Capítulo 3

PERCEPTRON MULTICAPA Y ALGORITMO BACKPROPAGATION

Introducción

Debido a la imposibilidad de solucionar problemas de clasificación no lineales que presenta el Perceptron propuesto por Rosenblatt, y redes algo más evolucionadas como el Adaline, el interés inicial que las redes neuronales artificiales habían despertado, decayó fuertemente y sólo quedaron unos pocos investigadores trabajando en el desarrollo de arquitecturas y algoritmos de aprendizaje capaces de solucionar problemas de alta complejidad. Esta situación fue ampliamente divulgada por Misky y Papert en su libro *Perceptrons*; lo que significó prácticamente el olvido científico de la propuesta de Rosenblatt.

Ronsenblatt, sin embargo, ya intuía la manera de solucionarlo, el autor del *Perceptron*, percibía que si incluía una capa de neuronas entre las capas de entrada y salida, podía garantizar que la red neuronal artificial sería capaz de solucionar problemas de este tipo y mucho más complejos como los que se presentarán a lo largo de este capítulo.

¿Pero cuál fue la mayor dificultad para poner en práctica la nueva propuesta? La modificación de los pesos sinápticos asociados a la capa de salida, se hace con el error entre la salida esperada y la salida de la red. Pero, ¿cómo calcular el error de la capa intermedia para modificar los pesos sinápticos de esta nueva capa, que llamaremos capa oculta? Justamente este interrogante quedó por varios años sin respuesta y fue la razón fundamental por la que las redes neuronales artificiales quedaran casi en el olvido.

A mediados de la década de los setenta, Paul Werbos en su tesis doctoral propone el Algoritmo Backpropagation, que permite entrenar al Perceptron

multicapa y posibilita su aplicación en la solución de una gran variedad de problemas de alta complejidad como lo veremos a lo largo de este capítulo.

Paul J. Werbos
Científico norteamericano que obtuvo su doctorado en la Universidad de Harvard en 1974, reconocido en el mundo de las redes neuronales artificiales, porque en su Tesis fue el primero en describir el algoritmo *Backpropagation* para el entrenamiento del Perceptron multicapa. Una ampliación de esta información puede encontrarse en su libro *The Roots of Backpropagation.* Por este aporte, fue galardonado por la IEEE con el *Neural Network Pioneer Award.* Actualmente, trabaja para la *National Science Foundation*.

Arquitectura general de un perceptron multicapa

En la figura 3.1 presentamos la estructura del Perceptron Multicapa (MLP de sus siglas en inglés *Multi Layer Perceptron*), que a diferencia del Perceptron y del Adaline, posee al menos tres niveles de neuronas, el primero es el de entrada, luego viene un nivel o capa oculta y, finalmente, el nivel o capa de salida. Podemos proponer más de una capa oculta, pero en general no lo recomendamos pues se aumenta fuertemente la complejidad computacional del algoritmo de aprendizaje y para la gran mayoría de las aplicaciones prácticas que propondremos en este libro, es suficiente un MLP con una única capa oculta.

En las redes neuronales artificiales, el término conectividad se refiere a la forma como una neurona de una capa cualquiera está interconectada con las neuronas de la capa previa y la siguiente. Para el MLP, la conectividad es total porque si tomamos una neurona del nivel de entrada, ésta estará conectada con todas las neuronas de la capa oculta siguiente, una neurona de la capa oculta tendrá conexión con todas las neuronas de la capa anterior y de la capa siguiente. Para la capa de salida, sus neuronas estarán conectadas con todas las neuronas de la capa oculta previa, para mayor claridad observemos la figura 3.1. Por lo general, se le implementan unidades de tendencia o umbral con el objetivo de hacer que la superficie de separación no se quede anclada en el origen del espacio *n*-dimensional en donde se esté realizando la clasificación.

La función de activación que utilizan las neuronas de una red MLP suele ser lineal o en la mayoría de los casos sigmoidal.

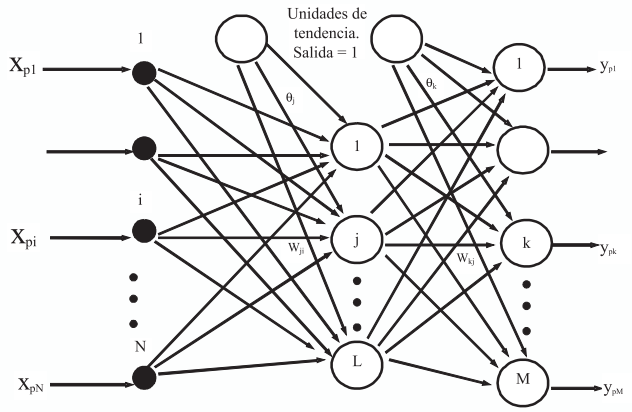
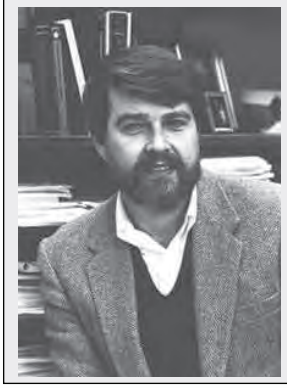


Fig. 3.1 Arquitectura General de un MLP



David Rumelhart

A este científico norteamericano se debe en gran parte, el resurgimiento de las Redes Neuronales Artificiales, cuando en 1986, publica su libro "*Parallel distributed processing: Explorations in the microstructure of cognition"*, en el cual difunde ampliamente el algoritmo de entrenamiento de un *Perceptron* multicapa. Con este algoritmo se da respuesta al interrogante que Papert y Minsky, le plantearon a Rosenblatt, de cómo entrenar la capa oculta del MLP, si no se tenían datos esperados de salida para esta capa.

Entrenamiento de un MLP

Muchos autores traducen al español el nombre de este algoritmo, quizá la mejor acepción para el término Backpropagation es la de *retropropagación,* pues nos da una idea conceptual de la esencia del algoritmo, justamente de propagar el error de la capa de salida hacia las capas ocultas; sin embargo, el nombre anglosajón se ha aceptado en la mayoría de las publicaciones en español y lo usaremos en este libro.

En el capítulo 2, vimos que uno de los principales inconvenientes que tiene el Perceptron es su incapacidad para separar regiones que no son linealmente separables y lo ilustramos al aplicarlo en la solución de un problema simple como la separación de las salidas de una función lógica XOR. Sin embargo, Rosenblatt ya intuía que un Perceptron multicapa sí podía solucionar este problema pues, de esta manera, se podían obtener regiones de clasificación mucho más complejas. Sin embargo, persistían algunas interrogantes sin respuesta ¿Cómo entrenar un Perceptron multicapa? ¿Cómo

evaluar el error en las capas ocultas si no hay un valor deseado conocido para las salidas de estas capas?

Una respuesta a estos interrogantes la planteó formalmente Werbos, cuando formuló el algoritmo de aprendizaje *Backpropagation*. Algoritmo que debe su amplia difusión y uso a David Rumelhart. Como el error de la capa de salida es el único que puede calcularse de forma exacta, el algoritmo propone propagar hacia atrás este error para estimar el error en las salidas de las neuronas de las capas ocultas, con el fin modificar los pesos sinápticos de estas neuronas.

Antes de profundizar matemáticamente en el algoritmo de aprendizaje Backpropagation, revisemos su funcionamiento de manera conceptual:

- Seleccionamos un conjunto de patrones claramente representativos del problema a solucionar, con los cuales vamos a entrenar la red. Esta fase es fundamental pues dependiendo de la calidad de los datos utilizados para el entrenamiento, será la calidad de aprendizaje de la red.
- Aplicamos un vector de entrada a la red y calculamos la salida de las neuronas ocultas, propagamos estos valores hasta calcular la salida final de la red.
- Calculamos el error entre el valor deseado y la salida de la red.
- Propagamos el error hacia atrás, es decir, estimamos el error en la capa oculta con base en el error de la capa de salida.
- Modificamos los pesos de la capa de salida y de las capas ocultas con base en una estimación del cambio de los pesos Δw en cada una de las capas que a su vez, depende del cálculo del error de la capa de salida y de la estimación del error en las capas ocultas realizado en los pasos anteriores.
- Verificamos la condición de parada del algoritmo ya sea por que el error calculado en la salida es inferior al impuesto por el problema bajo análisis o por ejemplo, porque hemos superado un número determinado de iteraciones, en cuyo caso consideraremos que es necesario hacer ajustes al diseño de la red, pues la solución no converge, o simplemente que debemos ampliar el número máximo de iteraciones a realizar
- En caso de no cumplir con ninguna de las condiciones de parada, volvemos a presentarle a la red un patrón de entrenamiento.

Nomenclatura del algoritmo backpropagation

Antes de formular matemáticamente el algoritmo, con la ayuda de la figura 3.1, definamos la notación que seguiremos a lo largo de este capítulo.

$\boldsymbol{x_p}$	Patrón o vector de entrada
x_{pi}	Entrada *i-ésima* del vector de entrada x_p
N	Dimensión del vector de entrada
P	Número de ejemplos, vectores de entrada y salidas diferentes.
L	Número de neuronas de la capa oculta: *h*
M	Número de neuronas de la capa de salida, dimensión del vector de salida
w_{ji}^{h}	Peso de interconexión entre la neurona *i-ésima* de la entrada y la *j-ésima* de la capa oculta.
θ_j^{h}	Término de tendencia de la neurona *j-ésima* de la capa oculta.
$Neta_{pj}^{h}$	Entrada neta de la *j-ésima* neurona de la capa oculta
i_{pj}	Salida de la *j-ésima* neurona de la capa oculta
f_j^{h}	Función de activación de la *j-ésima* unidad oculta
w_{kj}^{o}	Peso de interconexión entre la *j-ésima* neurona de la capa oculta y la *k-ésima* neurona de la capa de salida.
θ_k^{o}	Término de tendencia de la *k-ésima* neurona de la capa de salida.
$Neta_{pk}^{o}$	Entrada neta de la *k-ésima* neurona de la capa de salida.
y_{pk}	Salida de la *k-ésima* unidad de salida
f_k^{o}	Función de activación de la *k-ésima* unidad de salida $o \in \Re^m; x \in \Re^n$
d_{pk}	Valor de salida deseado para la *k-ésima* neurona de la capa de salida.
e_p	Valor del error para el *p-ésimo* patrón de aprendizaje.
α	Taza o velocidad de aprendizaje
δ^{o}_{pk}	Término de error para la *k-ésima* neurona de la capa de salida.
δ^{h}_{pj}	Término de error para la *j-ésima* neurona de la capa oculta *h*
f'^{h}_{j}	Derivada de la función de activación de la *j-ésima* neurona de la capa oculta.
f'^{o}_{k}	Derivada de la función de activación de la *k-ésima* neurona de la capa de salida.

Algoritmo backpropagation: regla delta generalizada

El objetivo buscado con el aprendizaje en las redes neuronales, multicapa, es el de poder establecer una transformación matemática $\Phi(x)$ que relacione adecuadamente los pares ordenados de ejemplos de entradas de excitación a la red y su correspondiente salida deseada. La calidad de la estimación va a depender, fundamentalmente, del número de ejemplos disponibles del problema bajo observación y que la muestra sea lo suficientemente representativa. A continuación vamos a describir las etapas de este algoritmo.

Procesamiento de datos hacia adelante "feedforward"

Como primer paso estimulamos la red neuronal con el vector de entrada:

$$\boldsymbol{x_p}=[x_{p1}, x_{p2},\ldots,x_{pi},\ldots, x_{pN}]^T \tag{3.1}$$

Calculamos la entrada neta de la *j-ésima* neurona de la capa oculta:

$$Neta_{pj}^{\ h} = \sum_{i}^{N} w_{ji}^{\ h} x_{pi} + \theta_j^{\ h} \tag{3.2}$$

Calculamos salida de la neurona *j-ésima* usando la función de activación y la entrada neta:

$$i_{pj}^{\ h} = f_j^h(Neta_{pj}^{\ h}) \tag{3.3}$$

Una vez calculadas las salidas de las neuronas de la capa oculta, éstas se convierten en las señales de excitación de las neuronas de la capa de salida y así podemos calcular la entrada neta de la *k-ésima* neurona de la capa de salida:

$$Neta_{pk}^{\ o} = \sum_{j=1}^{L} w_{kj}^{\ o} i_{pj}^{\ o} + \theta_k^{\ o} \tag{3.4}$$

Con base en la función de activación de la *k-ésima* neurona de la capa de salida podemos calcular la salida estimada por la red neuronal ante el estímulo de entrada:

$$y_{pk} = f_k^{\ o}(Net_{pk}^{\ o}) \tag{3.5}$$

Error en las capas oculta y de salida

Antes de continuar con el cálculo del error en las capas oculta y de salida, recordemos como modifica la Regla Delta o LMS (*Least Mean Square*) los pesos sinápticos de una red neuronal con base en la ecuación 3.6.

$$\begin{aligned} w_i(t+1) &= w_i(t) + 2\alpha e_p x_i \\ e_p &= d - y \end{aligned} \tag{3.6}$$

En la Regla Delta, e_p se define como el error generado por una única neurona o elemento de procesamiento. Con el algoritmo *Backpropagation* calculamos el error global, considerando todas las unidades de procesamiento y sumando el aporte al error de cada una de las neuronas,

$$E_p = \frac{1}{2}\sum_{p=1}^{P}\sum_{k=1}^{M} e_{pk}^2 \tag{3.7}$$

El aporte unitario al error global e_{pk} se define como el error de la *k-ésima* neurona y se calcula con la ecuación 3.8, donde d_{pk} representa la salida deseada.

$$e_{pk} = (d_{pk} - y_{pk}) \tag{3.8}$$

La búsqueda del mínimo en la superficie de error se fundamenta en el cálculo de su ***gradiente descendente*** ∇E_p. Para efectos de la deducción de este algoritmo vamos a considerar que el análisis lo hacemos considerando *p-ésimo* patrón de aprendizaje, por lo que en la ecuación 3.7, eliminamos los términos correspondientes a la sumatoria desde *p=1* hasta *P*.

En un primero paso vamos a calcular la derivada del error global respecto del peso w_{kj}^{o}. Para efectos de este procedimiento, sustituimos en la ecuación 3.7 el valor de e_{pk} que obtuvimos en la ecuación 3.8. y procedemos al cálculo de esta derivada.

$$E_p = \frac{1}{2}\sum_{k=1}^{M}(d_{pk} - y_{pk})^2$$

$$\frac{\partial E_p}{\partial w_{kj}^{o}} = \frac{\partial}{\partial w_{kj}^{o}}\left[\frac{1}{2}\sum_{k=1}^{M}(d_{pk} - y_{pk})^2\right]$$

$$\frac{\partial E_p}{\partial w_{kj}^{o}} = \frac{\partial}{\partial w_{kj}^{o}}\left[\frac{1}{2}\sum_{k=1}^{M}(d_{pk} - f_k^{o}(Neta_{pk}^{o}))^2\right]$$

$$\frac{\partial E_p}{\partial w_{kj}^{o}} = -(d_{pk} - f_k^{o}(Neta_{pk}^{o}))\cdot f_k^{o'}\frac{\partial Neta_{pk}^{o}}{\partial w_{kj}^{o}}$$

Ahora calculemos la derivada de la entrada neta, usando la ecuación 3.4,

$$\frac{\partial Neta_{pk}^{o}}{\partial w_{kj}^{o}} = \frac{\partial}{\partial w_{kj}^{o}}\left[\sum_{j=1}^{L} w_{kj}^{o} i_{pj}^{h} + \theta_k^{o}\right] = i_{pj}^{h}$$

Con base en este resultado, podemos calcular el gradiente que corresponde a la derivada del error global respecto del peso w_{kj}^{o},

$$\frac{\partial E_p}{\partial w_{kj}^{\ o}} = -(d_{pk} - y_{pk}^{\ o}) \cdot f_k^{\ o'}(Neta_{pk}^{\ o}) \cdot i_{pj}^h$$

En el capítulo 2, vimos que la intención de aplicar concepto del gradiente descendente es buscar paulatinamente un punto de error mínimo siempre guiados por la valor negativo de la derivada del error global, por lo que la expresión que utilizaremos corresponde al negativo del gradiente del error $(-\nabla E_p)$.

$$-\nabla E_p = -\frac{\partial E_p}{\partial w_{kj}^{\ o}} = (d_{pk} - y_{pk}^{\ o}) \cdot f_k^{\ o'}(Neta_{pk}^{\ o}) \cdot i_{pj}^h \tag{3.9}$$

El proceso de entrenamiento de la red busca como objetivo fundamental modificar el peso $w_{kj}^{\ o}$, esta modificación se realiza con base en la ecuación 3.10.

