CAPÍTULO 3

PERCETRON MULTICAPA CON UNA CAPA OCULTA O SUPERFICIAL

1. 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 solo 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 que efectivamente, se solucionan con esta nueva arquitectura propuesta.

¿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 la diferencia o 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 como lo veremos a lo largo de este capítulo.

Las redes neuronales superficiales contribuyeron al posicionamiento de la RNA como una de las técnicas más usadas del aprendizaje automático sin embargo, la imposibilidad de extender sus buenos resultados a redes de muchas capas o profundas produjo el llamado segundo invierno de las RNA.

2. ARQUITECTURA GENERAL DE UN PERCEPTRÓN MULTICAPA SUPERFICIAL

En la figura 3.1 presentamos la estructura del Perceptron Multicapa Superficial (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.

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 en la capa oculta suele ser tangente-sigmoidal, sigmoidal y recientemente tipo ReLU. Para la capa de salida se puede trabajar neuronas con función de activación sigmoidales, lineales o *softmax*. El tipo de función de activación que se trabaja en la salida depende del problema que se esté resolviendo que en general puede ser de dos tipos.

- De regresión
- De clasificación

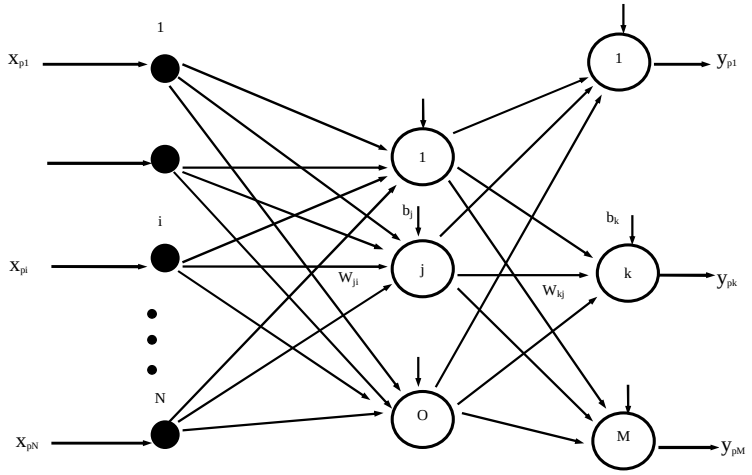

Fig. 3.1 Arquitectura General de un MLP

3. 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 o propagación inversa 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 si 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 planteó 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.

3.1. Nomenclatura para la Redes Superficiales

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. Número de neuronas en la capa de entrada
P	Número de ejemplos, vectores de entrada y salidas diferentes.
O	Número de neuronas de la capa oculta
M	Número de neuronas de la capa de salida, dimensión del vector de salida
w_{ji}^{o}	Peso de interconexión entre la neurona *i-ésima* de la entrada y la *j-ésima* de la capa oculta.

b_j^o	Término de tendencia de la neurona *j-ésima* de la capa oculta.
$Neta_{pj}^o$	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^o	Función de activación de la *j-ésima* unidad oculta
w_{kj}^s	Peso de interconexión entre la *j-ésima* neurona de la capa oculta y la *k-ésima* neurona de la capa de salida.
b_k^s	Término de tendencia de la *k-ésima* neurona de la capa de salida.
$Neta_{pk}^s$	Entrada neta de la *k-ésima* neurona de la capa de salida.
y_{pk}	Salida de la *k-ésima* unidad de salida
f_k^s	Función de activación de la *k-ésima* unidad de salida
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'^o_j	Derivada de la función de activación de la *j-ésima* neurona de la capa oculta.
f'^s_k	Derivada de la función de activación de la *k-ésima* neurona de la capa de salida.

3.2. Algoritmo de Entrenamiento

Gradiente Descendente Estocástico y Gradiente Descendente

El gradiente descendente se aplica a redes neuronales para la actualización de pesos sin embargo, tal y como lo definimos en el capítulo anterior, para su cálculo se lo habitual es usar todos los patrones de entrenamiento o el lote completo de datos (*batch*). En caso de tener muchos datos para entrenar la red, el cálculo del gradiente se puede hacer computacionalmente dispendioso e inclusive prohibitivo.

Para evitar esto existe una alternativa que es calcular el gradiente en un subconjunto de los datos de entrenamiento. A este subconjunto de datos se le denomina minilote (*minibatch*). La selección de los datos usados para el cálculo del gradiente se hace de manera aleatoria. A esto se le ha denominado gradiente descendente estocástico (GDE)..

Generalmente el GDE converge mucho más rápido en comparación con GD, sin embargo, no se suele llegar tan cerca al mínimo de la función de pérdida como lo hace el GD. En la mayoría de los casos el GDE queda oscilando alrededor del

mínimo mencionado. Este comportamiento lo podemos observar en la figura 3.2. En ella verificamos que si usan todos los datos de entrenamiento (GD) la convergencia al mínimo de la función de costo o pérdida es mucho más definida y suave. En el caso de usar un solo dato de entrenamiento (GDE) la convergencia es más errática sin embargo, se alcanza un punto muy cercano al mínimo buscado.

