PREDICCIÓN DE COMPORTAMIENTOS A TRAVES DE REDES NEURONALES EN MATLAB.

MSc. Laureano E. Suárez Matínez[1]

1. Universidad de Matanzas "Camilo Cienfuegos", Vía Blanca Km.3, Matanzas, Cuba.

Resumen.

La obtención de funciones de pronóstico que puedan emplearse como criterio de decisión, constituye en muchos casos el objetivo fundamental en tareas investigativas de numerosos disciplinas de la ciencia y la técnica. El empleo de las redes neuronales artificiales con este objetivo ha sido utilizado con éxito aprovechando los avances de la computación e informática, en este estudio se muestra un acercamiento a este novedoso tema con recomendaciones reunidas de la literatura especializada, así como el desarrollo de un ejemplo ilustrativo utilizando para ello el software MatLab R2010a

***Palabras claves**: Redes neuronales artificiales; MatLab; Pronóstico.*

1.- Introducción

Las Redes Neuronales Artificiales (RNA) son sistemas de procesamiento de la información cuya estructura y funcionamiento original estuvo inspirado en el sistema nervioso biológico. Consisten en un gran número de elementos simples de procesamiento llamados nodos o neuronas que están conectados entre sí y organizados en capas [Kumar, 2008].

Cada neurona o nodo es un elemento simple que recibe de una fuente externa una o varias entradas sobre las que se aplica una función *f* de las sumas ponderadas mediante los pesos *w* asociados y que se van modificando en el proceso de aprendizaje (ecuación 1,1). En los pesos se encuentra el conocimiento que tiene la RNA acerca del problema analizado.

$$y_i = \sum w_{ij} y_j \qquad (1,1)$$

El paralelismo de cálculo, la memoria distribuida y la adaptabilidad al entorno, han convertido a las RNA en potentes instrumentos con capacidad para aprender relaciones entre variables sin necesidad de imponer presupuestos o restricciones de partida en los datos.

En el presente documento, nos proponemos realizar un discreto acercamiento a los conceptos básicos utilizados en el campo de las redes neuronales artificiales además de la descripción del funcionamiento de una red perceptrón multicapa entrenada mediante la regla de aprendizaje backpropagation utilizando como herramienta el software MATLAB R2010a

2.- El perceptrón multicapa

Entre los diversos modelos que existen de RNA el más utilizado como aproximador universal de funciones resulta el perceptrón multicapa asociado al algoritmo de aprendizaje *backpropagation error* (propagación del error hacia atrás), también denominado método de gradiente decreciente [Casacuberta, 2012].

Este tipo de red exhibe la capacidad de "aprender" la relación entre un conjunto de entradas y salidas a través del ajuste de los pesos de manera iterativa en la etapa de aprendizaje y posteriormente aplica esta relación a nuevos vectores de entrada que el sistema no ha visto nunca en su etapa de entrenamiento dando una salida activa si la nueva entrada es parecida a las presentadas durante el aprendizaje.

Esta capacidad de generalización convierte a las redes perceptrón multicapa en herramientas de propósito general, flexibles y no lineales.

2.1.- Arquitectura típica.

Un perceptrón multicapa está compuesto por una capa de entrada, una de salida y una o más capas ocultas; en la figura 1.1 podemos observar una red de este tipo. Para identificar la estructura de una red multicapa se utiliza la notación $R:S^1:S^2$ donde el

número de entradas (*R*) va seguido por el número de neuronas en cada capa (*S*) y el exponente indica la capa a la cual la neurona corresponde.

En esta disposición, los elementos del vector de entrada P están conectados solo hacia delante sin que exista retroalimentación por lo que comúnmente se les clasifica en el grupo de redes con arquitectura *feedforward*. Las entradas se vinculan a través de la matriz de pesos (*W*) a cada una de las neuronas donde se transforman en una suma ponderada con valores limitados por los biases (*b*) o umbrales de la neurona, el cual puede verse como un número que indica a partir de que valor de potencial postsináptico la neurona produce una salida significativa [Marcano, 2010], este término es añadido a la suma ponderada para formar el vector modificado [*n*] de *S* elementos (uno por cada neurona) de la red , los que finamente son afectados por la función de activación para formar el vector columna de las salidas [a]; si existe más de una capa estas salidas se pueden convertir en entradas de la próxima sobre las que se repite nuevamente el proceso. La relación matemática que describe el proceso anterior para el ordenamiento de la figura 1.1 queda expresada en la ecuación 2,1.

