UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA

ESCUELA UNIVERSITARIA DE INGENIERÍA TÉCNICA DE TELECOMUNICACIÓN

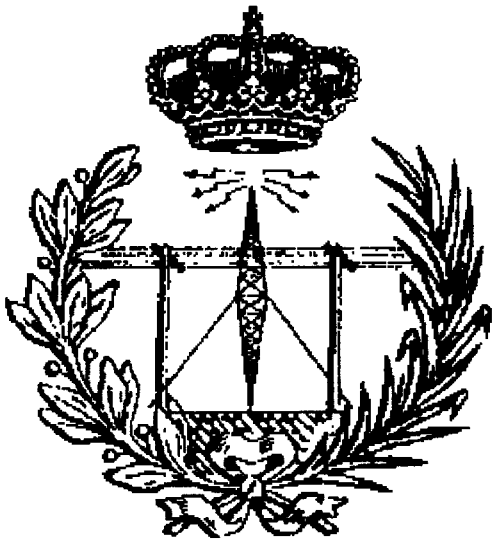
PROYECTO FIN DE CARRERA

LINEALIZACIÓN DEL ALGORITMO DE BACKPROPAGATION PARA EL ENTRENAMIENTO DE REDES NEURONALES

JAVIER JESÚS SÁNCHEZ MEDINA

Las Palmas de Gran Canaria, 1998

UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA
ESCUELA UNIVERSITARIA DE INGENIERÍA TÉCNICA DE TELECOMUNICACIÓN



PROYECTO FIN DE CARRERA:

LINEALIZACIÓN DEL ALGORITMO DE BACKPROPAGATION PARA EL ENTRENAMIENTO DE REDES NEURONALES.

ESPECIALIDAD: Telefonía y transmisión de datos.
AUTOR: Javier Jesús Sánchez Medina.
TUTORA: Itzíar Goretti Alonso González.

Septiembre, 1998.

UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA
ESCUELA UNIVERSITARIA DE INGENIERÍA TÉCNICA DE TELECOMUNICACIÓN

PROYECTO FIN DE CARRERA

TÍTULO:

LINEALIZACIÓN DEL ALGORITMO DE BACKPROPAGATION PARA EL ENTRENAMIENTO DE REDES NEURONALES.

ESPECIALIDAD: Telefonía y transmisión de datos.
AUTOR: Javier Jesús Sánchez Medina.
TUTORA: Itzíar Goretti Alonso González.
FECHA: Septiembre de 1998.

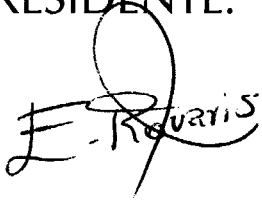
TUTORA:

Fdo: ITZIAR G. ALONSO.

AUTOR:

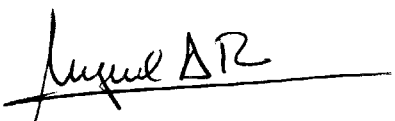
Fdo: JAVIER J. SÁNCHEZ M.

PRESIDENTE:



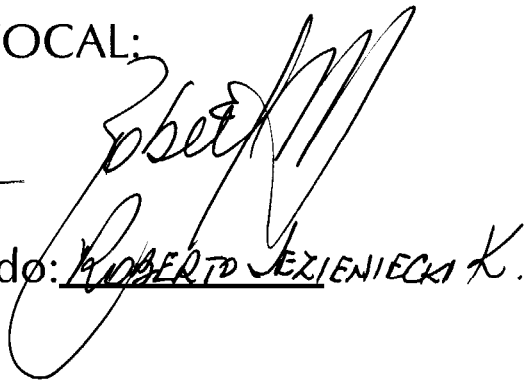
Fdo: E. ROVARIS

SECRETARIO:



Fdo: Miguel A Ferro

VOCAL:



Fdo: ROBERTO JEZIENIECKI K.

NOTA: SOBRESALIENTE (10)

A todo el que
sueña con un
futuro mejor.

- **DOCUMENTO I:**

  MEMORIA

- **DOCUMENTO II:**

  ANEXOS

- **DOCUMENTO III:**

  PRESUPUESTO

- **BIBLIOGRAFÍA**

ÍNDICE GENERAL

DOCUMENTO I: MEMORIA

DOCUMENTO II: ANEXOS

MEMORIA

CAPÍTULO 0: INTRODUCCIÓN.

0.1 Objetivos.

El objetivo de este proyecto de fin de carrera es estudiar un nuevo algoritmo de entrenamiento de Redes Neuronales el cual ha sido propuesto por los investigadores alemanes Siegfried Ergezinger y Erk Thomsen denominado OLL (Optimization Layer by Layer). Ver [5]. Este se basa en el ajuste de la red neuronal capa a capa y en la linealización del algoritmo Backpropagation clásico.

En [5] este algoritmo se aplicó a predecir muestras en una serie de números.

Así mismo también es objetivo de este trabajo de investigación comparar la velocidad y precisión de este nuevo algoritmo con el clásico de entrenamiento de la red neuronal Perceptrón Multicapa, el Backpropagation (BP).

Esta comparación se llevará a cabo mediante la implementación de una prueba concreta. Como se verá, se trata de mejorar la tasa de reconocimiento de patrones de voz, concretamente de los dígitos del 0 al 9, añadiendo una Red Neuronal a la salida de los modelos ocultos de Marcov.

Todas las pruebas efectuadas en este proyecto se llevaron a cabo en el lenguaje de programación MATLAB, en su versión 5.0, aprovechando su gran versatilidad en el manejo de matrices y su abundante librería de funciones.

0.2 Estructura de la Memoria.

La memoria está compuesta por cuatro capítulos.

- CAPÍTULO 1: Una breve introducción a las Redes Neuronales, en especial al Perceptrón Multicapa (MLP).
- CAPÍTULO 2: Se explica el algoritmo clásico de entrenamiento del MLP, el Backpropagation
- CAPÍTULO 3: Explicación del nuevo algoritmo, objeto principal de este estudio que es el algoritmo OLL..
- CAPÍTULO 4: Se aplicacian tanto el nuevo algoritmo (OLL) y el clásico (Backpropagation) a un problema concreto. Se recogen los resultados de las pruebas realizadas en tablas y gráficos, y se realiza un análisis comparativo entre los dos métodos.

Fuera ya de la memoria se añaden tres documentos más:

- Los anexos necesarios con los archivos de Matlab utilizados.
- El presupuesto del proyecto.
- Y la bibliografía consultada.

CAPÍTULO 1: INTRODUCCIÓN A LAS REDES NEURONALES.

Existe una cuestión que se plantean los programadores, ingenieros y diseñadores de computadoras: ¿Por qué es tan difícil que una máquina que efectúa cientos de millones de operaciones en coma flotante por segundo sea capaz de distinguir entre dos objetos, o reconocer un patrón de voz, etc...?

Sería deseable que esta máquina, tan potente para ciertas funciones fuera capaz de aprender a partir de la experiencia, tal y como aprendemos nosotros mismos. Que no fuera necesario introducir por el programador todas las posibles situaciones.

Hay muchas aplicaciones que pueden ser susceptibles de resolución con computadoras convencionales: resolución de cálculos matemáticos y problemas científicos; creación, mantenimiento y gestión de bases de datos; comunicaciones electrónicas; procesamiento de textos, gráficos, imágenes y un largo etcétera.

Por el contrario hay muchas aplicaciones que se han intentado automatizar resultando extraordinariamente complicada la resolución con estas máquinas. No es imposible en la mayoría de los casos, sino que con computadoras secuenciales se convierte en una tarea extremadamente acaparadora de recursos. Como la única herramienta de la que disponíamos era el computador secuencial, intentábamos encontrar la solución con está máquina por todos los medios.

Ahora surge una cuestión. Las transmisiones nerviosas en nuestro cerebro van a velocidades de hasta séptimo orden inferior al tiempo de conmutación en un ordenador secuencial. ¿Cómo es posible que nosotros seamos capaces de distinguir un lápiz de un bolígrafo en centésimas de segundo y un ordenador secuencial sea incapaz o tarde muchísimo tiempo en conseguirlo?

Procesamiento paralelo masivo. Esa es la clave y la gran ventaja de nuestro cerebro respecto de cualquier computadora secuencial. El gran paralelismo y la interconectabilidad que se observan en los sistemas biológicos complejos son las causas de la capacidad del cerebro para llevar a cabo complejos reconocimientos de tramas en unos pocos centenares de milisegundos.

En muchos aspectos de la vida se desearía la automatización de complejos problemas de reconocimiento de tramas. Para intentar resolverlos se utilizan unas estructuras que se denominan **Redes Neuronales** (Neural Networks, NN's).

Para empezar se define las estructuras de Redes Neuronales como colecciones de procesadores paralelos conectados entre sí. Tomando la figura 1.1 como referencia de una red típica, se puede representar esquemáticamente cada **elemento de procesamiento** de la red como un **nodo** o **neurona**, indicando las conexiones entre nodos mediante **flechas**. Las flechas apuntan la dirección de flujo

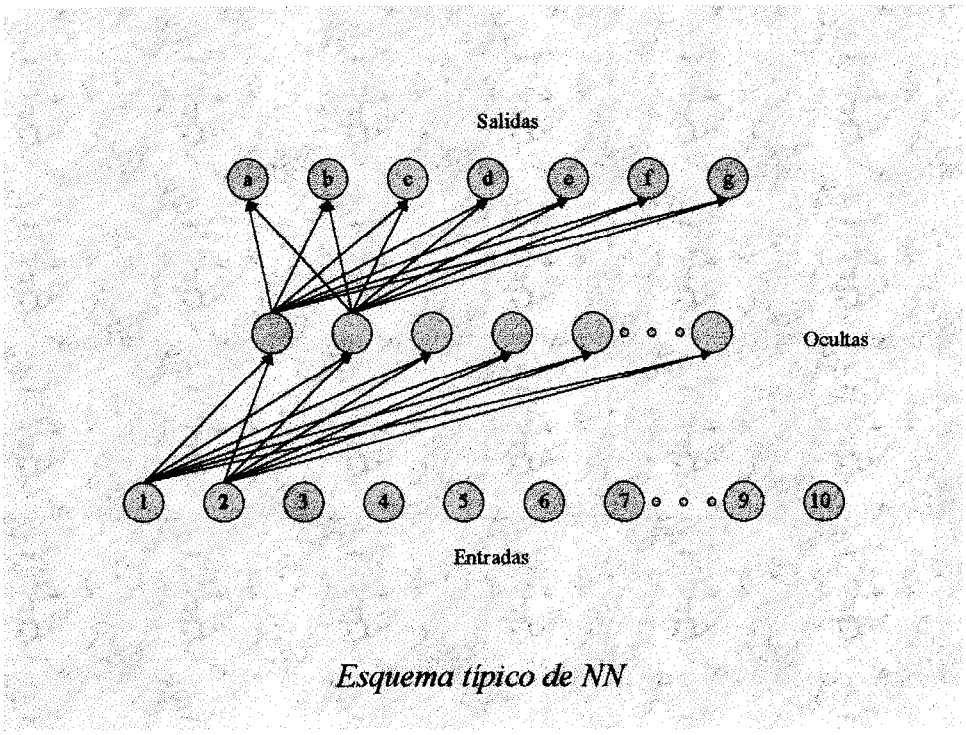


Figura 1.1

de las señales. En la figura 1.1 podemos apreciar que hay una capa de entrada, unas capas ocultas y una capa de salida. El número de nodos de la capa de entrada ha de ser tal que permita la adquisición de todos los parámetros de entrada. Por ejemplo tantos nodos de entrada como bits tenga el número binario que se pretende utilizar como entrada. El número de nodos de la capa de salida depende de la información de salida que se desee obtener. Por ejemplo, si se desea que nos indique si la entrada pertenecía a una de cinco clases o grupos, necesitamos cinco nodos de salida. El número de capas ocultas y los nodos de cada una, dependiendo de la aplicación se estudia para que el **aprendizaje** o **entrenamiento** sea lo más óptimo posible.

El proceso de **aprendizaje** o **entrenamiento** se explica seguidamente. Los nodos se conectan utilizando **conexiones ponderadas** y se entrena la red mediante ejemplos de pares de datos. Este concepto de aprendizaje mediante ejemplos es muy importante. Una gran ventaja de la aproximación mediante NN's a la resolución de problemas es que no es necesario tener un proceso bien definido para transformar algorítmicamente una entrada en una salida. Más bien, lo que se necesita para la mayoría de las redes es una colección de ejemplos representativos

de la traducción deseada. La NN se adapta entonces de forma que cuando se le presenten las entradas dadas como ejemplo, producirá la salida deseada.

Además es incluso capaz de reconocer entradas que no sean exactamente las dadas como ejemplos, sino que presenten ruido o que no sean las utilizadas como ejemplo pero pertenezcan a la misma clase. Si se reflexiona es evidente que esto es importantísimo en reconocimiento de patrones de voz, o caracteres escritos por ejemplo. Nunca una de estas entradas va a ser idéntica a otra, sino a lo sumo bastante parecida.

Es tarea del entrenamiento el que la red quede lo suficientemente "holgada" como para que reconozca entradas con ligeras variaciones en la clase que corresponde; y lo suficientemente "ajustada" como para que no se confunda y clasifique una entrada en la clase que no es. Estas dos peculiaridades confrontadas se denominan respectivamente generalidad y fidelidad.

El proceso de entrenar la red es simplemente cuestión de modificar los pesos de conexión sistemáticamente, para codificar las relaciones de entrada-salida deseadas.

1.1 Circuitos neuronales.

En la figura 1.2 se muestran varios circuitos neuronales básicos que se encuentran en el sistema nervioso central. La figura 1.2 a y b ilustra los principios de convergencia y divergencia en la circuitería neural. Cada neurona envía impulsos a muchas otras neuronas (divergencia) y recibe impulsos procedentes de muchas neuronas (convergencia). Esta sencilla idea parece ser el fundamento de toda la actividad del sistema nervioso central, y forma la base de la mayoría de los modelos de Redes Neuronales.

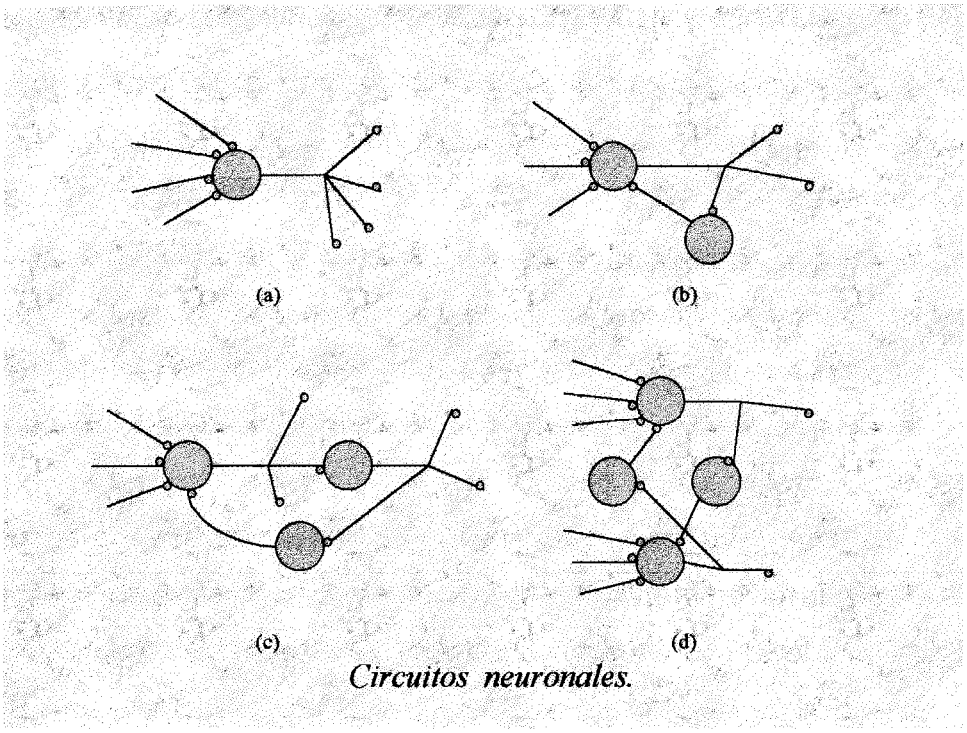


Figura 1.2

1.2 Aprendizaje de Hebb.

Los sistemas neuronales de nuestro cerebro no nacen con el conocimiento y las capacidades que puede llegar a tener. Es necesario un proceso de aprendizaje a lo largo de nuestra vida para adquirir aquellas. La pregunta que surge es: ¿Cómo aprendemos?

La teoría básica procede de un libro que escribió Hebb en 1949, *Organization of behaviour* y se puede extraer de [1]. La idea principal es la siguiente:

> Cuando un axón de la célula A está suficientemente próximo para excitar a una célula B o toma parte en su disparo de forma persistente, tiene lugar algún proceso de crecimiento o algún cambio metabólico en una de las células, o en las dos, de tal modo que la eficiencia de A como una de las células que desencadena el disparo de B, se ve incrementada.

Esta ley de aprendizaje aparece de una forma u otra en muchas redes neuronales de la actualidad. En concreto, el Perceptrón Multicapa que es la red utilizada en este proyecto se inspira en este principio.

1.3 Funciones y aplicaciones de las Redes Neuronales.

Antes de entrar en las interioridades de los modelos de redes neuronales conviene perfilar claramente cuales son sus funciones y aplicaciones.

Funciones:

- Establecer correspondencias de carácter general (vector de salida cualquier función no lineal del vector de entrada); así se pueden emplear para filtrado (extraer una señal deseada de un registro ruidoso), modelado (simular un sistema con relación entrada-salida desconocida), etc.
- Tomar decisiones en virtud de la clase a que pertenezca una entrada; es decir, resolver problemas de clasificación; de tal modo que se ha extendido su uso en diagnóstico (establecer una decisión a partir de la aparición de ciertos síntomas), reconocimiento (determinar si una forma corresponde o no a una cierta familia), etc.
- Establecer asociaciones, sean autoasociaciones (de un vector degradado con una prototípico) o heteroasociaciones (de un conjunto de vectores con otros que representan clases o subclases); por ello, cabe emplearlas en restauración (de una información degradada), extracción de características (presentes en una cierta entrada), etc.
- Por último también pueden llevar a cabo trabajos de optimización, haciendo corresponder las variables que intervienen en la función a optimizar a los parámetros de la NN, y obligando a evolucionar a esta en el sentido debido; siendo de gran utilidad en planificación, diseño, etc.

Aplicaciones:

- Procesado de señales y datos:
    - Tratamiento de la imagen
    - Tratamiento de la voz
    - Bioingeniería
    - Control
    - Robótica
    - Comunicaciones
    - Radar,sonar,etc.
- Toma de decisiones:
    - Medicina
    - Negocios
    - Gestión.
- Optimización de problemas técnicos o metodológicos:
- Actuación inteligente, como interfase inteligente y en simulación, formación y entrenamiento.

1.4 El modelo de nodo o neurona.

Los elementos individuales de cálculo que forman la mayoría de los modelos de Redes Neuronales se les denomina nodos, neuronas, unidades o elementos de procesamiento (PE's).

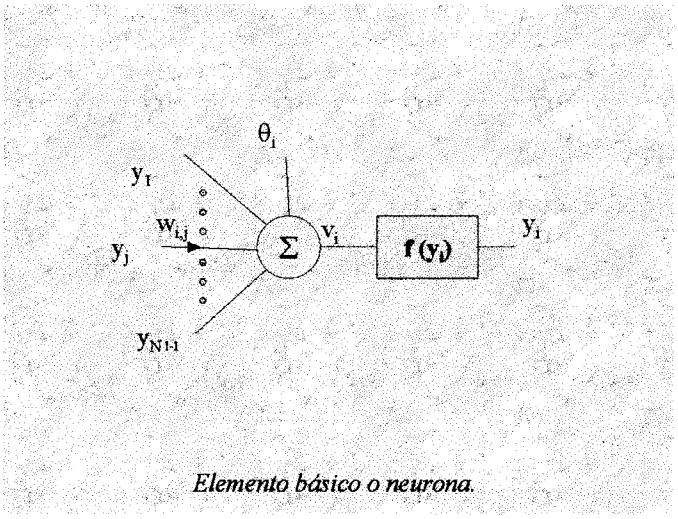


Figura 1.4

En la figura 1.4 se muestra un modelo general de nodo. Cada nodo ejecuta una función dada por la ecuación 1.1.

$$v_i = \sum_{j=1}^{N} w_{i,j} y_j + \theta_i \qquad (a)$$

$$y_i = f(v_i) \qquad (b)$$

Ecuación 1.1

En la ecuación 1.1:

- v_i: entrada a la función de activación del nodo o neurona i de una capa l.
- $w_{i,j}$: peso de la conexión entre la neurona j de la capa l-1 y la neurona i de la capa l.
- y_j : salida de la neurona j de la capa l-1.
- θ_i: offset de la neurona i de la capa l.
- y_i: salida del nodo o neurona i de una capa l.
- f: función de activación de la neurona.