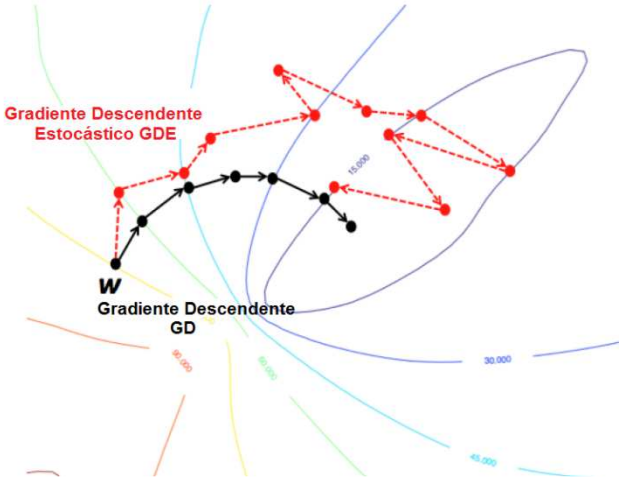

Fig. 3.2 Diferencia en el Comportamiento de GDE y GD
Fuente: https://wikidocs.net/3413

Aunque en uso del GDE se ha hecho muy popular principalmente en las redes profundas, usarlo en las redes superficiales es una buena estrategia para disminuir la carga computacional en el entrenamiento de las mismas.

Gradiente Descendente Estocástico (GDE)

En este caso supondremos que el tamaño del minilote es de un patrón de entrenamiento. Este patrón será seleccionado de manera aleatoria del conjunto de datos de entrenamiento. 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 del patrón seleccionado p:

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

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

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

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

$$y_{pj}{}^{o} = f_j^{o}(Neta_{pj}{}^{o}) \tag{3.3}$$

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 siguiente y así podemos calcular la entrada neta de la *k-ésima* neurona de la capa de salida:

$$Neta_{pk}{}^{s} = \sum_{j=1}^{O} w_{kj}\ y_{pj}{}^{o} + b_k{}^{o} \tag{3.4}$$

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:

$$y_{pk} = f_k{}^{s}(Neta_{pk}{}^{s}) \tag{3.5}$$

Actualización de pesos para la capa de salida

En este caso se aplicará el concepto de gradiente descendente presentado en el capítulo 2 pero aplicado al p-ésimo patrón. Esto es lo que denominados gradiente descendente estocástico:

$$\begin{aligned} w_{kj}{}^{s}(t+1) &= w_{kj}{}^{s}(t) + \Delta w_{kj}{}^{s}(t) \\ w_{kj}{}^{s}(t+1) &= w_{kj}{}^{s}(t) + \left(-\alpha \frac{\partial L}{\partial w_{kj}^{s}(t)} \right) \end{aligned} \tag{3.6}$$

En este caso la función de pérdida queda así:

$$\begin{aligned} L_p &= \frac{1}{M} \sum_{k=1}^{M} (d_{pk} - y_{pk})^2 \\ L_p &= \frac{1}{M} \sum_{j=1}^{M} e_{pk}{}^{2} \end{aligned} \tag{3.7}$$

Si en la ecuación anterior aplicamos la regla de la cadena para obtener la expresión del gradiente, este queda en términos de la derivada de la función de pérdida respecto al error local de la k-ésima neurona, la derivada del error local

respecto de la salida de la red, la derivada de la salida respecto a la entrada neta y la derivada de la neta respecto del peso de la w_{kj}.

$$\Delta w_{kj}{}^{s}(t) = -\alpha \frac{\partial L_p}{\partial e_{pk}} \cdot \frac{\partial e_{pk}}{\partial y_{pk}} \cdot \frac{\partial y_{pk}}{\partial Neta_{pk}} \cdot \frac{\partial Neta_{pk}}{\partial w_{kj}^{s}(t)} \tag{3.8}$$

Tras evaluar las derivadas parciales de la ecuación 3.8, obtenemos los siguientes valores,

$$\begin{aligned}
&\frac{\partial L_p}{\partial e_{pk}} = \frac{2}{M}(d_{pk} - y_{pk}) = \frac{2}{M} e_{pk} \\
&\frac{\partial e_{pk}}{\partial y_{pk}} = -1 \\
&\frac{\partial y_{pk}}{\partial Neta_{pk}} = f_k{}^{'s}(Neta_{pk}{}^{s}) \\
&\frac{\partial Neta_{pk}}{\partial w_{kj}^{s}(t)} = y_{pj}{}^{o}
\end{aligned} \tag{3.9}$$

Estos valores de las derivadas parciales los reemplazamos en la ecuación de actualización para obtener el valor final de la variación del peso de la *i-ésima* conexión.

$$\begin{aligned}
&\Delta w_{kj}^{s}(t) = -\alpha \frac{2}{M}(d_{pk} - y_{pk})(-1) f_k{}^{'s}(Neta_{pk}{}^{s}) y_{pj}{}^{o} \\
&\Delta w_{kj}^{s}(t) = \alpha \frac{2}{M} e_{pk} f_k{}^{'s}(Neta_{pk}{}^{s}) y_{pj}{}^{o}
\end{aligned} \tag{3.10}$$

