\dot{f}^M(n_{d^m})\end{pmatrix}_{d^M}\right\}_Q\right]\rightarrow\left(d^m \mathrm{x}\, d^M\cdot Q\right) \qquad (49)$$

Teniendo en cuenta que $\mathbf{S}^{m+1}$ de dimensión $\left(d^{m+1} \mathrm{x}\, d^M\cdot Q\right)$ también está extendida a todas las muestras, y que las dimensiones de W^{m+1} son $\left(d^{m+1} \mathrm{x}\, d^m\right)$, su producto $\left(W^{m+1}\right)^T\cdot\mathbf{S}^{m+1}$ tiene dimensión: $\left(d^m \mathrm{x}\, d^{m+1}\right)\mathrm{x}\,\left(d^{m+1} \mathrm{x}\, d^M\cdot Q\right)=\left(d^m \mathrm{x}\, d^M\cdot Q\right)$, que coincidente con (49), como corresponde para poder realizar el producto matricial elemento a elemento.

$$\mathbf{S}^m = \left[\begin{pmatrix} \dot{f}(n_1) \\ \vdots \\ \dot{f}(n_{d^m}) \end{pmatrix}^m \otimes (1,\ldots,1)_{d^M} \right] .* \left[\left(W^{m+1}\right)^T \mathbf{S}^{m+1} \right] \rightarrow \left(d^m \,\mathrm{x}\, d^M \cdot Q\right) .* \left(d^m \,\mathrm{x}\, d^M \cdot Q\right) \qquad (50)$$

En el cálculo de la sensibilidad de la matriz jacobiana del error, para la última capa

Para la expresión (44):

$$\mathbf{S}^M = \left[\left\{\mathbf{S}^M\right\}_1, \ldots, \left\{\mathbf{S}^M\right\}_Q \right] = \left[\left\{\dot{\mathbf{F}}^M(\mathbf{n}^M)\right\}_1, \ldots, \left\{\dot{\mathbf{F}}^M(\mathbf{n}^M)\right\}_Q \right] \rightarrow \left(d^M \,\mathrm{x}\, d^M \cdot Q\right)$$

Para una muestra q:

$$\left\{\dot{\mathbf{F}}^M(\mathbf{n}^M)\right\}_q = \left\{ \begin{pmatrix} \dot{f}^M(n_1^M) & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \dot{f}^M(n_{d^M}^M) \end{pmatrix} \right\}_q = \left\{ \begin{pmatrix} \dot{f}^M(n_1),\ldots,\dot{f}^M(n_1) \\ \vdots \\ \dot{f}^M(n_{d^M}),\ldots,\dot{f}^M(n_{d^M}) \end{pmatrix}_{d^M} .* \begin{pmatrix} 1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 1 \end{pmatrix}_{d^M} \right\}_q$$

Para todas las muestras: $\left[\begin{pmatrix} \dot{f}^M(n_1) \\ \vdots \\ \dot{f}^M(n_{d^M}) \end{pmatrix} \otimes (1,\ldots,1)_{d^M} \right] =$

$$= \left[\left\{ \begin{pmatrix} \dot{f}^M(n_1), \overset{d^M \text{ veces}}{\cdots}, \dot{f}^M(n_1) \\ \vdots \\ \dot{f}^M(n_{d^M}), \ \ldots \ , \dot{f}^M(n_{d^M}) \end{pmatrix}_{d^M} \right\}_1, \overset{Q \text{ veces}}{\cdots}, \left\{ \begin{pmatrix} \dot{f}^M(n_1),\ldots,\dot{f}^M(n_1) \\ \vdots \\ \dot{f}^M(n_{d^M}),\ldots,\dot{f}^M(n_{d^M}) \end{pmatrix}_{d^M} \right\}_Q \right] \rightarrow \left(d^M \,\mathrm{x}\, d^M \cdot Q\right)$$

donde se ha tenido en cuenta que $\left(\dot{f}^M(n_1),\cdots,\dot{f}^M(n_{d^M})\right)$ ya están extendidas a todas las muestras.

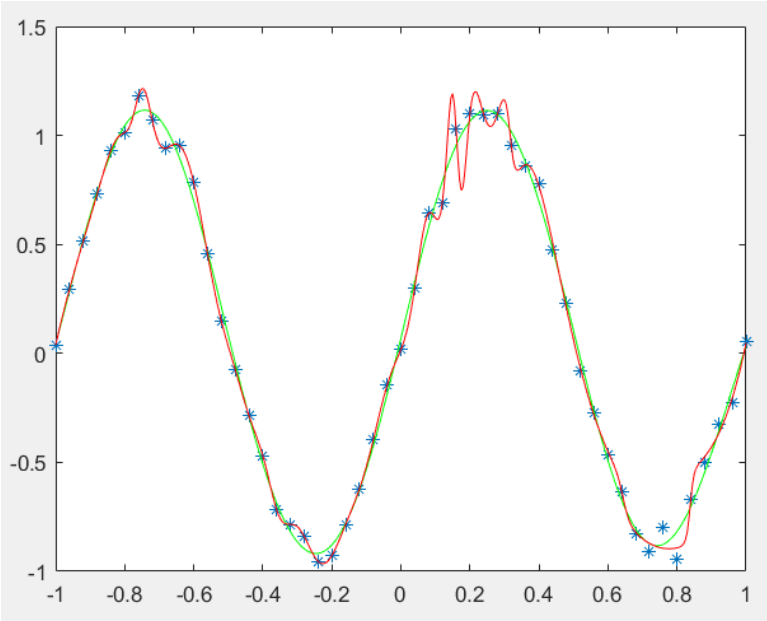
Para quedarnos solo con los miembros de la diagonal de cada uno de los bloques q, bastará multiplicar elemento a elemento por la matriz identidad en cada uno de los bloques q:

$$(1,\ldots,1)_Q \otimes \begin{pmatrix} 1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 1 \end{pmatrix}_{d^M} = \left[\begin{pmatrix} 1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 1 \end{pmatrix}_{d^M}, \overset{Q \text{ veces}}{\cdots}, \begin{pmatrix} 1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 1 \end{pmatrix}_{d^M} \right]$$

$$\mathbf{S}^M = \left[\begin{pmatrix} \dot{f}(n_1) \\ \vdots \\ \dot{f}(n_{d^m}) \end{pmatrix}^m .* \otimes (1,\ldots,1)_{d^M} \right] .* \left[(1,\ldots,1)_Q \otimes \begin{pmatrix} 1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 1 \end{pmatrix}_{d^M} \right] \rightarrow \left(d^m \,\mathrm{x}\, d^M \cdot Q\right) .* \left(d^m \,\mathrm{x}\, d^M \cdot Q\right)$$

Generalización

Uno de los principales riesgos a la hora de realizar el entrenamiento de la red es el sobreajuste (u overfitting) a los datos de la muestra, corriendo el riesgo de que no aproxime correctamente los valores externos a este conjunto. En el ejemplo siguiente tenemos la adaptación de la función $\sin(2\cdot\pi\cdot x)$, pero los datos tienen un componente aleatorio uniformemente distribuido en el intervalo (0, 0.2).



8. Ajuste función seno con error aleatorio

La línea roja se corresponde con una red entrenada con el algoritmo de Levenberg-Marquardt tradicional y la verde mediante regularización bayesiana (una variante que explicaremos más adelante). Como se puede comprobar, la roja no tiene en cuenta el factor aleatorio y se ajusta a los datos de la muestra, alejándose de la función sinusoidal. La curva verde, por su parte, omite el factor aleatorio, adaptando la función $\sin(2\cdot\pi\cdot x)+0.1$.

Comentaremos los dos métodos principales para evitar el overfitting (ver ref. [6]):

- Detención temprana (Early Stopping). Consiste en dividir el conjunto de datos en entrenamiento y validación (en una proporción, normalmente, alrededor de 80%/20%). El conjunto de entrenamiento se utiliza para calcular los pesos (como se ha explicado) y el conjunto de validación para comprobar que la red adapta correctamente los valores ajenos al conjunto de entrenamiento. Así, si durante un número determinado de iteraciones consecutivas el error cuadrático del conjunto de validación E_v aumenta, detenemos las iteraciones y asignamos los pesos que minimizaban el error del conjunto de entrenamiento.

- Regularización bayesiana. Los valores altos de los pesos son debidos a adaptaciones a puntos concretos, ya que permiten pendientes elevadas. Entonces, en los problemas de regularización se penalizan los pesos muy elevados cambiando la función objetivo:

$$E_c(\mathbf{c}) = \beta \cdot \sum_{i=1}^{n} \left(f(x_i) - y_i \right)^2 + \alpha \cdot \sum c^2 = \beta \cdot E_D + \alpha \cdot E_W$$

  con α y β parámetros que determinan la penalización a los pesos elevados.

  La regularización bayesiana, parte de la visión de los pesos como variables aleatorias, así escogeremos los pesos que maximicen la probabilidad:

  - $P(\boldsymbol{c}|D,\alpha,\beta,M) = \dfrac{P(D|\boldsymbol{c},\beta,M)\cdot P(\boldsymbol{c}|\alpha,M)}{P(D|\alpha,\beta,M)}$.

  - $P(\boldsymbol{c}|D,\alpha,\beta,M)$ es la probabilidad de los pesos, conocido el conjunto de datos, α, β y la estructura de la red (M).

  - $P(D|\boldsymbol{c},\beta,M)$ es la función de verosimilitud (probabilidad de que sucedan unos datos conocidos los pesos de la red, β y M) y suponiendo que el ruido de los datos tiene distribución gaussiana. $P(D|\boldsymbol{c},\beta,M) = \dfrac{1}{Z_D(\beta)} \cdot \exp(-\beta \cdot E_D)$.

  - $P(\boldsymbol{c}|\alpha,M)$ es la densidad a priori, indica la probabilidad de los pesos antes de recoger cualquier dato. $P(\boldsymbol{c}|\alpha,M) = \dfrac{1}{Z_D(\alpha)} \cdot \exp(-\alpha \cdot E_D)$.

  - $P(D|\alpha,\beta,M)$ es un término normalizador que no depende de **c**.

  Con todo esto, $P(\boldsymbol{c}|D,\alpha,\beta,M) = \dfrac{1}{Z_F(\alpha,\beta)} \cdot \exp(-E_c(\boldsymbol{c}))$ y el valor de los pesos que maximiza la probabilidad es el que minimiza la función objetivo.

Para el cálculo de α y β, se buscan los valores que minimicen $P(D|\alpha,\beta,M)$, de donde se obtiene:

$$\alpha = \frac{\gamma}{2 \cdot E_w(w)} \qquad \beta = \frac{N-\gamma}{2 \cdot E_D(w)}$$

donde N es el número de datos y $\gamma = n - 2 \cdot \alpha \cdot tr(H^{-1})$ es el número efectivo de parámetros (si γ es muy bajo en comparación con el número de pesos, quizá sea mejor cambiar la estructura de la red).

Así, el algoritmo de regularización bayesiana queda:

0. Inicializar α, β y **c**. Calcular E_W y E_D. Fijar γ=n y calcular α y β.
1. Realizar un paso del algoritmo de LM.
2. Calcular $\gamma = n - 2 \cdot \alpha \cdot tr(H^{-1})$, donde $H = \nabla^2 E_c(x) \approx 2 \cdot \beta \cdot J^T \cdot J + 2 \cdot \alpha \cdot I_n$, con J la matriz jacobiana de los errores del conjunto de entrenamiento.
3. Calcular $\alpha = \frac{\gamma}{2 \cdot E_w(w)}$ y $\beta = \frac{N-\gamma}{2 \cdot E_D(w)}$.
4. Repetir los pasos 1 a 3 hasta la convergencia de LM.

Otras funciones

A parte de las funciones principales de la red, existen otras complementarias, necesarias para su aplicación:

- Inicialización. Busca encontrar los mejores valores iniciales para los pesos, de tal forma que se minimicen las iteraciones del algoritmo BP. Una inicialización aleatoria de los pesos en [-1, 1] es una opción válida.

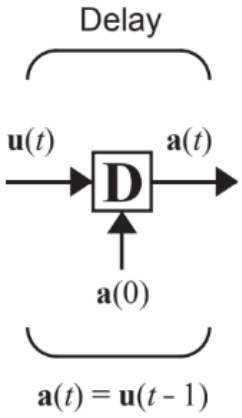
  Otra opción, propuesta por Nguyen y Widrow, consiste en hacer que cada función sigmoidea (también aplicable a otras funciones como la tangente hiperbólica) de una capa cubra una región de la zona de activación. Entonces, a cada fila j de la matriz de pesos se le asigna una dirección aleatoria y un módulo igual a $\left\|W_j^m\right\| = 0.7 \cdot \left(S^m\right)^{\frac{1}{S^{m-1}}}$. Los sesgos (b_j^m) se asignan con un valor aleatorio en $\left[-\left\|W_j^m\right\|, \left\|W_j^m\right\|\right]$.

- Pre y Post-procesado. Pretende evitar la saturación de las funciones sigmoideas, o para evitar que una variable con valores muy altos tenga más importancia que otra con valores pequeños (por ejemplo, determinar si conceder una hipoteca a partir de información como la edad del comprador (decenas) y el valor de la casa (centenas de millar), la edad podría ser desestimada al haber 'poco' cambio). Para ello, los datos de entrada y salida de la red se escalan, tipicamente, dentro del intervalo [-1,1].

Redes Dinámicas

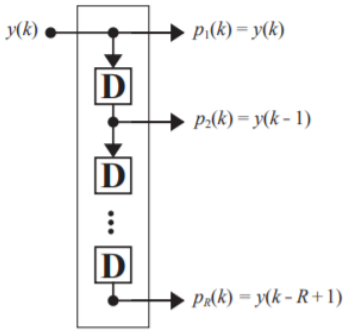
La red estudiada hasta ahora (feedforward (FF)) puede adaptar cualquier función que sea estática, pero no es capaz de adaptar funciones con una dependencia temporal. Mediante nuevas estructuras podremos realizar el control de un avión o predecir el mercado financiero. Sin embargo, conseguir esta capacidad aumenta considerablemente la complejidad de los algoritmos, por lo que el trabajo se centrará en una estructura particular, las redes NARX.

Antes de comenzar con estas redes hay que introducir el concepto de retraso ('delay'). Como se puede comprobar en la imagen siguiente, es que un bloque que da a su salida la entrada en el instante de tiempo anterior.



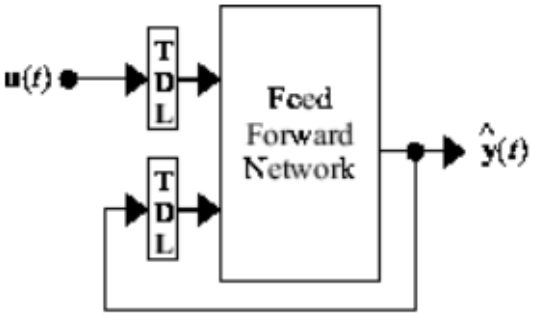
9. Esquema bloque Delay

Combinando delays podemos formar un bloque TDL que transforma la entrada **p**, adosando en un vector columna **p** en distintos instantes de tiempo anteriores. Un TDL {0,..,R-1} vendría dado por:



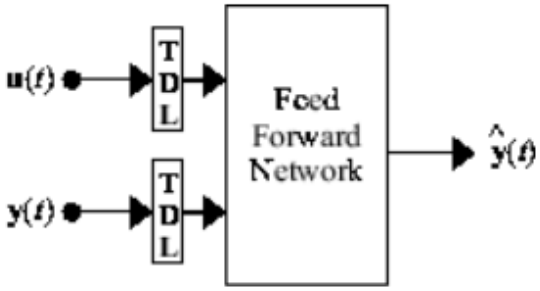
10. Esquema bloque TDL

Las redes NARX, son una clase de redes dinámicas, muy similares a las FF, pero en las que la entrada tiene retrasos y la salida se conecta a la primera capa mediante retrasos.



11. Esquema red NARX, bucle cerrado

Para realizar el entrenamiento de estas redes se abre el bucle (figura 12), convirtiendo la red en una FF, donde la entrada es la composición de **p** y **t** en tiempos anteriores, permitiendo un entrenamiento más rápido y preciso al utilizar los valores reales de la salida (targets **t**) como entradas.



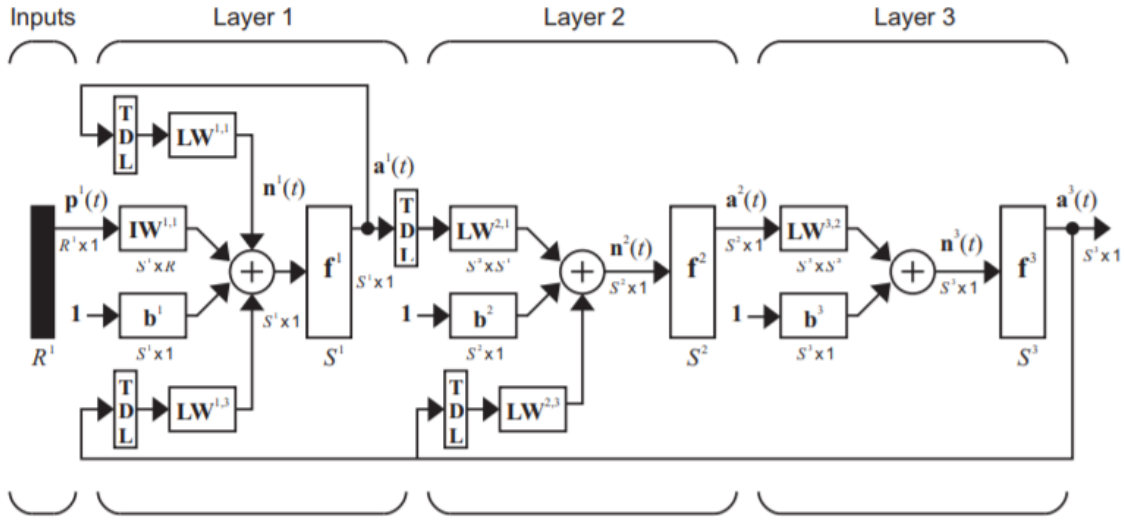
12. Esquema red NARX, bucle abierto

Por ejemplo si ambos TDLs son {1,2} de la entrada final resultante se obtienen los siguientes errores.

$$\mathbf{p} = \begin{bmatrix} \mathbf{u}(t-1) \\ \mathbf{u}(t-2) \\ \mathbf{y}(t-1) = \mathbf{t}(t-1) \\ \mathbf{y}(t-2) = \mathbf{t}(t-2) \end{bmatrix} \rightarrow \mathbf{a}(\mathbf{c}) \rightarrow \mathbf{e}(\mathbf{c}) = \mathbf{t} - \mathbf{a}(\mathbf{c})$$

Y a continuación aplicamos los métodos de entrenamiento vistos para FF. Posteriormente, entrenada la red, para la simulación de cualquier entrada se vuelve a cerrar el bucle (figura 11), y se pueden afrontar problemas como la predicción.