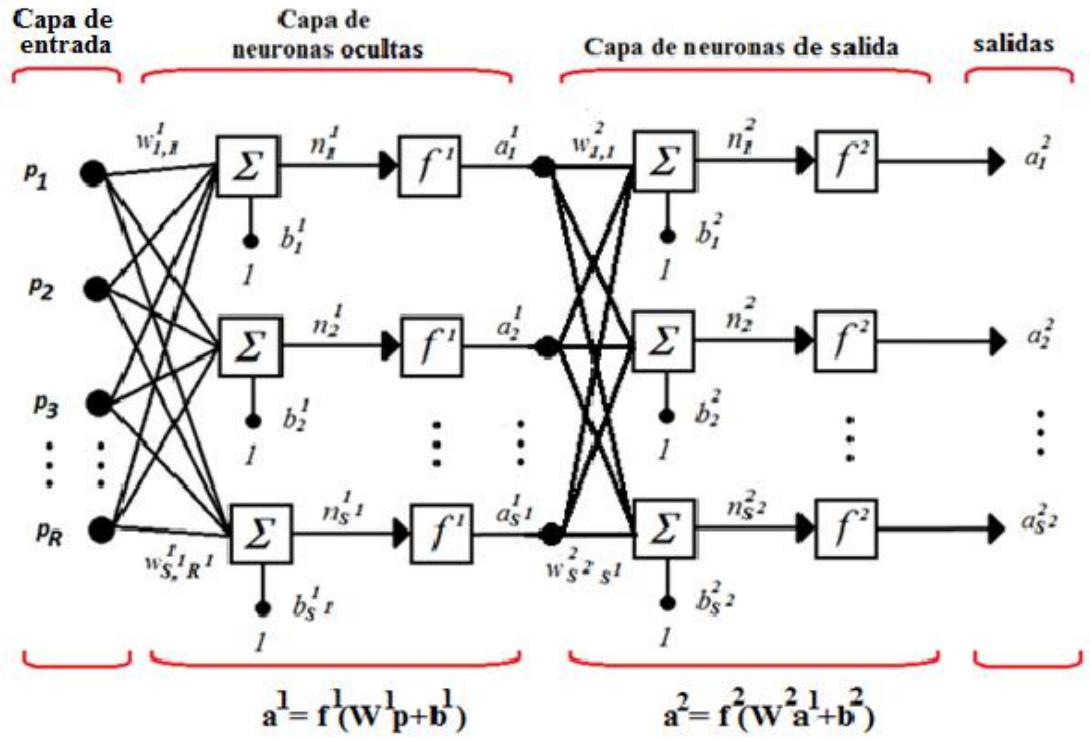

Figura 1.1. Representación de una RNA de una capa con *S* neuronas alimentada por un simple vector de entrada *p* formado por *R* elementos.

$$a^2 = f^2(W^2 f^1(W^1 p + b^1) + b^2) \qquad (2,1)$$

2.2.- Algoritmo *backpropagation*.

El algoritmo *backpropagation* o de retroalimentación debe su nombre a la forma en que se propagan los errores a través de la red para la actualización del valor de los pesos en cada iteración haciendo uso del método del gradiente decreciente hasta encontrar el valor deseado en la etapa de entrenamiento, de manera que coincida la salida de la red

con la salida esperada por el usuario ante la presentación de un determinado patrón. Por este motivo, se dice que el aprendizaje en las redes que emplean este esquema es de tipo supervisado.

La función de error que se pretende minimizar viene dado por la ecuación 2.2

$$E_p{}^2 = \frac{1}{2}\sum_{s=1}^{m}(d_{ps} - a_{ps})^2 \tag{2,2}$$

Aquí *dps* representa la salida deseada en la neurona s ante la presentación de la entrada *p* y *aps* es valor obtenido a la salida para el mismo patrón y Ep^2 es el error cuadrático medio para cada patrón de entrada. A partir de 2.2 se puede obtener una medida general del error en el proceso de aprendizaje en una iteración (ecuación 2.3)

$$E^2 = \sum_{p=1}^{p} E_p{}^2 \tag{2,3}$$

Teniendo en cuenta que Ep^2 es función de los pesos de la red (ecuación 2,1 y 2,2) podemos expresar el gradiente decreciente del error (∇_w) como la derivada parcial de Ep^2 respecto a cada uno de los pesos de la red con signo negativo para garantizar su decremento (ecuación 2,4)

$$\nabla_w = -\sum_{p=1}^{p} \frac{\partial E_p{}^2}{\partial W_{j,i}} \tag{2,4}$$

A nivel práctico, la forma de modificar los pesos de manera iterativa consiste en aplicar la regla de la cadena a la ecuación del gradiente (debido a que el error no es una función implícita de los pesos) y añadir una tasa de aprendizaje η y un factor momento α (ecuación 2,5).

$$\Delta_{Wji(n+1)} = \eta\left(\sum_{p=1}^{p} \delta pj * xpi\right) + \alpha\Delta_{Wji(n)} \tag{2,5}$$

dónde:

δpj representa el error de los pesos en la capa *j* afectada por la función de activación a la salida del nodo *j* (ecuación 2,6) y *xpi* es el valor *i* que tiene el patrón de entrada *p*.

$$\delta pj = Epj * f(net_{pj}) \quad (2,6)$$

aquí:

Epj error de los pesos en la capa j

f(netpj) función de activación a la salida de la neurona.

El uso de estos coeficientes además de procurar optimizar el proceso de convergencia se debe a que la superficie del error puede contener mínimos locales y si el algoritmo no está bien implementado los resultados pueden ser desacertados. La tasa de aprendizaje (η) decide el tamaño del cambio de los pesos en cada iteración: un valor de η demasiado pequeño puede ocasionar una disminución importante en la velocidad de convergencia y la posibilidad de acabar atrapado en un mínimo local por el contrario un ritmo de aprendizaje demasiado grande puede conducir a inestabilidades en la convergencia debido a que se darán saltos en torno al mínimo sin poder alcanzarlo. Por ello, se recomienda elegir el valor de la tasa de aprendizaje entre 0.05 y 0.5. Por su parte el valor del momento (α) ajusta la velocidad de descenso por la superficie del error teniendo en cuenta el signo que toma el incremento de un peso en la iteración anterior, para el que se recomienda utilizar un valor cercano a 1

Existe un cierto número de variaciones en el algoritmo básico las cuales están fundamentadas en otras técnicas de optimización, y que pueden hacer converger el valor de los pesos hasta 100 veces más rápido tales como el gradiente conjugado (TRAINCGF), el Levenberg-Marquardt (TRAINLM) o el método quasi-Newton (TRAINBFG) entre otros todas ellas implementadas en *MatLab R2010a*.

