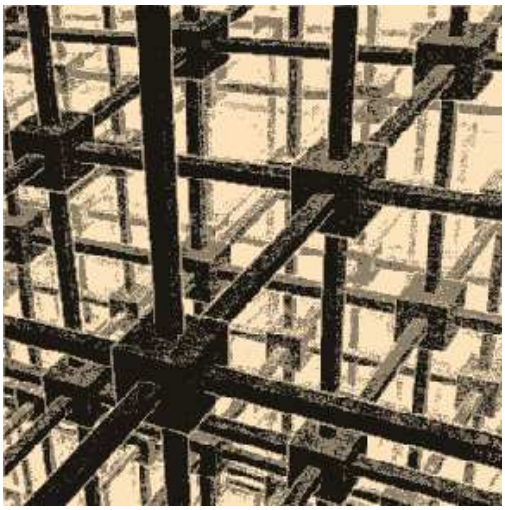
Sistemas Basados en el Conocimiento II: Introducción a la Neurocomputación



Texto base de la asignatura del mismo nombre perteneciente al Plan de Estudios de la Universidad Nacional de Educación a Distancia.

Equipo Docente de la Asignatura

Universidad Nacional de Educación a Distancia
2004

Índice general

Capítulo 1

Introducción

Objetivo: En este primer capítulo se va a tratar de dar una perspectiva histórica del desarrollo de las redes neuronales como modelo de computación. Al hilo de este repaso histórico, introduciremos los conceptos fundamentales de un modelo general de neurocomputación. En capítulos siguientes veremos cómo las distintas arquitecturas y algoritmos de aprendizaje se pueden ver como casos particulares del modelo general.

1.1. Introducción

Sin que sea éste el propósito principal de este texto, sí es conveniente contextualizar el campo de la Neurocomputación dentro del marco más general de la Inteligencia Artificial ayudándonos de algunos conceptos clave de la *Ingeniería del Conocimiento*. Desde esta perspectiva, la neurocomputación es una forma, inspirada en la biología, de resolver algunas tareas encontradas frecuentemente cuando se trata de reproducir mediante computadoras habilidades atribuidas a la inteligencia humana. Al analizar estas habilidades nos encontramos que las más complejas se pueden descomponer en tareas y subtareas hasta llegar a inferencias primitivas que añaden información a la disponible inicialmente sin que podamos subdividir el proceso en pasos más simples. A las distintas formas de implementar una tarea se las conoce como Métodos de Resolución de Problemas (o *Problem Solving Methods* en inglés). Estos métodos en la práctica combinan módulos que pertenen a muchas categorías distintas de entre las que podemos señalar los sistemas basados en reglas, en lógica, en redes o en marcos/objetos. Para cada módulo se emplea el paradigma que mejor se ajuste a las características de la subtarea o inferencia que se desea implementar. Aquí vamos a profundizar en los métodos conexionistas de resolver tareas que se caracterizan por su naturaleza distribuida, de grano fino y autoprogramable. Estos métodos nacieron de la apuesta de los pioneros de la Inteligencia Artificial por intentar resolver los problemas a los que se enfrenta esta disciplina con la misma metodología de la propia Naturaleza. En otras palabras, buscaron en la Biología inspiración para resolver con computadoras tareas propiamente humanas. Esto implica necesariamente recurrir a la Neurobiología para tratar de entender la forma en que nuestros cerebros realizan esta tareas. Aunque de forma superficial, dedicaremos lo que resta de esta sección introductoria a resumir lo que sabemos hoy en día de la forma como funcionan las neuronas biológicas.

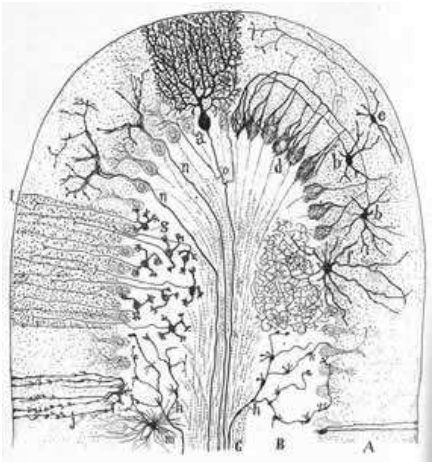


Figura 1.1: Esquema-dibujo de D. Santiago Ramón y Cajal de las distintas tipologías neuronales del cerebelo.

Aunque hay muchos tipos diferentes de neuronas en el sistema nervioso central (SNC, ver figura 1.1) su estructura funcional es la misma: un cuerpo celular denominado soma, un conjunto de ramificaciones que parten del soma denominadas dendritas y una eferencia de mayor longitud y menos ramificada que las dendritas denominada axón (ver figura 1.2).

Las dendritas funcionan a modo de antenas receptoras de estímulos procedentes de otras neuronas y dependiendo del número de estímulos recibidos en las dendritas y trasmitidos al soma, éste a su vez puede generar una señal de activación a lo largo del axón que termine excitando a otras neuronas. Por lo tanto, podemos considerar que el soma, respecto de las entradas procedentes de otras neuronas, desempeña una función de toma de decisión: si el nivel de excitación de dichas neuronas sobrepasa un cierto umbral el soma se dispara y envía a su vez una señal excitadora a lo largo del axón. Al lugar donde se produce la transmisión de información entre el axón de una neurona y la dendrita de otra se lo conoce como sinapsis y, dado que la información siempre se transmite en un sentido (del axón de una neurona hacia la dendrita de otra[a]) vamos a llamar neurona presináptica a aquélla y postsináptica a ésta. La naturaleza exacta de los términos empleados anteriormente -como estímulo, activación o excitación- está relacionada con la fisiología de la neurona y, muy particularmente, de su membrana; con la presencia de numerosos canales iónicos que pueden abrirse y cerrarse como respuesta a la presencia de determinadas sustancias químicas (los neurotransmisores) dando lugar a una permeabilidad eléctrica variable y a cambios en la diferencia de potencial entre el interior y el exterior de la célula. Las señales a las que hacemos referencia están codificadas en el potencial intracelular y en su evolución temporal. Sin embargo, y como ocurrirá a menudo a lo largo de los próximos capítulos, no podemos perseguir hasta sus últimas consecuencias este cuadro general y recomendamos al lector interesado la lectura de cualquiera de los númerosos textos introductorios al

[a]Desgraciadamente no podemos matizar adecuadamente esta afirmación por escaparse del objetivo global de este escrito. Baste decir aquí que investigaciones recientes en el campo de la Neurobiología apuntan a la existencia de mecanismos químicos que permiten a la neurona presináptica recibir información sobre si su estado de activación contribuyó a disparar un potencial de acción en la neurona postsináptica.

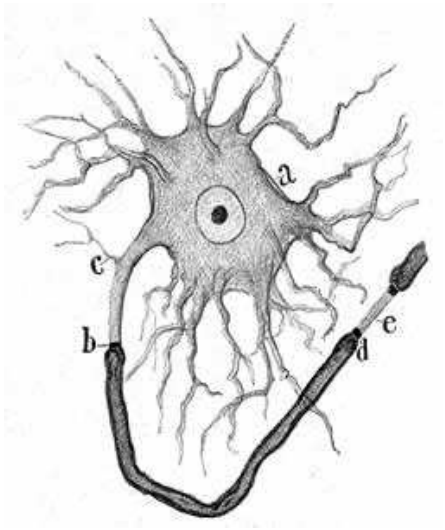


Figura 1.2: Esquema-dibujo de D. Santiago Ramón y Cajal de las partes que componen una neurona tipo.

campo (por ejemplo, [BCP04]).

Como decíamos, esta descripción es necesariamente simplificadora. Sin embargo, nos hemos dejado fuera de la descripción un elemento fundamental que debemos recoger ahora y es el hecho de que las señales procedentes de las neuronas pueden ser excitadoras o inhibidoras y, de igual modo, la señal enviada a lo largo del axón de una neurona, puede servir para excitar o inhibir la excitación de la neurona postsináptica. Además, la capacidad excitatoria/inhibitoria de una sinapsis puede variar con el tiempo y potenciarse o disminuir.

En resumen, tenemos que desde un punto de vista computacional la neurona funciona como la suma de un campo receptivo dendrítico, una función de decisión no lineal (el soma) y una línea de transmisión de la respuesta (axón). La realidad es infinitamente más compleja y rica que lo que se ha descrito aquí. Pero, como punto de partida para construir modelos de neurona artificial tendrá que servir.

1.2. Modelos de neurona

Históricamente, el primer modelo de neurona artificial, basado en los conocimientos de neurofisiología disponibles durante la primera mitad del siglo XX, es el de la neurona formal de Warren McCulloch y Walter Pitts [MP43]. Dicho modelo, adaptado a la notación y nomenclatura unificada que se empleará en el resto del libro, se muestra de forma esquemática en la figura 1.3.

Dicho esquema representa una neurona que recibe N_e entradas. Las conexiones aparecen en la figura como flechas que parten de la zona izquierda de la figura y que llevan asociado un peso ω_i donde el subíndice i se refiere a la entrada y toma valores entre 1 y N_e. McCulloch y Pitts, inspirados en el comportamiento de las neuronas biológicas,

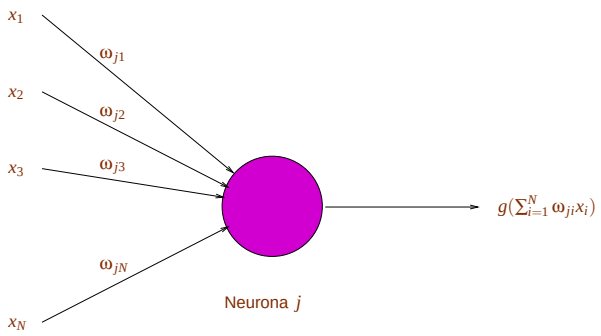


Figura 1.3: Representación esquemática de un elemento de computación neuronal.

definieron el nivel de activación de la neurona como

$$y_j = g(a) = g(\sum_{i=1}^{N_e} \omega_i x_i) \tag{1.1}$$

Funciones de activación

donde a es la suma de las entradas x_i ponderadas por los pesos de las conexiones ω_i y g es una función (llamada función de activación) que trasforma dicha suma en la activación final de la neurona, de forma que si a es inferior a un cierto umbral b la neurona no se activa ($y = 0$), y si lo supera la neurona tendrá nivel de activación y igual a uno. Expresado de forma matemática

$$g(a) = \theta(a-b) = \begin{cases} 1 & \text{si} \quad a \geq b \\ 0 & \text{si} \quad a < b \end{cases} \tag{1.2}$$

donde b es el umbral que debe sobrepasar la suma ponderada para activar la neurona. Dicho umbral recibe también el nombre de polarización o *bias* en inglés. A la función g tal y como la hemos definido en la ecuación 1.2 se la conoce como función paso o función umbral (*threshold* en inglés) y su representación gráfica se muestra en la figura 1.4.

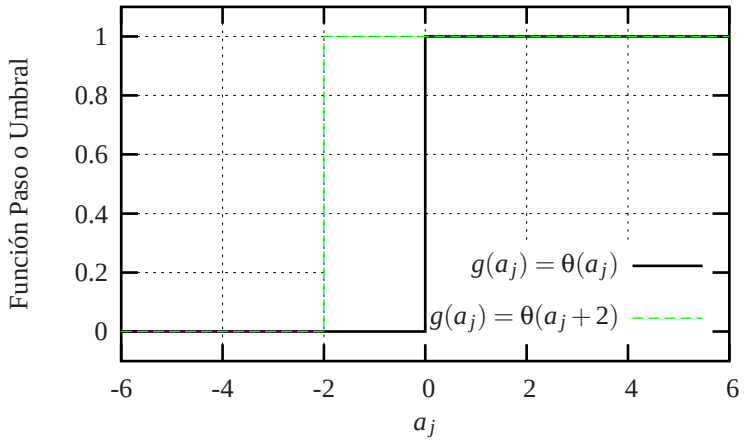


Figura 1.4: Representación cartesiana de la función paso para dos valores de la polarización $b = 0$ y $b = -2$.

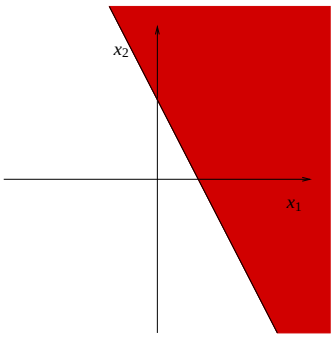


Figura 1.5: Representación en forma de sombreado rojo de los puntos x_1, x_2 del plano que activan la neurona del ejemplo.

donde, de hecho, se muestran dos ejemplos de la función paso con valores diferentes del umbral o polarización b. El valor de g para el caso $a < b$ se podría haber definido como -1 y las propiedades formales de la neurona serían idénticas. Más adelante veremos un ejemplo en el que se emplea esta elección de la función umbral. Ésta no es la única definición posible de la función de activación g, pero antes de estudiar las alternativas posibles a la elección de McCulloch y Pitts conviene deternos a analizar su significado. Para ello vamos a estudiar un ejemplo sencillo en el que una neurona formal de McCulloch y Pitts posee únicamente dos entradas. Supongamos que el peso de la primera conexión vale 0,5 y el segundo 0,25, y la polarización de la neurona vale 0,1. En ese caso, podemos preguntarnos qué posibles combinaciones de la entrada hacen que se active la neurona. Un cálculo sencillo nos lleva a la condición

$$\sum_{i=1}^{2} \omega_i \cdot x_i = 0{,}5 \cdot x_1 + 0{,}25 \cdot x_2 \geq 0{,}1 \tag{1.3}$$

que se cumple exactamente para puntos del semiplano por encima de la recta

$$0{,}5 \cdot x_1 + 0{,}25 \cdot x_2 = 0{,}1, \tag{1.4}$$

que aparece sombreado en la figura 1.5.

Funciones discriminantes

De esta forma, vemos que una neurona formal de McCulloch y Pitts actúa como una función discriminante que tiene valor 1 en la región del plano sombreada y 0 en el semiplano restante. En otras palabras, la neurona es capaz de distinguir puntos de una región de los de otra y clasificarlos según la activación resultante. Veremos más adelante las implicaciones de este sencillo hallazgo cuando tratemos conjuntos de neuronas agrupados en capas y de las tareas genéricas para las que el empleo de redes neuronales es apropiado. Por ahora baste señalar el papel desempeñado por la polarización o *bias* en el cálculo de la neurona, que no es otro que permitir que la recta que divide las dos zonas de activación no tenga que pasar por el origen. Efectivamente, si la misma neurona hubiese tenido polarización 0 la ecuación discriminante hubiera sido $0{,}5 \cdot x_1 + 0{,}25 \cdot x_2 = 0$ que, como se puede comprobar pasa por el origen de coordenadas $(0,0)$.

Finalmente, antes de pasar a otros modelos de neurona, vamos a introducir un pequeño cambio en la notación que nos permitirá simplificar algunas fórmulas. En lugar de incluir explícitamente la polarización dentro de la definición de g, vamos a tratarla

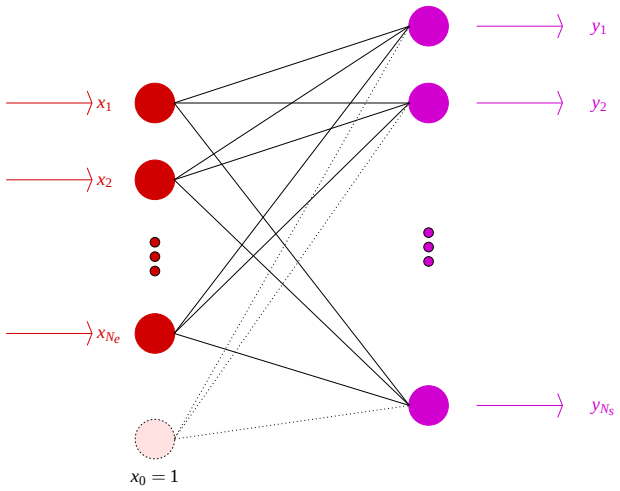


Figura 1.6: Modelo esquemático de la red neuronal denominada perceptrón.

como si fuese el peso de una entrada adicional (con subíndice 0) de valor igual a 1. De esta forma, $\omega_0 = b$, $x_0 = 1$ y las ecuaciones 1.1 y 1.2 se transforman en

$$y_j = g(a) = g(\sum_{i=0}^{N_e} \omega_i x_i) = g(\omega_0 \cdot 1 + \sum_{i=1}^{N_e} \omega_i x_i) \tag{1.5}$$

$$g(a) = \theta(a) = \begin{cases} 1 & \text{si} \quad a \geq 0 \\ 0 & \text{si} \quad a < 0 \end{cases} \tag{1.6}$$

Podéis comprobar que la función discriminante sigue siendo la misma porque la neurona se activará en aquellas regiones donde a sea mayor que cero, y éstas vienen dadas por la nueva definición de a que ahora incluye el término adicional de subíndice 0.

Una vez incluida la polarización en el conjunto de pesos de la neurona podemos expresar la ecuación 1.5 en notación vectorial

$$y = g(a) = g(\sum_{i=0}^{N_e} \omega_i x_i) = g(\vec{\omega}^T \cdot \vec{x}) \tag{1.7}$$

donde $\vec{\omega}^T$ es el vector de pesos de la neurona traspuesto y $\vec{x}$ es el vector de entradas $(1, x_1, x_2, ..., x_{N_e})$.

El perceptrón

Siguiendo el desarrollo histórico del campo de la Neurocomputación nos encontramos con la extensión del modelo formal de McCulloch y Pitts denominada *perceptrón* . Un perceptrón, tal como lo presenta Rosenblatt (1962) [Ros62] consiste en un conjunto de neuronas formales de McCulloch y Pitts que comparten las mismas entradas y que no están conectadas entre sí (ver figura 1.6).

En dicha figura hemos introducido nuevos códigos y convenciones que seguiremos empleando a lo largo de todo el libro y que es conveniente explicar. En primer lugar, hemos añadido unos objetos especiales a la izquierda del grafo. Hemos empleado

círculos para representar dichos objetos porque en muchos textos del campo y en los neurosimuladores de los que hablaremos más adelante, a dichos objetos se les llama neuronas de entrada (y a la estructura que forman, capa de entrada). En realidad, no se trata de neuronas puesto que no realizan ningún cómputo sobre la entrada que cada una de ellas recibe y se limitan a trasmitir su valor a las neuronas de la capa siguiente. A pesar de ello, esta nomenclatura está tan extendida que se va a mantener en este libro y por ello las hemos representado con círculos. En la figura 1.6 podemos encontrar dos grupos de neuronas: al primer grupo ya hemos hecho alusión, es la capa de neuronas de entrada. El segundo constituye la capa de neuronas de salida y está compuesto de N_s neuronas formales de McCulloch-Pitts. Finalmente, hemos añadido una entrada $x_0 = 1$ asociada al valor de la polarización.

Vamos a utilizar una red sencilla como el perceptrón para introducir los conceptos más importantes en el campo de las redes neuronales. A partir de este bloque básico iremos, a lo largo del libro, exponiendo distintas variantes y modificaciones que darán lugar a los diferentes paradigmas de computación neuronal.

El perceptrón se introduce como clasificador de vectores. Ya hemos visto cómo cada neurona formal de McCulloch-Pitts con dos entradas es un clasificador capaz de agrupar las entradas en dos regiones separadas por una recta. Una neurona de McCulloch-Pitts con N_e entradas divide el espacio de entrada en dos regiones separadas por un hiperplano en N_e dimensiones de forma que los vectores de entrada $\vec{x}$ que cumplen $(\vec{\omega}_j)^T \cdot \vec{x} > 0$ activan la neurona. La novedad del perceptrón no radica en el modelo neuronal que, en lo fundamental, corresponde a la neurona formal de McCulloch-Pitts, sino en el algoritmo de ajuste de los pesos. Por simplicidad y sin falta de generalidad puesto que todas las neuronas de la capa de salida son iguales, vamos a estudiarlo para el caso de una única neurona.

Dado un conjunto de vectores $\vec{x}[n]$ de dimensión N_e con $n = 1, 2, .., d$, cuya pertenencia a una de dos clases $\mathcal{C}_1$ o $\mathcal{C}_2$ es conocida de antemano, deseamos hallar el conjunto de pesos tales que, para todos los vectores $\vec{x}[n]$ pertenecientes a la clase $\mathcal{C}_1$ la neurona presente activación 1 y, al contrario, para todos los de clase $\mathcal{C}_2$, activación -1 (a este conjunto de d vectores cuya clase es conocida se lo conoce como conjunto de entrenamiento). Al valor deseado de la salida de la neurona para el n-ésimo vector $\vec{x}[n]$ vamos a llamarlo $t[n]$ y será igual a 1 para vectores de la clase $\mathcal{C}_1$ e igual a -1 para vectores de la clase $\mathcal{C}_2$ [b]. Lo que vamos a exponer a continuación es el proceso de cálculo de los parámetros de la red (pesos) realizado con el objetivo de que la red proporcione una salida determinada (los valores $t[n]$). Ésto es lo que se conoce como *entrenamiento supervisado* de la red. En general, se llama entrenamiento al proceso de cálculo de los párametros de la red. Más específicamente, se emplea el término entrenamiento supervisado cuando dicho cálculo se realiza a partir de un conjunto de pares entrada/salida de la red prefijados de antemano. En este texto, dichos pares de entrenamiento se representarán como $\vec{x}[n]/\vec{t}[n]$ donde $\vec{x}[n]$ representa el vector de entrada y $\vec{t}[n]$ el vector de salida que queremos que se produzca al presentar $\vec{x}[n]$ a la entrada de la red (si sólo tenemos una neurona de entrada y una de salida en lugar de vectores tendremos escalares $x[n]/t[n]$). En resumen, los pesos de la red se van modificando para obtener la salida (los vectores $\vec{t}[n]$) deseada. Existe también una modalidad de entrenamiento denominada *entrenamiento no supervisado* que consiste calcular los pesos de la red a

Conjunto de entrenamiento

Entrenamiento supervisado

Entrenamiento no supervisado

[b] Nótese que aquí hemos modificado ligeramente la definición de la función paso que pasa de valer 0 cuando $(\vec{\omega})^T \cdot \vec{x}$ es menor que 0 a valer -1. Este cambio facilita la exposición del algoritmo de entrenamiento sin modificar las propiedades formales de la neurona.

partir del conjunto de vectores de entrada sin forzar de ninguna manera la salida de la red. De ella hablaremos más adelante en el capítulo 3.