El entrenamiento que se acaba de explicar solo es aplicable en algunas estructuras (como las NARX), siendo, además, el que mejor resultados ofrece. Sin embargo, estrucutras más complejas (figura 13) obligan a realizar entrenamientos más complejos.



13. Ejemplo red dinámica

Todo lo explicado sobre entrenamiento puede ser aplicado a las redes dinámicas, pero a la hora de calcular las derivadas observamos que un cambio en los pesos tiene un efecto directo (el calculado en las redes estáticas) y un efecto indirecto (**a**(t-d) depende también de los pesos). En el apéndice I, podemos encontrar el algoritmo de RTRL para el cálculo del gradiente (también se calcula el Jacobiano, necesario para LM). La complejidad aumenta considerablemente, introduciéndose nuevas definiciones de capas de entrada (X) y salida (U) o nuevas agrupaciones en función de como se conectan las capas $\left(L_m^b\right)$ (se pueden encontrar en el apartado 'Preliminary definitios' y 'Sensibilities' (comienzo de la página 14-16) del tema 14 de [1], respectivamente). Además, también se complica el algoritmo, apareciendo distintos bucles for.

La interpretación realizada del cálculo del jacobiano para las redes NARX ha sido considerarlo igual al BP trabajado, pero añadiendo la última fila [1], que simplificaba considerablemente su programación, sin embargo, no hemos logrado una convergencia adecuada. Entre las posibles causas hemos encontrado el riesgo de inestabilidad en el entrenamiento dinámico.

Una aplicación característica de las redes dinámicas es la identificación de un sistema dinámico que consiste en la definición de un modelo matemático (p. ej. una ecuación diferencial) y el ajuste de los parámetros del modelo mediante experimentación. Este método muchas veces no es aplicable, entre otros motivos, impedido por las propias matemáticas que no son capaces de ofrecer funciones que describan estos comportamientos. Tampoco elimina la necesidad de la experimentación para el ajuste de los parámetros del modelo, así, otra solución consiste en la toma de muestras experimentales y, mediante métodos como los estudiados, determinar una función que los aproxime.

[1] $$\frac{\partial \mathbf{a}^u(t)}{\partial \mathbf{x}^T} = \frac{\partial^e \mathbf{a}^u(t)}{\partial \mathbf{x}^T} + \sum_{x \in E_s^X(u)} S^{u,x} \cdot \sum_{u' \in E_{LW}^U(x)} \sum_{d \in DL_{x,u'}} LW^{x,u'}(d) \times \frac{\partial \mathbf{a}^{u'}(t-d)}{\partial \mathbf{x}^T}$$

Tareas y Planificación

Búsqueda de información

La primera fase de este proyecto consiste en la búsqueda de información acerca de las redes neuronales. Utilizando [1], [2] y [3] se buscará un primer acercamiento a las redes neuronales, permitiendo marcar los objetivos del trabajo. También se utiliza y analiza la toolbox RN de Matlab, aprendiendo sobre su funcionamiento. Tiempo empleado: 2 semanas.

Backpropagation

Se busca sustituir el entrenamiento de la RN de Matlab, por uno programado por nosotros. A partir del algoritmo de Newton para aproximación de funciones, programado por Juan José Anza en 2002, se adapta al algoritmo de LM. El cálculo del Jacobiano se hace mediante el algoritmo de BP. La comprensión y programación del algoritmo son el objetivo de esta fase. Tiempo empleado: 1 mes.

Red Neuronal propia

Habiendo avanzado en el BP, eje de la red neuronal, se busca desprenderse por completo del uso de la RN de Matlab. Para ello se necesita programar otras funciones suplementarias, inicialización (aleatoria), actualización (transforma el vector de pesos usado en el entrenamiento, en las matrices de pesos y sesgos de la red) y simulación. Para ello se utilizan las estructuras de Matlab. Tiempo empleado: 2 semanas.

Otras funciones

Una vez se ha logrado una red funcional, se busca programar otras funciones que la completen (inicialización NW y RB), aumentando la variedad de problemas a abordar. Tiempo empleado: 3 semanas.

Programación orientada a objeto

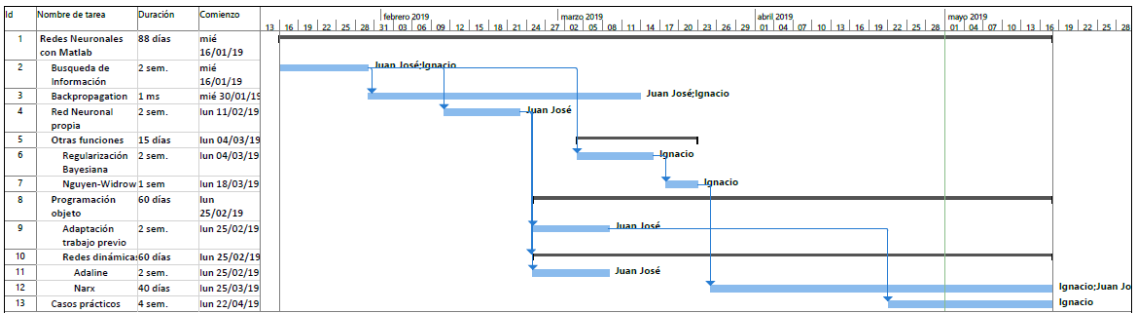
Es la parte central del trabajo. A fin de poder sacar el máximo provecho a las capacidades de Matlab, se adapta el trabajo realizado a programación orientada a objeto. El trabajo se ordena y se puede trabajar de forma sencilla, sin riesgo de estropear el trabajo previo al realizar nuevas funciones. También permite al usuario escoger las funciones a utilizar de forma sencilla desde el problema, sin tener que cambiar el programa. Tiempo empleado: 2 meses.

Redes dinámicas

El siguiente paso en las RN es su adaptación para simular comportamientos dependientes del tiempo. En esta fase la idea original buscaba aplicar la variante del algoritmo de BP, Real Time Recurrent Learning, pero no se ha conseguido. Sin embago, ello no ha sido obstáculo para el entrenamiento de las redes NARX ya que como hemos indicado anteriormente, este puede realizarse con bucle abierto obteniendo mayor precisión. Tiempo empleado: 1 mes y medio.

Casos prácticos

La fase final del trabajo consiste en estudiar las RN programadas con ejercicios prácticos, analizar las ventajas de cada método y ver sus aplicaciones en ejemplos sencillos. Tiempo empleado: 3 semanas.



14. Diagrama Gantt del trabajo[1]

El proyecto comienza el 16 de enero y finaliza el 16 de mayo, suponiendo una extensión de 4 meses.

[1] En el apéndice puede encontrarse en mayor tamaño.

Propuesta de Solución

En esta sección se describe la metodología orientada a objeto utilizada para la implementación de los algoritmos descritos en las páginas anteriores y que permiten experimentar el comportamiento de las redes neuronales (estáticas y dinámicas), tanto en modos de simulación, cálculo o ejecución, como de entrenamiento.

Motivación metodología orientada a objeto

El diseño de los objetos basado en la herencia (cuando conviene reutilizar los datos y funciones del padre) y en la agregación (cuando conviene discriminar automáticamente entre distintas opciones y facilitar incorporaciones futuras) permite proponer una metodología flexible, basada en la reutilización, la sobrecarga (mismo lenguaje) y en su caso la encapsulación.

Un objeto queda definido por sus datos (properties) y sus funciones (methods), lo cual facilita mucho la transferencia de argumentos entre funciones, en particular cuando los desarrollos crecen en complejidad, porque al transferir un objeto a una función, se transfieren en un solo argumento todos sus datos sin limitación de número, que quedan disponibles en el interior de la función.

La aplicación MATLAB está programada orientado a objeto y contiene distintos tipos de datos (data types) organizados en clases (objetos) como: double (números y matrices reales), struct (estructuras de datos) cell (celdas), etc. Matlab permite al usuario crear sus propias clases mediante ficheros .m encabezados con la palabra clave "classdeff". Cuando se ejecuta un fichero .m de nombre "redff" por ejemplo, que se inicia en primera línea con: *classdef redff* se genera en el espacio de trabajo una variable especial con el nombre arbitrario que queramos darle (por ejemplo, objred) que pertenece a la clase *redff* y que tiene dos características diferenciadoras de otros tipos de datos:

1. Contiene la estructura de datos en la sección "properties" del fichero reddff.m.
2. Tiene vinculadas un conjunto de funciones que solo se ejecutan asociadas a la clase *redff*. Por ejemplo, sal=f1(objred, ...) ejecutara a la función f1 asociada a la clase *redff* si ha sido correctamente definida. Para ello existen dos maneras:
    1) Definir la función en el fichero redff.m que contiene "classdef" en la sección "methods". El fichero redff.m deberá estar en el itinerario (path) de búsqueda de Matlab.
    2) definir la función en el fichero f1.m y guardarla en una carpeta de nombre @redff que deberá contener también el fichero redd.m.

Ambas formas se utilizan en este trabajo según circunstancias. Entre las funciones vinculadas a una clase, existe el método constructor, que deberá llevar el nombre de la clase y escribirse en redff.m s se desea invocar a la clase con datos para las propiedades. En este ejemplo la definición del método podría ser:

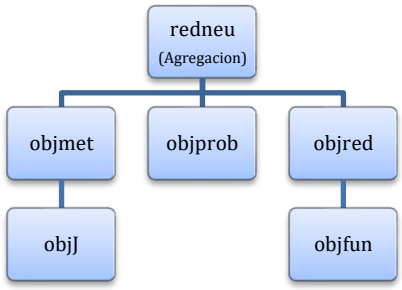
```
function obj=redff(dat1, dat2, ...)
....
end
```

de forma que si en la línea de ordenes de matlab escribimos o1=redff, obtendremos en la variable o1 un objeto *redff* sin inicializar las propiedades, mientras que si escribimos o2=redff(a1,a2, ...) obtendremos en la variable o2 un objeto *redff* con algunas o todas las propiedades inicializadas.

A continuación, se expone la arquitectura orientada a objeto resultante de este trabajo donde se distinguen tres partes:

1. Se define la red neuronal en el objeto red, especificando el número de capas, el número de neuronas por capa con sus pesos, sesgos y funciones de transferencia, lo cual permite calcular la salida **a** para cualquier entrada **p.**
2. Se define el método de entrenamiento en el objeto método que proporciona el algoritmo de resolución no lineal correspondiente.
3. Se definen en el objeto problema aspectos específicos del tipo de problema tratado: estático, dinámico, u otro tipo. En particular, se definen los objetos específicos a utilizar, de acuerdo con lo especificado por el usuario mediante un fichero de texto.

Una imagen de la organización de los objetos resultantes se muestra a continuación:



15. Estructura redneu

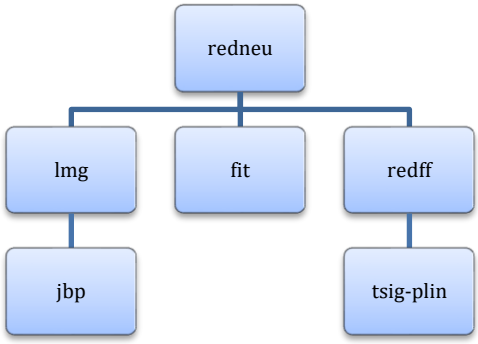
En el objeto "red neuronal" (redneu) se agregan los objetos método, problema y red. En ellos reside toda la información: datos internos y procedimientos, que permiten obtener resultados de las redes y entrenarlas.

Agregar un objeto método (objmet), por ejemplo, significa que puede ser uno cualquiera de los 4 implementados hasta este momento, según indique el usuario en la entrada de datos de su problema. Por tanto, el esquema anterior no define una única red neural sino posibilita 10 opciones distintas, ya que se han implementado 4 métodos y 2 jacobianos para las redes FF y 1 método y 2 jacobianos para las redes NARX y eso sin contabilizar que una red puede tener distinto número de capas (con distintas funciones) y de neuronas.

El siguiente guion script es un ejemplo donde se definen los datos del problema, se generan los objetos, se ordena su resolución, y se postprocesan los resultados.

```
%Guion gfit4
%-----------------
% Datos
[x,t] = simplefit_dataset;   plot(x,t,'*'),  hold on

%Definición de la red
fun={@tsig @plin};
dim=[10];   red=@redff;   tipoini='NgWi';
met=@lmg; metJ=@jbp;   prob=@fit;

p=x;  guarda_datos
%--------------------------------------

objprob=feval(@fit);

[redn,objprob,objmet,objJ,objred,objfun]=haz_obj(objprob);

[c,Ec]=entrena(redn);

objred=act(objred,c(:,end));

xx=0:0.05:10;    yy=sim(objred,xx);   plot(xx,yy)
```

La red neuronal contiene en su interior un objeto red tipo redff, con 2 capas de 10 y 1 neurona respectivamente y funciones tsig y plin; otro objeto problema del tipo "fit", y un objeto método Levenberg-Marquardt (lmg) con jacobiano calculado mediante el algoritmo "backpropagation" (jbp), para resolver las incógnitas del entrenamiento. El esquema concreto es:



16. Esquema red neuronal ejemplo

Los objetos (clases) generadas en este trabajo se agrupan en 5 tipos: red, función, método, jacobiano y problema. En cada grupo se ha definido un primer objeto instrumental que contiene datos y funciones comunes a los demás objetos del grupo, que los heredan. Su nombre comienza con la palabra base, por ejemplo, base_red y no requieren constructor porque sus datos no se inicializan de manera interactiva. La definición de clase (classdef) en ese caso ya proporciona implícitamente un constructor por defecto.

El código completo del trabajo se lista en el anexo II. En lo que sigue, se describen los principales aspectos del mismo.

Objeto Red Neuronal (redneu)

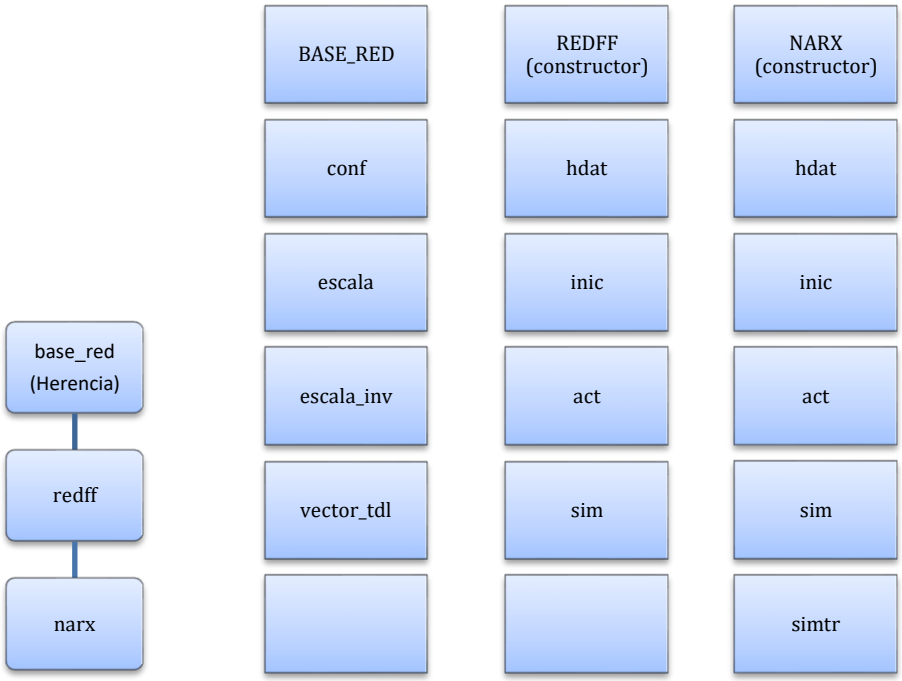
Se define el constructor del objeto redneu que, a su vez, se compone de tres objetos: objmet, objprob, objred.

También encontramos la función 'entrena', que devuelve el valor de los pesos y del error cuadrático obtenido al aplicar el método elegido.

Objetos Red

Se han implementado varios objetos red y se describen aqui los 2 principales: redff (feedforward) y narx, junto con base_red.

Base_red es un objeto instrumental que contiene datos y funciones comunes a los objetos del grupo. Redff hereda de base_red y narx de redff.



17. Estructura tipored

Los datos o propiedades que se declaran o definen son:

- R: dimensión (número de filas) del vector **p** de entrada.
- ncapas: número de capas.
- dim: dimensiones (número de neuronas) de las capas.
- nx: número total de pesos durante el entrenamiento.
- objfun: objeto con la definición de las funciones de transferencia.
- pmax: valor máximo de las entradas **p**.
- pmin: valor mínimo.
- tmax: valor máximo de las salidas **t**.
- tmin: valor mínimo.

- escal=true; (escalado activo, ya está inicializado).
- tipoini='rand'; (tipo inicialización, random por defecto).

Las funciones definidas en base_red son:

- conf: se configura la red, calculando y asignando valor a: R, dim, nx y los valores máximos y mínimos.
- escala: se escalan los datos en $[-1,1]$.
- "escala_inv" lo mismo en sentido contrario.
- vector_tdl: modifica los vectores de entrada a las capas, según lo especificado para los delays en tdl y tdla. Posteriormente se comentará esta función con mayor profundidad.

