4. EL PERCEPTRÓN MULTICAPA

4.1. INTRODUCCIÓN

El perceptrón multicapa o multinivel es un red neuronal *feedforward* con una o varias capas ocultas entre la de entrada y la de salida. El perceptrón multicapa es una generalización del perceptrón simple que es capaz de clasificar patrones que no son linealmente separables. Son las capas ocultas los que dan a la red la capacidad de aproximar cualquier tipo de función o relación entre las entradas y las salidas de la red neuronal convirtiendo así al perceptrón multicapa es una red neuronal de propósito general, flexible y no lineal.

En la Figura 4.1 se muestra un ejemplo de esta red neuronal con cuatro capas, esto es, una capa de entrada, una de salida y dos capas ocultas.

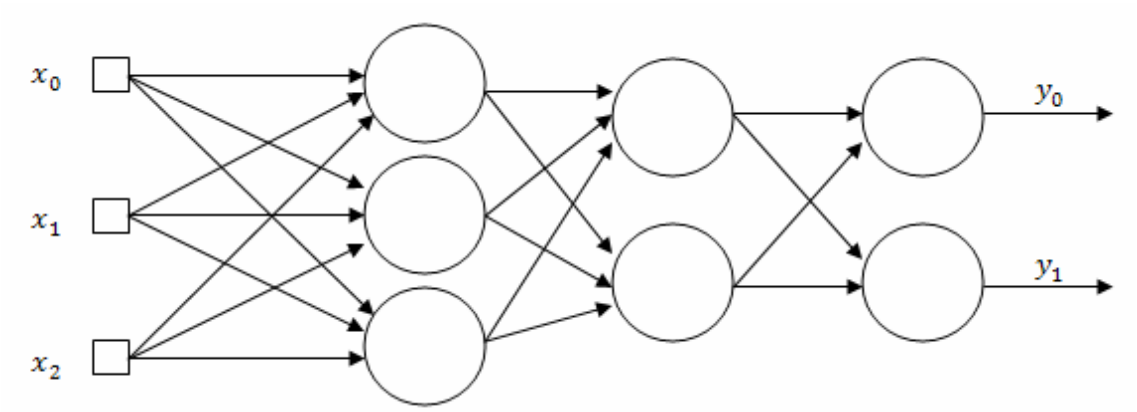

Figura 4.1: Modelo de un Perceptrón Multicapa.

El diseño de la arquitectura del perceptrón multicapa, es decir, la elección del número de capas y del número de neuronas de cada capa junto con el tipo de función de activación queda en manos del investigador y de su experiencia. El comportamiento final de la red dependerá tanto de estos parámetros como del aprendizaje, por tanto es importante saber cómo diseñar la red multicapa y como entrenarla correctamente. En el siguiente apartado se verá unas cuantas consideraciones básicas en cuanto al diseño de la red neuronal y en apartados posteriores se abarcará el entrenamiento de la red.

4.2. DISEÑO DE LA RED NEURONAL

Uno de los problemas al trabajar con redes neuronales es que es difícil determinar a priori la mejor arquitectura para resolver un problema dado. Normalmente se experimenta con distintas redes y se elige el que ofrece mejores resultados. En la práctica las redes neuronales realizan correctamente la tarea que ha aprendido durante el entrenamiento, sin embargo, tras dicho entrenamiento es difícil explicar o interpretar cómo funciona la red.

Aún así, existen algunas consideraciones que nos pueden ayudar a la hora de diseñar una red neuronal:

Número de capas ocultas:

El número de capas a elegir dependerá del caso, aunque normalmente con una capa oculta es suficiente. Una red neuronal de tres capas será capaz de aprender una función formada por un número finito de puntos o funciones continuos definido en un dominio compacto, esto es, las entradas tienen un límite definido. Muchas funciones que no cumplan estas condiciones pueden ser aprendidas también por una red de tres capas bajo ciertas condiciones.

Si una función no puede ser aprendida por una red de tres capas, en general, podrá aprenderse con una de cuatro. Así un perceptrón multicapa con dos capas ocultas se puede considerar como un aproximador universal. En la práctica nos decantaremos por dos capas ocultas cuando nos encontremos con funciones continuas pero con algunas discontinuidades. Teóricamente nunca se necesitará más de dos capas ocultas.