2.3. Consideraciones generales para la correcta aplicación de un perceptrón multicapa.

Con la red perceptrón multicapa, del tipo *backpropagation*, se puede aproximar cualquier función si se escoge una adecuada configuración y una cantidad apropiada de neuronas en la capa oculta, aunque se ha demostrado que para la mayoría de problemas bastará con una sola capa oculta. Este tipo de red, a pesar de ser una excelente aproximador de funciones [Cortina Januch, 2012], tiene como desventaja, la imposibilidad de determinar su configuración exacta para cada aplicación, y en cada caso deben ser probados diferentes ordenamientos hasta encontrar la respuesta deseada. Por esta razón no es aconsejable su uso en relaciones lineales, pues su costo computacional o de implementación no se justifica en comportamientos entre variables sobre las que se sospecha o destaca una linealidad declarada. [Wu et al, 2001] y pueden fácilmente ser reveladas a través de técnicas estadísticas tradicionales.

2.3.1. Selección de las variables de entrada

Como en cualquier otro método de aproximación de funciones debe procurarse que las variables de entrada sean linealmente independientes para lo que puede emplearse la técnica de descomposición de valores singulares, luego de obtener el rango de una matriz a partir de sus valores singulares [Velilla, 2009] o utilizar el método más laborioso de ensayo y error que consiste en entrenar la red con todas las variables e ir

eliminando una variable de entrada cada vez y reentrenar la red. La variable cuya eliminación causa el menor decremento en la ejecución de la red es eliminada. Este procedimiento se repite sucesivamente hasta que llegados a un punto, la eliminación de más variables implica una disminución sensible en la ejecución del modelo [Palmer, 2001].

2.3.2. Cantidad de neuronas.

Aunque no existe una técnica para determinar el número de neuronas que debe contener cada problema específico, esto puede encontrarse a partir de la cantidad de parámetros o pesos a estimar, [W. Sha, 2004] según la relación 2.7, procurando que tal cifra no supere la cuantía de datos disponibles para el entrenamiento, pues, matemáticamente quedaría indeterminado el sistema:

$$P_R \geq N_{param} = (N_i + 1) * N_{h1} + (N_{h1} + 1) * N_{h2} + (N_{h2} + 1) \quad (2,7)$$

Aquí:

N_{param} es el número de parámetros a estimar

N_i es la cantidad de vectores de entrada.

N_{h1} número de neuronas en la primera capa

N_{h2} número de neuronas en la segunda capa.

Un efecto directo del uso de neuronas en cantidades excesivas se manifiesta en la incapacidad de generalización de la red por el fenómeno de sobreajuste, pues la cantidad de pesos resultará superior al número de incógnitas del problema en cuestión y al número de entradas disponibles. En ese estado a la red le será difícil proporcionar una respuesta correcta ante patrones que no han sido empleados en su entrenamiento y que tengan su mismo comportamiento.

2.3.3. Función de activación

Para aprovechar la capacidad de las RNA de aprender relaciones complejas o no lineales entre variables, es absolutamente imprescindible la utilización de funciones no lineales al menos en las neuronas de la capa oculta [Noriega, 2005], en general se utilizará la función sigmoidal logística cuando las variables de salida solo puedan tomar valores positivos dentro de un rango que va desde 0 a 1 o la función tangente hiperbólica cuando se le permite a la función oscilar entre valores positivos y negativos, en el rango de -1 a 1, mientras que en las neuronas de la capa de salida en tareas de predicción o aproximación de una función, generalmente emplean la función de activación lineal, las que reproducen exactamente el valor resultante después de la sumatoria en la neurona (Figura 1.2).

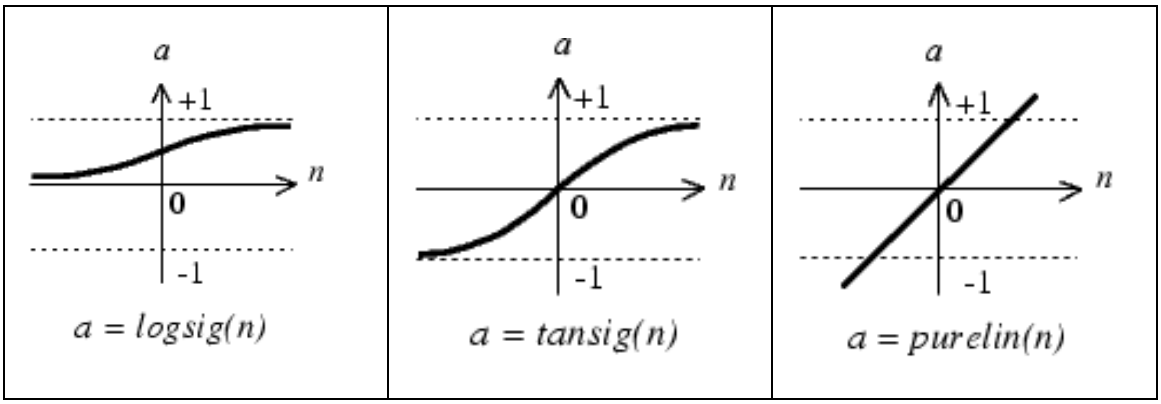

Figura 1.2. Representación de las distintas funciones de activación utilizadas en redes neuronales para tareas de predicción

2.3.4. Elección del valor inicial de los pesos.

En la etapa de entrenamiento el valor de los pesos en la capa de entrada (IW) y en las capas ocultas (LW) puede ser asignado manualmente si se tiene algún conocimiento previo del compromiso o importancia de las variables que intervienen en el fenómeno estudiado, en caso contrario es recomendable asignar estos de manera aleatoria, en un rango de valores entre -0.5 y 0.5 o algo similar y dejar que el programa ajuste sus valores automáticamente.