Redff hereda los datos de base_red y añade los suyos propios que son los pesos (W) y los sesgos (b). Su constructor, la función redff, recibe una estructura generada por la función hdat, con los datos a inicializar en el objeto y la función "obj_st" se encarga de ello.

```
function dat=hdat(objred,objfun)

load datos_rn dim p t escal tipoini

R=[];   nx=[];
ncapas=length(objfun);
red=struct('R',R,'ncapas',ncapas,'dim',dim,'nx',nx)

red.objfun=objfun;

W=cell(ncapas,1);  b=cell(ncapas,1);
red.W=W;   red.b=b;

dat.redff=red;
dat.p=p;   dat.t=t;

if ~isempty(escal), dat.redff.escal=escal; end
if ~isempty(tipoini), dat.redff.tipoini=tipoini; end

end
```

```
classdef redff < base_red

    properties
        W
        b
    end

    methods
        function obj=redff(varargin)
            if nargin>0
                if (isa(varargin{1},'struct'))
                    datos=varargin{1};
                    obj=obj_st(obj,datos.redff);
                    p=datos.p;    t=datos.t;
                    obj=conf(obj,p,t);
                end
            end
        end
    end
end
```

Además, redff tiene la función "act", "inic" y la función "sim"; "act" actualiza las matrices de pesos y sesgos de la RN con el vector **c** de incógnitas propuesto para la siguiente iteración de la resolución no lineal. "inic" inicializa el vector **c** siguiendo el método de Nguyen-Widrow, inicialización aleatoria en $[-0.5, 0.5]$, todos los pesos inicializados a uno o todos a cero. "sim" realiza el cálculo a través de las capas de la RN para obtener, a partir de la entrada **p**, la salida **a** (17). Además, según el número de argumentos de entrada diferenciamos el entrenamiento de la simulación, que afecta a la necesidad de escalar y desescalar los datos (en el entrenamiento **p** y **t** se guardan ya escalados).

```
function a=sim(objred,p,~)

ncapas=objred.ncapas;
[~,Q]=size(p);

W=objred.W;  b=objred.b;  objfun=objred.objfun;

if nargin==2
    p=escala(objred,p,'p');
end
n=W{1}*p+b{1};
a=calf(objfun{1},n);
for j1=2:ncapas
    n=W{j1}*a+b{j1};
    a=calf(objfun{j1},n);
end

if nargin==2
    a=escala_inv(objred,a);
end
```

La clase narx hereda de redff e incluye un enlace regresivo de la última capa a la primera. Sus datos propios son:

- tdl: donde se define el delay de entrada.
- tdla: donde se define el delay en el enlace regresivo.
- Wr: la matriz de pesos que afectan a la entrada a la primera capa del enlace regresivo.

Las funciones propias son las mismas que las de "redff" pero están sobrecargadas; es decir tienen el mismo nombre, pero son distintas al estar asociadas a distintas clases u objetos. Ello permite utilizar un lenguaje común en los desarrollos que afectan a cualquiera de las opciones: red progresiva (redff) y red progresiva con enlace regresivo (narx). Además, la clase "narx" contiene la función simtr (ver anexo II) necesaria para el entrenamiento en bucle abierto, de la que se muesta el siguiente fragmento:

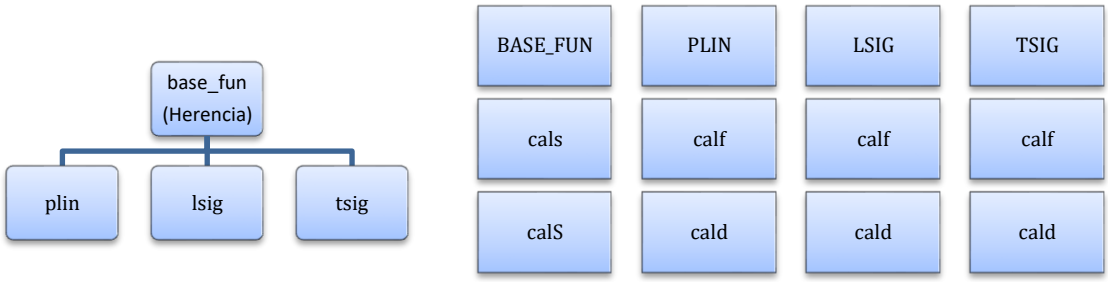
```
pp=vector_tdl(objred,p,tdl,dmaxp,dmax);
Q=size(pp,2);
unos=ones(1,Q);
np=W{1}*pp+b{1}*unos;

tt=vector_tdl(objred,t,tdla,dmaxa,dmax);   %whos p t pp tt
nt=Wr{1}*tt;    %  pause
n=np+nt;
a=calf(objfun{1},n);

ncapas=objred.ncapas;
Q=size(a,2);
unos=ones(1,Q);
for j1=2:ncapas
    n=W{j1}*a+b{j1}*unos;
    a=calf(objfun{j1},n);
end
```

Se hace uso de la función vector_tdl, que simula el comportamiento de los tdl, devolviendo una matriz formada por la secuencia al aplicar los delays. Por ejemplo, sea una secuencia $\{a_1, a_2, ..., a_n\}$ y los retrasos $tdl = \{1,2\}$, entonces, devolverá: $\begin{Bmatrix} a_1, a_2, ..., a_{n-1} \\ a_2, ..., a_{n-1}, a_n \end{Bmatrix}$.

Objetos Función

Los objetos función proporcionan los recursos asociados a las funciones de transferencia que son claves en el funcionamiento de la red neuronal. Se agregan entre los datos de los objetos red.

Se han implementado 3 tipos de funciones, lineal, sigmoidea y tangente hiperbólica. Los datos y procedimientos comunes residen en la clase "base_fun" de la que heredan las 3 clases mencionadas, completando sus datos y métodos específicos.



18. Estructura tipofun

Los métodos cals y calS son comunes a todos los objetos función y calculan la sensibilidad gradiente (45) y la sensibilidad jacobiana (50) respectivamente. Los métodos calf y cald son específicos de cada objeto función y proporcionan el valor de la función y de su derivada respectivamente. Obviamente, cada uno de los objetos función podrá acceder a 4 funciones, dos de ellas sobrecargadas con los mismos nombres; calf y cald.

Este esquema organizativo orientado a objeto permite añadir nuevos objetos función solo con su definición de clase y los métodos calf y cald sin afectar al trabajo anterior, como se observa a continuación:

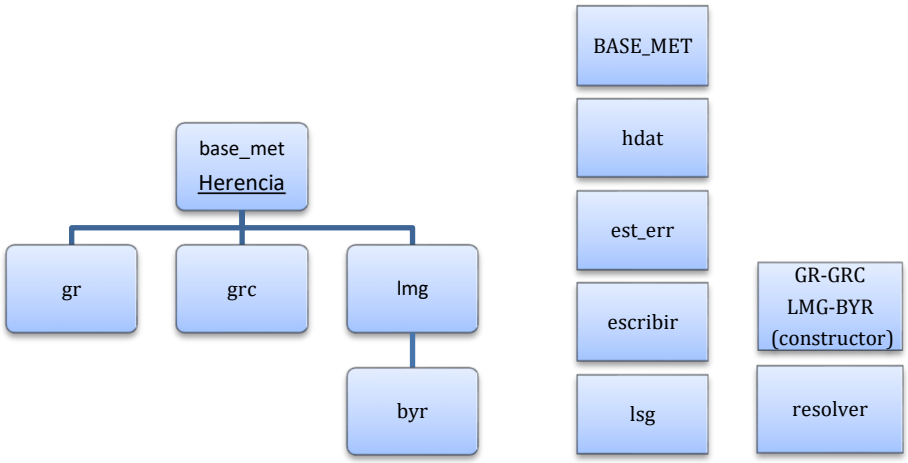
```
classdef lsig < base_fun
    methods
        function val=calf(objfun,n)
            val=1./(1+exp(-n));
        end
                    ----------------------------------
        function vald = cald(objfun,a)
            vald=(ones-a).*a;
        end
    end
end
```

Para las funciones lsig y tsig, observamos como se puede obtener la derivada a partir del valor de la función (la demostración se encuentra en el anexo I).

No tienen propiedades y no precisan método constructor y por tanto tampoco la función "hdat". Para más detalles, ver el anexo II.

Objetos Método

Los objetos método proporcionan los recursos necesarios para resolver las incógnitas (pesos y sesgos) del aprendizaje. Se han implementado para ello 4 algoritmos no lineales: gradiente o máxima pendiente (gr), gradiente conjugado (grc), Levenberg-Marquardt (lmg) y Regularización Bayesiana (byr), siguiendo lo indicado en los desarrollos matemáticos de las páginas anteriores. Los datos y procedimientos comunes residen en la clase "base_met" de la que heredan las 4 clases adicionales mencionadas, completando sus datos y métodos específicos.



19. Estructura tipomet

Los datos o propiedades que se declaran o definen son:

- objJ: albergará el objeto jacobiano para el cálculo de las derivadas que precisan los algoritmos de resolución.
- xini: valores iniciales de los pesos y sesgos incógnita.
- tol_gr=1e-4: tolerancia para la convergencia a cero del gradiente.
- tol_Ec=1e-4: tolerancia para la convergencia a cero del error cuadrático.
- maxiter=600: número máximo de iteraciones permitidas.
- tol_lsg=0.001: tolerancia para la convergencia a cero del intervalo del algoritmo lsg (linesearch golden ratio (sección áurea)).
- h_lsg=0.01: tamaño de los intervalos para la búsqueda inicial del mínimo.
- esc=[]: decide si se informa en pantalla del transcurso de las iteraciones.
- mumax=1e10: límite máximo permitido al parámetro μ del algoritmo lmg.
- emax=6: número máximo permitido de validaciones con error creciente.

Todos ellos precisan constructor y por tanto función hdat que en este caso es la misma para los 4 objetos método, por lo que reside como recurso común en la clase (objeto) base_met.

Todos ellos comparten las funciones:

- hdat: genera la estructura de datos para el constructor.
- est_err: calcula la norma del error de acuerdo con el criterio elegido (residuo por defecto).
- escribir: informa en pantalla del transcurso de las iteraciones del proceso no lineal.
- lsg: algoritmo linesearch que calcula el mínimo unidimensional mediante el algoritmo de la sección aurea (golden ratio).

Todos ellos tienen una única función no compartida sobrecargada con el mismo nombre "resolver" que utiliza el dato objeto (property) "objJ" para calcular las derivadas que necesitan para obtener los pesos (vector **c**) que minimizan el error cuadrático. A continuación, se muestra la más sencilla que corresponde al método del gradiente o máxima pendiente:

```
function [x,Ec]=resolver(objmet,objprob,objred)

objJ=objmet.objJ; tol=objmet.tol_gr;     tol_Ec=objmet.tol_Ec;
e_max=objmet.emax;  cont_e=0;  maxiter=objmet.maxiter;   conv='R';

if isempty(tol),  tol=.001;  end
if isempty(maxiter), maxiter=20;  end

x0=objmet.xini;
x=zeros(length(x0),maxiter+1);  Ec=zeros(maxiter+1,1);
k=0; escribir(objmet,1,k); x(:,1)=x0;
[g0,Eck,Ev]=calcG(objJ,objprob,objred,x0);  Ec(1)=Eck;
est_err=err(objmet,g0,x,k,conv);   escribir(objmet,2,est_err,tol)
gk=g0;  Evm=Ev;
while est_err>tol && k<maxiter  && Eck>tol_Ec && cont_e<e_max
    k=k+1;   escribir(objmet,1,k)
    x(:,k+1)=lsg(objmet,objprob,objred,x(:,k),-gk);
    [gk,Eck,Ev]=calcG(objJ,objprob,objred,x(:,k+1));
    Ec(k+1)=Eck;
    if Ev>Evm
        cont_e=cont_e+1;
    else
        Evm=Ev;  cont_e=0;
    end
    est_err=err(objmet,gk,x,k,conv);   escribir(objmet,2,est_err,tol)
end
niter=k;  x=x(:,1:niter+1);  Ec=Ec(1:niter+1);
end
```

Objetos Jacobiano

Los objetos jacobiano proporcionan los recursos necesarios para el cálculo de las derivadas que forman parte del gradiente del error cuadrático respecto a los pesos, o de la matriz jacobiana del error respecto a las mismas variables. Ambas magnitudes se requieren desde los algoritmos gradiente (gr y grc) y los algoritmos Levenberg-Marquardt (lmg, byr) respectivamente. Por ello, el objeto método contiene entre sus datos otro objeto (agregación) que es del tipo objeto jacobiano.

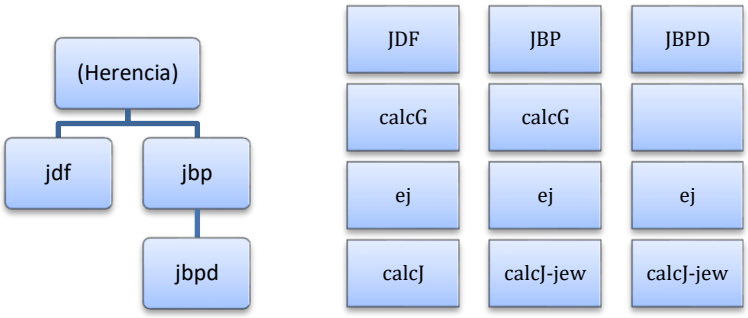
Se han implementado 3 objetos (clases) jacobiano para el cálculo:

a) por diferencias finitas (jdf).

b) mediante el algoritmo de propagación regresiva (BP) (jbp).

c) mediante el algoritmo de propagación regresiva (jbpd) para redes dinámicas.

No existen en este caso datos ni métodos (funciones) compartidas por lo que no existe objeto (clase) base de partida. jdf y jbp se definen directamente y jbpd hereda de jbp. Los 3 objetos implementados tampoco tienen datos por lo que no precisan método constructor ni tampoco función hdat de apoyo.



20. Estructura tipoJ

Las tres clases comparten las mismas 3 funciones sobrecargadas con el mismo nombre, pero con distinto contenido. calcG calcula el vector gradiente, calcJ calcula la matriz jacobiana y ej proporciona el error y la matriz jacobiana. A modo de ejemplo, se lista a continuación la función (method) calcJ del objeto jbp.

```
function  jac=calcJ(objJ,red,p)

ncapas=red.ncapas;
[~,Q]=size(p);
W=red.W;   b=red.b;

objfun=red.objfun;    dM=red.dim(end);

a=cell(ncapas+1,1);  a{1}=p;
for m=2:ncapas+1
    n=W{m-1}*a{m-1}+b{m-1};
    a{m}=calf(objfun{m-1},n);
end

S=calS(objfun{end},a{end});
dw=jew(objJ,a{end-1},S,dM);    jac=[S' dw'];
for m=ncapas-1:-1:1
    S=calS(objfun{m},a{m+1},S,W{m+1});
    dw=jew(objJ,a{m},S,dM);
    jac=[S' dw' jac];
end
```

En calcJ[1] se calcula el jacobiano siguiendo la explicación de la sección 'Backpropagation' de 'Redes Neuronales'. Primero se calcula hacia delante el valor de las salidas de las capas[2] y después, hacia atrás, se propaga el cálculo de las derivadas.

El algoritmo "backpropagation" es uno de los métodos más rápidos en el cálculo del jacobiano, pero también es de los que más memoria ocupa, para reducirlo, la sensibilidad S se va sobrescribiendo cuando se utiliza. Para lograrlo se ha combinado en un mismo bucle el cálculo de las sensibilidades y del jacobiano, lo que también acelera los cálculos.

jew es una función de apoyo donde se multiplican adecuadamente los elementos de s por los de a, aplicando (46), (47) y (48):

```
function jac = jew(objJ,a,S,dM)

dm=size(S,1);       %d(m)
dm_1=size(a,1);     %d(m-1)

jac=kron(a,ones(dm,dM)).*kron(ones(dm_1,1),S);
```

[1] CalcG tiene una estructura similar, por lo que omitiremos su explicación.
[2] Teniendo en cuenta que las matrices de celdas comienzan su enumeración en 1 y con el objetivo de poder introducir en un mismo bucle el cálculo de las sensibilidades y derivadas, se ha asignado el valor a^1 a la entrada, que en la teoría se correspondía con a^0.

jbpd es similar, con la excepción de que se combinan primero W^1 y Wr (matriz de pesos que une la capa de salida con la de entrada) y se trata como si fuese una única matriz con una única entrada, en lugar de trabajar con **p**(t) y **a**(t) como valores separados. En el caso de jbpd no encontramos calcG, porque la falta de precisión en los métodos del gradiente suponía un error que se iba propagando en la simulación.

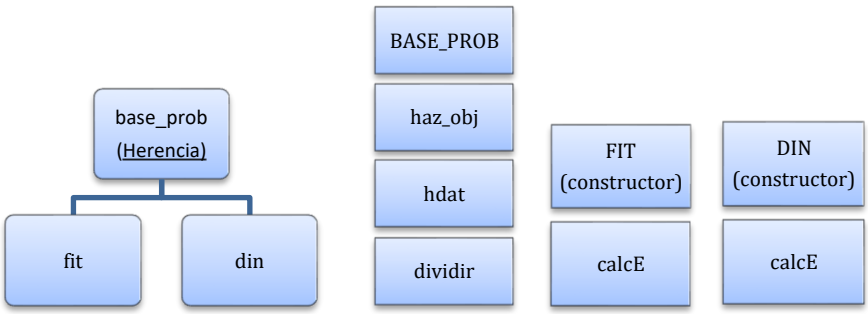
Objetos Problema

El objeto problema es quien construye al inicio los objetos adecuados para la construcción, entrenamiento y simulación de las redes neuronales. También proporciona aspectos específicos asociados al problema: estático, dinámico u otro tipo, cuando no pueden ser atendidos debidamente por los objetos restantes. Como en casos anteriores, los datos y procedimientos comunes residen en la clase "base_prob" de la que heredan 2 clases adicionales: fit para el problema estático y din para el dinámico, completando sus datos y métodos específicos.

Los datos o propiedades que se declaran o definen son:

- p: conjunto de entrada de entrenamiento.
- t: conjunto objetivo (targets) de entrenamiento.
- p_val: conjunto de entrada de validación (vacío si no se quiere utilizar ES).
- t_val: conjunto objetivo (targets) de validación (vacío si no se quiere utilizar ES).

Ambos objetos fit y din, reciben argumentos de entrada para asignar valor a los datos: **p**, **t**, **p_val**, **t_val** por lo que requieren método constructor y por tanto función hdat para generar los datos en la forma apropiada. Ambas funciones hdat son coincidentes en este caso, por lo que se tratará de una función compartida que se encuentra en base_prob.



21. Estrucutra tipoprob

Haz_obj es la función que se encarga de construir los objetos especificados en la entrada de datos. La función coincide para ambos objetos fit y din por lo que se encuentra en base_prob para evitar su duplicación. Por su relevancia se lista a continuación:

```
function [redn,objprob,objmet,objJ,objred,objfun]=haz_obj(objprob)

    load datos_rn red fun met metJ prob p t prop

    objfun=cell(1);
    for j1=1:length(fun)
        objfun{j1}=feval(fun{j1});
    end

    objred=feval(red);   datred=hdat(objred,objfun);
    objred=feval(red,datred);
    p=escala(objred,p,'p');
    t=escala(objred,t,'t');
    [p,t,p_val,t_val]=dividir(prop,p,t);

    datprob=hdat(objprob,p,t,p_val,t_val);
    objprob=feval(prob,datprob);

    objJ=feval(metJ);
    c0=inic(objred,p,t);
    objmet=feval(met); datmet=hdat(objmet,objJ,c0);
    objmet=feval(met,datmet);
    redn=redneu(objmet,objprob,objred);
end
```

La función dividir, divide la muestra de forma aleatoria en un conjunto de entrenamiento y otro de validación en la proporción especificada. calcE, por su parte, devuelve el error del conjunto de entrenamiento y de validación. Si no se realiza división de los datos, devuelve 0 como error de validación. En el caso de las redes dinámicas (al no tener sentido la división, pues estaríamos cortando una secuencia) no se devuelve este valor.