El caso más común para que ningún perceptrón multicapa pueda aprender una función es cuando dicha función no está en un dominio compacto.

Número de neuronas:

El número de neuronas de la capa de entrada y de la de salida será el número de entradas y salidas que nuestro problema. Sin embargo la elección del número de neuronas en cada capa oculta no es tan trivial.

En general, el número de neuronas ha de ser lo suficientemente grande para que la red neuronal pueda aprender a realizar la tarea deseada. En contra, si el número de neuronas es demasiado elevado el tiempo de aprendizaje se alarga y en el peor de los casos nunca alcanzará los objetivos deseados. Otro problema de un número demasiado grande de neuronas es el llamado sobreentrenamiento. Esto se da cuando una red tiene una gran capacidad de procesamiento de información que aprende aspectos insignificantes e indeseados de los patrones mostrados durante el entrenamiento y la red pierde la capacidad de generalizar. Un ejemplo de este fenómeno se muestra en la Figura 4.2.

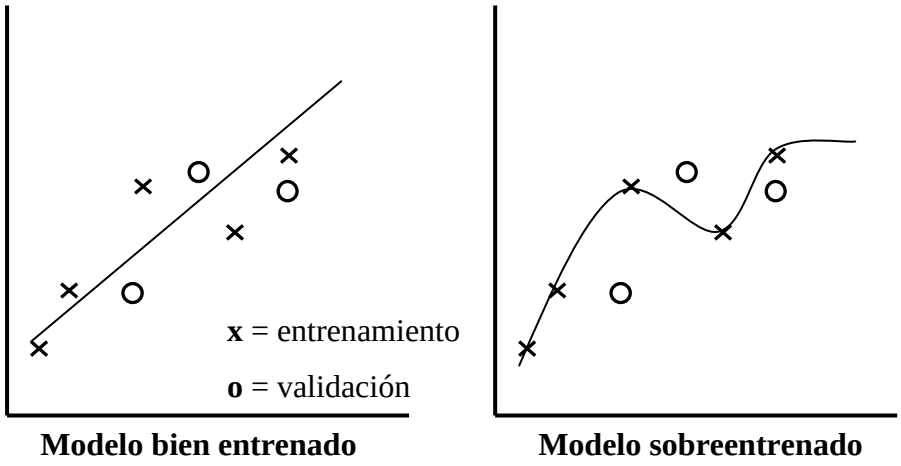

Figura 4.2: Ejemplo de sobreentrenamiento.

La mejor forma para proceder en la elección del número de neuronas en las capas ocultas es comenzar con un número pequeño de ellas, como por ejemplo 2 o 3 neuronas, entrenar la red y verificar que el comportamiento de la red es satisfactorio. En caso que no sea así, se incrementa ligeramente el número de neuronas y se vuelve a comprobar la validez de la red neuronal. Este procedimiento se repite hasta que el error obtenido de la red durante la validación es aceptable o no hay mejoras significantes. No nos interesará seguir aumentando el número de neurona ya que se perderá la habilidad de generalizar de la red.

Función de activación:

El algoritmo de aprendizaje que se empleará para entrenar la red es el *Backpropagation* o la regla delta generalizada. Esta técnica necesita que la función de activación sea continua, es decir, diferenciable. En la mayoría de los casos la función utilizada es una función sigmoidal. La función sigmoidal más popular es la función logística aunque también existen muchas otras como puede ser la tangente hiperbólica o la arcotangente.

La forma exacta de la función no tiene un gran efecto en el comportamiento final de la red neuronal, sin embargo, su influencia puede ser significante en la velocidad de convergencia durante el entrenamiento.

Ha de tenerse en cuenta que una función sigmoidal tiene un rango determinado y estas funciones nunca alcanzan estos valores extremos. Por ejemplo, si se emplea la función logística, las entradas han de ser normalizadas entre 0 y 1 y las salidas se considerarán activadas en torno al valor 0.9 y desactivadas en torno al valor 0.1.

4.3. ENTRENANDO LA RED

4.3.1. Introducción

El algoritmo de retropropagación o de propagación hacia atrás, más conocido como *Backpropagation*, fue el primer método práctico para el entrenamiento de una red neuronal multicapa con conexiones hacia adelante o *feedforward.*

Este es un algoritmo iterativo donde se realiza dos pasadas de cálculo por iteración. En la primera pasada se presenta un conjunto de patrones de entrenamiento a la red uno por uno y se calcula el error cometido por la red. En la segunda pasada se actualizan los pesos de la red neuronal en función del error cometido. La forma que se propaga el error a la hora de actualizar los pesos es hacia atrás, esto es, desde la capa de salida hacia la capa de entrada dando el nombre de retropropagación al algoritmo de aprendizaje.