Retomando la ecuación 3.9 y 3.10, la expresión para la actualización del peso de la k-j-*ésima* conexión está dada por la ecuación 3.11.

$$\begin{aligned}
&w_{kj}{}^{s}(t+1) = w_{kj}{}^{s}(t) + \Delta w_{kj}{}^{s}(t) \\
&w_{kj}{}^{s}(t+1) = w_{kj}{}^{s}(t) + \alpha \frac{2}{M}(d_{pk} - y_{pk}) f_k{}^{'s}(Neta_{pk}{}^{s}) y_{pj}
\end{aligned} \tag{3.11}$$

Podemos reescribir la ecuación 3.11 para condensarla definiendo el término de error δ_k. Este término lo podemos interpretar como el error visto a la entrada de

la red neuronal, en contraposición del error local e_k que es el error en la salida de la neurona

$$\begin{aligned} w_{kj}^{\ s}(t+1) &= w_{kj}^{\ s}(t) + \alpha \frac{2}{M} \delta_k^s y_{pj} \\ \delta_k^s &= (d_{pk} - y_{pk}) f_k^{'s}(Neta_{pk}^{\ s}) \\ \delta_k^s &= e_{pk} f_k^{'s}(Neta_{pk}^{\ s}) \end{aligned} \qquad (3.12)$$

Actualización de los bias para la capa de salida

Para la actualización de los bias usaremos las mismas ecuaciones que usamos para los pesos. Es decir, se actualizan usando los mismos términos de error, la diferencia es que en el bias la entrada siempre vale uno por lo que su actualización no depende de un valor de entrada específico

$$\begin{aligned} b_k^s(t+1) &= b_k^s(t+1) + \alpha \frac{2}{M} \delta_k^s \\ \delta_k^s &= (d_{pk} - y_{pk}) f_k^{'s}(Neta_{pk}^{\ s}) \\ \delta_k^s &= e_{pk} f_k^{'s}(Neta_{pk}^{\ s}) \end{aligned} \qquad (3.13)$$

Actualización de pesos para la capa oculta

Para la capa oculta tendremos la siguiente ecuación

$$\begin{aligned} w_{ji}^{\ o}(t+1) &= w_{ji}^{\ o}(t) + \Delta w_{ji}^{\ o}(t) \\ w_{ji}^{\ o}(t+1) &= w_{ji}^{\ o}(t) + \left(-\alpha \frac{\partial L}{\partial w_{ji}^{\ o}(t)} \right) \end{aligned} \qquad (3.14)$$

Haciendo un análisis similar al realizado para la capa de salida, en la capa oculta se tendría

$$\begin{aligned} w_{ji}^{\ o}(t+1) &= w_{ji}^{\ o}(t) + \alpha \frac{2}{M} \delta_j^o x_i \\ \delta_j^o &= (d_{pj}^o - y_{pj}^o) f_k^{'s}(Neta_{pk}^{\ s}) \\ \delta_j^o &= e_{pj}^o f_j^{'o}(Neta_{pj}^{\ o}) \end{aligned} \qquad (3.15)$$

Donde δ_j es el error a la entrada de la entrada de la k-ésima neurona de la capa oculta. Este error lo expresamos en término del error e_j que hay en la salida de dicha neurona. En esta ecuación se hace evidente el inconveniente de las redes

multicapa pues al no conocer el valor deseado de la capa oculta d^o_j no es posible calcular el error en dichas neuronas y por lo tanto, no podemos encontrar una expresión para modificar las conexiones que llegan a dichas neuronas

Backpropagation

Para solucionar este inconveniente aplicaremos el concepto de *backprogration*. Este nos dice que es posible estimar el error de una neurona con los errores de la neurona en la capa siguiente. En otras palabras, para este caso podemos estimar el error de las neuronas de la capa oculta propagando hacia el interior de la red los errores de la capa de salida que si se pueden calcular de manera exacta. En la figura 3.3 mostramos el flujo de información cuando se aplica este concepto para el cálculo de los errores de la capa oculta

