2.3 BACKPROPAGATION

2.3.1 Antecedentes. La regla de aprendizaje del Perceptrón de Rosenblatt y el algoritmo LMS de Widrow y Hoff fueron diseñados para entrenar redes de una sola capa. Como se discutió anteriormente, estas redes tienen la desventaja que sólo pueden resolver problemas linealmente separables, fue esto lo que llevó al surgimiento de las redes multicapa para sobrepasar esta dificultad en las redes hasta entonces conocidas.

El primer algoritmo de entrenamiento para redes multicapa fue desarrollado por Paul Werbos en 1974, éste se desarrolló en un contexto general, para cualquier tipo de redes, siendo las redes neuronales una aplicación especial, razón por la cual el algoritmo no fue aceptado dentro de la comunidad de desarrolladores de redes neuronales. Fue sólo hasta mediados de los años 80 cuando el algoritmo Backpropagation o algoritmo de propagación inversa fue redescubierto al mismo tiempo por varios investigadores, David Rumelhart, Geoffrey Hinton y Ronal Williams, David Parker y Yann Le Cun. El algoritmo se popularizó cuando fue incluido en el libro “Parallel Distributed Processing Group” por los sicólogos David Rumelhart y James McClelland. La publicación de éste trajo consigo un auge en las investigaciones con redes neuronales, siendo la Backpropagation una de las redes más ampliamente empleadas, aun en nuestros días.

Uno de los grandes avances logrados con la Backpropagation es que esta red aprovecha la naturaleza paralela de las redes neuronales para reducir el tiempo

requerido por un procesador secuencial para determinar la correspondencia entre unos patrones dados. Además el tiempo de desarrollo de cualquier sistema que se esté tratando de analizar se puede reducir como consecuencia de que la red puede aprender el algoritmo correcto sin que alguien tenga que deducir por anticipado el algoritmo en cuestión.

La mayoría de los sistemas actuales de cómputo se han diseñado para llevar a cabo funciones matemáticas y lógicas a una velocidad que resulta asombrosamente alta para el ser humano. Sin embargo la destreza matemática no es lo que se necesita para solucionar problemas de reconocimiento de patrones en entornos ruidosos, característica que incluso dentro de un espacio de entrada relativamente pequeño, puede llegar a consumir mucho tiempo. El problema es la naturaleza secuencial del propio computador; el ciclo tomar – ejecutar de la naturaleza Von Neumann sólo permite que la máquina realice una operación a la vez. En la mayoría de los casos, el tiempo que necesita la máquina para llevar a cabo cada instrucción es tan breve (típicamente una millonésima de segundo) que el tiempo necesario para un programa, así sea muy grande, es insignificante para los usuarios. Sin embargo, para aquellas aplicaciones que deban explorar un gran espacio de entrada o que intentan correlacionar todas las permutaciones posibles de un conjunto de patrones muy complejo, el tiempo de computación necesario se hace bastante grande.

Lo que se necesita es un nuevo sistema de procesamiento que sea capaz de examinar todos los patrones en paralelo. Idealmente ese sistema no tendría que

ser programado explícitamente, lo que haría es adaptarse a sí mismo para aprender la relación entre un conjunto de patrones dado como ejemplo y ser capaz de aplicar la misma relación a nuevos patrones de entrada. Este sistema debe estar en capacidad de concentrarse en las características de una entrada arbitraria que se asemeje a otros patrones vistos previamente, sin que ninguna señal de ruido lo afecte. Este sistema fue el gran aporte de la red de propagación inversa, Backpropagation.

La Backpropagation es un tipo de red de aprendizaje supervisado, que emplea un ciclo propagación – adaptación de dos fases. Una vez que se ha aplicado un patrón a la entrada de la red como estímulo, éste se propaga desde la primera capa a través de las capas superiores de la red, hasta generar una salida. La señal de salida se compara con la salida deseada y se calcula una señal de error para cada una de las salidas.

Las salidas de error se propagan hacia atrás, partiendo de la capa de salida, hacia todas las neuronas de la capa oculta que contribuyen directamente a la salida. Sin embargo las neuronas de la capa oculta sólo reciben una fracción de la señal total del error, basándose aproximadamente en la contribución relativa que haya aportado cada neurona a la salida original. Este proceso se repite, capa por capa, hasta que todas las neuronas de la red hayan recibido una señal de error que describa su contribución relativa al error total. Basándose en la señal de error percibida, se actualizan los pesos de conexión de cada neurona, para hacer que la

red converja hacia un estado que permita clasificar correctamente todos los patrones de entrenamiento.

La importancia de este proceso consiste en que, a medida que se entrena la red, las neuronas de las capas intermedias se organizan a sí mismas de tal modo que las distintas neuronas aprenden a reconocer distintas características del espacio total de entrada. Después del entrenamiento, cuando se les presente un patrón arbitrario de entrada que contenga ruido o que esté incompleto, las neuronas de la capa oculta de la red responderán con una salida activa si la nueva entrada contiene un patrón que se asemeje a aquella característica que las neuronas individuales hayan aprendido a reconocer durante su entrenamiento. Y a la inversa, las unidades de las capas ocultas tienen una tendencia a inhibir su salida si el patrón de entrada no contiene la característica para reconocer, para la cual han sido entrenadas.

Varias investigaciones han demostrado que, durante el proceso de entrenamiento, la red Backpropagation tiende a desarrollar relaciones internas entre neuronas con el fin de organizar los datos de entrenamiento en clases. Esta tendencia se puede extrapolar, para llegar a la hipótesis consistente en que todas las unidades de la capa oculta de una Backpropagation son asociadas de alguna manera a características específicas del patrón de entrada como consecuencia del entrenamiento. Lo que sea o no exactamente la asociación puede no resultar evidente para el observador humano, lo importante es que la red ha encontrado una representación interna que le permite generar las salidas deseadas cuando se

le dan las entradas, en el proceso de entrenamiento. Esta misma representación interna se puede aplicar a entradas que la red no haya visto antes, y la red clasificará estas entradas según las características que compartan con los ejemplos de entrenamiento.

2.3.2 Estructura de la Red. La estructura típica de una red multicapa se observa en la figura 2.3.1

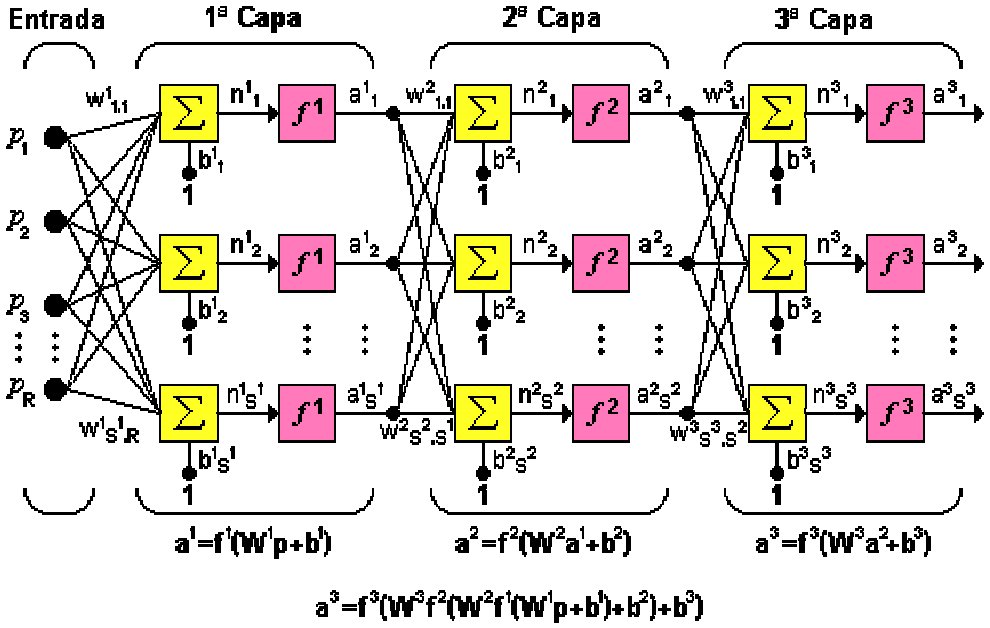

Figura 2.3.1 Red de tres capas

Puede notarse que esta red de tres capas equivale a tener tres redes tipo Perceptrón en cascada; la salida de la primera red, es la entrada a la segunda y la salida de la segunda red es la entrada a la tercera. Cada capa puede tener diferente número de neuronas, e incluso distinta función de transferencia.

