CAPÍTULO 3

REDES NEURONALES ARTIFICIALES

3.1 Introducción

Las redes neuronales artificiales, ANNs por sus siglas en inglés, son el resultado de varias décadas de investigaciones desarrolladas en torno a las neuronas del cerebro humano. Las ANNs son estructuras basadas en las redes neuronales del ser humano; son capas de nodos (neuronas) conectados en diferentes configuraciones que dan una respuesta en una capa de *nodos de salida* ante una determinada entrada en una capa de *nodos de entrada* [3].

La red neuronal del cerebro humano se compone de decenas de millones de neuronas interconectadas entre si para conformar el complejo que nos hace reaccionar de manera distinta ante cada evento que se nos presenta. Las redes neuronales artificiales tienen como objetivo ejecutar la misma tarea de reacción que aquellas redes neuronales naturales, esto con el fin de simular los procesos cerebrales en aplicaciones prácticamente en cualquier área. Al igual que en el proceso natural, una ANN debe pasar por situaciones como entrenamiento y aprendizaje [4].

La red neuronal biológica funciona a partir de la interconexión que existe entre las neuronas simples que la conforman. Una neurona simple puede estar conectada a unas cuantas neuronas que se encuentran cerca, o a miles de neuronas que se encuentran a su alrededor. La conexión entre neuronas se hace por medio de los axones y las dendritas. Los

axones son fibras largas por las cuales se transporta la información proveniente de una neurona a otra. Las dendritas son las fibras de una neurona por las cuales entra la información proveniente de otras neuronas por medio de los axones. A las conexiones entre los axones y las dendritas se les llama sinapsis [11].

La información en una red neuronal biológica se presenta en forma de impulsos eléctricos que varían de acuerdo a los impulsos de entrada de cada neurona. En lo que se llama el cuerpo de la neurona es donde son sumados todos los impulsos que existen en sus dendritas y se genera un nuevo impulso que puede valer cero (no hay reacción) en su axón, el cual les llegará a las dendritas de otras neuronas. Basados en este comportamiento biológico es como surge la idea de emularlo en lo que se llama redes neuronales artificiales [3, 11].

Básicamente, las ANNs emplean el mismo principio biológico: las neuronas simples (nodos) pueden estar conectadas con unas cuantas o muchas neuronas dentro de la red, suman todas las señales que entren a ella, y ofrecen una salida que corresponda a dichas entradas. En las ANNs las entradas son multiplicadas por un factor distinto cada una, que equivale a la importancia que tiene cada entrada para generar la salida correspondiente. Las entradas multiplicadas por su respectivo factor son sumadas y la suma total se toma como argumento de una *función de activación* que es la que clasifica la respuesta de una neurona a las entradas (impulsos) presentadas. La salida de cada neurona puede pertenecer a las entradas de otras neuronas o ser la salida o una de las salidas de toda la red neuronal [3].

Los tipos más básicos de red neuronal artificial se describen a continuación, y el

concepto de *función de activación* se describe más adelante en este capítulo.

3.2 Perceptrón

El perceptrón simple o perceptrón es el tipo de red neuronal más sencillo que se puede implementar. Este tipo de red sólo sirve para clasificar los elementos pertenecientes a dos clases linealmente separables [3]. En la Figura 3.1 se muestra un ejemplo de esta ANN con una capa de entrada de tres nodos más su respectivo valor de umbral, una capa de una neurona, y una capa de salida con un solo nodo.