$$w_{kj}^{\ o}(t+1) = w_{kj}^{\ o}(t) + \Delta w_{kj}^{\ o}(t) = w_{kj}^{\ o}(t) + \alpha\left(-\nabla E_p\right) \tag{3.10}$$

Sustituimos el valor del gradiente en esta ecuación y obtenemos la ecuación 3.11 para la modificación de los pesos,

$$w_{kj}^{\ o}(t+1) = w_{kj}^{\ o}(t) + \alpha(d_{pk} - y_{pk}^{\ o}) f_k^{\ o'}(Neta_{pk}^{\ o}) i_{pj}^h \tag{3.11}$$

De esta expresión inferimos que la función de activación $f_k^{\ o}$ debe ser derivable, por consiguiente, tanto en las capas ocultas como de salida, generalmente escogemos una función de activación lineal o sigmoidal. En el primer caso, la derivada es uno, ecuación 3.12 y, en el segundo caso, el resultado lo entrega la ecuación 3.13.

$$f_k^{\ o}(Neta_{pk}^{\ o}) = Neta_{pk}^{\ o} \rightarrow f_k^{\ o'} = 1 \tag{3.12}$$

$$f_k^{\ o}(Neta_{pk}^{\ o}) = \frac{1}{1+e^{-Net_{pk}^{\ o}}} \tag{3.13}$$

$$f_k^{\ o'} = f_k^{\ o}(1 - f_k^{\ o})$$

$$f_k^{\ o'} = y_{kp}^{\ o}(1 - y_{pk}^{\ o})$$

Luego de calcular las derivadas dependiendo del tipo de función de activación que escojamos, la ecuación 3.11 se cambia por la 3.14, si la función de activación es lineal,

$$w_{kj}^{\ o}(t+1) = w_{kj}^{\ o}(t) + \alpha(d_{pk} - y_{pk}^{\ o}) i_{pj}^h \tag{3.14}$$

Para el caso de una función de activación sigmoidal, resulta la ecuación 3.15.

$$w_{kj}^{o}(t+1) = w_{kj}^{o}(t) + \alpha(d_{pk} - y_{pk}^{o})y_{kp}^{o}(1 - y_{pk}^{o})i_{pj}^{h} \tag{3.15}$$

Con el fin de simplificar las expresiones 3.14 y 3.15 en una única ecuación para facilitar su representación en el algoritmo, definimos el término del error en las neuronas de la capa de salida, con base en la ecuación 3.16.

$$\begin{aligned} \delta_{pk}^{o} &= (d_{pk} - y_{pk}^{o}) f_{k}^{o'}(Neta_{pk}^{o}) \\ \delta_{pk}^{o} &= e_{pk} f_{k}^{o'}(Neta_{pk}^{o}) \end{aligned} \tag{3.16}$$

Por lo que la ecuación de modificación de pesos la unificamos en la expresión 3.17

$$w_{kj}^{o}(t+1) = w_{kj}^{o}(t) + \alpha\delta_{pk}^{o} i_{pj}^{h} \tag{3.17}$$

Actualización de pesos de las capas ocultas

Para la capa oculta no es posible calcular el error directamente, ya que no se conocen las salidas deseadas de esta capa. Sin embargo, si usamos el concepto de retropropagación, observamos que existe una manera de estimar este error de la capa oculta partiendo del error en la capa de salida. Veamos como podemos actualizar los pesos de la capa oculta y, en particular, el peso w_{ji}^{h}. Retomemos en la ecuación 3.18, el error global en la capa de salida para el patrón p-ésimo.

$$E_p = \frac{1}{2}\sum_{k=1}^{M}(d_{pk} - y_{pk})^2 \tag{3.18}$$

Ahora podemos calcular el gradiente descendente $-\nabla E_p$ con respecto a, w_{ji}^{h}

$$\frac{\partial E_p}{\partial w_{ji}^{h}} = \frac{\partial}{\partial w_{ji}^{h}}\left[\frac{1}{2}\sum_{k=1}^{M}(d_{pk} - y_{pk})^2\right] \tag{3.19}$$

$$\frac{\partial E_p}{\partial w_{ji}^{h}} = -\sum_{k=1}^{M}(d_{pk} - y_{pk}) \cdot \frac{\partial y_{pk}^{h}}{\partial w_{ji}^{h}} \tag{3.20}$$

Apliquemos la regla de la cadena sucesivamente para el cálculo de la derivada interna,

$$\frac{\partial E_p}{\partial w_{ji}^{\ h}} = -\sum_{k=1}^{M}(d_{pk} - y_{pk}) \cdot \frac{\partial y_{pk}^{\ h}}{\partial Neta_{pk}^{\ o}} \cdot \frac{\partial Neta_{pk}^{\ o}}{\partial i_{pj}} \cdot \frac{\partial i_{pj}^{h}}{\partial Neta_{pj}^{\ h}} \cdot \frac{\partial Neta_{pj}^{\ h}}{\partial w_{ji}^{\ h}} \quad (3.21)$$

$$\frac{\partial E_p}{\partial w_{ji}^{\ h}} = -\sum_{k=1}^{M}(d_{pk} - y_{pk}) \cdot f_k^{\ o}(Net_{pk}^{\ o}) \cdot w_{kj}^{\ o} \cdot f_j^{\ h'} \cdot x_{pi} \quad (3.22)$$

En la ecuación 3.22 calculamos el gradiente del error global respecto de los pesos de la capa oculta y con base en este valor, definamos la expresión para la modificación de los pesos de la capa oculta,

$$w_{ji}^{\ h}(t+1) = w_{ji}^{\ h}(t) + \Delta w_{ji}^{\ h}(t) = w_{ji}^{\ h}(t) + \alpha\left(-\frac{\partial E_p}{\partial w_{ji}^{\ h}}\right) \quad (3.23)$$

Al reemplazar en esta ecuación la expresión del gradiente, obtenemos la ecuación 3.24 que nos permite actualizar los pesos de la capa oculta.

$$w_{ji}^{\ h}(t+1) = w_{ji}^{\ h}(t) + \alpha\left(\sum_{k=1}^{M}(d_{pk} - y_{pk}) \cdot f_k^{\ o}(Neta_{pk}^{\ o}) \cdot w_{kj}^{\ o} \cdot f_j^{\ h'} \cdot x_{pi}\right) \quad (3.24)$$

De manera similar a lo planteado en la capa de salida, definamos en la ecuación 3.25 el término del error en la capa oculta,

$$\delta_{pj}^{\ h} = f_j^{\ h'}(Neta_{pj}^{\ h})\sum_{k=1}^{M}\delta_{pk}^{\ o} w_{kj}^{\ o} \quad (3.25)$$

En la ecuación 3.25 observamos que el término de la sumatoria, representa matemáticamente el concepto de *Backpropagation*, ya que el error de la capa oculta está dado en función de los pesos sinápticos y de los términos de error de la capa de salida. Finalmente, la modificación de los pesos sinápticos de la capa oculta la realizaremos con base en la ecuación 3.26.

$$w_{ji}^{\ h}(t+1) = w_{ji}^{\ h}(t) + \alpha\delta_{pj}^{\ h} x_{pi} \quad (3.26)$$

Pasos del algoritmo backpropagation

1. Inicializamos los pesos del MLP.
2. Mientras la condición de parada sea falsa ejecutamos los pasos 3 a 12
3. Aplicamos un vector de entrada $\boldsymbol{x_p} = [x_{p1}, x_{p2}, \ldots, x_{pi}, \ldots, x_{pn}]^T$.
4. Calculamos los valores de las entradas netas para la capa oculta.

$$Neta_{pj}^{\ h} = \sum_{i}^{N} w_{ji}^{\ h} x_{pi} + \theta_j^{\ h}$$

5. Calculamos la salida de la capa oculta.

$$i_{pj}^{\ h} = f_j^h(Neta_{pj}^{\ h})$$

6. Calculamos los valores netos de entrada para la capa de salida.

$$Neta_{pk}^{\ o} = \sum_{j=1}^{L} w_{kj}^{\ o} i_{pj}^{\ o} + \theta_k^{\ o}$$

7. Calculamos las salidas de la red.

$$y_{pk} = f_k^{\ o}(Neta_{pk}^{\ o})$$

8. Calculamos los términos de error para las unidades de salida.

$$\delta_{pk}^{\ o} = (d_{pk} - y_{pk}^{\ o}) f_k^{\ o'}(Neta_{pk}^{\ o})$$

9. Estimamos los términos de error para las unidades ocultas.

$$\delta_{pj}^{\ h} = f_j^{\ h'}(Neta_{pj}^{\ h}) \sum_{k=1}^{M} \delta_{pk}^{\ o} w_{kj}^{\ o}$$

10. Actualizamos los pesos en la capa de salida.

$$w_{kj}^{\ o}(t+1) = w_{kj}^{\ o}(t) + \alpha \delta_{pk}^{o} i_{pj}^{h}$$

11. Actualizamos pesos en la capa oculta.

$$w_{ji}^{\ h}(t+1) = w_{ji}^{\ h}(t) + \alpha \delta_{pj}^{\ h} x_{pi}$$

12. Verificamos si el error global cumple con la condición de finalizar.

$$E_p = \frac{1}{2} \sum_{p=1}^{P} \sum_{k=1}^{M} \left(d_{pk} - y_{pk}\right)^2$$

Algoritmo gradiente descendente con alfa variable

Uno de los inconvenientes que presenta el algoritmo básico de gradiente descendente con *Backpropagation* es tener el parámetro de aprendizaje α fijo. Este parámetro cuyo valor depende de la aplicación que estemos solucionando, también puede variar en el proceso de aprendizaje con el fin de

modificar el tamaño de la variación de los pesos Δw_i, para acelerar la convergencia del algoritmo de aprendizaje. Una estrategia efectiva para variar el parámetro de aprendizaje es el incrementarlo o disminuirlo en cada iteración, dependiendo de la manera como evolucione el error de entrenamiento, con base en la regla descrita en la ecuación 3.27.

$$\alpha_{k+1} = \begin{cases} \rho\alpha_k, si & E_p(w_{k+1}) < E(w_k) \\ \sigma\alpha_k, si & E_p(w_{k+1}) \geq E(w_k) \end{cases} \tag{3.27}$$

Donde, $\rho>1$ y $0<\sigma<1$.

Los parámetros ρ, σ y α_0 son iniciados de manera heurística. Generalmente ρ es definido con un valor cercano a uno (por ejemplo ρ =1.1), para evitar incrementos exagerados en el error de entrenamiento y σ con un valor tal que reduzca α rápidamente para de esta manera abandonar valores elevados de la razón de aprendizaje (por ejemplo σ = 0.5).

Pasos del algoritmo gradiente descendente con α variable

1. Definir los pesos iniciales. ***Condición_parar*** =Falsa
2. Mientras ***Condición_parar*** = Falsa ejecutar los pasos 3 a 12.
3. Aplicamos un vector de entrada
   $\boldsymbol{x_p = [x_{p1}, x_{p2}, ..., x_{pi}, ..., x_{pn}]^T}$.
4. Calculamos los valores de las entradas netas para la capa oculta.
   $$Neta_{pj}^{\ h} = \sum_i^N w_{ji}^{\ h} x_{pi} + \theta_j^{\ h}$$
5. Calculamos la salida de la capa oculta.
   $$i_{pj}^{\ h} = f(Neta_{pj}^{\ h})$$
6. Calculamos los valores netos de entrada para la capa de salida.
   $$Neta_{pk}^{\ o} = \sum_{j=1}^L w_{kj}^{\ o} i_{pj}^{\ o} + \theta_k^{\ o}$$
7. Calculamos las salidas de la red.
   $$y_{pk} = f_k^{\ o}(Neta_{pk}^{\ o})$$
8. Calculamos el error para las unidades de salida.
   $$\delta_{pk}^{\ o} = (d_{pk} - y_{pk}) f_k^{\ o'}(Neta_{pk}^{\ o})$$

9. Estimamos el error para las unidades ocultas.

$$\delta_{pj}{}^{h} = f_j{}^{h'}(Neta_{pj}{}^{h})\sum_{k=1}^{M}\delta_{pk}{}^{o} w_{kj}{}^{o}$$

10. Calculamos el error global de salida.

$$E_p = \frac{1}{2P}\sum_{p=1}^{P}\sum_{k=1}^{M}\left(d_{pk} - y_{pk}\right)^2$$

11. Hacemos $\boldsymbol{E_{prev}} = \boldsymbol{E_p}$
12. Si $\boldsymbol{E_{prev}} < \boldsymbol{E_{min}}$ entonces
    Condición_parar = Verdadera y salir
    Sino
    Condición_parar = Falsa
13. Actualizamos los pesos en la capa de salida.

$$w_{kj}{}^{o}(t+1) = w_{kj}{}^{o}(t) + \alpha\delta_{pk}{}^{o} i_{pj}$$

14. Actualizamos los pesos en la capa oculta.

$$w_{ji}{}^{h}(t+1) = w_{ji}{}^{h}(t) + \alpha\delta_{pj}{}^{h} x_{pi}$$

15. Calculamos el error $\boldsymbol{E_p}$
16. Si $\boldsymbol{E_p} < \boldsymbol{E_{prev}}$ entonces

    $E_{prev} = E_p$

    $\alpha_{k+1} = \rho\alpha_k$, *donde* $\rho > 1$

    Volvemos al paso 3
    sino

    $\alpha_{k+1} = \sigma\alpha_k$, *donde* $0 < \sigma < 1$

    Volvemos al paso 13

ALGORITMOS DE ALTO DESEMPEÑO PARA REDES NEURONALES MLP

El algoritmo *backpropagation* basado en el gradiente descendente resulta ser un método de entrenamiento lento para ciertas aplicaciones donde requerimos una alta velocidad de convergencia. En esta sección presentaremos dos nuevos algoritmos de alto desempeño para el aprendizaje de redes MLP que emplean técnicas de optimización numérica estándar:

- Algoritmo de Aprendizaje del Gradiente Conjugado
- Algoritmo de Aprendizaje de Levenberg Marquardt

Algoritmo de aprendizaje del gradiente conjugado

En esta sección, presentamos una de las más populares y efectivas familias de algoritmos de optimización multivariable para el aprendizaje de redes tipo MLP. El algoritmo del Gradiente Conjugado es un método avanzado de aprendizaje supervisado fuera de línea para la red tipo MLP. Usualmente funciona mejor que el algoritmo *Backpropagation* y lo podemos utilizar en el mismo tipo de aplicaciones. Este algoritmo lo recomendamos para cualquier red neuronal con un gran número de pesos (más de varios centenares) y/o múltiples neuronas de salida.

Lo que buscamos con los algoritmos de aprendizaje para redes MLP es minimizar la función de error o índice de desempeño de la red, considerando una superficie cuadrática *n-dimensional*, de tal manera que al finalizar el aprendizaje se encuentre un vector de pesos cuyos elementos, al reemplazarse en la función de error, entreguen el mínimo deseado. En el método original del gradiente descendente calculamos el gradiente en un punto o vector inicial de pesos y buscamos el mínimo moviéndonos hacia la dirección definida por el negativo del gradiente, entonces, en el nuevo punto que encontramos el gradiente es perpendicular a la dirección de búsqueda inicial o del gradiente anterior. En la mayoría de los casos tal como observamos en la figura 3.2, el nuevo punto que encontramos no es muy diferente al anterior y se pierde un gran esfuerzo computacional minimizando en esta dirección.

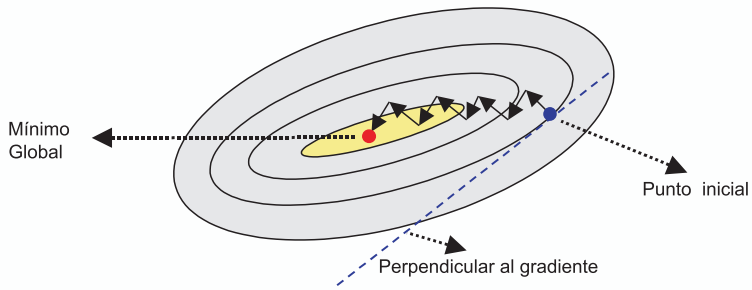


Fig. 3.2 Evolución del Gradiente

Aunque la función decrezca rápidamente alrededor del negativo del gradiente, esto no necesariamente garantiza una mayor velocidad de convergencia. En el algoritmo de gradiente conjugado buscamos minimizar el error sobre una dirección que es conjugada a la que en este momento tenemos, lo cual produce generalmente una convergencia más rápida, que si la búsqueda la hacemos únicamente en la dirección del gradiente descendente. Con el fin de ilustrar este método, consideremos dos vectores ***r*** y **s** que son conjugados. Esto quiere decir que al moverse a lo largo de uno de ellos, por ejemplo por ***r***, el cambio en el gradiente de la función es perpendicular al vector ***s***, que matemáticamente se representa con la ecuación 3.28.

$$\mathbf{r}^{\mathrm{T}}\mathbf{H}\mathbf{s} = 0 \tag{3.28}$$

Donde ***H*** es la Matriz *Hessiana* y se define en la ecuación 3.29

$$\mathbf{H} = \begin{bmatrix} \dfrac{\partial^2 y}{\partial^2 w_1} & & \dfrac{\partial^2 y}{\partial w_1 w_j} \\ \cdot & & \cdot \\ \cdot & \dfrac{\partial^2 y}{\partial w_i w_j} & \cdot \\ \cdot & & \cdot\cdot \\ \dfrac{\partial^2 y}{\partial w_i w_1} & & \dfrac{\partial^2 y}{\partial^2 w_n} \end{bmatrix} \tag{3.29}$$

De la expresión 3.28 deducimos que para obtener un vector conjugado, debemos calcular la Matriz Hessiana, que implica una alta carga computacional, por lo que propondremos un método alternativo que no implique este cálculo. Antes de explicarlo, veamos gráficamente en las curvas de nivel de la función de error de la figura 3.3, la ventaja de minimizar una función de error utilizando esta idea de movernos en una dirección conjugada al gradiente.