En la figura 2.3.1, W^1 representa la matriz de pesos para la primera capa, W^2 los pesos de la segunda y así similarmente para todas las capas que incluya una red. Para identificar la estructura de una red multicapa, se empleará una notación abreviada, donde el número de entradas va seguido del número de neuronas en cada capa:

$$R : S^1 : S^2 : S^3 \qquad (2.3.1)$$

Donde S representa el número de neuronas y el exponente representa la capa a la cual la neurona corresponde.

La notación de la figura 2.3.1 es bastante clara cuando se desea conocer la estructura detallada de la red, e identificar cada una de las conexiones, pero cuando la red es muy grande, el proceso de conexión se torna muy complejo y es bastante útil utilizar el esquema de la figura 2.3.2

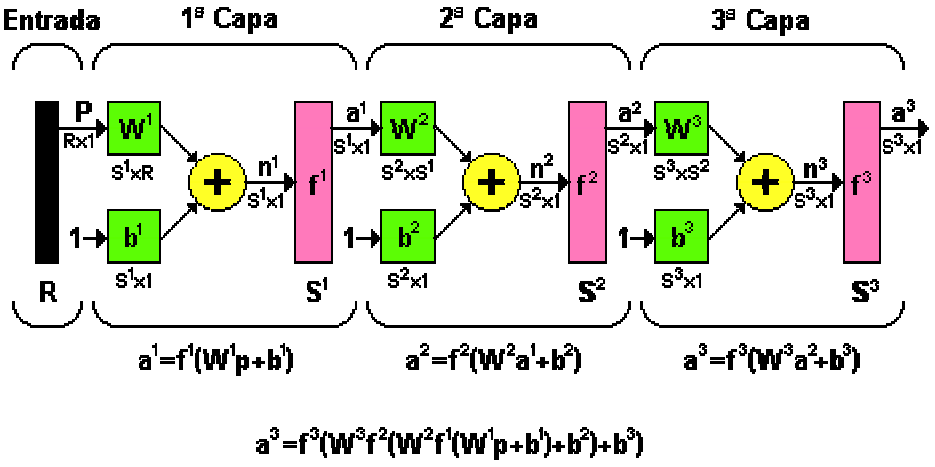

Figura 2.3.2 Notación compacta de una red de tres capas

2.3.3 Regla de Aprendizaje. El algoritmo Backpropagation para redes multicapa es una generalización del algoritmo LMS, ambos algoritmos realizan su labor de actualización de pesos y ganancias con base en el error medio cuadrático. La red Backpropagation trabaja bajo aprendizaje supervisado y por tanto necesita un set de entrenamiento que le describa cada salida y su valor de salida esperado de la siguiente forma:

$$\{p_1,t_1\}, \{p_2,t_2\}, \ldots ,\{p_Q, t_Q\} \qquad (2.3.2)$$

Donde p_Q es una entrada a la red y t_Q es la correspondiente salida deseada para el patrón q-ésimo. El algoritmo debe ajustar los parámetros de la red para minimizar el error medio cuadrático.

El entrenamiento de una red neuronal multicapa se realiza mediante un proceso de aprendizaje, para realizar este proceso se debe inicialmente tener definida la topología de la red esto es: número de neuronas en la capa de entrada el cual depende del número de componentes del vector de entrada, cantidad de capas ocultas y número de neuronas de cada una de ellas, número de neuronas en la capa de la salida el cual depende del número de componentes del vector de salida o patrones objetivo y funciones de transferencia requeridas en cada capa, con base en la topología escogida se asignan valores iniciales a cada uno de los parámetros que conforma la red.

Es importante recalcar que no existe una técnica para determinar el número de capas ocultas, ni el número de neuronas que debe contener cada una de ellas para un problema específico, esta elección es determinada por la experiencia del diseñador, el cual debe cumplir con las limitaciones de tipo computacional.

Cada patrón de entrenamiento se propaga a través de la red y sus parámetros para producir una respuesta en la capa de salida, la cual se compara con los patrones objetivo o salidas deseadas para calcular el error en el aprendizaje, este error marca el camino mas adecuado para la actualización de los pesos y ganancias que al final del entrenamiento producirán una respuesta satisfactoria a todos los patrones de entrenamiento, esto se logra minimizando el error medio cuadrático en cada iteración del proceso de aprendizaje.

La deducción matemática de este procedimiento se realizará para una red con una capa de entrada, una capa oculta y una capa de salida y luego se generalizará para redes que tengan más de una capa oculta.

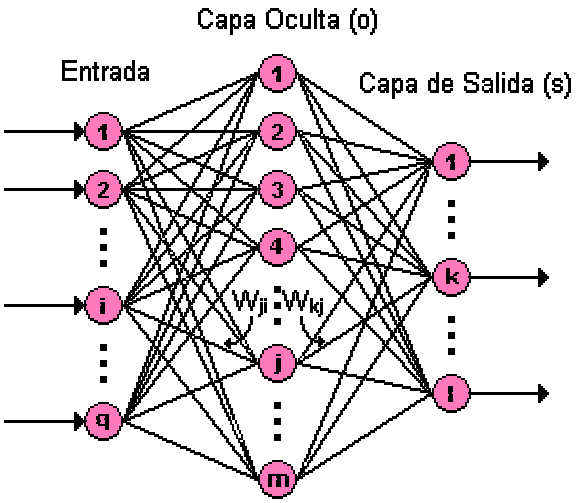

Figura 2.3.3 Disposición de una red sencilla de 3 capas

Es importante aclarar que en la figura 2.3.3

q: equivale al número de componentes el vector de entrada.

m: número de neuronas de la capa oculta

l: número de neuronas de la capa de salida

Para iniciar el entrenamiento se le presenta a la red un patrón de entrenamiento, el cual tiene q componentes como se describe en la ecuación (2.3.3)

$$P = \begin{bmatrix} p_1 \\ p_2 \\ \vdots \\ p_i \\ \vdots \\ p_q \end{bmatrix} \qquad (2.3.3)$$

Cuando se le presenta a la red una patrón de entrenamiento, este se propaga a través de las conexiones existentes produciendo una entrada neta n en cada una las neuronas de la siguiente capa, la entrada neta a la neurona j de la siguiente capa debido a la presencia de un patrón de entrenamiento en la entrada esta dada por la ecuación (2.3.4), nótese que la entrada neta es el valor justo antes de pasar por la función de transferencia

$$n_j^o = \sum_{i=1}^{q} W_{ji}^o p_i + b_j^o \qquad (2.3.4)$$

W^o_{ji}: Peso que une la componente i de la entrada con la neurona j de primera capa oculta

p_i: *Componente* i del vector p que contiene el patrón de entrenamiento de q componentes

b^o_j: Ganancia de la neurona j de la capa oculta

Donde el superíndice (^{o}) representa la capa a la que pertenece cada parámetro, es este caso la capa oculta.