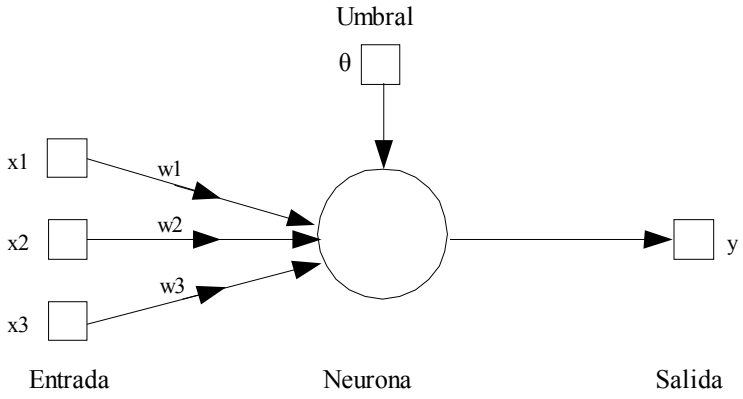

Figura 3.1. Perceptrón simple con tres entradas.

En este tipo de ANN se tienen tantas entradas (x_1, x_2,…, x_n) como se necesiten que se "pesan", es decir, se multiplican por su respectivo *peso* (w_1, w_2,…, w_n), y se suman junto con un valor de umbral (θ), dando como resultado una salida que se toma como el valor del argumento de una función, obteniendo así una respuesta de la red ante un conjunto de entradas particulares. El valor de umbral sirve para establecer la respuesta de la neurona como perteneciente a una clase u otra, dependiendo de los valores de entrada "pesados" [4, 11]:

$$\textit{Clase 1}: x_1 \cdot w_1 + x_2 \cdot w_2 + \ldots + x_n \cdot w_n > \theta \qquad \textbf{(3.1)}$$

$$\textit{Clase 2}: x_1 \cdot w_1 + x_2 \cdot w_2 + \ldots + x_n \cdot w_n < \theta \qquad \textbf{(3.2)}$$

Dadas las ecuaciones (3.1) y (3.2), claramente se ve que el perceptrón simple sólo tiene la capacidad de responder de dos formas distintas dependiendo de las entradas, es decir, cualquier conjunto de entradas sólo puede pertenecer a uno de los dos conjuntos de respuesta del perceptrón: a la izquierda o derecha de θ. La función que típicamente se usa para clasificar las respuestas de esta ANN es la función *hard limiter* mostrada en la Figura 3.2 [3].

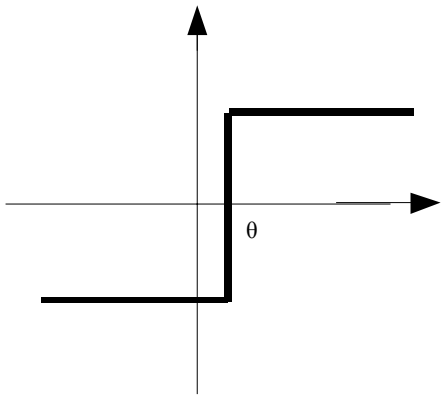

Figura 3.2. Función *hard limiter*.

Para que un perceptrón simple sea funcional, inicialmente se le dan a los pesos valores aleatorios pequeños, usualmente en el rango -0.5<wn<0.5. En seguida se "muestra" el *set* de entradas con su respectiva salida deseada y es así como empieza el algoritmo de entrenamiento. La diferencia entre la salida deseada y la salida que se obtiene del perceptrón es la clave para ajustar los pesos a valores que optimizarán el desempeño de la

red. Al actualizar los pesos para ajustarlos a salidas deseadas se esta hablando de un entrenamiento supervisado [3]. La ecuación (3.3) es de gran utilidad y muy usada en el proceso de actualización de los pesos, donde n es el número de *pattern* (entrada-salida), **w**(n) es el vector de pesos, **x**(n) es el vector de entradas, d(n) es la salida deseada, y(n) es la salida obtenida en la iteración n, y η es un parámetro de aprendizaje (*learning rate)* entre 0 y 1 [3, 8].

$$\mathbf{w}(n+1)= \mathbf{w}(n)+\eta[d(n)-y(n)]\mathbf{x}(n) \qquad \textbf{(3.3)}$$

El proceso de aprendizaje del perceptrón, se basa en la ecuación (3.3) de forma iterativa hasta obtener d(n)-y(n)=0 para todos los *patterns*. Esto se logra presentando a la red una y otra ves todos los *patterns* actualizando sólo aquellos que están mal clasificados. Al conjunto de todos los *patterns* se le llama *epoch*.