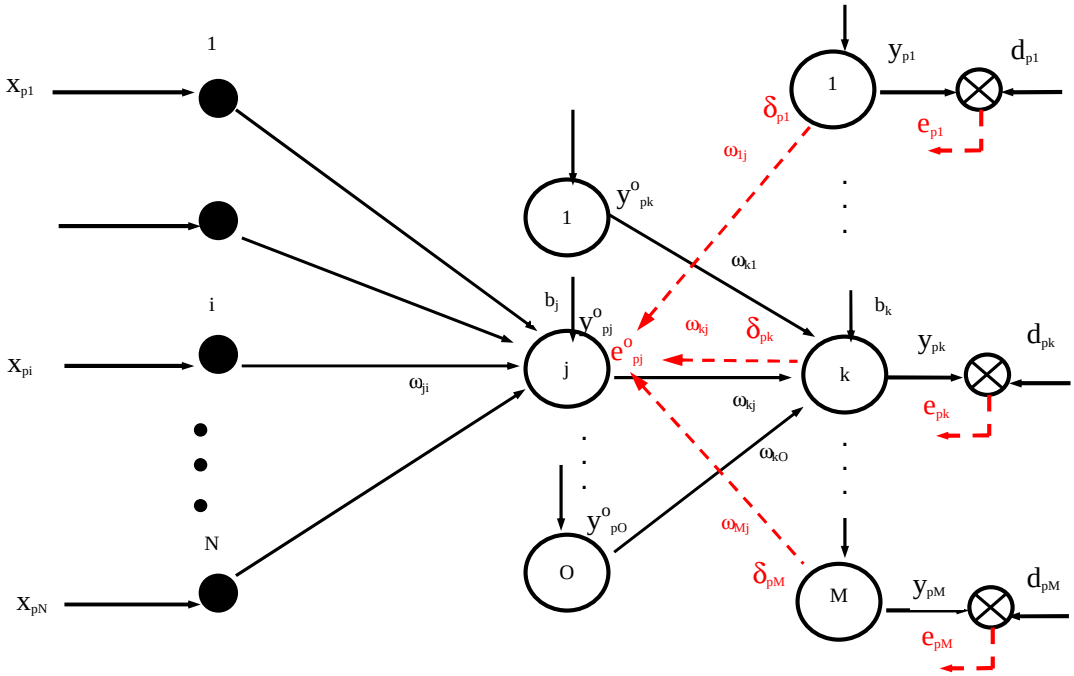

Fig. 3.3 Flujo del error (en líneas punteadas rojas) aplicando *backpropagation* para estimar el error de las neuronas de la capa oculta

Al aplicar *backpropagation* tenemos la expresión de la ecuación 3.16 donde mostramos que se puede estimar el error de la j-ésima neurona de la capa oculta propagando al interior de la red los errores de la capa de salida.

$$e_{pj}^{\ o} = \sum_{k=1}^{M} \delta_{pk}^{\ s} w_{kj}^{\ s} \qquad (3.16)$$

Con esta expresión, podemos reescribir la ecuación de entrenamiento para las neuronas de la capa oculta donde el error de dichas neuronas lo estimamos con la ecuación de *backpropagation*

$$
\begin{aligned}
w_{ji}^{\ o}(t+1) &= w_{ji}^{\ o}(t) + \alpha \frac{2}{M} \delta_j^o x_i \\
\delta_j^o &= (d_{pj}^o - y_{pj}^o) f_k^{\ 's}(Neta_{pk}^{\ s}) \\
\delta_j^o &= e_{pj}^o f_j^{\ 'o}(Neta_{pj}^{\ o}) \\
\delta_j^o &= \sum_{k=1}^{M} \delta_{pk}^{\ s} w_{kj}^{\ s} f_j^{\ 'o}(Neta_{pj}^{\ o})
\end{aligned}
\qquad (3.17)
$$

Actualización de los bias para la capa oculta

Para la actualización de los bias usaremos las mismas ecuaciones que usamos para los pesos. Es decir, se actualizan usando los mismos términos del error estimado usando *backpropagation*, la diferencia es que en el bias la entrada siempre vale uno por lo que su actualización no depende de un valor de entrada específico

$$
\begin{aligned}
b_j^o(t+1) &= b_j^o(t+1) + \alpha \frac{2}{M} \delta_j^o \\
\delta_j^o &= (d_{pj}^o - y_{pj}^o) f_k^{\ 's}(Neta_{pk}^{\ s}) \\
\delta_j^o &= e_{pj}^o f_j^{\ 'o}(Neta_{pj}^{\ o}) \\
\delta_j^o &= \sum_{k=1}^{M} \delta_{pk}^{\ s} w_{kj}^{\ s} f_j^{\ 'o}(Neta_{pj}^{\ o})
\end{aligned}
\qquad (3.18)
$$

Para actualizar las neuronas de la capa oculta y de la capa de salida necesitamos calcular las derivadas de las funciones de activación, esto nos trae una condición cuando estamos entrenando redes neuronales multicapa y es la necesidad de tener funciones de activación derivables.

En la tabla 3.1 mostramos las derivadas de las funciones de activación más representativas

Función Lineal	*Función Sigmoidal*
$f(neta) = neta$ $f'(neta) = 1$	$f(neta) = \frac{1}{1+e^{-neta}}$

	$f'(neta) = f(neta)(1 - f(neta))$
Función Tangente-Sigmoidal $f(neta) = \frac{2}{1+e^{-neta}} - 1$ $f'(neta) = 0.5 * (1 + f(neta))(1 - f(neta))$	*Función ReLU* $f(neta) = \max(0, neta)$ $f'(neta) = \begin{cases} 1 & \text{si neta} \geq 0 \\ 0 & \text{si neta} < 0 \end{cases}$

Cuadro 3.1 Pasos del Algoritmo GDE en línea para entrenar una red MLP superficial

1. Inicializamos los pesos del MLP.
2. Aplicamos un vector de entrada $\boldsymbol{x_p} = [x_{p1}, x_{p2}, \ldots, x_{pi}, \ldots, x_{pn}]^T$.
3. Propagamos el vector de entrada por la la red hasta generar la salida y_{pk}.
4. Calculamos los términos de error para las unidades de salida.