Cada una de las neuronas de la capa oculta tiene como salida a^o_j que está dada por la ecuación (2.3.5)

$$a^o_j = f^o\left(\sum_{i=1}^{q} W^o_{ji}\, p_i + b^o_j\right) \qquad (2.3.5)$$

f^o: Función de transferencia de las neuronas de la capa oculta

Las salidas a^o_j de las neuronas de la capa oculta (de l componentes) son las entradas a los pesos de conexión de la capa de salida, $a^o_k \Rightarrow n^s_k$ este comportamiento esta descrito por la ecuación (2.3.6)

$$n^s_k = \sum_{j=1}^{m} W^s_{kj} a^o_j + b^s_k \qquad (2.3.6)$$

W^s_{kj}: Peso que une la neurona j de la capa oculta con la neurona k de la capa de salida, la cual cuenta con s neuronas

a^o_j: Salida de la neurona j de la capa oculta, la cual cuenta con m neuronas.

b^s_k: Ganancia de la neurona k de la capa de salida.

n^s_k: Entrada neta a la neurona k de la capa de salida

La red produce una salida final descrita por la ecuación (2.3.7)

$$a^s_k = f^s\left(n^s_k\right) \tag{2.3.7}$$

f^s: Función de transferencia de las neuronas de la capa de salida

Reemplazando (2.3.6) en (2.3.7) se obtiene la salida de la red en función de la entrada neta y de los pesos de conexión con la ultima capa oculta

$$a^s_k = f^s\left(\sum_{j=1}^{m} W^s_{kj} a^o_j + b^s_k\right) \tag{2.3.8}$$

La salida de la red de cada neurona a^s_k se compara con la salida deseada t_k para calcular el error en cada unidad de salida (2.3.9)

$$\delta_k = \left(t_k - a^s_k\right) \tag{2.3.9}$$

El error debido a cada patrón p propagado está dado por (2.3.11)

$$ep^2 = \frac{1}{2}\sum_{k=1}^{s}(\delta_k)^2 \qquad (2.3.10)$$

ep^2: Error medio cuadrático para cada patrón de entrada p

δ_k: Error en la neurona k de la capa de salida con l neuronas

Este proceso se repite para el número total de patrones de entrenamiento (r), para un proceso de aprendizaje exitoso el objetivo del algoritmo es actualizar todos los pesos y ganancias de la red minimizando el error medio cuadrático total descrito en (2.3.11)

$$e^2 = \sum_{p=1}^{r} ep^2 \qquad (2.3.11)$$

e^2: Error total en el proceso de aprendizaje en una iteración luego de haber presentado a la red los r patrones de entrenamiento

El error que genera una red neuronal en función de sus pesos, genera un espacio de n dimensiones, donde n es el número de pesos de conexión de la red, al evaluar el gradiente del error en un punto de esta superficie se obtendrá la dirección en la cual la función del error tendrá un mayor crecimiento, como el objetivo del proceso de aprendizaje es minimizar el error debe tomarse la dirección negativa del gradiente para obtener el mayor decremento del error y de esta forma

su minimización, condición requerida para realizar la actualización de la matriz de pesos en el algoritmo Backpropagation:

$$W_{k+1} = W_k - \alpha \nabla ep^2 \qquad (2.3.12)$$

El gradiente negativo de ep^2 se denotará como $-\nabla ep^2$ y se calcula como la derivada del error respecto a todos los pesos de la red

En la capa de salida el gradiente negativo del error con respecto a los pesos es:

$$-\frac{\partial ep^2}{\partial W_{kj}^s} = -\frac{\partial}{\partial W_{kj}^s}\left(\frac{1}{2}\sum_{k=1}^{l}\left(t_k - a_k^s\right)^2\right) = \left(t_k - a_k^s\right) \times \frac{\partial a_k^s}{\partial W_{kj}^s} \qquad (2.3.13)$$

$-\frac{\partial ep^2}{\partial W_{kj}^s}$: Componente del gradiente $-\nabla ep^2$ respecto al peso de la conexión de la neurona de la capa de salida y la neurona j de la capa oculta W_{kj}^s

$\frac{\partial a_k^m}{\partial W_{kj}^m}$: Derivada de la salida de la neurona k de la capa de salida respecto, al peso $W^s{}_{kj}$

Para calcular $\frac{\partial a_k^s}{\partial W_{kj}^s}$ se debe utilizar la regla de la cadena, pues el error no es una función explícita de los pesos de la red, de la ecuación (2.3.7) puede verse que la salida de la red $a^s{}_k$ esta explícitamente en función de $n^s{}_k$ y de la ecuación (2.3.6)

puede verse que $n^s{}_k$ esta explícitamente en función de $W^s{}_{kj}$, considerando esto se genera la ecuación (2.3.13)

$$\frac{\partial a_k^s}{\partial W_{kj}^s} = \frac{\partial a_k^s}{\partial n_k^s} \times \frac{\partial n_k^s}{\partial W_{kj}^s} \tag{2.3.14}$$

Tomando la ecuación (2.3.14) y reemplazándola en la ecuación (2.3.13) se obtiene,

$$-\frac{\partial ep^2}{\partial W_{kj}^s} = \left(t_k - a_k^s\right) \times \frac{\partial a_k^s}{\partial n_k^s} \times \frac{\partial n_k^s}{\partial W_{kj}^s} \tag{2.3.15}$$

$\frac{\partial n_k^s}{\partial W_{kj}^s}$: Derivada de la entrada neta a la neurona k de la capa de salida respecto a los pesos de la conexión entre las neuronas de la última capa oculta y la capa de salida

$\frac{\partial a_k^s}{\partial n_k^s}$: Derivada de la salida de la neurona k de la capa de salida respecto a su entrada neta.

Reemplazando en la ecuación (2.3.15) las derivadas de las ecuaciones (2.3.6) y (2.3.7) se obtiene

$$-\frac{\partial ep^2}{\partial W_{kj}^s} = \left(t_k - a_k^s\right) \times f'^s\left(n_k^s\right) \times a_j^o \qquad (2.3.16)$$

Como se observa en la ecuación (2.3.16) las funciones de transferencia utilizadas en este tipo de red deben ser continuas para que su derivada exista en todo el intervalo, ya que el término $f'^s(n^s_k)$ es requerido para el cálculo del error.

Las funciones de transferencia f más utilizadas y sus respectivas derivadas son las siguientes:

logsig: $f(n) = \frac{1}{1+e^{-n}}$ $\quad f'(n) = f(n)(1-f(n))$ $\quad f'(n) = a(1-a)$ (2.3.17)

tansig: $f(n) = \frac{e^n - e^{-n}}{e^n + e^{-n}}$ $\quad f'(n) = 1-(f(n))^2$ $\quad f'(n) = \left(1-a^2\right)$ (2.3.18)

purelin: $f(n) = n$ $\quad f'(n) = 1$ (2.3.19)

De la ecuación (2.3.16), los términos del error para las neuronas de la capa de salida están dados por la ecuación (2.3.20), la cual se le denomina comúnmente sensitividad de la capa de salida.

$$\delta_k^s = \left(t_k - a_k^s\right) f'^s\left(n_k^s\right) \qquad (2.3.20)$$

Este algoritmo se denomina Backpropagation o de propagación inversa debido a que el error se propaga de manera inversa al funcionamiento normal de la red, de esta forma, el algoritmo encuentra el error en el proceso de aprendizaje desde las capas más internas hasta llegar a la entrada; con base en el cálculo de este error se actualizan los pesos y ganancias de cada capa.