3.3 Perceptrón Multicapa

El perceptrón multicapa, MLP por sus siglas en inglés, es uno de los tipos más sencillos de redes neuronales multicapa y se deriva del concepto del perceptrón simple, presentando capas de neuronas, en lugar de una sola neurona, que van "ocultas", es decir, no pertenecen ni a la entrada ni a la salida de la red. Este tipo de redes, al componerse de varias capas, tiene la característica de no ser lineal, es decir, es capaz de clasificar entradas que pertenecen a dos o más clases que no son linealmente separables [11].

La figura 3.3 muestra la configuración de un MLP completamente conectado (las

entradas de cada neurona vienen de todos los nodos de la capa anterior) con tres entradas, dos capas de neuronas ocultas, y tres salidas.

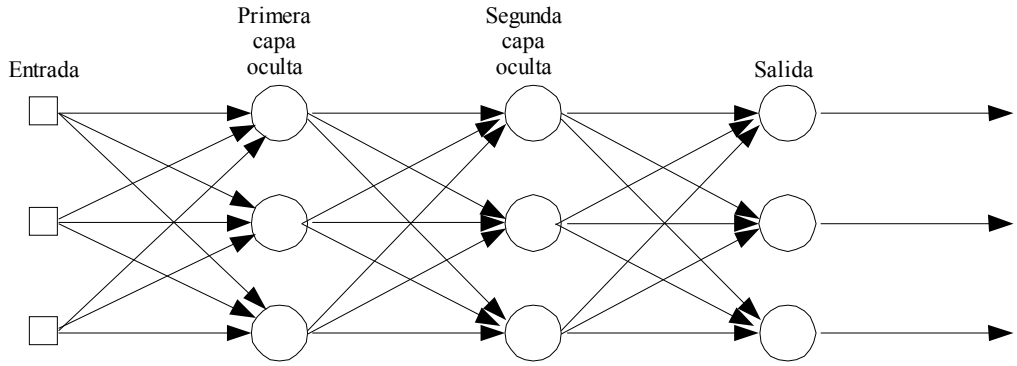

Figura 3.3. Perceptrón multicapa con dos capas ocultas.

Al tener varias neuronas interconectadas, el MLP tiene un desempeño demasiado eficaz en clasificación dentro de aplicaciones más complejas donde existen más de dos grupos de clasificación para las posibles entradas que se puedan presentar. Este tipo de red también es muy eficaz en la aproximación de casi cualquier función. En el MLP, así como en el perceptrón simple, cada salida de un nodo es "pesada" antes de llegar a su nodo destino. Pueden existir infinidad de configuraciones para un MLP jugando con el número de capas ocultas, número de neuronas en cada capa, número de salidas, entradas en cada neurona, e incluso con el número de entradas a la red, este último dependiendo de la aplicación [3, 8, 11]. Un MLP con dos capas ocultas es una configuración suficiente y muy eficiente en la mayoría de las aplicaciones. Otra configuración es la interconexión de MLPs, con lo que se obtiene una red más grande y más compleja por consiguiente. [8]

El diseñar una arquitectura de MLP es una tarea que depende de diversos factores.