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

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

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

6. Actualizamos los pesos en la capa de salida.

$$w_{kj}^{\ s}(t+1) = w_{kj}^{\ s}(t) + \alpha \frac{2}{M} \delta_{pk}^{o} y^{o}{}_{pj}$$

7. Actualizamos pesos en la capa oculta.

$$w_{ji}^{\ o}(t+1) = w_{ji}^{\ o}(t) + \alpha \frac{2}{M} \delta_{j}^{o} x_{pi}$$

8. Si el conjunto de entrenamiento no es muy grande, podemos verificar si

la función de pérdida total cumple con el criterio de mínimo valor para detener el algoritmo L_{min},

$$L = MSE = \frac{1}{PM}\sum_{p=1}^{P}\sum_{k=1}^{M}(d_{pk} - y_{pk})^2 < L_{\min}$$

Nota: En GDE el criterio de parada generalmente es el número de iteraciones

3.3 Variaciones del gradiente descendente

El Algoritmo de gradiente descendente 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 que depende del problema que estemos resolviendo. 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. La ecuación 3.19 muestra la expresión para actualizar los pesos de una red neuronal considerando el gradiente descendente básico.

$$\Delta w(t) = \left(-\alpha \frac{\partial L}{\partial w(t)} \right) \qquad (3.19)$$
$$w(t+1) = w(t) + \Delta w(t)$$

3.3.1 Algoritmo Gradiente Descendente con Alfa Variable

Uno de los inconvenientes que presenta el algoritmo básico de gradiente descendente 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 cómo evolucione la función de pérdida durante el entrenamiento, con base en la regla descrita en la ecuación 3.20.

$$\alpha(t+1) = \begin{cases} \rho\alpha(t), si & L(w(t+1)) < L(w(t)) \\ \sigma(t), si & L(w(t+1)) \geq L(w(t)_k) \end{cases} \qquad (3.20)$$

Donde, ρ>1 y 0<σ<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).

La variación que acabamos de presentar se suele usar si podemos usar todos los datos de entrenamiento pues, de esta manera, tendremos información global del proceso de aprendizaje para modificar la razón de aprendizaje. En caso de usar el GDE donde se usa para el entrenamiento un minilote extraído de todos los patrones de entrenamiento, esta estrategia no es aplicable pues al ser los minilote diferentes no es posible establecer una relación entre los valores de la función de pérdida para cada iteración

3.3.2 Algoritmo Gradiente Descendente con Momentum Clásico

Una forma de acelerar la convergencia en el proceso de aprendizaje es usar para la variación de los pesos, además del valor del gradiente, un término que dependa de la variación de pesos usada en la iteración anterior. Este término lo 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 la iteración anterior. El valor de β es recomendable tomarlo como positivo y menor que la unidad. La expresión para la variación de los pesos la mostramos en la ecuación 3.21

$$\Delta w(t) = \beta \Delta w(t-1) + \left(-\alpha \frac{\partial L}{\partial w(t)} \right) \qquad (3.21)$$

$$w(t+1) = w(t) + \Delta w(t)$$

En la figura 3.4 se observa como es el proceso de actualización de pesos usando la idea del momentum

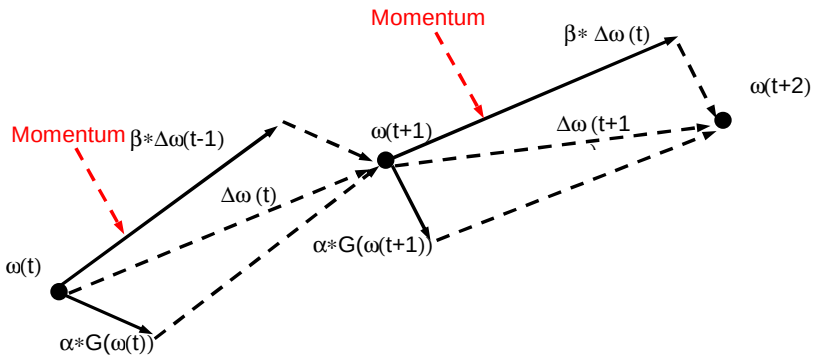

Fig. 3.4 Actualización de pesos usando el momentun clásico

Otro aspecto que deseamos mencionar cuando se implementa el momentum es la atenuación de las oscilaciones que puede tener la convergencia del algoritmo del gradiente descendente. En la figura 3.5 observamos el comportamiento oscilatorio alrededor del mínimo de la función de costo cuando solo se trabaja con el GD. El GD direcciona la actualización de pesos dependiendo de donde esté el mínimo. Si el mínimo está a la derecha del peso actual, el GD apunta en esa dirección. La actualización genera un cambio de peso esa dirección de tal manera que el peso actualizado se "pasa" del mínimo. En la siguiente iteración, el GD corrige la dirección de actualización de pesos apuntando hacia la izquierda, el peso se modifica en esa dirección pero se vuelve a "pasar" y se puede quedar en este proceso produciéndose una oscilación alrededor del mínimo.