Después de conocer (2.3.20) se procede a encontrar el error en la capa oculta el cual esta dado por:

$$-\frac{\partial ep^2}{\partial W_{ji}^o} = -\frac{\partial}{\partial W_{ji}^o}\left(\frac{1}{2}\sum_{k=1}^{l}\left(t_k - a_k^s\right)^2\right) = \sum_{k=1}^{l}\left(t_k - a_k^s\right)\times\frac{\partial a_k^s}{\partial W_{ji}^o} \qquad (2.3.21)$$

Para calcular el último término de la ecuación (2.3.21) se debe aplicar la regla de la cadena en varias ocasiones como se observa en la ecuación (2.3.22) puesto que la salida de la red no es una función explícita de los pesos de la conexión entre la capa de entrada y la capa oculta

$$\frac{\partial a_k^s}{\partial W_{ji}^o} = \frac{\partial a_k^s}{\partial n_k^s}\times\frac{\partial n_k^s}{\partial a_k^o}\times\frac{\partial a_k^o}{\partial n_j^o}\times\frac{\partial n_j^o}{\partial W_{ji}^o} \qquad (2.3.22)$$

Todas los términos de la ecuación (2.3.23) son derivados respecto a variables de las que dependan explícitamente, reemplazando (2.3.22) en (2.3.21) tenemos:

$$-\frac{\partial ep^2}{\partial W_{ji}^o} = \sum_{k=1}^{l} \left(t_k - a_k^s\right) \times \frac{\partial a_k^s}{\partial n_k^s} \times \frac{\partial n_k^s}{\partial a_k^o} \times \frac{\partial a_k^o}{\partial n_j^o} \times \frac{\partial n_j^o}{\partial W_{ji}^o} \qquad (2.3.23)$$

Tomando las derivadas de las ecuaciones (2.3.4) (2.3.5) (2.3.6) (2.3.7) y reemplazándolas en la ecuación (2.3.23) se obtiene la expresión del gradiente del error en la capa oculta

$$-\frac{\partial ep^2}{\partial W_{ji}^o} = \sum_{k=1}^{l} \left(t_k - a_k^s\right) \times f'^s\left(n_k^s\right) \times \boldsymbol{W}_{kj}^s \times f'^o\left(n_j^o\right) \times \boldsymbol{p}_i \qquad (2.3.24)$$

Reemplazando la ecuación (2.3.20) en la ecuación (2.3.24) se tiene:

$$-\frac{\partial ep^2}{\partial W_{ji}^o} = \sum_{k=1}^{l} \delta_k^s \times \boldsymbol{W}_{kj}^s \times f'^o\left(n_j^o\right) \times \boldsymbol{p}_i \qquad (2.3.25)$$

Los términos del error para cada neurona de la capa oculta están dados por la ecuación (2.3.26), este término también se denomina sensitividad de la capa oculta

$$\delta_j^o = f'^o\left(n_j^o\right) \times \sum_{k=1}^{l} \delta_k^s \boldsymbol{W}_{kj}^s \qquad (2.3.26)$$

Luego de encontrar el valor del gradiente del error se procede a actualizar los pesos de todas las capas empezando por la de salida, para la capa de salida la actualización de pesos y ganancias está dada por (2.3.27) y (2.3.28).

$$\boldsymbol{W}_{kj}(t+1) = \boldsymbol{W}_{kj}(t) - 2\alpha\delta_k^s \tag{2.3.27}$$

$$\boldsymbol{b}_k(t+1) = \boldsymbol{b}_k(t) - 2\alpha\delta_k^s \tag{2.3.28}$$

α : Rata de aprendizaje que varía entre 0 y 1 dependiendo de las características del problema a solucionar.

Luego de actualizar los pesos y ganancias de la capa de salida se procede a actualizar los pesos y ganancias de la capa oculta mediante las ecuaciones (2.3.29) y (2.3.30)

$$W_{ji}(t+1) = W_{ji}(t) - 2\alpha\delta_j^o p_i \tag{2.3.29}$$

$$b_j(t+1) = b_j(t) - 2\alpha\delta_j^o \tag{2.3.30}$$

Esta deducción fue realizada para una red de tres capas, si se requiere realizar el análisis para una red con dos o más capas ocultas, las expresiones pueden derivarse de la ecuación (2.3.26) donde los términos que se encuentran dentro de la sumatoria pertenecen a la capa inmediatamente superior, este algoritmo es

conocido como la regla Delta Generalizada desarrollada por Rumelhart D [32], la cual es una extensión de la regla delta desarrollada por Widrow [34] en 1930

Algunos autores denotan las sensitividades de las capas por la letra $\boldsymbol{S}$, reescribiendo las ecuaciones (2.3.20) y (2.3.26) con esta notación se obtienen las ecuaciones (2.3.31) y (2.3.32)

$$\boldsymbol{S}^M = -2\dot{f}^M\left(\boldsymbol{n}^M\right)\left(\boldsymbol{t}-\boldsymbol{a}\right) \qquad (2.3.31)$$

$$\boldsymbol{s}^m \equiv \dot{f}^m\left(\boldsymbol{n}^m\right)\left(\boldsymbol{W}^{m+1}\right)^T \boldsymbol{s}^{m+1}, \; para\; m = M-1,....,2,1. \qquad (2.3.32)$$

En la ecuación (2.3.31) M representa la última capa y S^M la sensitividad para esta capa, la ecuación (2.3.32) expresa el cálculo de la sensitividad capa por capa comenzando desde la última capa oculta, cada uno de estos términos involucra que el término para la sensitividad de la capa siguiente ya esté calculado.

Como se ve el algoritmo Backpropagation utiliza la misma técnica de aproximación en pasos descendientes que emplea el algoritmo LMS, la única complicación está en el cálculo del gradiente, el cual es un término indispensable para realizar la propagación de la sensitividad.

En las técnicas de gradiente descendiente es conveniente avanzar por la superficie de error con incrementos pequeños de los pesos; esto se debe a que tenemos una información local de la superficie y no se sabe lo lejos o lo cerca que

se está del punto mínimo, con incrementos grandes, se corre el riesgo de pasar por encima del punto mínimo, con incrementos pequeños, aunque se tarde más en llegar, se evita que esto ocurra. El elegir un incremento adecuado influye en la velocidad de convergencia del algoritmo, esta velocidad se controla a través de la rata de aprendizaje α, la que por lo general se escoge como un número pequeño, para asegurar que la red encuentre una solución. Un valor pequeño de α significa que la red tendrá que hacer un gran número de iteraciones, si se toma un valor muy grande, los cambios en los pesos serán muy grandes, avanzando muy rápidamente por la superficie de error, con el riesgo de saltar el valor mínimo del error y estar oscilando alrededor de él, pero sin poder alcanzarlo.

Es recomendable aumentar el valor de α a medida que disminuye el error de la red durante la fase de entrenamiento, para garantizar así una rápida convergencia, teniendo la precaución de no tomar valores demasiado grandes que hagan que la red oscile alejándose demasiado del valor mínimo. Algo importante que debe tenerse en cuenta, es la posibilidad de convergencia hacia alguno de los mínimos locales que pueden existir en la superficie del error del espacio de pesos como se ve en la figura 2.3.4.