Para hallar el conjunto de valores de los parámetros de un perceptrón, vamos a adoptar la estrategia de definir una función que mida el error con que clasifica la red para un conjunto Ω de pesos y, a continuación, vamos a calcular qué modificaciones en dichos parámetros harían disminuir el error de clasificación.

En una clasificación perfecta, los valores de los pesos ω serían tales que tendríamos que los vectores $\vec{x}[n]$ del conjunto de entrenamiento que pertenecen a la clase $\mathcal{C}_1$ cumplen $(\vec{\omega})^T \cdot \vec{x}[n] > 0$ y por lo tanto, $y[n] = t[n] = 1$. Por otra parte, los de clase $\mathcal{C}_2$ cumplirían que $(\vec{\omega})^T \cdot \vec{x}[n] < 0$ y, por lo tanto, $y[n] = t[n] = -1$.

Todo lo anterior implica que conseguir una clasificación perfecta, como objetivo, es equivalente a disminuir el valor de la función

$$E(\Omega) = -\sum_{\mathcal{M}} (\vec{\omega})^T \cdot \vec{x}[n] \cdot t[n] \tag{1.8}$$

donde $\mathcal{M}$ es el conjunto de vectores de entrada mal clasificados dado el conjunto de pesos Ω. La afirmación anterior se puede ver de forma sencilla si consideramos que un vector de entrada $x[n]$ puede ser clasificado incorrectamente porque,

1. o bien es de clase $\mathcal{C}_1$ pero $\vec{\omega} \cdot \vec{x}[n] < 0$ en cuyo caso habría que aumentar el valor de $\vec{\omega} \cdot \vec{x}[n]$ o, lo que es lo mismo puesto que $t[n] = 1$, disminuir el de $-\vec{\omega} \cdot \vec{x}[n] \cdot t[n]$

2. o bien es de clase $\mathcal{C}_2$ pero $\vec{\omega} \cdot \vec{x}[n] > 0$ en cuyo caso habría que disminuir el valor de $\vec{\omega} \cdot \vec{x}[n]$ o, lo que es lo mismo puesto que $t[n] = -1$, el de $-\vec{\omega} \cdot \vec{x}[n] \cdot t[n]$

Como vemos, introducir el factor multiplicativo $t[n]$ resulta en que para cada vector mal clasificado por la red, el objetivo siempre consiste en disminuir $-\vec{\omega} \cdot \vec{x}[n] \cdot t[n]$.

Resumiendo lo expuesto hasta ahora, tenemos que un perceptrón es un conjunto de neuronas formales de McCulloch-Pitts cuyas conexiones deben ser ajustadas (mediante los pesos) de forma que minimicen el valor de una función $E(\Omega)$ llamada error. En esto consiste el entrenamiento de cualquier red neuronal: las distintas variaciones sobre el tema central del perceptrón consistirán en elegir diferentes arquitecturas (ver sección 1.3), funciones de activación, definiciones del error o algoritmos de minimización de dicho función del error. A lo largo del temario del libro iremos analizando diferentes elecciones posibles y su significado.

Ya hemos justificado intuitivamente la función de error del perceptrón:

$$E(\Omega) = -\sum_{\mathcal{M}} (\vec{\omega})^T \cdot \vec{x}[n] \cdot t[n] \tag{1.9}$$

Resta ahora definir un algoritmo de minimización de dicha función. La opción más natural es la que sirvió históricamente para la definición del perceptrón. Para comprender esta definición es necesario entender el concepto de gradiente. No nos vamos a ocupar aquí de justificar que el gradiente de una función en un punto marca la dirección de máximo crecimiento de dicha función, pero dicha justificación se puede encontrar en cualquier texto de cálculo elemental. Dándolo por supuesto, entonces el gradiente cambiado de signo apuntará en la dirección de máxima disminución que es

precisamente lo que estamos buscando: disminuir la función error. Por ello, el algoritmo de aprendizaje del perceptrón (y de otros muchos tipos de redes) se basa en la fórmula

$$\Delta\vec{\Omega} = -\frac{\partial E(\omega)}{\partial\vec{\omega}} \tag{1.10}$$

donde $\Delta\vec{\omega}$ es la cantidad que habrá que añadir al vector de pesos para minimizar la función de error. La ecuación 1.10 representa un grupo de métodos conocidos habitualmente como de descenso del gradiente.

Descenso del gradiente

Si sustituimos la ecuación 1.9 en 1.10 obtendremos que

$$\Delta\vec{\omega} = \sum_{\mathcal{M}} \vec{x}[n] \cdot t[n] \tag{1.11}$$

fórmula equivalente a

$$\Delta\vec{\omega} = \sum_{n=1}^{d} \vec{x}[n] \cdot \frac{(t[n] - y[n])}{2} \tag{1.12}$$

en la que hemos incluido los vectores correctamente clasificados que no contribuirán a $\Delta\vec{\omega}$ puesto que para ellos $t[n] - y[n] = 0$.

Considérese ahora la situación descrita en la figura 1.7: en ella vemos cómo aplicando el método de descenso del gradiente podemos pasar de largo el mínimo en la función de error si el gradiente es muy pronunciado. Para evitar este tipo de situaciones se suele aplicar sólamente una fracción del incremento de pesos $\Delta\vec{\omega}$ dado por la ecuación 1.12. A dicha fracción se la conoce como velocidad de aprendizaje y en este texto se representará por la letra griega α. Una fracción del 10 % del valor dado por la ecuación 1.12 corresponde a $\alpha = 0{,}1$. Como vemos, su función es ralentizar el aprendizaje para evitar actualizaciones de los pesos demasiado grandes que pudieran desestabilizar el proceso, lo que podría ocurrir si la función de error tuviera varios mínimos locales (caso no representado en la figura).

velocidad de aprendizaje

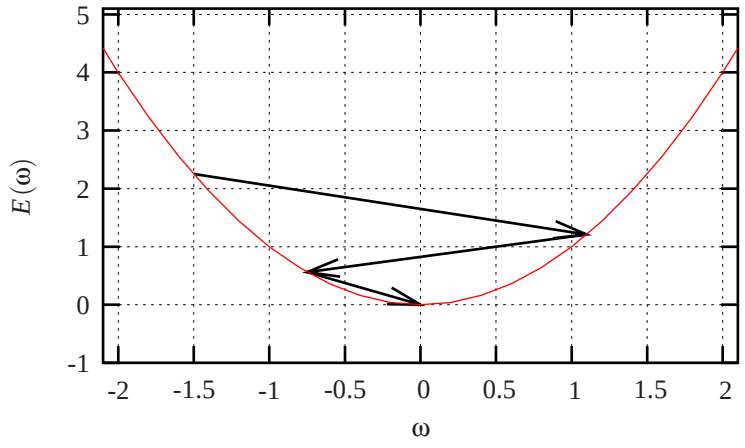


Figura 1.7: Ejemplo de aprendizaje con un único mínimo local. El proceso de aprendizaje converge finalmente al valor del error mínimo tras una serie de oscilaciones. La introducción del factor α (velocidad de aprendizaje) ayuda a evitar estas oscilaciones y otras posibles dificultades asociadas a la existencia de varios mínimos locales.

En ese caso, al introducir la velocidad de aprendizaje la ecuación de actualización de

pesos se transforma en

$$\Delta\vec{\omega} = \alpha \cdot \sum_{n=1}^{d} \vec{x}[n] \cdot \frac{(t[n] - y[n])}{2} \tag{1.13}$$

o, si la función de activación hubiese tomado valores 0 para los vectores de clase $\mathcal{C}_2$ la regla de actualización en

$$\Delta\vec{\omega} = \alpha \cdot \sum_{n=1}^{d} \vec{x}[n] \cdot (t[n] - y[n]). \tag{1.14}$$

Ambas ecuaciones proporcionan el incremento de los pesos que hace disminuir la función error de acuerdo con el método del descenso del gradiente *una vez presentado un ciclo completo de pares de entrenamiento (de ahí el sumatorio en n)*. Ésto se conoce como entrenamiento por lotes (*batch learning* en inglés) porque no se modifican los pesos hasta haber presentado el lote completo de pares de entrenamiento disponibles. Una alternativa posible es hacer la modificación correspondiente a cada par inmediatamente después de ser calculada. Ésto se conoce como entrenamiento secuencial (*sequential* o *pattern-based learning* en inglés). En ese caso, el algoritmo de actualización de pesos sería

Entrenamiento por lotes

Entrenamiento secuencial

1. Inicializar el vector de pesos con componentes aleatorias.
2. Inicializar *n* con el valor 1.
3. Introducir el vector $\vec{x}[n]$ en la capa de entrada y calcular la activación de la neurona.
4. Calcular las modificaciones al vector de pesos $\vec{\omega}$ de acuerdo con la ecuación 1.13 o 1.14 según sea la función de activación.
5. Si *n* es menor que *d* aumentar n una unidad y retornar al paso anterior. Si *n* es igual a *d*, modificar los pesos según la suma de los incrementos calculados en el paso anterior y retornar al paso 2.

y finaliza cuando el perceptrón clasifica correctamente todos los pares de entrenamiento o cuando el error de clasificación (dado por la ecuación 1.9) disminuye por debajo de un umbral aceptable marcado de antemano.

Error de la red

Ejemplo 1 (Perceptrón): Calculad los parámetros de la red más sencilla posible que clasifica correctamente los siguientes puntos del plano en las dos clases definidas por $t[n] = 0 \Rightarrow \mathcal{C}_1$ y $t[n] = 1 \Rightarrow \mathcal{C}_2$. Supóngase que la velocidad de aprendizaje α es constante e igual a 0,1.

n	(x_1,x_2)	$t[n]$
1	(-0.5, -0.5)	1
2	(-0.5, 0.5)	1
3	(0.3, -0.5)	0
4	(0.0, 1.0)	0

SOLUCIÓN 1: Inicializamos los pesos aleatoriamente en el intervalo $[-1,+1]$ de forma que obtenemos

ω_0	ω_1	ω_2
0.3577	-0.5621	-0.9059

donde hemos representado la polarización como un nuevo peso ω_0 asociado a una entrada $x_0 = 1$. Comenzamos el primer ciclo de aprendizaje cuyos resultados se resumen a continuación:

n	a	$y[n]$	$t[n]-y[n]$	$\Delta\omega_0$	$\Delta\omega_1$	$\Delta\omega_2$
1	1.0917	1	0	0	0	0
2	0.1858	1	0	0	0	0
3	0.6420	1	-1	-0.1	-0.03	0.05
4	-0.5482	0	0	0	0	0

Aplicamos las correcciones a los pesos y obtenemos

ω_0 (polarización)	ω_1	ω_2
0.2577	-0.5921	-0.8559

Dado que el resultado no clasifica correctamente los pares de entrenamiento realizamos un segundo ciclo de entrenamiento:

n	a	$y[n]$	$t[n]-y[n]$	$\Delta\omega_0$	$\Delta\omega_1$	$\Delta\omega_2$
1	0.9817	1	0	0	0	0
2	0.1258	1	0	0	0	0
3	0.5080	1	-1	-0.1	-0.03	0.05
4	-0.5982	0	0	0	0	0

Aplicamos las correcciones correspondientes a este segundo ciclo y obtenemos

ω_0 (polarización)	ω_1	ω_2
0.1577	-0.6221	-0.8059

Para el tercer ciclo tendremos

n	a	$y[n]$	$t[n]-y[n]$	$\Delta\omega_0$	$\Delta\omega_1$	$\Delta\omega_2$
1	0.8717	1	0	0	0	0
2	0.0658	1	0	0	0	0
3	0.3740	1	-1	-0.1	-0.03	0.05
4	-0.6482	0	0	0	0	0

No tiene sentido continuar reproduciendo los cálculos detallados del proceso de aprendizaje. Baste decir que, al cabo de pocas iteraciones la red clasifica correctamente los cuatro vectores de entrada. La figura 1.8 muestra la situación de los vectores en el plano y de la función discriminante al comienzo y al final del proceso de aprendizaje.

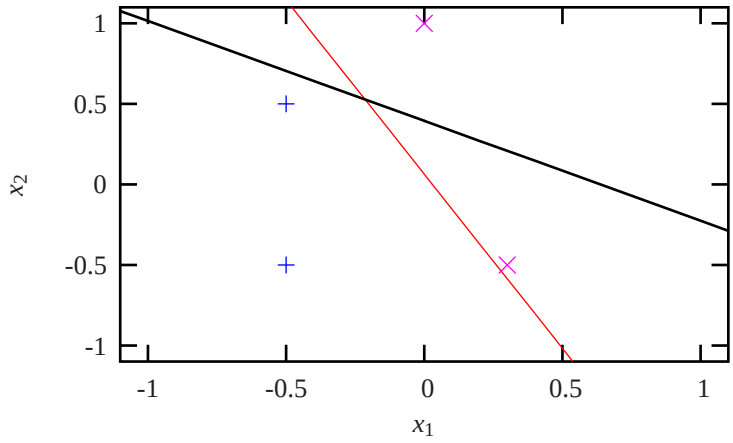


Figura 1.8: Hiperplano de separación al comienzo (línea recta de trazo negro) y al final (trazo rojo) del proceso de aprendizaje.

Como se ha demostrado anteriormente, una neurona formal de McCulloch-Pitts es un discriminante que divide el espacio de entrada en dos regiones separadas por un hiperplano (o una recta si, como en los ejemplos, el espacio de entrada es bidimensional). Por lo tanto, un perceptrón, que no es más que un conjunto de N_s neuronas de McCulloch-Pitts que comparten las entradas, divide el espacio de entrada en 2^{N_s} regiones separadas por hiperplanos. Así pues, un perceptrón será una herramienta de clasificación apropiada siempre que las clases definidas en el espacio de entrada sean separables mediante rectas (o hiperplanos) como es el caso del ejemplo anterior y del que se incluye al final del capítulo. Considérese sin embargo el conjunto de pares de entrenamiento mostrado en la figura 1.9. Como se podrá comprobar rápidamente, no existe ningún perceptrón de una única neurona capaz de separar el espacio de entrada en dos regiones que contengan todos los puntos correspondientes a cada clase. Si aumentamos el número de neuronas en el perceptrón aumentaremos el número de clases pero no conseguiremos que los puntos de la misma clase produzcan una misma salida del perceptrón. El ejemplo de la figura 1.9 es lo que se conoce como un problema no separable linealmente y representa una limitación muy seria de la capacidad de clasificación de los perceptrones.

Separabilidad lineal

Sin embargo, para problemas con la propiedad de separabilidad lineal tenemos que cada actualización del vector de pesos hace disminuir la función error dada. Si llamamos $\vec{\omega}^{\tau}$ y $\vec{\omega}^{\tau+1}$ a los vectores de pesos antes y después de la actualización respectivamente, tenemos que la nueva contribución al error debida a un patrón dado después de actualizar los pesos según 1.9 será

$$\begin{aligned} -(\vec{\omega}^{\tau+1})^T \cdot \vec{x}[n] \cdot t[n] &= -(\vec{\omega}^{\tau})^T \cdot \vec{x}[n] \cdot t[n] - (\alpha \cdot \vec{x}[n] \cdot t[n])^T \cdot \vec{x}[n] \cdot t[n] \\ &= -(\vec{\omega}^{\tau})^T \cdot \vec{x}[n] \cdot t[n] - \alpha \cdot (\vec{x}[n] \cdot t[n])^2 \\ &< -(\vec{\omega}^{\tau})^T \cdot \vec{x}[n] \cdot t[n], \end{aligned} \tag{1.15}$$

es decir, menor que la contribución previa a la actualización de los pesos.

Teorema de convergencia del perceptrón

Más aún, el *teorema de convergencia* garantiza que un perceptrón es capaz de hallar la solución que clasifica todos los patrones de entrenamiento en un número finito de pasos. La demostración del teorema basada en la interpretación de Bishop [Bis95] del original de Hertz et al. [HKP91] se incluye en el apéndice A.

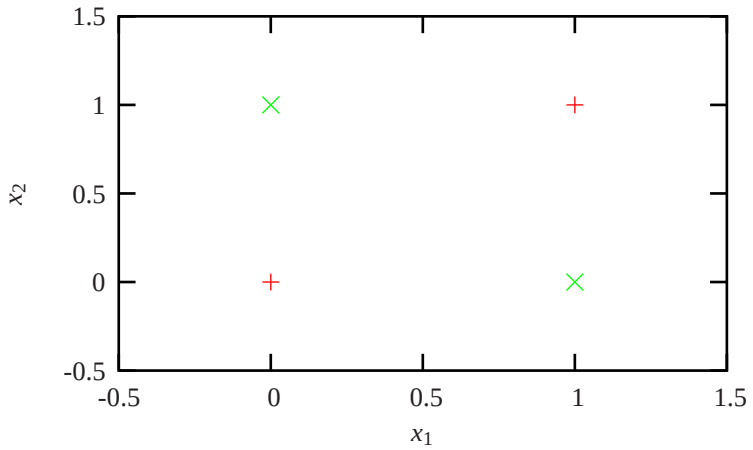


Figura 1.9: Representación gráfica del problema XOR. Un perceptrón de una capa no puede sintetizar artificialmente este problema por no ser linealmente separable en la variables x_1, x_2. ¿Se le ocurre al lector alguna forma de resolverlo con una cambio de variables?

La extensión de todo lo antedicho a un perceptrón de más de una neurona implica la definición del peso de la entrada i a la neurona de salida j como ω_{ji}. Ésto a su vez dará lugar a que cada neurona de salida del perceptrón posea su propio vector de pesos $\vec{\omega}_j$ de componentes ω_{ji} para $i = 1, 2, ..., N_e$. Finalmente, para el par de entrenamiento $(\vec{x}[n], \vec{t}[n])$ de N_e y N_s componentes respectivamente tendremos

$$a_j = \sum_{i=0}^{N_e} \omega_{ji} \cdot x_i = (\vec{\omega}_j)^T \cdot \vec{x}, \tag{1.16}$$

$$\Delta\vec{\omega}_j = \alpha \cdot \sum_{n=1}^{d} \vec{x}[n] \cdot \frac{(t_j[n] - y_j[n])}{2} \tag{1.17}$$

y

$$\Delta\vec{\omega} = \alpha \cdot \sum_{n=1}^{d} \vec{x}[n] \cdot (t_j[n] - y_j[n]). \tag{1.18}$$

para las dos posibles funciones de activación del perceptrón.

Adalines/Madalines

Bernard Widrow y Marcian Hoff [WH60] introdujeron en 1960 una variante de las neuronas de McCulloch-Pitts con propiedades interesantes. La variación de un *ADAptive LINear Element* (elemento adaptativo lineal o ADALINE) respecto a las neuronas de McCulloch-Pitts consiste en utilizar una nueva función de activación denominada función de activación lineal dada por la ecuación $g(a) = a$ y representada en la figura 1.10. Esta modificación permite reescribir la definición de la función error de una forma que se demostrará mucho más general y potente que la de la ecuación 1.9 basada, ya no en la suma ponderada de las entradas $(\vec{\omega})^T \cdot \vec{x}[n]$ sino directamente en la salida de la neurona $y[n]$. La nueva definición del error vendrá dada por la ecuación

$$E(\omega) = -\frac{1}{2} \sum_{n=1}^{d} \|t[n] - y[n]\|^2 = -\frac{1}{2} \sum_{n=1}^{d} (t[n] - y[n])^T \cdot (t[n] - y[n]) \tag{1.19}$$

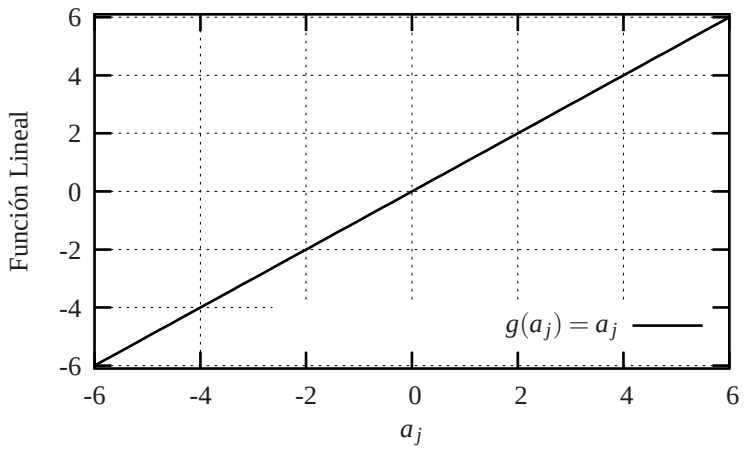


Figura 1.10: Función de activación lineal.