Ambos, fit y din, tienen una única función no compartida sobrecargada con el mismo nombre "calcE" que calcula el vector error que se obtiene a la salida de la RN.

Otras funciones independientes de los objetos

Recoge las funciones auxiliares necesarias: guardadatos (guarda en un archivo los datos de la estructura de la red, creando las variables, aunque no se hayan definido, evitando problemas de ejecución) y obj_st (convierte una estructura de datos en los datos (properties) de los objetos).

Descripción de los resultados

En esta sección se estudiarán en ejercicios prácticos los distintos algoritmos trabajados, analizando las ventajas y desventajas que ofrecen y comparándolos con algunos métodos clásicos.

Backpropagation

En primer lugar, se estudiará el efecto que supuso el backpropagation, comparando el desempeño del BP frente al cálculo de derivadas por diferencias finitas. Para ello, los tiempos necesitados y los resultados obtenidos serán los criterios de comparación. Las bases de datos empleadas se pueden obtener en Matlab.

Simplefit dataset

Conjunto de datos de ejemplo. Conjunto de 94 datos de entrada y salida (escalares).

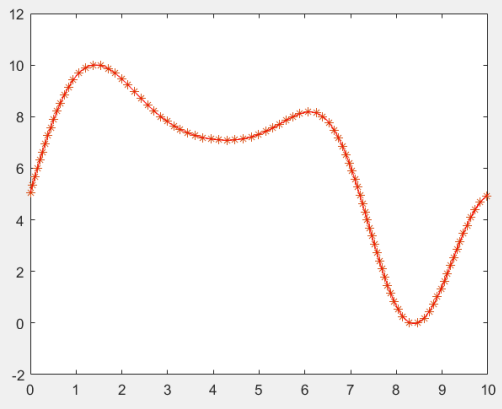
Red neuronal: Feedforward.

Número de capas: 2.

Función de transferencia: Tangente hiperbólica – Función lineal.

Número de neuronas: 10 - 1.

Inicialización: Nguyen-Widrow.



22. Función Simplefit

Profile Summary

Generated 25-Apr-2019 16:30:04 using performance time.

Function Name	Calls	Total Time	Self Time*	Total Time Plot (dark band = self time)
gfit4	1	0.501 s	0.011 s	
redneu.entrena	1	0.334 s	0.001 s	
Img.resolver	1	0.332 s	0.056 s	
jbp.ej	138	0.158 s	0.007 s	

23. Tiempo entrenamiento simplefit (jbp)

Profile Summary

Generated 24-Apr-2019 18:57:13 using performance time.

Function Name	Calls	Total Time	Self Time*	Total Time Plot (dark band = self time)
gfit4	1	12.632 s	0.057 s	
redneu.entrena	1	12.199 s	0.001 s	
Img.resolver	1	12.198 s	0.956 s	
jdf.ej	1997	9.804 s	0.560 s	

24. Tiempo entrenamiento simplefit (jdf)

Comparando los resultados obtenidos, se observa como las dos funciones se superponen perfectamente y pasan por los puntos de los datos de entrenamiento.

Sin embargo, en lo referente al tiempo necesario, al calcularlo mediante diferencias finitas se ha necesitado 12s más que con BP (trabajando solamente con una red de dos capas, 10 y 1 neuronas respectivamente).

Abalone Rings

Conjunto de datos que permite estimar el número de anillos en un abalón (un tipo de marisco) a partir de sus características físicas (peso, tamaño…). 4177 datos, entrada vectorial (dimensión 8) y salida escalar.

Red neuronal: Feedforward.

Número de capas: 3.

Función de transferencia: Tangente hiperbólica – Tangente hiperbólica - Función lineal.

Número de neuronas: 10 – 2 – 1.

Inicialización: Nguyen-Widrow.

Profile Summary

Generated 24-Apr-2019 19:23:36 using performance time.

Function Name	Calls	Total Time	Self Time*	Total Time Plot (dark band = self time)
gfit4	1	28.475 s	0.047 s	
redneu.entrena	1	28.323 s	0.002 s	
lmg.resolver	1	28.322 s	4.605 s	
jbp.ej	1203	22.152 s	0.080 s	

25. Tiempo entrenamiento abalone (jbp)

Profile Summary

Generated 24-Apr-2019 19:22:04 using performance time.

Function Name	Calls	Total Time	Self Time*	Total Time Plot (dark band = self time)
gfit4	1	137.917 s	0.025 s	
redneu.entrena	1	137.809 s	0.001 s	
lmg.resolver	1	137.807 s	4.130 s	
jdf.ej	1201	132.251 s	5.020 s	

26. Tiempo entrenamiento abalone (jdf)

```
**************** iteracion600*********************************
Estimación error =0.037251       Tolerancia =1e-07

Eck =

   78.6606


mu =

    0.1000
```

27. Resultados entrenamiento abalone (jbp)

```
**************** iteracion600*********************************
Estimación error =0.71472       Tolerancia =1e-07

Eck =

   79.4770


mu =

   1.0000e-03
```

28. Resultado entrenamiento abalone (jdf)

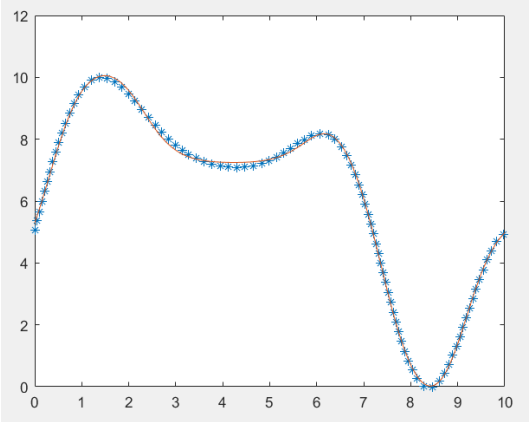
Sin más que añadir una capa adicional con, tan solo, 2 neuronas vemos la notable diferencia al aplicar BP. En cuanto al error final, se debe a los distintos valores de la inicialización.

Levenberg-Marquardt

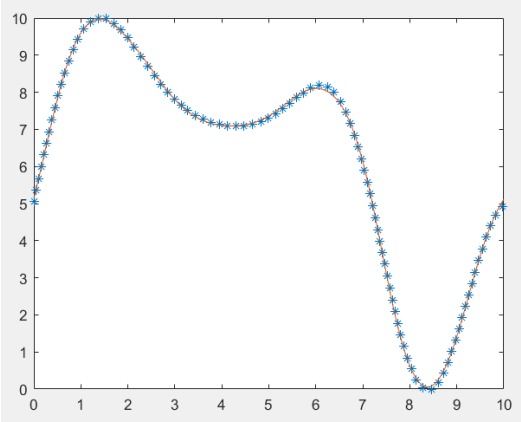
Como ya se ha mencionado en la descripción de la Propuesta de solución, el algoritmo de Levenberg-Marquardt es de los más rápidos, así como muy preciso, para comprobarlo, se comparan los resultados obtenidos con los métodos LM, Gradiente y Gradiente conjugado.

Simplefit dataset II

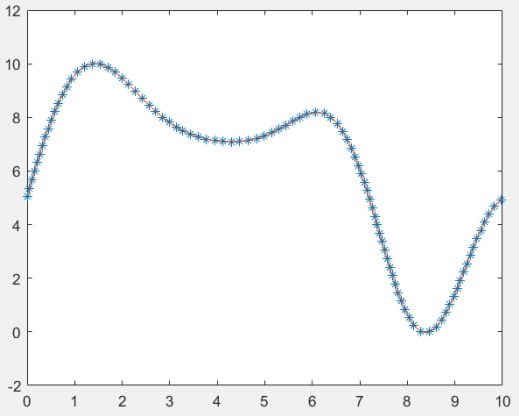
Volviendo otra vez sobre el ejemplo inicial (los resultados son más gráficos que en otros ejemplos).



29. Resultado método del gradiente



30. Resultado método del gradiente conjugado



31. Resultado método Levenberg-Marquardt

Además de los errores que se aprecian a simple vista (siendo máximos en el método del gradiente, seguido por el del gradiente conjugado y finalmente LM, donde no se aprecian), también se ven las ventajas en el tiempo empleado en el cálculo del gradiente (del orden de 0.002s en el primer caso y 0.001s en el segundo) frente al empleado en el cálculo del jacobiano (0.0005s con LM).

Inicialización

En esta sección se analiza el efecto del tipo de inicialización en la velocidad de convergencia del algoritmo. Compararemos el algoritmo de Nguyen-Widrow (NG) y una inicialización aleatoria de los pesos en $[-0.5, 0.5]$. Se espera obtener una leve mejora con NG, sin embargo, se sabe que los resultados son similares.

Para el experimento, se realizarán 200 simulaciones con cada uno de los métodos y se realizará un contraste de hipótesis $\begin{cases} H_0 : \mu_{rand} \geq \mu_{NW} \\ H_1 : \mu_{rand} < \mu_{NW} \end{cases}$, mediante el estimador: $\frac{\overline{x}_{NW} - \overline{x}_{rand}}{\sqrt{\frac{S_{rand}^2}{200} + \frac{S_{NW}^2}{200}}} \to N(0,1)$.

El intervalo de confianza vendrá dado por: $(\overline{x}_{NW} - \overline{x}_{rand}) \mp \sqrt{\frac{S_{rand}^2}{200} + \frac{S_{NW}^2}{200}} \cdot z_{\alpha/2}$.

Bodyfat

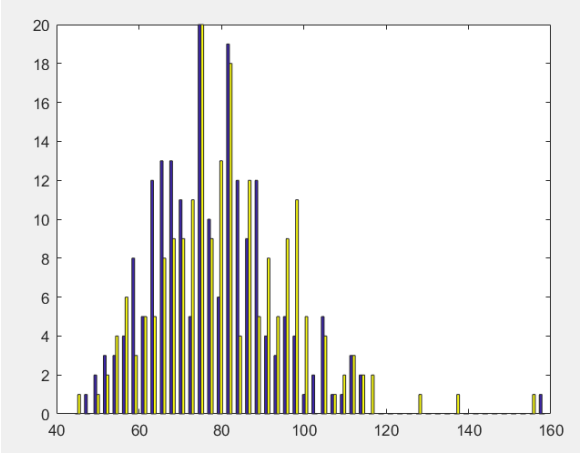
Conjunto de datos que permite estimar el índice de masa corporal a partir de sus características físicas (peso, edad, altura…). 252 datos, entrada vectorial (dimensión 13) y salida escalar.

Red neuronal: Feedforward.

Número de capas: 2.

Función de transferencia: Tangente hiperbólica - Función lineal.

Número de neuronas: 20 – 1.



32. Histograma de frecuencias del número de iteraciones Bodyfat

Azul (inicialización aleatoria) – Amarillo (NW)

Los datos obtenidos son: $\overline{x}_{rand} = 77.83$, $\overline{x}_{NW} = 81.305$, $S^2_{rand} = 232.2624$, $S^2_{NW} = 272.6754$

En este caso, $\overline{x}_{rand} < \overline{x}_{NW}$, por lo que invertiremos las hipótesis:

$$\begin{cases} H_0 : \mu_{rand} \leq \mu_{NW} \\ H_1 : \mu_{rand} > \mu_{NW} \end{cases} \text{ y el p-valor será: } P\left\{ \frac{\overline{x}_{NW} - \overline{x}_{rand}}{\sqrt{\frac{S^2_{rand}}{200} + \frac{S^2_{NW}}{200}}} < z \right\}, z \to N(0,1) .$$

A la vista de los resultados, el p-valor $= 0.8857$, aceptando la hipótesis nula $H_0 : \mu_{NW} \geq \mu_{rand}$. El intervalo de confianza para $\mu_{rand} - \mu_{NW}$, con $\alpha = 0.05$, será $[0.3607, 6.5893]$, intervalo que no contiene a 0, confirmando el resultado anterior.

Building Energy

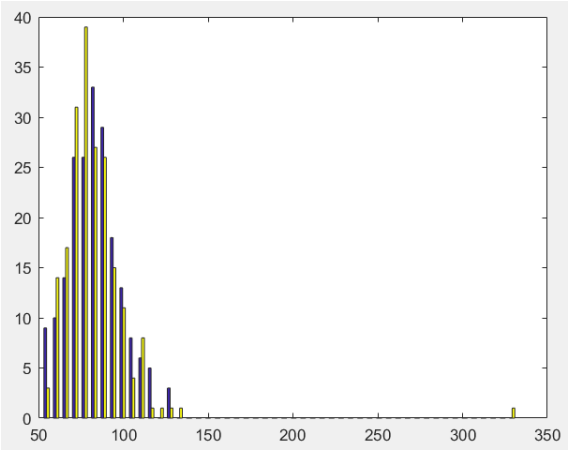
Conjunto de datos que permite estimar la energía consumida por un edificio en función de las condiciones climatológicas. 4208 datos, entrada vectorial (dimensión 14) y salida vectorial (dimensión 3).

Red neuronal: Feedforward.

Número de capas: 3.

Función de transferencia: Función sigmoide – Función sigmoide - Función lineal.

Número de neuronas: 10 – 2 – 3.



33. Histograma de frecuencias del número de iteraciones (building)

A la vista de la gráfica, se observa que hay un valor en el número de iteraciones NW (amarillo) que dista mucho del resto, por lo que se omitirá en el estudio. Los resultados obtenidos son: $\overline{x}_{rand} = 83.2750$, $\overline{x}_{NW} = 81.5276$, $S^2_{rand} = 233.2255$, $S^2_{NW} = 201.9374$

El p-valor=0.119 no es concluyente. El intervalo de confianza con $\alpha = 0.05$, será $[-1.147, 4.64]$, que incluye el 0. Por tanto, parece lógico pensar que el número de iteraciones mediante inicialización aleatoria no es mayor que con NW.

A la vista de los resultados obtenidos en este apartado, no parece que el algoritmo de Nguyen-Widrow mejore (incluso empeora) los resultados obtenidos con una inicialización aleatoria. A pesar, de que esperábamos una leve mejora con NW, tampoco son de extrañar los resultados obtenidos. El algoritmo está diseñado para la función sigmoidea (en el primer ejemplo se usa la tangente hiperbólica) y tampoco han sido diferencias muy notables.

Regularización Bayesiana

Las dos ventajas que otorga este método frente al de LM, son: una mayor generalización y conocer el número de pesos necesarios (facilitando escoger la estructura de red más eficiente).

Función seno

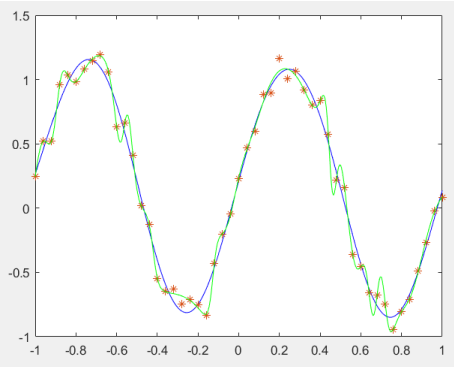
En el siguiente ejemplo se ha aproximado la función $y = \sin(2 \cdot \pi \cdot x)$ en el intervalo $[-1,1]$. Los datos utilizados para el entrenamiento tenían un error distribuido en $[0, 0.3]$.

Red neuronal: Feedforward.

Número de capas: 2.

Función de transferencia: Función sigmoide - Función lineal.

Número de neuronas: 20 – 1.



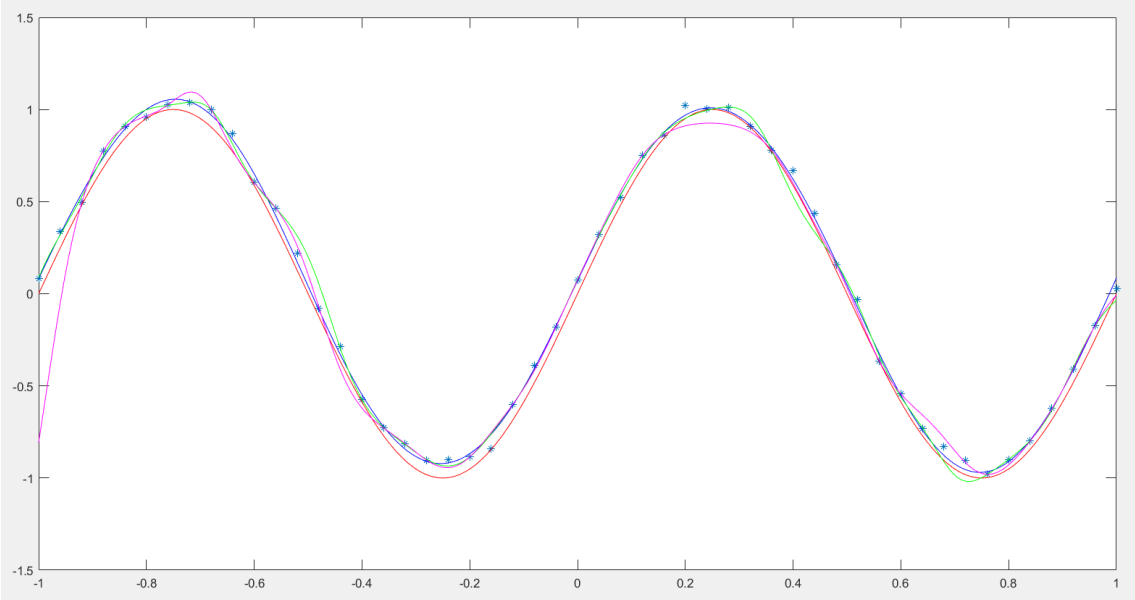
34. Aproximación de la función seno con error

Se ha simulado utilizando el método de LM (verde) y RB (azul), como se puede comprobar, la línea verde busca ajustarse perfectamente a los datos de entrenamiento, alejándose de la sinusoide, sin embargo, la azul, omite los errores. (realmente está aproximando la función

$y = \sin(2 \cdot \pi \cdot x) + 0.15$, ya que los errores tienen media 0.15). Además, el método de LM, ha requerido 521 iteraciones frente a las 19 de RB.

Otra de las formas de solucionar el problema de la generalización es utilizar el early stopping o detención temprana, como se ha mencionado en el apartado 'Estado del Arte'.

Sobre esta misma función vamos a comparar los resultados obtenidos mediante RB y ES (método de LM) dividiendo los datos en una proporción de 50/50 y 70/30.



35. Generalización seno

En rojo observamos la función seno, en azul la respuesta entrenada mediante RB, verde LM con una proporción 70/30 y en rosa con 50/50.

El principal inconveniente que observamos en el ES es que (especialmente en este ejemplo con pocos datos) al aumentar el tamaño de la muestra de validación, se forman zonas en las que la red no tenía información suficiente (al principio o en el segundo máximo, en el caso de la rosa), por lo que la respuesta es pobre. Por otro lado, se pueden seguir observando varias zonas donde existe overfitting.