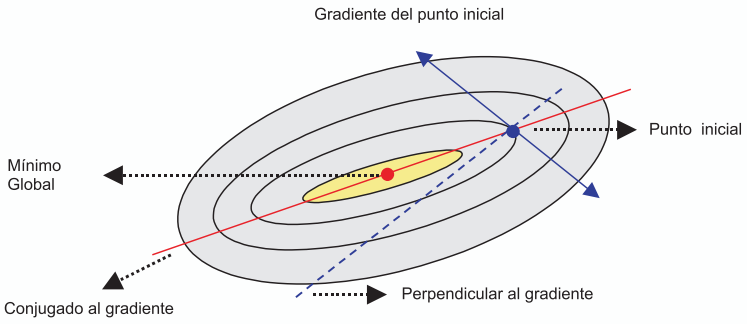


Fig. 3.3 Curvas de nivel de la función de error

Como se muestra en la figura 3.3, si en lugar de calcular una dirección perpendicular a la dirección previa, se calcula una dirección conjugada y se minimiza a lo largo de ésta, se llega más rápido al mínimo global de la función que si seguimos la dirección del gradiente. Por lo que podemos extender esta idea ilustrado en dos dimensiones a un caso de *n* dimensiones, donde para minimizar una función cuadrática de *n* variables (pesos de la red), se requieren ***n*** minimizaciones lineales en direcciones que son mutuamente conjugadas.

Actualización de pesos y determinación de la dirección conjugada

La Matriz *Hessiana* juega un papel importante en el momento de determinar la dirección conjugada. Pero en problemas de optimización complejos como el entrenamiento de una red neuronal para ser aplicada en el reconocimiento de voz o de caracteres manuscritos, es difícil o imposible calcularla. Por esta razón, propondremos un método que permite calcular esta dirección conjugada al gradiente sin requerir el cálculo formal de esta matriz.

Consideremos a $\boldsymbol{g}_i$ como el vector con dirección negativa al gradiente de la función de error en la iteración *i*-ésima. Sea $\boldsymbol{h}_i$ la dirección de búsqueda en esta iteración. Entonces se desarrolla una línea de búsqueda para determinar la distancia óptima para moverse a lo largo de la dirección de búsqueda común, luego el vector de pesos de la red es actualizado de acuerdo con la ecuación 3.30.

En este nuevo algoritmo, el cambio en los pesos lo calculamos con la ecuación 3.30, donde ***k(t)*** es el factor de multiplicación resultado de la búsqueda y ***h(t)*** es el vector de la dirección en la cual vamos a llevar a cabo el proceso de minimización e incluye el aporte de la dirección conjugada del gradiente.

$$\mathbf{w}(t+1) = \mathbf{w}(t) + \mathbf{k}(t)\ \mathbf{h}(t) \tag{3.30}$$

Para inicializar el algoritmo, igualamos $\boldsymbol{g}(0)$ y ***h(0)*** al negativo del gradiente en el punto inicial.

$$\mathbf{g}(0) = \mathbf{h}(0) = -\overline{\nabla} E(0) \tag{3.31}$$

En cada paso debemos considerar la evolución del algoritmo llevando un registro del gradiente y la dirección de búsqueda del paso anterior. Cada vector de dirección de búsqueda lo calculamos como una combinación lineal del vector gradiente en el instante actual y el vector de dirección de búsqueda previa.

$$\mathbf{h}(t) = -\mathbf{g}(t) + \gamma \mathbf{h}(t-1) \tag{3.32}$$

Donde γ es un nuevo parámetro variante en el tiempo y, para su ajuste, existen varias reglas que nos permiten determinar su valor en términos de ***g(t)*** y ***g(t-1)***. Las diferentes versiones que se tienen del algoritmo del gradiente conjugado son distinguidas por la manera como calculamos la constante γ de las cuales presentaremos tres variaciones.

La primera, es la regla de actualización Fletcher-Reeves cuya actualización de γ se hace con la ecuación 3.33, que corresponde a la razón de

cambio de la norma cuadrada del gradiente actual a la norma cuadrada del gradiente previo. Este algoritmo basado en el Gradiente Conjugado requiere sólo un poco más de memoria que otros algoritmos más simples, así que ofrecen una buena opción para redes con un gran número de neuronas, conexiones y por ende pesos sinápticos.

$$\gamma = \frac{\mathbf{g}^T(t)\mathbf{g}(t)}{\mathbf{g}^T(t-1)\mathbf{g}(t-1)} \tag{3.33}$$

Otra forma de calcular la constante γ es la Regla de actualización Polak-Ribiére, que al igual que con el algoritmo Fletcher-Reeves, la dirección de búsqueda en cada iteración es determinada por la ecuación (3.32) y la constante γ es calculada con la ecuación 3.34.

$$\gamma = \frac{\big(\mathbf{g}(t)-\mathbf{g}(t-1)\big)^T\mathbf{g}(t)}{\mathbf{g}^T(t-1)\mathbf{g}(t-1)} \tag{3.34}$$

Este es el producto interno entre el cambio en el gradiente previo y el gradiente actual dividido por la norma cuadrada del gradiente previo. Los requerimientos de memoria para Polak-Ribiére (cuatro vectores) son ligeramente mas grandes que para Fletcher-Reeves (tres vectores).

La última regla para calcular la constante γ la presentamos en la ecuación 3.35. La propusieron Hestenes y Steifel y corresponde al producto interno entre el cambio en el gradiente previo y el gradiente actual, dividido por el producto interno entre el cambio del gradiente previo y la dirección de búsqueda en el instante anterior.

$$\gamma = \frac{\big(\mathbf{g}(t)-\mathbf{g}(t-1)\big)^T\mathbf{g}(t)}{\big(\mathbf{g}(t)-\mathbf{g}(t-1)\big)^T\mathbf{h}(t-1)} \tag{3.35}$$

Una vez calculamos el valor de γ podemos encontrar con la ecuación 3.32 la nueva dirección de búsqueda ***h(t).*** Para calcular el tamaño de la variación de peso en la dirección de búsqueda ***h(t)*** utilizamos una rutina de minimización lineal. En los algoritmos de aprendizaje hasta ahora estudiados, el valor de la variación de peso Δw es proporcional al parámetro de aprendizaje α, el cual es generalmente constante. En el algoritmo de gradiente conjugado no tenemos un parámetro de aprendizaje, sino que el valor de Δw se ajusta en cada iteración usando una rutina de minimización lineal.

Finalmente, el algoritmo del Gradiente Conjugado requiere de algunos cálculos empleados en el Algoritmo Backpropagation para determinar el gradiente de la función de error con respecto a todos los pesos de la red. A diferencia del algoritmo *Backpropagation* que presentamos en la sección

3.2, en este algoritmo la actualización de los pesos se lleva a cabo en un entrenamiento por lotes, es decir, en cada iteración todos lo patrones deben ser presentados a la red para determinar el error cuadrático promedio (MSE) de la ecuación 3.36, cada vez que se requiera evaluar la función de error durante el proceso de aprendizaje.

$$E_p = \frac{1}{2P}\sum_{p=1}^{P}\sum_{k=1}^{M}\left(d_{pk} - y_{pk}\right)^2 \tag{3.36}$$

La actualización de pesos por lotes tiene como ventaja incrementar la rata de aprendizaje de una red MLP, ya que en cada iteración se presentan todos los patrones de entrada antes de realizar la actualización.

Pasos del algoritmo gradiente conjugado

1. Definimos los parámetros iniciales:
   Condición_parar = Falsa
   Valores iniciales de los pesos y bias de la red $\boldsymbol{w}$.
   Factor de aprendizaje α.
   Error mínimo deseado $\boldsymbol{E_{min}}$
2. Mientras la *Condición_parar* sea falsa ejecutamos los pasos 3 a 7
3. Presentamos todos los patrones de entrada a la red
   $x_p = \{x_1, x_2, ..., x_i, ..., x_n\}$.
4. Calculamos la entrada neta para las neuronas de la capa oculta.

   $$Neta_{pj}^{h} = \sum_{i=1}^{N} w_{ji}^{h} x_{pi} + \theta_j^{h}$$

5. Calculamos la salida de la capa oculta.

   $$i_{pj}^{h} = f(Neta_{pj}^{h})$$

6. Calculamos la entrada neta para las neuronas de la capa de salida.

   $$Neta_{pk}^{o} = \sum_{j=1}^{L} w_{kj}^{o} i_{pj}^{o} + \theta_k^{o}$$

7. Calculamos la salida de la red neuronal.

   $$y_{pk} = f_k^{o}(Neta_{pk}^{o})$$

8. Calculamos el gradiente para las capas de salida y oculta en la primera iteración; en las siguientes usamos la calculada en el paso 17.

9. Inicializamos el gradiente y la dirección de búsqueda para la primera iteración, para las dos capas, en las siguientes iteraciones se omite este paso

$$\mathbf{h}(0) = -\mathbf{g}(0)$$

10. Calculamos el error global promedio de la red.

$$E_p = \frac{1}{2P}\sum_{p=1}^{P}\sum_{k=1}^{M}(d_{pk} - y_{pk})^2$$

11. Si $E_p > E_{min}$ ejecutamos los pasos 12 a 18.
En caso contrario *Condición_parar* = *Verdadera* y salir.
12. Calculamos el **k** *(t)* próximo usando una rutina de minimización lineal *mlin*

$$\mathbf{k}(t) = mlin\left(\mathbf{w}(t), \mathbf{g}(t), E_p\right)$$

13. Actualizamos los pesos de la red para cada una de las capas.

$$\mathbf{w}(t+1) = \mathbf{w}(t) + \mathbf{k}(t)\,\mathbf{h}(t)$$

14. Calculamos la regla de actualización por cualquiera de las variantes.

Flecher -Reeves $\quad \gamma = \dfrac{\mathbf{g}^T(t)\mathbf{g}(t)}{\mathbf{g}^T(t-1)\mathbf{g}(t-1)}$

Polak -Ribiére $\quad \gamma = \dfrac{\left(\mathbf{g}(t) - \mathbf{g}(t-1)\right)^T \mathbf{g}(t)}{\mathbf{g}^T(t-1)\mathbf{g}(t-1)}$

Hestenes -Steifel $\quad \gamma = \dfrac{\left(\mathbf{g}(t) - \mathbf{g}(t-1)\right)^T \mathbf{g}(t)}{\left(\mathbf{g}(t) - \mathbf{g}(t-1)\right)^T \mathbf{h}(t-1)}$

15. Calculamos la siguiente dirección de búsqueda para cada capa.

$$\mathbf{h}(t) = -\mathbf{g}(t) + \gamma \mathbf{h}(t-1)$$

16. Normalizamos la dirección de búsqueda $nor\_h(t) = \dfrac{\mathbf{h}(t)}{\|\mathbf{h}(t)\|}$

17. Volvemos al paso 2

Algoritmo de aprendizaje levenberg marquardt

En esta sección presentamos una alternativa al método de Gradiente Conjugado para acelerar la fase de aprendizaje de una red neuronal artificial, en honor a sus inventores se ha denominado: Algoritmo Levenberg

Marquardt. Debido a la gran carga computacional que implica su ejecución, no recomendamos el uso de este algoritmo cuando se vaya a entrenar una red con alto número de conexiones. Para redes con una arquitectura con pocas neuronas y conexiones, este algoritmo será más rápido y efectivo que el Algoritmo del Gradiente Conjugado.

Una de las grandes fortalezas que presenta este algoritmo es la combinación de dos estrategias de minimización, el método de gradiente descendente y el método de Newton. Vamos a revisar, rápidamente, este segundo método y en la ecuación recursiva 3.37 presentamos como se localiza un valor mínimo de una función de una variable *f(x)*, utilizando la primera y segunda derivada.

$$x_{\min}(t+1) = x_{\min}(t) - \frac{f'(x_{\min}(t))}{f''(x_{\min}(t))} \tag{3.37}$$

Con base en esta ecuación podemos inferir la ecuación 3.38, donde vamos a minimizar el error global $\boldsymbol{E}_p$ en el espacio de los pesos sinápticos representado por la matriz $\boldsymbol{W}$.

$$W_{\min}(t+1) = W_{\min}(t) - \frac{E_p'}{E_p''} \tag{3.38}$$

La segunda derivada del error global corresponde a la Matriz Hessiana $\boldsymbol{H}$ y la primera derivada la conocemos como el vector gradiente $\boldsymbol{g}$. El vector gradiente y la matriz Hessiana de la función de error los podemos calcular utilizando la regla de la cadena. Así, para una neurona de la capa de salida el vector gradiente está compuesto por las derivadas parciales del error con respecto a cada uno de los pesos $\boldsymbol{w}_i$ de la red, el elemento ***(i,j)*** de la matriz Hessiana lo calculamos con las segundas derivadas parciales del error con respecto a los pesos $\boldsymbol{w}_i$ y $\boldsymbol{w}_j$.

Recordemos la expresión para calcular el error para una neurona de salida de Perceptron multicapa,

$$e = (d - y)^2 \tag{3.39}$$

La aplicación directa de la regla de la cadena sobre esta expresión genera el siguiente resultado:

$$\frac{\partial e_p}{\partial w_{kj}^{\,o}} = -2(d_{pk} - y_{pk})\frac{\partial y_{pk}}{\partial w_{kj}^{\,o}} \tag{3.40}$$

$$\frac{\partial^2 e_p}{\partial w_{kj}^{\ o} \partial w_{ji}^{\ h}} = 2\left[\frac{\partial y_{pk}}{\partial w_{kj}^{\ o}} \frac{\partial y_{pk}}{\partial w_{ji}^{\ h}} - (d_{pk} - y_{pk}) \frac{\partial^2 y_{pk}}{\partial w_{kj}^{\ o} \partial w_{ji}^{\ h}}\right] \tag{3.41}$$

Las ecuaciones 3.40 y 3.41, nos permiten calcular las componentes del vector gradiente y de la matriz Hessiana.

Los factores que componen la expresión para la segunda derivada parcial del error, es decir, la derivada parcial de la salida con respecto a los pesos, son producidos en el algoritmo gradiente descendente con *Backpropagation.*

En la ecuación 3.41 es posible eliminar el segundo término, considerando que éste se encuentra multiplicado por el error. Es razonable asumir que los errores generados en cada iteración del entrenamiento son completamente independientes y distribuidos alrededor de cero. Algunas veces el error será positivo y otras negativo. Esta situación mantiene un balance que permite cancelar el segundo término de la ecuación 3.41. Estos errores producidos por la aproximación de la matriz Hessiana no afectarán la precisión de los resultados. Su efecto se reflejará en un proceso de convergencia más lento, haciendo más confiable la tendencia hacia el antiguo método de Gradiente Descendente.

La obtención del gradiente y la matriz Hessiana, después de realizar la aproximación para esta última, requieren sólo del cálculo de la derivada de la salida de cada neurona de la última capa con respecto a cada uno de los pesos. De acuerdo con la teoría aplicada al método de aprendizaje gradiente descendente con *Backpropagation*, es posible calcular las expresiones para la derivada teniendo en cuenta el tipo de pesos:

$$\frac{\partial y_{pk}}{\partial w_{kj}^{\ o}} = f_k^{\ o'}(Neta_{pk}^{\ o}) \cdot i_{pj}^{h} \tag{3.42}$$

$$\frac{\partial y_{pk}}{\partial w_{ji}^{\ h}} = f_j^{\ h'}(Neta_{pj}^{\ h}) w_{ji}^{\ h} \cdot x_{pi} f_k^{\ o'}(Neta_{pk}^{\ o}) \tag{3.43}$$

Asumiendo una red de dos capas de procesamiento, la primera ecuación determina la derivada de la salida con respecto a uno de los pesos que conectan la capa oculta con la capa de salida, y la segunda representa la derivada de la salida con respecto a uno de los pesos que conectan las neuronas de entrada con la capa oculta, ***f*** representa la función de activación, ***i*** corresponde a la salida de la capa oculta.