La problemática se encuentra principalmente en el número de neuronas que cada capa debe tener. La teoría matemática, si bien no es compleja, es un tanto complicada cuando se trata de problemas donde se tienen elementos a clasificar dentro de un número grande de clases posibles, o donde se trata de aproximar funciones. Para estos casos, es más fácil experimentar con diversas arquitecturas, con dos capas ocultas si se trata de un problema complicado, y encontrar la solución más adecuada u óptima. Cabe mencionar que en la mayoría de los problemas tanto de clasificación como de aproximación de funciones no existe una solución única para la arquitectura de la red neuronal que se debe utilizar, pero si probablemente un tamaño de arquitectura mínimo que nos facilita el trabajo en el diseño, ya que entre más grande sea el número de neuronas en las capas ocultas, mejor será la clasificación o aproximación obtenida. Por otro lado, no se debe exagerar en el número de neuronas por capa ya que se puede presentar inestabilidad en el sistema, y aunque en muchos casos se obtienen mejores resultados en clasificación y aproximación de funciones, también se aumenta el tiempo tanto de entrenamiento como de clasificación o aproximación. En sí, pocos son los problemas en los que se puede determinar la configuración óptima, principalmente el número de capas y neuronas ocultas que dan la mejor solución, por lo que aún se están haciendo investigaciones en relación al diseño de redes neuronales [8].

El perceptrón multicapa es una muy buena opción en problemas que plantean la aproximación de casi cualquier función, debido al sustento del *Universal Approximation Theorem* y de las distintas aplicaciones en las que ha demostrado un gran potencial. La ecuación (3.4) define al *Universal Approximation Theorem* que establece que cualquier función contínua puede aproximarse con un perceptrón multicapa con una capa oculta

como sigue:

$$f(x_1,...,x_p) \approx F(x_1,...,x_p) = \sum_{i=1}^{M} \alpha_i \cdot \varphi(\sum_{j=1}^{p} w_{ij} \cdot x_j - \theta_i) \qquad \textbf{(3.4)}$$

donde *f* es la función que se quiere aproximar, F es la función que aproxima a *f*, x_1 a x_p son las variables de la función que se quiere aproximar, M es el número de neuronas en la capa oculta, α_i representa el peso entre la conexión de la iésima neurona oculta y la neurona de salida, p es el número total de variables de la función a aproximar y que se toman como entradas de la red, w_{ij} es el peso entre la conexión de la iésima entrada y la neurona oculta j, y θ_i es el valor de umbral de la neurona i [3].

El algoritmo de entrenamiento/aprendizaje supervisado más común del MLP es el *Back-Propagation Algorithm*, de hecho, este algoritmo surgió de la búsqueda de un algoritmo para entrenar específicamente al MLP [3]. Es un tanto complejo ya que se toma información del comportamiento de la red en el sentido directo de la red y en el sentido inverso, esto por la necesidad de modificar el comportamiento de las capas ocultas.

3.4 Back-Propagation

El *Back-Propagation Algorithm* tiene el mismo objetivo que aquel usado para entrenar un perceptrón simple: usar la diferencia entre las salidas deseadas y las salidas actuales en la capa de salida de la red para cambiar los pesos (iniciados con valores aleatorios pequeños) con el fin de reducir al mínimo esta diferencia (error). Esto se logra mediante una serie de

iteraciones donde se modifica cada peso de derecha a izquierda (sentido inverso de la propagación de información en la red) hasta modificarse los pesos de la capa de entrada prosiguiendo nuevamente con la propagación de la información de entrada, esto hasta que la diferencia entre la salida deseada y la obtenida en cada neurona de salida sea mínima [3, 4, 8, 11].

El *Back-Propagation Algorithm* es el método que desde un principio se desarrolló con el fin de entrenar redes neuronales multicapa y se demostró su eficiencia en el entrenamiento de redes para resolver diversos problemas, pero en muchos casos resultó ser muy lento [2, 3]. A través de los años han surgido algoritmos más poderosos, aunque más complejos, la mayoría teniendo el mismo principio del *Back-Propagation* -propagar el error hacia atrás-, debido a que el algoritmo ha demostrado ser una buena solución para el entrenamiento de MLPs, pero muchas veces se requiere de un proceso más rápido. De cualquier forma, es recomendable el estudio del *Back-Propagation Algorithm* cuando se trata el diseño de perceptrones multicapa, ya que no es demasiado complejo, se entiende fácilmente su finalidad, y sirve para comprender más rápido los algoritmos que se basan en el.