Error cuadrático medio

conocida como error cuadrático medio (*Mean Square Error* en inglés). En el caso de tener más de una neurona de salida tendremos que $t[n]-y[n]$ pasará a $\vec{t}[n]-\vec{y}[n]$ donde tanto $\vec{t}[n]$ como $\vec{y}[n]$ serán vectores de N_s componentes. Con esta definición del error, la ecuación de actualización de los pesos obtenida aplicando el método de descenso del gradiente (recuérdese que no es el único posible) será

$$\Delta\vec{\omega}_j = \alpha \cdot \sum_{n=1}^{d} (t_j[n] - y_j[n]) \cdot \vec{x}[n]. \tag{1.20}$$

donde hemos utilizado la ecuación 1.10 y la definición de g.

A un conjunto de ADALINES se lo conoce como red MADALINE (Multiple ADAptive LINear Element).

1.3. Arquitecturas

El primer modelo de red neuronal presentado hasta ahora ya ha introducido varios conceptos fundamentales en la definición de una arquitectura de red. En primer lugar, hemos visto que el bloque constructivo fundamental de una red neuronal es, obviamente, la neurona entendida como un módulo computacional con una o varias entradas y una salida definida como una función de activación aplicada a la suma de las entradas ponderada por los pesos de las conexiones respectivas[c]. Más aún, hemos empleado el

[c] Sería posible, en principio, una definición más general de neurona si incluimos en la suma ponderada, no sólo las entradas de la neurona, sino también combinaciónes de orden superior de las entradas. Por ejemplo, para una neurona de dos entradas, podríamos definir a_j como

$$a_j = \omega_1 \cdot x_1 + \omega_2 \cdot x_2 + \omega_{11} \cdot x_1^2 + \omega_{22} \cdot x_2^2 + \omega_{12} \cdot x_1 \cdot x_2.$$

Ésta definición, que incluye términos de segundo orden o cuadráticos, podría a su vez ser extendida a órdenes superiores. Sin embargo, la introducción de términos de orden superior al lineal en capas distintas de la capa de entrada introduce una complicación computacional no compensada por una mayor efectividad. El caso de la capa de entrada se tratará con más detalle en la sección dedicada al preprocesado de los datos.

término capa en numerosas ocasiones sin haber dado una definición precisa del término. En este texto, se entenderá por capa *un conjunto de neuronas con la misma definición funcional de las operaciones que realiza sobre sus entradas y que recibe entradas o envía salidas a un mismo conjunto de neuronas entre las que se pueden encontrar miembros de la misma capa*. Así, en el perceptrón del ejemplo anterior hemos denominado capa de entrada a un conjunto de neuronas que no recibía conexiones de ninguna neurona pero que enviaba su salida a un mismo conjunto de neuronas y, de nuevo, hemos empleado el término capa (de salida) para el conjunto restante de neuronas que no envía su salida a ninguna neurona, pero sí recibe sus entradas de un mismo grupo de neuronas. Las primeras comparten una misma definición funcional: toman su entrada y la trasmiten a las neuronas a las que están conectadas sin realizar ninguna operación sobre ellas. Podríamos decir que tienen una única entrada, que el peso de la entrada es la unidad, y que su función de activación es lineal. Por el contrario las segundas realizan una suma de las entradas ponderadas por los pesos de las conexiones respectivas y al resultado le aplican la función de activación paso. En el caso de MADALINES tendremos funciones de activación lineales en vez de funciones paso pero de nuevo tendremos dos capas que cumplen la definición dada más arriba. Finalmente, siguiendo un orden creciente de complejidad, nos encontramos con la definición de la arquitectura de una red neuronal en términos del número de capas, del número de neuronas en cada capa y de la conectividad entre capas e *intra* capa. Según este último aspecto (la conectividad de la red o topología conectiva) nos encontraremos redes unidireccionales o de alimentación hacia delante (*feedforward* en inglés) si existe una relación de orden parcial entre capas, de forma que cada capa recibe entradas únicamente de la capa anterior y envía su salida únicamente a la siguiente. El perceptrón de la figura 1.6 es un ejemplo de arquitectura unidireccional muy simple puesto que sólo lo componen dos capas (veremos ejemplos más complejos en capítulos posteriores). Si desde una capa dada se realizan conexiones a capas anteriores tendremos lo que se conoce como realimentación y la arquitectura de la red será recurrente (ejemplos de arquitecturas recurrentes se dan en las redes neuronales basadas en la teoría de resonancia adaptativa ART o *Adaptive Resonance Theory* en inglés y en las memorias asociativas bidireccionales BAM o *Bidirectional Associative Memories* en inglés). Finalmente, si existen conexiones entre las neuronas de una misma capa, diremos que existe interacción lateral en la red. Un ejemplo de red neuronal de tres capas con conexiones recurrentes (en trazo discontinuo) e interacción lateral (trazo discontinuo con puntos intercalados) se muestra en la figura 1.11.

1.4. Tareas genéricas en Neurocomputación

A lo largo de todo este capítulo hemos utilizado problemas de clasificación para ilustrar diversos aspectos de las redes neuronales presentadas. Sin embargo, la clasificación es un caso particular de tarea genérica realizable por una red neuronal. El conjunto de tareas para cuya realización se recurre a redes neuronales se ajusta a un esquema conceptual de tarea conexionista en el que el elemento central lo constituye una relación entre espacios de representación y la tarea consiste en hallar el objeto/objetos del espacio de representación de salida que la relación asocia a un objeto de entrada dado. En el caso de la tarea genérica de clasificación, una red neuronal bien entrenada se convierte en un modelo de la relación que media entre los objetos del espacio de entrada (puntos del plano en los ejemplos de este capítulo) y las clases a las que pertenecen (en el espacio de representación de salida). Esta relación de 'pertenencia a la clase C_k' queda impresa en los valores de las conexiones de la red de forma que

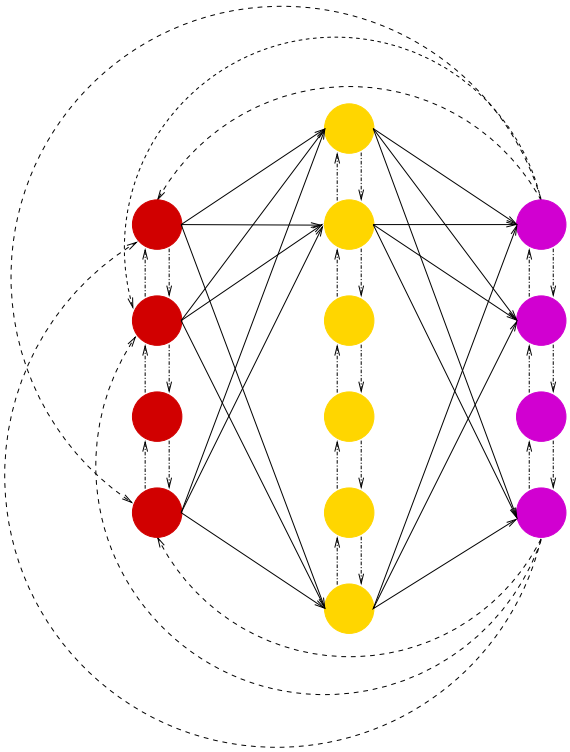


Figura 1.11: Ejemplo de arquitectura de red con conexiones recurrentes (en trazo discontinuo) e interacción lateral (trazo discontinuo con puntos intercalados).

Generalización

incluso es capaz de clasificar correctamente objetos que no le habían sido presentados con anterioridad (propiedad conocida como generalización). A continuación, y sin pretensión de ser exhaustivos, examinamos varias tareas que se ajustan al esquema conceptual descrito.

1.4.1. Redes neuronales en tareas de clasificación

En realidad ya hemos visto con el nivel de profundidad adecuado a este capítulo introductorio las características de una red neuronal clasificadora. Aquí sólo vamos a repasar nociones básicas ya introducidas informalmente en secciones anteriores y a apuntar una interpretación probabilista de los resultados de una clasificación neuronal. La clasificación consta de tres etapas: en la primera se realiza un proceso de abstracción que, partiendo del universo de objetos de los que trata el problema de clasificación, extrae características suficientes para reconocer como similares objetos de la misma clase y como disímiles aquéllos de distintas clases; en la segunda, se comparan las características del objeto que deseamos clasificar con un modelo de dicho universo y se selecciona la clase a la que se encuentra más próximo; finalmente en la tercera, se genera una respuesta de acuerdo con el resultado de la comparación que, generalmente, se interpreta como las probabilidades *a posteriori* de pertenencia del objeto de entrada a cada una de las clases/particiones del espacio de entrada. De esta forma, las activaciones $y_k(\vec{x})$ de las neuronas de la capa de salida serían funciones del vector de entrada que, interpretadas en el dominio del observador que crea la red, co-

rresponderían a la probabilidad $\mathcal{P}(\mathcal{C}_k|\vec{x})$ de que, dado el vector $\vec{x}$, éste pertenezca a la clase $\mathcal{C}_k$. Existen una serie de condiciones que ha de cumplir una red neuronal para que la activación de sus neuronas de salida pueda ser interpretada en sentido probabilístico que serán objeto de análisis en capítulos posteriores.

1.4.2. Redes neuronales para regresión

Ésta tarea está relacionada con la tarea de clasificación y con problemas de interpolación de funciones. Formulada en términos consistentes con la anterior descripción se trata de, dado un conjunto de pares de vectores $\mathcal{A} = (\vec{x}[n], \vec{t}[n])$, con $\vec{x}[n] \in \mathbb{R}^{N_e}$ y $\vec{t}[n] \in \mathbb{R}^{N_s}$, predecir la asociación $\vec{y}$ en el espacio de salida, para un vector $\vec{x}$ no perteneciente al conjunto $\mathcal{A}$[d]. La trasposición del esquema conceptual de tarea conexionista al caso de la regresión es inmediata si suponemos que el conjunto $\mathcal{A}$ define una función vectorial de varias variables tal que $\vec{t}[n] = \vec{f}(\vec{x}[n])$ donde

$$\vec{f} : \mathbb{R}^{N_e} \rightarrow \mathbb{R}^{N_s}. \tag{1.21}$$

En este caso, la relación entre el espacio de entrada y el espacio de salida viene dada por las componentes f_1, f_2, f_{N_s} de la función vectorial $\vec{f}$ y el modelo de la relación queda impreso en los pesos de las conexiones de la red. Por analogía vemos que el proceso de aprendizaje, en el que se modifican los pesos de la red para ajustar la función $\vec{f}$ a partir de los pares de entrenamiento del conjunto $\mathcal{A}$ es enteramente equivalente a una regresión sobre dicho conjunto.[e]. Un ejemplo de este tipo de aplicaciones consistiría en obtener a partir del conjunto de puntos mostrado en la figura 1.12, una aproximación de la función continua generatriz (dibujada en trazo negro continuo). Este tipo de situaciones se da cuando la función generatriz se ha medido mediante un procedimiento que introduce ruido de distribución gaussiana. Se trataría entonces de encontrar una aproximación a la función $y(x)$ generatriz mediante una red neuronal.

Veremos más adelante las propiedades de diferentes algoritmos de aprendizaje en lo referente a este tipo de tarea genérica.

1.4.3. Redes neuronales en tareas de control

En los dos casos anteriores hemos visto cómo la red neuronal se constituía en puente entre dos espacios de representación, de forma que a un elemento dado del espacio de entrada (ya fuera éste un objeto que deseamos clasificar o un vector de $\mathbb{R}^{N_e}$) le corresponde una salida que el creador de la red interpreta en el espacio de representación de salida como una clase o un valor de la función generatriz ajustada. Como hemos tenido ocasión de comprobar, nada hay en la implementación de una red neuronal que haga

[d] Repárese en la similitud con el concepto de generalización en el caso de tareas de clasificación. En última instancia, interpretada a la luz del esquema conceptual de tarea conexionista, se trata de la misma propiedad

[e] Los conocimientos necesarios para entender los conceptos introducidos en el último párrafo se pueden encontrar en cualquier manual de introducción a la Estadística o al Cálculo. En particular, los alumnos de la UNED deben dominar estos conceptos como resultado de haber cursado previamente las asignaturas de **Estadística** I y **Ampliación de Matemáticas**

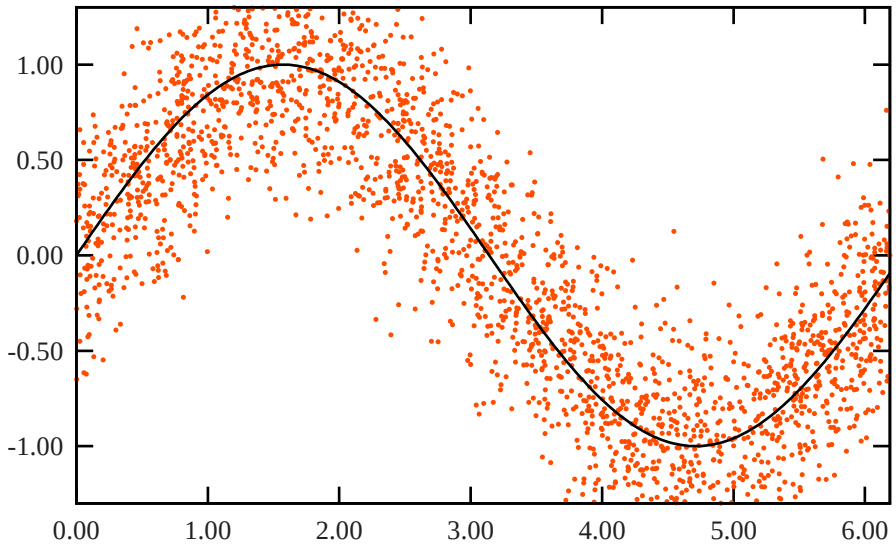


Figura 1.12: Problema típico de regresión: dado el conjunto de puntos de la gráfica (que se supone generado a partir de una función continua como la representada en trazo negro) recuperar la mejor aproximación a la función generatriz. Por supuesto, no se conoce de antemano dicha función. En este ejemplo artificial, los datos se han obtenido a partir de una función seno por adición de ruido gaussiano.

referencia explícita a clases, regresiones, probabilidades etc. Todo está en la interpretación del creador de la red. Pues bien, en el caso del empleo de redes neuronales para tareas de control, las entradas se interpretan como variables de estado de un sistema, y las salidas, como un conjunto de instrucciones de actuación sobre dicho sistema. Por ejemplo, podríamos entrenar una red neuronal para conducir un coche de forma que la entrada de la red fuese una imagen de la carretera por la que circulamos (incluyendo referencias visuales como la señalización horizontal de los carriles) y la salida, el ángulo de giro del volante necesario para mantener el vehículo dentro de las señales de referencia. Otro ejemplo podría ser el de una red neuronal que monitoriza una planta nuclear y recibe como entradas parámetros del reactor como temperatura, presión, etc y cuyas respuestas se interpretan como acciones tendentes a mantener dichos valores dentro de unos rangos de seguridad determinados.

1.4.4. Redes neuronales en tareas de predicción temporal

Un último ejemplo de aplicación de red neuronal consiste en la predicción de series temporales. En este caso, el espacio de entrada lo componen registros temporales de los valores de una (o varias) variables de estado de un sistema y la salida se interpreta como una predicción del valor de la variable para valores de la coordenada temporal posteriores al registro. El ejemplo clásico de este tipo de aplicaciones es el de la predicción de valores bursátiles a partir de series temporales de índices de mercado. Tanto las tareas de control como las de predicción temporal puedes ser vistas como casos particulares de tareas de regresión en los que las funciones que queremos ajustar son de una clase especial.

Repaso de conocimientos

Ejercicios de consolidación del aprendizaje.

Ejemplo 2 (Perceptrón 2D)

Entrénese un perceptrón para que clasifique correctamente el siguiente conjunto de entrenamiento.

$n \rightarrow$	1	2	3	4	5	6	7	8	9	10
$\vec{x}[n]$	$\begin{bmatrix}+0,1\\+1,2\end{bmatrix}$	$\begin{bmatrix}+0,7\\+1,8\end{bmatrix}$	$\begin{bmatrix}+0,8\\+1,6\end{bmatrix}$	$\begin{bmatrix}+0,8\\+0,6\end{bmatrix}$	$\begin{bmatrix}+1,0\\+0,8\end{bmatrix}$	$\begin{bmatrix}+0,3\\+0,5\end{bmatrix}$	$\begin{bmatrix}+0,0\\+0,2\end{bmatrix}$	$\begin{bmatrix}-0,3\\+0,8\end{bmatrix}$	$\begin{bmatrix}-0,5\\-1,5\end{bmatrix}$	$\begin{bmatrix}-1,5\\-1,3\end{bmatrix}$
$\vec{t}[n]$	$\begin{bmatrix}+1,0\\+0,0\end{bmatrix}$	$\begin{bmatrix}+1,0\\+0,0\end{bmatrix}$	$\begin{bmatrix}+1,0\\+0,0\end{bmatrix}$	$\begin{bmatrix}+0,0\\+0,0\end{bmatrix}$	$\begin{bmatrix}+0,0\\+0,0\end{bmatrix}$	$\begin{bmatrix}+1,0\\+1,0\end{bmatrix}$	$\begin{bmatrix}+1,0\\+1,0\end{bmatrix}$	$\begin{bmatrix}+1,0\\+1,0\end{bmatrix}$	$\begin{bmatrix}+0,0\\+1,0\end{bmatrix}$	$\begin{bmatrix}+0,0\\+1,0\end{bmatrix}$

Supóngase que la velocidad de aprendizaje α es 1.0, que la actualización de los pesos se realiza sólo al final de un ciclo (entrenamiento por lotes) y que el proceso de aprendizaje termina cuando todos los vectores de entrada del conjunto de entrenamiento son clasificados correctamente.

Solución 2: Del enunciado del ejercicio se desprende que debemos dividir el espacio de entrada en cuatro clases a las que se les asigna las cuatro posibles salidas de un perceptrón con $N_s = 2$: $(0,0),(0,1),(1,0),(1,1)$. Además, dicho espacio de entrada está constituido por puntos del plano, por lo que $N_e = 2$. La figura 1.13 muestra el conjunto de entrenamiento dado por el enunciado.

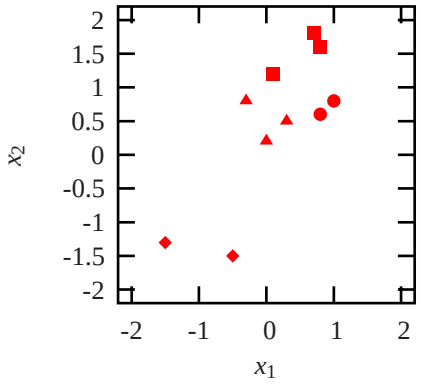


Figura 1.13: Representación gráfica del conjunto de entrenamiento. Las cuatro salidas posibles (0,0), (0,1), (1,0) y (1,1) se han representado con diamantes, cuadrados, círculos y triángulos.