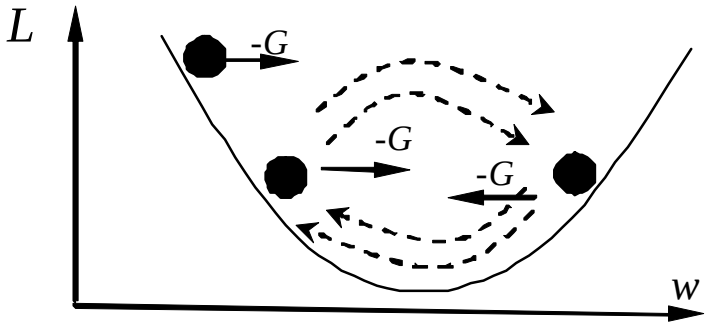

Fig. 3.5 Oscilaciones alrededor del mínimo producidas por el gradiente descendente

En la figura 3.6 observamos la atenuación de la oscilación que se produce alrededor del mínimo de la función de pérdida cuando se trabaja con el GD y el momentum. El GD direcciona la actualización de pesos dependiendo de donde esté el mínimo. Si el mínimo está a la derecha del peso actual, el GD apunta en esa dirección, si el peso no ha quedado a la derecha del mínimo el momento apunta en la misma dirección del gradiente y se produce una aceleración del

algoritmo de aprendizaje. La actualización genera un cambio de peso esa dirección de tal manera que el peso actualizado se "pasa" del mínimo. En la siguiente iteración, el GD corrige la dirección de actualización de pesos apuntando hacia la izquierda sin embargo, el momentum apunta en dirección contraria produciendo un efecto de atenuación o frenado en dicha actualización. El peso se modifica en esa dirección, se vuelve a "pasar" pero queda más cerca del mínimo. En la siguiente iteración, nuevamente el gradiente y el momentum se contraponen generándose nuevamente el efecto de frenado. El resultado es que la oscilación se atenúa y se tiene una convergencia más suave hacia el mínimo de la función de pérdida.

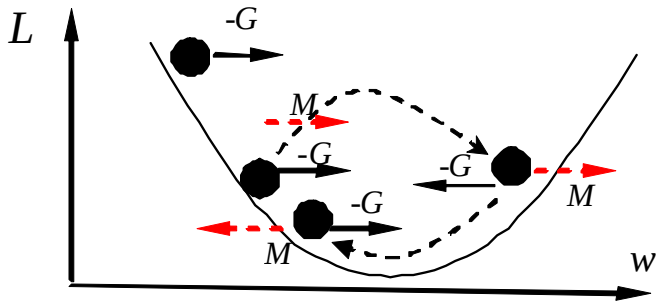

Fig. 3.6 Atenuación de las oscilaciones alrededor del mínimo cuando se usa gradiente descendente con momentum

3.3.3 Algoritmo Gradiente Descendente con Momentum Nesterov

Cuando se aplica el momemtum clásico, el gradiente se calcula respecto al valor actual de los pesos. El momento de Nesterov propone una sutil variación que consiste en modificar el vector de pesos de acuerdo al momentum que tiene el algoritmo y calcular el gradiente respecto a ese vector de pesos ya modificado.

$$\Delta w(t) = \beta \Delta w(t-1) + \left(-\alpha \frac{\partial L}{\partial (w(t) + \beta v(t))} \right) \qquad (3.22)$$

$$w(t+1) = w(t) + \Delta w(t)$$

En la figura 3.7 observamos como es el proceso de actualización de pesos usando el momentum con la variación de Nesterov

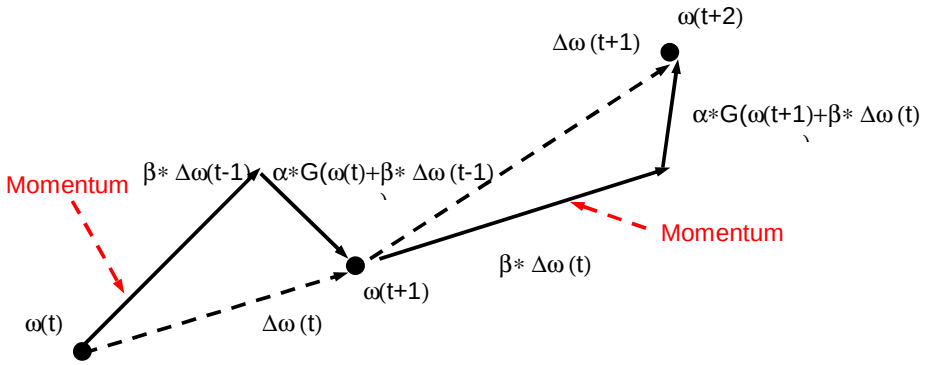

Fig. 3.7 Actualización de pesos usando el momentun con variacion de Nesterov

3.3.4 Algoritmo Gradiente Descendente tipo AdaGrad

Hemos visto que unos de los problemas del gradiente descendente es que se trabaja con una razón de aprendizaje fija. El algoritmo AdaGrad busca que la razón de aprendizaje se modifique en el transcurso del proceso de entrenamiento. Esto se logra considerando la sumatoria durante el entrenamiento de las contribuciones al gradiente de cada parámetro.