El *Back-Propagation* hace uso las ecuaciones (3.5), (3.6) (*delta rule*), (3.7) y (3.8) (*gradient descent*), y (3.9) (función de activación) en el proceso del ajuste de los pesos que conectan la salida de la neurona i a la entrada de la neurona j en la iteración n [3]:

$$w_{ji}(n+1)=w_{ji}(n)+\Delta w_{ji}(n) \qquad \textbf{(3.5)}$$

$$\Delta w_{ji}(n) = \eta \cdot \delta_j(n) \cdot y_i(n) \qquad \textbf{(3.6)}$$

$$\delta_j(n) = \begin{cases} [d_j(n) - y_j(n)] \cdot y_j'(n) & \textit{para la capa de salida} \\ y_j'(n) \cdot \sum_k \delta_k(n) \cdot w_{kj}(n) & \textit{para las capas ocultas} \end{cases} \qquad \textbf{(3.7)}$$

$$\delta_k(n) = [d_k(n) - y_k(n)] \cdot y_k'(n) \qquad \textbf{(3.8)}$$

$$y_j(n) = \frac{1}{1 + \exp(-v_j(n))} \qquad \textbf{(3.9)}$$

donde k es un subíndice específico de la capa de salida y se usa en lugar de j como segunda opción (j también se usa en las demás capas), y $v_j(n)$ es la salida inmediata de la neurona j en el *pattern* n. Al igual que en el perceptrón simple, $d_j(n)$ es la salida deseada, $y_j(n)$ es la salida obtenida de la función de activación de cada neurona, $y_j'(n)$ es la derivada de la función de activación de cada neurona con respecto a $v_j(n)$, y η es el *learning rate* usualmente con valor inicial entre 0 y 1.

En el *Back-Propagation*, $\delta_j(n)$ es el gradiente local del valor instantáneo de la suma de los errores cuadrados obtenidos en las neuronas de salida. Este gradiente es el que orienta el entrenamiento hacia el error mínimo de todos los *patterns* en conjunto y en el *Back-Propagation* depende directamente de la magnitud de $y_j'(n)$ [3]. La demostración del gradiente local es un tanto complicada y queda fuera de los objetivos de este trabajo.

3.4.1 Mean Squared Error (MSE)

El método empleado para que el error en los algoritmos de entrenamiento supervisado converja rápidamente hacia un mínimo es el Mean Squared Error. Esto es una búsqueda para obtener el valor mínimo posible de la suma de los errores cuadrados de las neuronas de salidas en cada *pattern*. La siguiente fórmula es la empleada para calcular el MSE en cada *epoch* [8]:

$$MSE = \frac{1}{N}\sum_{n=1}^{N} E(n) \qquad \textbf{(3.10)}$$

donde N es el número total de *patterns* presentados y E(n) es la suma de los errores cuadrados de todas las neuronas de salida en el *pattern* n:

$$E(n) = \sum_{k \in C} e_k^2(n) \qquad \textbf{(3.11)}$$

$$e_k(n) = d_k(n) - y_k(n) \qquad \textbf{(3.12)}$$

donde C es el conjunto de todas las neuronas de salida de un MLP. Así, se establece un mínimo del MSE en el cual se dejan de actualizar los pesos y se da por terminado el entrenamiento de una red neuronal. El valor mínimo óptimo del MSE depende enteramente de la aplicación, más que nada de las características de los valores deseados a la salida de la red: si se trata de números enteros, un valor común del MSE es 0.1; si se trata de valores

decimales, entonces el MSE debe ser menor dependiendo de la exactitud deseada [3, 8].