La matriz ***H*** no la calculamos exactamente sino que recurrimos a su estimación, por lo que debemos establecer un mecanismo de control para garantizar la convergencia del algoritmo. En primer lugar se prueba la ecuación

del método de Newton, si al evaluarla el algoritmo no converge sino que el valor del error comienza a crecer, eliminamos este valor e incrementamos el valor de λ en la ecuación 3.44, con el fin de minimizar el efecto de la matriz **H** en la actualización de los pesos. Si λ es muy grande, el efecto de la matriz **H** prácticamente desaparece y la actualización de pesos se hace esencialmente con el algoritmo de gradiente descendente. Si el algoritmo tiene una clara tendencia hacia la convergencia disminuimos el valor de λ con el fin de aumentar el efecto de la matriz **H** y de esta manera garantizamos que el algoritmo se comporta con un predominio del Método de Newton.

El método Levenberg Marquardt mezcla, suavemente, el método de Newton y el método Gradiente Descendente en una única ecuación para estimar ***W(t+1).***

$$\mathbf{W}(t+1) = \mathbf{W}(t) - (\mathbf{H} + \lambda \mathbf{I})^{-1}\mathbf{G} \tag{3.44}$$

Selección del parámetro λ

Existe un último aspecto a tratar antes de mostrar los pasos del algoritmo Levenberg Marquardt, se trata de la selección del valor del parámetro λ**.** Algunos autores recomiendan escoger un valor inicial fijo tal como λ=0.001; sin embargo, tal valor puede resultar lo suficientemente grande para algunas aplicaciones, y muy pequeño para otras. Un método para optimizar este parámetro es buscar en la diagonal principal de la matriz ***H*** el valor más grande y tomarlo como el valor inicial de λ**.**

El parámetro λ debe reducirse, si existe una tendencia a bajar el error producido por los pesos estimados en la ecuación 3.44, en caso contrario debemos incrementar el valor de λ**.** En el algoritmo utilizaremos ρ como factor de reducción y σ como factor de incremento.

Solución de la ecuación para determinar la variación de pesos Δ

De acuerdo con la ecuación 3.44, la variación de pesos Δ la definimos con la ecuación 3.45.

$$\mathbf{W}(t+1) = \mathbf{W}(t) - \Delta \tag{3.45}$$

El cálculo directo de Δ empleando está ecuación es lento y puede resultar inestable debido al cálculo de la inversa de la matriz ***H.*** Para solucionar estos inconveniente proponemos calcular Δ con la ecuación 3.46.

$$(\mathbf{H} + \lambda \mathbf{I})\Delta = \mathbf{G} \tag{3.46}$$

Bajo condiciones ideales estas dos ecuaciones son matemáticamente equivalentes, y uno de los métodos de álgebra lineal más indicados para

resolver el sistema es conocido como *Backsustitution*. La ventaja del método es que asegura que la matriz de coeficientes será no singular previendo siempre una solución aceptable.

Pasos del algoritmo levenberg marquardt

En el algoritmo Levenberg Marquardt actualizamos el vector de pesos de la red MLP en un aprendizaje por lotes, al igual que en el Algoritmo Gradiente Conjugado. La matriz Hessiana y el vector gradiente también los calculamos para cada patrón y promediados por el número total de éstos, como resultado de la realización de un aprendizaje por lotes.

1. Inicializamos los parámetros de la red:
   Definimos los valores iniciales de los pesos y *bias* de la red
   Definimos el valor inicial del parámetro de aprendizaje α
   Definimos el error mínimo deseado $\boldsymbol{E}_{min}$
   Definimos la ***Condición_parar*** = *Falsa* y *RESET* = *Falso*
2. Si *RESET* = *Falso* ejecutamos los pasos 3 a 13, sino vamos al paso 14.
3. Presentamos todos los patrones de entrenamiento a la red.
   $x_p = \{x_1, x_2, \ldots, x_i, \ldots, x_p\}$
4. Calculamos los valores de las entradas netas para la capa oculta.

$$Neta_{pj}^{h} = \sum_{i=1}^{N} w_{ji}^{h} x_{pi} + \theta_{j}^{h}$$

5. Calculamos la salida de la capa oculta.

$$i_{pj}^{h} = f(Neta_{pj}^{h})$$

6. Calculamos los valores netos de entrada para la capa de salida.

$$Neta_{pk}^{o} = \sum_{j=1}^{l} w_{kj}^{o} i_{pj}^{o} + \theta_{k}^{o}$$

7. Calculamos la salida de la red neuronal.

$$y_{pk} = f_{k}^{o}(Neta_{pk}^{o})$$

8. Calculamos el error global de la red y lo almacenamos en la variable.
   $\boldsymbol{E}_{prev}$ (Error previo)

$$Eprev = E = \frac{1}{2P} \sum_{p=1}^{P} \sum_{k=1}^{M} (d_{pk} - y_{pk})^2$$

9. Calculamos el gradiente para el conjunto de patrones.

$$\mathbf{G}_{save} = -\nabla E(0)$$

10. Calculamos la matriz ***H***.

$$\mathbf{H}_{save} = \mathbf{H}$$

11. Si $\boldsymbol{E}_{prev} > \boldsymbol{E}_{min}$ (el error previo es mayor al error mínimo deseado), ejecutamos los pasos 12 a 18, sino hacemos ***Condición_parar*** = Verdadera y salimos.
12. Inicializamos el parámetro λ.

$$\lambda = \mathbf{H}_{ijmax}$$

$$i = j$$

13. Inicializamos indicador RESET a verdadero y volver a paso 2.
14. Si RESET = verdadero, guardamos el vector gradiente y la matriz ***H***.

$$\mathbf{H} = \mathbf{H}_{save}$$

$$\mathbf{G} = \mathbf{G}_{save}$$

15. Incrementamos cada elemento de la diagonal de **H** en λ.

$$\mathbf{H} = \mathbf{H} + \lambda\mathbf{I}$$

16. Resolvemos el sistema de ecuaciones para Δ.

$$(\mathbf{H} + \lambda\mathbf{I})\Delta = \mathbf{G}$$

17. Adicionamos Δ al actual vector de pesos para actualización.

$$\mathbf{W}(t+1) = \mathbf{W}(t) - \Delta$$

18. Calculamos el error ***E***, el gradiente ***G*** y la matriz ***H*** con el nuevo vector de pesos.

$$E = \frac{1}{2P}\sum_{p=1}^{P}\sum_{k=1}^{M}(d_{pk} - y_{pk})^2$$

19. Si $\boldsymbol{E}_{prev} <= \boldsymbol{E}_{min}$, hacer ***Condición _ parar*** = Verdadero, en caso contrario ***Condición _ parar*** = Falsa

20. Si $\boldsymbol{E} < \boldsymbol{E}_{prev}$ hacemos $\boldsymbol{E}_{prev} = \boldsymbol{E}$, en caso contrario hacemos los siguientes ajustes y volvemos al paso 14:
    RESET = verdadero

$$\lambda = \lambda * \rho \quad \text{donde } \rho<1$$

21. Si $\boldsymbol{E} > \boldsymbol{E}_{prev}$ hacemos los siguientes ajustes y volvemos al paso 14.
RESET = falso
$\lambda = \lambda * \sigma$
$\mathbf{H}_{save} = \mathbf{H}$ donde $\sigma > 1$
$\mathbf{G}_{save} = \mathbf{G}$

Consideraciones de diseño

Antes de empezar a utilizar las redes MLP en algunas aplicaciones, muy seguramente nos surgen algunas preguntas: ¿Cómo entrenamos una red? ¿Cómo seleccionamos su arquitectura? ¿Cuántas neuronas definimos en cada capa? ¿Cómo saber que la red aprendió nuestro problema? ¿Cómo garantizar que la solución es satisfactoria? La respuesta a estas preguntas es difícil de dar y como todo en Ingeniería, a lo largo de la experimentación iremos ganando criterios para diseñar correctamente una red neuronal artificial para la solución de un problema específico. De todas maneras, en este apartado revisaremos cada una de estas preguntas y trataremos de dar una respuesta satisfactoria o al menos algunas recomendaciones o sugerencias.

Conjuntos de aprendizaje y de validación

El conjunto de datos que disponemos para describir el fenómeno o problema a solucionar debe ser representativo de toda la información, es decir, debemos evitar que tenga algún tipo de tendencia. En principio, destacamos dos grupos, el primero, que llamamos de entrenamiento y lo utilizaremos para esta primera fase de ejecución del algoritmo de aprendizaje, se suele utilizar entre el 50% y 70% de los datos. En el segundo grupo tenemos los datos que utilizaremos para verificar el grado de aprendizaje. Recomendamos que sea totalmente diferente a los datos utilizados en la fase de entrenamiento. Se suele utilizar entre el 30% y 50% de los datos del conjunto total disponible. Cuando estudiemos los mecanismos para mejorar la capacidad de generalización del aprendizaje de las redes neuronales artificiales, introduciremos un nuevo conjunto denominado datos para validación, el cual será explicado ampliamente.

En la figura 3.4 presentamos un ejemplo de datos de entrenamiento y validación para una aplicación donde la red neuronal aprenderá la dinámica de una función $y = f(t)$. En éste a partir de un conjunto de muestras tomamos las parejas ordenadas (t_p, y_p). Para el entrenamiento tomamos los puntos en azul para el entrenamiento y los puntos en rojo para el proceso de validación del aprendizaje.

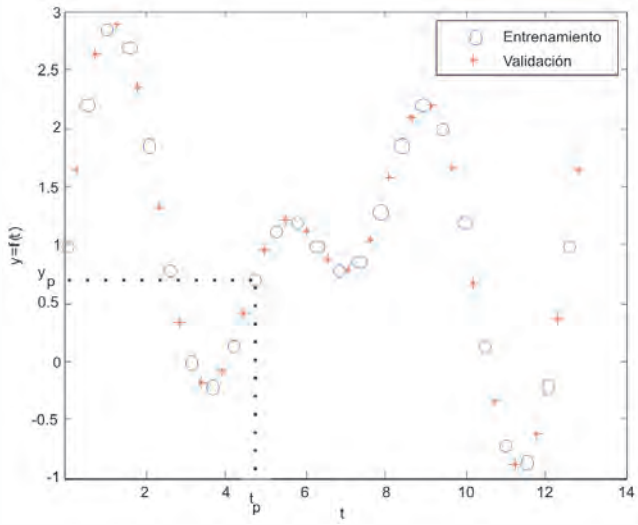


Fig. 3.4. Conjunto de datos de entrenamiento y validación

Dimensión de la red neuronal

Cuando nos enfrentamos en el diseño de la arquitectura de un MLP ante la pregunta de ¿Cuántas neuronas elegir?, la respuesta para las capas de entrada y salida es sencilla ya que el número de neuronas en dichas capas lo definimos con base en el problema a solucionar, pues debe coincidir con la dimensión de los vectores de entrada y salida que lo describen. Por ejemplo, queremos diseñar un clasificador de caracteres numéricos (0 al 9) en código de siete segmentos y salida en código BCD. ¿Cuántas neuronas escoger en la entrada? Para todos es claro que debemos disponer de siete neuronas para saber si se activa o no cada uno de los siete segmentos con los que representamos los dígitos decimales. Para la salida el problema claramente nos define que debemos escoger cuatro neuronas.

La dificultad se centra en la capa oculta, ya que la definición del número de neuronas queda supeditada a la experticia del diseñador. Existen dos tendencias, empezar desde un muy alto número de neuronas y evaluar su funcionamiento y, luego, paulatinamente, ir decreciendo el número de neuronas hasta el instante en el cual, el error de aprendizaje y verificación se mantengan en el mínimo requerido. La otra es empezar por un número pequeño de neuronas en la capa oculta, por ejemplo, igual al 50% de la dimensión del vector de entrada y, depués, empezar a crecer el tamaño de esta capa oculta verificando si su desempeño mejora; llegará un momento en el cual, aunque incrementemos el número de neuronas en la capa oculta, los errores no van a decrecer sustancialmente, consideramos que en este instante hemos llegado a un buen diseño de red. Si este desempeño no es satisfactorio debemos revisar la arquitectura de la red, el algoritmo de aprendizaje o la

calidad de los datos.

Finalmente, surge la pregunta ¿Cuántas capas ocultas debe tener un MLP?, para una gran cantidad de aplicaciones de alta complejidad, tres capas son suficientes, es decir recomendamos utilizar solo una capa oculta. El utilizar más de una capa oculta, aumenta drásticamente la carga computacional de la red, sobretodo en los algoritmos de entrenamiento de segundo orden. En general recomendamos aumentar el tamaño de la capa oculta y no incrementar el número de capas ocultas, por ejemplo, si tenemos una red neuronal con diez neuronas en la capa oculta, computacionalmente es mejor, incrementar a 20 neuronas en esta capa y no generar una nueva capa oculta con diez neuronas más.

Por supuesto, que hay aplicaciones de alta complejidad, como por ejemplo el reconocimiento y clasificación de patrones (reconocimiento y clasificación de imágenes o caracteres manuscritos), en las cuales es muy importante generar superficies de decisión altamente complejas y una segunda capa oculta es muy útil en este caso.

Velocidad de convergencia del algoritmo

El Algoritmo *Backpropagation* encuentra un valor mínimo de error (local o global) navegando con el gradiente descendente por la superficie de error. La velocidad de convergencia se controla con el valor de parámetro **α**, normalmente dicho parámetro debe ser un número pequeño, en el rango de 0.05 a 0.25. Un valor de α muy pequeño trae como consecuencia un aprendizaje seguro, pero puede representar un alto número de iteraciones y tardar mucho el tiempo de ejecución de este algoritmo. En contraposición, un valor de α muy alto, puede generar oscilaciones en el aprendizaje.

Una forma de acelerar la convergencia en el proceso de aprendizaje y tal como lo muestran las ecuaciones 3.47 y 3.48, además de utilizar el valor de los pesos del instante *t*, utilizamos una fracción del valor de los pesos del instante anterior *t-1*, que la controlaremos con el valor del parámetro β. Con esto queremos emular al *momentum* físico considerando la tendencia en el aprendizaje que traía la red en el instante anterior. El valor de β es recomendable tomarlo como positivo y menor que la unidad.

Para los pesos de la capa de salida, se tiene la siguiente expresión:

$$w_{kj}^{o}(t+1) = w_{kj}^{o}(t) + \alpha\delta_{pk}{}^{o} i_{pj} + \beta\left(w_{kj}^{o}(t) - w_{kj}^{o}(t-1)\right) \quad (3.47)$$

Para los pesos de la capa oculta se tiene la siguiente expresión:

$$w_{ji}^{h}(t+1) = w_{ji}^{h}(t) + \alpha\delta_{pj}{}^{h} x_{pi} + \beta\left(w_{ji}^{h}(t) - w_{ji}^{h}(t-1)\right) \quad (3.48)$$

Funciones de activación

En la mayor cantidad de problemas resueltos con redes MLP, la función de activación a utilizar en las capas ocultas es del tipo sigmoidal, pues le garantiza la capacidad de procesamiento no-lineal. Para la capa de salida, vamos a depender del tipo de problema que estemos solucionando; por ejemplo, en el modelado e identificación de sistemas, en la capa de salida es preferible utilizar una función de activación lineal, con el fin de no saturar la salida de la red, hecho que se presentaría si usamos una función de activación sigmoidal.

Si de lo que se trata es de solucionar problemas de clasificación y reconocimiento de patrones, lo recomendable es utilizar una función de activación sigmoidal en su salida con el fin de garantizar una salida selectiva, similar a una salida binaria.

Pre y pos-procesamiento de datos

Consideremos un problema donde nos proponen diseñar un sistema para estimar el valor del dólar en nuestro país, que naturalmente debe ser solucionado aplicando redes neuronales artificiales. La primera pregunta que nos surge es: ¿Qué parámetros de entrada debemos considerar como datos para entrenar esta red?

Lo usual es que tomemos valores anteriores del valor del dólar al día que vamos a realizar la estimación. Por ejemplo, podríamos tomar los valores correspondientes a dos o tres días anteriores. Pero esto normalmente es insuficiente y la experiencia nos ha mostrado que debemos considerar otros parámetros, entre los que destacamos:

- *Tasa de interés*, es el porcentaje de utilidad que normalmente el sector financiero concede a los diferentes tipos de depósitos que hacen los usuarios con sus ahorros.
- *Masa de dinero disponible*, es el dinero en efectivo disponible en el país para realizar transacciones financieras.