2.4. Validación o generalización de la red.

Con el objetivo de verificar la capacidad de generalización de la red es conveniente disponer de un grupo de los datos independientes de las variables de entrada que no se hayan utilizados en la fase de entrenamiento, y una vez que se obtenga el modelo cuya configuración de parámetros arroje los mejores resultados, estos deberán ser suministrados a la red y observar el comportamiento de los errores de estimación; en la literatura científica el parámetro más comúnmente utilizado con este propósito es el error cuadrático medio que viene dado por la expresión 2,8.

$$CME = \frac{\sum_{s=1}^{m}(d_{ps} - a_{ps})^2}{m} \qquad (2,8)$$

De forma similar para verificar la inexistencia de sobreajuste en la red se puede realizar la comprobación de la normalidad de los errores en la fase de entrenamiento y de validación además de probar la igualdad de las medias de ambas muestras con las pruebas Kolmogorov-Smirnov (KS) o Chi cuadrado (χ^2)

2.5. Interpretación de los pesos.

La relación entre los coeficientes de ajuste (pesos) de cada variable predictora sobre la salida del modelo puede ser probada a través del análisis de sensibilidad empleando procedimientos numéricos, como la obtención de la matriz jacobiana [Urquizo, 2011; Rivals, 2004].

En la matriz Jacobiana *S* –de orden KxI--, cada fila representa una salida de la red y cada columna representa una entrada, de forma que el elemento S_{ki} de la matriz representa la sensibilidad de la salida k respecto a la entrada i. Cada uno de los elementos S_{ki} se obtiene calculando la derivada parcial de una salida y_k respecto a una entrada x_i, esto es: $\frac{\partial y_k}{\partial x_i}$

Así, cuanto mayor sea el valor de S_{ki}, más importante es x_i en relación a y_k. El signo de S_{ki} nos indicará si la relación entre ambas variables es directa o inversa.

2.6. Ejemplo de aplicación.

En MatLab R2010a existen varias formas de utilizar el *toolbox* de redes neuronales artificiales:

- A través de la línea de comandos para crear una red neuronal artificial personalizada.
- A través de un asistente que carga la interfaz gráfica para utilizar las RNA en el agrupamiento de datos con el comando *nctool*
- El uso del asistente para usar las RNA en el reconocimiento de patrones con el comando *nprtool*
- Utilizando la Interfaz gráfica de usuario para ajuste de funciones linealmente separables a partir de RNA de una sola capa que aparece al teclear en la ventana de comandos la orden *nftool*
- Usando la Interfaz gráfica de usuario para la implementación de RNAs de tipo general que aparece al teclear el comando *nntool*

Con el objetivo de poner en práctica las ideas señaladas con anterioridad utilizaremos la primera de las opciones es decir usaremos línea de comandos para editar directamente las propiedades de la red.

Descripción del problema:

Con fines demostrativos procederemos a ajustar un modelo de RNA a los datos de la tabla 1.1 en la que aparecen en las columnas 1 y 2 los valores de las variables independientes y la última columna contiene la variable dependiente. Para desarrollar la estrategia que señalamos anteriormente utilizaremos los 14 primeros valores (70%) de la tabla en la etapa de entrenamiento y ajuste del modelo y en la etapa de validación y generalización emplearemos los restantes.

No.	P		T
	Var1	Var2	Var3
1	107,1	113,5	112,7
2	113,5	112,7	114,7
3	112,7	114,7	123,4
4	114,7	123,4	123,6

5	123,4	123,6	116,3
6	123,6	116,3	118,5
7	116,3	118,5	119,8
8	118,5	119,8	120,3
9	119,8	120,3	127,4
10	120,3	127,4	125,1
11	127,4	125,1	127,6
12	125,1	127,6	129,0
13	127,6	129,0	124,6
14	129,0	124,6	134,1
15	124,6	134,1	146,5
16	134,1	146,5	171,2
17	146,5	171,2	178,6
18	171,2	178,6	172,2
19	178,6	172,2	171,5
20	172,2	171,5	163,6

El ajuste estadístico por regresión lineal arroja el modelo que aparece en la ecuación 2.9 con los estadígrafos que se aparecen a continuación entre los que destaca el índice de determinación (R^2) de 50% a pesar del comportamiento aproximadamente lineal que revela el gráfico de dispersión de las variables que se muestra en la figura 1.3.

$$var3 = 32{,}76 + 0{,}13 * var1 + 0{,}61 * var2 \qquad (2{,}9)$$

R^2= 50,1171 %

R^2 (ajustado) = 39,032 %

Error estándar de los estimados. = 4,16702

Error medio absoluto = 2,73018

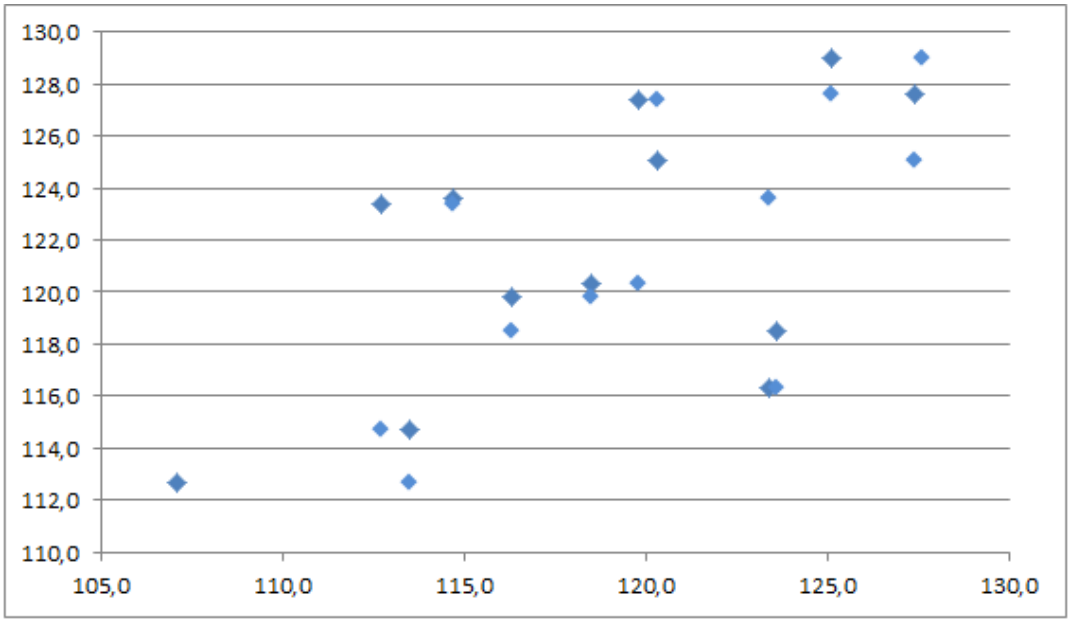

Figura 1.3. Gráfico de dispersión de las variables bajo estudio.