Para el *Back-Propagation* es usual el utilizar en el MSE la suma instantánea de los errores cuadrados de las neuronas de salida, así, E(n) está dado por [3]:

$$E(n) = \frac{1}{2}\sum_{k \in C} e_k^2(n) \qquad \textbf{(3.13)}$$

donde C es el conjunto de todas las neuronas de salida.

3.4.2 Funciones de Activación

En la Figura 3.2 se mostró la función de activación común para las neuronas de un perceptrón simple y en la ecuación (3.9) se definió la función de activación para las neuronas de un perceptrón multicapa cuya gráfica se muestra en la Figura 3.4.

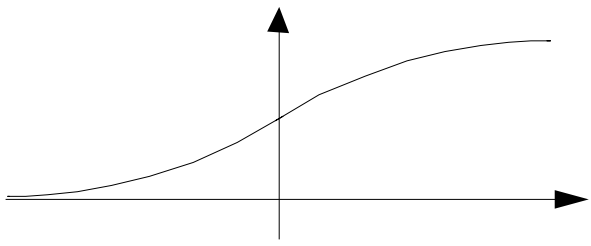

Figura 3.4. Función *logarithmic sigmoid*.

Existen varias funciones que pueden ser usadas como funciones de activación en las redes neuronales siempre y cuando cumplan ciertos requisitos de la red y de la aplicación.

El perceptrón simple no presenta tantos requisitos para una función de activación, aunque la más usada es la *hard limiter* de la Figura 3.2. Para el perceptrón multicapa no es el mismo caso, ya que la función de activación debe ser derivable en todo su dominio para poder calcular el gradiente local del error de las neuronas de salida [3, 8].

La función *logarithmic sigmoid* mostrada en la Figura 3.4 es en la que se basó principalmente el *Back-Propagation Algorithm* pero ya no es muy usada debido a que hace que el error converja muy lento cuando se tiene una gran cantidad de *patterns* y más cuando se encuentra en los extremos donde la derivada es cercana a cero (y_j'(n)) [3].

Para acelerar la convergencia del MSE se usan otras funciones como lo es la *hyperbolic tangent sigmoid* que se muestra en la Figura 3.5 y la *linear* que especialmente se aplica a las neuronas de salida y se muestra en la Figura 3.6 [8].

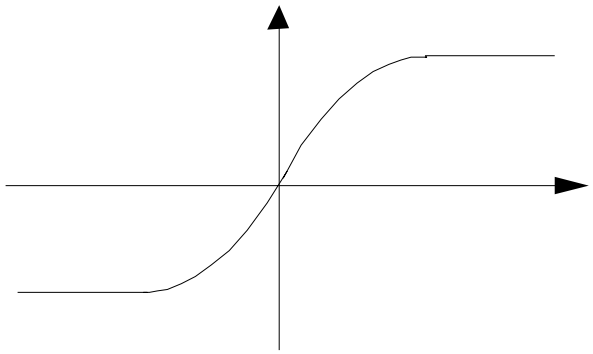

Figura 3.5. Función *hyperbolic tangent sigmoid*.

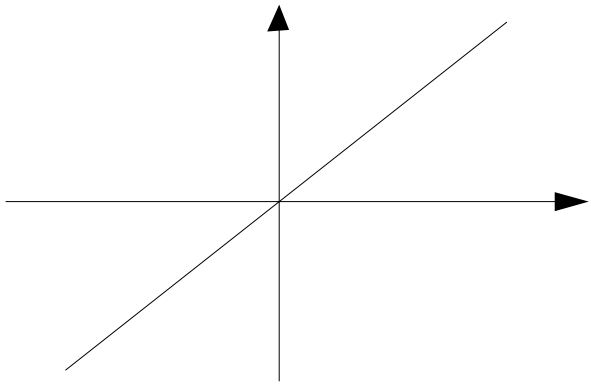

Figura 3.6. Función *linear*.