Observemos las magnitudes que tienen los valores que estamos manejando aplicándolas para el caso colombiano. Los valores anteriores del dólar están dados en miles de pesos, por ejemplo $2.400, la tasa de interés actualmente es del orden del 7% (0.07) y la masa de dinero es del orden de miles de millones de pesos.

Si observamos la entrada a la capa oculta, una neurona tendría que sumar estos valores ponderados por los pesos sinápticos. Claramente se ve que el parámetro Masa de Dinero anularía los efectos de los valores anteriores del dólar y estos a su vez anularían el valor de la Tasa de Interés, para evitar esta situación es fundamental normalizar los datos de entrada.

Con esta técnica de pre-procesamiento buscamos que tanto los datos de

entrada como de salida queden en el intervalo [-1, +1]. La expresión 3.49 normaliza un valor *x,* a partir del valor máximo del conjunto de datos x_{max} y del valor mínimo x_{min}.

$$x_n = 2\left(\frac{x - x_{\min}}{x_{\max} - x_{\min}}\right) - 1 \tag{3.49}$$

Otra forma de normalizar los datos es utilizando la técnica de Normalización Gaussiana que viene definida por la ecuación 3.50, donde se busca que los datos de entrada y salida tengan una desviación estándar igual a 1 y el valor medio igual a 0.

$$x_n = \left(\frac{x - \mu}{\sigma}\right) \tag{3.50}$$

donde,

μ : Valor medio de los datos

σ : Desviación estándar de los datos

Luego de entrenar la RNA con los datos normalizado y siguiendo con el ejemplo de la predicción del valor del dólar, la salida de la red tal vez no es fácil de interpretar pues sus valores están en el rango de –1 a +1. Necesariamente debemos volver a los valores originales para cuando la red esté interactuando con el usuario, lo cual lo podemos hacer con la siguiente expresión, donde *x* corresponde al valor en la escala original y x_n es el valor normalizado.

$$x = 0.5(x_n + 1)(x_{\max} - x_{\min}) + x_{\min} \tag{3.51}$$

Regularización

En algunas aplicaciones veremos que las redes neuronales artificiales pueden caer en un problema que se conoce como sobre-entrenamiento, en donde la red no es capaz de responder adecuadamente ante datos de entrada diferentes a los datos que se utilizaron para el proceso de aprendizaje, pero que hacen parte del problema que se quiere solucionar. Visto de otra manera, la red se especializa o "memoriza" un conjunto de datos determinados con un error muy pequeño.

Este sobre-entrenamiento trae como consecuencia que el error de *test* o verificación de la red sea mucho mayor que el de entrenamiento lo cual es indeseable cuando la red está trabajando en la solución de una aplicación específica.

Surge entonces esta técnica, que llamaremos en este libro como Regula-

rización, cuyo objetivo es minimizar el fenómeno del sobre-entrenamiento y por ende sus efectos. El fenómeno del sobre-entrenamiento se hace más evidente cuando los datos de entrada están contaminados con ruido. El objetivo de la regularización es garantizar un adecuado aprendizaje evitando este problema.

En la figura 3.5 observamos un caso típico de sobre-entrenamiento, dado para una aplicación donde la red neuronal artificial debe aproximar una función cuadrática, línea a trazos en la figura, cuyos datos vienen contaminados con ruido. En el ejemplo, hemos escogido una red neuronal con muchas neuronas en la capa oculta, con el fin de minimizar el error de aprendizaje; como era de esperarse, la salida de la red, representada en línea sólida, intenta ajustarse a cada uno de los puntos del conjunto de aprendizaje, situación muy distante de la salida deseada que corresponde a la línea a trazos.

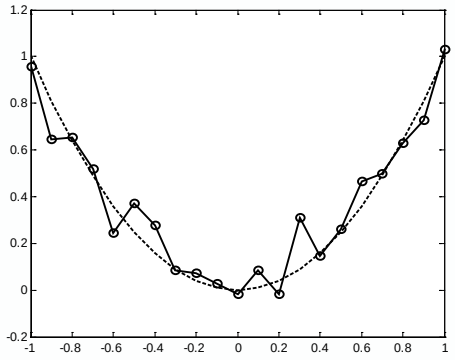


Fig. 3.5 Sobre-entrenamiento en una Red Neuronal Artificial

La solución obvia que surge para eliminar el sobre-entrenamiento es bajar la complejidad del modelo de red neuronal, reduciendo el número de neuronas en la capa oculta. Esta solución no siempre es la mejor, pues podemos caer en le fenómeno opuesto que hemos denominado sub-entrenamiento, tal como se presenta en la figura 3.6. En este caso, la salida de la red, línea sólida, trata de ajustarse a los datos de entrada a través de un modelo lineal, lo cual claramente es inadecuado.

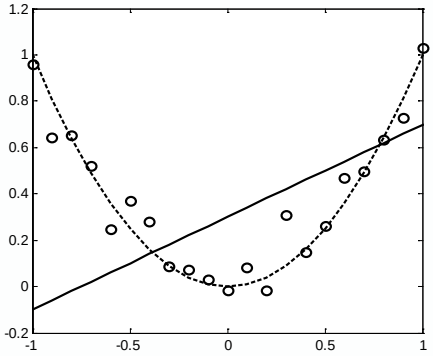


Fig. 3.6 Sub-entrenamiento en una Red Neuronal Artificial

En este libro plantearemos dos técnicas de regularización que buscan solucionar este problema:

- Regularización por parada temprana.
- Regularización por limitación de la magnitud de los pesos.

Regularización por parada temprana

Cuando definimos los conjuntos de datos para el proceso de aprendizaje de la red, habíamos mencionado dos, uno para aprendizaje y otro para *test* o verificación. Con esta técnica surge un nuevo conjunto de datos que denominaremos conjunto de datos de validación, el cual utilizaremos en la fase de entrenamiento, pero que no se toma en cuenta para la modificación de los pesos sinápticos de la red. Normalmente, dejamos entre un 20 y 30 por ciento de los datos para esta actividad de validación del proceso de aprendizaje.

En el proceso de aprendizaje ahora definiremos dos tipos de error, el primero es el de aprendizaje, cuya evolución la mostramos en la figura 3.7 con la línea azul y se define como la diferencia entre la salida de la red y el valor deseado, y este error es el que utilizamos para modificar los pesos de la red.

Con esta técnica, surge ahora, un nuevo error que llamamos de validación, cuya evolución se muestra con la línea a trazos y se define de igual manera, como la diferencia entre la salida de la red y el valor deseado, pero evaluado en un dato de entrada que pertenece al nuevo conjunto de datos de validación. Es importante aclarar que este error no lo consideraremos para la modificar los pesos sinápticos de la red, por lo que en este momento estamos observando el comportamiento de la red frente a nuevos datos que pertenecen al conjunto universo del problema, pero diferentes a los utilizados en el entrenamiento.

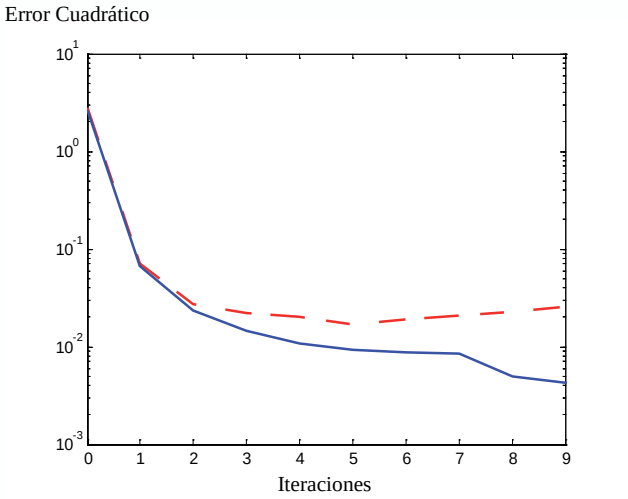


Fig. 3.7 Regularización por Parada Temprana

En la gráfica observamos que si seguimos el proceso con la tendencia marcada por la línea sólida, caeremos en un sobre-entrenamiento puesto que el error seguirá bajando pero para los datos del conjunto de entrenamiento; sin embargo, la línea a trazos nos muestra que para el conjunto de datos de validación, el error tiene una tendencia creciente.

La solución es simple y la técnica nos recomienda parar el proceso de entrenamiento de la red justo antes de que el error de validación empiece con una tendencia ascendente, por esta razón, le llamamos "Regularización por Parada Temprana", debido a que evitamos que el proceso de aprendizaje continúe bajando excesivamente el error y para ello tomamos como criterio de finalización la tendencia del error de validación.

Es bueno aclarar, que al finalizar este proceso, continuamos con la prueba final de desempeño de la red, para lo cual usamos los datos de verificación.

Regularización por limitación de la magnitud de los pesos

La experiencia nos ha mostrado que podemos garantizar una función de salida suave si logramos mantener los pesos sinápticos de la red en unos valores relativamente pequeños. Con el fin de limitar la magnitud de estos pesos redefinimos el cálculo del error así:

$$E_R = \gamma E_D + \lambda E_W \tag{3.52}$$

En la ecuación 3.52, el error se calcula con dos términos, por lo que utilizamos una nueva notación y llamaremos E_R al Error Regularizado. El primer término, ecuación 3.53, corresponde al error cuadrático promedio

que es la forma como tradicionalmente hemos estimado el error de entrenamiento, pero afectado por el parámetro γ.

$$E_D = \frac{1}{2P}\sum_{p=1}^{P}\sum_{k=1}^{M}(d_{pk} - y_{pk})^2 \tag{3.53}$$

El segundo término utilizado para estimar el error de aprendizaje regularizado, introduce la sumatoria de los pesos sinápticos de la red, ecuación 3.54, ponderados con el parámetro λ. Ya que el algoritmo de aprendizaje tiende a minimizar E_R, el resultado final es que la sumatoria de pesos igualmente tiende a ser minimizada para garantizar una suavidad en la salida de la red.

$$E_W = \sum_{n=1}^{N} w_n^2 \tag{3.54}$$

APROXIMACIÓN PRÁCTICA

Luego de haber introducido los diferentes métodos de aprendizaje aplicados al Perceptron Multicapa, proponemos una serie de proyectos cuyo seguimiento nos permitirá llevar a cabo las siguientes actividades:

- Simular redes neuronales tipo MLP con MATLAB®
- Implementar redes neuronales tipo MLP con MATLAB®
- Validar redes neuronales entrenadas con MATLAB®
- Diseñar y simular redes neuronales tipo MLP con UV-SRNA

Solución del problema de la función xor con MATLAB®

Una de las mayores dificultades que presenta el Perceptron es el no poder solucionar problemas no – lineales, razón por la cual cayeron en desuso las redes neuronales artificiales. En este capítulo hemos afirmado que al MLP solventa esta deficiencia, para evaluar esta potencialidad de las redes MLP, ejecutemos el siguiente programa escrito para MATLAB® donde se aplica el MLP en la solución de la función XOR. El programa al final nos entrega la superficie de separación no lineal cuando entrenamos una red MLP y una de las salidas la vemos en la figura 3.9.

```
function xorback(metodo,n)
% xorback.m
% Entrenamiento de una red neuronal MLP para resolver el problema de la XOR
% Además se construye la superficie de separación que genera la red entrenada
% Argumentos:
% metodo = método de aprendizaje
%          1 = Gradiente descendente
%          2 = Gradiente descendente con β variable
%          3 = Gradiente descendente con momentum
%          4 = Gradiente descendente con momentum y α variable
% n = Número de iteraciones
% Ejemplo:
%          xorback(1,50)
%          Se realizan 50 iteraciones de entrenamiento con el método del
%          gradiente descendente

close all;
x=[0 0 1 1;
   0 1 0 1];
y=[0 1 1 0];

 switch metodo
 case 1

 met_ent='traingd'
 case 2
 met_ent='traingda'
 case 3
 met_ent='traingdm'
 case 4
 met_ent='traingdx'
 otherwise
 met_ent='traingda'
 end

red=newff(minmax(x),[10 1],{'tansig','purelin'},met_ent);
red.trainParam.epochs=n;
figure;
red=train (red,x,y);
button = questdlg('Ver Superficie?',...
```

```
'Ver Superficie','Si','No','No');
if strcmp (button,'Si')

figure;hold on;axis([-0.2 1.2 -0.2 1.2]),title('Azul=Salida en Cero
Rojo=Salida en Uno')
plot(x(1,1),x(2,1),'ob','LineWidth',7);
plot(x(1,4),x(2,4),'ob','LineWidth',7);
plot(x(1,2:3),x(2,2:3),'or','LineWidth',7);

for i=-0.1:0.1:1.1
for j=-0.1:0.1:1.1
yred=sim(red,[i;j]);
if yred>0.7
plot(i,j,'or');
end;
if yred<0.3
plot(i,j,'ob');
end;
end;
end;
hold off

elseif strcmp(button,'No')
disp ('El programa ha terminado')
end
```

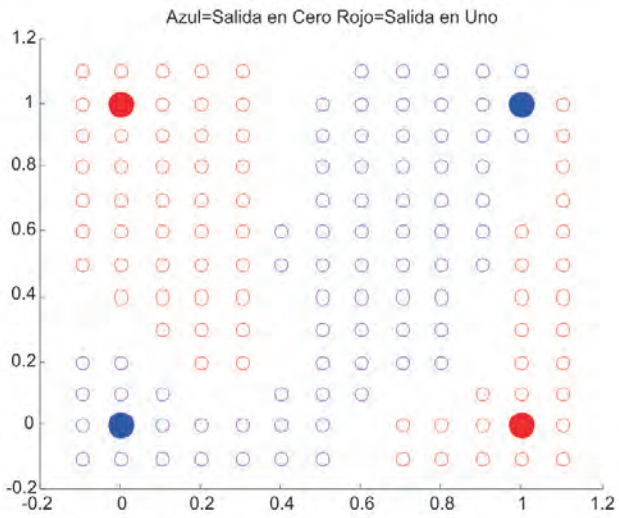


Fig. 3.9 Separación lograda por una red MLP

Aprendizaje de una función seno con MATLAB®

Las redes MLP son consideradas aproximadores universales de funciones, ya que es posible entrenar una red de este tipo para realizar una transformación matemática de un espacio *n-dimensional* a uno *m-dimensional* con un grado de precisión predefinido.

Como ejemplo de esta característica, con el programa en MATLAB® que presentamos en el recuadro, entrenaremos una red MLP que está en capacidad de aprender la dinámica de una función seno ($y = sen\ x$). Para ilustrar este ejemplo utilizaremos una red MLP de tres capas, como la presentada en la figura 3.10, es decir una neurona en la capa de entrada, un número de neuronas en la capa oculta que define el usuario y una neurona en la capa de salida. El resultado del aprendizaje de la función lo presentamos en la figura 3.11, donde observamos que la salida de la red neuronal sigue adecuadamente la dinámica de la función seno de entrada.