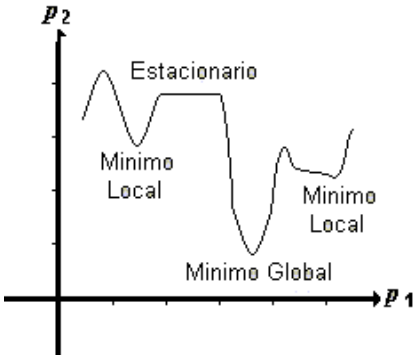

Figura 2.3.4 Superficie típica de error

En el desarrollo matemático que se ha realizado para llegar al algoritmo Backpropagation, no se asegura en ningún momento que el mínimo que se encuentre sea global, una vez la red se asiente en un mínimo sea local o global cesa el aprendizaje, aunque el error siga siendo alto. En todo caso, si la solución es admisible desde el punto de vista del error, no importa si el mínimo es local o global o si se ha detenido en algún momento previo a alcanzar un verdadero mínimo.

Para ilustrar el cálculo de cada uno de estos términos, utilizamos el algoritmo Backpropagation, para aproximar la siguiente función:

$$t = sin\frac{\pi}{4}p \quad para\ el\ \text{int}ervalo\ -2 \le p \le 2 \qquad (2.3.33)$$

La función se ha restringido al intervalo entre –2 y 2 para conservarla dentro de límites observables, como se observa en la figura 2.3.5

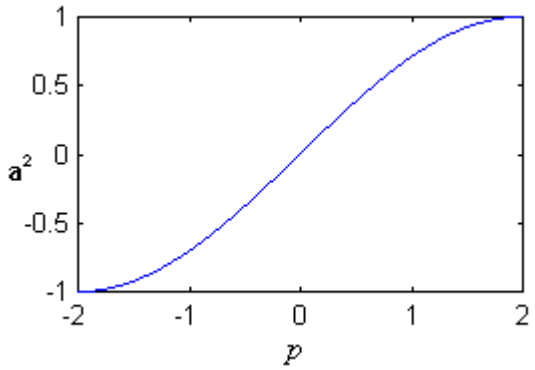

Figura 2.3.5 Intervalo de la función t

La configuración escogida para la red corresponde a una red 1:2:1 según la notación definida con anterioridad, es decir una entrada, dos neuronas en la capa oculta y una salida; esta estructura se visualiza en la figura 2.3.6

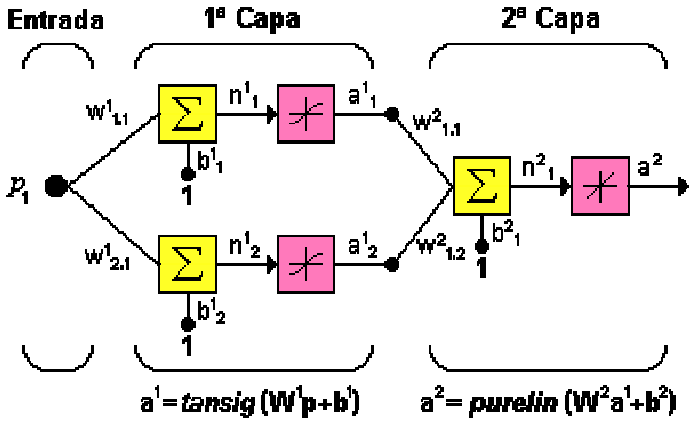

Figura 2.3.6 Red utilizada para aproximar la función

Como se observa la salida de la red para la primera capa está dada por

$$a^1 = tansig(W^1 p^T + b) \qquad (2.3.34)$$

Las redes tipo Backpropagation utilizan principalmente dos funciones de transferencia en la primera capa: *logsig,* cuando el rango de la función es siempre positivo y *tansig* como en este caso, cuando se le permite a la función oscilar entre valores positivos y negativos limitados en el intervalo –1, 1.

La salida de la segunda capa está determinada siempre por la función de transferencia *purelin,* la cual reproduce exactamente el valor resultante después de la sumatoria.

$$a^2 = purelin(W^2 * a^1 + b^2) \qquad (2.3.35)$$

Al evaluar la ecuación (2.3.33) en los diferentes patrones de entrenamiento, se obtienen los valores de las entradas y sus salidas asociadas, ya que como se dijo antes la red Backpropagation es una red de aprendizaje supervisado. Es importante destacar, que no es estrictamente necesario el conocimiento de la función a aproximar, basta con conocer la respuesta a una entrada dada, o un registro estadístico de salidas para modelar el comportamiento del sistema, limitando el problema a la realización de pruebas a una caja negra.

Los parámetros de entrada y sus valores de salida asociados, se observan en la tabla 2.3.1

	1	2	3	4	5	6
p	-2	-1,2	0,4	0,4	1,2	2
t	-1	-0,81	-0,31	0,309	0,809	1

Tabla 2.3.1 Set de entrenamiento de la red

Los valores iniciales para la matriz de pesos y el vector de ganancias de la red se escogieron en forma aleatoria así:

$$W^1 = \begin{bmatrix} -0.2 \\ 0.5 \end{bmatrix}, \quad b^1 = \begin{bmatrix} 0.7 \\ -0.2 \end{bmatrix}, \quad W^2 = [0.1 \quad 0.3], \quad b^2 = [0.8], \quad \alpha = 0.1$$

Para el proceso de cálculo, se le presenta a la red el parámetro $p_{1,}$ de esta forma la primera iteración es como sigue

$$a^1 = tansig\left(\begin{bmatrix} -0.2 \\ 0.5 \end{bmatrix}(-0.2) + \begin{bmatrix} 0.7 \\ -0.2 \end{bmatrix}\right) = \begin{bmatrix} 0.8 \\ -0.83 \end{bmatrix}$$

$$a^2 = [0.1 \quad 0.3]\begin{bmatrix} 0.8 \\ -0.83 \end{bmatrix} + [0.8] = 0.63$$

$$e = t - a = -1 - (0.63) = -1.63$$

Como se esperaba la primera iteración no ha sido suficiente, para aproximar la función correctamente, así que se calculará la sensitividad para iniciar el proceso de actualización de los valores de los pesos y las ganancias de la red.

Los valores de las derivadas del error medio cuadrático son:

$$\overline{f^1}(n) = 1 - a_1^2$$

$$\overline{f^2}(n) = 1$$

Y las sensitividades, empezando desde la última hasta la primera capa,

$$s^2 = -2(1)\,(-1.63) = 3.26$$

$$s^1 = \begin{bmatrix} \left(1 - 0.8^2\right) & 0 \\ 0 & \left(1 - 0.83^2\right) \end{bmatrix}\begin{bmatrix} 0.1 \\ 0.3 \end{bmatrix}(3.26) = \begin{bmatrix} 0.1171 \\ 0.2983 \end{bmatrix}$$

Con estos valores, y de acuerdo a la regla de actualización descrita anteriormente, los nuevos parámetros de la red son:

$$W^2(1) = [0.1 \quad 0.3] - 0.1(3.26)[0.8 \quad -0.83] = [-0.161 \quad 0.5718]$$

$$b^2(1) = [0.8] - 0.1(3.26) = 0.474$$

$$W^1(1) = \begin{bmatrix} -0.2 \\ 0.5 \end{bmatrix} - 0.1\begin{bmatrix} 0.1171 \\ 0.2983 \end{bmatrix}(-2) = \begin{bmatrix} -0.1766 \\ 0.5957 \end{bmatrix}$$

$$b^1(1) = \begin{bmatrix} 0.7 \\ -0.2 \end{bmatrix} - 0.1\begin{bmatrix} 0.1171 \\ 0.2983 \end{bmatrix} = \begin{bmatrix} 0.688 \\ -0.2298 \end{bmatrix}$$

Con esto se completa la primera iteración, y el algoritmo queda listo para presentar a la red el siguiente patrón y continuar el proceso iterativo hasta obtener un valor de tolerancia aceptable para el error.