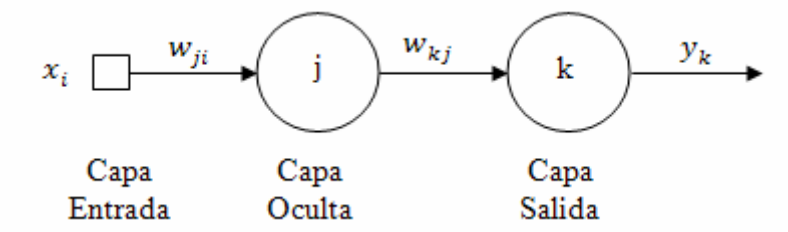

Figura 4.3: Conexiones entre las capas del perceptrón multicapa.

Dado un patrón de entrenamiento *p*, llamemos la respuesta deseada para una neurona *k* como d_{pk} y la respuesta observada por la red como o_{pk}. Si la red posee *n* neuronas de salida, el error para este patrón de entrenamiento es:

$$E_p = \frac{1}{n}\sum_{j=0}^{n-1}(d_{pk} - o_{pk})^2 \qquad \textit{Ec.(4.1)}$$

El error cometido por la red con *m* patrones de entrenamiento en una iteración es:

$$E = \frac{1}{m}\sum_{p=0}^{m-1}E_p \qquad \textit{Ec.(4.2)}$$

El *Backpropagation* es un algoritmo basado en el gradiente. El gradiente de una función determina el incremento más rápido de dicha función. Aplicado a nuestro caso, si tomamos el sentido contrario del gradiente del error, iremos decrementando el error. Entonces, para reducir el error cometido por la red neuronal se ajustará los pesos desde una neurona *i* a la siguiente neurona *j* en la dirección:

$$-\frac{\partial E}{\partial w_{ji}} \qquad \textit{Ec.(4.3)}$$

Este ajuste de pesos se repite todas las veces que sea necesario, y como siempre nos dirigimos en el sentido en el que decrece el error supondremos que alcanzaremos el mínimo error en algún momento. La velocidad y la precisión a la que alcanzamos ese mínimo dependen de la tasa de aprendizaje. Este parámetro es la distancia del paso tomado en la dirección del mínimo y si es demasiado pequeño la convergencia será muy lenta. Si, por el contrario, es demasiado elevado nunca convergerá debido a que el error oscilará en torno al mínimo.

Para reducir las oscilaciones durante el entrenamiento se añade un término de *momento*. Cada nueva dirección calculada es una suma ponderada del gradiente y la dirección tomada en la iteración anterior. Al disminuir el número de oscilaciones conseguimos aumentar la velocidad de convergencia. Nos interesa un valor alto de *momento* para disminuir lo máximo posible dichas oscilaciones, sin embargo, un valor demasiado alto supondría que el algoritmo no puede seguir los giros tan comunes en el espacio de los pesos.

4.3.2. El algoritmo Backpropagation

A continuación se presenta los pasos del algoritmo *Backpropagation* introducido en el apartado anterior detalladamente.

1) *Inicialización de los pesos:*

En este paso se inicializarán todos los pesos de la red de manera aleatoria y de pequeño valor.

2) *Presentación del patrón de entrenamiento:*

El patrón de entrenamiento presentado a la red estará formado por patrón de entrada $X_p = (x_0, x_1, ..., x_{N-1})$ y la salida deseada para esta entrada $d_p = (d_0, d_1, ..., d_{M-1})$, siendo *N* y *M* el número de entradas y salidas respectivamente.

3) *Cálculo de la salida:*

Se obtiene la salida de la red para la entrada presentada, para ello se van calculando los resultados intermedios capa a capa hasta llegar a la capa de salida hallando así el resultado de la red $y_p = (y_0, y_1, ..., y_{M-1})$.