En cada neurona se llevan a cabo dos operaciones elementales: función de base y función de activación.

La función de base es la operación que se efectúa a las señales de entrada a la neurona. Por ejemplo se podría efectuar la suma de todas las señales de entrada, o el producto, o la media etc...

A rasgos generales se puede ver cada neurona como un elemento que discrimina si un patrón que tiene a la entrada está en un semiespacio o en el otro. Para hacerlo, una vez efectuada la función de base se realiza una discriminación con la función de activación, que decide en que semiespacio clasifica la información de entrada.

En cualquier tipo de NN se necesita algún tipo de ponderación o alguna constante que se asocia a cada conexión. Sobre estos elementos propios de cada conexión, mediante su ajuste, es precisamente sobre los cuales se va a efectuar el entrenamiento.

En el caso de que estos elementos antes citados sean de ponderación, es decir que cada señal de entrada a la neurona debe ir multiplicada por ellos, se denominan pesos y la notación que se va a usar es la siguiente:

$W_{l,i,j}$

l= nivel hacia el que va la conexión;

i= nodo destino del nivel l;

j= nodo origen del nivel l-1;

$y_{l,i}$

y= es la salida de la neurona en cuestión.

En general se puede decir que todos los parámetros en la ecuación 1.1 son variables en el tiempo.

1.4.1 La función de activación.

En el caso más simple, a la salida de la neurona se puede tener dos estados: nivel alto o nivel bajo. La función de activación que implementa este tipo de discriminación se denomina función de activación dura, y es la que se representa en la figura 1.5 y en la ecuación 1.2.

En la ecuación 1.2, T representa el umbral de decisión. De esto se saca que la operación básica que realiza la neurona es activarse, si el resultado de la función de base supera el umbral, o permanecer inactiva si no lo supera. El problema de esta función es que al presentar una discontinuidad no es derivable, lo cual, como se verá puede resultar un problema serio.

$$f_{HL}(y) = \begin{cases} 1 & y > T \\ 0 & y \leq T \end{cases}$$

Ecuación 1.2

Para evitar este inconveniente se establece otra función de activación, esta sí derivable, a la cual se denomina función de activación blanda y supone una transición gradual de nivel bajo a nivel alto. En concreto esta función es una sigmoidal y la tenemos en la ecuación 1.3 y figura 1.6.

$$f_{HL}(y) = (1 + e^{-gv})^{-1}$$

Ecuación 1.3

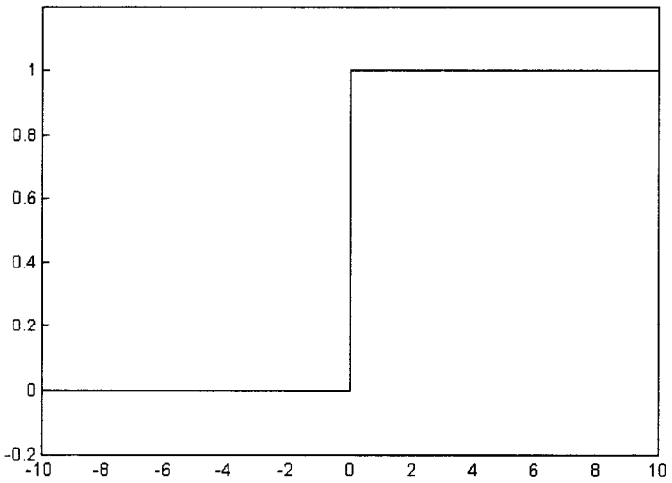


Figura 1.5

En la ecuación 1.3 ,la constante g nos mide el grado de aproximación de la sigmoide a la activación dura. En la figura 1.6 apreciamos tres ejemplos de sigmoide con g=5, 1 y 0.5.

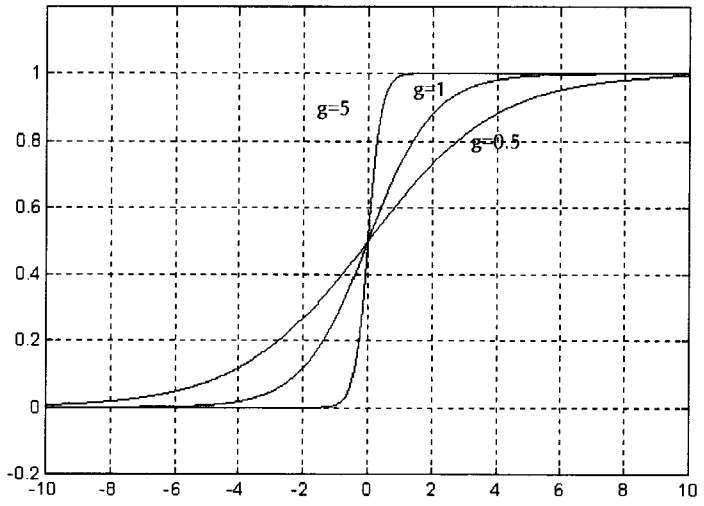


Figura 1.6

Tanto el umbral duro como la sigmoide pueden tener por asíntotas [-1,1] en vez de [0,1] como es el caso del Adaline que se verá con posterioridad.

1.4.2 La función de base.

La función de base, como se adelantó, es la operación que se realiza a la entrada del nodo o neurona con todas las señales que tienen conexión con esta.

Hay varios tipos de función de base: lineal, radial, polinómica. En este punto se va a profundizar en la función de base lineal porque es la que se utiliza en el Perceptrón Multicapa, la red que se va a utilizar en esté proyecto.

1.4.2.1 Función de base lineal.

La función de base lineal se caracteriza porque la operación que realiza es la suma de todas las señales que tiene de entrada, ponderadas por los pesos de cada conexión.

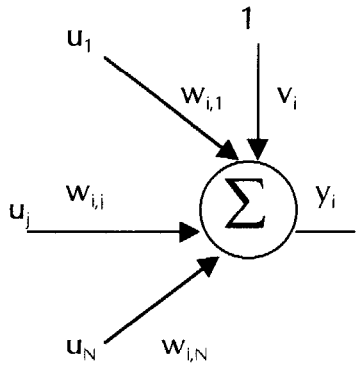


Figura 1.7

Es decir, multiplica cada u_j por el peso de la conexión respectiva, $w_{i\,j}$, y las suma todas. Además se puede decir que el offset v_i se puede considerar a efectos de programación como el peso de una conexión cuya señal de entrada es siempre 1, tal y como se indica en la figura 1.7. La ecuación de la función de base lineal es la siguiente:

$$y_i = \sum_{j=1}^{N_{l-1}} w_{i,j} \times u_j + v_i$$

Ecuación 1.4

Donde N_{l-1} es el nº de nodos del nivel anterior. Como se puede observar, este tipo de función de base ya fue utilizado en el ejemplo del apartado 1.4.

1.5 Origen: el Perceptrón de Rosenblatt.

Fue la primera Red Neuronal propuesta, y el primer antecesor del Perceptrón Multicapa.. El esquema es el que se presenta en la figura 1.8.

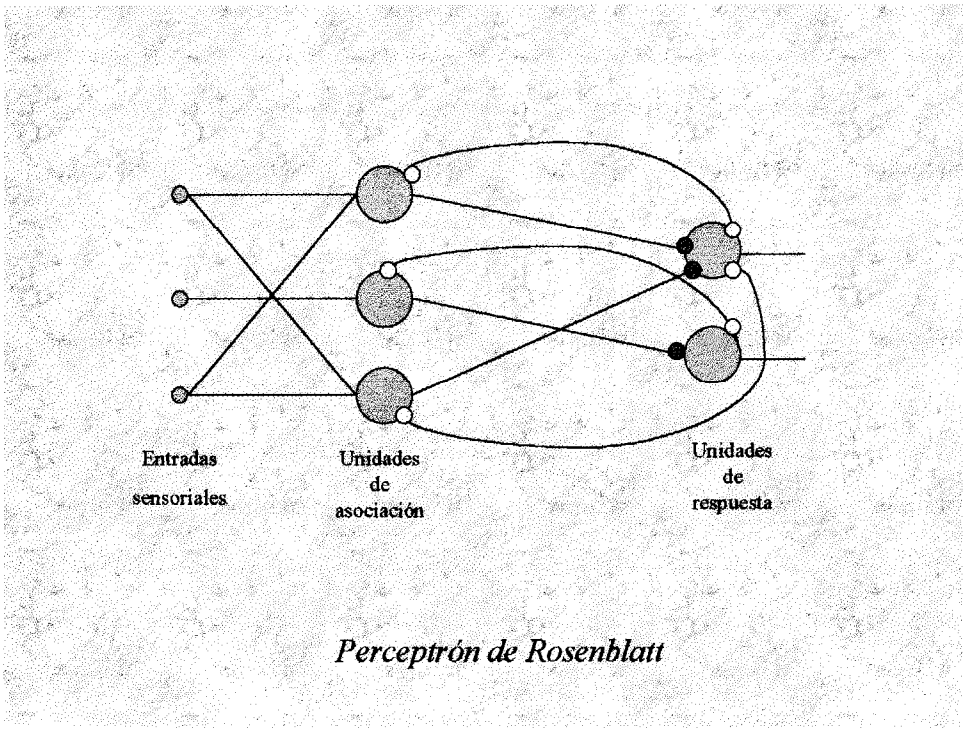


Figura 1.8

Las entradas sensoriales se distribuyen (aleatoriamente) sobre las unidades de asociación y éstas se conectan a las de respuesta, de las que se realimentan conexiones inhibitorias hacia atrás: mediante entrenamiento se pretende la clasificación de las entradas.

El éxito que logró fue bastante limitado: La red se degradaba cuando las entradas tenían una elevada complejidad.

1.6 El ADALINE (Perceptrón Monocapa).

El Adaline es el inmediato predecesor del Perceptrón Multicapa El Adaline es un dispositivo que consta de un único elemento de procesamiento o nodo. Por ello, estrictamente no es una *Red* Neuronal. Sin embargo merece su estudio dado que es una estructura muy importante en lo siguiente. Fue propuesto a comienzos de los años sesenta por Widrow.

Adaline son siglas. Inicialmente significaba ADAptive Linear Neuron (neurona lineal adaptativa). Posteriormente se introdujo una variación y las mismas siglas pasaron a significar ADAptive LINear Element (elemento lineal adaptativo).

En la figura 1.9 y la ecuación 1.6 se observa el modelo de un adaline. La función de activación es un umbral duro como el de la ecuación 1.2 y la figura 1.5 pero sus asíntotas pasan a ser [-1,1] tal y como se observa en la figura 1.10.

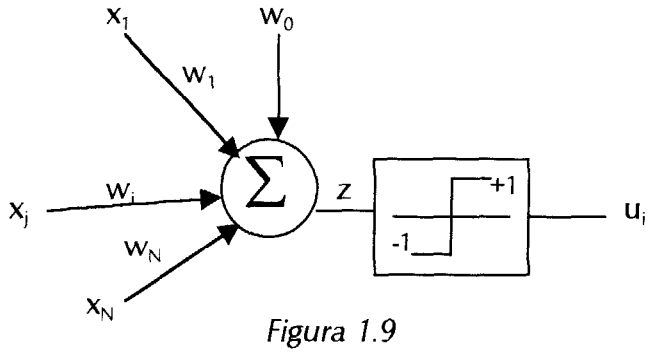


Figura 1.9

$$\sum_{i=1}^{N} w_i x_i + w_0 = 0 \quad W^T X = 0$$

Ecuación 1.8

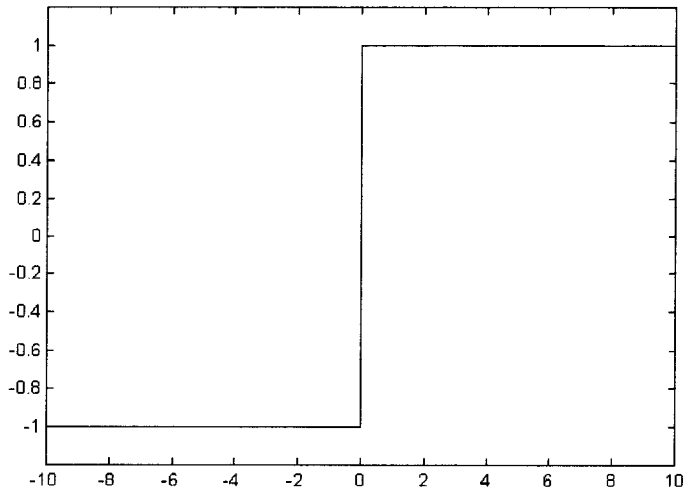


Figura 1.10

Lo que se citaba en 1.4 como offset de la neurona se denomina tendencia (w_2) cuando se habla de Adaline.

En la ecuación 1.8 se muestra el sistema de ecuaciones también en notación matricial donde W^T es el vector de pesos traspuesto y X es el vector de entrada.

Básicamente la función que realiza el adaline es subdividir el espacio de muestras en dos semiespacios.

Se denomina combinador lineal adaptativo (ALC) a la parte del Adaline que efectúa la suma de las señales de entrada. La discriminación que efectúa el Adaline se lleva a cabo de la siguiente manera. Si la salida del ALC es positiva a la salida del Adaline devuelve +1. Si por el contrario la salida del ALC es negativa a la salida del Adaline es -1.

El Adaline (o el ALC) es ADAptativo en el sentido de que existe un procedimiento bien definido para modificar los pesos con objeto de hacer posible que el dispositivo proporcione el valor de salida *correcto* para la entrada dada. La manera de *entrenar* al Adaline se muestra a continuación.

1.6.1 El algoritmo de aprendizaje LMS.

LMS significa del error cuadrático medio (Least Mean Square). Fue propuesto por Widrow y Hoff para el entrenamiento del ADALINE: minimiza la función de coste:

$$C(w) = \frac{1}{2} \sum_{k=1}^{K} [d_k - z_k]^2 \qquad z_k = \sum_{i=1}^{N} w_i \times x_i + w_0$$

Ecuación 1.9

K es el número de pares de entrenamiento del tipo $\{X_k , d_k\}$ donde a cada entrada X_k le corresponde la salida deseada d_k.

Para actualizar los pesos W del Adaline se obtiene el gradiente de la función de coste y se procede de la siguiente manera según se realice un entrenamiento en bloque:

$$w(m+1) = w(m) + \eta \sum_{k=1}^{K} [d_k - z_k(m)] \times x_k$$

Ecuación 1.10

O si se realiza un entrenamiento paso a paso:

$$w(k+1) = w(k) + \eta [d_k - z_k] \times x_k$$

Ecuación 1.11

Que suele ser más rápido pero más ruidoso, esto es con más rizado en la curva de error hacia la convergencia.

En la práctica el algorítmo LMS resulta ser muy robusto que alcanza la convergencia en muchos casos. A diferencia de la regla del Perceptrón, el empleo del LMS produce resultados razonables aunque no haya separabilidad lineal.

1.6.2 Limitaciones del perceptrón monocapa.

Los investigadores Minsky y Papert reseñaron que la principal traba de este es que sólo es un discriminador lineal por lo que no puede resolver problemas tan sencillos como por ejemplo la implementación de una OR exclusiva, donde (1,0) y (0,1) constituyen una clase y (0,0) y (1,1) otra.

Por otra parte el uso de cascadas de capas de Adalines permítiría más capacidades, pero la presencia de umbrales duros impide el entrenamiento basándonos en técnicas de gradiente.

Además los autores pronosticaron que aunque se salvara esta última dificultad, el Adaline sólo resolvería pequeños problemas presentando poca capacidad de escalabilidad.

1.7 El Perceptrón Multicapa (MLP)

Widrow planteó dos posibles soluciones a los problemas del Adaline:

MADALINE I: estructura construida mediante la combinación lógica de salidas de ADALINES. MADALINE viene de Múltiples ADALINE'S.

MADALINE II: redes compuestas de la disposición en cascada de ADALINE'S.

Para los Madalne II, que ya se pueden llamar **Perceptrones Multicapa** (MLP) es bastante ilustrativa la discusión de Lippmann:

- Una red monocapa permite definir semiespacios, según ilustra la figura 1.11

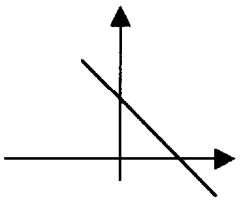


Figura 1.11

- Una red bicapa, regiones tales como las de la figura 1.12.

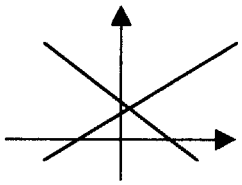


Figura 1.12

- Una red tricapa, figura 1.13.

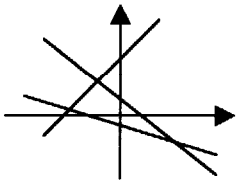


Figura 1.13

Y así, quedando claro el potencial de estas redes para clasificación e incluso para correspondencia.

CAPÍTULO 2: EL ALGORITMO DE BACKPROPAGATION (BP) PARA EL MLP.

Este algoritmo lo publicó Rumelhart en 1986. Desde entonces se han realizado muchas variaciones algunas de las cuales pueden encontrarse en [1], [2] y [6]. Aquí se expone el algoritmo básico. En la figura 2.1 es observa un MLP de dos capas ocultas que nos va a servir de ejemplo para la deducción del algoritmo.

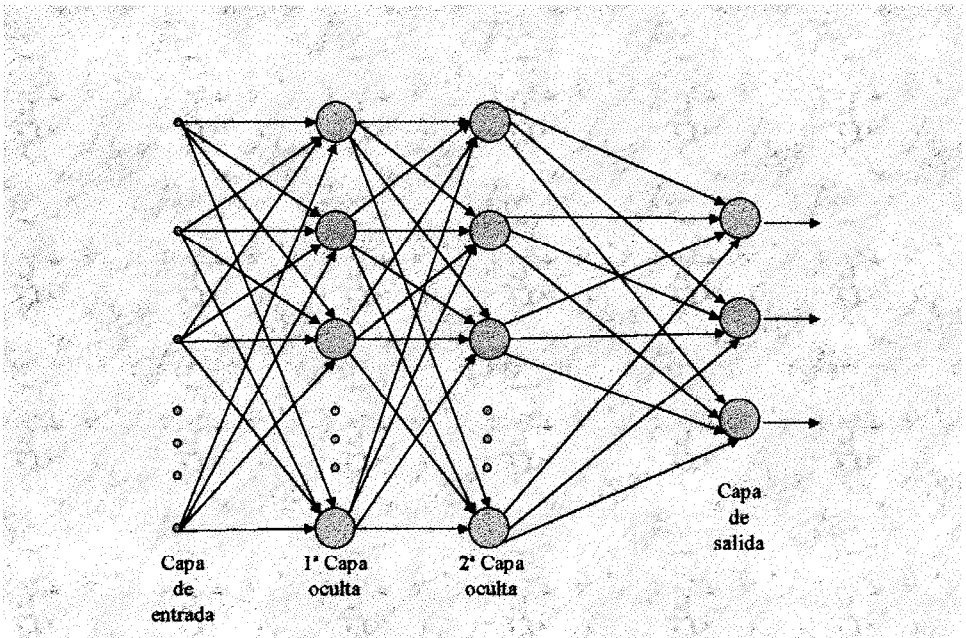


Figura 2.1

Se parte de esta red en la que en cada conexión hay un peso $w_{l,i,j}$ tal y como se expuso en el apartado 1.4. La función de activación es una sigmoide (Ecuación 1.3) y la función de base es lineal. Es decir que sigue un funcionamiento idéntico al del ejemplo planteado en la ecuación 1.1 en el capítulo primero.

Se dispone de un conjunto de K pares de entrenamiento que se denota $\{X_k, d_k\}$, donde X es la entrada y d la salida deseada.

Se define la señal de error para el nodo j de la capa de salida y el par de entrenamiento k de la siguiente manera:

$$e_j(k) = d_j(k) - y_j(k)$$

Ecuación 2.1

Se define el valor del error cuadrático de la neurona j como $\frac{1}{2}e_j^2(k)$.

Consecuentemente se define el valor instantáneo ξ(k) de la suma de los errores cuadráticos como la suma de $\frac{1}{2}e_j^2(k)$ en todas las neuronas en la capa de salida.

La suma instantánea del error cuadrático de la red puede ser escrita entonces así:

$$\xi(k) = \frac{1}{2}\sum_{j \in N_L} e_j^2(k)$$

Ecuación 2.2

donde el conjunto N_L incluye a todos las neuronas en la capa de salida de la red. El error cuadrático medio se obtiene sumando ξ(k) para todos los pares de entrenamiento y normalizando, dividiendo por K tal y como se muestra en la ecuación 2.3.

$$E = \frac{1}{K}\sum_{k=1}^{K} \xi(k)$$

Ecuación 2.3

El algoritmo BP para el entrenamiento del MLP

La suma instantánea de los errores cuadráticos $\xi(k)$, y el error cuadrático medio E, son función de todos los parámetros libres de la red, por ejemplo los pesos y offsets. Para un conjunto de pares de entrenamiento, E representa la función de coste como medida del proceso de aprendizaje. El objetivo del proceso de aprendizaje es ajustar los parámetros libres de la red de forma que se minimice E. Para llevar a cabo la minimización se realiza un procedimiento similar al del apartado 1.6.1 pero en este caso la función de activación ya no es un umbral duro, lo que permite el uso de gradientes. Los ajustes de los pesos se realizan de acuerdo con los respectivos errores calculados para cada par de entrenamiento introducido a la red. El promedio de esos cambios individuales de los pesos sobre todos los pares de entrenamiento es una estimación del cambio que deben sufrir los pesos para minimizar la función de coste E sobre el conjunto de entrenamiento entero.