En 1989 Funahashi [15] demostró matemáticamente que una red neuronal multicapa puede aproximar cualquier función no lineal o mapa lineal multivariable, $f(x) = R^n \rightarrow R$ Este teorema es de existencia, pues prueba que la red existe pero no indica como construirla y tampoco garantiza que la red aprenderá función.

El algoritmo Backpropagation es fácil de implementar, y tiene la flexibilidad de adaptarse para aproximar cualquier función, siendo una de las redes multicapa más potentes; esta característica ha convertido a esta red en una de las más ampliamente utilizadas y ha llevado al desarrollo de nuevas técnicas que permitan su mejoramiento. Dentro de estas técnicas se encuentran dos métodos heurísticos y dos métodos basados en algoritmos de optimización numérica.

Dentro de los métodos heurísticos tenemos:

2.3.3.1 Red Backpropagation con momentum [30]. Esta modificación está basada en la observación de la última sección de la gráfica del error medio cuadrático en el proceso de convergencia típico para una red Backpropagation; este proceso puede verse en la figura 2.3.7 en la cual se nota la caída brusca del error en la iteración para la cual alcanza convergencia

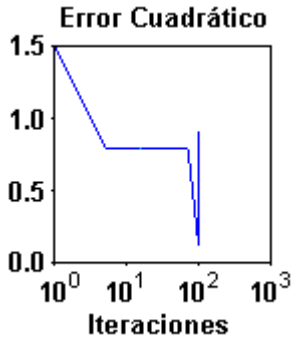

Figura 2.3.7 Comportamiento típico del proceso de convergencia para una red Backpropagation

Este comportamiento puede causar oscilaciones no deseadas, por lo que es conveniente suavizar esta sección de la gráfica incorporando un filtro pasa-bajo al sistema. Para ilustrar el efecto positivo del filtro en el proceso de convergencia, se analizará el siguiente filtro de primer orden:

$$y(k) = \gamma y(k-1) + (1-\gamma)w(k) \qquad (2.3.36)$$

Donde $w(k)$ es la entrada al filtro, $y(k)$ su salida y γ es el coeficiente de momentum que está en el intervalo: $0 \le \gamma \le 1$

El efecto del filtro puede observase en la figura 2.3.8, en la cual se tomó como entrada al filtro la función:

$$w(k) = 1 + \text{sen}\left(\frac{2\pi k}{16}\right) \qquad (2.3.37)$$

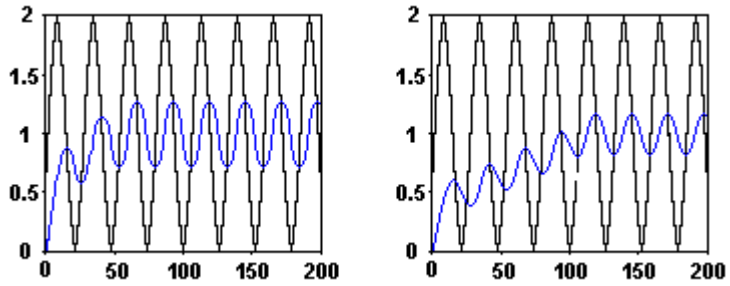

Figura 2.3.8 Efecto del coeficiente de momentum

El coeficiente de momentum se asumió $\gamma = 0.9$ para la gráfica de la izquierda y $\gamma = 0.98$ para la gráfica de la derecha. De esta figura puede notarse como la oscilación es menor a la salida del filtro, la oscilación se reduce a medida que γ se decrementa, el promedio de la salida del filtro es el mismo que el promedio de entrada al filtro aunque mientras γ sea incrementado la salida del filtro será más lenta.

Recordando los parámetros de actualización empleados por la red Backpropagation tradicional:

$$\Delta \boldsymbol{W}^m(k) = -\alpha s^m (a^{m-1})^T \qquad (2.3.38)$$

$$\Delta \boldsymbol{b}^m(k) = -\alpha \boldsymbol{s}^m \qquad (2.3.39)$$

Al adicionar el filtro con momentum a este algoritmo de actualización, se obtienen las siguientes ecuaciones que representan el algoritmo Backpropagation con momentum:

$$\Delta \boldsymbol{W}^m(k) = \gamma \Delta \boldsymbol{W}^m(k-1) - (1-\gamma)\alpha \boldsymbol{s}^m (\boldsymbol{a}^{m-1})^T \qquad (2.3.40)$$

$$\Delta \boldsymbol{b}^m(k) = \gamma \Delta \boldsymbol{b}^m(k-1) - (1-\gamma) - \alpha \boldsymbol{s}^m \qquad (2.3.41)$$

Este algoritmo, hace que la convergencia sea estable e incluso más rápida, además permite utilizar una rata de aprendizaje alta.

La figura 2.3.9 referencia el comportamiento del algoritmo con momentum en el punto de convergencia:

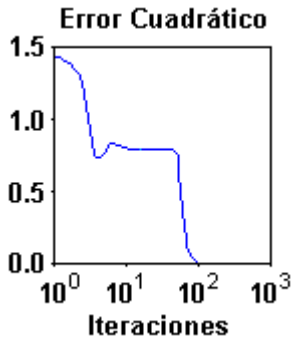

Figura 2.3.9 Trayectoria de convergencia con momentum

2.3.3.2 Red Backpropagation con rata de aprendizaje variable [30]: Del análisis de la sección 2.3.3 se vio que $\nabla \boldsymbol{e(x)}$ es el gradiente del error, de igual

forma se definirá $\nabla e^2(x)$ como la Hessiana de la función de error, donde x representa las variables de las cuales depende el error (pesos y ganancias), esta matriz es siempre de la forma:

$$\nabla^2 e(x) = \begin{bmatrix} \frac{\partial^2 e(x)}{\partial x_1^2} & \frac{\partial^2 e(x)}{\partial x_1 \partial x_2} & \cdots \frac{\partial^2 e(x)}{\partial x_1 \partial x_n} \\ \frac{\partial^2 e(x)}{\partial x_2 \partial x_1} & \frac{\partial^2 e(x)}{\partial x_2^2} & \cdots \frac{\partial^2 e(x)}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \cdots \vdots \\ \frac{\partial^2 e(x)}{\partial x_n \partial x_1} & \frac{\partial^2 e(x)}{\partial x_n \partial x_2} & \cdots \frac{\partial^2 e(x)}{\partial x_n^2} \end{bmatrix} \qquad (2.3.42)$$

La superficie del error medio cuadrático para redes de una sola capa es siempre una función cuadrática y la matriz Hessiana es por tanto constante, ésto lleva a que la máxima rata de aprendizaje estable para el algoritmo de pasos descendientes sea el máximo valor propio de la matriz Hessiana dividido 2,HBD[]. Para una red multicapa la superficie del error no es una función cuadrática, su forma es diferente para diferentes regiones del espacio, la velocidad de convergencia puede incrementarse por la variación de la rata de aprendizaje en cada parte de la superficie del error, sin sobrepasar el valor máximo para aprendizaje estable definido anteriormente.