Para el ejemplo de la Figura 4.3, primero se halla las salidas para la capa oculta, siendo la salida de una neurona *j*:

$$y_{pj} = f\left(\sum_{i=0}^{N-1} w_{ij} x_{pi} + \theta_j\right) \qquad \textit{Ec.(4.4)}$$

Se repite los cálculos con la capa de salida, que para una neurona *k*:

$$y_{pk} = f\left(\sum_{j=0}^{L-1} w_{kj} y_{pj} + \theta_k\right) \qquad \textit{Ec.(4.5)}$$

Siendo *L* el número de neuronas de la capa oculta.

4) *Cálculo del error para todas las neuronas:*

Si la neurona *k* es una neurona de la capa de salida, el valor de δ es:

$$\delta_{pk} = f'(net_{pk})(d_{pk} - y_{pk}) \qquad Ec.(4.6)$$

Si la neurona *j* es una neurona perteneciente a una capa oculta no se puede hallar directamente la δ, sino que necesitamos las *deltas* de la capa posterior. Si *k* se refiere a estas *deltas* de la siguiente capa, entonces, la δ para la neurona *j* se hallará de la siguiente manera:

$$\delta_{pj} = f'(net_j)\sum_k \delta_{pk} w_{kj} \qquad Ec.(4.7)$$

La función de activación, como se ha comentado en apartados anteriores y como se observa en las ecuaciones, ha de ser derivable. En general tendremos una función lineal o una función sigmoidal.

La derivada de la función lineal es: $f'(x) = 1$ y si se utiliza la función logística, la derivada es: $f'(x) = f(x)(1 - f(x))$.

5) *Modificación de los pesos:*

Al igual que en el cálculo de los errores, la actualización de los peso se realiza desde la capa de salida hacia atrás hasta la capa de entrada.

Los nuevos pesos para la capa de salida se obtienen de la forma:

$$w_{kj}(t+1) = w_{kj}(t) + \alpha\delta_{pk} y_{pj} \qquad Ec.(4.8)$$

Y para la capa oculta:

$$w_{ji}(t+1) = w_{ji}(t) + \alpha\delta_{pj}x_{pi} \qquad \textit{Ec.(4.9)}$$

El parámetro α es la tasa de aprendizaje con un valor entre 0 y 1 que se ajusta para controlar tanto la velocidad de aprendizaje de la red como la estabilidad en las estimaciones de los pesos.

Se puede añadir a las actualizaciones anteriores el término *momento* que para la capa de salida se halla de la siguiente forma:

$$\beta\left(w_{kj}(t) - w_{kj}(t-1)\right) \qquad \textit{Ec.(4.10)}$$

Y para la capa oculta:

$$\beta\left(w_{ji}(t) - w_{ji}(t-1)\right) \qquad \textit{Ec.(4.11)}$$

6) *Volver al paso 2:*

Los pasos del 2 al 5 se repiten de manera iterativa hasta que la red haya aprendido a realizar su tarea y el error cometido para cada patrón de entrenamiento sea menor de un valor deseado y los pesos de la red se estabilicen.

$$E_p = \frac{1}{M}\sum_{k=0}^{M-1}\delta_{pk}^2 \leq \varepsilon \qquad \textit{Ec.(4.12)}$$

4.3.3. Mínimos locales

La función de error forma una superficie donde cada punto corresponde a unos valores de los pesos de la red neuronal. El algoritmo de *Backpropagation* busca un valor

mínimo de esta función. Sin embargo, el algoritmo no siempre encuentra un mínimo global y puede detenerse en un mínimo local.

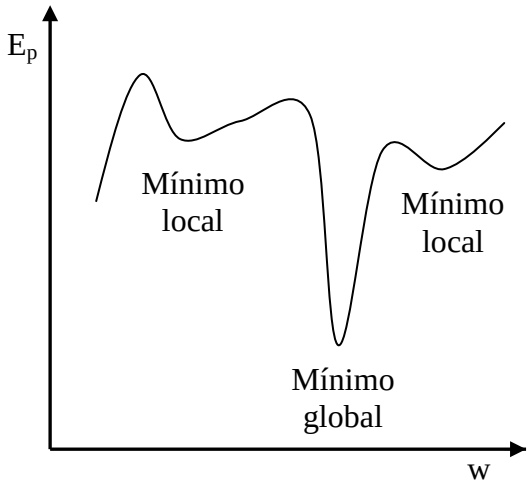

Figura 4.4: Ejemplo representativo de una superficie de error.