La única ventaja que encontramos en el método de ES, es su sencillez frente a la RB, que tiene un desarrollo matemático detrás mayor, y la facilidad de adaptar algoritmos ya escritos para que lo apliquen (como ha sido el caso del LM programado, que en su versión inicial no tenía en cuenta esta opción).

Simplefit dataset III

La otra ventaja principal de este método es que nos permite conocer el número de pesos que realmente se están utilizando en el entrenamiento (γ). Para comprobarlo, el problema de aproximación de funciones se hará sobredimensionando la red y corrigiendo hasta obtener la red más eficiente.

Volviendo al conjunto de datos de ejemplo:

Red neuronal: Feedforward.

Número de capas: 3.

Función de transferencia: Tangente hiperbólica – Tangente hiperbólica - Función lineal.

Número de neuronas: 20 – 10– 1.

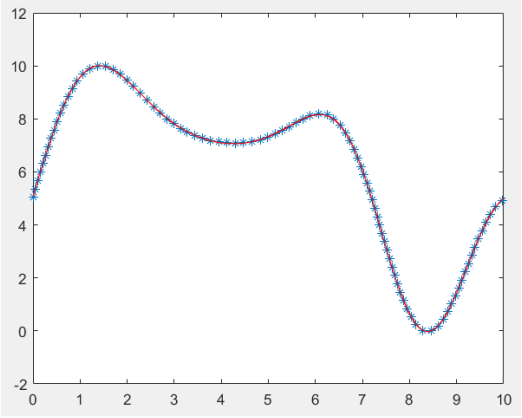
La RB ha devuelto γ=18.59, por tanto, una red con (6 y 1 neuronas) sería suficiente (19 pesos).

Profile Summary

Generated 25-Apr-2019 23:53:06 using performance time.

Function Name	Calls	Total Time	Self Time*	Total Time Plot (dark band = self time)
gfit4	1	0.394 s	0.008 s	
redneu.entrena	1	0.281 s	0.001 s	
lmq.resolver	1	0.279 s	0.047 s	
jbp.ej	110	0.131 s	0.006 s	

36. Tiempo entrenamiento simplefit (6n)



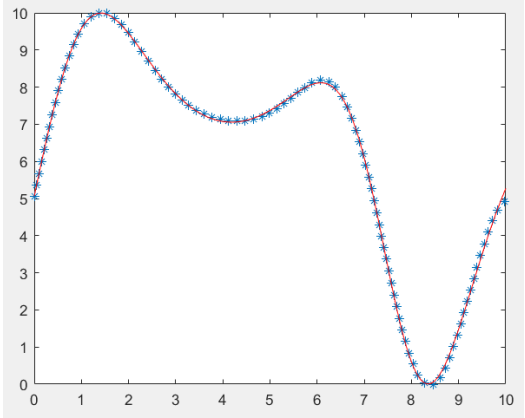
37. Resultados simplefit (6n)

Profile Summary

Generated 25-Apr-2019 23:51:47 using performance time.

Function Name	Calls	Total Time	Self Time*	Total Time Plot (dark band = self time)
gfit4	1	2.720 s	0.008 s	
redneu.entrena	1	2.561 s	0.001 s	
lmq.resolver	1	2.560 s	0.438 s	
jbp.ej	1202	1.153 s	0.053 s	

38. Tiempo entrenamiento simplefit (4n)



39. Resultados simplefit (4n)

Como se esperaba, la precisión de la red (con 6 neuronas en la capa intermedia) es óptima y hemos reducido el número de operaciones frente a, por ejemplo, el ejemplo inicial (10 neuronas), aunque en este caso no haya gran diferencia. También se observa como al reducir a 4 las neuronas, la red no dispone de pesos suficientes y los errores son considerables. Además, los tiempos también aumentan ya que la red no es capaz de conseguir la tolerancia necesaria.

Entre las desventajas de la RB encontramos que, en este último ejemplo (en el que no hay errores aleatorios), las iteraciones necesarias (y el tiempo de ejecución) eran mayores que con LM. También, dependiendo de la inicialización (no del método, sino de los valores) la convergencia era errónea[1].

Redes dinámicas

En esta sección se estudiará el resultado ofrecido por las NARX y se comparará con la solución ofrecida por la ecuación diferencial.

Magnetic Levitation

Conjunto de datos que permite estimar la altura de un levitador magnético estimulado por una corriente eléctrica. A partir de dos secuencias temporales i(t) e y(t) se entrenará la red en bucle abierto y se simulará en bucle cerrado.

Red neuronal: NARX.

Número de capas: 2.

Función de transferencia: Tangente hiperbólica - Función lineal.

Número de neuronas: 10– 1.

Delays entrada: 1 – 2.

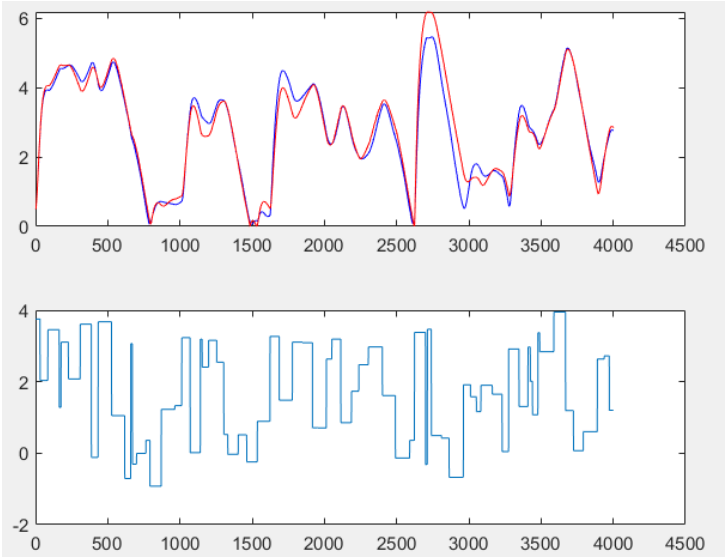
Delays salida: 1 – 2.

Para este problema se dispone de lasiguiente ecuación diferencial que describe el movimiento:

$$\frac{d^2y}{dt^2} = -g + \frac{\alpha}{M}\cdot\frac{i^2(t)}{|y|} - \frac{\beta}{M}\cdot\frac{dy}{dt}$$, con β=12, α=15, g=9.81 y M=3.

Esto ha permitido obtener los datos para el entrenamiento de manera computacional en vez de experimental. Para ello se ha utilizado la función ode15s de Matlab que resuelve numéricamente la ecuación diferencial.

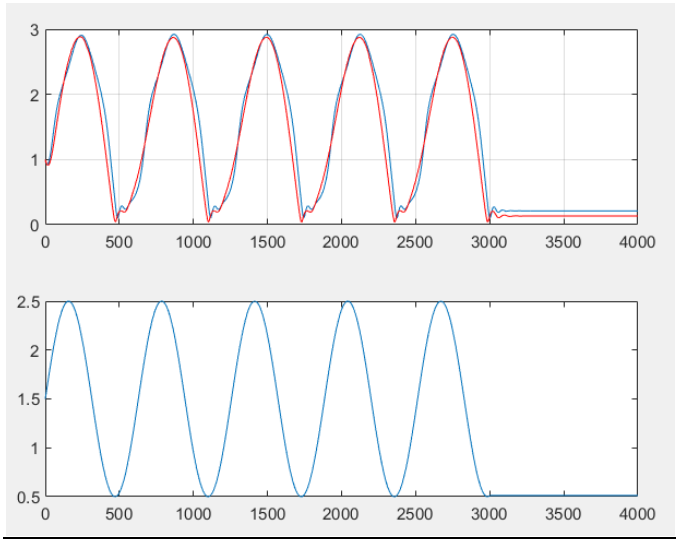
[1] En el anexo se pueden encontrar ejemplos.



40. Levitador magnético

Ante una intensidad como la de la figura (parte de abajo), que se ha generado a base de escalones para obtener valores tanto en estacionario como en transitorio, la ode15s proporciona la altura del levitador (rojo). Con estos datos de muestra se ha entrenado la red en bucle abierto y con los pesos obtenidos se ha simulado la respuesta de la red neuronal ante la misma intensidad utilizada para el entrenamiento. En la respuesta obtenida (azul) se observa que la respuesta es muy buena y los errores pequeños.

A continuación, utilizamos la red entrenada para predecir el comportamiento ante otra entrada (intensidad) y la respuesta obtenida por la RN la comparamos con el valor de referencia (obtenido resolviendo la ecuación diferencial mediante ode15s).



41. Levitador magnético - sinusoide + estacionario

Se puede observar como la red (azul) representa correctamente el comportamiento del levitador. Este procedimiento de identificación del sistema es una alternativa, por ejemplo en aplicaciones de control, cuando se desconoce la ecuación que rige el sistema pero se dispone de suficiente número de datos experimentales de su comportamiento.

Conclusiones

Una vez realizado el trabajo, considero que se han cumplido satisfactoriamente los objetivos propuestos al principio. He logrado una base teórica y práctica de los algoritmos básicos que se aplican en las RN, entendiendo, ahora, que sucede al aplicarlos en la toolbox de Matlab. Además, se ha logrado una RN operativa, que permite ver el funcionamiento de las redes de una forma más clara, que también me ha permitido ampliar mis conocimientos de Matlab (especialmente la programación orientada a objeto).

Las conclusiones más destacadas que he obtenido en este trabajo son:

- La aplicación de las redes neuronales se está extendiendo a pasos agigantados por los distintos sectores. Las ventajas observadas durante el desarrollo del trabajo refuerzan la idea de su prometedor futuro.
- El trabajo se ha realizado sin ningún problema con un ordenador portátil, sin problemas de velocidad o viéndose atascado en ciertos momentos. A pesar de que las RN ejecutadas no eran grandes, se desprende que su uso no está restringido a superordenadores, reafirmando, aún más, la idea anterior.

Finalmente, este trabajo ha supuesto para mí una buena forma de introducirme en este sector, aumentando mi interés en él, y espero que me ayude de cara a futuros trabajos personales que haga relacionados.

Bibliografía

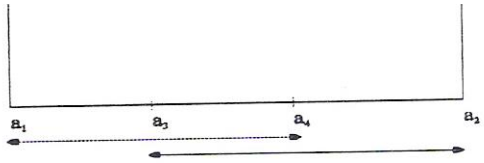
[1] M. Hagan, H. Demuth, M. Beale y O. De Jesús, *Neural Network Design*. 2ª edición. Oklahoma: Martin Hagan, 2014.
[2] M. Beale, M. Hagan, H. Demuth, *Neural Network Toolbox User's Guide*. Massachusetts: the Mathworks, Inc, 2017.
[3] M. Beale, M. Hagan, H. Demuth, *Neural Network Toolbox Getting Started Guide*. Massachusetts: the Mathworks, Inc, 2017.
[4] G. Strang, *Introduction to applied mathematics*. Massachusetts: Wellesley-Cambridge Press, 1986.
[5] Matlab 7, *Classes and object oriented programming*. Massachusetts: the Mathworks, Inc, 2008.
[6] F. Foresee (1979). *Generalization and neural networks*. Oklahoma State University, Oklahoma
[7] O. De Jesús (1985). *Training general dynamic neural networks*. Universidad Simón Bolivar, Venezuela

Anexo I

Método de enmarcado. Sección áurea.

Una vez encuadrado el primer intervalo $(x_{i-1},\ x_{i+1})$ donde se encuentra el extremo local de la función $f(x)$ se pueden definir nuevos subintervalos conteniendo al extremo local de la siguiente manera:

1. Llamaremos a_1 y a_2 a los dos extremos del primer intervalo de encuadre.
2. Definiremos 2 nuevos puntos internos (simétricos con respecto al centro del intervalo) que llamaremos a_3 y a_4 respectivamente, quedando los 4 puntos ordenados en sentido creciente (abscisas) de la siguiente manera: {a_1, a_3, a_4, a_2}.



$$\left.\begin{aligned} a_3 &= (1-\alpha)a_1 + \alpha a_2 \\ a_4 &= \alpha a_1 + (1-\alpha)a_2 \end{aligned}\right\} \quad 0 < \alpha < 1/2 \tag{51}$$

Y calculamos los valores de la función $f(x)$ en los 2 nuevos puntos a3 y a4:

3. Analizaremos en cuál de los dos subintervalos {a_1, a_3, a_4} o {a_3, a_4, a_2} se encuentra el extremo. Para un mínimo seleccionaremos el primer subintervalo {a_1, a_3, a_4} siempre que no se verifique la condición:

$$f(a_1) > f(a_3) > f(a_4)$$

En ese caso elegiremos el segundo subintervalo {a_3, a_4, a_2} aunque el mínimo podría estar en la parte común de ambos subintervalos. De esa forma, siempre se reduce el intervalo y el proceso de "enmarcado" es convergente, a costa de evaluar dos veces la función por paso o iteración.

La eficiencia computacional del proceso se puede mejorar substancialmente, ya que, si elegimos adecuadamente la posición de los puntos intermedios, podremos pasar de dos evaluaciones de $f(x)$ por iteración, a una.

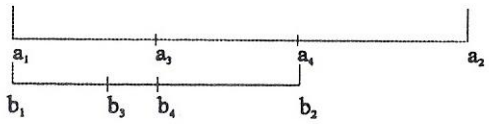
Supongamos que el mínimo está en el primer subintervalo {a_1, a_3, a_4} y que repetimos los pasos de la siguiente iteración de la siguiente forma:

1. Llamaremos b_1 y b_2 a los extremos del intervalo. Por tanto: $\left\{\begin{matrix} b_1 = a_1 \\ b_2 = a_4 \end{matrix}\right\}$.

2. Calcularemos los nuevos valores internos b_3 y b_4 según:

$$\left\{\begin{matrix} b_3 = (1-\beta)b_1 + \beta b_2 \\ b_4 = \beta b_1 + (1-\beta)b_2 \end{matrix}\right\} \quad 0 < \beta < 1/2 \tag{52}$$

y para aprovechar alguno de los puntos conocidos de la iteración anterior haremos coincidir b_4 con a_3:



$$\beta b_1 + (1-\beta)b_2 = (1-\alpha)a_1 + \alpha a_2$$

Teniendo en cuenta (51) y (52), y operando: $\boxed{\beta = \frac{1-2\alpha}{1-\alpha}}$

Por tanto, si los valores del parámetro α en las sucesivas iteraciones se calcula según

$$\alpha_{i+1} = \frac{1-2\alpha_i}{1-\alpha_i} \tag{53}$$

solo será necesario evaluar una vez la función por paso (o iteración). Para mantener el mismo valor del parámetro α en todas las iteraciones, deberá cumplirse:

$$\alpha = \frac{1-2\alpha}{1-\alpha} \;\rightarrow\; \alpha^2 - 3\alpha + 1 = 0 \;\rightarrow \begin{cases} \alpha = \dfrac{3+\sqrt{5}}{2} = 2.62 \\ \alpha = \dfrac{3-\sqrt{5}}{2} = 0.381966 \end{cases} \tag{54}$$

siendo solo válida la solución comprendida entre 0 y 0.5, valor α=0.381966 que nos permite enmarcar de manera convergente, los sucesivos subintervalos con sólo una evaluación de la función por iteración. Es fácil comprobar a partir de (51) que:

$$\frac{a_4 - a_1}{a_2 - a_1} = 1 - \alpha = 0.618...$$

lo cual quiere decir, que los sucesivos intervalos se reducen en un factor constante (de valor 0.618...) por iteración.

Se denomina sección áurea, al número: $\varphi = \frac{1}{1-\alpha} = \frac{1+\sqrt{5}}{2} = 1.618...$ que ya fue objeto de estudio en la antigua Grecia y verifica la propiedad: $\frac{1}{\varphi} = \varphi - 1;$ entonces podemos decir que el factor de reducción es: $1-\alpha = \frac{1}{\varphi} = \varphi - 1$

razón por la cual este método de enmarcado recibe el nombre de sección áurea.

Tabla I

Name	Input/Output Relation	Icon	MATLAB Function
Hard Limit	$a = 0 \quad n < 0$ $a = 1 \quad n \geq 0$		hardlim
Symmetrical Hard Limit	$a = -1 \quad n < 0$ $a = +1 \quad n \geq 0$		hardlims
Linear	$a = n$		purelin
Saturating Linear	$a = 0 \quad n < 0$ $a = n \quad 0 \leq n \leq 1$ $a = 1 \quad n > 1$		satlin
Symmetric Saturating Linear	$a = -1 \quad n < -1$ $a = n \quad -1 \leq n \leq 1$ $a = 1 \quad n > 1$		satlins
Log-Sigmoid	$a = \frac{1}{1+e^{-n}}$		logsig
Hyperbolic Tangent Sigmoid	$a = \frac{e^{n}-e^{-n}}{e^{n}+e^{-n}}$		tansig
Positive Linear	$a = 0 \quad n < 0$ $a = n \quad 0 \leq n$		poslin
Competitive	$a = 1$ neuron with max n $a = 0$ all other neurons		compet

42. Tabla funciones de transferencia

Algoritmo RTRL

Real-Time Recurrent Learning Gradient

Initialize:

$$\frac{\partial \mathbf{a}^u(t)}{\partial \mathbf{x}^T} = \mathbf{0},\ t \le 0\text{, for all } u \in U,$$

For $t = 1$ to Q,

$U' = \varnothing$, $E_S(u) = \varnothing$ and $E_S^X(u) = \varnothing$ for all $u \in U$.