Se parte de un nodo ejemplo como el de la figura 1.4 y ecuación 1.1 deñ capítulo I. La corrección de los pesos sinápticos se realiza mediante un incremento en estos como se indica en la ecuación 2.4.

$$w'_{l,i,j} = w_{l,i,j} + \Delta w_{l,i,j}$$

Ecuación 2.4

El algoritmo de Backpropagation aplica una corrección $\Delta w_{l,i,j}(k)$ a los pesos sinápticos $w_{l,i,j}(k)$, la cual es proporcional al gradiente $\partial\xi(k) / \partial w_{l,i,j}(k)$. De acuerdo a la regla de la cadena se puede expresar el gradiente de la siguiente forma:

$$\frac{\partial \xi(k)}{\partial w_{l,i,j}(k)} = \frac{\partial \xi(k)}{\partial e_j(k)} \frac{\partial e_j(k)}{\partial y_j(k)} \frac{\partial y_j(k)}{\partial v_j(k)} \frac{\partial v_j(k)}{\partial w_{l,i,j}(k)}$$

Ecuación 2.5

El gradiente $\partial\xi(k)/\partial w_{l,i,j}(k)$ representa la dirección de búsqueda en el espacio de los pesos para el peso de la conexión sináptica $w_{l,i,j}$.

Derivando a ambos lados de la ecuación 2.2 se tiene:

$$\frac{\partial \xi(k)}{\partial e_j(k)} = e_j(k)$$

Ecuación 2.6

Derivando a ambos lados de la ecuación 2.12 se tiene:

$$\frac{\partial e_j(k)}{\partial y_j(k)} = -1$$

Ecuación 2.7

Ahora derivando la ecuación 1.1 b respecto $v_j(k)$ se tiene:

$$\frac{\partial y_j(k)}{\partial v_j(k)} = f_j'(v_j(k))$$

Ecuación 2.8

donde la prima significa derivada respecto de su argumento. Finalmente derivando 1.1 a con respecto a $w_{i,j}$ se tiene:

$$\frac{\partial v_j(k)}{\partial w_{i,j}(k)} = y_i(k)$$

Ecuación 2.9

Sustituyendo en la ecuación 2.5 las ecuaciones 2.8 a la 2.9 se tiene:

$$\frac{\partial \xi(k)}{\partial w_{i,j}(k)} = -e_j(k) f_j'.(v_j(k)) y_i(k)$$

Ecuación 2.10

La corrección $\Delta w_{i,j}(k)$ aplicada a $w_{i,j}(k)$ es definida por la regla delta:

$$\Delta w_{i,j}(k) = -\eta \frac{\partial \xi(k)}{\partial w_{i,j}(k)}$$

Ecuación 2.11

donde η es una constante que determina la velocidad de aprendizaje; se denomina parámetro de velocidad de aprendizaje del algoritmo de backpropagation. De acuerdo a las ecuaciones 2.11 y 2.12 se tiene:

$$\Delta w_{i,j}(k) = \eta \delta_j(k) y_i(k)$$

Ecuación 2.12

donde el gradiente local $\delta_j(k)$ es definido por:

$$\delta_j(k) = -\frac{\partial \xi(k)}{\partial e_j(k)} \frac{\partial e_j(k)}{\partial y_j(k)} \frac{\partial y_j(k)}{\partial v_j(k)}$$
$$= e_j(k) f_j'.(v_j(k))$$

Ecuación 2.13

El gradiente indica los cambios requeridos en los pesos sinápticos.

De la ecuación 2.13 se deduce que un elemento clave envuelto en el cálculo del ajuste en los pesos $\Delta w_{i,j}(k)$ es la señal de error $e_j(k)$ en la neurona de salida j. En este contexto se pueden identificar dos casos distintos dependiendo de en que parte de la red se encuentre l a neurona j. Está el caso I en el que la neurona j está en la capa de salida. Por otra parte está el caso II en el que la neurona está en cualquier otra capa que no sea la capa de salida.

El caso I es sencillo de manipular porque para calcular la señal de error en cada nodo simplemente hay que restar de la salida deseada la señal que se tiene naturalmente, tal y como indica la ecuación 2.12.

El caso II no es tan inmediato porque como las neuronas no son directamente accesibles, comparten la *culpabilidad* en el error que se produzca en los nodos de salida. La cuestión es como penalizar a esos nodos que comparten la responsabilidad. Esta cuestión se resuelve mediante una propagación hacia atrás (back-propagation) de la señal de error de los nodos de salida.

Caso I: La neurona j es un nodo de salida.

Cuando la neurona j está ubicada en la capa de salida de la red, debe ser contrastada la salida natural con la respuesta deseada en sí misma. De ahí que se use la ecuación 2.12 para calcular la señal de error $e_j(k)$ asociada a la neurona. Con $e_j(k)$ es un problema trivial el cómputo del gradiente local $\delta_j(k)$ usando la ecuación 2.13.

Caso II: La neurona j es un nodo oculto.

Cuando la neurona j está ubicada en una capa oculta de la red no hay una respuesta deseada especificada para esta neurona. Para calcular la señal de error de esta neurona se determina recursivamente en términos de las señales de error de todas las neuronas a las que está conectada la neurona en cuestión. Aquí es donde el algoritmo de Backpropagation se complica. De acuerdo con la ecuación 2.13 se puede redefinir el gradiente local $\delta_j(k)$ para la neurona oculta j como:

$$\delta_j(\mathrm{k}) = -\frac{\partial \xi(k)}{\partial \mathrm{y}_j(k)} \frac{\partial \mathrm{y}_j}{\partial \mathrm{v}_j} =$$

$$= -\frac{\partial \xi(k)}{\partial y_j(k)} f_j'.(v_j(k))$$

Ecuación 2.14

donde en la segunda línea se ha hecho uso de la ecuación 2.8. Para calcular la derivada parcial $\partial\xi(k)/\partial y_j(k)$, se procede como sigue.

Se toma la ecuación 2.2 y se reescribe cambiando el índice j por m. Esto se hace para evitar confusión con el índice j que se usa ahora para la neurona oculta.

$$\xi(k) = \frac{1}{2} \sum_{m \in N_L} e_m^2(k)$$

Ecuación 2.15

Si se deriva esta ecuación respecto la señal $y_j(k)$ se tiene:

$$\frac{\partial \xi(k)}{\partial y_j(k)} = \sum_m e_m \frac{\partial e_m(k)}{\partial y_j(k)}$$

Ecuación 2.16

Ahora se puede utilizar la regla de la cadena para el cálculo de la derivada parcial $\partial e_m(k)/\partial y_j(k)$, y reescribir la ecuación 2.16 en la forma equivalente:

$$\frac{\partial \xi(k)}{\partial y_j(k)} = \sum_m e_m(k) \frac{\partial e_m(k)}{\partial v_m(k)} \frac{\partial v_m(k)}{\partial y_j(k)}$$

Ecuación 2.17

Además, de la figura 2.2 se observa que:

$$e_m(k) = d_m(k) - y_m(k) =$$
$$= d_m(k) - f_m(v_m(k))$$

Ecuación 2.18

siendo la neurona m una neurona de la capa de salida. De esto se sigue:

$$\frac{\partial e_m(k)}{\partial v_m(k)} = -f_m'(v_m(k))$$

Ecuación 2.19

El nivel de actividad interna de la red para la neurona k es:

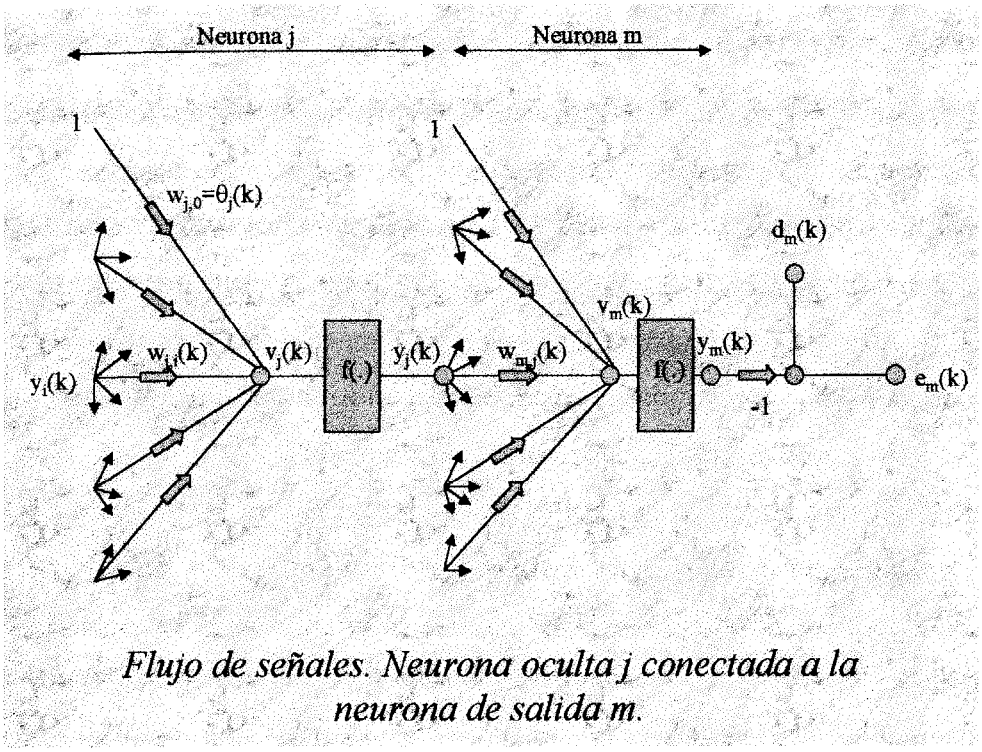
$$v_m(k) = \sum_{j=0}^{q} w_{m,j}(k) y_j(k)$$

Ecuación 2.20

donde q es el número total de entradas (sin contar el offset) aplicadas a la neurona k. Aquí de nuevo, el peso sináptico $w_{m,0}(k)$ es igual al offset $\theta_m(k)$ aplicado a la neurona k y la correspondiente entrada y_0 se fija al valor 1. Para cualquier caso, derivando la ecuación 2.20 con respecto a $y_j(k)$ conduce a:

$$\frac{\partial v_m(k)}{\partial y_j(k)} = w_{m,j}(k)$$

Ecuación 2.21



Flujo de señales. Neurona oculta j conectada a la neurona de salida m.

Figura 2.2

De esto, usando las ecuaciones 2.19, 2.21 y 2.17 se tiene que:

$$\frac{\partial \xi(k)}{\partial y_j(k)} = -\sum_m e_m(k) f_m'(v_m(k)) \times w_{m,j}(k) =$$
$$= -\sum_m \delta_m(k) \times w_{m,j}(k)$$

Ecuación 2.22

donde en la segunda línea, se ha usado la definición del gradiente local $\delta_m(k)$ dada en la ecuación 2.13 con el índice m sustituido por j.

Finalmente, usando la ecuación 2.22 en 2.14, se obtiene el gradiente local $\delta_m(k)$ para una neurona oculta j, tras la reorganización de términos como sigue:

$$\delta_j(k) = f_j'(v_j(k)) \sum_m \delta_m(k) w_{m,j}(k)$$

Ecuación 2.23

siendo la neurona j oculta. El factor $f_j'(v_j(k))$ envuelto en la computación del gradiente local $\delta_j(k)$ en la ecuación 2.23 depende solamente de la función de activación asociada a la neurona oculta j.

El otro factor, el sumatorio depende de dos conjuntos de términos. El primer conjunto de términos, $\delta_m(k)$, requiere del conocimiento de todas las señales de error $e_m(k)$, de todas las neuronas de la capa siguiente a la neurona oculta j, que están directamente conectadas a esta. El segundo conjunto términos, $w_{m,j}(k)$, consiste en los pesos sinápticos asociados a las conexiones.

Resumiendo ahora las relaciones que se han deducido para el algoritmo de backpropagation. Se tiene por una parte el factor de corrección $\Delta w_{j,i}(k)$ aplicado a los pesos sinápticos de las conexiones entre la neurona i con la j, definido por la regla delta:

$$\begin{pmatrix} Corrección \\ pesos \\ \Delta w_{j,i}(k) \end{pmatrix} = \begin{pmatrix} Parámetro \\ velocidad \\ aprendizaje \\ \eta \end{pmatrix} \cdot \begin{pmatrix} Gradiente \\ local \\ \delta_j(k) \end{pmatrix} \cdot \begin{pmatrix} Señal \\ entrada \\ neurona \quad j \\ y_i(k) \end{pmatrix}$$

Ecuación 2.24

Por otra parte el gradiente local depende de si la neurona j está en la capa de salida o en una capa oculta:

1. Si la neurona j es una neurona de salida entonces $\delta_j(k)$ es simplemente igual al producto de la derivada $f_j'(v_j(k))$ por la señal de error $e_j(k)$, ambos asociados a la neurona j (Ecuación 2.13).
2. Si la neurona j es un nodo oculto, $\delta_j(k)$ es igual al producto de la derivada $f_j'(v_j(k))$ por la suma ponderada de las δ's computadas en las neuronas de la capa siguiente a las que está conectada la neurona j (Ecuación 2.23).

2.1 Implementación del algoritmo.

Antes de presentar una implementación genérica del algoritmo reseñar que como la función de activación que se esta utilizando es una sigmoide (Ecuación 1.3 y Figura 1.6 del capítulo primero), por lo que la derivada que aparece en los cálculos resulta:

$$f'(x) = g \cdot f(x) \cdot (1 - f(x))$$

Ecuación 2.25

Un ejemplo de la implementación de este algoritmo puede ser la siguiente:

Procedimiento BACKPROPAGATION;
  Inicializar los pesos aleatoriamente;
  repetir
    Poner entrada de entrenamiento;
    FEED_FORWARD;
    COMPUTO_GRADIENTE;
    ACTUALIZA_PESOS;
  hasta encontrar condición de final;
fin {BACKPROPAGATION}

Procedimiento FEED_FORWARD;
  for capa=1 to L do
    for nodo=1 to N_{capa} do $y_{capa,nodo} = f(\sum_{i=1}^{N_{capa-1}} w_{capa,nodo,i} \times y_{capa-1,i})$;
    end
  end
end; {FEED_FORWARD}

Procedimiento COMPUTO_GRADIENTE;
  for capa=L to 1 do
    for nodo=1 to N_{capa} do
      if (capa=L) then $e_{L,nodo} = y_{L,nodo} - d_{nodo}$;
      else

$$e_{capa,nodo} = \sum_{m=1}^{N_{capa+1}} e_{capa+1,m} y_{capa+1,m}(1 - y_{capa+1,m}) \times w_{capa+1,m,nodo};$$

    end;
    for todos los pesos en todas las capas do
      $g_{capa,j,i} = e_{capa,j} y_{capa,j}(1 - y_{capa,j}) y_{capa-1,i}$;
    end;
end; {COMPUTO_GRADIENTE}

Procedimiento ACTUALIZA_PESOS;
  for todos los pesos $w_{l,j,i}$ do $w_{l,j,i}(k+1) = w_{l,j,i}(k) - \eta \cdot g_{l,j,i}$;
  end;
end; {ACTUALIZA_PESOS}

Algoritmo de aprendizaje Backpropagation

En este algoritmo conviene aclarar:

- L: nº de capas en la red neuronal.
- N_{capa}: nº de nodos en el nivel *capa*.
- $g_{capa,j,i}$: gradiente asociado a la conexión entre la neurona i del nivel *capa-1* y la neurona j del nivel *capa*.
- $W_{capa,j,i}$: peso asociado a la conexión entre la neurona i del nivel *capa-1* y la neurona j del nivel *capa*.

Condiciones de fin pueden ser:

- Que el error cuadrático medio sea inferior a un máximo establecido.
- Que el número de iteraciones sea superior a un máximo establecido.
- Que el error relativo, entendiendo por error relativo la variación relativa del error cuadrático medio actual respecto al anterior.

$$er(I) = \frac{E(I-1) - E(I)}{E(I-1)}$$

Ecuación 2.26

2.2 Aceleramiento del algoritmo mediante η variable.

En la figura 2.3 se puede observar la disminución del error cuadrático medio durante el entrenamiento de una red neuronal. En las figuras se representa iteraciones frente a error cuadrático.

Entrenamiento con η fija.

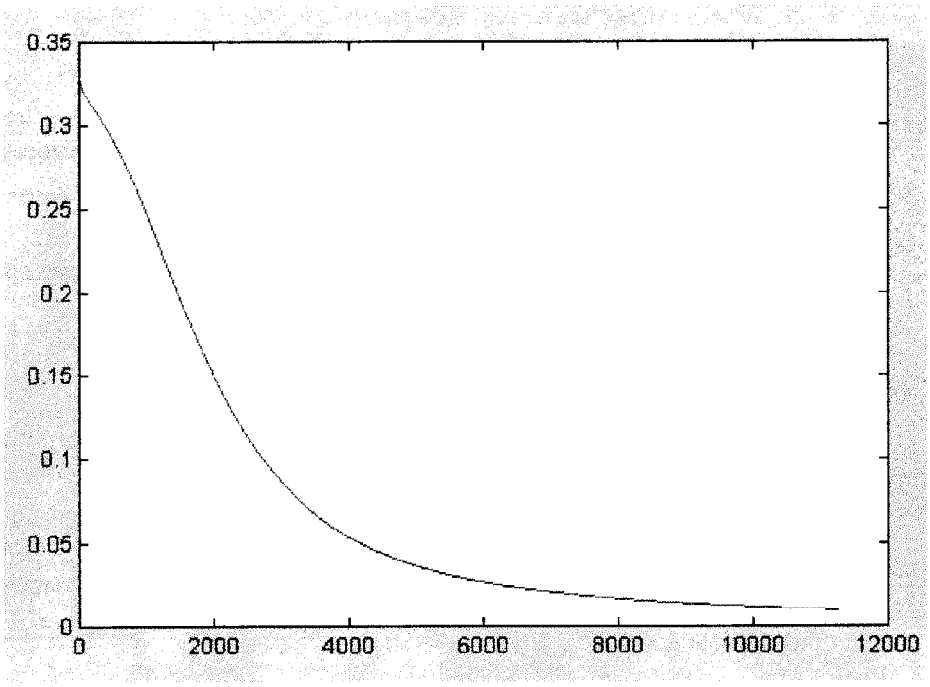


Figura 2.3

Entrenamiento con η variable

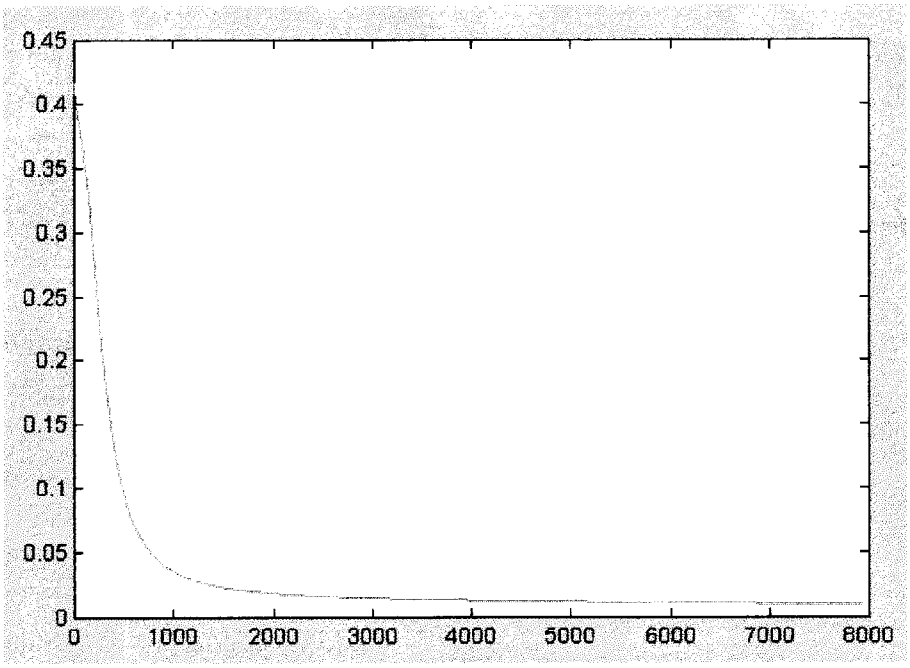


Figura 2.4

En la figura 2.4 se muestra la curva de error de un entrenamiento acelerado. Los autores afirman que es posible disminuir el tiempo de entrenamiento reduciendo el valor de la constante η gradualmente, al tiempo que aumenta el nº de iteraciones. Basado en esto propongo que el valor de la constante siga una ley como la que se indica en la ecuación 2.27, donde

- η_i: valor inicial de la constante.
- η_f: valor final de la constante.
- f: factor que calibra la velocidad a la que se tiende al valor final.

$$\eta(i) = (\eta_i - \eta_f) \times \exp(-f \cdot i) + \eta_f$$

Ecuación 2.27

En la figura 2.5 se muestra como es la ley que he propuesto. Las absisas representan iteraciones.