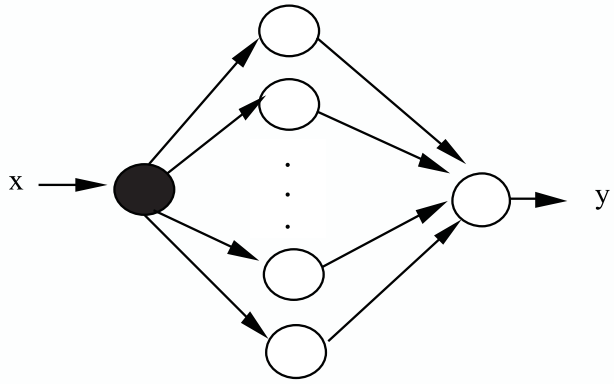


Fig. 3.10 Estructura de la red a entrenar

```
function senoback(metodo,nco,n)
% senoback.m
% Entrenamiento de una red neuronal MLP para aprender una fun-
ción      seno
% Argumentos:
% metodo = método de aprendizaje
%          1 = Gradiente descendente
%          2 = Gradiente descendente con eta variable
%          3 = Gradiente descendente con momentum
%          4 = Gradiente descendente con momentum y eta variable
% nco = Número de Neuronas en la capa oculta
% n = Número de iteraciones
% Ejemplo:
%          senoback(1,10,50)
```

```
%    Se realizan 50 iteraciones de entrenamiento con el método del
%    gradiente descendente a una red de 10 neuronas en la capa
     oculta

close all;
x=0: pi/10:2*pi;
y=sin(x);

switch metodo
case 1
met_ent='traingd'
case 2
met_ent='traingda'
case 3
met_ent='traingdm'
case 4
met_ent='traingdx'
otherwise
met_ent='traingda'
end

red=newff(minmax(x),[nco 1],{'tansig','purelin'},met_ent);
yred=sim(red,x);
figure; hold on;
plot(x,y,'+b');plot(x,yred,'or');title('Situación Inicial (+=Seno
original o=Salida de la red)')
red.trainParam.epochs=n;
figure;
red=train(red,x,y); yred=sim(red,x);
figure, hold on;
plot(x,y,'+b'); plot(x,yred,'or');title('Situación Final (+=Seno
original o=Salida de la red)')
```

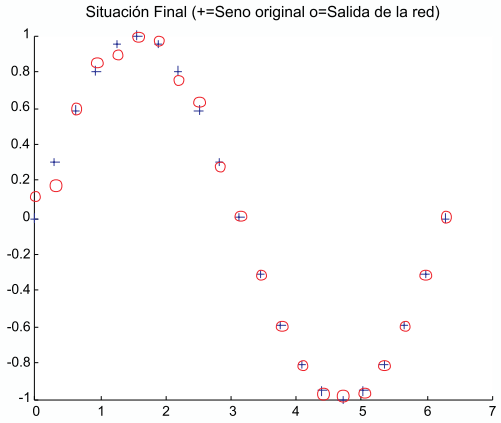


Fig. 3.11 Situación del aprendizaje de la red

Aprendizaje de la función silla de montar con MATLAB®

En este nuevo proyecto, evaluaremos la capacidad que tiene una red MLP para aprender funciones de dos variables, para lo cual tomamos como ejemplo la función Silla de Montar definida por la ecuación 3.55 y cuya superficie resultante la mostramos en la figura 3.12.

$$z = f(x, y) = x^2 - y^2 \tag{3.55}$$

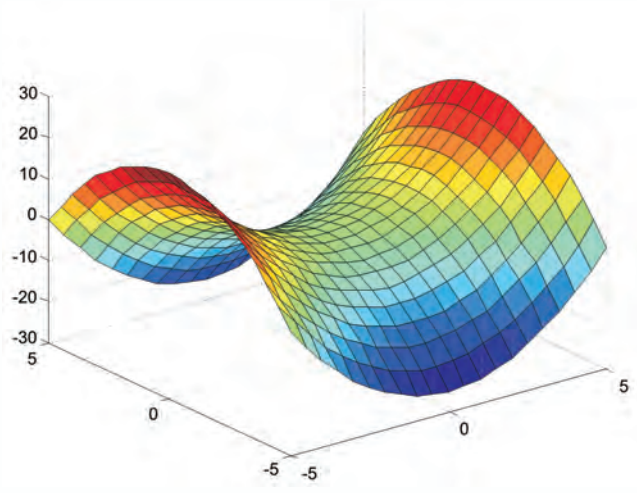


Fig. 3.12 Superficie original

En este caso la red neuronal que utilizaremos para el entrenamiento de esta función tiene la estructura de la figura 3.13 con dos neuronas de entrada, una capa oculta cuya dimensión la define el usuario como parámetro de entrada del programa del recuadro y una neurona de salida.

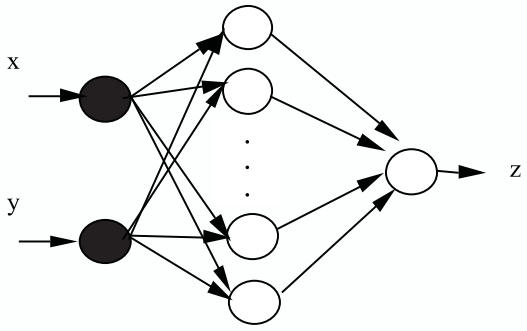


Fig. 3.13 Estructura de la red a entrenar

En las figuras 3.14 a 3.17, observamos la evolución del algoritmo de aprendizaje de la superficie definida. En la figura 3.14, vemos la salida de la red cuando inicia su proceso de entrenamiento por lo que el nivel de acercamiento a la función objetivo de silla de montar es nulo. En la figura 3.15, tomamos un punto intermedio en el proceso y notamos una clara mejoría en la aproximación a la función que hace la red neuronal. Observemos en la misma figura, con círculos representamos los patrones de entrenamiento de la función silla de montar cuya forma queremos aprender con la red, con asteriscos representamos la salida de la red en este instante de aprendizaje. Finalmente, en la figura 3.16, presentamos la salida de la red en formato de superficie, luego de 200 iteraciones, donde se observa que la superficie aprendida de la red es prácticamente igual función objetivo. En la figura 3.17, podemos apreciar en detalle la similitud de las dos funciones, tanto la objetivo cuyos patrones de aprendizaje los representamos con círculos, como la de salida de la red que la representamos con asteriscos.

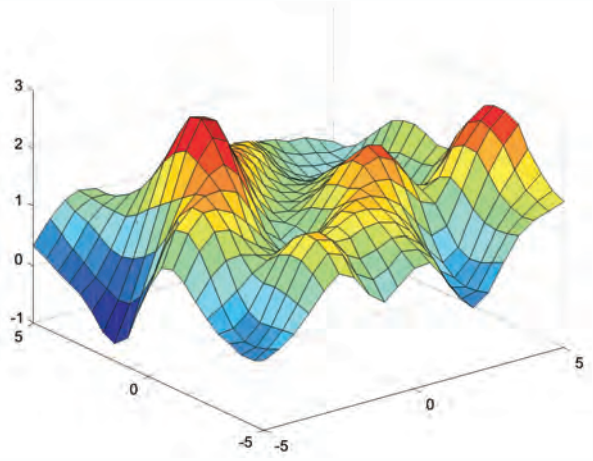


Fig. 3.14 Superficie aprendida por la red al inicio del aprendizaje

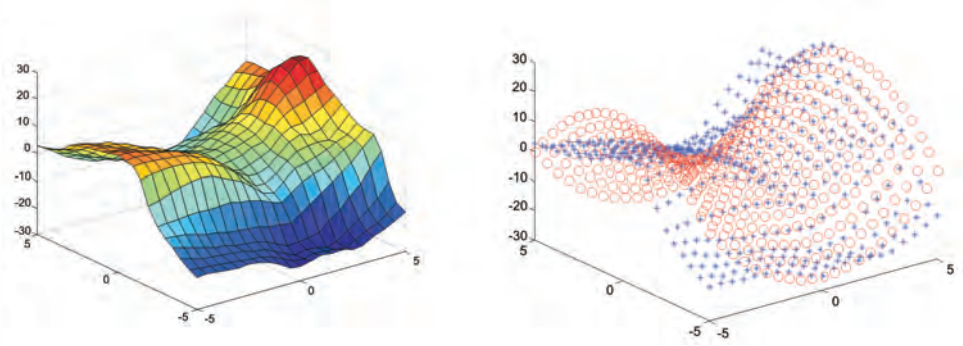


Fig. 3.15 Superficie aprendida por la red en una etapa intermedia del aprendizaje

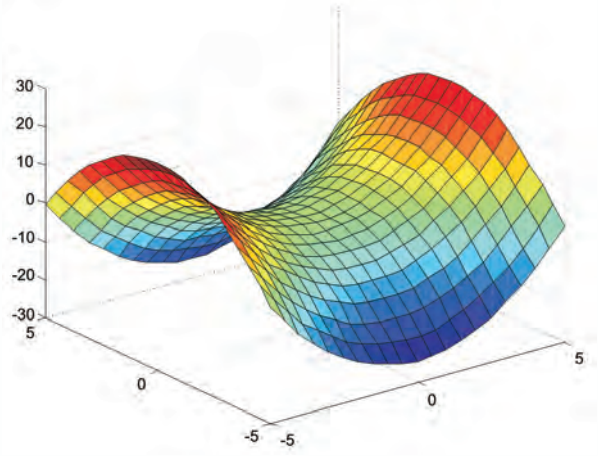


Fig. 3.16 Superficie aprendida por la red al fin del aprendizaje

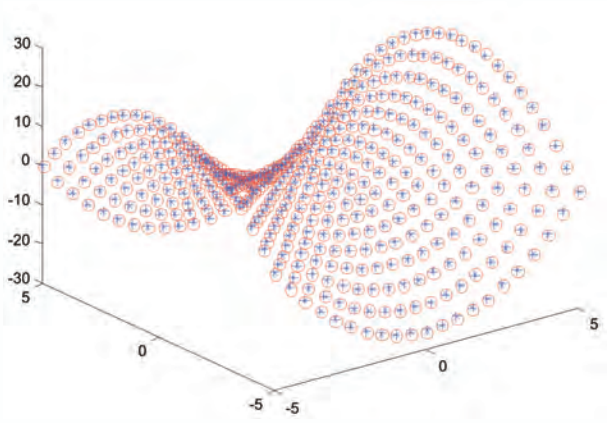


Fig. 3.17 Relación entre los patrones de aprendizaje (círculos) y la salida de la red (asteriscos)

A continuación presentamos el código en MATLAB® que permite implementar la aproximación de la función Silla de Montar usando una red neuronal tipo MLP.

```
%AproxSillaMontar.m
%Programa que permite aprender la función Silla de Montar.
close all;
x=-5:0.5:5;
y=x;
[X,Y]=meshgrid(x,y);
Z=X.^2-Y.^2;
figure;
surf(X,Y,Z);
%%
[f,c]=size(X);
N=f*c;
Xred=[reshape(X,1,N);
        reshape(Y,1,N)];
Zdred=[reshape(Z,1,N)];
red=newff(minmax(Xred),[20 1],{'tansig','purelin'},'trainlm');
red.trainparam.epochs=20;
Zredini=sim(red,Xred);
Zredinigra=reshape(Zredini,f,c);
figure;
surf(X,Y,Zredinigra)
%%
```

```
red=train(red,Xred,Zdred);
Zredfin=sim(red,Xred);
Zredfingra=reshape(Zredfin,f,c);
figure;
surf(X,Y,Zredfingra)
figure
plot3(reshape(X,1,N),reshape(Y,1,N),reshape(Z,1,N),'or');
hold on
plot3(reshape(X,1,N),reshape(Y,1,N),Zredfin,'*b');
```

Solución del problema de la xor con uv-srna

La herramienta de simulación UV-SRNA posee un módulo con el que se pueden entrenar redes tipo MLP con el algoritmo de gradiente descendente (Backpropagation), cuya interfaz la mostramos en la figura 3.18.

Fig. 3.18 Módulo Para Entrenar Redes MLP con un Algoritmo Gradiente Descendente en UV-SRNA

Si deseamos resolver el problema de la *XOR* con UV-SRNA el primer paso es crear un archivo texto con los patrones de dicha función, tal como lo mostramos en la Tabla 3.1.

Tabla 3.1 Codificación de los patrones de entrenamiento para la función lógica XOR

Datos en el archivo	*Significado*
4	Número de patrones de entrenamiento
2	Número de entradas de cada patrón
1	Número de salidas de cada patrón
0 0 0	Patrón No. 1
0 1 1	Patrón No. 2
1 0 1	Patrón No. 3
1 1 0	Patrón No. 4

Una vez creado el archivo de patrones, lo debemos cargar desde UV-SRNA, usando la opción *leer patrones de entrada* del menú de *archivo.*

Con los patrones disponibles procedemos a inicializar la red, para lo cual solo es necesario oprimir el botón de *inicializar*. Luego, procedemos a su entrenamiento oprimiendo el botón *entrenar*. Podemos observar como evoluciona el error de entrenamiento y el proceso se dará por terminado cuando el error es menor que el error mínimo aceptado. La aplicación muestra un ventana donde se visualiza la manera como cambia el error de entrenamiento, figura 3,19

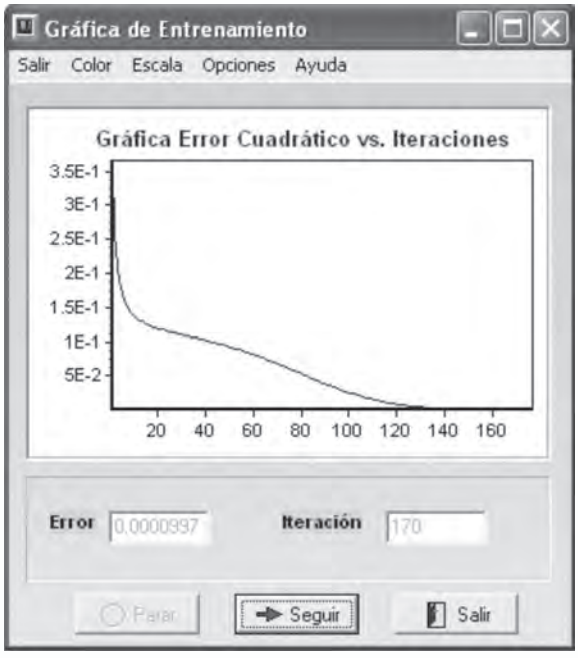


Fig. 3.19 Evolución del error de entrenamiento en UVSRNA

Una vez alcanzado el error mínimo, procedemos a validar la red, para lo cual la herramienta de simulación UV-SRNA nos provee dos formas, una por teclado y la otra por archivo.

Validación por teclado

Al seleccionar el *item validar la red por teclado* del menú *validar* se despliega una ventana como la mostrada en la figura 3.20, en ella el usuario digita unas posibles entradas para la red y usando el botón *probar* genere la salida ante dichas entradas.

Fig. 3.20 Ventana de Validación por teclado en UVSRNA

Validación por archivo

Esta opción se invoca al seleccionar la opción *validar la red por archivo* del menú validar. Para usar esta posibilidad es necesario haber creado un archivo de validación (extensión **.val*) con las entradas que se desean probar en la red neuronal. Básicamente un archivo de validación es un archivo de patrones pero sin la información de la salida deseada.

Tabla 3.2 Codificación de un archivo de validación para la función lógica XOR

Datos en el archivo	*Significado*
4	Número de patrones de validación
2	Número de entradas de cada patrón
0 0	Entrada a evaluar No. 1
0 1	Entrada a evaluar No. 2
1 0	Entrada a evaluar No. 3
1 1	Entrada a evaluar No. 4

UV-SRNA recibe este archivo y la salida generada la guarda en un archivo de salida *(*.sal*). El programa pregunta los nombres tanto del archivo de validación a usar como del archivo de salida a generar. Como el archivo de salida es un archivo texto, se puede abrir en un editor de texto cualquiera. Un ejemplo de un archivo de salida se muestra en la figura 3.21.

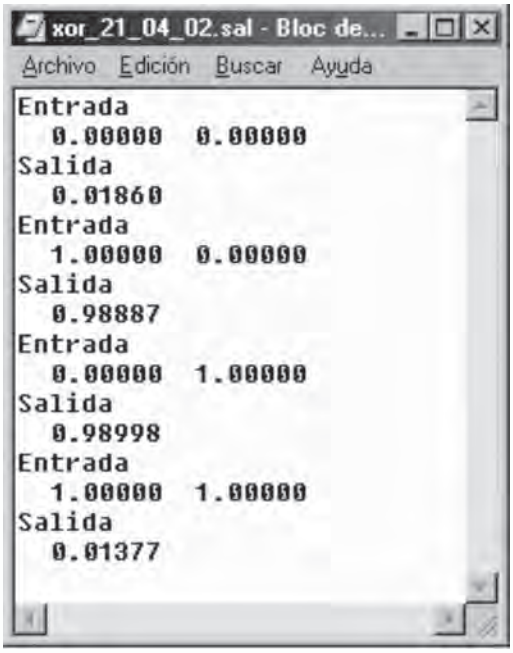


Fig. 3.21 Archivo de salida generado por UVSRNA

Identificación de sistemas usando un MLP

En este proyecto asumiremos un reto muy interesante que se soporta en la propiedad de las redes neuronales de aproximar una función con una

precisión predefinida, el objetivo es realizar la identificación de un sistema a partir de datos experimentales de entrada y salida, que utilizaremos como datos de entrenamiento de la red. En general, un sistema dinámico lo podemos describir como una función *f* que procesa los valores de la entrada y salida del sistema en instantes anteriores para proporcionar el muestreo en el instante *k*, es decir:

$$y(k) = f\left(y(k-1), y(k-2), \ldots, y(k-n), u(k-1), u(k-2), \ldots, u(k-m)\right) \quad (3.56)$$

El problema se reduce a entrenar una red neuronal que aprenda la función *f*, tal como se esquematiza en la figura 3.22, donde la red neuronal hace la estimación de la salida en el instante *k* con base en valores anteriores de la entrada y salida del sistema.