For m decremented through the BP order

For all $u \in U'$, if $E_S(u) \cap L_m^b \neq \varnothing$

$$\mathbf{S}^{u,m}(t) = \left[\sum_{l \in E_S(u) \cap L_m^b} \mathbf{S}^{u,l}(t)\mathbf{LW}^{l,m}(0)\right]\dot{\mathbf{F}}^m(\mathbf{n}^m(t))$$

add m to the set $E_S(u)$

if $m \in X$, add m to the set $E_S^X(u)$

EndFor u

If $m \in U$

$$\mathbf{S}^{m,m}(t) = \dot{\mathbf{F}}^m(\mathbf{n}^m(t))$$

add m to the sets U' and $E_S(m)$

if $m \in X$, add m to the set $E_S^X(m)$

EndIf m

EndFor m

For $u \in U$ incremented through the simulation order

For all weights and biases ($\mathbf{x}$ is a vector containing all weights and biases)

$$\frac{\partial^e \mathbf{a}^u(t)}{\partial vec(\mathbf{IW}^{m,l}(d))^T} = [\mathbf{p}^l(t-d)]^T \otimes \mathbf{S}^{u,m}(t)$$

$$\frac{\partial^e \mathbf{a}^u(t)}{\partial vec(\mathbf{LW}^{m,l}(d))^T} = [\mathbf{a}^l(t-d)]^T \otimes \mathbf{S}^{u,m}(t)$$

$$\frac{\partial^e \mathbf{a}^u(t)}{\partial (\mathbf{b}^m)^T} = \mathbf{S}^{u,m}(t)$$

EndFor weights and biases

$$\frac{\partial \mathbf{a}^u(t)}{\partial \mathbf{x}^T} = \frac{\partial^e \mathbf{a}^u(t)}{\partial \mathbf{x}^T} + \sum_{x \in E_S^X(u)} \mathbf{S}^{u,x}(t) \sum_{u' \in E_{LW}^U(x)} \sum_{d \in DL_{x,u'}} \mathbf{LW}^{x,u'}(d) \times \frac{\partial \mathbf{a}^{u'}(t-d)}{\partial \mathbf{x}^T}$$

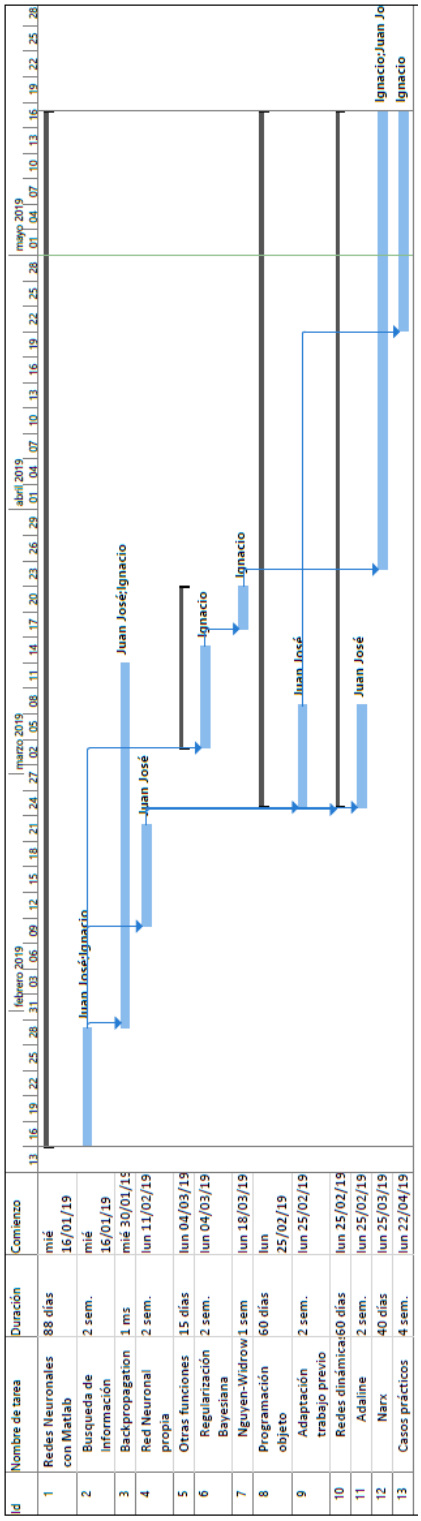
EndFor u

EndFor t

Compute Gradients

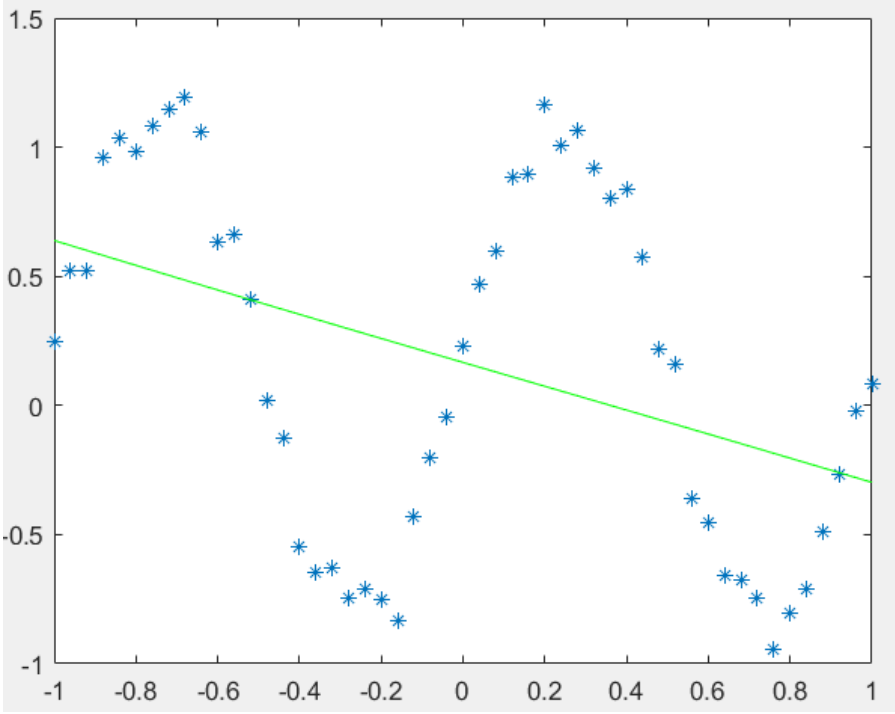
$$\frac{\partial F}{\partial \mathbf{x}} = \sum_{t=1}^{Q} \sum_{u \in U} \left[\left[\frac{\partial \mathbf{a}^u(t)}{\partial \mathbf{x}^T}\right]^T \times \frac{\partial^e F}{\partial \mathbf{a}^u(t)}\right]$$

Diagrama Gantt



Id	Nombre de tarea	Duración	Comienzo
1	Redes Neuronales con Matlab	88 días	mié 16/01/19
2	Busqueda de Información	2 sem.	mié 16/01/19
3	Backpropagation	1 ms	mié 30/01/19
4	Red Neuronal propia	2 sem.	lun 11/02/19
5	Otras funciones	15 días	lun 04/03/19
6	Regularización Bayesiana	2 sem.	lun 04/03/19
7	Nguyen-Widrow	1 sem	lun 18/03/19
8	Programación objeto	60 días	lun 25/02/19
9	Adaptación trabajo previo	2 sem.	lun 25/02/19
10	Redes dinámicas	60 días	lun 25/02/19
11	Adaline	2 sem.	lun 25/02/19
12	Narx	40 días	lun 25/03/19
13	Casos prácticos	4 sem.	lun 22/04/19

Error Regularización Bayesiana



Demostración derivadas

Función sigmoidea

$$f(x)=\frac{1}{1+e^{-x}}$$

$$f'(x)=\frac{e^{-x}}{\left(1+e^{-x}\right)^2}=\frac{e^{-x}+1-1}{\left(1+e^{-x}\right)^2}=\frac{1}{1+e^{-x}}-\frac{1}{\left(1+e^{-x}\right)^2}=f(x)-f(x)^2=f(x)\cdot(1-f(x))$$

Tangente hiperbólica

$$f(x)=\frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$$

$$f'(x)=\frac{\left(e^{x}+e^{-x}\right)^2-\left(e^{x}-e^{-x}\right)^2}{\left(e^{x}+e^{-x}\right)^2}=1-\left(\frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}\right)^2=1-f(x)^2=\left(1-f(x)\right)\cdot\left(1+f(x)\right)$$

Anexo II: Código

Para organizar este anexo se ha optado por juntar en un mismo recuadro (varios, si es muy extenso) las funciones de un mismo objeto, que viene indicado en la parte superior derecha. Además, en el pie de página, a la izquierda, encontramos el tipo de objeto del que se trata[1].

Al realizar la organización de los distintos ficheros durante el trabajo se ha optado por agruparlos por carpetas según el tipo de objeto que se tratara (esta organización es la indicadad en los títulos en azul). Así, por ejemplo, base_red, redff y narx irían dentro de la carpeta tipored y jbp, jdf y jbpd dentro de la carpeta tipoJ.

redneu

@redneu > entrena

```
function [c,Ec]=entrena(redn)

objmet=redn.objmet;
objprob=redn.objprob;
objred=redn.objred;

[c,Ec]=resolver(objmet,objprob,objred);
```

@redneu > redneu

```
classdef redneu

    properties
        objmet
        objprob
        objred
    end

    methods
        function obj = redneu(objmet,objprob,objred)

            obj.objmet=objmet;
            obj.objprob=objprob;
            obj.objred=objred;
        end
    end
end
```

[1] Funciones, Ejemplos y Documentos no se tratan de objetos, pero por dar el mismo formato a todo el dcumento se ha procedido igual.

tipored > base_red

```
classdef base_red
    properties
        R
        ncapas
        dim
        nx
        objfun
        pmax
        pmin
        tmax
        tmin
        escal=true
        tipoini='rand'      %aleatorio [-.5 .5]

    end

    methods

        function objred=conf(objred,p,t)
            % objred=conf(objred,p,t)

            Qp=size(p,2);   Qt=size(t,2);
            if Qp~=Qt, error('Error, p y t deben tener el mismo número de
columnas'); end
            if Qp<2,  error('Para configurar hacen falta al menos 2
muestras (p,t)'); end

            RR=size(p,1);
            dM=size(t,1);
            d=[objred.dim dM];
            nxx=d(1)*RR+d(1);
            for m=2:objred.ncapas
                nxx=nxx+d(m)*d(m-1)+d(m);
            end
            objred.R=RR; objred.dim=d;  objred.nx=nxx;

            objred.pmax=max(p,[],2);   objred.pmin=min(p,[],2);
            objred.tmax=max(t,[],2);   objred.tmin=min(t,[],2);
        end
                    ---------------------------------
        function pp=escala(objred,p,tipo)
            % pp=escala(objred,p,tipo)
            if objred.escal==false, pp=p; return,  end
            switch tipo
                case 'p'
                    den=objred.pmax-objred.pmin;
                    pp=-1+2*(p-objred.pmin)./den;
                case 't'
                    den=objred.tmax-objred.tmin;
                    pp=-1+2*(p-objred.tmin)./den;
            end
        end
```

base_red

```
        function tt=escala_inv(objred,t)
            % t=escala_inv(objred,t)
            %
            % Deshace el escalado de t (targets)
            if objred.escal==false, tt=t; return,  end
            tt=(objred.tmax+objred.tmin)/2+(objred.tmax-objred.tmin)*t/2;
        end
                    ---------------------------------
        function aa=vector_tdl(~,a,tdl,dmaxa,dmax)
            %  aa=vector_tdl(~,a,tdl,dmaxa,dmax)
            %  p=[p;0 p(1:end-1)...];
            %  aa entra con las CI , se repite por filas con el traslado
            %  indicado en tld, y la matriz correspondiente sale
            %  recortada en las primeras dmax columnas para sincronizar
            %  los delays de p y a regresivo.

            Q=size(a,2);
            nd=length(tdl);
            aa=zeros(nd,Q);
            for j1=1:nd
                d=tdl(j1);
                aa(j1,d+1:end)=a(1:end-d);
            end
            ind1=dmaxa+1;
            ind2=Q-(dmax-dmaxa);
            aa=aa(:,ind1:ind2);
        end
    end
end
```

redff

tipored > @redff > redff

```
classdef redff < base_red

    properties
        W
        b
    end

    methods
        function obj=redff(varargin)
            if nargin>0
                if (isa(varargin{1},'struct'))
                    datos=varargin{1};
                    obj=obj_st(obj,datos.redff);
                    p=datos.p;    t=datos.t;
                    obj=conf(obj,p,t);
                end
            end
        end
    end
end
```

tipored > @redff > hdat

```
function dat=hdat(~,objfun)

load datos_rn dim p t escal tipoini

R=[];   nx=[];
ncapas=length(objfun);
red=struct('R',R,'ncapas',ncapas,'dim',dim,'nx',nx);

red.objfun=objfun;

W=cell(ncapas,1);  b=cell(ncapas,1);
red.W=W;   red.b=b;

red.pmax=[];   red.pmin=[];   red.tmax=[];   red.tmin=[];

dat.redff=red;
dat.p=p;   dat.t=t;

if ~isempty(escal), dat.redff.escal=escal; end
if ~isempty(tipoini), dat.redff.tipoini=tipoini; end

end
```

redff

tipored > @redff > act

```
function red=act(red,c)

%Para cada capa m, el vector de incógnitas c, tiene primero b y
%despues W por columnas

ncapas=red.ncapas;
R=red.R;
d=red.dim;

k2=0;
k1=k2+1;  k2=k2+d(1);
red.b{1}=c(k1:k2);
k1=k2+1;  k2=k2+d(1)*R;
V=c(k1:k2);
red.W{1}=reshape(V,[],R);

for m=2:ncapas
    k1=k2+1;  k2=k2+d(m);
    V=c(k1:k2);
    red.b{m}=c(k1:k2);
    k1=k2+1;  k2=k2+d(m)*d(m-1);
    V=c(k1:k2);
    red.W{m}=reshape(V,[],d(m-1));
end
```

tipored > @redff > sim

```
function a=sim(objred,p,~)

ncapas=objred.ncapas;
[~,Q]=size(p);
unos=ones(1,Q);

W=objred.W;
b=objred.b;
objfun=objred.objfun;

if nargin==2
    p=escala(objred,p,'p');
end
n=W{1}*p+b{1}*unos;
a=calf(objfun{1},n);
for j1=2:ncapas
    n=W{j1}*a+b{j1}*unos;
    a=calf(objfun{j1},n);
end

if nargin==2
    a=escala_inv(objred,a);
end
```

redff

tipored > @redff > inic

```
function c0=inic(objred,x,t)
Qp=size(x,2);   Qt=size(t,2);
if Qp~=Qt, error('Error, p y t deben tener el mismo número de columnas');
end
if Qp<2,  error('Para configurar hacen falta al menos 2 muestras (p,t)');
end

switch objred.tipoini
    case 'NgWi'
        R=size(x,1);
        S_sal=size(t,1);
        S=[objred.dim S_sal];
        WW=randn(S(1),R);
        norm=kron(ones(1,R),sqrt(sum(WW.^2,2)));
        mod=0.7*(S(1))^(1/R);
        k2=0;
        k1=k2+1;  k2=k2+S(1);
        c(k1:k2)=-mod+2*mod*rand(S(1),1);
        k1=k2+1;  k2=k2+S(1)*R;
        c(k1:k2)=mod.*(WW./norm);
        for j1=2:objred.ncapas
            WW=randn(S(j1),S(j1-1));
            norm=kron(ones(1,S(j1-1)),sqrt(sum(WW.^2,2)));
            mod=0.7*(S(j1))^(1/S(j1-1));
            k1=k2+1;  k2=k2+S(j1);
            c(k1:k2)=-mod+2*mod*rand(S(j1),1);
            k1=k2+1;  k2=k2+S(j1)*S(j1-1);
            c(k1:k2)=mod./norm.*WW;
        end
        c0=c';
    case 'rand'
        rng('default');
        c0=rand(objred.nx,1);
        % c0=-1+2*c0; %Ahora los calores aleatorios son entre -1 y +1
        c0=-0.5+c0;     %Ahora los calores aleatorios son entre -0.5 y +0.5
    case 'unos'
        c0=ones(nx,1)*0.5;
    case 'ceros'
         c0=zeros(nx,1);
end
end
```

narx

tipored > @narx > narx

```
classdef narx < redff

    properties
        tdl=0;
        tdla=0;
        Wr
    end

    methods
        function obj=narx(varargin)

            if nargin==0 || isempty(varargin{1})
                args{1}=[];
            elseif isstruct(varargin{1})
                datos=varargin{1};
                args{1}=datos;
            else
                error('arg invalido')
            end
            obj=obj@redff(args{:});

            if nargin>0
                    R=obj.R;
                    delays=varargin{1}.delays;
                    obj=obj_st(obj,delays);
                    obj.nx=obj.nx+((length(obj.tdl)-1)*R+...
                      length(obj.tdla)*obj.dim(end))*obj.dim(1);
            end
        end
    end
end
```

tipored > @narx > hdat

```
function dat=hdat(~,objfun)

load datos_rn tdl tdla

dat=hdat(redff,objfun);  %equivalente a dat=hdat@redff(objfun);

delays.tdl=tdl;
delays.tdla=tdla;
dat.delays=delays;
```

narx

tipored > @narx > act

```
function objred=act(objred,c)

%Para cada capa m, el vector de incógnitas c, tiene primero b y
%despues W por columnas

ncapas=objred.ncapas;
R=objred.R;
d=objred.dim;
tdl=objred.tdl;    nd=length(tdl);
R=nd*R;

k2=0;
k1=k2+1;  k2=k2+d(1);
objred.b{1}=c(k1:k2);
k1=k2+1;  k2=k2+d(1)*R;
V=c(k1:k2);
objred.W{1}=reshape(V,[],R);

tdla=objred.tdla;    nda=length(tdla);
RR=nda*d(end);
k1=k2+1;  k2=k2+d(1)*RR;
V=c(k1:k2);
objred.Wr{1}=reshape(V,[],RR);

for m=2:ncapas
    k1=k2+1;  k2=k2+d(m);
    objred.b{m}=c(k1:k2);
    k1=k2+1;  k2=k2+d(m)*d(m-1);
    V=c(k1:k2);
    objred.W{m}=reshape(V,[],d(m-1));
end
```

tipored > @narx > sim

```
function a=sim(objred,p,t)

W=objred.W;
b=objred.b;
objfun=objred.objfun;
R2=size(t,1);
tdl=objred.tdl;    dmaxp=max(tdl);
Wr=objred.Wr;
tdla=objred.tdla;     dmaxa=max(tdla);
dmax=max(dmaxp,dmaxa);

p=escala(objred,p,'p');
a0=escala(objred, t(:,1:dmaxa),'t');

%No hay que recortar el nº de columnas más que en las CI (dmax->dmaxp)
pp=vector_tdl(objred,p,tdl,dmaxp,dmaxp);
Q=size(pp,2);
ncapas=objred.ncapas;

if tdla==0            %No se quitan las CI, porque no estan incluidas en pp
    unos=ones(1,Q);
    np=W{1}*pp+b{1}*unos;
    a=calf(objfun{1},np);
    for j2=2:ncapas
        n=W{j2}*a+b{j2}.*unos;
        a=calf(objfun{j2},n);
    end
else
    a=zeros(R2,Q+dmax);   a(:,1:dmaxa)=a0;
    for j1=1:Q
        n=W{1}*pp(:,j1)+b{1};
        k=j1+dmaxa;
        aa=[];
        for i=1:length(tdla)
            aa=[aa;a(:,k-tdla(i))];
        end

        nr=Wr{1}*aa;
        n=n+nr;
        aa=calf(objfun{1},n);
        for j2=2:ncapas
            n=W{j2}*aa+b{j2};
            aa=calf(objfun{j2},n);
        end
        a(:,k)=aa;
    end
    a=a(:,dmax+1:end);        %Se quitan las CI
end
a=escala_inv(objred,a);
a=[t(:,1:dmaxa) a];

end
```