Ejemplos de ley η variable

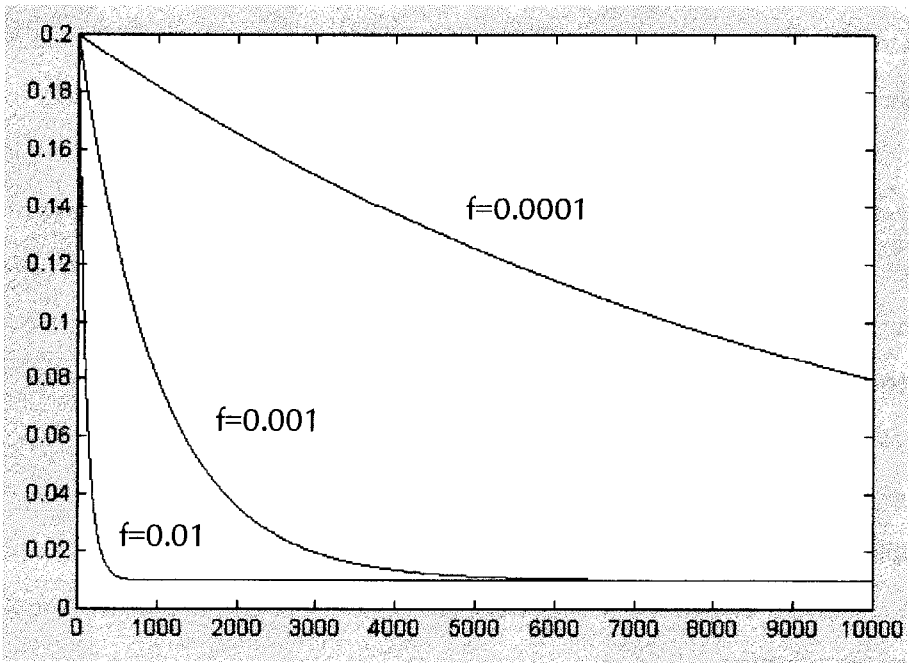


Figura 2.5

CAPÍTULO 3: ALGORITMO DE LINEALIZACIÓN OLL.

Desde su aparición, el algoritmo de aprendizaje Back Propagation (BP) ha sido ampliamente utilizado para el entrenamiento de los perceptrones multicapa (MLP) en aplicaciones tales como reconocimiento de patrones, control de sistemas dinámicos, predicción de series caóticas, y un largo etcétera. Sin embargo desde siempre este algoritmo se encuentra con dos problemas constantes: Es muy lento y necesita que se le introduzcan parámetros iniciales.

Se han intentado salvar estos problemas de muchas maneras. Algunos algoritmos mejorados emplean reglas heurísticas para encontrar los parámetros de aprendizaje óptimos. Otros trabajan en la mejora del método de gradiente descendente para acelerar la convergencia. Y más métodos de optimización no lineal como el método del gradiente conjugado, el método de Newton o las técnicas cuasi-newtonianas.

Para muchas aplicaciones estas mejoras son suficientes. El problema surge con las aplicaciones que requieren una gran precisión a la salida, como la predicción de series temporales caóticas, en donde los algoritmos conocidos aún son ineficientes y lentos.

Para intentar reducir las mencionadas dificultades S. Ergezinger y E. Thomsen propusieron un nuevo algoritmo que no se basa en la evaluación de los gradientes locales [5]. El nuevo algoritmo que proponen se basa en una optimización del MLP capa a capa. De ahí su nombre: OLL, Optimization Layer by

Layer. Ellos proponen que con la optimización de los pesos de cada capa por separado, el proceso total de aprendizaje puede realizarse mucho más eficientemente. El aprendizaje OLL reduce el problema de optimización de cada capa a un problema lineal que puede resolverse con exactitud. Al reducir a un problema lineal, se produce un inevitable error de linealización que será contrarrestado mediante la adición a la función de coste de lo que denominan un término de penalización.

En la red ejemplo que vamos a analizar vamos a asumir M nodos de entrada, H nodos de la capa oculta y un nodo de salida.

Las funciones de activación para la capa oculta son sigmoidales como la de la ecuación 1.3 en el capítulo I.

Y las de la capa de salida son funciones lineales:

$$f(s) = s$$

Ecuación 3.1

Además para tener en cuenta y entrenar los offsets de la capa oculta a la entrada x se le añade una señal más x_0 cuyo valor será siempre 1 y cuyo peso de la conexión a los nodos de la capa oculta será la bias de esta capa. Igualmente a las entradas de la capa de salida se le anade la señal O_0 de valor siempre 1 y cuyo peso será la bias de la capa de salida. El resultado es la red ejemplo de la figura 3.1.

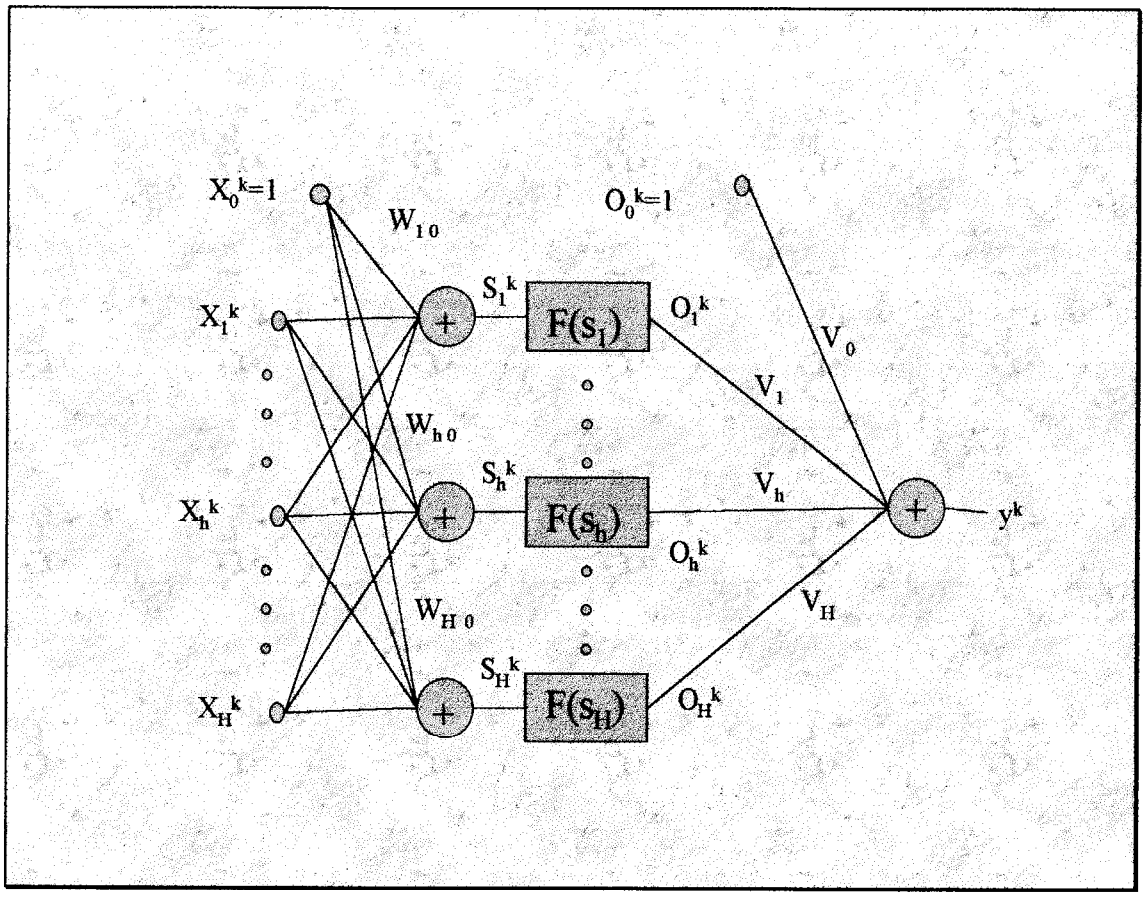


figura 3.1

Así pues partimos de una matriz de pesos W(i,j) de dimensión (M+1)*H , los cuales conectan la capa de entrada con la capa oculta; y un vector de pesos V(i) de longitud (H+1) que conectan la capa oculta con la capa de salida. La ecuación general será:

$$y^k = \sum_{h=0}^{H} v_h o_h^k = v_0 + \sum_{h=1}^{H} v_h f(s_h^k) =$$
$$= v_0 + \sum_{h=1}^{H} v_h f\left(\sum_{m=0}^{M} w_{hm} x_m^k \right)$$

Ecuación 3.2

3.1 Aprendizaje.

Para efectuar el entrenamiento de esta red necesitamos un conjunto de K pares (entrada, salida), que representaremos de la forma $\{x^k, d^k\}$, siendo k el índice que nos índica el par al que nos referimos.

La función de coste a minimizar es el error cuadrático medio:

$$E(W,v)=\frac{1}{K}\sum_{k=1}^{K}E^k \quad \text{con} \quad E^k(W,v)=\frac{1}{2}(d^k-y^k)^2$$

Ecuación 3.3

En el caso del algoritmo de Back Propagation los pesos de cada capa se iban variando cada uno independientemente del resto de acuerdo a su derivada parcial. El error se propaga hacia atrás desde la capa de salida a la capa de entrada. Además todos los pesos de todas las capas se modifican a la vez. Esta actualización tiene lugar tras cada par de entrenamiento en la versión dinámica, o tras un bloque de pares en el entrenamiento estático.

El nuevo algoritmo propuesto tiene tres características a destacar:

- Los pesos de cada capa son variados dependiendo todos de todos pero cada capa por separado.
- La optimización de la capa de salida es un problema lineal.
- La optimización de la capa oculta se reduce a un problema lineal también previa linealización de las funciones de activación sigmoidales.

Como se ha dicho la optimización de los pesos consta de dos partes:
Capa de salida y Capa oculta y a continuación se analizan ambos pasos por separado.

3.1.1 Optimización de la capa de salida.

Partimos de considerar constantes las salidas de la capa oculta $\{O_0^k, O_1^k, \ldots, O_H^k\}$. Dado que la función de activación de la capa de salida es lineal se puede considerar la capa de salida como una red ADALINE tal como se indicaba en la figura 1.4 y que se vuelve a traer por comodidad a la figura 3.2.

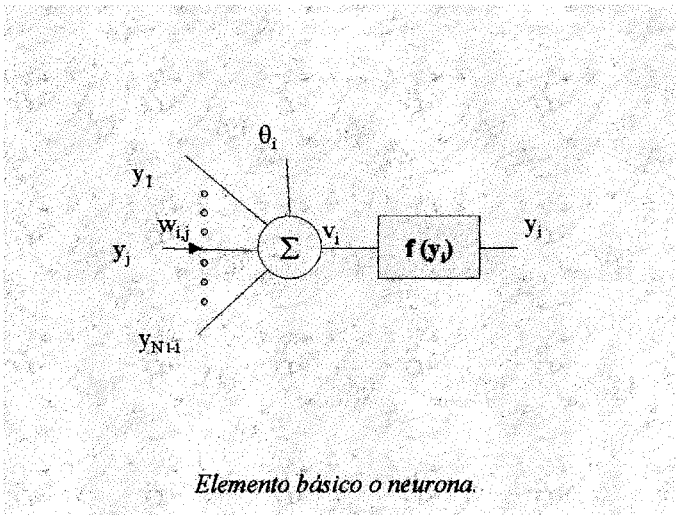


Figura 3.2

La salida de la capa es simplemente:

$$y^k = \sum_{h=0}^{H} v_h o_h^k = v^T o^k$$

Ecuación 3.4

"v" es un vector de H+1 elementos. "v^T" es la traspuesta de este vector, es decir una columna. "o" es una columna de la matriz "O" de H+1 filas por K columnas.

Por lo tanto para obtener y^k es suficiente con multiplicar matricialmente como se indica en la ecuación 3.4.

De las ecuaciones 3.3 y 3.4 se busca los pesos v^{opt} que minimicen el error cuadrático medio.

$$E(v^{opt}) = \min_{v} E(v)$$

siendo E(v)

$$E(v) = \frac{1}{K} \sum_{k=1}^{K} \frac{1}{2} (d^k - v^T o^k)^2$$

Ecuación 3.5

Para obtener los pesos óptimos calculamos el gradiente de E(v) con respecto a v y lo igualamos a 0:

$$\frac{\partial}{\partial v} E(v) = \frac{1}{K} \sum_{k=1}^{K} (v^T o^k - d^k) o^k = 0$$

Ecuación 3.6

De aquí deducimos un conjunto de H+1 ecuaciones lineales

$$Av = b$$

Ecuación 3.7

Las cuales han de ser resueltas para encontrar el vector de pesos óptimos

$$v^{opt} = A^{-1} b$$

Ecuación 3.8

Donde la matriz A y el vector b las obtenemos así:

$$A = \{a_{i,j}\}, \quad a_{i,j} = \sum_{k=1}^{K} o_i^k o_j^k; \quad i, j = 0, \ldots, H;$$

Ecuación 3.9

$$b = \{b_i\}; \quad b_i = \sum_{k=1}^{K} d^k o_i^k; \quad i = 0, \ldots, H;$$

Ecuación 3.10

Una forma más directa para el cálculo de A y b es multiplicando matricialmente como se indica en las ecuaciones 3.11 y 3.12.

$$A = O \times O^T$$

Ecuación 3.11

$$b = O \times d^T$$

Ecuación 3.12

Aquí se está considerando d como un vector de k elementos de salidas deseadas.

Hay que tener en cuenta que estos v^{opt} son los pesos óptimos para unos pesos W concretos de la capa oculta.

Es importante reseñar que la matriz A cumple ser simétrica, definida positiva. Esta cualidad puede aprovecharse para agilizar el cálculo porque a la hora de resolver el sistema de ecuaciones lineales de la ecuación 3.7, se puede resolver directamente mediante el método de Cholesky, sin necesidad de calcular la inversa en la ecuación 3.8. Este método se extrajo de [3] y [4] y se expone en el Anexo I.

3.1.2 Linealización de la capa oculta.

Como se ha visto, en la capa de salida solo ha sido necesario resolver un conjunto de ecuaciones lineales para obtener los pesos óptimos de las conexiones entre la capa oculta y la capa de salida.

Si fuera posible linealizar la capa oculta, sería muy fácil obtener los pesos resolviendo también un sistema de ecuaciones, al menos mientras el error de linealización introducido fuera pequeño.

La optimización no puede llevarse a cabo en un solo paso por capa. Será un algoritmo iterativo que irá optimizando los pesos de las dos capa con el fin de minimizar la función de coste (Ecuación 3.3).

Cuando se realice la optimización de los pesos de las conexiones entre la capa de entrada y la oculta, dando lugar al incremento $\Delta w_{h,m}$, los pesos de aquellas se actualizarán de la siguiente forma:

Algoritmo de linealización OLL

$$w_{new,hm}=w_{old,hm}+\Delta w_{h.m}$$

$$m=0,...,M; \quad h=1,...,H$$

Ecuación 3.13

De ahí que las nuevas entradas de los nodos ocultos para el patrón de entrenamiento k-ésimo vienen dados por:

$$s^k_{new,h} = \sum_{m=0}^{M} w_{new,hm}x^k_m =$$

$$= \sum_{m=0}^{M} w_{old,hm}x^k_m + \sum_{m=0}^{M} \Delta w_{hm}x^k_m =$$

$$= s^k_{old,h} + \Delta s^k_h.$$

Ecuación 3.14

Vemos como cambios en los pesos en las conexiones entre la capa de entrada y la oculta dan lugar a cambios Δs^k_h de la entrada a las funciones de activación sigmoidal.

Para calcular los efectos de Δw_{hm} en la nueva salida de la red y^k_{new}, primero las nuevas salidas $o^k_{new,h}$ de las funciones de activación no lineales f(s).

La expansión de la serie de Taylor de f(s) con respecto al punto $s^k_{new,h}$ viene dada por:

$$f(s^k_{new,h}) = f(s^k_{old,h} + \Delta s^k_h) =$$

$$= f(s^k_{old,h}) + \frac{\partial . f(s^k_{old,h})}{\partial . s^k_{old,h}} \Delta s^k_h + \frac{\partial^2 f(s^k_{old,h})}{2\partial^2 s^k_{old,h}} (\Delta s^k_h)^2 + R$$

Ecuación 3.15

Donde R representa los términos de orden mayor. De ahí que para pequeños incrementos Δs^k_h la salida de la función de activación puede ser aproximada por la serie de Taylor solo con el primer término:

$$O^k_{new,h} = \begin{cases} f(s^k_{old,h}) + \dfrac{\partial . f(s^k_{old,h})}{\partial . s^k_{old,h}} \Delta s^k_h & h = 1, \cdots, H \\ O^k_{old,h} = 1 & h = 0. \end{cases}$$

Ecuación 3.16

Usando esta aproximación obtenemos la nueva salida de la siguiente forma:

$$y^k_{new} = \sum_{h=0}^{H} v_h o^k_{new,h} \approx$$

$$\approx \sum_{h=0}^{H} v_h o^k_{old,h} + \sum_{h=1}^{H} f.'(s^k_{old}) v_h \Delta s^k_h =$$

$$= y^k_{old} + \Delta y^k.$$

Ecuación 3.17

Para la optimización de la matriz de pesos de la capa oculta solo han de ser considerados los cambios Δs^k_h y Δy^k causados por ΔW.

$$\Delta s^k_h = \sum_{m=0}^{M} \Delta w_{hm} x^k_m$$
$$h = 1, \ldots, H \quad k = 1, \ldots, K$$

Ecuación 3.18

Mediante la definición de $v^k_{lin,h}$:

$$v^k_{lin,h} = f.'(s^k_{old,h}) \cdot v_h$$
$$h = 1, \ldots, H \quad k = 1, \ldots, K$$

Ecuación 3.19

Los cambios en la salida de la red pueden escribirse:

$$\Delta y^k = \sum_{h=1}^{H} v^k_{lin,h} \Delta s^k_h \quad k = 1, \ldots, K.$$

Ecuación 3.20

La red linealizada resultante viene descrita en la figura 3.3.

Nótese que los pesos $v^k_{lin,h}$ en la capa de salida dependen ahora de los patrones de entrenamiento $\{x^k,d^k\}$ que están siendo aplicados.

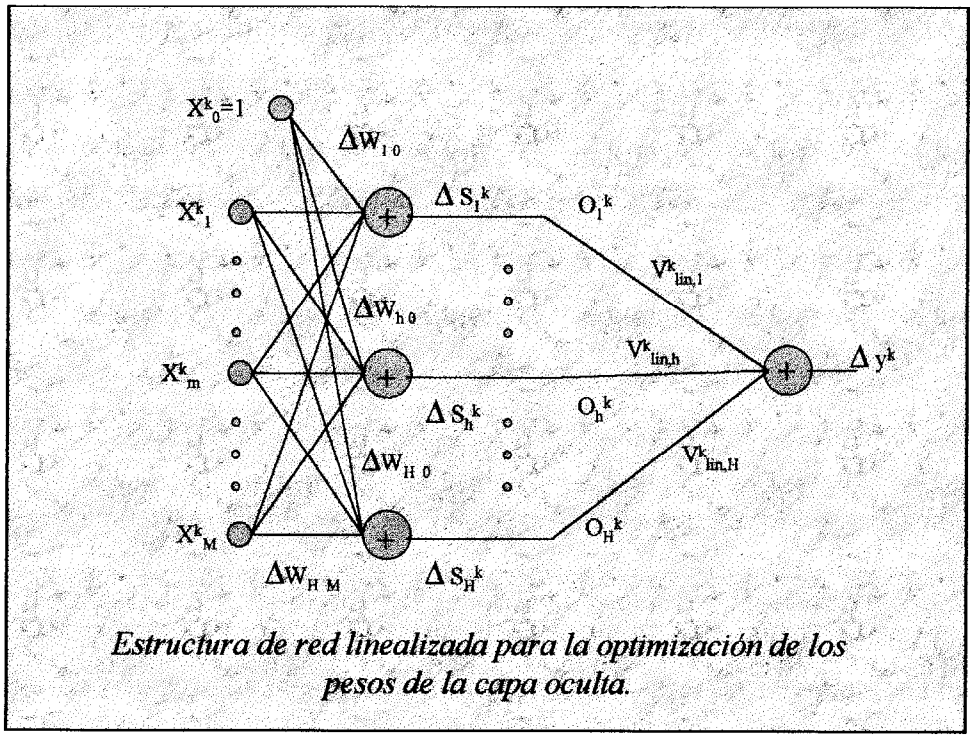
3.1.3 Optimización de la capa oculta.