Comenzaremos por reservar un espacio de memoria en la que ubicaremos el objeto tipo red neuronal (neural network), tecleando en la ventana de comandos

net=network;

A continuación debemos introducir los datos, para lo que agruparemos los valores de las variables 1 y 2 en el vector *X* y para la variable 3 utilizaremos el vector *T* como se muestra a continuación (puede utilizar los comandos copiar y pegar si no desea teclear los valores en la ventana de MatLab).

X=[107.1 113.5 112.7 114.7 123.4 123.6 116.3 118.5 119.8 120.3 127.4 125.1 127.6 129.0 124.6 134.1 146.5 171.2 178.6 172.2; 113.5 112.7 114.7 123.4 123.6 116.3 118.5 119.8 120.3 127.4 125.1 127.6 129.0 124.6 134.1 146.5 171.2 178.6 172.2 171.5];

T=[112.7 114.7 123.4 123.6 116.3 118.5 119.8 120.3 127.4 125.1 127.6 129.0 124.6 134.1 146.5 171.2 178.6 172.2 171.5 163.6];

Para decidir sobre la arquitectura de la red notemos que se dispone de 40 valores en la variable *X* (que almacena las dos variables de entrada), y precisamos una salida,

Probaremos entonces con una red perceptrón de 1 entrada y dos capas, con 7 neuronas en la capa oculta y 1 en la capa de salida lo que según la notación establecida anteriormente queda 1:7:1 y que será entrenada con los valores almacenados en la variable *P* hasta aproximar la salida *T*; para crear esta configuración el comando de MatLab es:

net=newff(X,T,7);

Si utilizamos la ecuación 2,7 para verificar la posibilidad de sobre ajuste en la configuración seleccionada, resulta un total de 24 parámetros, valor que no rebasa la cantidad de pesos a estimar (2,10).

$$N_{param} = (1+1)*7 + (7+1)*1 + (1+1) = 24 \quad (2,10)$$

Para definir el número de vectores de entrada tecleamos:

net.numInputs = 1;

La próxima propiedad será el número de capas que faltan las que decidimos establecer una para la capa oculta (con 5 neuronas) y otra para la capa de salida con una neurona.

net.numLayers = 2;

net.layers{1}.size = 7;

net.layers{2}.size = 1;

Conectando las capas

Al realizar la conexión entre capas debemos comenzar por decidir a que capa están conectados los valores de entrada, normalmente esto será con la primera capa y la conexión la realizamos poniendo a 1 la propiedad net.inputConnect(i,j) que representa una conexión en la *i*th capa de la *j*th entrada. Para conectar la primera entrada a la primera capa tecleamos:

net.inputConnect(1,1) = 1;

La conexión entre el resto de las capas quedará definida por la matriz de conectividad llamada **net.layerConnect (i,j)**, la cual toma valores 0 o 1 como elementos de entrada. Si el elemento (i,j) es 1, entonces la salida de la capa j estará conectada a la entrada de la capa *i* así para conectar la salida de la capa 1 a la entrada de la capa 2 ponemos:

net.layerConnect(2, 1) = 1;

Y por último nos queda definir cuál es la capa de salida poniendo la propiedad net.outputConnect(i) a 1 para la capa *i* de salida en nuestro caso la capa 2 y a 0 para el resto de las capas que no se conectan a ella.

net.outputConnect(1) = 0;

net.outputConnect(2) = 1;

Funciones de activación

Cada capa tiene su propia función de transferencia la cual se establece por defecto pero puede ser modificada a través de la propiedad **net.layers{i}.transferFcn**. Asi para establecer en la primera capa la función de activación sigmoidal tangente y en la segunda capa la función lineal (figura 1.2) usamos

net.layers{1}.transferFcn = 'tansig';

net.layers{2}.transferFcn = 'purelin';

En este punto estamos en condiciones de observar la topografía de nuestra red neuronal personalizada la cual debe coincidir con la que aparee en la Figura 1,4 donde se muestra el vector de entrada (Input) la capa oculta, la capa de salida y el vector de salidas (output), para obtener este gráfico teclee en la ventana de comandos

view(net)

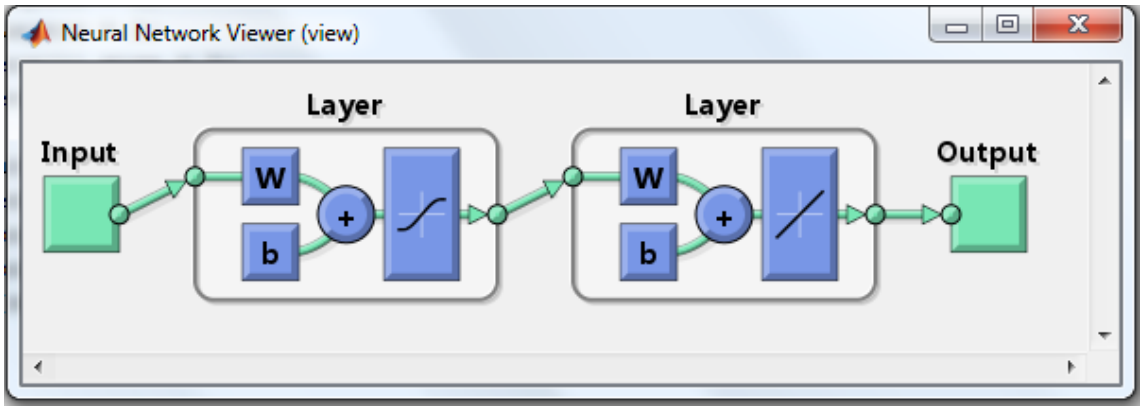

Figura 1,4. Estructura de la red neuronal personalizada.