Como se puede comprobar, define cuatro clases separables linealmente, por lo que tiene sentido utilizar un perceptrón para realizar la tarea de clasificación demandada. Si inicializáramos aleatoriamente los pesos de la red, podríamos obtener los siguientes valores:

Neurona de salida 1	ω_{10}	ω_{11}	ω_{12}
	-0.2693	-0.3435	0.5128
Neurona de salida 2	ω_{20}	ω_{21}	ω_{22}
	-0.5059	0.2653	0.9821

donde, de nuevo, hemos representado la polarización como el peso ω_{j0} asociado a una nueva entrada $x_0 = 1$. Comenzamos el primer ciclo de aprendizaje cuyos resultados se resumen a continuación. Téngase en cuenta que el enunciado indica que debemos realizar un aprendizaje por lotes. Ésto implica que no se actualizarán los pesos hasta haber presentado al perceptrón el conjunto entero de pares de entrenamiento.

n	a_1	a_2	$\vec{y}[n]$	$\vec{t}[n]-\vec{y}[n]$	$\Delta\vec{\omega}_0$	$\Delta\vec{\omega}_1$	$\Delta\vec{\omega}_2$
1	+0.3117	+0.6991	$\begin{bmatrix}+1,0\\+1,0\end{bmatrix}$	$\begin{bmatrix}+0,0\\-1,0\end{bmatrix}$	$\begin{bmatrix}+0,0\\-1,0\end{bmatrix}$	$\begin{bmatrix}+0,0\\-0,1\end{bmatrix}$	$\begin{bmatrix}+0,0\\-1,2\end{bmatrix}$
2	+0.4133	+1.4476	$\begin{bmatrix}+1,0\\+1,0\end{bmatrix}$	$\begin{bmatrix}+0,0\\-1,0\end{bmatrix}$	$\begin{bmatrix}+0,0\\-1,0\end{bmatrix}$	$\begin{bmatrix}+0,0\\-0,7\end{bmatrix}$	$\begin{bmatrix}+0,0\\-1,8\end{bmatrix}$
3	+0.2764	+1.2777	$\begin{bmatrix}+1,0\\+1,0\end{bmatrix}$	$\begin{bmatrix}+0,0\\-1,0\end{bmatrix}$	$\begin{bmatrix}+0,0\\-1,0\end{bmatrix}$	$\begin{bmatrix}+0,0\\-0,8\end{bmatrix}$	$\begin{bmatrix}+0,0\\-1,6\end{bmatrix}$
4	-0.2364	+0.2956	$\begin{bmatrix}+0,0\\+1,0\end{bmatrix}$	$\begin{bmatrix}+0,0\\-1,0\end{bmatrix}$	$\begin{bmatrix}+0,0\\-1,0\end{bmatrix}$	$\begin{bmatrix}+0,0\\-0,8\end{bmatrix}$	$\begin{bmatrix}+0,0\\-0,6\end{bmatrix}$
5	-0.2026	+0.5451	$\begin{bmatrix}+0,0\\+1,0\end{bmatrix}$	$\begin{bmatrix}+0,0\\-1,0\end{bmatrix}$	$\begin{bmatrix}+0,0\\-1,0\end{bmatrix}$	$\begin{bmatrix}+0,0\\-1,0\end{bmatrix}$	$\begin{bmatrix}+0,0\\-0,8\end{bmatrix}$
6	-0.1159	+0.0647	$\begin{bmatrix}+0,0\\+1,0\end{bmatrix}$	$\begin{bmatrix}+1,0\\+0,0\end{bmatrix}$	$\begin{bmatrix}+1,0\\+0,0\end{bmatrix}$	$\begin{bmatrix}+0,3\\+0,0\end{bmatrix}$	$\begin{bmatrix}+0,5\\+0,0\end{bmatrix}$
7	-0.1667	-0.3095	$\begin{bmatrix}+0,0\\+0,0\end{bmatrix}$	$\begin{bmatrix}+1,0\\+1,0\end{bmatrix}$	$\begin{bmatrix}+1,0\\+1,0\end{bmatrix}$	$\begin{bmatrix}+0,0\\+0,0\end{bmatrix}$	$\begin{bmatrix}+0,2\\+0,2\end{bmatrix}$
8	+0.2440	+0.2002	$\begin{bmatrix}+1,0\\+1,0\end{bmatrix}$	$\begin{bmatrix}+0,0\\+0,0\end{bmatrix}$	$\begin{bmatrix}+0,0\\+0,0\end{bmatrix}$	$\begin{bmatrix}+0,0\\+0,0\end{bmatrix}$	$\begin{bmatrix}+0,0\\+0,0\end{bmatrix}$
9	-0.8667	-2.1117	$\begin{bmatrix}+0,0\\+0,0\end{bmatrix}$	$\begin{bmatrix}+0,0\\+1,0\end{bmatrix}$	$\begin{bmatrix}+0,0\\+1,0\end{bmatrix}$	$\begin{bmatrix}+0,0\\-0,5\end{bmatrix}$	$\begin{bmatrix}+0,0\\-1,5\end{bmatrix}$
10	-0.4207	-2.1806	$\begin{bmatrix}+0,0\\+0,0\end{bmatrix}$	$\begin{bmatrix}+0,0\\+1,0\end{bmatrix}$	$\begin{bmatrix}+0,0\\+1,0\end{bmatrix}$	$\begin{bmatrix}+0,0\\-1,5\end{bmatrix}$	$\begin{bmatrix}+0,0\\-1,3\end{bmatrix}$

Sumando todas las modificaciones de los pesos obtenemos

Neurona de salida 1	$\Delta\omega_{10}$	$\Delta\omega_{11}$	$\Delta\omega_{12}$	ω_{10}	ω_{11}	ω_{12}
	+2.0	+0.3	+0.7	+1.7307	-0.0435	+1.2128
Neurona de salida 2	$\Delta\omega_{20}$	$\Delta\omega_{21}$	$\Delta\omega_{22}$	ω_{20}	ω_{21}	ω_{22}
	-2.0	-5.4	-8.6	-2.5059	-5.1347	-7.6179

Este ciclo se repite hasta que, como resultado de las actualizaciones de los pesos, la red clasifica adecuadamente todos los pares de entrenamiento. Ésto ocurre, para las condiciones dadas por el enunciado, después de 13 ciclos de actualización como muestra la figura 1.14. Los valores de los pesos en ese momento son:

Neurona de salida 1	$\Delta\omega_{10}$	$\Delta\omega_{11}$	$\Delta\omega_{12}$	ω_{10}	ω_{11}	ω_{12}
	+2.0	+0.3	+0.7	-0.2693	-4.5435	+5.8128
Neurona de salida 2	$\Delta\omega_{20}$	$\Delta\omega_{21}$	$\Delta\omega_{22}$	ω_{20}	ω_{21}	ω_{22}
	+0.0	+0.0	+0.0	+4.4941	-4.8347	-4.1179

La figura muestra también, para facilitar comparaciones, la posición inicial de los discriminantes cuando los valores de los pesos son inicializados aleatoriamente y después de 7 actualizaciones.

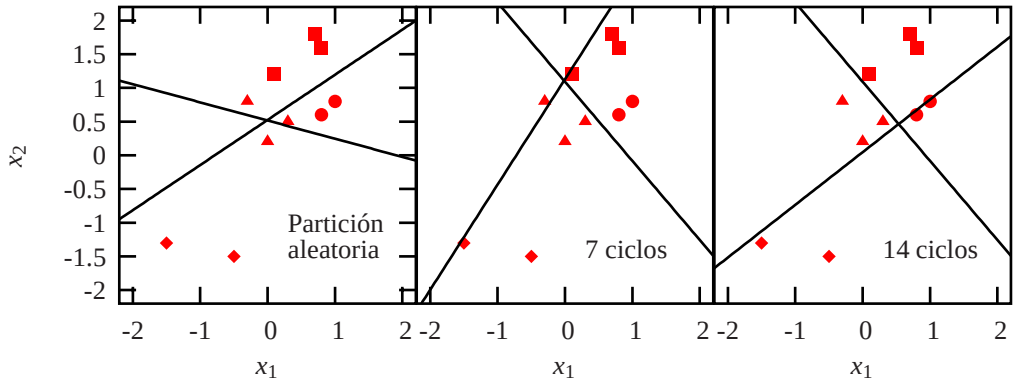


Figura 1.14: Posición de los hiperplanos separadores para el ejemplo de clasificación en diversos puntos del proceso de aprendizaje.

Capítulo 2

Retropropagación del error

Objetivo: El presente módulo se propone como objetivo introducir el que probablemente sea el método de aprendizaje más popular en aplicaciones reales. A medida que vayamos describiéndolo, iremos introduciendo nuevos conceptos que completarán y extenderán el marco conceptual introducido en el capítulo anterior.

2.1. El formalismo matemático del algoritmo.

2.1.1. La arquitectura de la red

El algoritmo de aprendizaje basado en la retropropagación del gradiente del error está diseñado para redes neuronales sin realimentación (*feedforward networks* en inglés). La arquitectura más simple es la de un perceptrón con más de una capa de procesamiento. Recordemos que el perceptrón clásico está compuesto de una capa de entrada (denominada a veces sensorial) cuya única misión es redistribuir las entradas y una capa de procesado que sí realiza cálculo neuronal (suma ponderada más función de activación). En este capítulo nos centraremos en redes neuronales de la misma arquitectura que el perceptrón (sin realimentación ni interacción lateral), pero con dos o más capas de procesamiento. A estas redes se las conoce como perceptrones multicapa y nos van a servir de marco para la exposición del proceso de aprendizaje por retropropagación del error aun cuando debe recordarse que existen múltiples variaciones que pueden aumentar la complejidad de la red sin modificar el fundamento teórico del algoritmo. Vamos a considerar, para fijar las ideas, que la red está compuesta de tres capas unidimensionales de N_e, N_o y N_s neuronas cada una, llamadas capa de entrada (*input layer* en inglés), capa oculta (*hidden layer*) y capa de salida (*output layer*). Las capas tienen conectividad total entre sí (cada neurona forma sinapsis con todas las de la siguiente capa) y conectividad interna nula (no existen conexiones sinápticas laterales dentro de una misma capa). La figura 2.1.1 muestra esquemáticamente la arquitectura de dicha red. Como vimos en el capítulo anterior, podemos considerar la salida de la red como una serie de funciones y_k de $\mathbb{R}^{N_e}$ en $\mathbb{R}$, que para cada conjunto de valores de entrada $(\vec{x})^T = (x_1, x_2, ..., x_{N_e})$ nos proporciona un vector a la salida $(\vec{y})^T = (y_1(\vec{x}), y_2(\vec{x}), ..., y_{N_s}(\vec{x}))$.

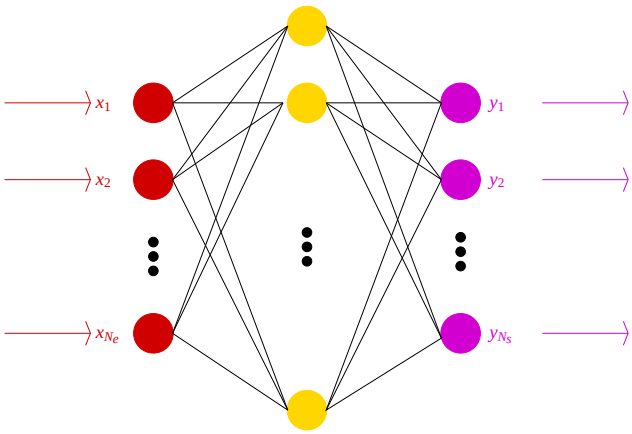


Figura 2.1: Perceptron multicapa.

Cada problema tendrá un espacio de entrada y uno de salida de dimensionalidad distinta por lo que los valores de N_e y N_s variarán dependiendo del caso concreto que estemos tratando. El valor de N_o depende de las características del espacio de entrada por lo que no se pueden enunciar reglas generales relativas a su valor en una aplicación concreta.Sólo poseemos algunas indicaciones sobre qué rangos de valores son más adecuados. Más adelante veremos algunas de estas normas y su interpretación.

Como hemos mencionado anteriormente, la capa de entrada no realiza ninguna operación algebraica con su única entrada, de forma que la neurona i de dicha capa envía a todas las neuronas de la capa siguiente el valor x_i. Las neuronas de la capa oculta sí transforman sus entradas en un valor distinto mediante operaciones matemáticas. Como vimos anteriormente, llamamos salida o activación de la neurona al resultado de las operaciones realizadas sobre sus entradas. Decimos que la arquitectura de la red va a ser la de un perceptrón multicapa porque las operaciones que van a realizar tanto las neuronas de la capa oculta como las de la de salida son las mismas que realiza el perceptrón. Para neuronas de la capa oculta, estas operaciones consisten en multiplicar las entradas procedentes de las neuronas de la capa anterior por sus pesos respectivos, sumar el resultado de las N_e multiplicaciones y, finalmente, aplicar la función de activación al resultado. Empleando la notación introducida en el capítulo anterior para los pesos de una conexión sináptica tendremos que la neurona j de la capa oculta realiza las siguientes operaciones:

- Multiplica la entrada x_1 por el peso de la conexión entre la neurona 1 de la capa de entrada y la neurona j de la capa oculta (ω_{j1})
- Multiplica la entrada x_2 por el peso de la conexión entre la neurona 2 de la capa de entrada y la neurona j de la capa oculta (ω_{j2})
- ...
- Multiplica la entrada x_{N_e} por el peso de la conexión entre la neurona N_e de la capa de entrada y la neurona j de la capa oculta (ω_{jN_e})
- Suma el resultado de todos los pasos anteriores más la polarización como entra-

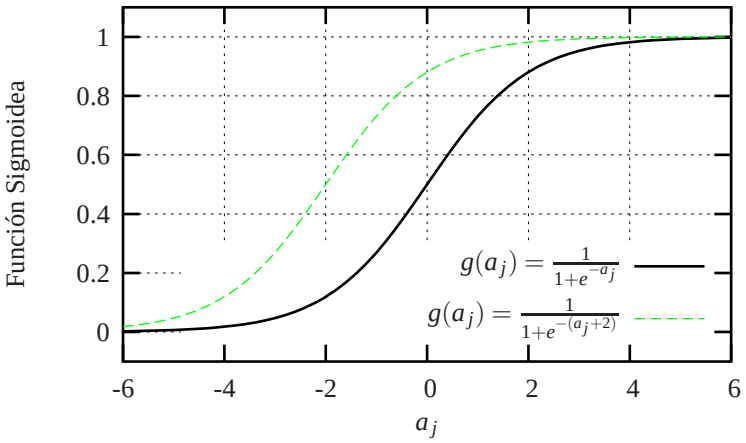


Figura 2.2: Función de activación sigmoide con $\kappa = 1$ y dos valores de la polarización, $\omega_{j0} = 0$ (línea contínua) y $\omega_{j0} = 2$ (línea verde discontínua). En la leyenda de la figura hemos considerado que a_j no incorpora la polarización para hacer más evidente su efecto.

da de subíndice cero:

$$a_j = \omega_{j0} \cdot 1 + \omega_{j1} \cdot x_1 + \omega_{j2} \cdot x_2 + ... + \omega_{jN_e} \cdot x_{N_e} = \sum_{i=0}^{N_e} \omega_{ji} \cdot x_i \qquad (2.1)$$

- Aplicar la función umbral g al resultado:

$$y_j = g(a_j) = g(\sum_{i=0}^{N_e} \omega_{ji} \cdot x_i) \qquad (2.2)$$

Recordemos que éstas son las transformaciones realizadas por las neuronas de un perceptrón y que hemos empleado la misma notación que en el capítulo anterior llamando a_j al resultado de sumar ponderadamente las entradas de la neurona j y g a la función de activación.

Funciones de activación sigmoideas

Como se mencionaba en el capítulo anterior existen varias funciones de activación que se pueden utilizar en neurocomputación. Cada una de ellas va a conferir unas propiedades determinadas a la neurona o capa de neuronas que la utilicen. Hasta ahora hemos visto dos funciones de activación distintas: la función paso y la función lineal. Aquí vamos a introducir una nueva función de activación denominada sigmoide. La función sigmoide se define como

$$g(a) = \frac{1}{1 + \exp(-\kappa \cdot a)} \qquad (2.3)$$

su derivada vale

$$\frac{dg(a)}{da} = \frac{-\kappa \cdot \exp(-\kappa \cdot a)}{(1 + \exp(-\kappa \cdot a))^2} = \kappa \cdot g(a) \cdot (1 - g(a)) \qquad (2.4)$$

y su representación gráfica se muestra en la figura 2.2 para el valor $\kappa = 1$. Dicha figura muestra en realidad dos gráficas para dos funciones sigmoideas con y sin polarización

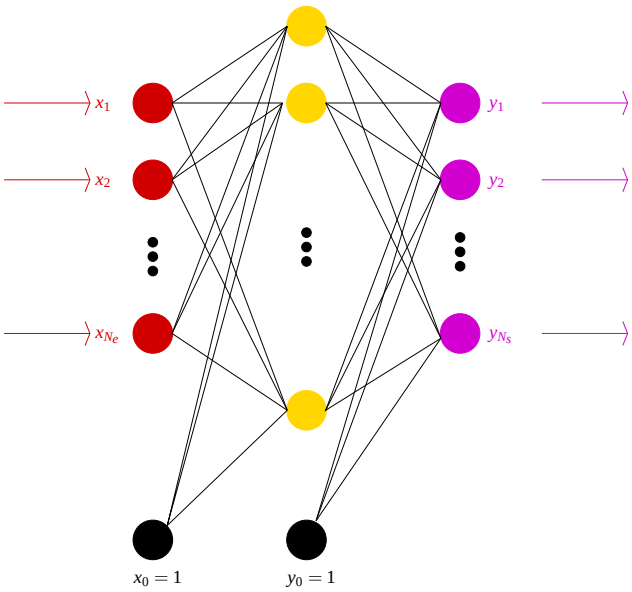


Figura 2.3: Perceptrón multicapa con polarizaciones incorporadas como neuronas de orden cero y entrada unidad.

o *bias*. Como hicimos en el capítulo anterior, vamos a incorporar las polarizaciones añadiendo una neurona de subíndice 0 a las capas de entrada y oculta según la figura 2.3. En la gráfica se distinguen dos regiones de saturación en las que el valor de la función es constante independientemente del valor de la entrada, y una región intermedia que depende linealmente del valor de a.

Se puede comprobar que la función paso empleada en el capítulo anterior es un caso particular de la función sigmoidea si tomamos el límite de κ tendiendo a infinito. En realidad κ es un parámetro proporcional a la pendiente de la zona lineal de la función, que se suele fijar de antemano sin que tenga un efecto importante en el proceso de aprendizaje. En el desarrollo de la exposición del algoritmo, vamos a suponer que las dos capas de procesamiento (la capa oculta y la capa de salida) van a tener funciones de activación sigmoideas. La extensión a funciones de activación lineales se verá en un ejemplo de aplicación.

En último lugar, después de la capa de entrada y de la intermedia u oculta, encontramos la capa de salida . Las neuronas de esta capa realizan la misma operación que las de la capa oculta, es decir, suman ponderadamente sus entradas y le aplican al resultado la función de activación, que, en esta ocasión, puede ser sigmoidea o lineal.

2.1.2. El algoritmo de ajuste de los pesos

De acuerdo con la descripción de la arquitectura hecha en la sección anterior, vemos que, efectivamente, podemos estudiar la red como una serie de N_s funciones paramétricas no lineales y_k (con $k = 1, 2, ..., N_s$) de la entrada $\vec{x}$ donde y_k representa la salida de la neurona k-ésima de la capa de salida. Decimos paramétrica porque las funciones y_k dependen de los valores de los pesos de la red, y para cada conjunto de valores de los pesos, la red será equivalente a un conjunto distinto de funciones y_k.

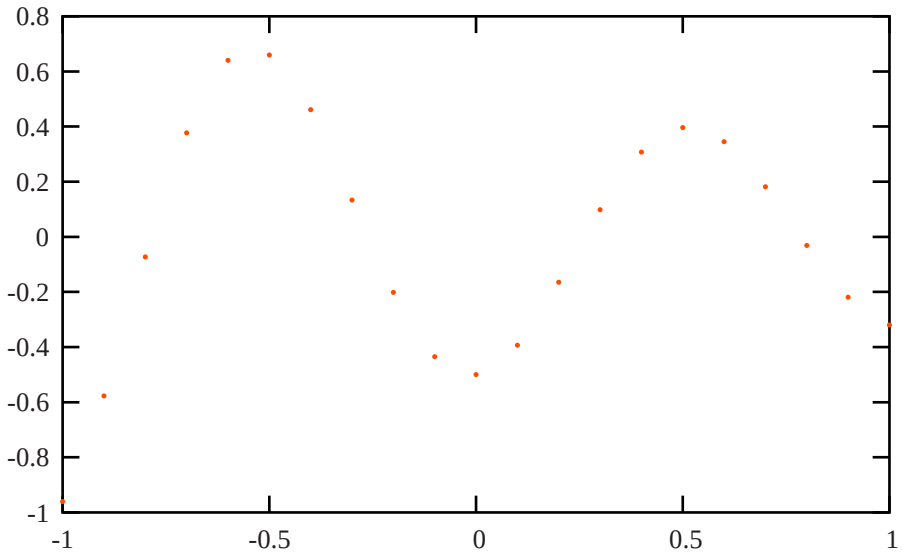


Figura 2.4: Representación gráfica del conjunto de pares de entrenamiento.

El objetivo entonces es encontrar los valores de las conexiones ω_{ji} que aproximen mejor unas funciones objetivo dadas. Las funciones objetivo pueden ser funciones reales de variable real o funciones de clasificación. Como hemos visto, en el primer caso, diremos que se trata de un problema de regresión; en el segundo, obviamente, de clasificación. Un ejemplo sencillo del primero sería el siguiente: supongamos que dispongo de una serie de pares $(x[n], t[n])$ como los siguientes, que definen una función:

$(x[n], t[n])$		$(x[n], t[n])$		$(x[n], t[n])$	
n=1	(-1.0, -0.9602)	n=2	(-0.9, -0.5770)	n=3	(-0.8, -0.0729)
n=4	(-0.7, 0.3771)	n=5	(-0.6, 0.6405)	n=6	(-0.5, 0.6600)
n=7	(-0.4, 0.4609)	n=8	(-0.3, 0.1336)	n=9	(-0.2, -0.2013)
n=10	(-0.1, -0.4344)	n=11	(0.0, -0.5000)	n=12	(0.1, -0.3930)
n=13	(0.2, -0.1647)	n=14	(0.3, 0.0988)	n=15	(0.4, 0.3072)
n=16	(0.5, 0.3960)	n=17	(0.6, 0.3449)	n=18	(0.7, 0.1816)
n=19	(0.8, -0.0312)	n=20	(0.9, -0.2189)	n=21	(1.0, -0.3201)

Si representamos gráficamente esos pares de puntos obtendremos la figura 2.4 en la que se puede comprobar que dichos puntos parecen dibujar una función continua.

Pues bien, podríamos utilizar la primera componente de los pares como entrada de una red cuya primera y última capa tuvieran una única neurona, y entrenar la red (ajustar los valores de los pesos) para que la salida de la red reproduzca la segunda componente del par. En esa situación y si el proceso de entrenamiento ha sido correcto en un sentido que matizaremos más adelante, la salida de la red será una aproximación a la función que parecían trazar los puntos aislados, de forma que si introducimos en la entrada de la red valores de x comprendidos entre -1 y 1 pero no pertenecientes a los pares de entrenamiento, la activación y de la única neurona de salida de la red será una interpolación a partir de los datos iniciales. Lo que acabamos de describir

es un proceso de entrenamiento supervisado para una tarea de regresión (ver capítulo anterior).