La optimización de la capa oculta pasa por la minimización del error cuadrático medio medido en la red original de la figura 1 con los pesos de

$$E=\frac{1}{K}\sum_{k=1}^{K}\frac{1}{2}(d^k-y^k_{old})^2=\frac{1}{K}\sum_{k=1}^{K}\frac{1}{2}(e^k_{old})^2$$

Ecuación 3.21

la capa de salida v^{opt}. Para lograrlo la red de la figura 3.4 es entrenada para predecir el error e^k_{old} de la red original.



Estructura de red linealizada para la optimización de los pesos de la capa oculta.

Figura 3.3.

De ahí que se use el conjunto de pares de entrenamiento $\{x^k,e^k_{old}\}^K_{k=1}$ definiéndose una nueva función de coste para la optimización de ΔW en la red linealizada:

$$E_{lin} = \frac{1}{K} \sum_{k=1}^{K} \frac{1}{2}(e_{old}^{k} - \Delta y^{k})^{2}.$$

Ecuación 3.22

Mientras el error de linealización se mantenga bastante pequeño la red de la figura 3.3 es una aproximación apropiada de la red de la figura 3.1 en lo que a los cambios ΔW se refiere.

En la red linealizada las funciones de activación de las neuronas ocultas fueron aproximadas por las expansión de la serie de Taylor de primer orden.

Para pequeños Δs^{k}_{h} el término cuadrático de la serie de Taylor es una buena aproximación del error que se comete debido a la linealización de los nodos de la capa oculta mediante el uso de la ecuación 3.14.

$$e_{h}^{k} = \frac{1}{2} f''(s_{old,h}^{k})(\Delta s_{h}^{k})^{2}$$

Ecuación 3.23

No es posible la formulación del problema de la minimización de la forma usual

$$E_{lin}(\Delta W^{opt}) = \min_{\Delta W} E_{lin}(\Delta W)$$
$$e_{h}^{k}(\Delta W) \leq e_{max}; \quad h = 1...H; \quad k = 1,...,K;$$

Ecuación 3.24

dado el enorme volumen de elementos que incluye (H*K). Más recomendable resulta modificar la función de coste (Ecuación 3.21) mediante la adición de un término de penalización que limite el error de linealización. Multiplicando e^{k}_{h} por los pesos de salida v_{h} tenemos una buena idea de la influencia de cada error de linealización en la salida y^{k}. La introducción del término de penalización estima la calidad de la aproximación lineal mediante el cálculo del promedio del valor absoluto de $|e^{k}_{h}v_{h}|$ para todos los nodos de la capa oculta y todos los pares de entrenamiento.

$$E_{pen} = \frac{1}{K.H} \sum_{k=1}^{K} \sum_{h=1}^{H} \left| e_h^k v_h \right|$$

Ecuación 3.25

La función de coste modificada para la optimización de la capa oculta queda:

$$E_{hid} = E_{lin} + \mu \times E_{pen}$$

Ecuación 3.26

estando E_{lin} de acuerdo con Ecuación 3.22.

El factor de ponderación μ determina la influencia del término de penalización E_{pen} frente a la función de coste lineal E_{lin}.

Es importante la elección de un factor μ adecuado para una rápida convergencia del algoritmo. Si μ es demasiado pequeña son posibles grandes incrementos ΔW dado que el error de linealización no está suficientemente limitado por el término de penalización. Esto provoca que, aunque la función de coste de la capa oculta E_{hid} se minimice, el error predecido del MLP original no decrecerá, sino que se incrementará en la mayoría de los casos siendo por tanto incorrecta la linealización. En el caso opuesto tenemos igualmente problemas. Si μ es demasiado grande solo se permiten pequenos cambios ΔW debido a la intensa influencia del término de penalización. A pesar de que la linealización es válida y la función de coste del MLP decrece, este decrecimiento es excesivamente lento, es decir, convergencia muy lenta.

Para determinar el óptimo valor de μ se recurre a un método heurístico de elección y adaptación de μ que se describe al final en el algoritmo genérico. Para los cálculos siguientes se toma μ como constante.

La minimización de la función de coste de la capa oculta se realiza de una forma parecida a la de la ecuación 3.5 en la capa de salida. Para el cálculo de la derivada parcial de E_{hid} se lleva a cabo separadamente para E_{lin} y E_{pen}.

Primero se calculará la parcial de E_{lin} respecto a los incrementos en los pesos Δw_{hm}.

$$\frac{\partial.E_{lin}}{\partial.\Delta w_{hm}} = \frac{1}{K}\sum_{k=1}^{K}\frac{\partial.E_{lin}^{k}}{\partial.\Delta w_{hm}} =$$
$$= \frac{1}{K}\sum_{k=1}^{K}(\Delta y^{k} - e_{old}^{k})\frac{\partial.\Delta y^{k}}{\partial.\Delta w_{hm}}.$$

Ecuación 3.27

Si se sustituye Δy^k por las ecuaciones 18 y 20 se tiene:

$$\frac{\delta.E_{lin}}{\delta.\Delta w_{hm}} = \frac{1}{K}\sum_{k=1}^{K}\left[\left(\sum_{j=1}^{H} v_{lin_j}^{k}\sum_{i=0}^{M} x_i^k \Delta w_{ji}\right) - e_{old}^{k}\right]$$

Ecuación 3.28

Cambiando el orden de los sumatorias

$$\frac{\delta.E_{lin}}{\delta.\Delta w_{hm}} = \sum_{i=0}^{M}\sum_{j=1}^{H}\Delta w_{ji}\frac{1}{K}\sum_{k=1}^{K} v_{lin_h}^{k} v_{lin_j}^{k} x_m^k x_i^k - \frac{1}{K}\sum_{k=1}^{K} e_{old}^{k} x_m^k v_{lin_h}^{k}$$

Ecuación 3.29

Y si definimos los parámetros

$$a_{hm,ji} = \sum_{k=1}^{K} v_{lin_h}^{k} x_m^k v_{lin_j}^{k} x_i^k;$$
$$\tilde{b}_{hm} = \sum_{k=1}^{K} e_{old}^{k} v_{lin_h}^{k} x_m^k;$$

Ecuación 3.30

Es decir que la derivada queda:

$$\frac{\delta.E_{lin}}{\delta.\Delta w_{hm}} = \frac{1}{K}\left[\sum_{i=0}^{M}\sum_{j=1}^{H}\Delta w_{ji} a_{hm,ji} - \tilde{b}_{hm}\right]$$
$$m = 0,\ldots,M; \quad h = 1,\ldots,H.$$

Ecuación 3.31

Ahora la derivada parcial del término de penalización E_{pen} (Ecuación 24) sustituyendo la ecuación 22 resulta:

$$\frac{\delta.E_{pen}}{\delta.\Delta w_{hm}} = \frac{1}{KH}\sum_{k=1}^{K}\sum_{j=1}^{H}\left|v_j f''(s^k_{old_j})\right|\Delta s^k_j \frac{\delta.\Delta s^k_j}{\delta.\Delta w_{hm}}$$

Ecuación 3.32

Se sustituye Δs^k_j de la ecuación 17 se llega a:

$$\frac{\delta.E}{\delta.\Delta w_{hm}} = \frac{1}{KH}\sum_{i=0}^{M}\Delta w_{hi}\left|v_h\right|\sum_{k=1}^{K}\left|f''(s^k_{old_h})\right|\cdot x^k_i x^k_m$$

Ecuación 3.33

Se define sobre el conjunto de entrenamiento el promedio

$$c_{hm,i} = \left|v_h\right|\sum_{k=1}^{K}\left|f''(s^k_{old_h})\right|\cdot x^k_m x^k_i$$

Ecuación 3.34

Quedando la derivada parcial reducida a:

$$\frac{\delta.E_{pen}}{\delta.\Delta w_{hm}} = \frac{1}{KH}\sum_{i=0}^{M}\Delta w_{hi}\cdot c_{hm,i}$$
$$h = 1,\ldots,H;\quad m = 0,\ldots,M.$$

Ecuación 3.35

Seguidamente se iguala a cero la suma de las dos derivadas parciales para obtener los cambios óptimos en los pesos de la capa oculta ΔW^{opt}:

$$\frac{\delta.E_{hid}}{\delta.\Delta w_{hm}} = 0 = \sum_{i=0}^{M}\sum_{j=1}^{H} a_{hm,ji}\Delta w_{ji} + \frac{\mu}{H}\sum_{i=0}^{M} c_{hmi}\Delta w_{hi} - \tilde{b}_{hm}.$$

Ecuación 3.36

Usando la abreviación se tiene:

$$\tilde{a}_{hm,ji} = \begin{cases} a_{hm,ji} & , j \neq h \\ a_{hm,ji} + \dfrac{\mu}{H} c_{hm,i} & , j = h \end{cases}$$

Ecuación 3.37

Así se obtiene un conjunto de (M+1)H ecuaciones lineales que definen los óptimos cambios en los pesos W de la capa oculta.

Si la matriz de incremento en los pesos ΔW^{opt} es interpretada como un vector de longitud (M+1)H tal que :

$$\Delta W^{opt} = (\Delta w_{10}^{opt}, \ldots, \Delta w_{1M}^{opt}, \ldots, \Delta w_{H0}^{opt}, \ldots, \Delta w_{HM}^{opt})^T$$

Ecuación 3.38

entonces se puede escribir el sistema de ecuaciones de la siguiente forma:

$$\tilde{A} \cdot \Delta W^{opt} = \tilde{b}.$$

Ecuación 3.39

y despejando queda:

$$\Delta W^{opt} = \tilde{A}^{-1} \tilde{b}.$$

Ecuación 3.40

Resulta también aquí que la matriz $\tilde{A}$ es simétrica, positiva definida por lo que se puede aplicar directamente Cholesky a la ecuación 3.39. Como se ha dicho, el método ha sido extraido de [3] y [4]. En el Anexo I se muestra en que consiste el método de Cholesky. También se puede aprovechar la simetría para reducir a la mitad el nº de coeficientes a calcular.

3.1.4 Implementación.

A continuación se muestra un esquema del algoritmo. En el Anexo II se ha incluido el algoritmo completo que se ha usado para la implementación del ejemplo planteado.

Procedimiento oll;
{Inicialización}
Pesos W^{start} y V^{start}.
Factor μ^{start}.
Nº max de iteraciones IC_{max}.
Error máximo permisible Er_max.
Inicializar contador IC=0;
{Entrenamiento}
While (E<Er_max)and(IC<IC_{max}) **do**
    IC=IC+1;
    se optimizan los pesos V
    cálculo de E(W,V)
    optimización_ ΔW
end;
end.

Procedimiento optimizacion_ ΔW ;
Cómputo del cámbio óptimo ΔW^{opt} ;
Cómputo de E^{test}(W,V);
While E^{test} >E **do**
    {incremento de penalización}
    $\mu := \mu \times \gamma$;
    Cómputo del cámbio óptimo ΔW^{opt} ;
    Cómputo de E^{test}(W,V);
end;
W:=W+ΔW^{opt};
E:=E^{test};
{decremento de penalización}
$\mu := \mu \times \beta$;
end.

CAPÍTULO 4: APLICACIÓN A UN PROBLEMA.

El problema que se plantea es el siguiente. A cinco locutores se les ha efectuado una grabación de voz pronunciando los dígitos del cero al nueve. A estas grabaciones se les efectúa un preprocesado clásico que consta de cuatro fases: Filtro de preénfasis, Segmentación, Enventanado y Cálculo de coeficientes. Luego estos coeficientes pasan por un cuantificador vectorial el cual dispone de una librería de 64 vectores código calculados anteriormente. A la salida del cuantificador se le ha hecho pasar por un grupo de diez estructuras, una por cada dígito, que se denominan modelos ocultos de Markov (hidden Markov models, HMM's). Cada una de estas estructuras devuelve por cada vector a su entrada un valor de numérico, que representa la probabilidad de ser el dígito respectivo. Por ejemplo la estructura HMM número 0 nos da, para cada vector cuantificado de cada señal de voz que se presente a su entrada, un valor numérico que es representativo de la probabilidad de que se trate de un "cero".

Actualmente se están realizando trabajos para mejorar la tasa de reconocimiento añadiendo estructuras/modelos tales como las Redes Neuronales a la salida de los modelos de Markov. Se ha querido comparar los resultados de aplicar este nuevo algoritmo, el OLL, al MLP clásico y estudiar sus ventajas y desventajas respecto del BP.

Aquí es donde comienza el cometido de la Red neuronal que se estudia en este proyecto fin de carrera .Con las diez salidas numéricas por cada señal de voz, esas diez probabilidades, se trata de entrenar diez redes neuronales, una por cada dígito, de las que se active la salida de la red correspondiente al dígito en cuestión. Siguiendo el ejemplo anterior, como entrada para cada una de las diez redes neuronales que se pretende entrenar se tienen las diez probabilidades de los diez HMM's.

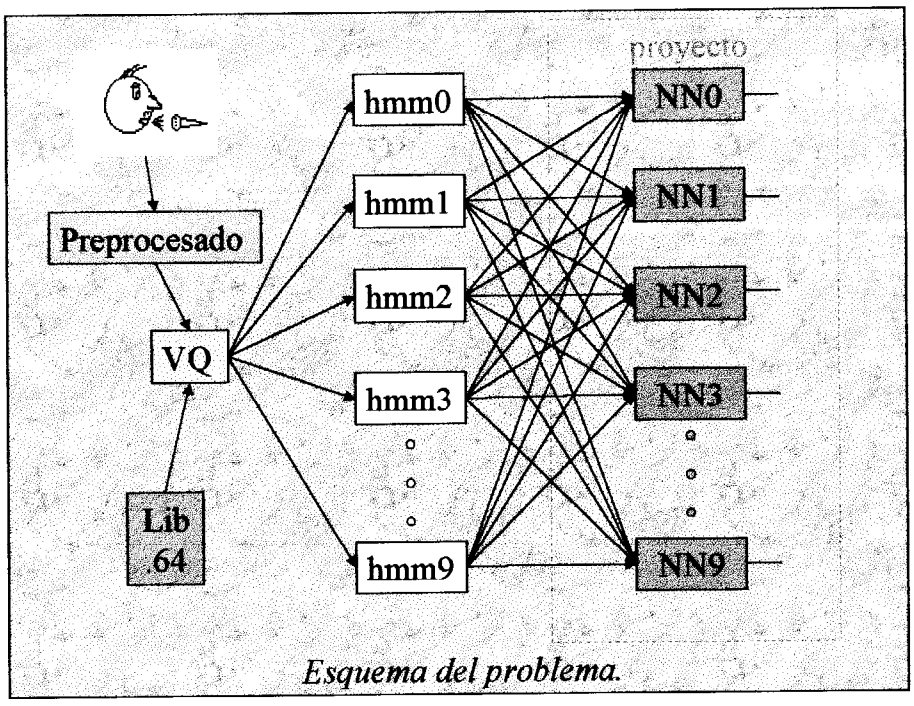


Figura 4.1

Como se trata de un "cero" la salida de todas las redes neuronales estarán inactivas excepto la del dígito 0. Para un mejor entendimiento véase la figura 4.1.

El propósito de este proyecto de fin de carrera es comparar la velocidad de entrenamiento y la precisión del mismo con el algoritmo clásico de backpropagation y con el algoritmo de linealización OLL.

En el anexo II y III se muestran dos de los algoritmos usados, empleando los métodos de entrenamiento BP y OLL respectivamente..

Los valores que dan las estructuras HMM siguen una ley logarítmica y no están normalizados. Estos valores han de ser normalizados antes de usarlos para el entrenamiento o prueba de las redes neuronales. El algoritmo de normalización se muestra en el anexo IV.

Se trata de probar con cada locutor diez redes neuronales. Cada red sigue un esquema como el de la figura 4.2.

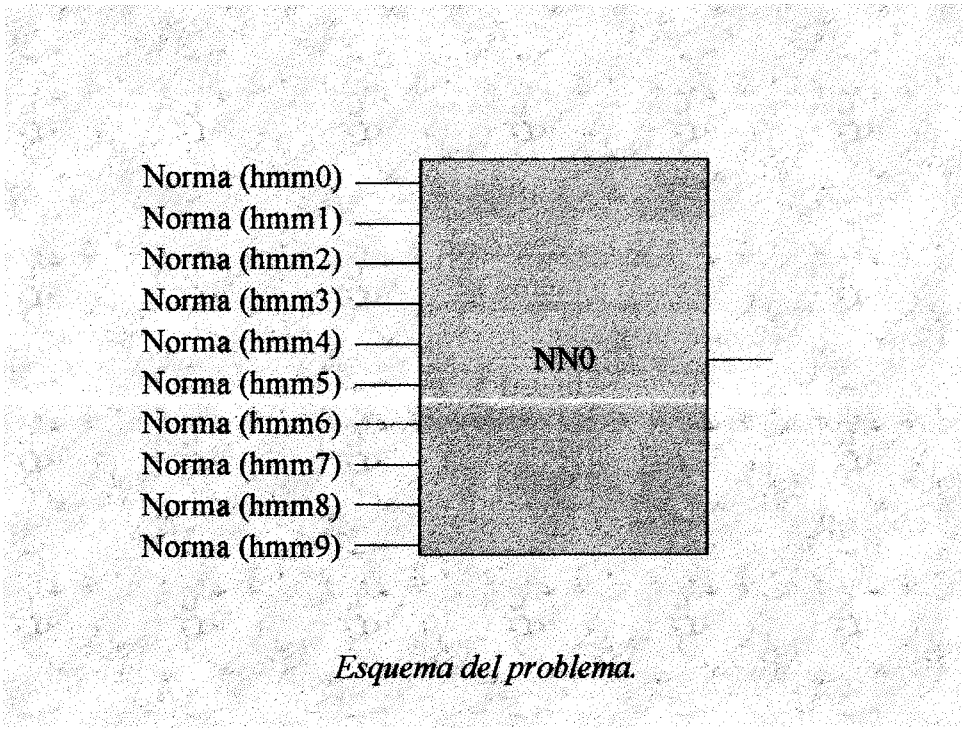


Figura 4.2.

4.1 Implementación con Backpropagation.

A continuación se muestran los resultados del entrenamiento con el algoritmo de Backpropagation. Se estudian en cada red entrenada tres parámetros para efectuar la comparación: La tasa de error con los pares de entrenamiento como entrada, la tasa de error con los pares de test como entrada y el tiempo aproximado que duró el entrenamiento. Las tasas de error se miden en tanto por ciento y el tiempo en minutos.

Por cada uno de los cinco locutores se dispone de dos conjuntos de pares. El primer conjunto, o conjunto de entrenamiento, consta de trescientos veinte pares, treinta y dos pares por cada dígito. El segundo conjunto son mil pares de entrenamiento, cien por cada dígito, y se denomina conjunto de test. El primer conjunto se utiliza para entrenar la red y el segundo para evaluar la bondad del entrenamiento. Se separan en dos conjuntos para garantizar la generalidad del entrenamiento, propiedad que se enunció en la introducción. Como el entrenamiento se realizó utilizando el conjunto de pares de entrenamiento, se apreciará que la tasa de error para los pares de entrenamiento es superior generalmente a la tasa de error para los pares de test.

A continuación se explica qué es y como se efectúa la medida de la tasa de error. La tasa de error la defino de la siguiente manera. Dado un conjunto de pares, el porcentaje de aciertos que se producen al poner como entrada todas las entradas del citado conjunto de pares y comparar con sus salidas las que se obtienen de la red neuronal .

Dado que en la práctica nunca se obtendrán valores exactos de "1" ó "0" a la salida de la red, es preciso efectuar una discriminación fiable. En este caso para abordar este problema he optado por realizar la discriminación tomando el valor medio entre el máximo y el mínimo que se obtienen como salida del conjunto que se esté evaluando. También se podría haber realizado calculando, de la misma manera, un umbral estático con el conjunto de pares de entrenamiento. En el anexo V se muestra el algoritmo utilizado.

A continuación se expone la tabla de resultados del entrenamiento de las cincuenta redes , diez por locutor, que se han entrenado con el algoritmo de backpropagation (Tabla 4.1). También se muestran en las figuras 4.3 a la 4.7 esos

resultados de manera más gráfica. En todas las gráficas donde se muestran las tasas de error la primera columna de cada dígito se refiere a tasa de error con los pares de test y la segunda con los pares de entrenamiento.