También podemos verificar las propiedades asociadas a las capas que hemos creado, para ello observe en la respuesta que aparece los parámetros de interés al teclear:

net.layers{1}

ans =

dimensions: 7

name: 'Hidden Layer'

transferFcn: 'tansig'

Igualmente para la capa 2

net.layers{2}

ans =

dimensions: 1

name: 'Output Layer'

transferFcn: 'purelin'

Pesos y biases

Para definir que capa de neuronas tiene biases pondremos **net.biasConnect** a 1, donde **net.biasConnect(i) = 1** significa que la capa *i* tiene biases establecidos.

net.biasConnect(1) = 1;

```
net.biasConnect(2) = 1;
```

También podemos definirlos a través de una sola línea de código:

```
net.biasConnect = [ 1 ; 1];
```

Debemos decidir un procedimiento para inicializar los pesos y biases, a menos que se tenga una rutina personalizada es suficiente establecer la función de inicialización que trae el programa por defecto, para lo que será suficiente teclear:

```
net.initFcn = 'initlay';
```

Pondremos la función de control a mse (cuadrado medio del error) y como función de entrenamiento para acelerar el proceso de convergencia a trainlm (algoritmo Levenberg-Marquardt backpropagation) tecleando:

```
net.performFcn = 'mse';
```

y

```
net.trainFcn = 'trainlm';
```

y por último modificaremos la función dividerand la cual divide los datos de entrenamiento de forma aleatoria desde divideParam con un 70% de los datos para el entrenamiento (*trainRatio*), y un 15% para validación(*valRatio*) y prueba (*testRatio*) además de establecer la función de gráficos plotFcns para observar el estado de convergencia durante el entrenamiento con los comandos que aparecen a continuación:

```
net.divideParam.trainRatio = 70/100;

net.divideParam.valRatio = 15/100;

net.divideParam.testRatio = 15/100;

net.plotFcns = {'plotperform','plottrainstate'};
```

ahora inicializamos la red para que los pesos y biases tomen un valor de inicio, con el código:

```
net = init(net);
```

ahora podemos ver los valores que toman los pesos al inicializar la red en la capa de entrada (pesosEntrada) y de la capa oculta a la capa de salida (pesosCapa) así como los biases de cada neurona en las capas oculta y de salida con los comandos:

```
pesosEntrada=net.IW{1,1}
pesosCapa=net.LW{2,1}
bias_capa_oculta=net.b{1}
bias_capa_salida=net.b{2}
```

Entrenamiento de la red

Antes de entrenar la red debemos normalizar sus valores para que se encuentren en el intervalo de -1 a 1 (recuerde que las funciones de activación se encuentran acotadas) a través de la función *mapminmax* que asume los valores de cada celda como números reales finitos y con al menos uno de ellos diferente de los demás (ecuación 2,10), y en el caso que se repitiesen estos son eliminados del grupo pues no aportan información adicional a la red para su entrenamiento con la orden *removeconstantrows*:

$$y = \frac{(y_{max} - y_{\min}) * (x - x_{min})}{(x_{max} - x_{min}) + y_{min}} \qquad (2.10)$$

Para normalizar empleamos el código:

```
net.inputs{1}.processFcns = {'removeconstantrows','mapminmax'};
```

Establecemos además el número de iteraciones (.epochs) a 50, la tasa de aprendizaje (.lr) a 0.06 y el factor momento (.mc) a 0.9 [epígrafe 2.2] a través de:

```
net.trainParam.epochs=50;

net.trainParam.lr=0.06;

net.trainParam.mc=0.9;
```

Y luego entrenamos la red con:

```
[net,tr] = train(net,X,T);
```

Por ultimo simulamos la red entrenada con los valores de validación y prueba con el comando:

```
Y = sim(net,X);
```

Y recogemos las salidas gráficas al teclear [Hahn, 2007]:

```
plot(Y,'o')

hold

plot(T,'*')

legend('valores estimados','salidas deseadas','Location','NorthWest')
```

Estos commandos muestran graficamente la ubicación de los valores estimados y las salidas deseadas que sean producidas por la red como aparece en la figura 1,5 (***nota:*** sus resultados pueden diferir ligeramente de acuerdo con el valor de los pesos calculados)