tipored > @narx > simtr

```
function a=simtr(objred,p,t)

W=objred.W;
b=objred.b;
objfun=objred.objfun;

tdl=objred.tdl;    dmaxp=max(tdl);

Wr=objred.Wr;
tdla=objred.tdla;     dmaxa=max(tdla);

dmax=max(dmaxp,dmaxa);

p0=p(:,1:dmaxp);   p1=p(:,dmaxp+1:end);
a0=t(:,1:dmaxa);   t1=t(:,dmaxa+1:end);
pp=vector_tdl(objred,p,tdl,dmaxp,dmax);
Q=size(pp,2);
unos=ones(1,Q);
np=W{1}*pp+b{1}*unos;

tt=vector_tdl(objred,t,tdla,dmaxa,dmax);   %whos p t pp tt
nt=Wr{1}*tt;    %  pause
n=np+nt;
a=calf(objfun{1},n);

ncapas=objred.ncapas;
Q=size(a,2);
unos=ones(1,Q);
for j1=2:ncapas
    n=W{j1}*a+b{j1}*unos;
    a=calf(objfun{j1},n);
end

end
```

tipored > @narx > inic

```
function c0=inic(objred,x,t)
Qp=size(x,2);   Qt=size(t,2);
if Qp~=Qt, error('Error, p y t deben tener el mismo número de columnas');
end
if Qp<2,  error('Para configurar hacen falta al menos 2 muestras (p,t)');
end
switch objred.tipoini
    case 'NgWi'
        ltdl=length(objred.tdl);
        ltdla=length(objred.tdla);
        R=size(x,1)*ltdl+objred.dim(end)*ltdla;
        S_sal=size(t,1);
        S=[objred.dim S_sal];
        WW=randn(S(1),R);
        norm=kron(ones(1,R),sqrt(sum(WW.^2,2)));
        mod=0.7*(S(1))^(1/R);
        k2=0;
        k1=k2+1;  k2=k2+S(1);
        c(k1:k2)=-mod+2*mod*rand(S(1),1);
        k1=k2+1;  k2=k2+S(1)*R;
        c(k1:k2)=mod.*(WW./norm);
        for j1=2:objred.ncapas
            WW=randn(S(j1),S(j1-1));
            norm=kron(ones(1,S(j1-1)),sqrt(sum(WW.^2,2)));
            mod=0.7*(S(j1))^(1/S(j1-1));
            k1=k2+1;  k2=k2+S(j1);
            c(k1:k2)=-mod+2*mod*rand(S(j1),1);
            k1=k2+1;  k2=k2+S(j1)*S(j1-1);
            c(k1:k2)=mod./norm.*WW;
        end
        c0=c';
    case 'rand'
        rng('default');
        c0=rand(objred.nx,1);
        % c0=-1+2*c0; %Ahora los calores aleatorios son entre -1 y +1
        c0=-0.5+c0;     %Ahora los calores aleatorios son entre -0.5 y +0.5
    case 'unos'
        c0=ones(nx,1)*0.5;
    case 'ceros'
        c0=zeros(nx,1);
end
end
```

base_fun

tipofun > base_fun

```
classdef base_fun

    %No tiene datos (properties)

    methods
        function S = calS(fun,a,S,W)
            if nargin == 2
                [dM,Q]=size(a);
                S=-kron(cald(fun,a),ones(1,dM))...
                  .*kron(ones(1,Q),eye(dM));
            else
                Q=size(a,2);
                dM=size(S,2)/Q;
                aa = kron(a,ones(1,dM));
                S = cald(fun,aa).*(W'*S);
            end
        end
                    ----------------------------------
        function s = cals(fun,a,s,W)
            if nargin == 3
                t=s;
                [dM,Q]=size(a);
                S=kron(cald(fun,a),ones(1,dM))...
                  .*kron(ones(1,Q),eye(dM));
                s=-2*(t-a).*cald(fun,a);
            else
                Q=size(a,2);
                dM=size(s,2)/Q;
                s = cald(fun,a).*(W'*s);
            end
        end

    end
end
```

tsig

tipofun > tsig

```
classdef tsig < base_fun
    %No tiene datos (properties)

    methods
        function val=calf(~,n)
            val=2./(1+exp(-2*n))-1;
        end
                    ----------------------------------
        function vald = cald(~,a)
            vald=(ones-a).*(ones+a);
        end
    end
end
```

lsig

tipofun > lsig

```
classdef lsig < base_fun
    %No tiene datos (properties)

    methods
        function val=calf(~,n)
            val=1./(1+exp(-n));
        end
                    ----------------------------------
        function vald = cald(~,a)
            vald=(ones-a).*a;
        end
    end
end
```

plin

tipofun > plin

```
classdef plin < base_fun
    %No tiene datos (properties)

    methods
        function val=calf(~,n)
            val=n;
        end
                    ----------------------------------
        function vald = cald(~,a)
            vald=ones(size(a));
        end
    end
end
```

tipomet > base_met

```
classdef base_met
    properties
        objJ=[];
        xini=[];
        tol_gr=1e-4
        tol_Ec=1e-4
        maxiter=600;
        tol_lsg=0.001;
        h_lsg=0.01;
        esc=[];
        mumax=1e10;
        emax=6;
    end

    methods
        function datos=hdat(~,objJ,c0)
            load datos_rn tol_gr tol_Ec h_lsg maxiter esc mumax
            datos.objJ=objJ;
            datos.xini=c0;
            if ~isempty(tol_gr), datos.tol_gr=tol_gr; end
            if ~isempty(tol_Ec), datos.tol_Ec=tol_Ec; end
            if ~isempty(h_lsg), datos.h_lsg=h_lsg; end
            if ~isempty(maxiter), datos.maxiter=maxiter; end
            if ~isempty(esc), datos.esc=esc; end
            if ~isempty(mumax), datos.mumax=mumax; end
            if ~isempty(emax), datos.emax=emax; end
        end
                        ---------------------------------

        function est_err=err(~,R,x,k,conv)
            switch conv
                case 'R'
                    est_err=norm(R);
                case 'x'
                    est_err=norm(x(:,k+1)-x(:,k));
                case 'xr'
                    est_err= (norm(x(:,k+1)-x(:,k))/norm(x(:,k+1)))*100;
                otherwise
                    error('Criterio de parada no valido; usar R, x, o xr ')
            end
        end
```

base_met

```
        function escribir(objmet,caso,arg1,arg2)
            if ~isempty(objmet.esc), return, end
            switch caso
                case 1
                    k=arg1;
                    leyenda1=strcat('**************** iteracion
',int2str(k),...
                        '********************************');
                    disp(leyenda1);
                case 2
                    est_err=arg1;    tol=arg2;
                    leyenda2=strcat('Estimación error =
',num2str(est_err),...
                        '       Tolerancia = ', num2str(tol));
                    disp(leyenda2);
            end
        end
                        ---------------------------------

        function [xmin,amin]=lsg(objmet,objprob,objred,x0,d0)
            %[xmin,amin]=lsg(objmet,objprob,objred,x0,d0)
            %
            %Line search con golden ratio
            %Se busca el mínimo de f(a)=Ec(x0+a*d0)

            tol=objmet.tol_lsg;
            h=objmet.h_lsg;
            a=0;    x=x0;          [~,fx]=calcE(objprob,objred,x);
            ap=h;   xp=x0+d0*ap;
            [~,fxp]=calcE(objprob,objred,xp);
            j1=1;
            while fxp>fx  && j1<=10
                h=h/2;  %h=-h;
                ap=h; xp=x0+d0*ap;   [~,fxp]=calcE(objprob,objred,xp);
                j1=j1+1;
            end
            if j1==10+1, error('h_lsg es demasiado grande'), end

            j1=1;
            while fxp<=fx && j1<=20
                aa=a;   xa=x;  fxa=fx;
                a=ap;   x=xp;  fx=fxp;
                ap=a+h; xp=x0+d0*ap;  [~,fxp]=calcE(objprob,objred,xp);
                j1=j1+1;
            end
```

base_met

```
            if j1==20
                error('h es demasiado pequeño ò no existe mínimo')
            end

            a1=aa;   fa1=fxa;
            a2=ap;   fa2=fxp;
            alfa=(3-sqrt(5))/2;

            a3=(1-alfa)*a1+alfa*a2;
            [~,fa3]=calcE(objprob,objred,x0+d0*a3);
            a4=alfa*a1+(1-alfa)*a2;
            [~,fa4]=calcE(objprob,objred,x0+d0*a4);
            interv=[a1 a2];  L=a2-a1;

            while L > tol*h
                if fa1>fa3 && fa3>fa4
                    interv=[a3 a2];
                    a1=a3;  fa1=fa3;
                    a3=a4;  fa3=fa4;
                    a4=alfa*a1+(1-alfa)*a2;
                    [~,fa4]=calcE(objprob,objred,x0+d0*a4);
                else
                    interv=[a1 a4];
                    a2=a4;    fa2=fa4;
                    a4=a3;    fa4=fa3;
                    a3=(1-alfa)*a1+alfa*a2;
                    [~,fa3]=calcE(objprob,objred,x0+d0*a3);
                end
                L=interv(2)-interv(1);
            end
            amin=(interv(2)+interv(1))/2;
            xmin=x0+d0*amin;

        end
    end
end
```

gr

tipomet > @gr > gr

```
classdef  gr < base_met

    methods

        function  obj= gr(varargin)

            if nargin>0
                if (isa(varargin{1},'struct')) % datos
                    datos=varargin{1};
                    obj=obj_st(obj,datos);
                end
            end
        end
    end
end
```

gr

tipomet > @gr > resolver

```
function [x,Ec]=resolver(objmet,objprob,objred,varargin)
%   [c,Ec]=resolver(objmet,objprob,objred,varargin)

%Funcion que resuelve un sistema no lineal de ecuaciones %mediante el
metodo del gradiente, a partir de un vector de %prueba inicial  "x0".
%
%En tol y maxiter se definen la tolerancia absoluta (en la norma euclidea)
%y el maximo numero de iteraciones permitido. Por defecto (introducir [])
%los valores son 0.001 y 20 respectivamente.

objJ=objmet.objJ;
tol=objmet.tol_gr;     tol_Ec=objmet.tol_Ec;
e_max=objmet.emax;
cont_e=0;
maxiter=objmet.maxiter;   conv='R';

if isempty(tol),  tol=.001;  end
if isempty(maxiter), maxiter=20;  end

x0=objmet.xini;
x=zeros(length(x0),maxiter+1);  Ec=zeros(maxiter+1,1);

k=0; escribir(objmet,1,k)
x(:,1)=x0;
[g0,Eck,Ev]=calcG(objJ,objprob,objred,x0);  Ec(1)=Eck;
est_err=err(objmet,g0,x,k,conv);   escribir(objmet,2,est_err,tol)
gk=g0;
Evm=Ev;
while est_err>tol && k<maxiter  && Eck>tol_Ec && cont_e<e_max
    k=k+1;
    escribir(objmet,1,k)
    x(:,k+1)=lsg(objmet,objprob,objred,x(:,k),-gk);
    tic
    [gk,Eck,Ev]=calcG(objJ,objprob,objred,x(:,k+1));
    toc
    Ec(k+1)=Eck;
    if Ev>Evm
        cont_e=cont_e+1;
    else
        Evm=Ev;
        cont_e=0;
    end
    est_err=err(objmet,gk,x,k,conv);   escribir(objmet,2,est_err,tol)
end
niter=k;  x=x(:,1:niter+1);  Ec=Ec(1:niter+1);
end
```

grc

tipomet > @grc > grc

```
classdef  grc < base_met

    methods

        function  obj= grc (varargin)

            %  obj=grc(datos);

            %  Constructor del Metodo de Gradiente Conjugado

            if nargin>0
                if (isa(varargin{1},'struct')) % datos
                    datos=varargin{1};
                    obj=obj_st(obj,datos);
                end
            end
        end
    end
end
```

tipomet > @grc > resolver

```
function [x,Ec]=resolver(objmet,objprob,objred,varargin)
%   [x,Ec]=resolver(objmet,objprob,objred,varargin)

%Funcion que resuelve un sistema no lineal de ecuaciones %mediante el
%metodo del gradiente conjugado, a partir de un %vector de prueba inicial
%"x0".
%En tol y maxiter se definen la tolerancia absoluta (en la norma euclidea)
%y el maximo numero de iteraciones permitido. Por defecto (introducir [])
%los valores son 0.001 y 20 respectivamente.

objJ=objmet.objJ;
tol=objmet.tol_gr;     tol_Ec=objmet.tol_Ec;
maxiter=objmet.maxiter;   conv='R';

if isempty(tol),  tol=.001;  end
if isempty(maxiter), maxiter=20;  end

x0=objmet.xini;   nx=objred.nx;
x=zeros(nx,nx+maxiter+1);  Ec=zeros(nx+maxiter+1,1);
emax=objmet.emax;
cont_e=0;

x(:,1)=x0;
k=0;   est_err=tol+1;  Eck=tol+1;
Evm=100000;
```

grc

```
while est_err>tol && k<=maxiter  && Eck>tol_Ec && cont_e<emax
    kk=0; escribir(objmet,1,k)

    [g0,Eck,Ev]=calcG(objJ,objprob,objred,x0);
    est_err=err(objmet,g0,x,k,conv);   escribir(objmet,2,est_err,tol)
    gk=g0;   dk=-g0; gka2=sum(gk.*gk);
    if Ev>Evm
        cont_e=cont_e+1;
    else
        Evm=Ev;
        cont_e=0;
    end
    while est_err>tol && kk<=objred.nx  && Eck>tol_Ec && cont_e<emax
        k=k+1;  kk=kk+1;   Ec(k)=Eck;
        escribir(objmet,1,k)
        x(:,k+1)=lsg(objmet,objprob,objred,x(:,k),dk);
        [gk,Eck,Ev]=calcG(objJ,objprob,objred,x(:,k+1));
        Ec(k+1)=Eck;
        gk2=sum(gk.*gk);
        beta=gk2/gka2;
        dk=-gk+beta*dk;
        gka2=gk2;
        if Ev>Evm
            cont_e=cont_e+1;
        else
            Evm=Ev;
            cont_e=0;
        end
        est_err=err(objmet,gk,x,k,conv);   escribir(objmet,2,est_err,tol)
    end
    x0= x(:,k+1);
end
niter=k;  x=x(:,1:niter+1);  Ec=Ec(1:niter+1);
end
```

lmg

tipomet > @lmg > lmg

```
classdef  lmg<base_met

    properties
        mu=0.001
        fac=10
    end

    methods

        function  obj= lmg (varargin)
            %
            %  obj=lmg(datos);
            %  Constructor del Metodo de Levenberg-Marquardt

            if nargin>0
                if (isa(varargin{1},'struct')) % datos
                    datos=varargin{1};
                    obj=obj_st(obj,datos);
                end
            end
        end
    end
end
```

tipomet > @lmg > resolver

```
function [x,Ec]=resolver(objmet,objprob,objred,varargin)
%   [c,Ec]=resolver(objmet,objprob,objred,varargin)

%Funcion que resuelve un sistema no lineal de ecuaciones mediante el metodo
%Levenberg-Marquardt, a partir de un vector de prueba inicial  "x0".
%En tol y maxiter se definen la tolerancia absoluta (en la norma euclidea)
%y el maximo numero de iteraciones permitido. Por defecto (introducir [])
%los valores son 0.001 y 20 respectivamente.

objJ=objmet.objJ;
tol=objmet.tol_gr;     tol_Ec=objmet.tol_Ec;
maxiter=objmet.maxiter;   conv='R';
mumax=objmet.mumax;
mu=objmet.mu;  fac=objmet.fac;
cont_e=0;
emax=objmet.emax;

if isempty(tol),  tol=.001;  end
if isempty(maxiter), maxiter=20;  end
```

lmg

```
x0=objmet.xini;
x=zeros(length(x0),maxiter+1);  Ec=zeros(maxiter+1,1);
%Ev=zeros(maxiter+1,1);

k=0; escribir(objmet,1,k)
x(:,1)=x0;
[e,j,Ec(k+1),Ev]=ej(objJ,objprob,objred,x0); R=j'*e;
Ra=R;  ja=j;  Eck=Ec(k+1);
n=length(R);  MI=eye(n);
J=j'*j+mu*MI;
est_err=err(objmet,R,x,k,conv);   escribir(objmet,2,est_err,tol)
Evm=Ev;
while est_err>tol && k<maxiter  && Eck>tol_Ec && mu<mumax && cont_e<emax
    k=k+1;
    escribir(objmet,1,k)
    incrx=-(J\R);
    x(:,k+1)=x(:,k)+incrx;
    [e,j,Ec(k+1),Ev]=ej(objJ,objprob,objred,x(:,k+1));   R=j'*e;
    est_err=err(objmet,R,x,k,conv);   escribir(objmet,2,est_err,tol)
    Eck=Ec(k+1);
    if Ec(k+1)<Ec(k)
        mu=mu/fac;
    else
        mu=mu*fac;
        R=Ra;  j=ja;  k=k-1;
    end
    if Ev>Evm
        cont_e=cont_e+1;
    else
        Evm=Ev;
        cont_e=0;
    end
    J=j'*j+mu*MI;          %   J=j'*j;   J=J+mu*diag(diag(J));
    Ra=R;  ja=j; %pause
end
niter=k;  x=x(:,1:niter+1);  Ec=Ec(1:niter+1);
end
```

byr

tipomet > @byr > byr

```
classdef  byr < lmg

    methods

        function  obj= byr (varargin)

            if nargin==0 || isempty(varargin{1})
                args{1}=[];
            elseif isstruct(varargin{1})
                datos=varargin{1};
                args{1}=datos;
            else
                error( 'arg invalido')
            end
            obj=obj@lmg(args{:});
        end
    end
end
```

tipomet > @byr > resolver

```
function [x,Ec]=resolver(objmet,objprob,objred,varargin)
%   [c,Ec]=resolver(objmet,objprob,objred,varargin)

%Funcion que resuelve un sistema no lineal de ecuaciones mediante el metodo
%Levenberg-Marquardt, a partir de un vector de prueba inicial  "x0".
%En tol y maxiter se definen la tolerancia absoluta (en la norma euclidea)
%y el maximo numero de iteraciones permitido. Por defecto (introducir [])
%los valores son 0.001 y 20 respectivamente.

objJ=objmet.objJ;
tol=objmet.tol_gr;     tol_Ec=objmet.tol_Ec;
maxiter=objmet.maxiter;   conv='R';
mumax=objmet.mumax

mu=objmet.mu;  fac=objmet.fac;

if isempty(tol),  tol=.001;  end
if isempty(maxiter), maxiter=20;  end

x0=objmet.xini;
x=zeros(length(x0),maxiter+1);  Ec=zeros(maxiter+1,1);

k=0; escribir(objmet,1,k)
x(:,1)=x0;
[e,j,Ec(k+1)]=ej(objJ,objprob,objred,x0);
```