Existen varias técnicas para modificar la rata de aprendizaje; este algoritmo emplea un procedimiento mediante el cual la rata de aprendizaje varia de acuerdo al rendimiento que va presentando el algoritmo en cada punto; si el error disminuye vamos por el camino correcto y se puede ir más rápido incrementando

la rata de aprendizaje, si el error aumenta, es necesario decrementar la rata de aprendizaje; el criterio de variación de α debe estar en concordancia con las siguientes reglas heurísticas:

1. Si el error cuadrático de todos los parámetros del set de entrenamiento se incrementa en un porcentaje ζ típicamente entre 1% y 5%, después de la actualización de los pesos, esa actualización es descartada, la rata de aprendizaje se multiplica por un factor $0 < \rho < 1$, y el coeficiente de momentum es fijado en cero.
2. Si el error cuadrático se decrementa después de la actualización de los pesos, esa actualización es aceptada y la rata de aprendizaje es multiplicada por un factor $\eta > 1$. Si γ había sido previamente puesto en cero, se retorna a su valor original.
3. Si el error cuadrático se incrementa en un valor menor a ζ, los pesos actualizados son aceptados, pero la rata de aprendizaje y el coeficiente de momentum no son cambiados.

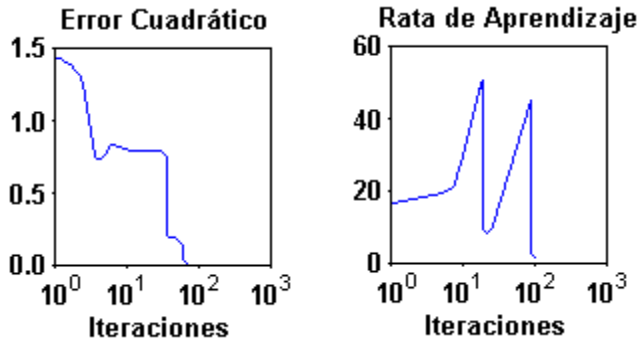

Figura 2.3.10 Característica de convergencia para una rata de aprendizaje variable

La figura 2.3.10, muestra la trayectoria de la rata de aprendizaje para este algoritmo en comparación con la característica de convergencia

Existen muchas variaciones de este algoritmo, por ejemplo Jacobs[22] propuso la regla delta-bar-delta, en la cual cada uno de los parámetros de la red, (pesos y ganancias) tenían su propia rata de aprendizaje. El algoritmo incrementa la rata de aprendizaje para un parámetro de la red si el parámetro escogido, ha estado en la misma dirección para varias iteraciones; si la dirección del parámetro escogido cambia, entonces la rata de aprendizaje es reducida. Los algoritmos Backpropagation con momentum y con rata de aprendizaje variable son los dos métodos heurísticos más utilizados para modificar el algoritmo Backpropagation tradicional. Estas modificaciones garantizan rápida convergencia para algunos problemas, sin embargo presentan dos problemas principales: primero, requieren de un gran número de parámetros (ζ, ρ, γ), los que la mayoría de las veces se definen por un método de ensayo y error de acuerdo a la experiencia del investigador, mientras que el algoritmo tradicional, sólo requiere definir la rata de aprendizaje; segundo, estas modificaciones pueden llevar a que el algoritmo nunca converja y se torne oscilante para problemas muy complejos.

Como se mencionó antes, existen también métodos de modificación basados en técnicas de optimización numérica, de esta clase de modificaciones se destacarán las más sobresalientes; es importante recalcar que estos métodos requieren una matemática más exigente, que el simple del dominio de cálculo diferencial.

2.3.3.3 Método del Gradiente Conjugado [30]. Este algoritmo no involucra el cálculo de las segundas derivadas de las variables y converge al mínimo de la función cuadrática en un número finito de iteraciones. El algoritmo del gradiente conjugado, sin aplicarlo aún al algoritmo de propagación inversa consiste en:

1. Seleccionar la dirección de p_0 , la condición inicial, en el sentido negativo del gradiente:

$$p_0 = -g_0 \tag{2.3.43}$$

Donde

$$g(k) \equiv \nabla e(x)\big|_{x=x_k} \tag{2.3.44}$$

2. Seleccionar la rata de aprendizaje α_k para minimizar la función a lo largo de la dirección

$$x_{k+1} = x_k + \alpha_k \boldsymbol{p_k} \tag{2.3.45}$$

3. Seleccionar la dirección siguiente de acuerdo a la ecuación

$$\boldsymbol{p}_k = -\boldsymbol{g}_k + \beta_k \boldsymbol{p}_{k-1} \tag{2.3.46}$$

con

$$\beta_k = \frac{\Delta \boldsymbol{g}_{k-1}^T \boldsymbol{g}_k}{\Delta \boldsymbol{g}_{k-1}^T \boldsymbol{p}_{k-1}} \; o \quad \beta_k = \frac{\boldsymbol{g}_k^T \boldsymbol{g}_k}{\boldsymbol{g}_{k-1}^T \boldsymbol{g}_{k-1}} \tag{2.3.47}$$

4. Si el algoritmo en este punto aún no ha convergido, se regresa al numeral 2

Este algoritmo no puede ser aplicado directamente a una red neural porque el error no es una función cuadrática; lo que afecta al algoritmo en dos formas, primero no es hábil para minimizar la función a lo largo de una línea como es requerido en el paso 2; segundo, el error mínimo no será alcanzado normalmente en un número finito de pasos y por esto el algoritmo necesitará ser inicializado después de un número determinado de iteraciones.

A pesar de estas complicaciones, esta modificación del algoritmo Backpropagation converge en muy pocas iteraciones, y es incluso uno de los algoritmos más rápidos para redes multicapa, como puede notarse en la figura 2.3.11

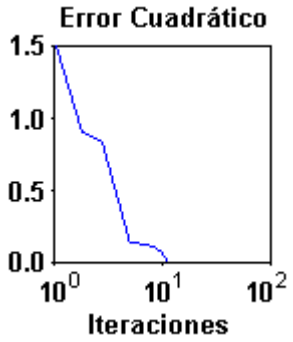

Figura 2.3.11 Trayectoria del Gradiente Conjugado

2.3.3.4 Algoritmo de Levenberg – Marquardt [30]. Este algoritmo es una modificación del método de Newton, el que fue diseñado para minimizar funciones que sean la suma de los cuadrados de otras funciones no lineales; es por ello que el algoritmo de Levenberg - Marquardt, tiene un excelente desempeño en el

entrenamiento de redes neuronales donde el rendimiento de la red esté determinado por el error medio cuadrático.

El método de Newton para optimizar el rendimiento $e(x)$ es:

$$\boldsymbol{X}_{k+1} = \boldsymbol{X}_k - \boldsymbol{A}_k^{-1}\boldsymbol{g}_k \tag{2.3.48}$$

$$\boldsymbol{A}_k \equiv \nabla^2 \boldsymbol{e(x)}\Big|_{x=x_k} \qquad \boldsymbol{g}_k \equiv \nabla \boldsymbol{e(x)}\Big|_{x=x_k} \tag{2.3.49}$$

Si asumimos que $e(x)$ es una suma de funciones cuadráticas:

$$\boldsymbol{e(x)} = \sum_{i=1}^{n} v_i^2(\boldsymbol{x}) = \boldsymbol{v^T(x)v(x)} \tag{2.3.50}$$

El gradiente puede ser escrito entonces en forma matricial:

$$\nabla \boldsymbol{e(x)} = 2\boldsymbol{J^T(x)v(x)} \tag{2.3.51}$$

Donde $\boldsymbol{J(x)}$ es la matriz Jacobiana.