Locutor	Dígito	0	1	2	3	4	5	6	7	8	9	Medias
1	Entrena.	100	100	100	100	100	100	100	100	100	100	100
	Test	96,8	99,4	99,6	99,3	99,8	98,4	99,8	98,8	98,3	97,4	98,76
	Tiempo	9,7	4,9	6,6	9,5	6,2	10,6	6,6	8,8	10,8	11,2	8,49
2	Entrena.	100	100	100	100	100	100	100	100	100	100	100
	Test	100	99,9	100	99,4	98,9	98,8	99,2	98,8	99,2	99,8	99,4
	Tiempo	43,5	10,1	7,4	11,5	13,2	4,9	12,6	8,2	12,1	6,6	13,01
3	Entrena.	100	100	100	100	100	100	100	100	100	100	100
	Test	96,1	99,3	98,9	99,3	96,2	98,4	97,9	96	98	98,5	97,86
	Tiempo	15,1	15,1	11,5	15,1	15,1	15,1	15,1	15,1	15,1	15,1	14,74
4	Entrena.	100	100	100	100	100	100	100	100	100	100	100
	Test	94,5	99,8	99,6	98,4	92,3	99,5	99,1	97,7	96,1	97	97,4
	Tiempo	15,1	15	8,8	12,2	15,1	6,1	15,1	10,8	15,1	9,5	12,28
5	Entrena.	100	100	100	100	100	100	100	100	100	100	100
	Test	96,7	100	92,2	98	90,2	91,7	90,1	90,5	98,7	96,7	94,48
	Tiempo	15,1	5,3	3,83	15,1	15,1	6,6	15,1	15	15,1	8	11,423

Tabla 4.1

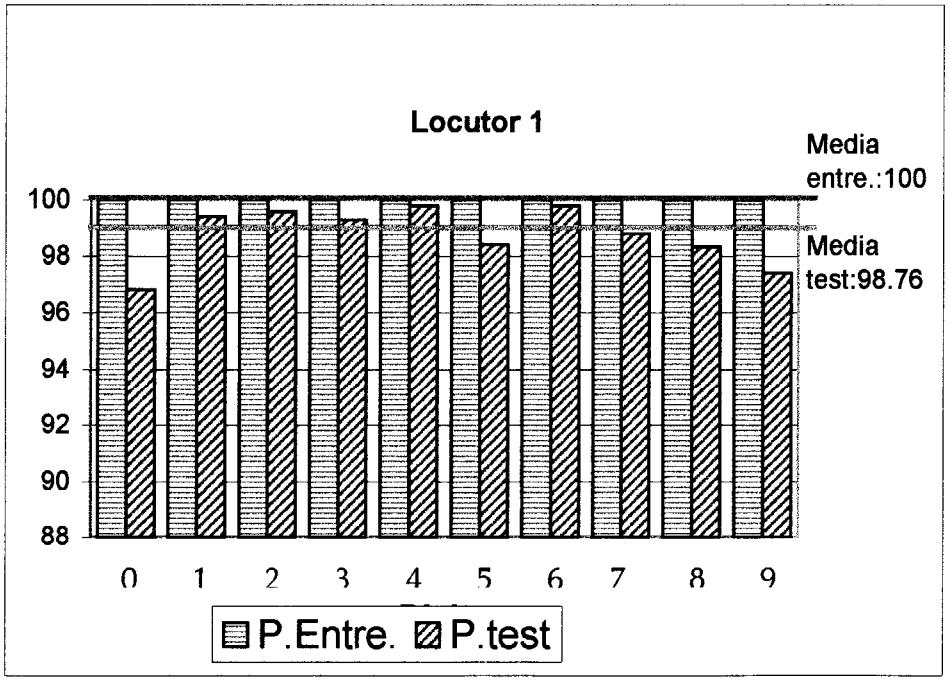


Figura 4.3.

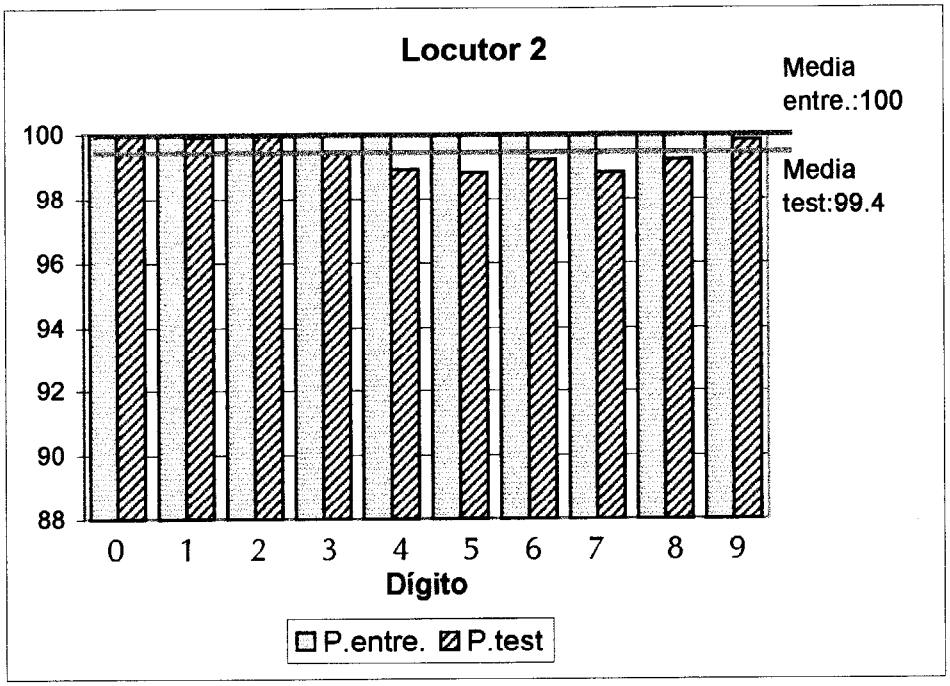


Figura 4.4.

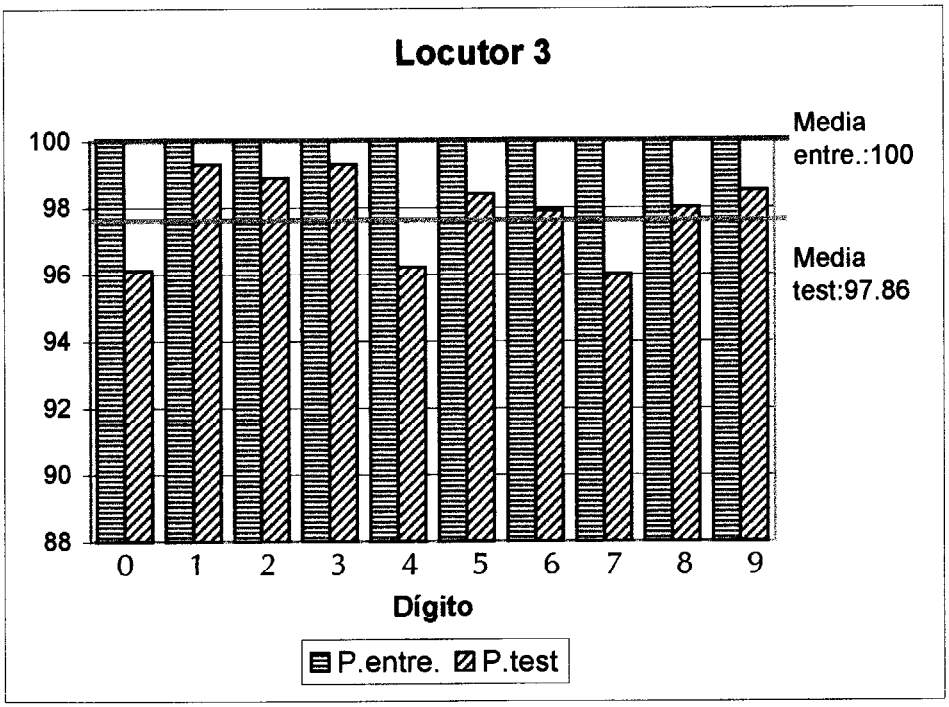


Figura 4.5.

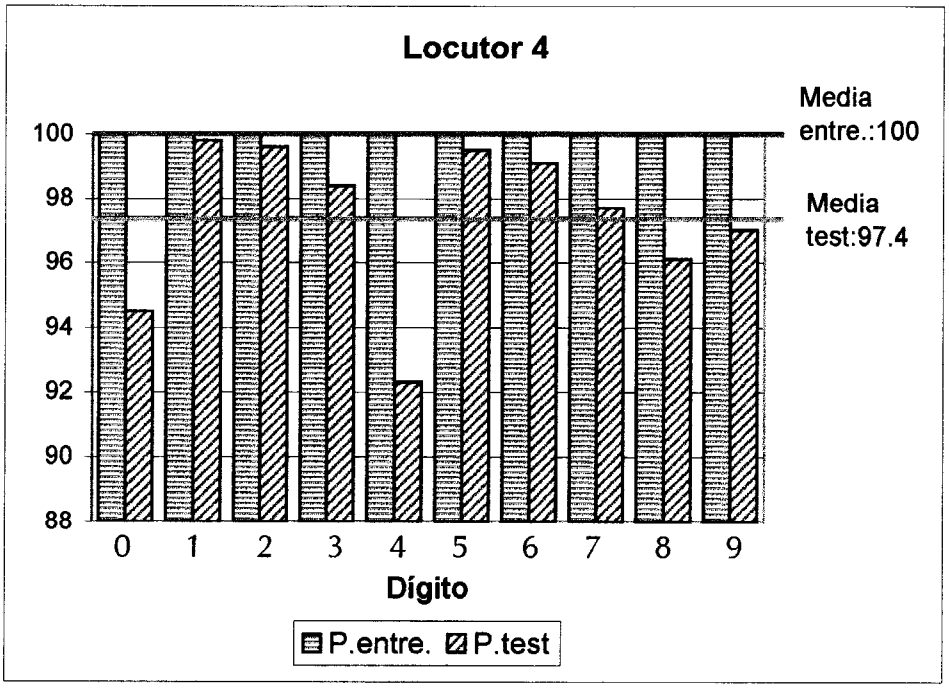


Figura 4.6.

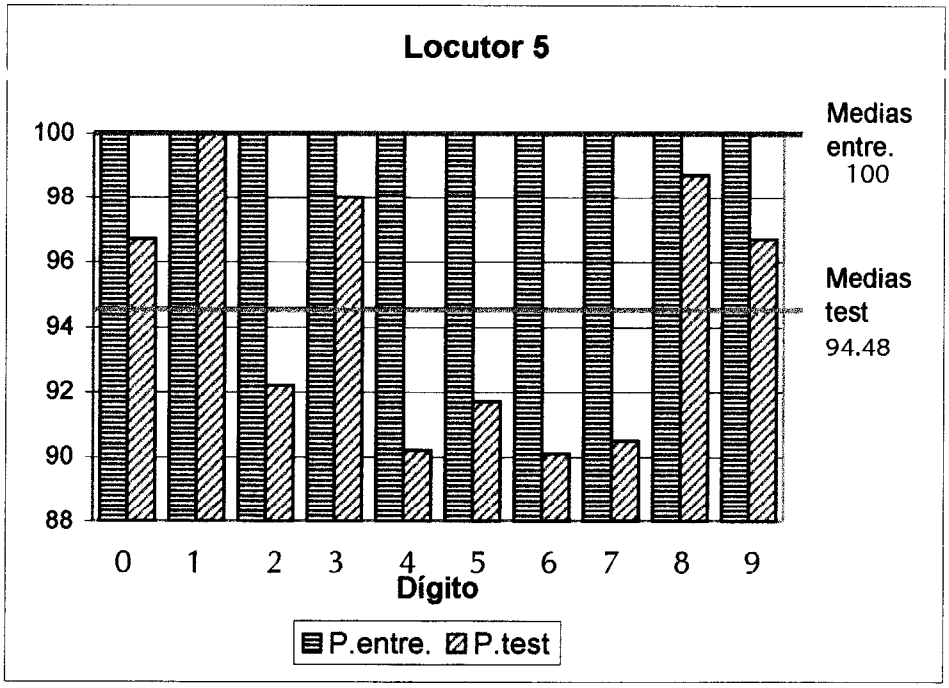


Figura 4.7.

Seguidamente se muestran los valores medios de las tasas de error respecto a los locutores y los tiempos medios, en las figuras 4.8 y 4.9.

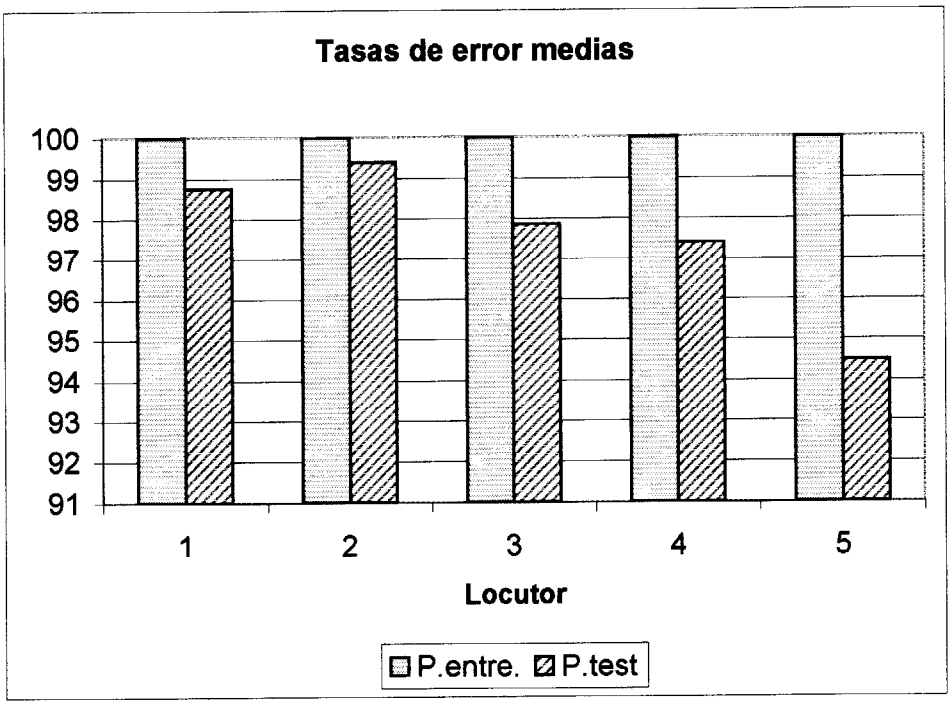


Figura 4.8

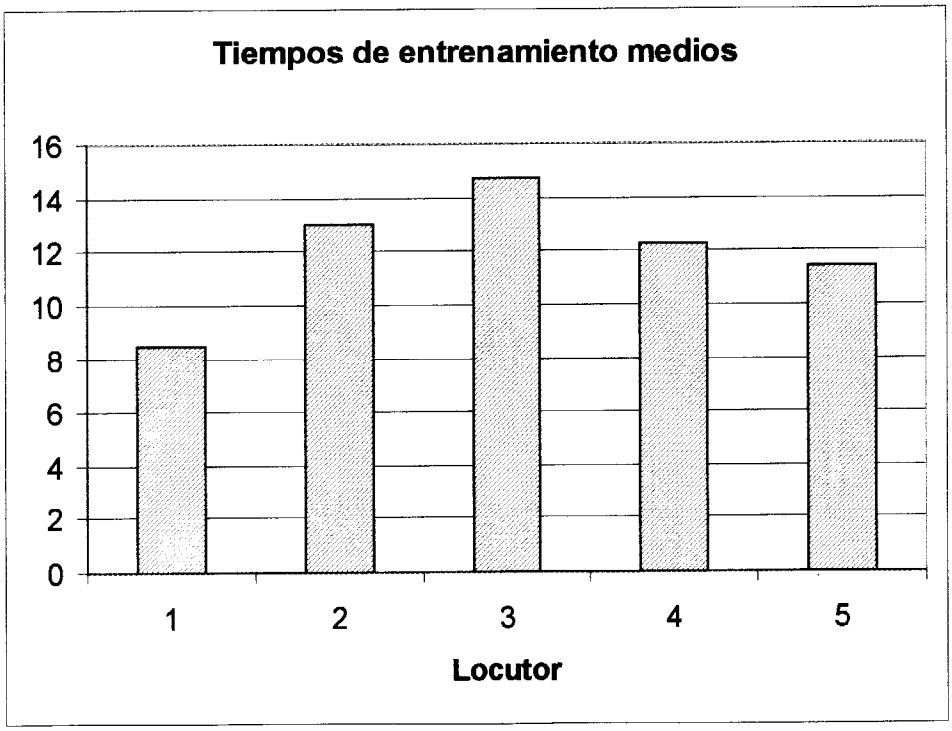


Figura 4.9

4.2 Implementación con el OLL.

Seguidamente se muestran los resultados obtenidos de una manera análoga a la anterior pero esta vez con el algoritmo de linealización. Se muestran las mismas variables que en el apartado anterior para los cinco locutores y también como en el apartado anterior se muestran los valores medios.

Locutor	Dígito	0	1	2	3	4	5	6	7	8	9	Medias
1	Entrena.	100	100	100	100	99,1	100	100	100	100	100	99,91
	Test	95,8	98	99,2	99,8	95,3	98	95,3	97,4	95,9	97,8	97,25
	Tiempo	7	5,9	2	2,7	0,1	6,5	1,2	6,1	7	2,8	4,13
2	Entrena.	100	100	97,8	100	100	100	100	98,1	100	100	99,59
	Test	99,6	99,6	91,4	99,4	99,7	99,1	99,4	95,5	99,9	100	98,36
	Tiempo	1,5	7	0,1	3,5	2	3,7	3,9	0,1	2,4	7	3,12
3	Entrena.	100	100	100	100	100	100	100	100	100	100	100
	Test	97,1	99,6	99	98,7	96,4	98,6	99,3	95,4	96,2	97,9	97,82
	Tiempo	5,4	3,6	7	3	2,7	3,2	1,4	3	2	2,7	3,4
4	Entrena.	100	100	100	100	100	100	100	100	100	94,7	99,47
	Test	94,1	96,1	97,1	98,9	93,1	97,6	98,8	99,9	95,2	97,6	96,84
	Tiempo	6,1	1,5	2,8	8	3	1,9	4,6	6,8	1,7	0,1	3,65
5	Entrena.	100	100	99,7	100	100	100	100	100	100	100	99,97
	Test	98,8	99,8	98,8	95,2	90,1	94,4	90,1	99,6	80,1	99,1	94,6
	Tiempo	6,8	4,2	0,1	2,7	1,8	1,3	1,9	6,9	4,1	6,8	3,66

Tabla 4.2

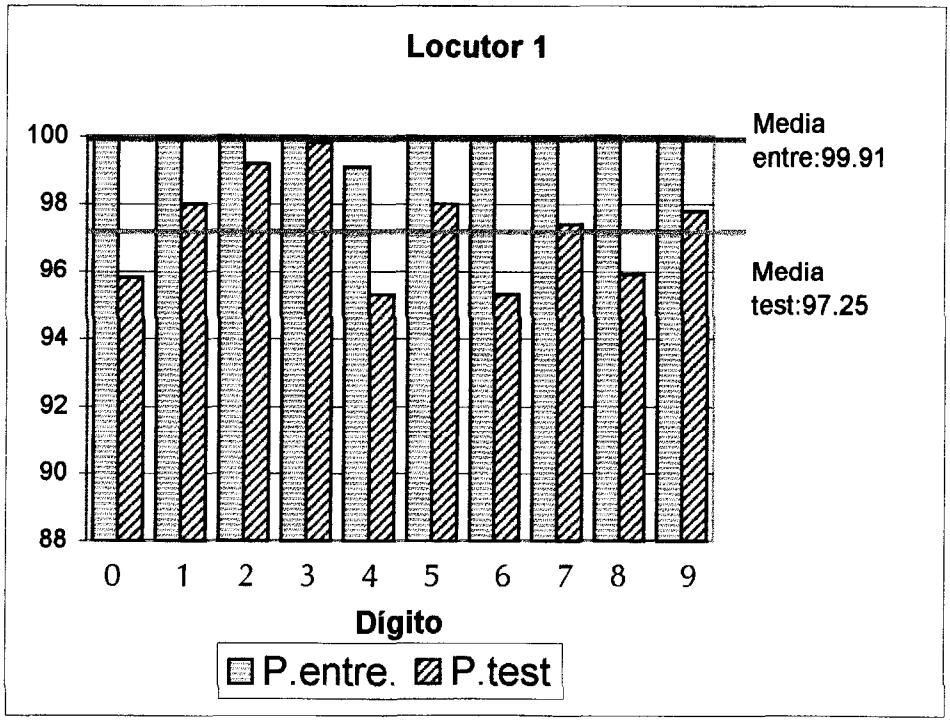


Figura 4.10.