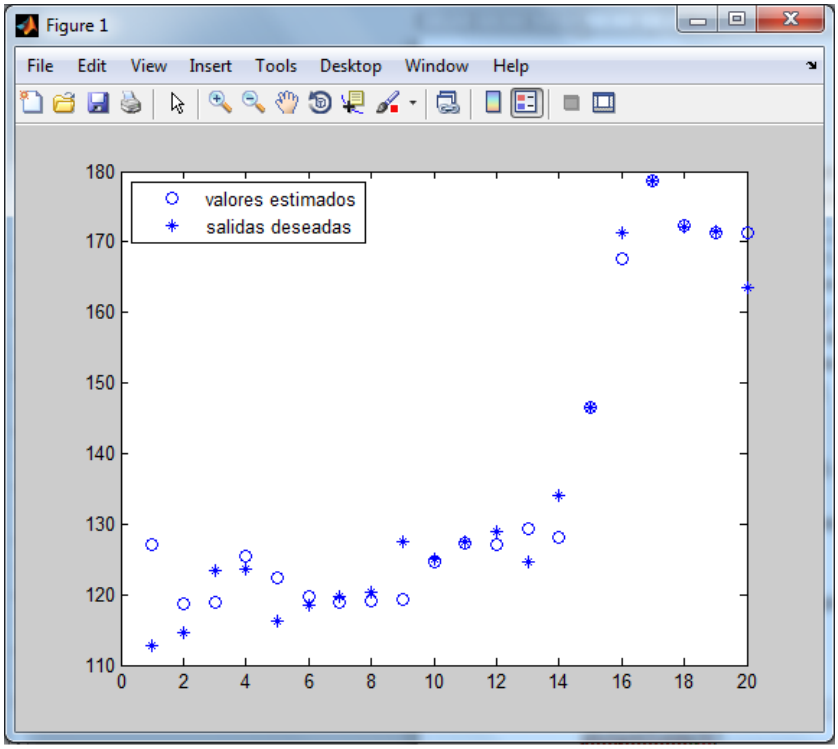

Figura 1,5. Gráfico de los valores estimados y deseados donde se nota el buen acuerdo entre ellos, las desviaciones que se notan se deben al efecto de la aleatoriedad.

plotperform(tr),

Este comando muestra gráficamente el funcionamiento de la red en las etapas de entrenamiento, validación y prueba, así como el mejor resultado durante estas fases como se muestra en la figura 1,6

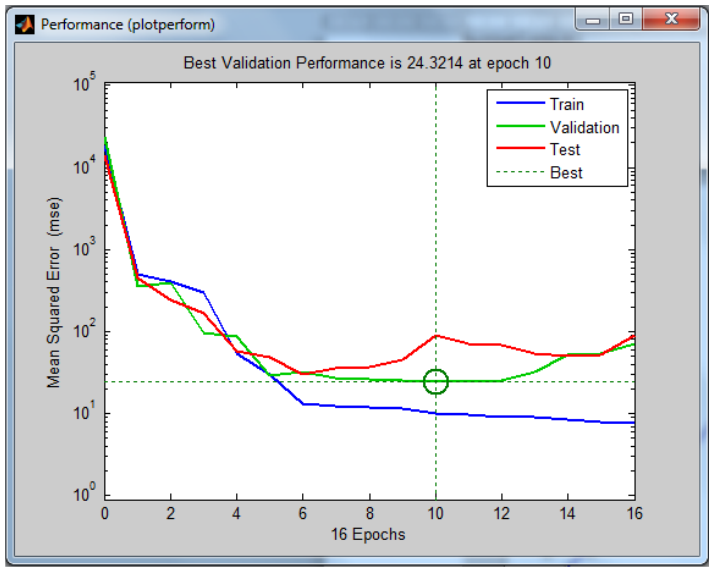

Figura 1,6. Funcionamiento de la red en las etapas de entrenamiento validación y prueba

Y para observar el ajuste entre los valores estimados y predichos tecleamos

plotregression(Y,T)

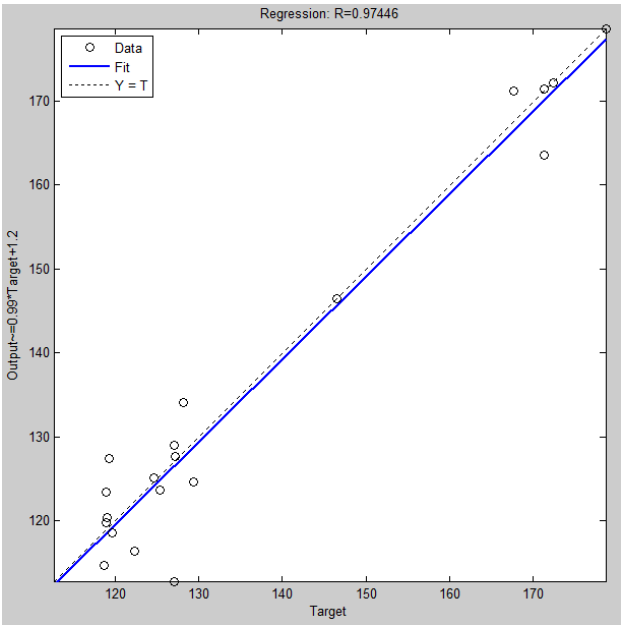

Figura 1,7. Valores estimados vs predichos.

Un valor de utilidad para el análisis del funcionamiento de la red lo constituye el error cuadrado medio el cual puede ser calculado a través de los comandos:

```
d=(Y-T).^2;

mse=mean(d)
```

Como último elemento de interés resulta el valor que toman los pesos de entrada y de capa así como los biases una vez realizado el ajuste de la red que se encuentran almacenados en las variables:

```
pesosEntrada=net.IW{1,1}
pesosCapa=net.LW{2,1}
net.b{1}
net.b{2}
```

Proponemos al lector que luego de realizar este ejemplo modifique la arquitectura de la red, esto es el número de neuronas en la capa oculta (net.layers{1}.size) así como el número de iteraciones (net.trainParam.epochs), los valores de la taza de aprendizaje (net.trainParam.lr) y el coeficiente momento (net.trainParam.mc) para que observe la variación que se refleja en el error cuadrado medio y en los pesos de entrada y de capa.

Conclusiones:

Muchas de las opciones establecidas manualmente en el ejemplo desarrollado son valores que toma la red por defecto como son las funciones de inicialización o de entrenamiento, sin embargo el investigador con frecuencia necesitará modificar estas para alcanzar un mejor desempeño de la RNA es por ello que han sido escogidas para mostrar en el ejercicio.