Ajustando el método de Newton, obtenemos el algoritmo de Levenberg Marquardt

$$\boldsymbol{x_{k+1}} = \boldsymbol{x_k} - \left[\boldsymbol{J^T(x_k)J(x_k)} + \mu_k \boldsymbol{I}\right]^{-1} \boldsymbol{J^T(x_k)v(x_k)} \tag{2.3.52}$$

o determinando directamente el incremento:

$$\Delta \boldsymbol{x_k} = -\left[\boldsymbol{J^T(x_k)J(x_k)} + \mu_k \boldsymbol{I}\right]^{-1} \boldsymbol{J^T(x_k)v(x_k)} \qquad (2.3.53)$$

La nueva constante μ determina la tendencia el algoritmo, cuando μ_k se incrementa, este algoritmo se aproxima al algoritmo de pasos descendientes para ratas de aprendizaje muy pequeñas; cuando μ_k se decrementa este algoritmo se convierte en el método de Gauss - Newton

El algoritmo comienza con un valor pequeño para μ_k, por lo general 0.01, si en ese paso no se alcanza el valor para *e(x)* entonces el paso es repetido con μ_k multiplicado por un factor $\vartheta > 1$. Si se ha escogido un valor pequeño de paso en la dirección de paso descendiente, *e(x)* debería decrecer. Si un paso produce un pequeño valor para *e(x),* entonces el algoritmo tiende al método de Gauss - Newton, el que se supone garantiza una rápida convergencia. Este algoritmo genera un compromiso entre la velocidad del método de Gauss-Newton y la garantía de convergencia del método de paso descendiente.

Los elementos de la matriz Jacobiana necesarios en el algoritmo de Levenberg-Marquardt son de este estilo:

$$[\boldsymbol{J}]_{h,l} = \frac{\partial e_{k,q}}{\partial x_1} \tag{2.3.54}$$

Donde x es el vector de parámetros de la red, que tiene la siguiente forma:

$$\boldsymbol{x}^T = [x_1, x_2, ...x_n] = \left\lfloor w^1_{1,1}, w^1_{1,2}, \ldots, w^1_{S^1,R}, b^1_1, .., b^1_{S^1} \right\rfloor \tag{2.3.55}$$

Para utilizar este algoritmo en las aplicaciones para redes multicapa, se redefinirá el término sensitividad de forma que sea más simple hallarlo en cada iteración.

$$s^m_{i,h} \equiv \frac{\partial e_{k,q}}{\partial n^m_{i,q}} \tag{2.3.56}$$

Donde $h=(q-1)S^M + k$

Con la sensitividad definida de esta manera, los términos de la matriz Jacobiana pueden ser calculados más fácilmente:

$$[\boldsymbol{J}]_{h,l} = \frac{\partial e_{k,q}}{\partial w^m_{i,j}} = \frac{\partial e_{k,q}}{\partial n^m_{i,q}} * \frac{\partial n_{i,q}}{\partial w^m_{i,j}} = s^m_{i,h} * \frac{\partial n_{i,q}}{\partial w^m_{i,j}} = s^m_{i,h} * a^{m-1}_{j,q} \tag{2.3.57}$$

o para las ganancias:

$$[\boldsymbol{J}]_{h,l} = \frac{\partial e_{k,q}}{\partial b_i^m} = \frac{\partial e_{k,q}}{\partial n_{i,q}^m} * \frac{\partial n_{i,q}}{\partial b_i^m} = s_{i,h}^m * \frac{\partial n_{i,q}}{\partial b_i^m} = s_{i,h}^m \qquad (2.3.58)$$

De esta forma, cuando la entrada p_Q ha sido aplicada a la red y su correspondiente salida $a^M{}_Q$ ha sido computada, el algoritmo Backpropagation de Levenberg-Marquardt es inicializado con:

$$\boldsymbol{S_q^M = -\dot{f}^M(n_q^M)} \qquad (2.3.59)$$

Cada columna de la matriz $S^M{}_Q$ debe ser propagada inversamente a través de la red para producir una fila de la matriz Jacobiana. Las columnas pueden también ser propagadas conjuntamente de la siguiente manera:

$$\boldsymbol{S_q^m = \dot{f}^m(n_q^m)(W^{m+1})^T S_q^{m+1}} \qquad (2.3.60)$$

La matrices sensitividad total para cada capa en el algoritmo de Levenberg-Marquardt son formadas por la extensión de las matrices computadas para cada entrada:

$$S^m = \left[S_1^m\right]\left[S_2^m\right] \ldots \left[S_Q^m\right] \qquad (2.3.61)$$

Para cada nueva entrada que es presentada a la red, los vectores de sensitividad son propagados hacia atrás, esto se debe a que se ha calculado cada error en

forma individual, en lugar de derivar la suma al cuadrado de los errores. Para cada entrada aplicada a la red habrá S^M errores, uno por cada elemento de salida de la red y por cada error se generara una fila de la matriz Jacobiana.

Este algoritmo puede resumirse de la siguiente manera:

1. Se presentan todas las entradas a la red, se calculan las correspondientes salidas y cada uno de los errores según

$$\boldsymbol{e_q = t_q - a_q^M} \tag{2.3.62}$$

se calcula después, la suma de los errores cuadrados para cada entrada *e(x)*

2. Se calculan las sensitividades individuales y la matriz sensitividad total y con estas, se calculan los elementos de la matriz Jacobiana.

3. Se obtiene Δx_k

4. Se recalcula la suma de los errores cuadrados usando $x_k + \Delta x_k$. Si esta nueva suma es más pequeña que el valor calculado en el paso 1 entonces se divide μ por ϑ, se calcula $x_{k+1} = x_k + \Delta x_k$ y se regresa al paso 1. Si la suma no se reduce entonces se multiplica μ por ϑ y se regresa al paso 3.

El algoritmo debe alcanzar convergencia cuando la norma del gradiente de

$$\nabla \boldsymbol{e(x)} = 2\boldsymbol{J}^{\boldsymbol{T}}\boldsymbol{(x)v(x)} \qquad (2.3.63)$$

sea menor que algún valor predeterminado, o cuando la suma de los errores cuadrados ha sido reducida a un error que se haya trazado como meta.

El comportamiento de este algoritmo se visualiza en la figura 2.3.12, la cual muestra la trayectoria de convergencia con $\mu = 0.01$ y $\vartheta = 5$

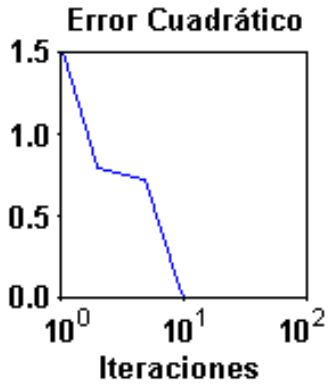

Figura 2.3.12 Trayectoria del algoritmo Levenberg-Marquardt

Como puede verse, este algoritmo converge en menos iteraciones que cualquier método discutido anteriormente, por supuesto requiere mucha más computación por iteración, debido a que implica el cálculo de matrices inversas. A pesar de su gran esfuerzo computacional sigue siendo el algoritmo de entrenamiento más rápido para redes neuronales cuando se trabaja con un moderado número de parámetros en la red, si el número de parámetros es muy grande utilizarlo resulta poco práctico.