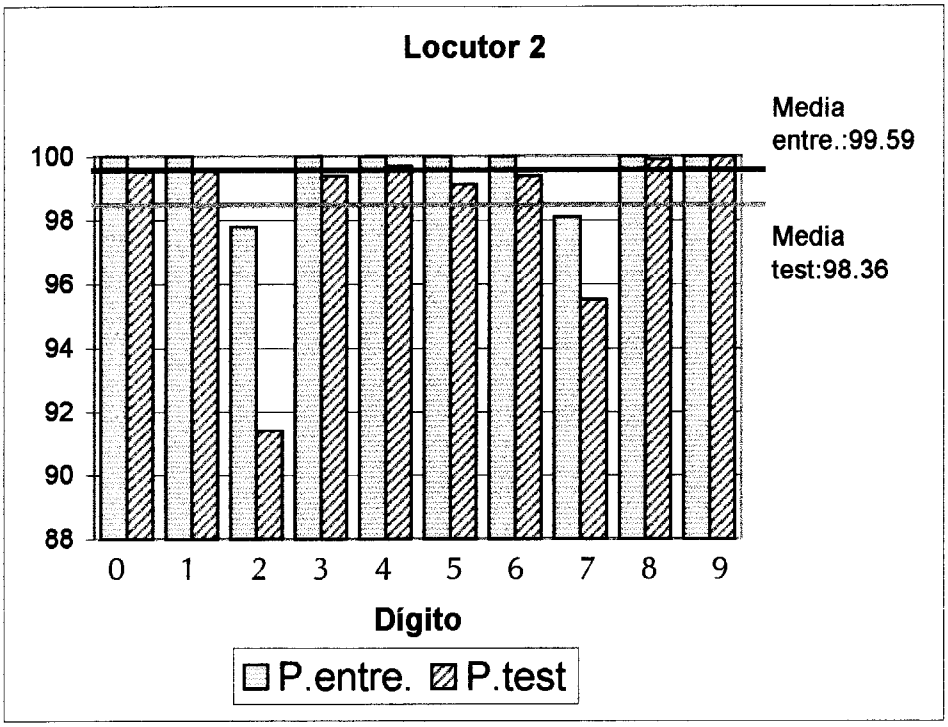


Figura 4.11.

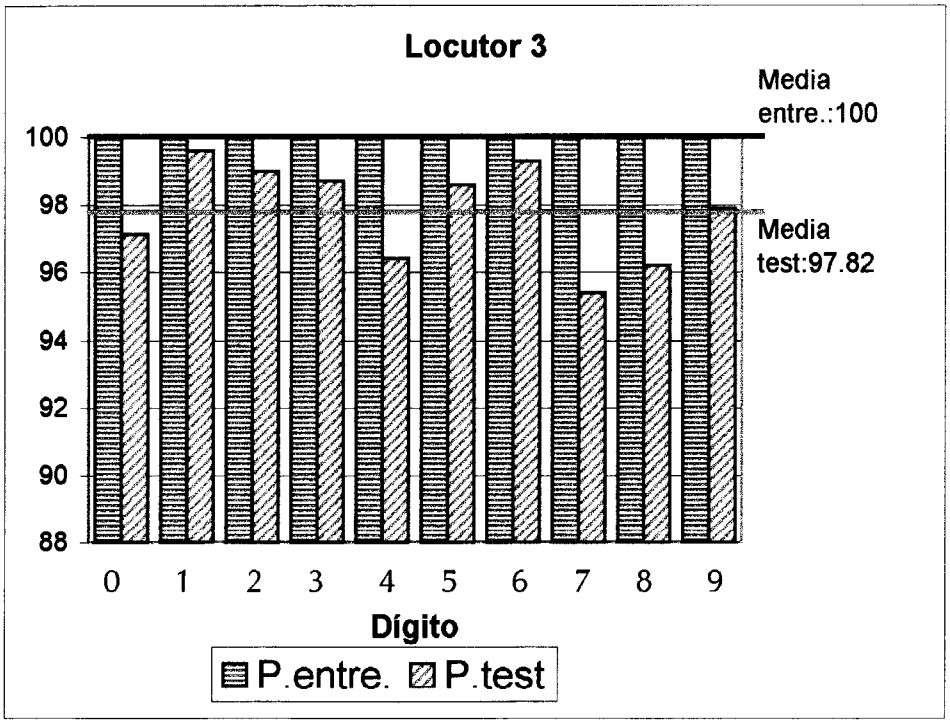


Figura 4.12.

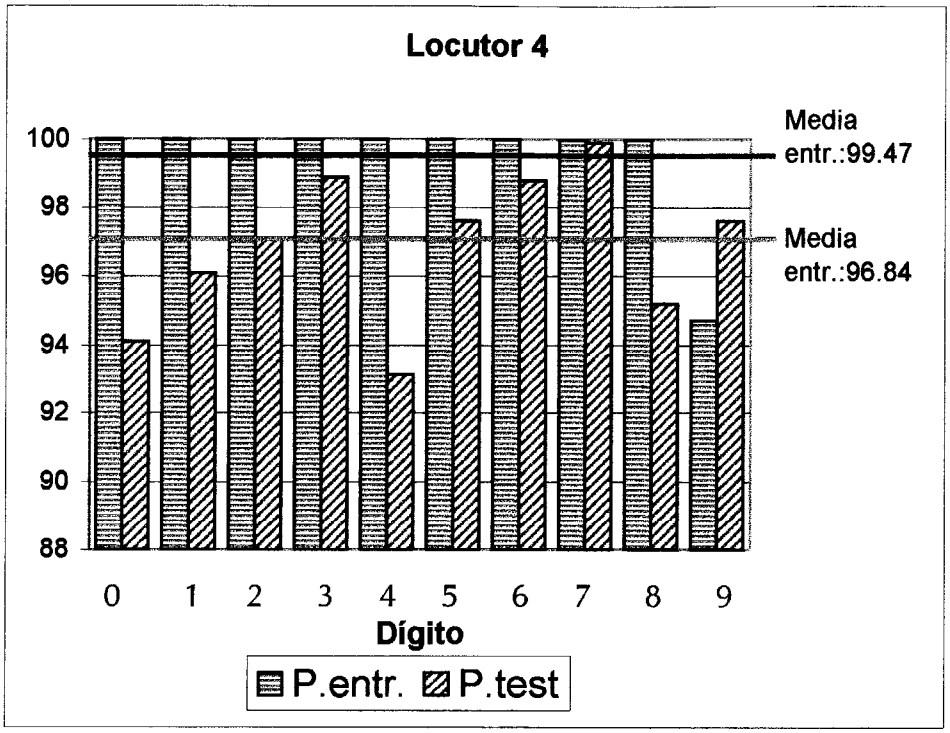


Figura 4.13

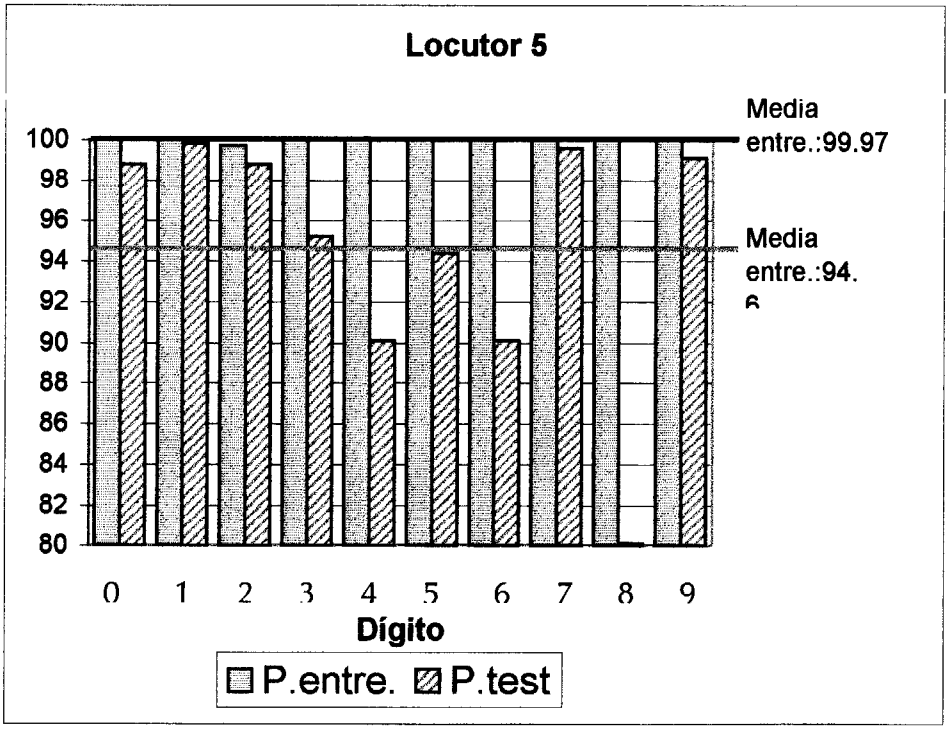


Figura 4.14.

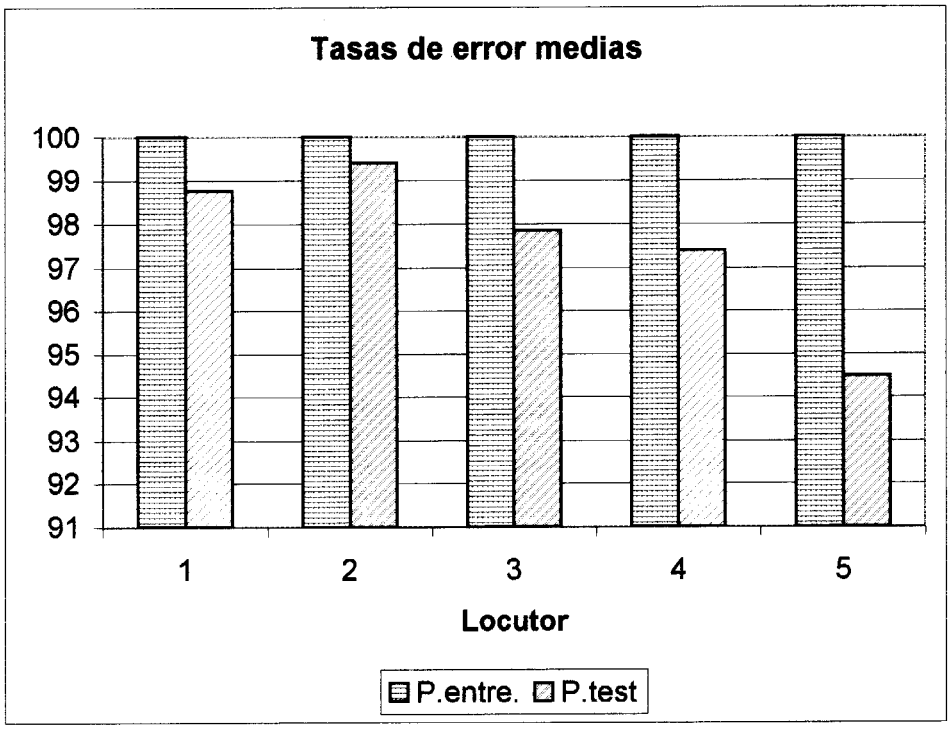


Figura 4.15.

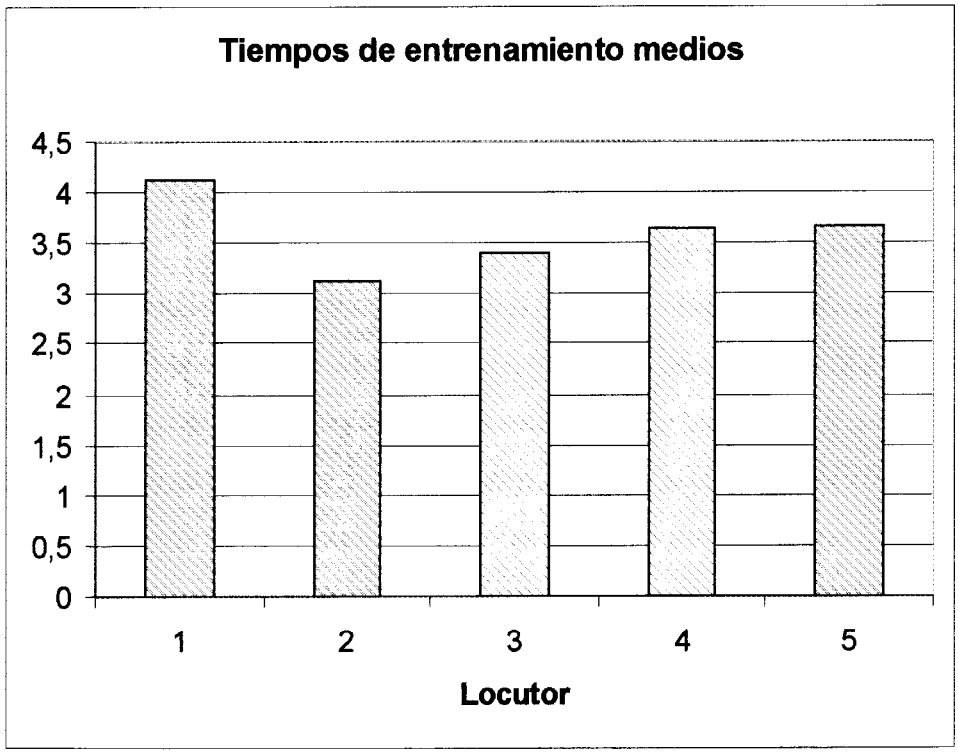


Figura 4.16.

4.3 Análisis comparativo de los dos algoritmos.

Para profundizar en el análisis se han elaborado la tabla 4.3 y las figuras 4.17, 4.18 y 4.19. A la vista de los resultados obtenidos en las pruebas efectuadas, mostrados en los dos apartados anteriores se pueden obtener varias conclusiones. En primer lugar se puede apreciar que los tiempos de entrenamiento efectivamente se reducen considerablemente con el nuevo algoritmo, respecto del entrenamiento con el Backpropagation. Véase figura 4.19. Se puede apreciar que los tiempos medios se reducen considerablemente para todos los locutores. Esta reducción se sitúa entre dos y cinco veces el tiempo de entrenamiento del algoritmo de Backpropagation.

Por otra parte en lo que a las tasas de error se refiere se puede observar en la tablas 4.3 que con los pares de entrenamiento, la tasa de error es un poco menos favorable. También las tasas medias con los pares de entrenamiento son un poco inferiores excepto en el locutor quinto, para el cual es ligeramente superior.

Cabe reseñar que el algoritmo OLL es de un nivel de complejidad muy superior al Backpropagation lo que puntúa en su contra, ya que puede presentar muchos más problemas a la hora de su implementación. Problemas típicos pueden ser los derivados de la necesidad de que ciertas matrices cumplan las propiedades que deben cumplir, ser simétricas, positiva y definidas, para poder aplicar el método de Cholesky que se mencionó.

Otro problema endémico de este algoritmo es que reduce el error hasta cierto nivel, a partir del cual ya no lo reduce más. El algoritmo de Backpropagation sin embargo, a pesar de se va ralentizando, si se le permite, mediante una contante μ suficientemente pequeña, seguirá reduciendo el error siempre, consiguiendo, a cambio de un enorme tiempo de entrenamiento la precisición que se desee.

Por último merita mención el hecho de que en el caso de que no se utilice el método de Cholesky para la resolución del sistema de ecuaciones, pueden presentarse problemas de precisión a la hora de resolver las inversas de las matrices. El rango de valores de esas matrices se va reduciendo hasta que hace abortar el programa sin conseguir la reducción del error.

En resumidas cuentas se puede decir que el algoritmo de linealización OLL tiene la deseable capacidad de reducir cuantiosamente el tiempo de entrenamiento con el coste de una pequeña pérdida de precisión en el entrenamiento y un gran incremento en la complejidad de los métodos a utilizar.

Locutor		BP	OLL
1	P. Entrenamiento	100	99.91
	P. Test	98.76	97.25
	Tiempo entrena.	8.49	4.13
2	P. Entrenamiento	100	99.59
	P. Test	99.4	98.36
	Tiempo entrena.	13.01	3.12
3	P. Entrenamiento	100	100
	P. Test	97.86	97.82
	Tiempo entrena.	14.74	3.4
4	P. Entrenamiento	100	99.47
	P. Test	97.4	96.84
	Tiempo entrena.	12.28	3.65
5	P. Entrenamiento	100	99.97
	P. Test	94.48	94.6
	Tiempo entrena.	11.42	3.66

Tabla 4.3

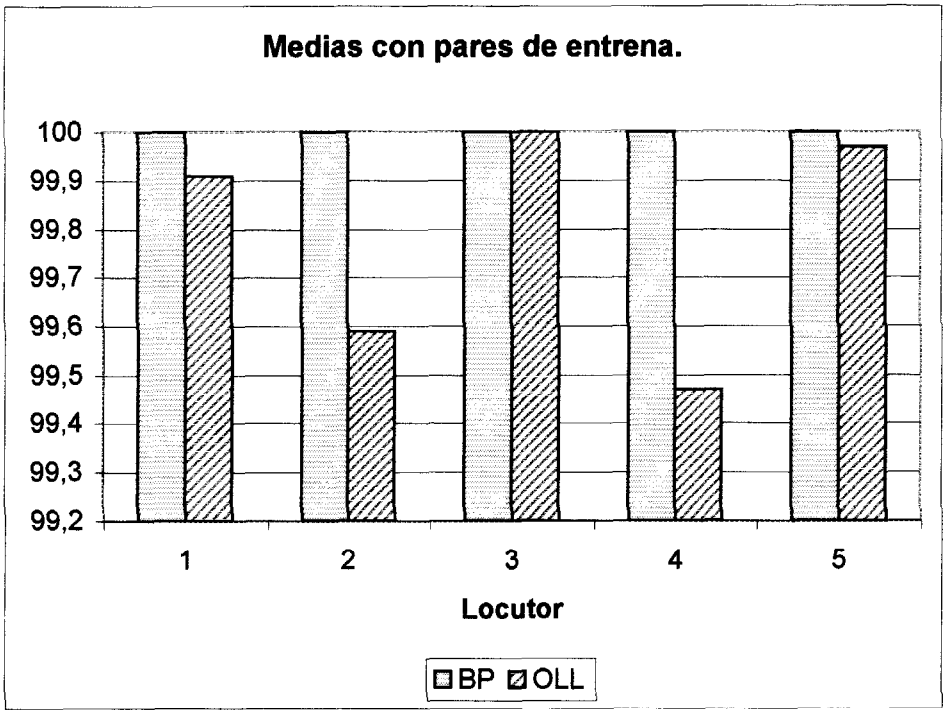


Figura 4.17

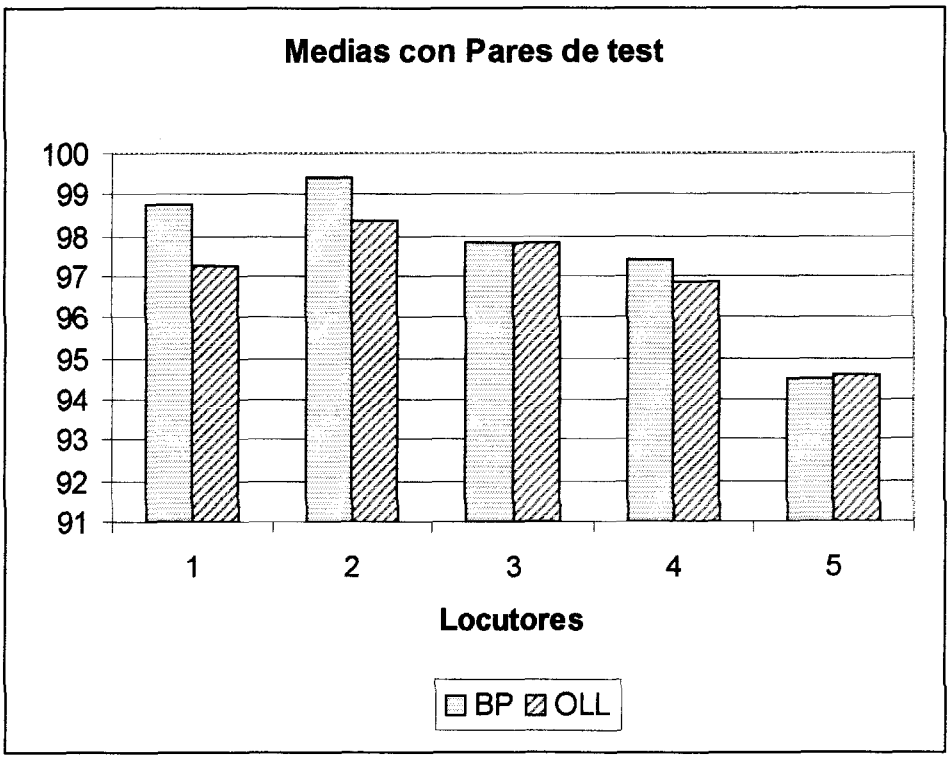


Figura 4.18

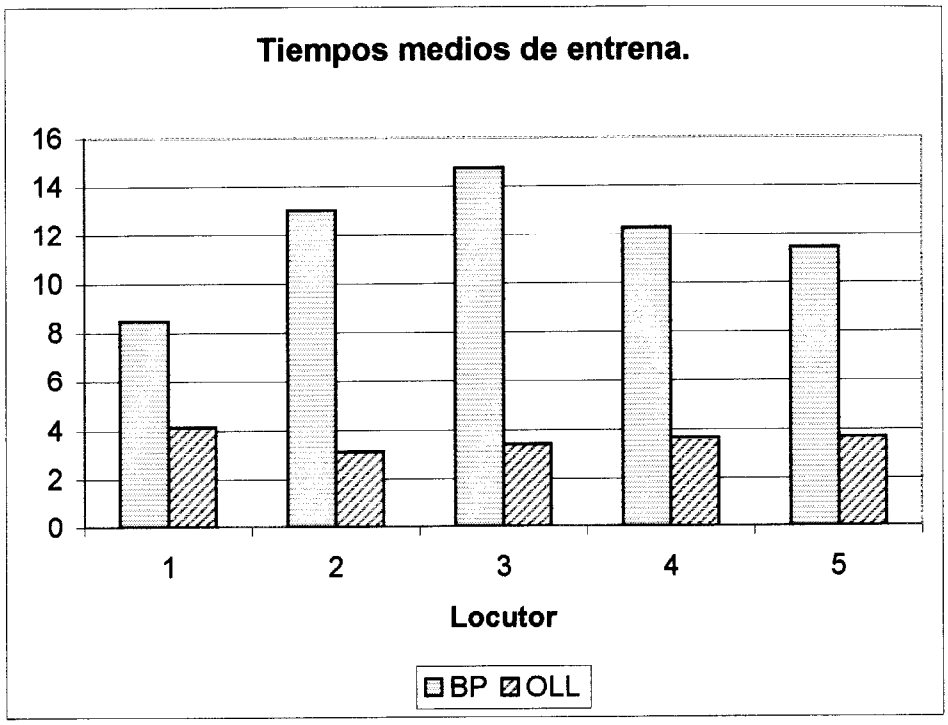

Tiempos medios de entrena.
16
14
12
10
8
6
4
2
0
1
2
3
4
5
Locutor
BP
OLL

Figura 4.19

ANEXOS

ANEXO I : MÉTODO DE CHOLESKY.