El uso de las redes neuronales artificiales se ha extendido a numerosos campos de la ciencia y la técnica para pronosticar funciones y en el reconocimiento de patrones entre

otras aplicaciones de interés permitiendo una elevada exactitud con gran rapidez de cálculo haciéndose insustituibles en el caso de comportamientos altamente no lineales. Aun cuando el factor subjetivo constituye un elevado por ciento en su implementación esta novedosa herramienta ha demostrado su utilidad al ser manejada por profesionales competentes.

Por este motivo el acercamiento inicial a las RNAs en MatLab que se propone en este texto intenta principalmente motivar su estudio y familiarizar a todos los interesados con esta herramienta.

Bibliografía:

1. Casacuberta Nolla, F. "Redes Neuronales. Multilayer perceptrón". ETSINF, 2012
2. Cortina Januch, M.G. "Aplicación de técnicas de inteligencia artificial a la predicción de contaminantes atmosféricos". Tesis doctoral, Universidad Politécnica de Madrid, 2012.
3. Hahn, B, D. Valentine, D,T. Essential MatLab for Engineer and Scientists. Third Edition. 2007. Published by Elsevier Ltd. ISBN 13: 9-78-0-75-068417-0
4. Kumar,G.Buchheit, R.G. ."Use of Artifi cial Neural Network Modelsvto Predict Coated Component Life from Short-Term Electrochemical Impedance Spectroscopy Measurements".2008. Corrosión, Vol.64, No.3.ISSN 0010-9312/08P000045. NACE International
5. Marcano Cedeño, Alexis E.,Un modelo neuronal basado en la metaplasticidad para la clasificación de objetos en señales 1-D y 2-D. Tesis Doctoral, Universidad Politécnica de Madrid, 2010.
6. Noriega, L. "Multilayer Perceptron Tutorial". School of Computing, Staffordshire University. Beaconside Staffordshire, ST18 0DG UK. 2005.
7. Palmer, A., Montaño, J.J. y Jiménez, R. Tutorial sobre redes neuronales artificiales: El Perceptrón Multicapa. REVISTA ELECTRÓNICA DE PSICOLOGÍA Vol. 5, No. 2, Julio 2001 ISSN 1137-8492.
8. R.H. Wu, J.T. Liu, H.B. Chang, T.Y. Hsu, X.Y. Ruan, Prediction of the flow stress of 0.4C–1.9Cr–1.5Mn–1.0Ni–0.2Mo steel during hot deformation, J. Mater. Process. Technol. 116 (2001) 211.
9. Rivals, I., Personnaz L. "Jacobian conditioning analysis for model validation". Neural Computation. Vol. 16 No. 2, pp. 401-418. February 2004.
10. Urquizo Rojas, D., Mendivil Gómez, F. "Aplicación de Redes Neuronales Artificiales para el Análisis de la Inflación en Bolivia". 4to. Encuentro de Economistas de Bolivia. p. 20. Julio 2011
11. Velilla, E., Villada F. Echeverría F. "Modelos de pérdida de masa de acero por corrosión atmosférica en Colombia usando inteligencia computacional". Rev. Fac. Ing. Univ. Antioquia No. 49. pp. 81-87. Septiembre, 2009
12. W. Sha. Comment on "Modeling of tribological properties of alumina fiber reinforced zinc–aluminum composites using artificial neural network" by K. Genel et al. [Mater. Sci. Eng. A 363 (2003) 203] , Materials Science and Engineering A 372 (2004) 334–335

Anexos:

Comandos utilizados para la implementación de la RNA en el ejercicio desarrollado:

```
X=[107.1 113.5 112.7 114.7 123.4 123.6 116.3 118.5 119.8 120.3 127.4 125.1 127.6
129.0 124.6 134.1 146.5 171.2 178.6 172.2; 113.5 112.7 114.7 123.4 123.6 116.3 118.5
119.8 120.3 127.4 125.1 127.6 129.0 124.6 134.1 146.5 171.2 178.6 172.2 171.5];
T=[112.7 114.7 123.4 123.6 116.3 118.5 119.8 120.3 127.4 125.1 127.6 129.0 124.6
134.1 146.5 171.2 178.6 172.2 171.5 163.6];

net=newff(X,T,7);

net.numInputs = 1;
net.numLayers = 2;
net.layers{1}.size = 7;
net.layers{2}.size = 1;

net.inputConnect(1,1) = 1;
net.layerConnect(2,1) = 1;
net.outputConnect(1) = 0;
net.outputConnect(2) = 1;

net.layers{1}.transferFcn = 'tansig';
net.layers{2}.transferFcn = 'purelin';

net.biasConnect(1) = 1;
net.biasConnect(2) = 1;

net.initFcn = 'initlay';
net.performFcn = 'mse';
net.trainFcn = 'trainlm';
net.divideParam.trainRatio = 70/100;
net.divideParam.valRatio = 15/100;
net.divideParam.testRatio = 15/100;
net.plotFcns = {'plotperform','plottrainstate'};
net.inputs{1}.processFcns = {'removeconstantrows','mapminmax'};

net = init(net);
pesosEntrada=net.IW{1,1}
pesosCapa=net.LW{2,1}
bias_capa_oculta=net.b{1}
bias_capa_salida=net.b{2}

net.trainParam.epochs=50;
net.trainParam.lr=0.06;
net.trainParam.mc=0.9;
[net,tr] = train(net,X,T);
Y = sim(net,X);
```

```
plot(Y,'o')
hold
plot(T,'*')
legend('valores estimados','salidas deseadas','Location','NorthWest')
plotperform(tr)
plotregression(Y,T)

d=(Y-T).^2;
mse=mean(d)
pesosEntrada=net.IW{1,1}
pesosCapa=net.LW{2,1}
bias_capa_oculta=net.b{1}
bias_capa_salida=net.b{2}
```