La función *hyperbolic tangent sigmoid* es una variante de la *logarithmic sigmoid* desplazada y alargada verticalmente y comprimida horizontalmente, lo que hace que su derivada cambie más rápidamente y por consiguiente también el gradiente local del error. La ecuación que define a esta función es la siguiente [3]:

$$y_j(n) = a \cdot \tanh(b \cdot v_j(n)) = a \cdot \left[\frac{1 - \exp(-b \cdot v_j(n))}{1 + \exp(-b \cdot v_j(n))} \right] = \frac{2a}{1 + \exp(-b \cdot v_j(n))} - a \quad \textbf{(3.14)}$$

La función *linear* es una simple línea dada por la ecuación:

$$y_j(n) = a \cdot v_j(n) + b \quad \textbf{(3.15)}$$

En ambos casos, a y b son constantes que se pueden ajustar libremente, y muchas veces se encuentra su valor óptimo mediante pruebas empíricas que llevan al mejor

desempeño para cada aplicación en especial.

Todas las funciones de activación definen el rango de valores que se puede obtener en una neurona. Este es un punto muy importante en el diseño de una red neuronal, ya que se debe definir el rango de valores que pueden tomar más que nada de las neuronas de salida. Comúnmente los rangos de las funciones de activación más usadas en redes neuronales son como sigue: *hard limiter* {-1, 1}, *logarithmic sigmoid* (0, 1), *hyperbolic tangent sigmoid* (-1, 1), *linear* $(-\infty, \infty)$ [3, 4, 8].

La mayoría de las aplicaciones que usan MLP para aproximar funciones requieren que las neuronas de las capas ocultas tengan función de activación *hyperbolic tangent sigmoid* y las neuronas de la capa de salida tengan función de activación *linear* o *hyperbolic tangent sigmoid*, dependiendo del rango que puedan tener las salidas deseadas [8]. Aquí es importante recalcar que se debe poner especial cuidado en el uso del *Back-Propagation* en la parte del uso de las derivadas de las funciones de activación.

3.4.3 Delta Rule y Learning Rate

Aparte de las funciones de activación, de otros parámetros depende el *Back-Propagation* para converger rápida y correctamente como son la *delta rule* y el *learning rate* [3].

La ecuación (3.5) es conocida comúnmente como la *delta rule* ya que depende de un gradiente, en este caso para actualizar los pesos de una red neuronal. Esta ecuación muchas veces hace que el MSE sea inestable, incluso que diverja y se vaya a infinito, especialmente

cuando se trata de un entrenamiento con un número grande de *patterns*. Para esto se cuenta con una modificación a la que se le llama *generalized delta rule* descrita por la ecuación 3.15 [3]:

$$\Delta w_{ji}(n)=\alpha\cdot\Delta\cdot w_{ij}(n-1)+\eta\cdot\delta_j(n)\cdot y_i(n) \qquad \textbf{(3.16)}$$

donde α es una constante a la que se le llama *momentum constant* con valor dentro del rango [0,1]. Esta constante hace más estable la actualización de cada peso al tomar en cuenta el valor y más que nada el signo del gradiente de la iteración anterior n-1 (*pattern*). Si se tienen dos gradientes consecutivos con el mismo signo entonces los pesos serán incrementados o decrecidos (según sea el caso) más rápidamente; si se tienen dos gradientes consecutivos con diferente signo entonces los pesos serán incrementados o decrecidos muy poco.

La inclusión de α hace que en las regiones donde el signo del gradiente oscila, los pesos se actualicen poco en magnitud con el fin de evitar una inestabilidad mayor, y en las regiones donde el signo es el mismo, los pesos se actualicen más rápido en una misma dirección que puede ser la correcta [3, 8].