Este método ha sido extraido de [3] y [4].

El método de Cholesky es un método de resolución de sistemas de ecuaciones lineales. No es más que una particularización del método de factorización LU. Este método consiste en dado un sistema de ecuaciones tal como el de la ecuación 7.1

$$A \times X = B$$

Ecuación 7.1

descomponer la matriz A en producto de dos matrices L y U triangulares inferior y superior respectivamente de tal forma que nos quede:

$$L \times U \times X = B$$

Ecuación 7.2

y haciendo el cambio de variable que se indica en la ecuación 7.3 se resuelve recursivamente el sistema de ecuaciones.

$$U \times X = Y$$
$$L \times Y = B$$

Ecuación 7.3

El método de Cholesky funciona de manera análoga pero con la particularidad de que la matriz A es positiva definida por lo que al obtener la descomposición en factores nos queda

$$A = R^T \times R$$
$$R^T = L; \quad R = U;$$

Ecuación 7.4

por lo que se acelera la resolución del sistema de ecuaciones porque sólo es necesario el cálculo de una matriz, R.

Una vez descompuesta la matriz A en dos matrices triangulares, se resuelven sendos sistemas utilizando dos métodos iterativos. Estos se denominan algoritmo de sustitución hacia delante cuando es triangular inferior, y algoritmo de

sustitución hacia atrás cuando es triangular superior. Ambos se muestran a continuación.

ABACK.M

```
function c=aback(U,b);
%U es una matriz triangular superior y b una columna.
%este algoritmo nos da la solución a la columna de incógnitas.

D=size(U);
D=D(1);

c=[];
for i=D:-1:1
   c(i)=b(i)/U(i,i);
   for j=D:-1:i+1
      c(i)=c(i)-U(i,j)*c(j)/U(i,i);
   end
end
```

AFORWARD.M

```
function x=aforward(L,c);
%L es una matriz triangular inferior y c una columna.

%este algoritmo nos da la solución a la columna de incógnitas.

D=size(L);
D=D(1);
x=[];
for i=1:D
   x(i)=c(i)/L(i,i);
   for j=1:i-1
      x(i)=x(i)-L(i,j)*x(j)/L(i,i);
   end
end
```

El algoritmo para la resolución del sistema de ecuaciónes usando el método de Cholesky es el siguiente. Para el cálculo de la matriz R citada arriba se utiliza la función de Matlab *chol*.

DEFIPOSI.M

```
%esta es una rutina que dado un sistema de ecuaciones tal como
%                          Ax=b
%donde A es una matriz definida positiva, resuelve los valores de x
%usando el algoritmo de Cholesky, de sustitución hacia adelante y
%de sustitución hacia atrás.
function [x]=defiposi(A,b);

n=length(b);

R=chol(A);  %R es triangular superior y R' triangular inferior tal que
%    R'*R=A  ==>  R'*c=b  ==> R*x=c

c=aforward(R',b);
```

```
x=aback(R,c');
x=x';
```

ANEXO II : ALGORITMO PARA EL ENTRENAMIENTO DE REDES NEURONALES BACKPROPAGATION.

A continuación se muestra el algoritmo Backpropagation y las funciones adicionales utilizadas. Para el entrenamiento en serie de todas las redes se parte de que son conocidas las variables *n* y *l*, que son respectivamente el dígito y el locutor.

BP.M

```
%constantes
mu=0.03;
max_iter=15000;
max_error=0.01;
umbral=0.000001;

%carga de los pares de entrenamiento X y Y
%inicialización
%red 10 10 1
M=10; %n° de nodos de la capa de entrada
H=10; %n° de nodos de la capa oculta
O=1;  %n° de nodos de la capa de salida

W10=zeros(H,M);
B10=zeros(H,1);
W20=zeros(1,H);
B20=zeros(O,1);

%atención que estoy inicializando todos los pesos a 0 buscando un
%menor tiempo de entrenamiento

%se parte de que se conoce l, n.
load ([num2str(l) '\logpoe32']); %se carga el conjunto de pares de
%entrenamiento
K=320;   %número de pares de entrenamiento
X=norma(logPO);   %normalización de las entradas
for i=1:K
   if mod(i,10)==mod(n+1,10) %seleccionamos las columnas respectivas
      Y(i)=1;                %al dígito que se esté entrenando.
   else
      Y(i)=0;
   end
end

%inicialización de los demás parámetros
error=1;
traza_error=[1];
traza_er=[1];
er=1;
i=0;
```

```
%entrenamiento
t0=cputime; %inicialización del reloj para el cálculo de la duración
while (i<max_iter)&(error>max_error)&((abs(er)>umbral))
   %feedforward
      u1=feval('funact',W10*X+B10*ones(1,K),1);
      u2=feval('funact',W20*u1+B20*ones(1,K),1);


   %gradiente
      E=Y-u2;
      E2=0.5*sum((sum(E.^2))'); %error cuadrático medio
      error=E2;

      d2=u2.*(1-u2).*E;
      d1=u1.*(1-u1).*(W20'*d2);
      %aprendizaje
      if E2>max_error
            W10=W10+mu*d1*X' ;B10=B10+mu*(sum(d1'))';
            W20=W20+mu*d2*u1' ;B20=B20+mu*(sum(d2'))';
      end

   %construcción dibujos
er=(E2-
traza_error(length(traza_error)))/traza_error(length(traza_error));
      traza_er=[traza_er er];
      traza_error=[traza_error E2];

      i=i+1;
end  %(while)
t=cputime-t0     %cálculo del tiempo

traza_error=traza_error(2:length(traza_error)); %construcción de
                                                %gráficas de error
traza_er=traza_er(2:length(traza_er));
```

FUNACT.M

```
function [y]=funact(x,g)    %función de activación:sigmoide

y=1./(1+exp(-g*x));
```

También se añaden las rutinas de test que se emplearon para el cálculo de las tablas. La rutina *test* es con los pares de test y la rutina *teste* es con los pares de entrenamiento. *tasa_er* calcula la tasa de error que se ha enunciado.

TEST.M

```
%Parto de que se conocen las matrices de pesos W y V

function [p]=test(n,l,W,V)
%n es el dígito y l el locutor
  load ([num2str(l) '\logpot32']);%se carga el conjunto de pares de
%test
   N=norma(logPO);   %normalización de las entradas
```

```
   for i=1:1000
      if mod(i,10)==mod(n+1,10)
         j(i)=1;   %salida correcta para el reconocimiento del dígito
      else
         j(i)=0;
      end
   end

p=tasa_er(N,j,W,V);
```

TESTE.M

```
%Parto de que se conocen las matrices de pesos W y V
function [p]=teste(n,l,W,V);
%n es el dígito y l el locutor
  load ([num2str(l) '\logpoe32']);%carga del conjunto de entrenamiento
   N=norma(logPO);      %normalización del conjunto
   for i=1:1000
      if mod(i,10)==mod(n+1,10)
         j(i)=1;              %salida correcta para el reconocimiento
                              %del dígito correspondiente
      else
         j(i)=0;
      end
   end
p=tasa_er(N,j,W,V);
```

TASA_ER.M

```
function [p]=tasa_er(X,Y,W,V);
%Función que devuelve el porcentaje de aciertos que se dan en la red
%ya entrenada usando una red backpropagation.
l=size(X); %dimensiones

X=[X;ones(1,l(2))];

o=feval('funact',W*X,1);
o=[o;ones(1,l(2))];    % entrada 1 para las bias de esta capa
y=feval('funact',V*o,1);
[y]=regenera(y);

contador=0;
for i=1:l(2)
    if y(i)==Y(i)
         contador=contador+1;
    end
end

p=contador*100/l(2);   %cálculo del porcentaje
```

ANEXO III : ALGORITMO LINEALIZADO O.L.L. PARA EL ENTRENAMIENTO DE REDES NEURONALES.

A continuación se muestra el algoritmo OLL y las funciones adicionales utilizadas. Para agilizar el entrenamiento en serie de todas las redes se parte de que son conocidas las variables *n* y *l* respectivamente el dígito y el locutor.

OLL.M

```
%implementación de una red OLL (Optimization
%Layer by Layer) desarrollado por Ergezinger y Thomsen.

%parto de Er_max,n y l.
%inicialización
%red 10 10 1
M=10; %n° de nodos de la capa de entrada
H=10; %n° de nodos de la capa oculta
O=1;  %n° de nodos de la capa de salida

W0=randn(H,M+1); %las bias se tratan como un peso de entrada más
V0=randn(1,H+1); %como si tuvieran señal 1 a su entrada

load ([num2str(l) '\logpoe32']); %carga de los pares de entrenamiento
K=320;   %número de pares de entrenamiento
X=norma(logPO);  %normalización de los pares de entrada
for i=1:K
   if mod(i,10)==mod(n+1,10) %seleccionamos las col
      Y(i)=1;                %al dígito correspondiente
   else
      Y(i)=0;
   end
end

X=[X;ones(1,K)]; %las entradas de las bias de la primera capa en la
                 %última fila
ICmax=100;     %n° de iteraciones máximo
mu=0.01;       %valor inicial de la constante mu
gamma=1.5;     %factor de crecimiento del término de penalización
beta=0.9;      %factor de decrecimiento del término de penalización
%Er_max=0.08; %Error máximo permitido

%entrenamiento
IC=0;
er=1;
traza_er=[];
E=1;E0=2;
traza_error=[];
%traza_mu=[];
t0=cputime;  %inicialización para la medida del tiempo
while (IC<ICmax)&(E>Er_max)
   IC=IC+1;
   %feedforward
   o=feval('funact',W0*X,1);
   o=[o;ones(1,K)];    % entrada 1 para las bias de esta capa
```

```
   A=o*o';
   b=o*Y';

   V0=defiposi(A,b); %cálculo de los pesos óptimos

   V0=V0';   %ojo con lo que son las filas y lo que son las columnas

   %feedforward
   y=V0*o;
   y=funact(y,1);%función de act. para reducir el valor del error

  %matriz de errores a la salida
   e=Y-y;          %la salida para estos pares ha de ser 1

   E=(1/K)*sum(0.5*(e.^2));
   %capa oculta
   Sold=W0*X;
   aux=feval('funactp',W0*X,1);
   Vlin=[];
   for h=1:H
      Vlin=[Vlin;aux(h,:)*V0(h)];
   end

   aux=Vlin*X';
   aux=aux';
   aux=aux(:);
   A2=aux*aux';

   aux=[];
   for k=1:K
      aux=[aux (e(k)*Vlin(:,k))];
   end
   b2=aux*X';
   b2=b2';
   b2=b2(:);

   aux=feval('funactpp',W0*X,1);

   C0=[];
   for h=1:H
      for m=1:M+1
         for j=1:M+1
C0((h-
1)*(M+1)+m,j)=abs(V0(h))*sum(abs(aux(h,:)).*X(m,:).*X(j,:));
         end
      end
   end

   Etest=E+1;
   IC2=0;
   auxtraza_mu=[];
   Etest0=0;
   while (Etest>=E)& ((Etest<Etest0)|(IC2==0))% el error de test >=
                                              %que el E inicial
      IC2=IC2+1;     % y el error de test <= error de test anterior
      Etest0=Etest; %se guarda el error de test anterior
      C=(mu/H)*C0;
      aux=zeros((M+1)*H,(M+1)*H);
      for h=0:H-1
```

```
         aux(((M+1)*h+1):((M+1)*(h+1)),((M+1)*h+1):((M+1)*(h+1)))=
          aux(((M+1)*h+1):((M+1)*(h+1)),((M+1)*h+1):((M+1)*(h+1)))
          +C(((M+1)*h+1):((M+1)*(h+1)),:);
      end

      LA2=A2+aux;

      DW=defiposi(LA2,b2);  %cálculo de los incrementos óptimos
      DWb=[];
      for h=0:H-1
         DWb=[DWb DW(h*(M+1)+1:(h+1)*(M+1),:)];
      end
      DWb=DWb';

      Wtest=W0+DWb; %act. de los pesos de test de la capa oculta

      otest=feval('funact',Wtest*X,1);
      otest=[otest;ones(1,K)];  % entrada 1 para las bias de esta capa
      ytest=V0*otest;

      ytest=funact(ytest,1);  %función de activación para dismi. error

      etest=Y-ytest;       %predicción de error

      Etest=0.5*(1/K)*sum(etest.^2);

      if Etest>= E
         mu=mu*gamma; %incrementamos la influencia del termino de pena.
      end
      auxtraza_mu=[auxtraza_mu Etest];
      %pause;
   end
   if Etest<E
      W0=Wtest; %actualizamos los pesos
      E=Etest; % actualizamos el error
   end
   mu=mu*beta; %decrementamos el término de penalización
   traza_error=[traza_error E];
end

t=cputime-t0       %tiempo de duración del entrenamiento
```

FUNACT.M

```
function [y]=funact(x,g);  %función de activación

y=1./(1+exp(-g*x));
```

FUNACTP.M

```
function [x]=funactp(x,g);  %1ª derivada de la función de activación

x=g*funact(x,g).*(1-funact(x,g));
```

FUNACTPP.M

```
function [x]=funactpp(x,g);  %2ª derivada de la función de activación

x=g*funact(x,g).*(1-funact(x,g)).*(1-2*funact(x,g));
```

También se añaden las rutinas de test que se emplearon para la obtención de las tablas. La rutina *test* es con los pares de test y la rutina *teste* es con los pares de entrenamiento. *tasa_er* calcula la tasa de error que se ha enunciado.

TEST.M

```
%este es un test para comprobar la bondad del entrenamiento mediante
%el empleo de las entradas de test y la tasa de error
%Parto de que se conocen las matrices de pesos W y V
function [p]=test(n,l,W,V);
%n es el dígito y l el locutor
  load ([num2str(l) '\logpot32']);   %cargar los pares de test
   N=norma(logPO);  % normalización de las entradas
   for i=1:1000
      if mod(i,10)==mod(n+1,10)
         j(i)=1;            %salida correcta para el reconocimiento
                            %del dígito correspondiente
      else
         j(i)=0;
      end
   end
p=tasa_er(N,j,W,V);
```

TESTE.M

```
%este es un test para comprobar la bondad del entrenamiento mediante
%el empleo de las entradas de entrenamiento y la tasa de error
%Parto de que se conocen las matrices de pesos W y V
function [p]=teste(n,l,W,V);
%n es el dígito y l el locutor
  load ([num2str(l) '\logpoe32']);     %cargar los pares de entrena.
   N=norma(logPO);      %normalización de las entradas
   for i=1:1000
      if mod(i,10)==mod(n+1,10)
         j(i)=1;            %salida correcta para el reconocimiento
                            %del dígito correspondiente
      else
         j(i)=0;
      end
   end
p=tasa_er(N,j,W,V);
```

TASA_ER.M

```
function [p]=tasa_er(X,Y,W,V);
%Función que devuelve el porcentaje de errores que se dan en la red ya
%entrenada usando una red oll.
```

```
l=size(X);     %dimensiones

X=[X;ones(1,l(2))];

o=feval('funact',W*X,1);
o=[o;ones(1,l(2))];     % entrada 1 para las bias de esta capa
y=V*o;
[y]=regenera(y);

contador=0;
for i=1:l(2)
    if y(i)==Y(i)
         contador=contador+1;
    end
end

p=contador*100/l(2);  %cálculo en porcentaje
```

ANEXO IV : ALGORITMO PARA LA NORMALIZACIÓN DE LAS ENTRADAS A LAS REDES NEURONALES A ENTRENAR.

El algoritmo utilizado para normalizar las entradas de probabilidad para las NN's es el que se muestra a continuación.

NORMA.M

```
function [V]=norma(v);
%Esta es una función que se encarga de devolvernos la matriz v que
%se mueve en un rango de valores [a,b] desplazado y escalado
%al rango [0,1]


V=v-min(min(v));%esto hace que todos los valores de v sean ahora posi.

V=V/max(max(V));   %esto hace que el mayor de los valores sea 1
```

ANEXO V : ALGORITMO PARA LA DISCRIMINACIÓN DEL UMBRAL A LA SALIDA DE LAS REDES NEURONALES A ENTRENAR.

REGENERA.M

```
function [y]=regenera(x);
m=size(x);  %dimensiones
umbral=(max(max(x))+min(min(x)))/2;  %se escoge el valor medio
for i=1:m(1)                         %entre el mínimo y el máximo
for j=1:m(2)                         %como umbral
      if x(i,j)>umbral y(i,j)=1;
      else y(i,j)=0;
      end
end
end
```

PRESUPUESTO

En lo que a presupuesto se refiere, en este proyecto fin de carrera se puede hacer una subdivisión en apartados correspondientes a tres fases bien diferenciadas en su realización.

Fase 1ª: Una primera parte de estudio genérico de Redes Neuronales en la que se tomó contacto con los conceptos básicos en esta tecnología, se practicó con las redes más elementales analizando métodos de entrenamiento, formas de ajuste e incremento de la precisión, etcétera.

Fase 2ª: La segunda parte se centró en el análisis del nuevo algoritmo propuesto por Ergezinger y Thomsen, la implementación en pequeñas pruebas para su estudio y para determinar su problemática.

Fase 3ª: Por último, una tercera fase en la que se efectuó la prueba de este nuevo algoritmo en un problema serio, la obtención de resultados con este y con el de Backpropagation y el análisis comparativo entre ambos para tratar de determinar que ventajas trae consigo el uso del OLL.

Para elaborar el presupuesto se ha estudiado de manera aproximada el número medio de horas diarias dedicadas a cada tarea así como el número de días empleados. Se indica de manera orientativa los meses durante los cuales se llevó a cabo la fase correspondiente.

Fase	Meses	Días	Horas/día	Ptas/hora	Subtotal
Fase 1ª	Enero - Abril	120	2.5	3100	930000
Fase 2ª	Mayo - Junio	46	3.5	3100	499100
Fase 3ª	Junio - Agosto	77	5	3100	1193500

Total
2622600

BIBLIOGRAFÍA

[1] James A. Freeman y David M. Skapura, *Redes Neuronales. Algoritmos, aplicaciones y técnicas de programación..*
Ed. Addison Wesley Iberoamericana, 1993.

[2] Simon Haykin, *Neural Networks. A comprehensive foundation.*
Ed. Macmillan College Publishing Company, 1994.

[3] Félix García Merayo y Antonio Nevot Luna, *Análisis Numérico.*
Ed. Paraninfo, 1990.

[4] Burden y Faires, *.Numerical Analysis.*
Ed. International Thomson Publishing (ITP), 1994.

[5] S.Ergezinger and E. Thomsen, "An accelerated Learning Algorithm for Multilayer Perceptrons: Optimization Layer by Layer", *IEEE transactions on Neural Networks,* vol 6, no 1, January 1995 pp. 31-42.

[6] Christopher M. Bishop, *Neural Networks for pattern recognition.*
Ed. Clarendon press, Oxford, 1997.

[7] Nils J. Nilsson, *The mathematical foundations of learning machines.*
Ed. Morgan Kaufmann, San Mateo, California, 1990.

[8] Stephen I. Gallant, *Neural network learning and expert system.*
Ed. Mitt Press, Cambridge, 1993.

[9] Clifford Lan, *Neural networks. Theoretical foundation and analisys.*
Ed. IEEE, New York, 1992.