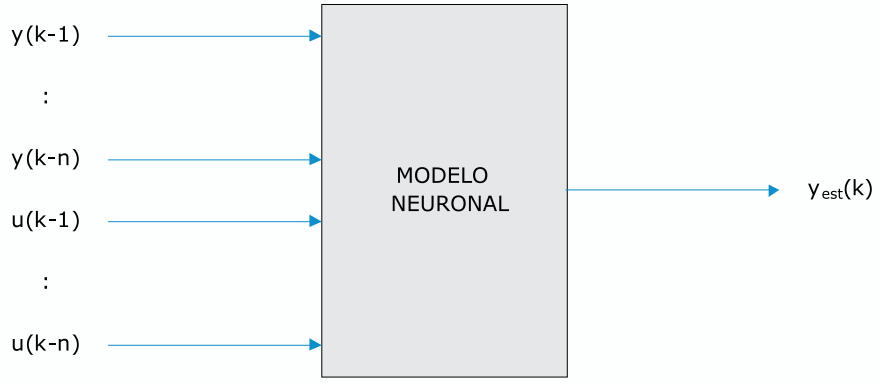


Fig. 3.22 Red Neuronal Usada para Identificar un Sistema Dinámico o Planta

Como un primer ejercicio, identifiquemos una planta conocida de primer orden que responde al modelo representado con la función de transferencia de la ecuación 3.57.

$$G(s) = \frac{1}{(s+2)} \quad (3.57)$$

Diseño del experimento y muestreo de datos

El primer paso es definir previamente el rango en el que se desea identificar la planta, para nuestro caso se diseñará un ejemplo que permita identificar el proceso propuesto en un rango de entrada entre 0 y 1.

Los datos experimentales se obtienen por simulación, suministrándole a la planta una entrada que cubre todo el rango de la entrada propuesto, esta simulación se realiza en la herramienta *Simulink* de MATLAB® con el esquema mostrado en la figura 3.22.

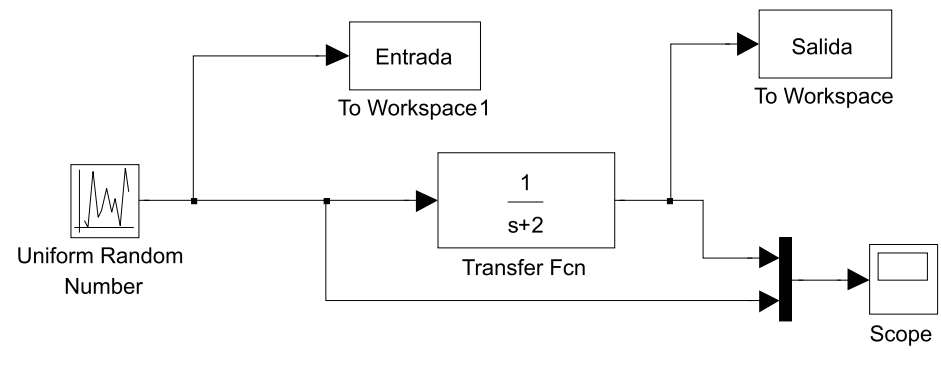


Fig. 3.23 Esquema en Simulink del experimento

El bloque de entrada es un bloque tipo *Uniform Random Number* que se encuentra en la biblioteca de *sources*.

Debemos ajustar los parámetros de estos bloques de manera adecuada, de tal manera que obtengamos datos de entrada y salida como los de la figura 3.24.

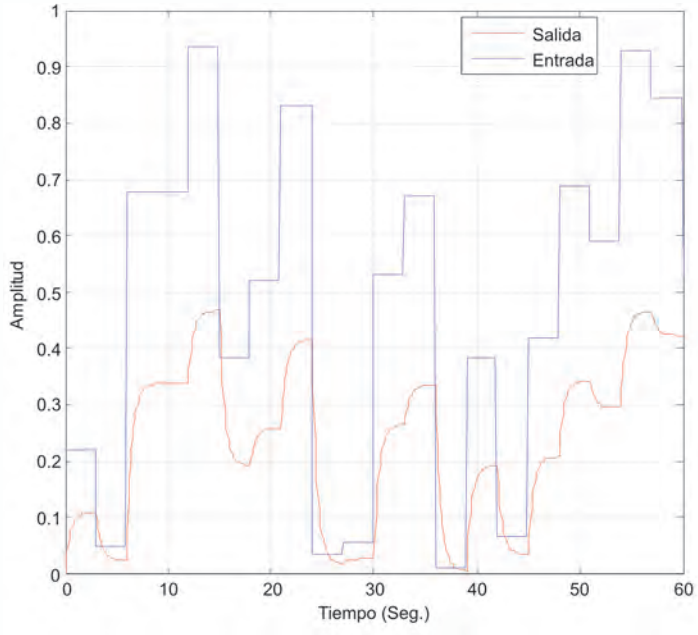


Fig. 3.24 Una posible entrada y su salida obtenida en el experimento

Recomendaciones:

- El tiempo de muestreo lo debemos ajustar entre 0.1 y 0.2 veces la constante de tiempo más rápida del sistema.
- El ancho de los escalones de entrada debe ser aproximadamente el tiempo de estabilización de la planta. Esto lo configuramos con el bloque de entrada usando el parámetro *Simple time.* Tener en cuenta que este parámetro no se debe interpretar como el tiempo de muestro que se configura en los bloques to workspace de la librería Sinks.
- La entrada del experimento debe cubrir todo el rango posible de valores de la variable de entrada del proceso.
- Por lo general la entrada del experimento se diseña para que cubra el rango de la entrada del proceso ± el 20%.

Modelo a usar y estimación de parámetros (entrenamiento de la red)

Con los datos experimentales, procedemos a definir una arquitectura de red neuronal que sirve para identificar la planta en cuestión. Como la planta es de primer orden y al usar un modelo ARX, se necesitan como entrada de la Red Neuronal un retardo de la entrada y un retardo de la salida para obtener la salida actual; esto se puede expresar en la ecuación 3.58.

$$y(k) = f(u(k - 1), y(k - 1))) \tag{3.58}$$

Lo anterior nos lleva a la siguiente arquitectura de Red Neuronal.

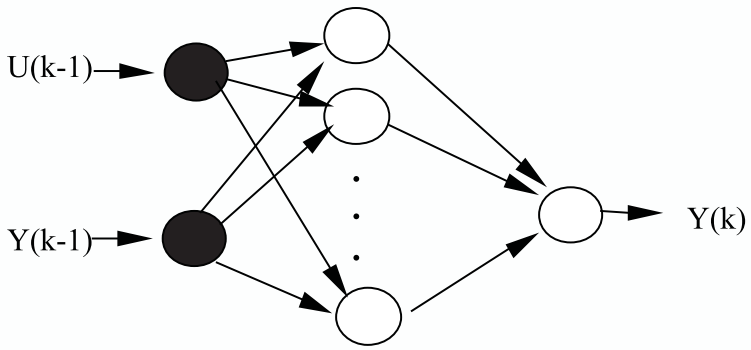


Fig. 3.25 Estructura de la RNA utilizada para la identificación

Como vemos en la figura 3.25, la red tiene dos neuronas en la capa de entrada, en la capa oculta el número de neuronas la define el usuario en la aplicación software que entregamos en el recuadro y una neurona en la capa de salida.

El paso siguiente, es organizar los datos que se obtuvieron durante el

experimento y definir los patrones de entrenamiento a utilizar en el proceso de aprendizaje de la Red Neuronal Artificial. Esto lo llevamos a cabo construyendo un programa **.m* en MATLAB®, el cual lo denominaremos *entrenar.m* y aparte de organizar los vectores para obtener las entrada que necesita la red neuronal, también realiza el entrenamiento de la red neuronal.

Creemos el archivo *entrenar.m* usando el código MATLAB® que se muestra a continuación.

```
% Inicio del archivo entrenar.m

% Entrenamiento de una red MLP para la identificación.
% de una planta lineal de primer orden.
% U debe contener los datos de entrada a la planta.
% Y debe contener los datos de salida a la planta.
% U, Y deben ser vectores columna.

% En el modelo neuronal que utilizamos, la planta se considera de
primer orden.
% esto significa que la salida actual depende de una muestra ante-
rior de la entrada % y de dos muestras anteriores de la salida así que
los datos de entrada-salida.
% deben llevarse a la forma:

%|  Entrada1  | Entrada2  | Salida |
%=========================================
===
%|  U(1 )       |  Y(1)      | Y(2)  |
%|  U(2 )       |  Y(2)      | Y(3)  |
%| ..................................................................
%|   U(K-1)   |  Y(K-1)  | Y(K)  |
%
% Se pierde el último dato del vector de entrada U
U=Entrada;
Y=Salida;
No_Datos = length(U);
% Primera entrada de la RNA
IN_RNA_1 = [U(1:No_Datos-1)];
% Segunda entrada de la RNA
IN_RNA_2 = [Y(1:No_Datos-1)];
% Salida de la RNA
OUT_RNA  = [Y(2:No_Datos)];
```

```
% Definición de la entrada de los patrones usados para el entrenamiento
P = [IN_RNA_1 IN_RNA_2];

% Cada Columna debe representar un vector de entrenamiento
% Luego P y OUT_RNA deben tener la misma cantidad de columnas
% Para lograr esto se trasponen los vectores tal y como se han definido.

X = P';
YD = OUT_RNA';

% Inicialización de la Red Neuronal.

% Una capa oculta de 10 neuronas con función de activación tangente sigmoidal.
% Una capa de salida de 1 neurona con función de activación lineal.

Limits = minmax(X);
red = newff(Limits,[10 1],{'tansig' , 'purelin'},'trainlm');

%Ahora se procederá a entrenar la RNA.
% Parámetros de Entrenamiento:
% Frecuencia de visualización del progreso del entrenamiento (en iteraciones).
red.trainParam.show = 20;
% Número máximo de iteraciones de entrenamiento
red.trainParam.epochs = 100;

% Error cuadrático promedio mínimo deseado.
red.trainParam.goal = (0.000000001);

% Llamada a la función de entrenamiento
red = train(red,X,YD);

% Fin del archivo entrenar.m
```

Validación del modelo obtenido con la rna

Luego de procesar el entrenamiento, se tiene en el objeto red una red neuronal que ha aprendido la dinámica de la planta, para validar este modelo se realiza un esquema en la herramienta *Simulink* de MATLAB® como el mostrado en la figura 3.26.

Primero debe obtenerse una representación en diagramas de bloques de la red neuronal que se ha acabado de entrenar, para esto se utiliza el comando *gensim* indicando el tiempo de muestreo usado para la obtención de los datos.

```
>> gensim(red,Ts)
```

donde,

red : Objeto de la red que se ha entrenado
Ts : Tiempo de muestreo seleccionado

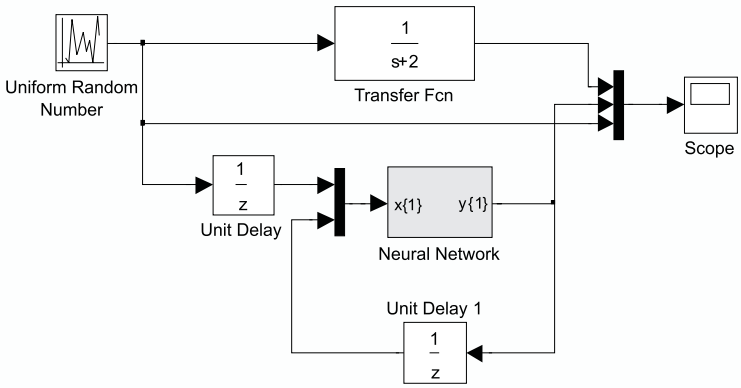


Fig. 3.26 Esquema en Simulink para validar el modelo neuronal

Cuando llevamos a cabo la simulación del sistema en Simulink, el programa nos entrega una salida como la mostrada en la figura 3.27.

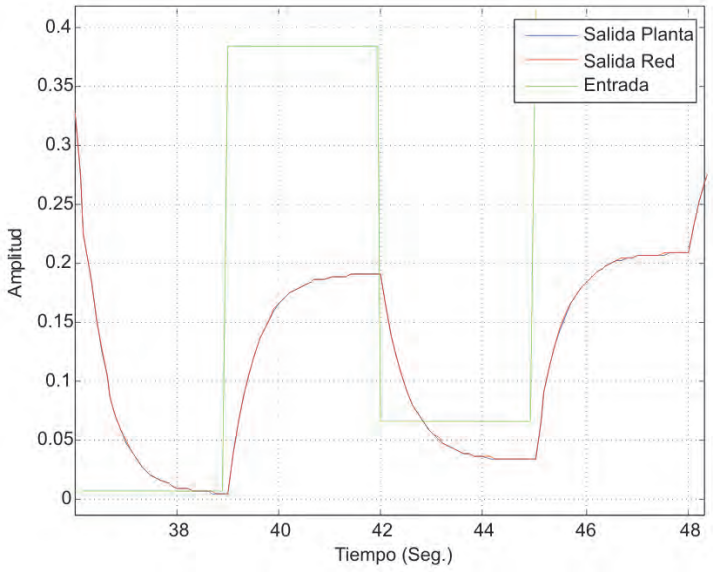


Fig. 3.27 Validación del modelo neuronal obtenido

Como podemos observar la salida de la red sigue plenamente la salida de la planta original, por lo que aseguramos que la red neuronal tipo MLP aprendió o identificó la dinámica de la planta.

Pronóstico de consumo de energía (demanda)

Un nuevo tipo de aplicación de las redes neuronales tipo MLP es en el pronóstico de series temporales, que ilustraremos un ejemplo de pronóstico de consumo de ener-gía para tres tipos de usuarios. En la Tabla 3.3 presentamos los consumos en KWH en diferentes horas del día de tres tipos básicos de consumidores de energía eléctrica; el objetivo es entrenar una red neuronal que permita pronosticar el consumo en KWH en cualquier hora del día, utilizando el simulador de redes neuronales de la Universidad del Valle, UV-SRNA.

Tabla 3.3 Demanda en KWH a diferentes horas del día

DEMANDA EN KW			
	Residencial	Comercial	Industrial
0: 00 p.m.	780	1650	5100
2: 00 a.m.	610	990	4750
4: 00 a.m.	590	990	4700
6: 00 a.m.	580	990	4850
8: 00 a.m.	710	2100	8500
10:00 a.m.	770	2990	9800
12:00 m.	790	3950	9550
14:00 p.m.	780	3900	8950
16:00 p.m.	810	3950	7450
18:00 p.m.	1450	3950	7790
20:00 p.m.	1995	3590	9980
22:00 p.m.	1380	2380	6450

Para el pronóstico proponemos una red con una neurona en la capa de entrada (la hora del día) y tres neuronas en la capa de salida para generar el consumo en los diferentes sectores.

Lo primero que tenemos que hacer es crear un archivo de patrones con la información dada en la tabla 3.3, es fundamental mencionar que debido a la naturaleza de la información es conveniente normalizar los datos de entrada y salida para que se ajusten en el rango entre 0 y 1. Para normalizar la entrada dividimos la hora entre 40, lo cual significa que en vez de pasarle a la red el 12 para indicar el medio día se le pasará (12/40) = 0.3 para representar este valor; la salida se normalizará dividiendo todos los valores entre 100, por lo tanto la red no entregará el valor 5000 sino 50. El procedimiento acabado de mencionar es lo que se conoce como escalamiento de los datos.

El archivo de patrones debe quedar como se muestra en la figura 3.28.

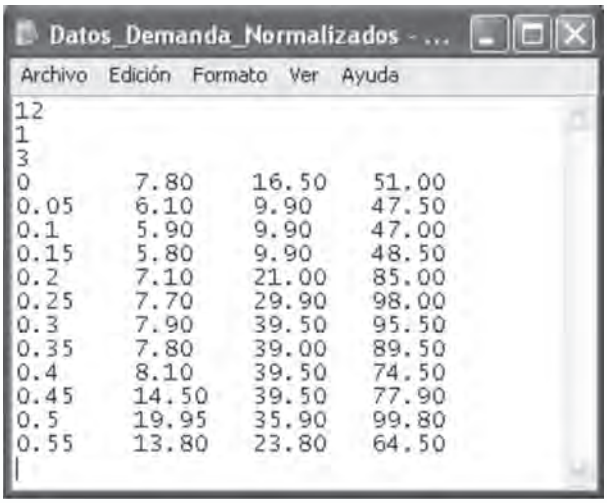


Fig. 3.28 Archivos de patrones

Teniendo el archivo de patrones cargado, se define una red con un capa oculta de 20 neuronas, la clave para lograr entrenar la red con este conjunto de datos es variar α cuando se perciba que el error de la red comienza a oscilar. Por ejemplo, con el α por defecto que tiene el simulador se obtuvo la curva de error mostrada en la figura 3.29.