byr

```
%Calcular gamma, alfa, beta y H
N=size(objprob.p,2);
n=length(x0);  MI=eye(n);   Ew=sum(x0.^2);  Ed=Ec(k+1);
gamma=n,    alfa=gamma/(2*Ew),  beta=(N-gamma)/(2*Ed)
H=2*beta*(j'*j)+2*alfa*MI;

%Calcular R y H
R=2*beta*(j'*e)+2*alfa*x0;
J=H+mu*MI;
gamma=n-2*alfa*trace(inv(H));
alfa=gamma/(2*Ew);   beta=(N-gamma)/(2*Ed);

%Guardar R y H
Ra=R;  Ha=H;

Eck=Ec(k+1)
est_err=err(objmet,R,x,k,conv);   escribir(objmet,2,est_err,tol)

while est_err>tol && k<maxiter  && Eck>tol_Ec && mu<mumax
    k=k+1;
    escribir(objmet,1,k)
    incrx=-(J\R);
    x(:,k+1)=x(:,k)+incrx;
    [e,j,Ec(k+1)]=ej(objJ,objprob,objred,x(:,k+1));

    Ew=sum(x(:,k+1).^2);  Ed=Ec(k+1);
    H=2*beta*(j'*j)+2*(alfa)*MI;
    gamma=n-2*alfa*trace(inv(H));
    alfa=gamma/(2*Ew);   beta=(N-gamma)/(2*Ed);

    R=2*beta*j'*e+2*alfa*x(:,k+1);
    est_err=err(objmet,R,x(:,k+1),k,conv);
    escribir(objmet,2,est_err,tol)

    Eck=Ec(k+1);
    if Ec(k+1)<Ec(k)
        mu=mu/fac;
    else
        mu=mu*fac;
        R=Ra;  H=Ha;  k=k-1;
    end
    J=H+mu*MI;

    %Guardar R y H
    Ra=R;  Ha=H;
end
niter=k;  x=x(:,1:niter+1);  Ec=Ec(1:niter+1);
alfa, beta, gamma
end
```

jdf

tipoJ > jdf > jdf

```
classdef jdf
   %No tiene datos (properties)
   %Los métodos están en carpeta @jdf
end
```

tipoJ > jdf > calcG

```
function [g,Ec,Ev]=calcG(~,objprob,objred,x)
%  [g,Ec,Ev]=calcG(~,objprob,objred,x)
%Funcion dato para la función aproximante neural network
%ek: vector (nx1) con los errores o desviaciones (tantas como n° de
%muestras)
%j: matriz jacobiana del vector error ek respecto al vector incógnita x

objred=act(objred,x);    %Actualiza las incognitas de objred

[~,Ec,Ev]=calcE(objprob,objred,x);   %  ek=(dM x Q),  ek(:)=(dM.Q x 1)

%Matriz jacobiana (nxm) del vector error(nx1) respecto a los m=nx
parámetros

m=length(x);    g=zeros(m,1);
ep=sqrt(eps);
for j1=1:m
    h=ep*abs(x(j1));
    if h<ep, h=ep; end
    xh=x;
    xh(j1)=xh(j1)+h;
    [~,Ech]=calcE(objprob,objred,xh);
    g(j1)=(Ech-Ec)/h;
end

end
```

tipoJ > jdf > calcJ

```
function j=calcJ(objJ,objprob,objred,x)
% J=calcJ(objJ,objprob,objred,varargin)

%Calcula el jacobiano de la funcion vectorial ek ( vector error)respecto a
%la variable vectorial x.  Mediante diferencias finitas.


[~,j,~]=ej(objJ,objprob,objred,x);
```

jdf

tipoJ > jdf > ej

```
function [ek,j,Ec,Ev]=ej(~,objprob,objred,x)
% function [ek,j,Ec,Ev]=ej(objprob,objJ,objred,c,pk,tk)
%  [ek,j,Ec]=ej(objprob,objJ,objred,c,pk,tk)
%Funcion dato para la función aproximante neural network
%ek: vector (nx1) con los errores o desviaciones (tantas como n° de
%muestras)
%j: matriz jacobiana del vector error ek respecto al vector incógnita x

objred=act(objred,x);    %Actualiza las incognitas de objred

[ek,Ec,Ev]=calcE(objprob,objred,x);   %  ek=(dM x Q),  ek(:)=(dM.Q x 1)

%Matriz jacobiana (nxm) del vector error(nx1) respecto a los m=nx
parámetros

m=size(ek(:),1); n=length(x);    j=zeros(m,n);   %j=(dM.Q x nx)
ep=sqrt(eps);
for j1=1:n
    h=ep*abs(x(j1));
    if h<ep, h=ep; end
    xh=x;
    xh(j1)=xh(j1)+h;
    ekh=calcE(objprob,objred,xh);
    j(:,j1)=(ekh(:)-ek(:))/h;
end

end
```

jbp

tipoJ > jbp > jbp

```
classdef jbp
   %No tiene datos (properties)
   %Los métodos están en carpeta @jbp
end
```

tipoJ > jbp > calcG

```
function [g,Ec,Ev]=calcG(~,objprob,objred,x)
%  [g,Ec,Ev]=calcG(~,objprob,objred,~)

p=objprob.p;    t=objprob.t;

objred=act(objred,x);    %Actualiza las incognitas de objred

ncapas=objred.ncapas;
[R,Q]=size(p);
unos=ones(1,Q);

W=objred.W;
b=objred.b;

objfun=objred.objfun;
d=objred.dim;
d=[R d];

a=cell(ncapas+1,1);
a{1}=p;
for m=1:ncapas
    n=W{m}*a{m}+b{m}*unos;
    a{m+1}=calf(objfun{m},n);
end

%s: Sensibilidad (gradiente) de las capas.

s = cals(objfun{end},a{end},t);    %Es la Sensibilidad de la ultima capa
db=sum(s,2);
ss=kron(ones(d(end-1),1),s);
aa=kron(a{end-1},ones(d(end),1));
dw=aa.*ss;   dw=sum(dw,2);
g =[db; dw];
for m=ncapas-1:-1:1
    s=cals(objfun{m},a{m+1},s,W{m+1});
    db=sum(s,2);
    ss=kron(ones(d(m),1),s);
    aa=kron(a{m},ones(d(m+1),1));
    dw=aa.*ss;   dw=sum(dw,2);
    g =[db; dw; g];
end

[~,Ec,Ev]=calcE(objprob,objred,x);
```

jbp

tipoJ > jbp > ej

```
function [ek,j,Ec]=ej(objJ,objprob,objred,c)
%Funcion dato para la función aproximante neural network
%ek: vector (nx1) con los errores o desviaciones (tantas como n° de
%muestras)
%j: matriz jacobiana del vector error ek respecto al vector incógnita c

objred=act(objred,c);    %Actualiza las incognitas de objred
[ek,Ec]=calcE(objprob,objred,c);
j=calcJ(objJ,objred,objprob.p);
end
```

tipoJ > jbp > calcJ

```
function  jac=calcJ(objJ,objred,p)

ncapas=objred.ncapas;
[~,Q]=size(p);
W=objred.W;
b=objred.b;

objfun=objred.objfun;
dM=objred.dim(end);
a=cell(ncapas+1,1);
a{1}=p;
for m=2:ncapas+1
    n=W{m-1}*a{m-1}+b{m-1};
    a{m}=calf(objfun{m-1},n);
end

S=calS(objfun{end},a{end});
dw=jew(objJ,a{end-1},S,dM);
jac=[S' dw'];
for m=ncapas-1:-1:1
    S=calS(objfun{m},a{m+1},S,W{m+1});
    dw=jew(objJ,a{m},S,dM);
    jac=[S' dw' jac];
end
```

tipoJ > jbp > jew

```
function jac = jew(objJ,a,S,dM)
%Fórmula backpropagation para una capa:
%matriz jacobiana del error e respecto a los pesos de la capa
%sen: sensibilidad  de la capa
%a: entrada a la capa= salida capa anterior

dm=size(S,1);      %d(m)
dm_1=size(a,1);      %d(m-1)

jac=kron(a,ones(dm,dM)).*kron(ones(dm_1,1),S);
```

jbpd

tipoJ > jbpd > jbpd

```
classdef jbpd < jbp
   %No tiene datos (properties)
   %Los métodos están en carpeta @jbpd
end
```

tipoJ > jbpd > ej

```
function [ek,j,Ec,a]=ej(objJ,objprob,objred,c)
% function [ek,j,Ec,a]=ej(objprob,objJ,objred,c,pk,tk)
%  [ek,j,Ec]=ej(objprob,objJ,objred,c,pk,tk)
%Funcion dato para la función aproximante neural network
%ek: vector (nx1) con los errores o desviaciones (tantas como n° de
%muestras)
%j: matriz jacobiana del vector error ek respecto al vector incógnita c

objred=act(objred,c);    %Actualiza las incognitas de objred

[ek,Ec]=calcE(objprob,objred,c);

%Matriz jacobiana (nxm) del vector error(nx1) respecto a los m=nx
parámetros

j=calcJ(objJ,objred,objprob.p,objprob.t);   %j=j';
a=0;
end
```

jbpd

tipoJ > jbpd > calcJ

```
function jac = calJ(objJ,objred,p,t)

W=objred.W;
Wr=objred.Wr;
W{1}=[W{1} Wr{1}];
b=objred.b;
objfun=objred.objfun;
dM=objred.dim(end);

tdl=objred.tdl;    dmaxp=max(tdl);
tdla=objred.tdla;     dmaxa=max(tdla);
dmax=max(dmaxp,dmaxa);

% p0=p(1:dmaxp);   p1=p(dmaxp+1:end);
% a0=t(1:dmaxa);   t1=t(dmaxa+1:end);

pp=vector_tdl(objred,p,tdl,dmaxp,dmax);
tt=vector_tdl(objred,t,tdla,dmaxa,dmax);
Q=size(pp,2);
unos=ones(1,Q);

ncapas=objred.ncapas;
a=cell(1,ncapas+1);
a{1}=[pp;tt];

for j1=1:ncapas
    n=W{j1}*a{j1}+b{j1}*unos;
    a{j1+1}=calf(objfun{j1},n);
end


S=calS(objfun{end},a{end});
dw=jew(objJ,a{end-1},S,dM);
jac=[S' dw'];
for m=ncapas-1:-1:1
    S=calS(objfun{m},a{m+1},S,W{m+1});
    dw=jew(objJ,a{m},S,dM);
    jac=[S' dw' jac];
end
```

base_prob

tipoprob > base_prob

```
classdef base_prob
    properties
        p
        t
        p_val
        t_val
    end


    methods

        function [redn,objprob,objmet,objJ,objred,objfun]...
                 =haz_obj(objprob)

            load datos_rn red fun met metJ prob p t prop

            objfun=cell(1);
            for j1=1:length(fun)
                objfun{j1}=feval(fun{j1});
            end

            objred=feval(red);   datred=hdat(objred,objfun);
            objred=feval(red,datred);

            p=escala(objred,p,'p');
            t=escala(objred,t,'t');
            [p,t,p_val,t_val]=dividir(prop,p,t);

            datprob=hdat(objprob,p,t,p_val,t_val);
            objprob=feval(prob,datprob);

            objJ=feval(metJ);
            c0=inic(objred,p,t);
            objmet=feval(met); datmet=hdat(objmet,objJ,c0);
            objmet=feval(met,datmet);

            redn=redneu(objmet,objprob,objred);
        end
                    ---------------------------------
        function datos=hdat(~,p,t,p_val,t_val)
            datos.p=p;  datos.t=t;
            datos.p_val=p_val;  datos.t_val=t_val;
        end
    end
end
```

tipoprob > @fit > fit

```
classdef fit < base_prob

   %Los datos (properties) los hereda de base_prob

    methods

        function obj = fit(varargin)    %Constructor objeto fit
            if nargin>0
                if (isa(varargin{1},'struct'))
                    datos=varargin{1};
                    obj=obj_st(obj,datos);
                end
            end
        end
    end

    %Los métodos restantes están en carpeta @fin

end
```

tipoprob > @fit > calcE

```
function [ek,Ec,Ev]=calcE(objprob,objred,c)
%  [ek,Ec]=calcE(~,c,pk,tk,objred)
%Funcion dato para la función aproximante neural network
%ek: vector (nx1) con los errores o desviaciones (tantas como n° de
%muestras)
%j: matriz jacobiana del vector error ek respecto al vector incógnita c

objred=act(objred,c);    %Actualiza las incognitas de objred
if isempty(objprob.p_val)
    Ev=0;
else
    pk=objprob.p_val;   tk=objprob.t_val;
    ak=sim(objred,pk,tk);   %whos pk ak
    ek=(tk-ak);   ek=ek(:);
    Ev=sum(ek.^2);
end
pk=objprob.p;   tk=objprob.t;
ak=sim(objred,pk,tk);   %whos pk ak
ek=(tk-ak);   ek=ek(:);
Ec=sum(ek.^2);
```

din

tipoprob > @din > din

```
classdef din < fit
   %Los datos (properties) los hereda de fit
   %Los métodos están en carpeta @din
end
```

tipoprob > @din > calcE

```
function [ek,Ec]=calcE(objprob,objred,c)

%Funcion dato para la función aproximante neural network
%ek: vector (nx1) con los errores o desviaciones (tantas como n° de
%muestras)
%j: matriz jacobiana del vector error ek respecto al vector incógnita c

pk=objprob.p;   tk=objprob.t;
objred=act(objred,c);    %Actualiza las incognitas de objred

ak=simtr(objred,pk,tk); % whos ak tk

tdla=objred.tdla;     dmaxa=max(tdla);
tk=tk(dmaxa+1:end);   % whos ak tk, pause
ek=(tk-ak);   ek=ek(:);

Ec=sum(ek.^2);

end
```

Funciones > guardadatos

```
if ~exist('escal'),escal=[];end
if ~exist('tipoini'),tipoini=[];end
if ~exist('h_lsg'), h_lsg=[]; end
if ~exist('tol_gr'), tol_gr=[]; end
if ~exist('tol_Ec'), tol_Ec=[]; end
if ~exist('maxiter'), maxiter=[]; end
if ~exist('esc'), esc=[]; end
if ~exist('prop'), prop=1; end
if ~exist('mumax'), mumax=[]; end
if ~exist('emax'), emax=[]; end
save datos_rn
```

Funciones > obj_st

```
function obj=obj_st(obj,st)

%Pasa los campos de la estructura st a campos del objeto obj
%Para la definición de los objetos a partir de las estructuras
%proporcionadas por hdat

nombres=fieldnames(st);
num=length(nombres);
for j1=1:num
    campo=nombres{j1};
    valor=getfield(st,campo);
    eval(['obj.',campo,'=','valor;'])
end
```

gfit4

Ejemplos > gfit4

```
%Guion gfit4
%Método de Levenberg-Marquardt

clear, preprn
%-----------------
% Datos
[x,t] = simplefit_dataset;
plot(x,t,'*'),  hold on

%Definición de la red
fun={@tsig @plin};
dim=[10];
red=@redff;
% prop=0.7;
tipoini='NgWi';

met=@lmg; metJ=@jbp;   %metJ=@jbp   %metJ=@jdf
prob=@fit;

p=x;
guarda_datos
%-------------------------------------

objprob=feval(@fit);

[redn,objprob,objmet,objJ,objred,objfun]=haz_obj(objprob);
% redn.objmet.maxiter=3

[c,Ec]=entrena(redn);

objred=act(objred,c(:,end));

xx=0:0.05:10;
yy=sim(objred,xx);

plot(xx,yy)
%-----------------
finrn
```

gfit5

Ejemplos > gfit5

```
%Guion gfit5
%Método de Levenberg-Marquardt con Regularización Bayesiana

clear, preprn
%-----------------
% Datos
x=-1:.04:1;
rng('default');
t=sin(2*pi*x)+0.1*rand(size(x));
% plot(x,t,'*'),  hold on

%----------------------
%Definición de la red
fun={@tsig @plin};
dim=[20];
red=@redff;   escal=true;
prop=0.5
met=@lmg; metJ=@jbp;   %metJ=@jbp   %metJ=@jdf
prob=@fit;
tipoini='NgWi';

p=x;  guarda_datos
%-------------------------------------

objprob=feval(@fit);

[redn,objprob,objmet,objJ,objred,objfun]=haz_obj(objprob);

[c,Ec]=entrena(redn);

objred=act(objred,c(:,end));

%-----------------
finrn

% xx=-1:0.005:1;   plot(xx,sin(2*pi*xx),'r')
yy=sim(objred,xx);

plot(xx,yy,'m')
```

gmag

Ejemplos > gmag

```
%Guion gmag, Levitación magnética
%Método de Levenberg-Marquardt

clear, preprn
%-----------------

% Datos

[p,t] = maglev_dataset;
p=cell2mat(p);    t=cell2mat(t);

%Definición de la red

fun={@tsig @plin};
dim=[10];
red=@narx

met=@lmg;  %esc='no';
%metJ=@jder;   %metJ=@jbp
metJ=@jbpd
prob=@fit   ,   prob=@din
tipoini='rand'

%Definición tdl
%  tdl=[0 1 2];       tdl=[1 2]; % tdl=[0 1];
%
%  tdla=[0 1 2];      tdla=[1 2];  % tdla=[0 1];

 tdl=[1:2];   tdla=[1:2];

guarda_datos
%--------------------------------------

objprob=feval(prob);

[redn,objprob,objmet,objJ,objred,objfun]=haz_obj(objprob)

[c,Ec]=entrena(redn)
objred=act(objred,c(:,end));

tdla=objred.tdla;     dmaxa=max(tdla);
a0=t(1:dmaxa);
y=sim(objred,p,a0);

figure, plot(y,'b'), hold on
plot(t,'r'), grid on

%-----------------
finrn
```

DOCUMENTOS > MATLAB > preprn

```
warning off
dir_actual=pwd;
ind1=findstr(dir_actual,'red_neu');
ind2=findstr(dir_actual(ind1:end),'\');
if isempty(ind2)
    dir2=dir_actual;
else
    ind=ind1+ind2-2;
    dir2=dir_actual(1:ind);
end

path(pathdef)
addpath(genpath(dir2))

dir_ej=[dir2,'\Ejemplos'];
orden=['cd(dir_ej)'];  eval(orden)
warning on
```

DOCUMENTOS > MATLAB > finrn

```
orden='cd(dir_actual)';
eval(orden)
```