Ejemplos del segundo caso, el empleo de redes neuronales para tareas de clasificación, se han empleado en el capítulo anterior para ejemplificar el entrenamiento de un perceptrón.

¿Cómo se van a modificar los pesos en un proceso de entrenamiento? El algoritmo de retropropagación del error consiste en darle a los pesos valores inicialmente aleatorios, presentarle a la red entradas cuya salida conozcamos y comparar el resultado con la salida esperada. Si no coinciden (que es lo esperable puesto que hemos inicializado los pesos aleatoriamente) modificar los pesos de forma que el error en la clasificación disminuya y repetir cíclicamente este procedimiento hasta que el error quede por debajo de una cota mínima que nosotros impondremos. Todo esto está expresado en un lenguaje no muy clarificador. Vamos a ver matemáticamente cómo se haría. La exposición se hará con una notación diferente a la empleada originalmente en la publicación que dió a conocer el método de retropropagación ([RHW86]).

Llamemos $\mathcal{A} = (\vec{x}[n], \vec{t}[n])$ al conjunto de ejemplos de que disponemos y a la salida esperada correspondiente. En general, tendremos un número grande de pares de entrenamiento al que llamaremos $N_{\mathcal{A}}$, aunque aquí nos restringimos a uno sólo (el primero, $\vec{x}[1], \vec{t}[1]$) para no complicar la notación. Ya hemos dicho que a los pares ($\vec{x}[n]$ y $\vec{t}[n]$) se les llama pares de entrenamiento porque constan de un patrón de entrada y de su correspondiente objetivo.

Si introducimos el vector $\vec{x}[1]$ compuesto de N_e componentes x_k con $k = 1, 2, ..., N_e$ en la capa de entrada de la red neuronal, obtendremos como resultado los valores y_k a la salida (donde k puede tomar cualquiera de los valores $k = 1, 2, ..., N_s$). Este resultado (que depende del valor de los pesos empleados) diferirá del esperado que hemos denominado $\vec{t}[1]$ (un vector con N_s componentes t_k, $k = 1, 2, ..., N_s$).

De entre las variadas formas de cuantificar el error cometido al obtener los y_k vamos a emplear el error cuadrático. Éste se define como:

$$E(\Omega) = \frac{1}{2} \cdot \sum_{n=1}^{N_{\mathcal{A}}} \|\vec{t}[n] - \vec{y}[n]\|^2 = \frac{1}{2} \cdot \sum_{n=1}^{N_{\mathcal{A}}} \sum_{k=1}^{N_s} (t_k - y_k)^2 \tag{2.5}$$

Esta fórmula hace explícita la dependencia del error con el conjunto de todos los pesos al que hemos dado en llamar Ω. El error depende de los pesos a través de su dependencia de las salidas y_j.

Como decíamos más arriba, vamos a realizar los cálculos para un único par de entrenamiento ($n = 1$) y comenzaremos por los pesos de la última capa. La forma más inmediata y sencilla de disminuir el error modificando los pesos consiste en actualizar los pesos siguiento la dirección opuesta al gradiente de la función $E(\Omega)$ como vimos en el capítulo anterior:

$$\Delta\omega_{kj} = -\alpha \cdot \frac{\partial E(\Omega)}{\partial \omega_{kj}}. \tag{2.6}$$

Regla delta

Para comprender esta fórmula (conocida como *regla delta*) es necesario entender el concepto de gradiente. No nos vamos a ocupar aquí de justificar que el gradiente de una función en un punto marca la dirección de máximo crecimiento de dicha función,

pero se puede encontrar en cualquier texto de cálculo elemental[a]. De igual manera, el gradiente cambiado de signo apuntará en la dirección de máxima disminución que es precisamente lo que estamos buscando: disminuir la función error. En el capítulo anterior nos hemos referido a esta estrategia de minimización como *descenso del gradiente*. La regla delta es un ejemplo de este tipo de estrategias. α es el parámetro, ya introducido en el capítulo anterior, que permite ajustar la velocidad de aprendizaje. Como vimos, es un número real que podemos ajustar para que el proceso de aprendizaje sea más o menos rápido (valores de α próximos a 1 darán velocidades de aprendizaje altas e inestabilidad y *viceversa* para valores próximos a 0). La elección de α está sujeta a condicionamientos que se tratarán en otro apartado, pero no es tan directa como la frase anterior pudiera dar a entender.

En lo que sigue, vamos a calcular el valor de $\Delta\omega_{11}$, es decir, la modificación del peso de la conexión entre la neurona $j = 1$ de la capa oculta y la neurona $k = 1$ de la capa de salida. El procedimiento de cálculo de un $\Delta\omega_{kj}$ (donde k es el subíndice de la neurona de salida y j el de la neurona oculta) cualquiera se puede obtener por generalización a partir del cálculo siguiente. Es importante señalar que el procedimiento que vamos a desarrollar a continuación es válido únicamente para pesos de conexiones entre la capa oculta y la capa de salida. El procedimiento para calcular las actualizaciones de los pesos de las conexiones de neuronas de la capa de entrada con neuronas de la capa oculta se mostrarán más adelante en esta misma sección. A partir de la ecuación 2.6 aplicando la regla de la cadena tenemos

$$\frac{\partial E}{\partial \omega_{kj}} = \frac{\partial E}{\partial y_k} \cdot \frac{\partial y_k}{\partial \omega_{kj}} \tag{2.7}$$

donde ya hemos omitido por razones de claridad la dependencia de E con Ω. El primer término de la derecha se puede obtener derivando parcialmente respecto a y_k la fórmula 2.5 con lo que se obtiene

$$\frac{\partial E}{\partial y_k} = -(t_k - y_k) = -\delta_k \tag{2.8}$$

donde hemos empleado la notación δ_k para referirnos a $(t_k - y_k)$. En lo que respecta al segundo término, tenemos que

$$\frac{\partial y_k}{\partial \omega_{kj}} = \frac{\partial g(\sum_{j=0}^{N_o} \omega_{kj} y_j)}{\partial \omega_{kj}} = g_k \cdot (1 - g_k) \cdot \frac{\partial (\sum_{j=0}^{N_o} \omega_{kj} y_j)}{\partial \omega_{kj}} = g_k \cdot (1 - g_k) \cdot y_j \tag{2.9}$$

donde, por claridad, hemos supuesto que $\kappa = 1$ y hemos sustituido la expresión de

$$g(a_k) = g(\sum_{j=0}^{N_o} \omega_{kj} y_j) \tag{2.10}$$

por g_k. Por lo tanto

$$\Delta\omega_{kj} = \alpha \cdot \delta_k \cdot g_k \cdot (1 - g_k) \cdot y_j \tag{2.11}$$

Vamos ahora a tratar de extender el formalismo a las conexiones entre neuronas de la capa de entrada y neuronas de la capa oculta. Para la modificación de cualquiera de esas conexiones tendremos

$$\Delta\omega_{ji} = -\alpha \cdot \frac{\partial E}{\partial \omega_{ji}} = -\alpha \cdot \frac{\partial E}{\partial y_j} \cdot \frac{\partial y_j}{\partial \omega_{ji}} \tag{2.12}$$

[a]De igual manera no se justificará la regla de la cadena de derivación por lo que se recomienda al lector que aproveche la ocasión para refrescar la memoria sobre todos estos aspectos del Cálculo.

Para calcular la primera derivada de la derecha tendremos en cuenta que $E(\Omega)$ no depende directamente de ningún y_j (es decir, no depende directamente de los valores de activación de las neuronas de la capa oculta). Efectivamente, la ecuación 2.5 nos indica que $E(\Omega)$ depende de los valores de activación de las neuronas de la capa de salida y_k y sólo a través de éstas e indirectamente, de los valores de y_j. Por lo tanto, será necesario aplicar la regla de la cadena para composición de funciones:

$$\frac{\partial E}{\partial y_j} = \sum_{k=1}^{N_s} \frac{\partial E}{\partial y_k} \cdot \frac{\partial y_k}{\partial y_j} \tag{2.13}$$

Puesto que para neuronas de la capa de salida, hemos empleado la notación

$$\frac{\partial E}{\partial y_k} = -\delta_k \tag{2.14}$$

ahora podemos, por analogía, llamar $-\delta_j$ a la derivada parcial de la función de error respecto al valor de activación de la neurona de la capa oculta y_j. Entonces,

$$\frac{\partial E}{\partial y_j} = \sum_{k=1}^{N_s} \frac{\partial E}{\partial y_k} \cdot \frac{\partial y_k}{\partial y_j} = -\delta_j \tag{2.15}$$

Ya conocemos que

$$\frac{\partial E}{\partial y_k} = -(t_k - y_k) = -\delta_k \tag{2.16}$$

y podemos calcular que

$$\frac{\partial y_k}{\partial y_j} = \frac{\partial g(\sum_{j'=0}^{N_o} \omega_{kj'} y_{j'})}{\partial y_j} = g_k \cdot (1 - g_k) \cdot \omega_{kj} \tag{2.17}$$

de donde

$$\frac{\partial E}{\partial y_j} = -\delta_j = -\sum_{k=1}^{N_s} g_k \cdot (1 - g_k) \cdot \delta_k \cdot \omega_{kj}. \tag{2.18}$$

Por otra parte,

$$\frac{\partial y_j}{\partial \omega_{ji}} = \frac{\partial g(\sum_{i=0}^{N_e} \omega_{ji} x_i)}{\partial \omega_{ji}} = g_j \cdot (1 - g_j) \cdot x_i \tag{2.19}$$

donde de nuevo se sobreentiende que

$$g_j = g(\sum_{i=0}^{N_e} \omega_{ji} x_i) \tag{2.20}$$

Combinando 2.18 y 2.19 con 2.12 obtenemos finalmente

$$\Delta\omega_{ji} = \alpha \cdot \delta_j \cdot g_j \cdot (1 - g_j) \cdot x_i \tag{2.21}$$

o, resumiendo todo el proceso, que

$$\Delta\omega_{ji} = -\alpha \cdot \frac{\partial E}{\partial \omega_{ji}} = -\alpha \cdot \frac{\partial E}{\partial y_j} \cdot \frac{\partial y_j}{\partial \omega_{ji}} = \alpha \cdot \delta_j \cdot g_j \cdot (1 - g_j) \cdot x_i \tag{2.22}$$

con

$$\delta_j = \sum_{k=1}^{N_s} g_k \cdot (1 - g_k) \cdot \delta_k \cdot \omega_{kj} \tag{2.23}$$

y

$$g_k = g(\sum_{j=0}^{N_o} \omega_{kj} y_j) \qquad g_j = g(\sum_{i=0}^{N_e} \omega_{ji} x_i). \tag{2.24}$$

Resumen de la notación empleada:

- Número de neuronas en cada capa: N_e para la capa de entrada, N_o para la capa oculta, N_s para la capa de salida.
- Vector de entrada a la red neuronal: $\vec{x}[n] = x_1, x_2, ..., x_{N_e}$.
- Objetos de la capa de entrada: subíndices i; de la capa oculta, j; de la de salida, k.
- Peso de la sinapsis que conecta la neurona i (capa de entrada) con la neurona j (capa oculta):ω_{ji}. Peso de la sinapsis que conecta la neurona j (capa oculta) con la neurona k (capa de salida):ω_{kj}.
- Suma ponderada:

$$a_j = \sum_{i=0}^{N_e} \omega_{ji} \cdot x_i \qquad a_k = \sum_{j=0}^{N_o} \omega_{kj} \cdot y_j.$$

- Función de activación: $g_j = g(a_j)$ (para neuronas de la capa oculta) o $g_k = g(a_k)$ (neuronas de salida).
- Salida de una neurona (y_j corresponde a la salida de la neurona j de la capa oculta e y_k a la neurona k de la capa de salida):

$$y_j = g(a_j) = g(\sum_{i=0}^{N_e} \omega_{ij} \cdot x_i) \qquad y_k = g(a_k) = g(\sum_{j=0}^{N_o} \omega_{kj} \cdot y_j).$$

- Conjunto $\mathcal{A}$ de entrenamiento: conjunto de $N_{\mathcal{A}}$ pares compuestos por un vector de entrada $\vec{x}[n]$ y su correspondiente vector de salida esperado $\vec{t}[n]$ con $n = 1, 2, ..., N_{\mathcal{A}}$.
- Función de error: $E(\Omega)$.
- Coeficiente de aprendizaje: α.

Ejemplo 1 (Retropropagación del error): Diseñar una red de tres capas con una sola neurona en la capa de entrada, 5 neuronas en la capa oculta y una neurona en la capa de salida y con funciones de transferencia tipo sigmoide en la capa oculta y lineal en la de salida, para el siguiente conjunto de entrenamiento:

	$x[n]$	$t[n]$		$x[n]$	$t[n]$
n=1	-1.0	-0.9602	n=12	0.1	-0.3930
n=2	-0.9	-0.5770	n=13	0.2	-0.1647
n=3	-0.8	-0.0729	n=14	0.3	0.0988
n=4	-0.7	0.3771	n=15	0.4	0.3072
n=5	-0.6	0.6405	n=16	0.5	0.3960
n=6	-0.5	0.6600	n=17	0.6	0.3449
n=7	-0.4	0.4609	n=18	0.7	0.1816
n=8	-0.3	0.1336	n=19	0.8	-0.0312
n=9	-0.2	-0.2013	n=20	0.9	-0.2189
n=10	-0.1	-0.4344	n=21	1.0	-0.3201
n=11	0.0	-0.5000			

Nota: Supongamos que la velocidad de aprendizaje α vale 0.1.

Solución: Es muy importante tener en cuenta que en la resolución del problema vamos a introducir una novedad (ya anunciada al presentar la función de activación sigmoidea) respecto al formalismo matemático descrito en esta sección: el enunciado dice explícitamente que la neurona de la capa de salida procesa la suma ponderada de sus entradas con una función de transferencia lineal y no con una sigmoide. Esto significa que $g_k = g(a_k) = a_k$ o, de forma más detallada y omitiendo la refencia a k:

$$g_k = g(\sum_{j=0}^{5} \omega_{kj} y_j) = \sum_{j=0}^{5} \omega_{kj} y_j.$$

Por lo tanto, la ecuación (2.9) deberá ser modificada:

$$\frac{\partial y_k}{\partial \omega_{kj}} = \frac{\partial g(\sum_{j'=0}^{5} \omega_{kj'} y_{j'})}{\partial \omega_{kj}} = \frac{\partial (\sum_{j'=0}^{5} \omega_{kj'} y_{j'})}{\partial \omega_{kj}} = y_j.$$

Finalmente, la fórmula para la actualización de los pesos de la capa de salida será:

$$\Delta \omega_{kj} = -\alpha \cdot \frac{\partial E(\Omega)}{\partial \omega_{kj}} = -\alpha \cdot \frac{\partial E}{\partial y_k} \cdot \frac{\partial y_k}{\partial \omega_{kj}} = \alpha \cdot \delta_k \cdot y_j \tag{2.25}$$

Además, habrá que modificar la ecuación de actualización de los pesos de la capa oculta (ecuación 2.17) que quedará

$$\frac{\partial y_k}{\partial y_j} = \frac{\partial g(\sum_{j'=0}^{N_o} \omega_{kj'} y_{j'})}{\partial y_j} = \frac{\partial (\sum_{j'=0}^{N_o} \omega_{kj'} y_{j'})}{\partial y_j} = \omega_{kj} \tag{2.26}$$

de donde, para la neurona j de la capa oculta tendremos

$$\frac{\partial E}{\partial y_j} = -\delta_j = -\sum_{k=0}^{N_s} \delta_k \cdot \omega_{kj}. \tag{2.27}$$

Dado que $N_s = 1$ podemos sustituir el subíndice k en todas las expresiones anteriores por un 1. La ecuación 2.22 no sufre ningún cambio salvo que ahora δ_j viene dada por la ecuación 2.27.

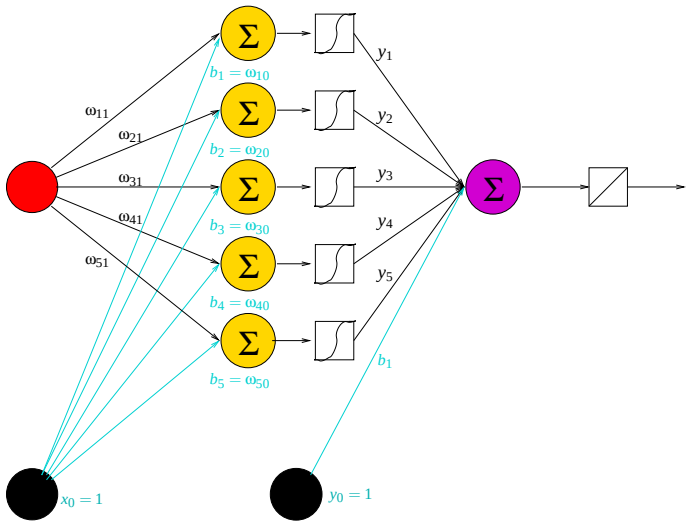


Figura 2.5: Red neuronal del ejemplo.

El esquema de nuestra red neuronal se muestra en la figura 2.5 en el que hemos añadido neuronas adicionales con subíndice cero, cuyas salidas x_0 e y_0 son siempre iguales a 1 y cuyos pesos son precisamente las polarizaciones que queremos introducir (ver capítulo anterior). Para el caso de funciones de activación no sigmoideas, la forma de introducir las polarizaciones es la misma: añadir una neurona ficticia de salida igual a 1 en la capa anterior e interpretar los pesos de conexión como las polarizaciones de las funciones de transferencia correspondientes.

Teniendo todo esto en cuenta y suponiendo una inicialización de los pesos y polarizaciones aleatoria ya podemos empezar a iterar el algoritmo de retropropagación de los errores. Sean los pesos de las sinapsis de la neurona de entrada con las de la capa oculta los siguientes: En lo que sigue se va a detallar el primer paso del procedi-

Entrada→Oculta		Oculta→Salida	
$\omega_{11} = -3{,}5$	$\omega_{10} = b_1 = -0{,}8155$	$\omega_{11} = +0{,}0594$	$\omega_{10} = b_1 = -0{,}1650$
$\omega_{21} = -3{,}5$	$\omega_{20} = b_2 = +0{,}1359$	$\omega_{12} = +0{,}3423$	
$\omega_{31} = +3{,}5$	$\omega_{30} = b_3 = +2{,}3168$	$\omega_{13} = -0{,}9846$	
$\omega_{41} = +3{,}5$	$\omega_{40} = b_4 = -3{,}2580$	$\omega_{14} = -0{,}2332$	
$\omega_{51} = +3{,}5$	$\omega_{50} = b_5 = -3{,}1258$	$\omega_{15} = -0{,}8663$	

miento iterativo de cálculo de los pesos de la red. El algoritmo de retropropagación del error consistirá en reproducir los cálculos descritos hasta que la función suma del error cuadrático medio sea menor que un cierto valor que consideremos suficiente.

1. El primer paso consiste en calcular y_j para cada una de las neurona de la capa oculta. Como se puede comprobar, las neuronas de la capa de entrada no son propiamente neuronas en la medida en que no procesan la información recibida sino que se limitan a transmitirla a las neuronas de la capa siguiente. Dada la primera entrada $x = -1$, el valor de y_j para cada neurona de la capa oculta será

$$\vec{a} = \begin{pmatrix} a_1 = (1 \times -0{,}8155) + (-3{,}5 \times -1) = +2{,}6845 \\ a_2 = (1 \times +0{,}1359) + (-3{,}5 \times -1) = +3{,}6359 \\ a_3 = (1 \times +2{,}3168) + (+3{,}5 \times -1) = -1{,}1832 \\ a_4 = (1 \times -3{,}2580) + (+3{,}5 \times -1) = -6{,}7580 \\ a_5 = (1 \times -3{,}1258) + (+3{,}5 \times -1) = -6{,}6258 \end{pmatrix}$$

$$\vec{y} = g(\vec{a}) = \begin{pmatrix} g(+2{,}6845) \\ g(+3{,}6359) \\ g(-1{,}1832) \\ g(-6{,}7580) \\ g(-6{,}6258) \end{pmatrix} = \begin{pmatrix} 0{,}9361 \\ 0{,}9743 \\ 0{,}2345 \\ 0{,}0012 \\ 0{,}0013 \end{pmatrix}$$

Donde el primer elemento del vector corresponde a la primera neurona de la capa oculta, el segundo a la segunda neurona y así sucesivamente.

2. El segundo paso consiste en calcular la salida de la única neurona de la tercera capa o capa de salida para el valor de entrada $x = -1$. Su valor sería

$$y_1 = g(a_1) = g(-0{,}0082) = -0{,}0082^{\text{b}}$$

3. El error cometido por la red neuronal para el valor de entrada $x = -1$ es

$$\delta_1 = (-0{,}9602 + 0{,}0082) = -0{,}9520$$

donde -0.9602 es el valor de la salida esperado para dicha entrada (ver enunciado del problema).