En la Figura 4.4 se muestra un ejemplo representativo de una superficie de error con sus mínimos. Como se observa en dicha figura, los mínimos locales abundan y su presencia es un problema a la hora de entrenar la red. Sin embargo, no es necesario encontrar el mínimo global para obtener una red que ofrezca buenos resultados, sino con un error mínimo preestablecido será más que suficiente.

4.4. VALIDANDO LA RED

Una red neuronal no se debe de poner en funcionamiento justo después de ser entrenado, primero ha de ser validado. Normalmente esto se lleva a cabo separando el conjunto de patrones o muestras conocidas en dos subconjuntos. Uno de estos

subconjuntos es utilizado para el entrenamiento, y el otro subconjunto servirá para validar la red.

Validar la red es tan importante o más que el entrenamiento. Una red puede dar buenos resultados con las muestras de entrenamiento y, sin embargo, dar unos resultados completamente erróneos en los patrones de validación. Esto se observa en el ejemplo de la Figura 4.5. Si la red se sobreentrena, ésta aprenderá detalles carentes de importancia de los patrones de entrenamiento en vez de la estructura básica de los datos. De esta forma, la red no generalizará correctamente.

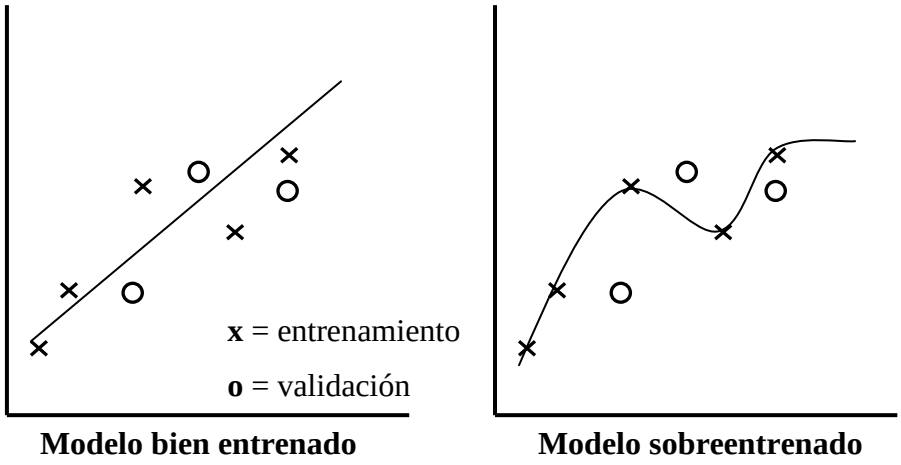

Figura 4.5: Diferencia entre un buen modelo y uno sobreentrenado.

Es normal que el error de los patrones de entrenamiento sea ligeramente menor que el de los patrones de validación. Sin embargo, si la diferencia es demasiado grande significa que los dos subconjuntos de patrones no son muestras representativos de la población o que se ha sobreentrenado la red.

Es importante no utilizar los patrones de validación para entrenar la red. Esto puede pasar por error si se es inexperto. Un ejemplo sería si, tras entrenar la red, la red neuronal no da buenos resultados en la validación y se vuelve a entrenar la red con una nueva inicialización de pesos. Esta nueva red neuronal se vuelve a probar con los

patrones de validación y si se obtiene buenos resultados se acepta la red, en caso contrario se vuelve a repetir el proceso hasta que la red de buenos resultados con los patrones de validación. Este método presenta el problema que esencialmente estamos utilizando los patrones de validación como patrones de entrenamiento.

Que la red no de buenos resultados durante la validación quiere decir que los patrones de validación poseen información esencial que la red no ha aprendido. Si este es el caso, se debería de añadir los patrones de validación a los de entrenamiento y volver a entrenar la red con todos los datos y se buscará nuevas muestras para la validación.

Si es absolutamente imposible obtener nuevas muestras para la validación se entrena con todos los patrones y se espera que la red neuronal funcione correctamente sin una validación. Entrenar con menos patrones que todas las que se tengan en este caso sería desechar información importante para el correcto aprendizaje de la red.

Una forma de validar la red sin la necesidad de utilizar patrones de validación para poder entrenar la red con todos los datos es con un método estadístico llamado *bootstrap* que se explicará en el siguiente apartado.