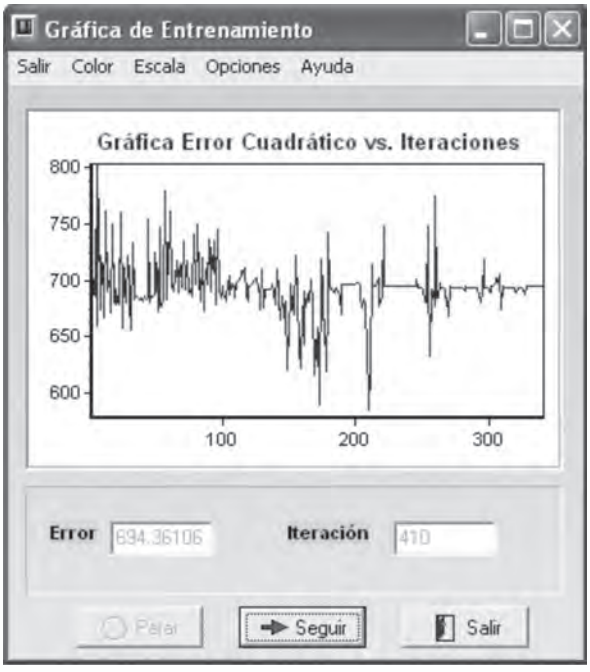


Fig. 3.29 Curva del error cuando $\alpha = 0.075$ $\beta = 0.1$

Para evitar esta oscilación detenemos el aprendizaje y hacemos $\alpha = 0.005$, el resultado mejora ostensiblemente como se observa en la figura 3.30.

Fig. 3.30 Curva del error cuando α = 0.005 β= 0.1

Es evidente que con este nuevo de α la red comienza a disminuir el error de una manera notoria. Siguiendo el entrenamiento y teniendo en cuenta lo acabado de mencionar, valide la red, para tal fin cree el archivo de validación respectivo y use la opción de validar la red por archivo, con lo cual se genera un archivo de resultados como el mostrado en la figura 3.31.

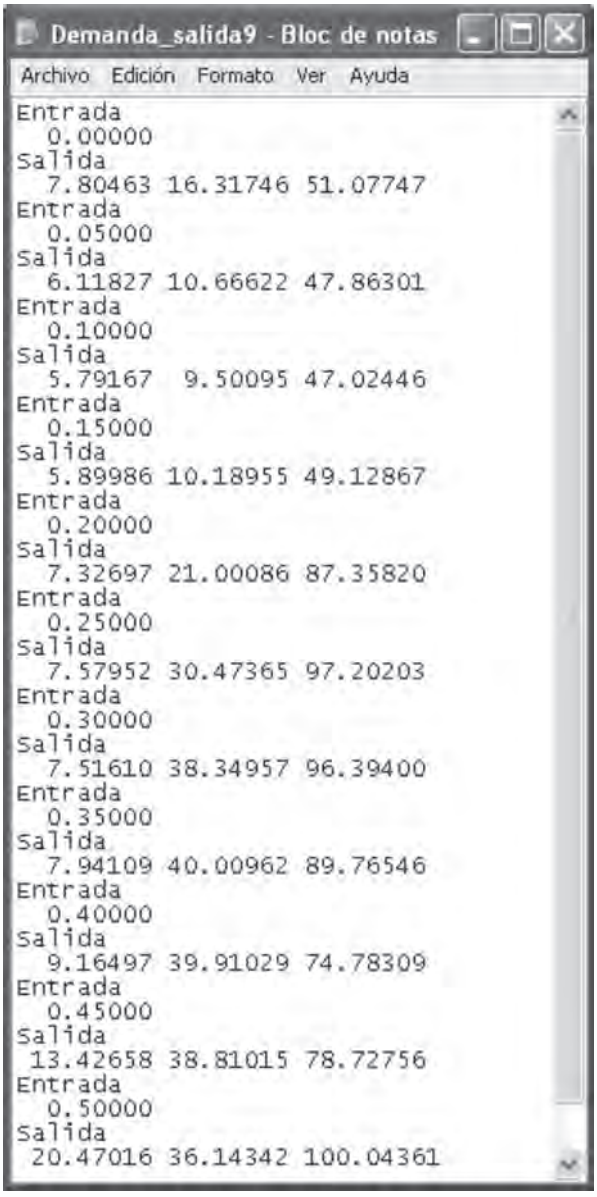


Fig. 3.31 Archivo de salida o resultados

Es necesario verificar si el entrenamiento de la red es el adecuado, con el fin de validar la red en el pronóstico de la demanda en cualquier hora, por ejemplo si deseamos saber cual es la demanda a las 23 horas escalamos este valor (17/40)=0.425, lo introducimos a la red y ésta nos genera los siguientes valores:

- 10.2335
- 39.6140
- 70.5042

Si ajustamos a la escala de salida se tiene que el pronóstico de consumo para dicha hora es:

- 1023 KWH para el sector residencial
- 13961 KWH para el sector comercial
- 7050 KWH para el sector industrial

Aplicación a la clasificación de patrones (el problema del IRIS)

El problema del IRIS es uno de los problemas mejor conocidos en la literatura de reconocimiento de patrones. El problema está definido por tres clases de especies de iris: *iris setosa canadensis, iris versicolor* y *iris virginica.* En la base de datos cada clase posee 50 ejemplos. Las características usadas para realizar la clasificación son cuatro: la longitud del sépalo en centímetros, el ancho del sépalo en centímetros, la longitud del pétalo en centímetros y el ancho del pétalo en centímetros.

El siguiente código en MATLAB® podemos entrenar una red neuronal tipo MLP con aprendizaje Levenberg-Marquardt. De los 50 ejemplos de cada una de las clases se seleccionamos el 70% de los datos para la etapa de entrenamiento, y el 30% para la validación.

```
% Ejemplo de clasificación de patrones usando la base de datos
IRIS
% El 70% de los datos los usamos para el entrenamiento
% y el 30% para validación

load iris_data2.txt;

Datos_train_C1=iris_data2(1:35,1:4);
Datos_val_C1=iris_data2(36:50,1:4);

Datos_train_C2=iris_data2(51:85,1:4);
Datos_val_C2=iris_data2(86:100,1:4);

Datos_train_C3=iris_data2(101:135,1:4);
Datos_val_C3=iris_data2(136:150,1:4);

X=[Datos_train_C1;Datos_train_C2;Datos_train_C3]';
  Yd=[ones(1,35)zeros(1,70);zeros(1,35)ones(1,35)
```

```
  zeros(1,35);zeros(1,70) ones(1,35) ];

red=newff(minmax(X),[12 3],{'tansig','logsig'},'trainlm');
red.trainparam.epochs=200;
red=train(red,X,Yd);

Xval=[Datos_val_C1;Datos_val_C2;Datos_val_C3]';
Yred1=sim(red,Xval);

figure
plot3(Datos_train_C1(:,1),Datos_train_C1(:,2),Datos_train_
C1(:,3),'*r'),hold on
plot3(Datos_train_C2(:,1),Datos_train_C2(:,2),Datos_train_
C2(:,3),'*g'),
plot3(Datos_train_C3(:,1),Datos_train_C3(:,2),Datos_train_
C3(:,3),'*b'), hold off
figure
plot3(Datos_val_C1(:,1),Datos_val_C1(:,2),Datos_val_
C1(:,3),'*r'),hold on
plot3(Datos_val_C2(:,1),Datos_val_C2(:,2),Datos_val_
C2(:,3),'*g'),
plot3(Datos_val_C3(:,1),Datos_val_C3(:,2),Datos_val_
C3(:,3),'*b'), hold off

Yred=round(Yred1);
figure; axis([4 8 2 4 1 7])
hold on
for i=1:15
if Yred(1,i)==1
plot3(Datos_val_C1(i,1),Datos_val_C1(i,2),Datos_val_
C1(i,3),'*r')
else
plot3(Datos_val_C1(i,1),Datos_val_C1(i,2),Datos_val_
C1(i,3),'*y')
end
end;

for i=1:15
if Yred(2,i+15)==1
plot3(Datos_val_C2(i,1),Datos_val_C2(i,2),Datos_val_
```

```
C2(i,3),'*g')
else
plot3(Datos_val_C2(i,1),Datos_val_C2(i,2),Datos_val_
C2(i,3),'*y')
end
end;

for i=1:15
if Yred(3,i+30)==1
plot3(Datos_val_C3(i,1),Datos_val_C3(i,2),Datos_val_
C3(i,3),'*b')
else
plot3(Datos_val_C3(i,1),Datos_val_C3(i,2),Datos_val_
C3(i,3),'*y')
end
end;

hold off
```

En la figura 3.32, podemos observar los datos utilizados para el entrenamiento, los asteriscos de diferente color representan los ejemplos de cada una de las clases usados para el proceso de aprendizaje. Para poder visualizar de manera adecuada el problema, seleccionamos las tres primeras características de las cuatro que en total tenemos a disposición.

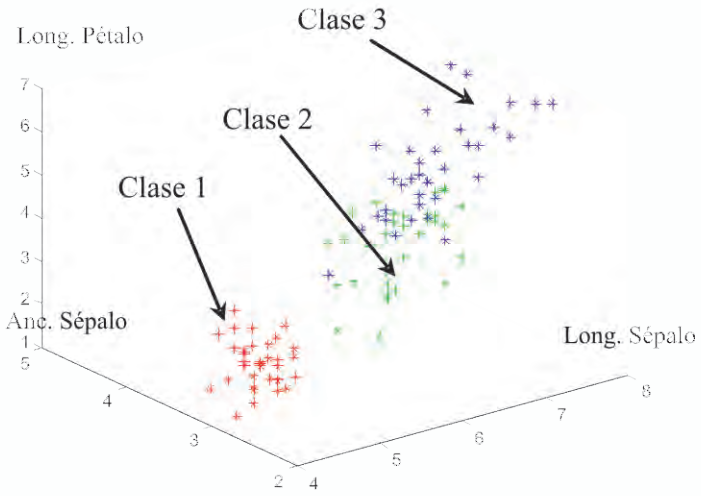


Fig. 3.32 Visualización de los datos de entrenamiento usando las tres primeras características

En la figura 3.33 podemos observar los datos utilizados para la validación.

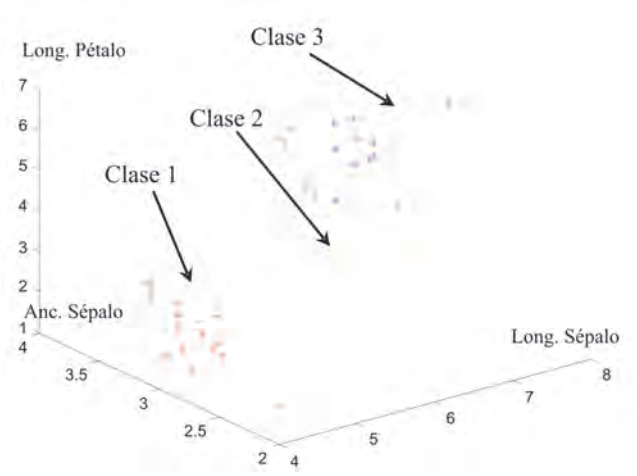


Fig. 3.33 Visualización de los datos de validación usando las tres primeras características

En la figura 3.34, podemos observar la salida de la red con los datos de validación, en donde un gran porcentaje de los ejemplos de validación fueron clasificados adecuadamente, para el ejemplo mostrado en la figura, sólo dos ejemplos fueron mal clasificados. Para distinguir dichos datos, se han representado en un color diferente a los utilizados para la definición de las clases.

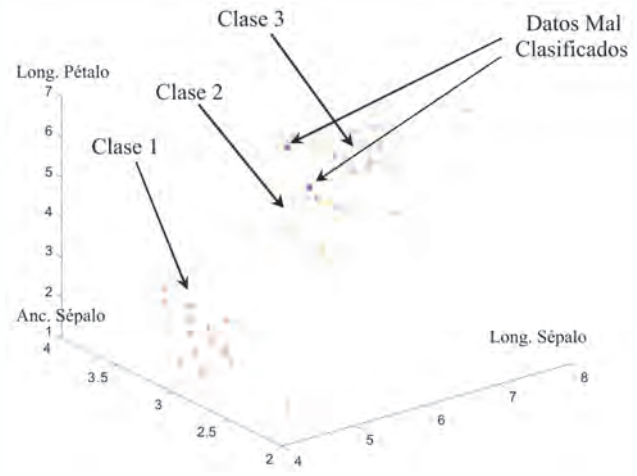


Fig. 3.34 Visualización de la clasificación realizada por la red neuronal las tres primeras características

Proyectos propuestos

1. Solucione el problema de la función XOR realizando el entrenamiento en UVSRNA.
2. Diseñe una aplicación usando UVSRNA, para que aprenda la dinámica de la función seno.
3. En el proyecto de la sección página 110, definir un número de iteraciones fijo, por ejemplo 100, y variar el método de aprendizaje para encontrar como cambia la calidad del aprendizaje de la red visualizando para tal fin la superficie de separación que la misma genera.
4. En el proyecto de la sección paginas 112 - 113, definir un número fijo de iteraciones y verificar el comportamiento de la red cuando se varia el número de neuronas en la capa oculta y el método de aprendizaje.
5. Proponga un método para evaluar la capacidad de generalización que tiene la red MLP al aprender una función, con base en el programa de la sección paginas 112 - 113.
6. Identifique la siguiente planta usando una red neuronal implementada en MATLAB®.

$$G(s) = \frac{5}{(s^2 + 4s + 10)}$$

7. Entrene con la herramienta UV-SRNA una red que sirva para encriptar palabras de 8 bits, para solucionar este problema se sugiere utilizar una red neuronal que en su entrada y salida tenga ocho neuronas y, en la capa oculta tres neuronas. Los datos de entrada y salida para el entrenamiento se sugieren en la siguiente tabla. Una vez entrenada la red la salida de la capa oculta responde al código de encriptación.

Sugerencia: Defina neuronas sigmoidales en la capa oculta y en la capa de salida. Esto hará que los códigos en la capa oculta y de salida se codifiquen con 0 y 1. Trate de lograr un error de entrenamiento de 10^{-5}

Código de Entrada	Código Encriptado (3 bits)	Código de Salida
1 0 0 0 0 0 0 0		1 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0		0 1 0 0 0 0 0 0
0 0 1 0 0 0 0 0		0 0 1 0 0 0 0 0
0 0 0 1 0 0 0 0		0 0 0 1 0 0 0 0
0 0 0 0 1 0 0 0		0 0 0 0 1 0 0 0
0 0 0 0 0 1 0 0		0 0 0 0 0 1 0 0
0 0 0 0 0 0 1 0		0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 1		0 0 0 0 0 0 0 1

8. Entrene una red MLP tanto en MATLAB® como en UV-SRNA que aprenda la función *(0.5*sin(x)+0.5*sin(2*x))* de tal forma que *x* esté en el rango $-\pi < x < \pi$. Para validar el entrenamiento verifique la generalización de la red entrenada.
9. Entrene una red MLP en MATLAB® que aprenda la función *Seno(x)/x* de tal forma que *x* esté en el rango *–10< x < 10*. Para validar el entrenamiento verifique la generalización de la red entrenada.
10. Entrene una red neuronal tipo MLP tanto en MATLAB® como en UV-SRNA que sirva para reconocer las vocales.
11. Entrene una red neuronal tipo MLP tanto en MATLAB® como en UVSRNA que sirva para reconocer los dígitos del 0-9.
12. Entrene una red MLP en MATLAB® que aprenda la siguiente función de dos variables que se muestra en la figura 3.35. Verifique la generalización de la red entrenada.

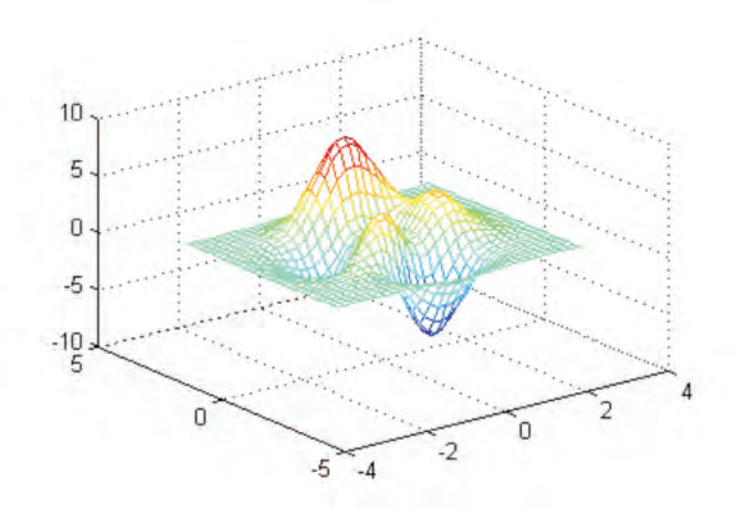


Fig. 3.35 Función de dos variables a identificar

Sugerencia: Utilice el siguiente código para generar la gráfica de la función a aprender.

```
Xini=[-3:0.2:3]; Yini=Xini;
[x,y]=meshgrid(Xini,Yini);
z= 3*(1-x).^2.*exp(-(x.^2) – (y+1).^2) ...
10*(x/5 – x.^3 – y.^5).*exp(-x.^2-y.^2) ...
1/3*exp(-(x+1).^2 – y.^2);
mesh(x,y,z)
```