4. Recalcular los pesos de la segunda capa: De acuerdo con la fórmula 2.25[c]

$$\begin{aligned} \Delta\omega_{11} &= \alpha \cdot \delta_1 \cdot y_1 = 0{,}1 \cdot (-0{,}9520) \cdot 0{,}9361 = -0{,}0891 \\ \Delta\omega_{21} &= 0{,}1 \cdot (-0{,}9520) \cdot 0{,}9743 = -0{,}0928 \\ \Delta\omega_{31} &= 0{,}1 \cdot (-0{,}9520) \cdot 0{,}2345 = -0{,}0223 \\ \Delta\omega_{41} &= 0{,}1 \cdot (-0{,}9520) \cdot 0{,}0012 = -0{,}0001 \\ \Delta\omega_{51} &= 0{,}1 \cdot (-0{,}9520) \cdot 0{,}0013 = -0{,}0001 \end{aligned}$$

5. Recalcular los pesos de la primera capa: de acuerdo con las ecuaciones 2.22 y 2.27 tendremos

$$\begin{aligned} &\Delta\omega_{ji} = \alpha \cdot \delta_j \cdot g_j \cdot (1 - g_j) \cdot x_i \\ &\Delta\omega_{11} = 0{,}1 \cdot (-0{,}9520 \cdot 0{,}0594) \cdot g(2{,}6845) \cdot (1 - g(2{,}6845)) \cdot (-1) \\ &\Delta\omega_{21} = 0{,}1 \cdot (-0{,}9520 \cdot 0{,}3423) \cdot g(3{,}6359) \cdot (1 - g(3{,}6359)) \cdot (-1) \\ &\Delta\omega_{31} = 0{,}1 \cdot (-0{,}9520 \cdot -0{,}9846) \cdot g(-1{,}1832) \cdot (1 - g(-1{,}1832)) \cdot (-1) \\ &\Delta\omega_{41} = 0{,}1 \cdot (-0{,}9520 \cdot -0{,}2332) \cdot g(-6{,}7580) \cdot (1 - g(-6{,}7580)) \cdot (-1) \\ &\Delta\omega_{51} = 0{,}1 \cdot (-0{,}9520 \cdot -0{,}8663) \cdot g(-6{,}6258) \cdot (1 - g(-6{,}6258)) \cdot (-1) \end{aligned}$$

[b] Recordad que la función de activación o transferencia de la neurona de salida es lineal y que a_1 se calcula según

$a_1 = 1 \cdot -0{,}1650 + 0{,}0594 \cdot 0{,}9361 + 0{,}3423 \cdot 0{,}9743 - 0{,}9846 \cdot 0{,}2345 - 0{,}2332 \cdot 0{,}0012 - 0{,}8663 \cdot 0{,}0013 = -0{,}0082$

.

[c] Al reproducir estos cálculos tengan en cuenta los errores de redondeo.

Para $x = -0{,}9$ repetiríamos el mismo procedimiento sustituyendo -1 por -0.9 en la entrada y recalculando las salidas de todas las neuronas de las capas oculta y de salida y los pesos de las conexiones entre todas las capas. Así sucesivamente para los 21 valores que van desde -1 hasta 1 en intervalos de 0.1.

Este procedimiento general, admite pequeñas variaciones que serán tratadas en secciones posteriores. Baste recordar aquí que, además de la posibilidad de aplicar las correcciones de los pesos obtenidas para cada par de entrenamiento inmediatamente después de realizado el cálculo (entrenamiento secuencial), existe la posibilidad de ir acumulando las modificaciones y aplicarlas sumadas al final del bloque (entrenamiento por lotes).

Al final de cada actualización se compara el error obtenido con el error que nos hayamos fijado como meta. Si aquél es menor que éste se detiene la iteración y se fijan los pesos.

2.2. Aspectos metodológicos del entrenamiento de una red neuronal.

En lo que sigue, vamos a revisar algunos aspectos del proceso de aprendizaje de especial relevancia. El primero de ellos se refiere a la forma óptima en que debemos presentar los datos a la red.

2.2.1. Preprocesado de las variables de entrada/salida.

Normalización/Estandarización

En principio, una red neuronal como las que estamos estudiando en este capítulo no es más que un mapa entre un espacio de entrada y uno de salida. Deseamos que, a un vector del espacio de entrada, la red le haga corresponder uno del de salida donde cada uno de esos espacios está compuesto de variables dotadas de significados precisos para el creador de la red. Por ejemplo, si quisiéramos identificar números a partir de imágenes digitales, las variables de entrada serían los valores de intensidad de los píxeles de la imagen.

Dada la naturaleza de las redes neuronales, no se puede descartar la posibilidad de entrenar una red con los datos de entrada/salida tal y como se definen en la especificación del problema y obtener un rendimiento que cumpla las expectativas iniciales de la fase de diseño. Sin embargo, podemos pensar en muchas circunstancias en las que una transformación previa de las variables mejora el rendimiento de la red o incluso es imprescindible para su correcto funcionamiento. La transformación más sencilla es la normalización o estandarización de las variables de entrada. Volvamos al caso del conjunto de entrenamiento $\mathcal{A}$ de $N_{\mathcal{A}}$ elementos $\vec{x}[n]$ con $n = 1, 2 \ldots N_{\mathcal{A}}$. Podemos definir la media de cada variable i de los vectores de entrada $\vec{x}$ del conjunto de entrenamiento como

$$\vec{x_i} = \frac{1}{N_{\mathcal{A}}} \sum_{n=1}^{N_{\mathcal{A}}} x_i[n] \tag{2.28}$$

y la varianza σ_i^2 como

$$\sigma_i^2 = \frac{1}{N_{\mathcal{A}} - 1} \sum_{n=1}^{N_{\mathcal{A}}} (x_i[n] - \vec{x}_i)^2. \tag{2.29}$$

Entonces, la transformación

$$x_i[n] \rightarrow \frac{x_i[n] - \vec{x}_i}{\sigma_i} \tag{2.30}$$

da como resultado que la variable *i* tome en todo el conjunto de entrenamiento valores cuya media es cero y cuya desviación estándar es 1. Si repetimos la transformación para todas las variables del espacio de entrada $x_1, x_2, ..., x_{N_e}$, tendremos que todas las variables tendrán las mismas propiedades estadísticas mencionadas.

Es cierto que por la propia definición del perceptrón, esta transformación es innecesaria porque la primera capa de neuronas realiza una operación de reescalado en la que se podría incluir la normalización. Sin embargo, de esta forma conseguimos que todas las entradas de la red tengan valores del mismo orden de magnitud y, por lo tanto, también los valores de los pesos estarán en el mismo rango. Finalmente, la inicialización aleatoria de los pesos con valores dentro de esos rangos dará lugar a una convergencia rápida del proceso de aprendizaje y evitará el estancamiento en mínimos locales. ¿Sabría explicar intuitivamente por qué?

El lector atento ya habrá observado que el proceso de normalización reduce la cantidad de información que pasamos a la red neuronal puesto que hemos eliminado las diferencias de magnitud entre unas variables y otras. Es el diseñador de la red neuronal quien debe evaluar la importancia de esa información y si los beneficios que obtenemos en el proceso de aprendizaje gracias a la normalización justifican esa pérdida.

Una última consideración sobre otro tipo de cambios de escala. La normalización propuesta anteriormente admite diversas variantes no todas de la misma utilidad. En particular, una transformación muy popular consiste en hacer que las componentes de los vectores de entrada se encuentren en el intervalo [0,1]. No existe ningún beneficio en ello y es en general preferible que la escala esté centrada en cero, por ejemplo, llevando los valores de entrada al intervalo [-1,1].

Compleción de patrones

Otro tipo de preprocesado es la compleción de los patrones o vectores de entrada. Pensemos, por centrar las ideas, en un caso de clasificación. Pues bien, en muchas ocasiones, la recolección de un conjunto completo de ejemplos de las clases que queremos reconocer con la red neuronal es una tarea complicada y no son inhabituales situaciones en las que no disponemos de suficientes casos. Pensemos que el conjunto de entrenamiento no sólo debe contener instancias de todos los casos que queremos tratar sino que ha de hacerlo en proporciones similares a las que esperamos que se encuentre la red durante su funcionamiento real. Ello implica el aprovechamiento de ejemplos que serían desechados si dispusiésemos de una gran cantidad de casos, en particular algunos cuyos vectores de entrada son incompletos. Existen varias formas de completarlos con el fin de poder utilizarlos como ejemplos de entrenamiento. La primera de ellas consiste en dar a una variable ausente de un patrón o vector de entrada el valor de la media de los valores de esas variables en los casos de entrenamiento en los que sí están presente. Otra posibilidad es realizar una regresión con los valores disponibles y rellenar los vacíos con las predicciones de la función obtenida.

Ambas posibilidades sesgan el patrón de entrada de forma no deseada, aunque evidentemente la segunda presta más atención a la parte del vector de que sí disponemos. Su principal desventaja es que emplear una función de regresión implica obviamente infravalorar la covarianza de los datos. La solución más deseable pasa por buscar los valores que maximizan la verosimilitud (*likelihood*, en inglés) y emplear un algoritmo EM (expectation-maximization, ver [GJ95]).

Selección de características. Análisis de componentes principales.

Para terminar esta sección, vamos a mencionar otra posible etapa del preprocesado de los datos que consiste seleccionar de entre todas las posibles variables de entrada a la red que hemos considerado inicialmente un subconjunto mínimo imprescindible para la realizar la tarea que hemos encomendado a la red. En la práctica, el subconjunto no es mínimo sino un compromiso entre un número manejable (no excesivamente grande) de variables y un rendimiento de la red aceptable. Obviamente si disminuimos demasiado el número de variables de entrada nos encontraremos con redes neuronales muy manejables, con pocas entradas y ciclos de aprendizaje rápidos pero a costa de disminuir el rendimiento de la red por haber prescindido de variables importante para la tarea. En el extremo contrario, podemos incluir demasiadas variables de entrada, algunas fuertemente correlacionadas y, por tanto, redundantes, lo que resulta en redes a las que proporcionamos toda la información de que disponemos pero que presentan ciclos excesivamente largos y a veces un rendimiento sub-óptimo precisamente porque se le hace llegar información de forma indiscriminada.

Azote de la dimensionalidad

En la bibliografía citada al final del texto se cita en numerosas ocasiones lo que en inglés se conoce como *curse of dimesionality* [BD62] y que podríamos traducir al castellano como la maldición de la dimensionalidad (aunque podríamos traducirlo alternativamente como el *azote de la dimensionalidad*, conservando un sentido parecido). La expresión hace referencia precisamente a los efectos perniciosos que la sobreabundancia de entradas tiene sobre el rendimiento de una red neuronal. Para explicar intuitivamente su significado, volvamos a la descripción de una red neuronal dada en el capítulo anterior. Allí veíamos cómo un problema de regresión[d] era equivalente a encontrar una función

$$\vec{f} : \mathbb{R}^{N_e} \rightarrow \mathbb{R}^{N_s}. \tag{2.31}$$

donde $\mathbb{R}^{N_e}$ era el espacio de entrada y $\mathbb{R}^{N_s}$ el de salida. Lo que nos interesa aquí es que para reconstruir esa función $\vec{f}$ (es decir, para entrenar la red neuronal) necesitamos ejemplos que cubran la mayor cantidad posible del espacio de entrada $\mathbb{R}^{N_e}$. La representación será tanto mejor cuanto menor sea la distancia entre cualquier caso nuevo que se le presente a la red y el ejemplo más próximo empleado en el entrenamiento. Pues bien, se puede probar que el número de ejemplos crece exponencialmente cuanto mayor es la dimensión N_e del espacio de entrada si queremos que esa distancia mínima entre un vector cualquiera y el ejemplo más próximo se mantenga constante. Ese crecimiento exponencial implica que la complejidad de la red y los tiempos de entrenamiento crecen muy rápidamente con la dimensionalidad del espacio de entrada además de requerir conjuntos de entrenamiento imposibles de conseguir en la práctica. Por ello, en muchos casos se requiere disminuir la dimensión de dicho espacio eliminando algunas variables de entrada y combinándolas entre sí.

[d] En realidad, los problemas de clasificación también pueden ser interpretados como un caso especial de regresión y viceversa.

Existen varias formas de seleccionar las variables o combinaciones de variables que queremos utilizar como entrada a la red neuronal pero aquí nos vamos a restringir a un formalismo general de gran utilidad en muchos campos del análisis de datos: el análisis de componentes principales o transformación de Karhunen-Loève. Para exponer dicha transformación necesitaremos repasar algunos conceptos de Álgebra propios de cualquier curso de Matemáticas de primeros cursos de carrera técnica. Entonces se estudiaba que un vector de N_e componentes reales se puede expresar como una combinación lineal de los vectores de una base de $\mathbb{R}^{N_e}$

$$\vec{x}[n] = \sum_{i=1}^{N_e} z_i[n] \cdot \vec{u}_i \tag{2.32}$$

donde $\vec{u}_i$ es la base vectorial de $\mathbb{R}^{N_e}$ y $z_i[n]$ son los coeficientes del vector de entrada $\vec{x}[n]$ en dicha base. La base fundamental del método es seleccionar una base de $\mathbb{R}^{N_e}$ en la que cada sumando de la suma de términos de 2.32 sea de menor valor absoluto que el anterior. De esta forma, podemos aproximar cualquier vector $\vec{x}[n]$ del conjunto de entrenamiento por la suma por los M primeros términos correspondientes a dicho vector más los $N_e - M$ restantes con los mismos coeficientes para todo el conjunto de entrenamiento. Si inicialmente necesitábamos los N_e valores de entrada para especificar el ejemplo $\vec{x}[n]$ ahora nos bastarán los M primeros porque las componentes del resto son iguales para todos los vectores de entrada del conjunto de entrenamiento, lo que implica una reducción efectiva de la dimensión del espacio de entradas. En resumen tenemos que

$$\vec{x}[n] \sim \tilde{x}[n] = \sum_{i=1}^{M} z_i[n] \cdot \vec{u}_i + \sum_{i=M+1}^{N_e} b_i \cdot \vec{u}_i \tag{2.33}$$

donde b_i con $M < i \leq N_e$ son los $N_e - M$ coeficientes constantes para todo el conjunto de entrenamiento.

Al hacer constantes los últimos coeficientes del desarrollo estaremos cometiendo un error

$$\vec{x}[n] - \tilde{x}[n] = \sum_{i=M+1}^{N_e} (z_i[n] - b_i) \cdot \vec{u}_i. \tag{2.34}$$

Así pues, se trata de reducir la dimensionalidad del espacio de entrada desde N_e hasta M encontrando unos valores b_i y una base de vectores en la que el error cometido sea mínimo. Minimizando el error cometido y empleando multiplicadores de Lagrange se obtiene que

$$b_i = \frac{1}{N_e} \sum_{n=1}^{N_e} z_i[n] \tag{2.35}$$

y que la base u_i que necesitamos es la base de autovectores o vectores propios de la matriz covarianza[e] Σ

$$\Sigma = \sum_{n=1}^{N_{\mathcal{A}}} (\vec{x}[n] - \vec{x}) \cdot (\vec{x}[n] - \vec{x})^T. \tag{2.36}$$

En resumen, dado un conjunto de entrenamiento $\mathcal{A}$ compuesto de $N_{\mathcal{A}}$ vectores $\vec{x}[n]$, el procedimiento consiste en:

[e]En la definición de la matriz covarianza dada por la fórmula 2.36 no aparece un factor $\frac{1}{N_{\mathcal{A}}-1}$ que el lector puede encontrar en algunos manuales en la bibliografía. En este caso hemos preferido mantener la definición de uno de los mejores textos de este área ([Bis95]) en la que dicho factor se ha omitido.

1. calcular la media de todos los vectores $\vec{x} = \frac{1}{N_{\mathcal{A}}} \sum_{n=1}^{N_{\mathcal{A}}} \vec{x}[n]$;
2. calcular la matriz de covarianza según 2.36 y sus valores y vectores propios;
3. seleccionar los M vectores propios correspondientes a los M valores propios de mayor valor absoluto;
4. proyectar los vectores $\vec{x}[n]$ del conjunto de entrenamiento en la nueva base de vectores propios.

Ejemplo 2 (Karhunen-Loève): Supongamos que queremos entrenar una red neuronal para que distinga el siguiente conjunto de vectores a partir de una única característica:

	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]	x[8]	x[9]	x[10]
Componente x	2.5	0.5	2.2	1.9	3.1	2.3	2.0	1.0	1.5	1.1
Componente y	2.4	0.7	2.9	2.2	3.0	2.7	1.6	1.1	1.6	0.9
Clase	A	B	A	A	A	A	B	B	B	B

El vector medio de este conjunto de entrenamiento es $(1{,}81, 1{,}91)$ y, puesto que los vectores de entrada son bidimensionales, la matriz de covarianza tendrá dimensiones 2×2 y será

$$\begin{bmatrix} 5{,}549 & 5{,}539 \\ 5{,}539 & 6{,}449 \end{bmatrix} \tag{2.37}$$

Para calcular los autovalores/autovectores debemos en primer lugar calcular el determinante

$$\begin{vmatrix} 5{,}549 - \lambda & 5{,}539 \\ 5{,}539 & 6{,}449 - \lambda \end{vmatrix} \tag{2.38}$$

lo que se traduce en obtener las raices del polinomio de segundo grado $\lambda^2 - 11{,}998\lambda + 5{,}105$. Las raices del polinomio son los autovalores que estamos buscando, en este caso 11.556 y 0.442. Para obtener los autovectores debemos recurrir a la definición de autovector. Según ésta, tendremos que

$$\begin{bmatrix} 5{,}549 & 5{,}539 \\ 5{,}539 & 6{,}449 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \lambda \cdot \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} \tag{2.39}$$

Para $\lambda = 11{,}556$ la ecuación anterior se convierte en el sistema de ecuaciones

$$5{,}549 \cdot v_1 + 5{,}539 \cdot v_2 = 11{,}556 \cdot v_1 \tag{2.40}$$

$$5{,}539 \cdot v_1 + 6{,}449 \cdot v_2 = 11{,}556 \cdot v_2 \tag{2.41}$$

y su solución es el vector propio correspondiente (-0.678, -0.735); el segundo autovalor es 0.442 y su vector propio (-0.735, 0.678). Dado que la dirección de máxima varianza viene dada por el autovector del autovalor de mayor valor absoluto, ésta vendrá dada por primer vector propio. Recomendamos encarecidamente al lector que represente gráficamente los vectores de entrada antes y después de sustraerles el vector medio y que trace los vectores propios sobre dicha gráfica. Intente contestar a las siguientes preguntas:

1. ¿Qué dirección marca el vector propio correspondiente al mayor autovalor?
2. ¿Qué ocurre cuando proyectamos los vectores de entrada sobre dicho autovector?

3. Al deshacernos del segundo vector propio ¿Estamos perdiendo información esencial para la tarea de clasificación?

4. El método de Karhunen-Loève ¿tiene en cuenta la clase a la que pertenece cada vector de entrada?

5. ¿Se le ocurre otra división del conjunto de entrenamiento en clases para la que el método de Karhunen-Loève no sirva o implique una pérdida total de la posibilidad de clasificar?

El equipo docente de la asignatura ha escrito un pequeño programa en c++ para generar diferentes conjuntos de datos que ilustren la aplicación de la transformación de Karhunen-Loève. Con él, el alumno puede comprobar qué tipo de reducción de la dimensionalidad realiza la transformación y, sobre todo, que hay casos en los que esa reducción no se debe aplicar porque elimina toda posibilidad de resolver correctamente el problema planteado. El programa en cuestión puede ser descargado de la página web de la asignatura y hace uso de la librería gsl (GNU Scientific Library). Para compilarlo en un sistema unix utilice el siguiente comando:

```
g++ gen-KL.cpp -lgsl -lgslcblas -o KL
```

KL genera un fichero con 200 puntos, 100 pertenecientes a una clase que denomina A, y 100 a la clase B. Cada punto viene dado por dos componentes, x_1 y x_2, extraidas de una distribución de probabilidad gaussiana bidimensional. **KL** espera ocho parámetros en línea de comandos: el coeficiente de correlación de la distribución de probabilidad gaussiana para la clase A, ídem para la clase B; la varianza de la primera componente para los miembros de la clase A, ídem para la clase B; la varianza de la segunda componente para los miembros de la clase A, ídem para la clase B; y el desplazamiento horizontal y vertical de los centroides de las dos clases.

Además de generar los datos, **KL** calcula la matriz de covarianza, sus autovalores y autovectores y, éstos últimos, los escribe al final del fichero de datos. El primer autovector corresponde al autovalor de mayor valor absoluto y, por lo tanto, a la dirección de máxima varianza.

Recomendamos al alumno que genere diferentes conjuntos de datos variando los parámetros de entrada al programa. Puede representarlos empleando gnuplot y el fichero de comandos que se proporciona en la misma dirección web que el programa **KL**.