Esto garantiza que la razón de aprendizaje se ajuste de acuerdo a esta acumulación. Si la acumulación es poca, significa que el gradiente ha sido bajo y se necesita incrementar la razón de aprendizaje para que dichos parámetros se modifiquen. Si la acumulación es mucha, significa que el parámetro ha estado en una zona de alto gradiente y la razón de aprendizaje se disminuye para evitar divergencias en el entrenamiento.

La ecuación 3.23 muestra las expresiones para el AdaGrad. La variable AG se usa para acumular el gradiente que se han ido generando. Esta variable se inicializa en cero y va sumando los cuadrados de los gradientes de cada iteración. α_0 es el valor de la razón de aprendizaje global, δ es una constante pequeña que se usa para estabilidad numérica pues la acumulación del gradiente puede ser cercana a cero y la razón de aprendizaje se dispararía a un valor muy grande

$$
\begin{aligned}
AG(t) &= AG(t-1) + \left(\frac{\partial L}{\partial w_i(t)} \right)^2 \\
\alpha_i(t) &= \frac{\alpha_0}{\delta + \sqrt{AG(t)}} \\
\Delta w_i(t) &= \left(-\alpha_i(t) \frac{\partial L}{\partial w_i(t)} \right) \\
w_i(t+1) &= w_i(t) + \Delta w_i(t)
\end{aligned}
\tag{3.23}
$$

3.3.5 Algoritmo Gradiente Descendente tipo RMSProp

Aunque el AdaGrad es una variación del gradiente descendente con buenos resultados cuando se aplica a redes profundas, tiene el problema que la razón de aprendizaje suele tender rápidamente a valores pequeños lo cual hace que el algoritmo de entrenamiento se estanque. Esto ocurre porque se acumula todos los gradientes que se han generado desde el comienzo del aprendizaje. El RMSProp propone para esta desventaja una estrategia que hace que los gradientes lejanos de la iteración actual tengan poco efecto en la acumulación del gradiente. Esto se hace con una un parámetro o razón de decaimiento ρ.
La ecuación 3.24 muestra las expresiones para el RMSProp. La variable AG se usa para acumular el gradiente que se han ido generando. Esta variable se inicializa en cero y va sumando los cuadrados de los gradientes de cada iteración. α_0 es el valor de la razón de aprendizaje global, δ es una constante pequeña que se usa para estabilidad numérica pues la acumulación del gradiente puede ser cero y la razón de aprendizaje se dispararía a valores muy grandes. ρ es la razón de decaimiento que hace que el efecto de los gradientes lejanos se vaya perdiendo .

$$
\begin{aligned}
AG(t) &= \rho * AG(t-1) + (1-\rho) * \left(\frac{\partial L}{\partial w_i(t)} \right)^2 \\
\alpha_i(t) &= \frac{\alpha_0}{\delta + \sqrt{AG(t)}} \\
\Delta w_i(t) &= \left(-\alpha_i(t) \frac{\partial L}{\partial w_i(t)} \right) \\
w_i(t+1) &= w_i(t) + \Delta w_i(t)
\end{aligned}
\tag{3.24}
$$

3.3.5 Algoritmo Gradiente Descendente tipo AdaDelta

Otra alternativa para contrarrestar las desventajas del AdaGrad es la propuesta en el algoritmo denominado AdaDelta. En este algoritmo se usa la idea del algoritmo RMSProp de utilizar una razón de decaimiento para la acumulación del gradiente. El AdaDelta usa también la acumulación de las actualizaciones de pesos que se hayan ido generando. El incluir este término permite que la actualización de los pesos quede en la mismas "escala" de los pesos. Esto se hace al tratar de emular de una manera aproximada lo que hacen los métodos de segundo orden que veremos en el próximo apartado.
En la ecuación 3.25 mostramos las expresiones para el AdaDelta. La variable AG representa la acumulación del gradiente al cuadrado que se maneja con una razón de decaimiento ρ. Para tener una variación de pesos en la escala adecuada, los autores del AdaDelta propone usar la acumulación de las actualizaciones al cuadrado. Esta se almacena en la variable $A\Delta\omega$. A la acumulación de las actualizaciones ($A\Delta\omega$.) se le aplica la misma razón de decaimiento ρ que se le aplica a la acumulación de los gradientes
Con las acumulaciones se puede calcular el RMS de las mismas para generar el valor de la variación del peso ($\Delta\omega$) con el cual se actualiza los parámetros de la red.

$$AG(t) = \rho * AG(t-1) + (1-\rho) * \left(\frac{\partial L}{\partial w_i(t)} \right)^2$$

$$\Delta w_i(t) = \left(- \frac{RMS\left[\Delta w(t-1)\right]}{RMS\left[G(t)\right]} \right) \left(\frac{\partial L}{\partial w_i(t)} \right) \qquad (3.25)$$

$$A\Delta w(t) = \rho * A\Delta w(t-1) + (1-\rho) * \left(\Delta w_i(t)\right)^2$$

$$w_i(t+1) = w_i(t) + \Delta w_i(t)$$

Donde

$$RMS\left[\Delta w(t-1)\right] = \sqrt{A\Delta w(t-1)}$$

$$RMS\left[G(t)\right] = \delta + \sqrt{AG(t)} \qquad 3.26$$

4. ALGORITMOS DE SEGUNDO ORDEN PARA REDES NEURONALES MLP

Los algoritmos basados en el gradiente descendente resultan ser un método de entrenamiento lento comparado con algoritmos que involucran la segunda derivada

de la función de pérdida o costo. Uno de los métodos más usados es el método de Newton que presentamos en la siguiente sección