El *learning rate* η es una constante positiva cuyo valor usualmente está dentro del rango [0,1]. Esta constante sirve para definir el *costo* que tiene el gradiente en la actualización de un peso. Entre mayor se defina su valor, mayor la magnitud en la que se incrementa o decrece el peso, lo cual puede ser bueno o afectar la convergencia del MSE. Para que el *learning rate* no se convierta en un problema que cause inestabilidad, es

recomendable que su valor sea actualizado cada ves que se actualicen los pesos de la siguiente manera: si se da el caso de que se presenta una disminución del MSE entonces el η se incrementa, por el contrario si se presenta un incremento del MSE entonces se disminuye el η y se cancela la última actualización de pesos que generó el incremento del MSE con el fin de evitar oscilaciones y divergencia [3].

3.4.4 Modos de Entrenamiento

Existen dos formas eficientes de entrenar una red neuronal: *pattern mode* o *incremental mode* y *batch mode* [3]. Estos modos se refieren a la manera en que son actualizados los pesos.

En el *pattern mode* los pesos se actualizan en cada *pattern* presentado mientras que en el *batch mode* los pesos se actualizan hasta presentar todos los *patterns*. Cada uno se puede usar en el *Back-Propagation* obteniendo diferentes resultados, uno mejor que otro dependiendo principalmente del problema tratado. Ambos algoritmos ofrecen mejores resultados si se cambia el orden de presentación de los *patterns* entre un *epoch* y otro [3, 8].

3.5 Levenberg-Marquardt

El *Back-Propagation* ha demostrado converger muy lentamente en varias aplicaciones, en especial cuando se tiene una gran cantidad de *patterns* donde suele converger pero a un MSE demasiado grande, lo que se llama mínimo local del MSE y que muchas veces no es útil ya que se busca una convergencia hacia el mínimo absoluto [8]. Es por eso que las

investigaciones en cuanto a redes neuronales no se detiene, en especial porque han demostrado un gran potencial. A la fecha existen diferentes algoritmos de entrenamiento supervisado que han surgido del *Back-Propagation* y que muestran velocidades mucho más rápidas de convergencia del MSE hacia el mínimo absoluto. Uno de ellos es el algoritmo de Levenberg-Marquardt.

El algoritmo de Levenberg-Marquardt se aplica principalmente a redes neuronales multicapa con un número grande de *patterns* ya que tiene la velocidad de convergencia del MSE más rápida hasta ahora , principalmente en problemas de aproximación de funciones a pesar de que su complejidad en cálculos es mayor [8, 13]. Usa la metodología del *Back-Propagation* empleando el concepto de la *generalized delta rule,* usando el concepto de *learning rate*, y aplicando el *batch mode*, sólo que el gradiente se calcula mediante la matriz Jacobiana de los errores de las neuronas de salida. La ecuación con la que se actualizan los pesos es la siguiente [8]:

$$w(n+1) = w(n) - \alpha \cdot w(n-1) - \frac{J^T \cdot e}{J^T \cdot J + \mu \cdot I} \qquad \textbf{(3.17)}$$

donde J es la matriz Jacobiana de los errores de las neuronas de salida, es decir, la matriz de las primeras derivadas de dichos errores con respecto a los pesos y umbrales (θ) de los que son función, J^T es la transpuesta de la matriz Jacobiana, I es la matriz identidad (unos en la diagonal y ceros en las demás localidades) del mismo tamaño que la matriz Jacobiana, *e* es el vector de errores de las neuronas de salida, α es la *momentum constant*, y μ es una constante equivalente al *learning rate* que es decrecida en cada iteración en la que se

observa una reducción del MSE, o incrementada y se descartan los pesos actualizados cuando se obtiene un aumento en el MSE.

Este algoritmo, aunque requiere de mayor número de cálculos que el *Back-Propagation*, evita más las oscilaciones del MSE y la matriz Jacobiana es la que hace que se tenga una convergencia demasiado rápida, incluso hasta más de 100 veces más rápida que la obtenida por el *Back-Propagation* con su *gradient-descent* [8, 13].