El lector deber recordar que el objetivo de la transformación de Karhunen-Loève (en el contexto en el que la hemos presentado aquí, es reducir la dimensionalidad del espacio de entrada. Puesto que los puntos que genera **KL** están en un espacio bidimensional (sólo tienen dos componentes) la reducción de dimensionalidad sólo puede ser de dos variables a una mediante la proyección de los puntos de entrada sobre la dirección que marca el autovector asociado al mayor autovalor. Dicha proyección se realiza multiplicando escalarmente el vector de entrada por el autovector. Finalmente, podemos preguntarnos si, tras la reducción de dimensionalidad, los puntos en espacio reducido (unidimensional) son separables. Para ello, recomendamos representar el resultado de la proyección en gnuplot con el comando

```
plot 'KL.points' u ($1*av11+$2*av12) index 0 ,\\
'KL.points' u ($1*av11+$2*av12) index 1
```

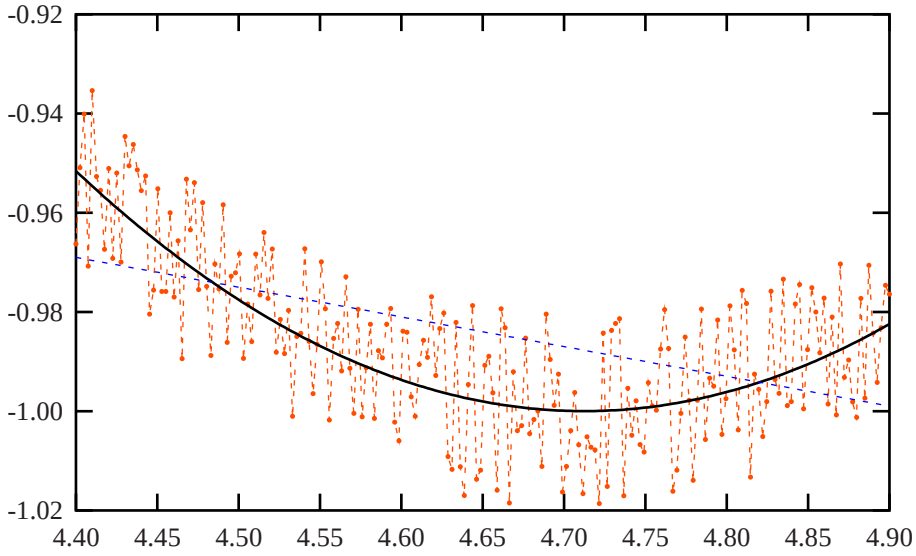


Figura 2.6: Ejemplo de problema de regresión en el que, de nuevo, los puntos naranja son pares de entrenamiento y se desea obtener la función generatriz representada en trazo negro continuo.

donde av11 es la primera componente del autovector asociado al mayor autovalor y av12 su segunda componente. Además de en el fichero de salida KL.points, los valores de los autovalores y autovectores asociados son mostrados en pantalla por el programa. La página web desde la que se puede descargar el código fuente sugiere algunos valores de prueba para los parámetros del programa.

2.2.2. Descomposición del error. Definición de términos. El compromiso entre sesgo y varianza

Considérese la figura 2.6 en la que se muestra un problema de regresión parecido al de la figura 1.12. La curva continua representa una función seno y los puntos alrededor de la curva han sido obtenidos a partir dicha función añadiendo ruido gaussiano.

El sobreajuste

En dicha figura se ejemplifican los dos casos extremos a los que nos puede llevar el entrenamiento inadecuado de una red neuronal. Por un lado, un número insuficiente de neuronas en la red neuronal o un entrenamiento insuficiente proporcionan funciones de regresión suaves como la recta discontinua azul pero que resultan en un ajuste pobre a los datos de partida: no son suficientemente flexibles como para ajustarse a los datos; en el otro extremo, un entrenamiento excesivo (*overfitting* en inglés) de una red suficientemente compleja hará que la curva ajustada (trazo naranja discontinuo) pase por todos los puntos empleados en la regresión (dando lugar por lo tanto a un error nulo para el conjunto de entrenamiento), pero alejándose al mismo tiempo de la curva suave que originó los datos distorsionados por el ruido (línea continua). El objetivo de un entrenamiento adecuado es el de encontrar un punto de compromiso entre ambas soluciones, esto es, la curva sinusoide continua que dió origen a los datos. Una red sobreentrenada ajusta los datos de entrenamiento perfectamente (error cero) pero, ante

un dato nuevo no contenido en el conjunto de entrenamiento, hará predicciones más erradas que la curva suave. Lo que necesitamos, por tanto, es que la red neuronal no se limite a reproducir exactamente el conjunto de entrenamiento, sino que a partir de él sea capaz de generalizar y realizar predicciones correctas para nuevas entradas. Vamos a ver por qué a este compromiso se lo conoce como el compromiso sesgo-varianza (*bias-variance trade-off* en inglés).

A continuación vamos a realizar un experimento mental que nos va a permitir conocer algo más sobre la naturaleza del error que va a afectar a cualquier red neuronal que construyamos. Para ello, necesitamos imaginar que en lugar de un solo conjunto de entrenamiento, tenemos varios con el mismo número de pares $(\vec{x},\vec{t})$ cada uno.

Supongamos, como hemos hecho anteriormente, que empleamos una función del error basada en la suma de cuadrados (ecuación 2.5 que reproducimos a continuación)

$$E(\Omega) = \frac{1}{2}\cdot\sum_{n=1}^{N_{\mathcal{A}}} \|\vec{t}[n]-\vec{y}[n]\|^2 = \frac{1}{2}\cdot\sum_{n=1}^{N_{\mathcal{A}}}\sum_{k=1}^{N_s}(t_k-y_k)^2$$

donde recordamos que $\vec{t}$ representa un vector salida del par de entrenamiento e $\vec{y}$ representa el vector obtenido por la red neuronal para la entrada correspondiente al par al que pertenece $\vec{t}$.

Esta ecuación esta basada en la suposición de que trabajamos con un conjunto de entrenamiento $\mathcal{A}$ de $N_{\mathcal{A}}$ elementos y de que la red neuronal tiene un espacio de salida de dimensión N_s. Para simplificar, tomemos $N_s = 1$ con lo que

$$E(\Omega) = \frac{1}{2}\cdot\sum_{n=1}^{N_{\mathcal{A}}} \|\vec{t}[n]-\vec{y}[n]\|^2 = \frac{1}{2}\cdot\sum_{n=1}^{N_{\mathcal{A}}}(t-y)^2 \tag{2.42}$$

Llamemos $\mathcal{D}$ al conjunto de todos los conjuntos de entrenamiento de nuestro experimento mental (todos ellos, como decíamos, de $N_{\mathcal{A}}$ elementos). A partir de cada conjunto de entrenamiento, obtendríamos una red neuronal distinta que generaría un valor de y diferente. Pues bien, para cada uno de ellos podemos escribir la suma a la derecha de 2.42 como

$$\sum_{n=1}^{N_{\mathcal{A}}}(t-y)^2 = \sum_{n=1}^{N_{\mathcal{A}}}(y-t)^2 = \sum_{n=1}^{N_{\mathcal{A}}}(y-\mathcal{E}_{\mathcal{D}}[y]+\mathcal{E}_{\mathcal{D}}[y]-t)^2 \tag{2.43}$$

donde $\mathcal{E}_{\mathcal{D}}[y]$ representa el valor esperado de la salida de la red neuronal para el conjunto de salidas generadas por cada conjunto $\mathcal{A}$ en $\mathcal{D}$. Esto es, dado que para cada vector de entrada x tenemos $N_{\mathcal{D}}$ salidas y de cada una de las $N_{\mathcal{D}}$ redes neuronales obtenidas a partir de los $N_{\mathcal{D}}$ conjuntos de entrenamiento, podemos pensar en el valor esperado de y como la media de dichos $N_{\mathcal{D}}$ valores. Reordenando los términos y realizando sencillas manipulaciones obtenemos

$$\sum_{n=1}^{N_{\mathcal{A}}}(y-\mathcal{E}_{\mathcal{D}}[y]+\mathcal{E}_{\mathcal{D}}[y]-t)^2 = \sum_{n=1}^{N_{\mathcal{A}}}(y-\mathcal{E}_{\mathcal{D}}[y])^2+(\mathcal{E}_{\mathcal{D}}[y]-t)^2+2(y-\mathcal{E}_{\mathcal{D}}[y])(\mathcal{E}_{\mathcal{D}}[y]-t) \tag{2.44}$$

Si ahora queremos calcular el valor esperado del error (que podría ser, para fijar ideas, la media de $E(\Omega)$ para los $N_{\mathcal{D}}$ conjuntos de entrenamiento) tendríamos

$$\begin{aligned}\mathcal{E}_{\mathcal{D}}[E(\Omega)] &= \frac{1}{2}\cdot\mathcal{E}_{\mathcal{D}}[\sum_{n=1}^{N_{\mathcal{A}}}(y-\mathcal{E}_{\mathcal{D}}[y])^2+(\mathcal{E}_{\mathcal{D}}[y]-t)^2+2(y-\mathcal{E}_{\mathcal{D}}[y])(\mathcal{E}_{\mathcal{D}}[y]-t)] \\ &= \frac{1}{2}\cdot\sum_{n=1}^{N_{\mathcal{A}}}\mathcal{E}_{\mathcal{D}}[(y-\mathcal{E}_{\mathcal{D}}[y])^2]+\mathcal{E}_{\mathcal{D}}[(\mathcal{E}_{\mathcal{D}}[y]-t)^2]+\mathcal{E}_{\mathcal{D}}[2(y-\mathcal{E}_{\mathcal{D}}[y])(\mathcal{E}_{\mathcal{D}}[y]-t)] \\ &= \frac{1}{2}\cdot\sum_{n=1}^{N_{\mathcal{A}}}\mathcal{E}_{\mathcal{D}}[(y-\mathcal{E}_{\mathcal{D}}[y])^2]+(\mathcal{E}_{\mathcal{D}}[y]-t)^2 \qquad (2.45)\end{aligned}$$

donde hemos tenido en cuenta que

$$\mathcal{E}_{\mathcal{D}}[(\mathcal{E}_{\mathcal{D}}[y]-t)^2]=(\mathcal{E}_{\mathcal{D}}[y]-t)^2 \qquad (2.46)$$

puesto que la media de $N_{\mathcal{D}}$ números iguales es ese mismo número, y que

$$\begin{aligned}\mathcal{E}_{\mathcal{D}}[2(y-\mathcal{E}_{\mathcal{D}}[y])(\mathcal{E}_{\mathcal{D}}[y]-t)] &= 2(\mathcal{E}_{\mathcal{D}}[y]-t)\cdot\mathcal{E}_{\mathcal{D}}[(y-\mathcal{E}_{\mathcal{D}}[y])] \\ &= 2(\mathcal{E}_{\mathcal{D}}[y]-t)\cdot(\mathcal{E}_{\mathcal{D}}[y]-\mathcal{E}_{\mathcal{D}}[y])=0. \qquad (2.47)\end{aligned}$$

Así pues, hemos llegado a la conclusión de que, si pudiéramos tener muchos conjuntos de entrenamiento, el error promedio que obtendríamos se descompondría en dos términos: $\mathcal{E}_{\mathcal{D}}[(y-\mathcal{E}_{\mathcal{D}}[y])^2]$ y $(\mathcal{E}_{\mathcal{D}}[y]-t)^2$. Al primer término se lo conoce como varianza y representa la dispersión de los valores de y obtenidos con cada conjunto de entrenamiento respecto a su valor medio. La varianza nos da una idea de la dependencia de la función que computa la red neuronal con el conjunto de entrenamiento utilizado. El segundo término es el sesgo al cuadrado y mide la diferencia entre el valor promedio de y obtenido con todos los conjuntos de entrenamiento y el valor verdadero de la función que deseamos que compute la red.

2.2.3. Análisis estadístico: cómo minimizar el error. La validación cruzada y la regularización.

La sección anterior plantea una cuestión interesante que consiste en saber si el conocimiento que hemos obtenido de la naturaleza del error nos puede ayudar a eliminarlo por completo. Por supuesto, el objetivo final sería una red neuronal que diese respuestas sin error en todos los casos que se le plantearan. Una red tal diríamos que ha generalizado a partir del conjunto de pares de entrenamiento. En la práctica eso es imposible salvo para casos simples en los que las soluciones conexionistas no aportan ninguna ventaja frente a soluciones más sencilla y directas basadas en el conocimiento *a priori*. Volviendo al concepto del compromiso entre sesgo y varianza y al ejemplo de la figura 2.6, veremos que en realidad una red que tenga muy poca varianza tendrá un sesgo desproporcionado y *vice versa* y que la solución óptima será un punto intermedio en el que la suma de ambos términos sea mínima pero no nula. En el ejemplo que acabamos de mencionar (el de la figura 2.6) imaginemos que para cualquier

conjunto de entrenamiento de $\mathcal{D}$ ajustamos la función que representa la línea recta discontinua de color azul. En este caso, en el que se le presta escasa atención a los pares de entrenamiento la varianza es nula porque la función ajustada es siempre la misma y no depende del conjunto de entrenamiento empleado. Sin embargo y por esa misma razón, puesto que no se han utilizado los pares de entrenamiento para obtener la función, el sesgo (la diferencia entre la función computada y la función objetivo) será en general inaceptable. Si por el contrario construyésemos una red neuronal que ajustase perfectamente todos los puntos del conjunto de entrenamiento (la otra posibilidad mostrada en la figura 2.6 con línea discontinua naranja), tendríamos que el sesgo se anularía pero la varianza sería muy alta puesto que estaríamos ajustando perfectamente conjuntos de entrenamiento distintos y, por lo tanto, la función calculada sería muy diferente para distintos conjuntos de entrenamiento. Además en ese caso, la capacidad de generalizar a partir de los ejemplos sería muy pequeña puesto que la red se ha especializado en reproducir, no solo la forma general de la función generatriz como era nuestro objetivo, sino también y muy especialmente el ruido. Veremos en el último capítulo que el ruido en cualquiera de sus formas es un componente fundamental de cualquier aplicación conexionista. De alguna forma, generalizar consiste en abstraer el ruido y quedarse con la función que subyace a éste. La solución ideal representada en trazo negro continuo es tal que sería muy parecida para diferentes conjuntos de entrenamiento (baja varianza) y estando además próxima a los valores de entrenamiento. Estas dos condiciones son, a partir de cierto punto, contradictorias y por eso se habla en general de compromiso sesgo-varianza. Ese punto intermedio que minimiza la suma de las dos fuentes de error es un compromiso entre prestar atención a los datos de entrenamiento para que la solución obtenida tenga sesgo pequeño pero no tanta atención como para que la varianza se dispare (*overfitting*). Para encontrar ese punto cuando partimos de un número limitado de pares de entrenamiento existen dos técnicas sencillas denominadas validación cruzada y regularización. Obviamente, la solución ideal es disponer de un conjunto de entrenamiento completo tan grande como precisemos. En ese caso, podemos construir redes neuronales arbitrariamente complejas capaces de ajustar cualquier función: aumentando la complejidad (el número de unidades de computación o neuronas o el de capas) disminuimos el sesgo y aumentando el conjunto de entrenamiento disminuimos la varianza[f]. Existen numerosos trabajos de investigación que tratan la relación que deben cumplir la complejidad de la red neuronal y el tamaño del conjunto de entrenamiento para disminuir la suma de sesgo y varianza. Desgraciadamente, en la mayoría de los casos de interés práctico se dispone de un conjunto limitado de pares de entrenamiento lo que hace necesario recurrir a las técnicas mencionadas anteriormente para hallar el punto óptimo de reducción del error.

Regularización

De la figura 2.6 podemos deducir que en el caso particular de la regresión, una solución sobreentrenada va a estar caracterizada por una curvatura alta. Por supuesto, una solución como la que representa la línea discontinua naranja sólo se puede alcanzar con redes de complejidad suficiente, esto es, con un número suficiente de capas y neu-

[f] Aclarar este último punto hasta sus últimas consecuencias está fuera de los objetivos globales de la asignatura; recomendamos al lector interesado en este aspecto la lectura del libro de C.M. Bishop incluido en la bibliografía. Baste decir aquí que, en el límite de un conjunto de entrenamiento infinito, el resultado de la red neuronal tendería al valor esperado de los valores de t para una x dada

ronas. Es evidente que una red neuronal de tres capas y dos neuronas en la capa oculta no tiene suficientes parámetros para sintetizarla, pero en el caso de que la red tuviera una arquitectura suficientemente compleja para ello, entonces la solución de regresión tendría una alta curvatura prácticamente en todo el rango de entradas. Podemos utilizar esta característica para guiar el proceso de aprendizaje hacia soluciones que eviten el sobreentrenamiento añadiendo al error total una penalización por curvatura:

$$\vec{E} = E(\Omega) + \nu\Lambda \tag{2.48}$$

donde ν es un número real que parametriza la importancia relativa del término de regularización frente al término clásico de error $E(\Omega)$ y Λ es una función que depende de la curvatura total de la solución[g]. De esta forma, conseguimos que el proceso de minimización reduzca progresivamente tanto el error cometido por la red durante el proceso de entrenamiento como la curvatura total de la función mencionada, llegando a un compromiso entre ambos términos.

La forma más habitual y sencilla de aplicar regularización de soluciones neuronales consiste en penalizar la curvatura mediante una función Λ que depende de los pesos:

$$\Lambda = \frac{1}{2} \cdot \sum_{\forall i,j} \omega_{ij}^2 \tag{2.50}$$

A este tipo de regularización se lo conoce como *weight decay* en inglés o disminución progresiva de los pesos. Aunque es difícil ver la relación directa entre los pesos y la curvatura de las soluciones, podemos aproximarnos al problema desde la perspectiva de neuronas con funciones de activación no lineales $g(x)$ de tipo sigmoide. Estas funciones de activación tienen tramos centrales lineales (para valores pequeños de x) y la no linealidad aparece con la saturación a valores absolutos de x mayores (ver figura 2.3). Puesto que x es la suma de las entradas ponderada con los pesos, unos pesos de alto valor absoluto darán con mayor probabilidad sumas ponderadas en la región no lineal de la función de activación y, evidentemente, para que una red neuronal sintetice soluciones de alta curvatura es necesario que una fracción significativa de sus neuronas se encuentren operando en regímenes no lineales. De nuevo, esto último se puede ver porque la composición de funciones lineales es una función lineal (una recta) y una red neuronal es una composición de funciones.

Validación cruzada

Como hemos dicho, uno de los problemas con los que nos encontramos al entrenar una red neuronal es el de poder ajustar perfectamente el conjunto de entrenamiento

[g]La forma de Λ más claramente relacionada con la curvatura en problemas de regresión en una dimensión (una neurona de entrada y una de salida con una o varias capas ocultas) es la dada por la clase de regularizadores de Tikhonov:

$$\Lambda = \frac{1}{2} \cdot \sum_{r=0}^{R} \int_a^b h_r(x) \cdot \left(\frac{d^r y}{dx^r}\right)^2 dx \tag{2.49}$$

en los que la curvatura se penaliza a través de las derivadas hasta orden R de la función que sintetiza la red neuronal.

incluyendo no sólo el modelo que subyace al conjunto sino también el ruido que lo afecta (esto es, llegar al sobreentrenamiento u *overfitting*) disminuyendo por tanto la capacidad de generalización de la red. A esta situación se llega simplemente cuando el número de ciclos de entrenamiento se prolonga indefinidamente. Para evitarlo, el método de validación cruzada divide el conjunto de ejemplos de que disponemos en tres grupos: un conjunto de entrenamiento propiamente dicho (*training set* en inglés), un conjunto de validación o *validation set* y un conjunto de evaluación o prueba *test set*. Durante el entrenamiento se monitorizan los valores del error cuadrático medio tanto del conjunto de entrenamiento como del de validación. Es de esperar que, a la luz de lo tratado en secciones anteriores, el error cuadrático medio del primer conjunto disminuya indefinidamente (si el conjunto es consistente, esto es, no contiene contradicciones; si ése no fuese el caso el error disminuiría hasta oscilar alrededor de un valor mínimo). Por el contrario, el error correspondiente al conjunto de validación disminuye durante la primera fase del entrenamiento (aunque, por lo general, con una pendiente de disminución menos pronunciada que en el caso del conjunto de entrenamiento), alcanza un valor mínimo y, a partir de ese momento, aumenta con una pendiente que depende de la selección particular de ambos conjuntos. La explicación de este comportamiento es la siguiente: durante la primera fase, ambos errores disminuyen puesto que el algoritmo de aprendizaje modifica una red neuronal cuyos parámetros iniciales son obtenidos aleatoriamente. En el punto en que el error cuadrático medio obtenido para el conjunto de validación alcanza el mínimo, la red ha impreso en sus conexiones los aspectos más generales del modelo que queremos reproducir (sea éste un sistema de clasificación, una función de regresión o un sistema no lineal de control o de predicción). A partir de ese punto, la red empieza a incorporar a sus conexiones detalles particulares del conjunto de entrenamiento asimilables al ruido presente en los datos. El error cuadrático medio del conjunto de entrenamiento disminuye porque cada vez ajustamos mejor sus detalles particulares pero el error del conjunto de validación aumenta porque al hacerlo la red pierde capacidad de generalización. El método de validación cruzada consiste entonces simplemente en detener el proceso de aprendizaje cuando se alcanza el valor mínimo del error correspondiente al conjunto de validación y estimar dicho error mediante el conjunto de prueba. La variante conocida en inglés como *k-fold cross validation*, y que podríamos traducir como validación cruzada de orden k, consiste en dividir el conjunto de pares de entrada-salida en k grupos y realizar el entrenamiento con $k-1$ de ellos y la validación con el N-ésimo. Como podemos dejar fuera del conjunto de entrenamiento a cualquiera de los k subconjuntos, lo que haremos será realizar la validación cruzada k veces y en cada ocasión emplearemos como conjunto de validación uno de los k subconjuntos. Así, tendremos finalmente k redes neuronales distintas (porque han sido entrenadas con conjuntos de $k-1$ bloques que difieren entre sí por lo menos en uno de los bloques) cuyos errores podremos promediar.