4.1 Método de Newton Aplicado a Redes Neuronales

El método de Newton surge originalmente para encontrar las raíces de un polinomio de manera iterativa. En la ecuación 3.27 se muestra la ecuación de Newton para estimar de manera iterativa las raíces o ceros de una función f(x). Para actualizar la estimación de la raíz, se necesita evaluar la estimación actual en la función y en la derivada de la función.

$$x_{raiz}(t+1) = x_{raiz}(t) - \frac{f(x_{raiz}(t))}{f'(x_{raiz}(t))} \qquad (3.27)$$

Podemos usar la ecuación 3.27 y utilizarla para encontrar el mínimo de una función. Recordemos que un mínimo es la raíz o un cero de la derivada de la función.
Con esta idea podemos escribir la expresión de la ecuación 3.28 donde mostramos la ecuación de Newton para estimar de manera iterativa los mínimos de la función f(x) o lo que es equivalente, las raíces o ceros de la función f'(x). Para actualizar la estimación del mínimo, necesitamos evaluar la estimación actual en la derivada y en la segunda derivada de la función a la cual le deseamos encontrar el mínimo.

$$x_{\min}(t+1) = x_{\min}(t) - \frac{f'(x_{\min}(t))}{f''(x_{\min}(t))} \qquad (3.28)$$

Con base en esta ecuación podemos inferir la ecuación 3.29, donde vamos a encontrar los pesos que minimicen la función de pérdida ***L*** en el espacio de los pesos sinápticos representado por la matriz ***W***.

$$W_{\min}(t+1) = W_{\min}(t) - \frac{L'}{L''} \qquad (3.29)$$

La segunda derivada de la función de pérdida corresponde a la Matriz Hessiana ***H*** y la primera derivada la conocemos como el vector gradiente G.

$$W_{\min}(t+1) = W_{\min}(t) - H^{-1}G \qquad (3.30)$$

En las ecuaciones 3.31 y 3.32 mostramos las expresiones para el gradiente (G) y la matriz Hessiana (H)

$$G = \left(\frac{dL}{dW(t)} \right) = \begin{bmatrix} \frac{\partial L}{\partial w_1(t)} \\ \frac{\partial L}{\partial w_2(t)} \\ \cdots \\ \frac{\partial L}{\partial w_N(t)} \end{bmatrix} \quad (3.31)$$

$$\mathbf{H} = \frac{d}{dW}\left(\frac{dL}{dW(t)} \right) = \begin{bmatrix} \frac{\partial^2 L}{\partial^2 w_1} & & \frac{\partial^2 L}{\partial w_1 w_j} \\ . & & . \\ . & \frac{\partial^2 L}{\partial w_i w_j} & . \\ . & & .. \\ \frac{\partial^2 L}{\partial w_i w_1} & & \frac{\partial^2 L}{\partial^2 w_N} \end{bmatrix} \quad (3.32)$$

El vector gradiente y la matriz Hessiana de la función de pérdida 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 de la función de pérdida L con respecto a cada uno de los pesos 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 w_i y w_j.

Para redes superficiales es posible usar el método de Newton y algoritmos que involucren la matriz Hessiana o estimaciones de ella como los algoritmos de gradiente conjugado o el entrenamiento basado en una técnica de optimización denominada Levenberg-Marquard.

4.2 Levenberg Marquardt

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

$$\mathbf{W}(t+1) = \mathbf{W}(t) - (\mathbf{H} + \lambda \mathbf{I})^{-1}\mathbf{G} \quad (3.33)$$

Para el 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 $\boldsymbol{H}$ el valor más grande y tomarlo como el valor inicial deλ.

El parámetro λ debe reducirse, si existe una tendencia a disminuir la función de pérdida producida por los pesos estimados en la ecuación 3.33, en caso contrario debemos incrementarlo.

Para las redes profundas, que son el tema de estudio de este libro, trabajar con métodos de segundo orden se vuelve impráctico pues los cálculos relacionados con la matriz Hessiana se vuelven complejos tanto en capacidad de memoria como en el tiempo necesario para realizarlos. Por ejemplo una red neuronal con un millón de conexiones, la matriz Hessiana tendría un tamaño de [1.000.000 x 1.000.000], es decir se tendría un billón de datos para calcular que ocuparían aproximadamente en RAM, 3725 gigabytes.
Otro problema que se presenta con estos algoritmos en redes profundas es que no se ha logrado que trabaje adecuadamente usando minilotes por lo que se requiere el lote completo de entrenamiento para un correcto funcionamiento. Hacer que estos algoritmos de segundo orden trabajen bien con minilotes es un área activa de investigación.