2.2.4. Tipos de error

En la sección 2.1 hemos desarrollado el formalismo matemático que describe el algoritmo de entrenamiento de una red neuronal de conectividad hacia delante (*feedforward*) mediante la retropropagación del error cuadrático dado por la fórmula 2.5 pero, a pesar de que intuitivamente parece razonable, en ningún momento hemos razonado por qué se ha elegido esa forma de representar el error. Aquí nos proponemos justificar la utilización de esa fórmula para medir el error cometido por la red en tareas de regresión y a proponer otras formas de medir el error más adecuadas para tareas

de clasificación. Para ello necesitaremos introducir notación probabilística. En primer lugar, dado un conjunto de vectores de entrada a la red $\vec{x}$ llamaremos $p(\vec{x})$ a la densidad de probabilidad correspondiente a ese conjunto. Más importante es la definición de $p(\vec{t}|\vec{x})$ que representa la probabilidad de obtener el conjunto de vectores de salida $\vec{t}$ dado el conjunto $\vec{x}$. Este tipo de probabilidades condicionales describen adecuadamente sucesos dependientes, por ejemplo, la probabilidad de que una persona tenga fiebre dado que padece malaria. Es sencillo ver que esa probabilidad es diferente (y mayor) que la probabilidad de que una persona tenga fiebre sin saber nada más de ella. En lo que sigue llamaremos a esta probabilidad condicionada, verosimilitud (*likelihood en inglés*). Finalmente, la densidad de probabilidad de tener al mismo tiempo los conjuntos $\vec{x}$ y $\vec{t}$ la designamos por $p(\vec{x},\vec{t})$

En este apartado queremos abordar el problema de cómo medir el error cometido por una red neuronal entrenada con un conjunto $\mathcal{A}$ compuesto de un conjunto de vectores de entrada $\vec{x}$ y otro de salida $\vec{t}$. Una buena red neuronal será aquella para la que la probabilidad condicionada $p(\vec{t}|\vec{x})$ sea alta porque eso querrá decir que dados nuestros vectores de entrada, será probable que obtengamos los vectores objetivo $\vec{t}$ o, lo que es equivalente, que obtendremos valores próximos a los deseados. A este principio se lo conoce como el de máxima verosimilitud y consiste en seleccionar la red neuronal que maximice $p(\vec{t}|\vec{x})$.

Como los pares de entrenamiento de A no dependen los unos de los otros sino que son casos independientes, podemos escribir la verosimilitud $\mathcal{L}$ como

$$\mathcal{L} = p(\vec{t}|\vec{x}) = \prod_{n=1}^{N_{\mathcal{A}}} p(\vec{t}[n]|\vec{x}[n]) \tag{2.51}$$

y el objetivo del algoritmo de entrenamiento será maximizar la verosimilitud, es decir, la probabilidad de obtener los vectores objetivo dadas las entradas correspondientes. En la práctica, es lo mismo maximizar la verosimilitud que minimizar el logaritmo negativo de dicha verosimilitud ($-\ln\mathcal{L}$) pero lo segundo es más sencillo y por eso es más habitual encontrarselo en la bibliografía. En ese caso y dadas las propiedades de los logaritmos[h] tendríamos que minimizar

$$-\ln\mathcal{L} = -\sum_{n=1}^{N_{\mathcal{A}}} p(\vec{t}[n]|\vec{x}[n]). \tag{2.52}$$

Así pues, hemos llegado intuitivamente a la conclusión de que $-\ln\mathcal{L}$ es una medida del error de la red de manera que, cuanto menor es $-\ln\mathcal{L}$ mejor es la red neuronal a la hora de reproducir el conjunto de entrenamiento. Definamos entonces el error como $E = -\ln\mathcal{L}$.

En tareas de **regresión** como la representada gráficamente en la figura 2.6, tenemos un conjunto de entrenamiento que suponemos que se originó a partir de una función (dibujada en trazo negro) pero a la que se le ha superpuesto ruido (por ejemplo porque no se ha medido con exactitud el valor). Es razonable esperar que para cada valor de la coordenada horizontal la probabilidad de encontrar los puntos del conjunto de entrenamiento sea mayor cuanto más próximos estén al valor real de la función que los generó y que no tengamos puntos del conjunto de entrenamiento demasiado alejados

[h] En este caso, que el logaritmo de un producto es la suma de los logaritmos de los factores.

de la curva negra continua. Matemáticamente, eso se escribe como que

$$p(\vec{t}[n]|\vec{x}[n]) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp(-\frac{(\vec{y}[n]-\vec{t}[n])^2}{2\sigma^2}) \tag{2.53}$$

donde $\vec{y}[n]$ es el vector de salida de la red neuronal correspondiente del vector de entrada $\vec{x}[n]$ y σ es una variable que mide la dispersión de los ejemplos alrededor de la función que los generó. A la distribución de probabilidad de la ecuación 2.53 se la conoce como distribución gaussiana.

Pues bien, el lector puede comprobar que si sustituimos la ecuación 2.53 en 2.52 obtenemos que

$$E = \frac{1}{2\sigma^2} \sum_{n=1}^{N_{\mathcal{A}}} (\vec{y}[n]-\vec{t}[n])^2 \tag{2.54}$$

más una serie de términos constantes. Como nuestra intención es minimizar el error, podemos prescindir de los términos constantes y quedarnos con la ecuación 2.54. Así pues, **hemos obtenido la fórmula del error cuadrático a partir del método de máxima verosimilitud para el caso de una tarea de regresión en la que los ejemplos del conjunto de entrenamiento tienen una distribución gaussiana alrededor de la función generatriz.**

¿Qué ocurre con los problemas de clasificación? En las tareas de clasificación no tiene sentido pensar en una distribución gaussiana de los vectores objetivo. Un vector de entrada pertenece a la clase $\mathcal{C}_1$ o a la $\mathcal{C}_2$ o a la $\mathcal{C}_8$ pero no a una clase $\mathcal{C}_{1,2}$ por que no existe tal cosa. Así pues, necesitamos otra función de error que obtendremos también por el mismo procedimiento de maximizar la verosimilitud. De hecho, en muchas ocasiones se utiliza el error cuadrático para entrenar redes de clasificación a pesar de que como decimos no es el más adecuado. La razón es que hacerlo simplifica extraordinariamente los algoritmos de aprendizaje convirtiéndolos en problemas de optimización lineal mucho más tratables en la implementación.

Supongamos que dado un vector $\vec{x}[n]$ del conjunto de entrenamiento $\mathcal{A}$, codificamos la clase mediante una representación 1-de-N_s. Volveremos sobre este tipo de codificaciones más adelante, pero baste aquí decir que consiste en emplear tantas neuronas en la capa de salida como clases posibles haya en el esquema de clasificación que queremos implementar de forma que si el vector de entrada $\vec{x}[n]$ pertenece a la clase $\mathcal{C}_k$ el vector objetivo en el conjunto de entrenamiento $\vec{t}[n]$ tendrá todas sus componentes nulas salvo la correspondiente a la neurona k que valdrá 1.

Si queremos emplear de nuevo la estrategia de maximizar la verosimilitud, necesitamos una ecuación equivalente a 2.53 para redes neuronales clasificadoras, es decir, necesitamos una fórmula para $p(\vec{t}[n]|\vec{x}[n])$. Nuestro objetivo es construir una red neuronal que dado un vector $\vec{x}[n]$ nos proporcione a la salida las probabilidades de que pertenezca a las clases $\mathcal{A}_1$, $\mathcal{A}_2$, ..., $\mathcal{A}_{N_s}$. Es decir, queremos que

$$y_k = p(\mathcal{C}_k|\vec{x}[n]) \tag{2.55}$$

lo que equivale a que

$$p(\vec{t}[n]|\vec{x}[n]) = \prod_{k=1}^{N_s} y_k[n]^{t_k[n]}. \tag{2.56}$$

Una vez que hemos obtenido esta ecuación podemos escribir el error como

$$E = -\ln \mathcal{L} = -\sum_{n=1}^{N_{\mathcal{A}}} \sum_{k=1}^{N_s} t_k[n] \cdot \ln y_k[n] \tag{2.57}$$

y ya hemos conseguido escribir una función de error apropiada para tareas de clasificación.

El proceso que nos ha llevado a obtener **la fórmula del error cuadrático a partir del método de máxima verosimilitud para el caso de una tarea de clasificación** parte de la exigencia de que **los valores de salida de la red deben ser interpretables como probabilidades de pertenencia a las distintas clases** (ver ecuación 2.55). Ello implica que las neuronas de salida deben tener valores comprendidos entre 0 y 1 porque la probabilidad de un suceso (condicionada o no) no puede exceder ese rango. Por ello, en problemas de clasificación se emplean neuronas *softmax* en la capa de salida. Las neuronas *softmax* tienen funciones de activación derivadas de la función de activación logística pero convenientemente modificada para cumplir el requisito de que sus valores de salida estén comprendidos en el rango [0,1]. Su expresión matemática en función de las sumas ponderadas a_k de cada neurona de salida es

$$y_k = \frac{\exp(a_k)}{\sum_{k'=1}^{N_s} \exp(a_{k'})} \tag{2.58}$$

Por supuesto, existen muchas otras definiciones del error que dan lugar a distintas soluciones cuando se emplean con el algoritmo de retropropagación del error. Aquí hemos mostrado sólo dos (las más populares) pero el lector puede explorar la bibliografía para descubrir diferentes aproximaciones a este tema y sus consecuencias prácticas para la red que resulta.

En cualquier caso, la exigencia que hemos impuesto a nuestra definición de error en el caso de la tarea de clasificación, a saber, que los resultados de la red sean interpretables directamente como probabilidades condicionadas, es deseable en general. En [Bis95] se puede encontrar una justificación teórica y matemática de las condiciones que debe cumplir una función de error para que sea así.

2.3. Mejoras del algorimo

La descripción del algoritmo hecha hasta aquí corresponde a una formulación básica sobre la que es posible, como se adelantaba en secciones anteriores, introducir variaciones que aumenten la eficiencia del algoritmo. Estas variaciones tienen como objetivo mejorar el proceso de búsqueda de los mínimos de la función error $E(\Omega)$. Recordemos que la estrategia descrita hasta ahora, denominada regla delta era un caso particular de descenso del gradiente y se resumía en la ecuación 2.6. La primera variante del método de retropropagación del error que vamos a presentar se conoce como *retropropagación con momento* o *retropropagación mejorada* (*Enhanced Backpropagation* en inglés) [PNH86] y se basa en siguiente fórmula de cálculo de las actualizaciones de los pesos:

Retropropagación con momento

$$\Delta\omega_{ji}^{(l)} = -\alpha \cdot \frac{\partial E(\Omega)}{\partial \omega_{ji}^{(l)}} + \mu \cdot \Delta\omega_{ji}^{(l-1)}. \tag{2.59}$$

El segundo término de la derecha es lo que se conoce como momento y μ es un factor comprendido entre 0 y 1 que mide la importancia relativa de dicho término. Su efecto sobre el algoritmo es el siguiente: en aquellas regiones del espacio de pesos en que el

gradiente sea constante la fórmula anterior se puede reescribir como

$$\Delta\omega_{ji}^{(l)} = -\alpha \cdot \frac{\partial E(\Omega)}{\partial \omega_{ji}^{(l)}} + \mu \cdot \Delta\omega_{ji}^{(l-1)} = -\alpha \cdot (1+\mu) \cdot \frac{\partial E(\Omega)}{\partial \omega_{ji}^{(l)}}. \tag{2.60}$$

lo que implica que el término de momento ha supuesto un aumento efectivo del parámetro de aprendizaje en un factor $(1+\mu)$. En otras palabras, si el gradiente es constante en una región del espacio aumentamos la velocidad de aprendizaje para atravesar rápido esa región. Sin embargo, en las proximidades de un mínimo local, el gradiente cambia de signo y de valor absoluto como muestra la figura 1.7 y, por tanto el efecto neto del término de momento se cancela.

Regla delta-barra-delta

La segunda variante del algoritmo que vamos a tratar se conoce como regla delta-barra-delta (*delta-bar-delta rule* en inglés) [Jac88] e introduce como novedad la existencia de velocidades de aprendizaje dependientes del peso que se esté ajustando. En lugar de un único valor de α para todos los pesos, ahora tendremos un conjunto de valores α_{ji} correspondiente al conjunto de pesos ω_{ji}. Matemáticamente, la regla de descenso del gradiente se transforma en

$$\Delta\omega_{ji}^{(l)} = -\alpha_{ji}^{(l)} \cdot \frac{\partial E(\Omega)}{\partial \omega_{ji}^{(l)}}, \tag{2.61}$$

de forma que, en regiones en que el gradiente respecto a un peso en particular tiene el mismo signo en pasos consecutivos, la velocidad de aprendizaje puede ser alta mientras que, por el contrario, en regiones en que el signo del gradiente oscila, la velocidad disminuye de valor y todo ello de forma independiente para cada peso.La implementación práctica ha de dar una prescrición del modo en que se van a modificar las velocidades de apendizaje que podemos resumir así

$$\Delta\alpha_{ji}^{(l)} = \begin{cases} \kappa & \text{si } \vec{q}_{ji}^{(l-1)} q_{ji}^{(l)} > 0 \\ -\phi \cdot \alpha_{ji}^{(l)} & \text{si } \vec{q}_{ji}^{(l-1)} q_{ji}^{(l)} < 0 \end{cases} \tag{2.62}$$

donde

$$q_{ji}^{(l)} = \frac{\partial E(\Omega)}{\partial \omega_{ji}^{(l)}} \tag{2.63}$$

$$\vec{q}_{ji}^{(l)} = (1-\theta) \cdot q_{ji}^{(l)} + \theta \vec{q}_{ji}^{(l-1)}. \tag{2.64}$$

El problema de esta variante es evidente: donde antes teníamos un parámetro (α) ahora tenemos tres (κ, ϕ y θ). Si además añadimos el término de momento, el número de parámetros pasa a cuatro lo cual puede convertirse en un serio inconveniente. Además, es razonable pensar que los pesos no son independientes entre sí, sino que existen correlaciones que ligan unos a otros con lo que las ventajas de esta variante no tiene por qué significar una mejora drástica de las capacidades de la red.

Quickpropagation

Quickpropagation o *Quickprop* [Fah88] es la tercera variante (también basada en la hipótesis de independencia de los pesos) que vamos a describir en este apartado. La fórmula de actualización de los pesos en este método es

$$\Delta\omega_{ji}^{(l+1)} = \frac{q_{ji}^{(l)}}{q_{ji}^{(l-1)} - q_{ji}^{(l)}} \cdot \Delta\omega_{ji}^{(l)} \tag{2.65}$$

y su justificación proviene de considerar que la función error dependiente de los pesos es, en primera aproximación, una suma de parábolas. En ese caso, podemos escribir que

$$E(\Omega) = c_2 \cdot \omega_{ji}^2 + c_1 \cdot \omega_{ji} + c_0 + E'(\Omega') \tag{2.66}$$

donde sólo hemos hecho explícita la dependencia de $E(\Omega)$ con ω_{ij}. El término $E'(\Omega'$ representa la dependencia del error con el resto de los pesos. La derivada de $E(\Omega)$ será

$$\frac{\partial E(\Omega)}{\partial \omega_{ij}} = 2 \cdot c_2 \cdot \omega_{ji} + c_1 \tag{2.67}$$

de forma que, en el mínimo $2 \cdot c_2 \cdot \omega_{ji} + c_1 = 0$ y, por lo tanto, $\omega_{ji} = \frac{-c_1}{2c_2}$. Ahora bien, existe una relación entre c_1 y c_2, y las derivadas de la función error puesto que

$$2 \cdot c_2 = \frac{\partial^2 E}{\partial \omega_{ji}^2} \tag{2.68}$$

donde hemos omitido por claridad la dependencia de Ω, y

$$c_1 = \frac{\partial E}{\partial \omega_{ij}} - \frac{\partial^2 E}{\partial \omega_{ji}^2} \cdot \omega_{ji}^{(l)}. \tag{2.69}$$

Escribiendo las derivadas parciales en diferencias finitas tenemos que

$$2 \cdot c_2 = \frac{\frac{\partial E}{\partial \omega_{ij}^{(l)}} - \frac{\partial E}{\partial \omega_{ij}^{(l-1)}}}{\omega_{ji}^{(l)} - \omega_{ji}^{(l-1)}} = \frac{q_{ji}^{(l)} - q_{ji}^{(l-1)}}{\Delta\omega_{ji}^{(l)}} \tag{2.70}$$

y

$$c_1 = q_{ji}^{(l)} - \frac{q_{ji}^{(l)} - q_{ji}^{(l-1)}}{\Delta\omega_{ji}^{(l)}} \cdot \omega_{ji}^{(l)} \tag{2.71}$$

y sustituyendo,

$$\Delta\omega_{ji}^{(l+1)} = \omega_{ji}^{(l+1)} - \omega_{ji}^{(l)} = \frac{-c_1}{2c_2} - \omega_{ji}^{(l)} = \frac{-(q_{ji}^{(l)} - \frac{q_{ji}^{(l)} - q_{ji}^{(l-1)}}{\Delta\omega_{ji}^{(l)}})}{\frac{q_{ji}^{(l)} - q_{ji}^{(l-1)}}{\Delta\omega_{ji}^{(l)}}} - \omega_{ji}^{(l)} \tag{2.72}$$

$$= -\frac{q_{ji}^{(l)}\Delta\omega_{ji}^{(l)} - (q_{ji}^{(l)} - q_{ji}^{(l-1)}) \cdot \omega_{ji}^{(l)}}{(q_{ji}^{(l)} - q_{ji}^{(l-1)})} - \omega_{ji}^{(l)} \tag{2.73}$$

$$\tag{2.74}$$

de donde se obtiene con facilidad la ecuación 2.65.

Finalmente, la última variante de retropropagación del error que vamos a mencionar es la que se conoce como *Resilient Backpropagation* o Retropropagación flexible [RB93]. Consiste en reescribir la regla de actualización de los pesos reteniendo la información que porta signo del gradiente pero prescindiendo de su valor absoluto. Matemáticamente, se expresa como

Resilient Backpropagation

$$\Delta_{ji}^{(l)} = \begin{cases} -\psi_{ji}^{(l)} & \text{si } \frac{\partial E}{\partial \omega_{ji}} > 0 \\ +\psi_{ji}^{(l)} & \text{si } \frac{\partial E}{\partial \omega_{ji}} < 0 \\ 0 & \text{si } \frac{\partial E}{\partial \omega_{ji}} = 0 \end{cases} \tag{2.75}$$

donde

$$\psi_{ji}^{(l)} = \begin{cases} \alpha^+ \cdot \psi_{ji}^{(l-1)} & \text{si } \frac{\partial E}{\partial \omega_{ji}}^{(l-1)} \cdot \frac{\partial E}{\partial \omega_{ji}}^{(l)} > 0 \\ \alpha^+ \cdot \psi_{ji}^{(l-1)} & \text{si } \frac{\partial E}{\partial \omega_{ji}}^{(l-1)} \cdot \frac{\partial E}{\partial \omega_{ji}}^{(l-1)} < 0 \\ \psi_{ji}^{(l-1)} & \text{si } \frac{\partial E}{\partial \omega_{ji}}^{(l-1)} \cdot \frac{\partial E}{\partial \omega_{ji}}^{(l)} = 0. \end{cases} \tag{2.76}$$

De esta forma, en regiones donde el gradiente cambia lentamente, el signo de la modificación se conserva respecto a la iteración anterior y el valor absoluto se incrementa en un factor α^+, donde α^+ es una constante ligeramente superior a 1. Un valor habitual suele estar entorno a 1,2. Por el contrario, en regiones en que el gradiente cambia de signo (es decir, cuando hemos atravesado un mínimo) el cambio en los pesos invierte el signo respecto a la iteración anterior y disminuye su valor absoluto en un factor α^- inferior a 1 (del orden de 0,5).

2.4. La notación matricial

En muchos textos que tratan el tema de las redes neuronales se extiende la notación empleada hasta aquí con la introducción del cálculo matricial lo que permite resumir el algoritmo de forma mucho más compacta. Las ecuaciones más importantes para la capa l-ésima de la red serían, escritas en forma matricial, las siguientes:

$$\vec{a}^{(l)} = \begin{pmatrix} a_1^{(l)} \\ a_2^{(l)} \\ \dots \\ a_N^{(l)} \end{pmatrix} = \begin{pmatrix} \omega_{11}^{(l-1)} \cdot y_1^{(l-1)} + \omega_{12}^{(l-1)} \cdot y_2^{(l-1)} + \dots + \omega_{1N_{l-1}}^{(l-1)} \cdot y_{N_{l-1}}^{(l-1)} \\ \omega_{21}^{(l-1)} \cdot y_1^{(l-1)} + \omega_{22}^{(l-1)} \cdot y_2^{(l-1)} + \dots + \omega_{2N_{l-1}}^{(l-1)} \cdot y_{N_{l-1}}^{(l-1)} \\ \dots \\ \omega_{N_l1}^{(l-1)} \cdot y_1^{(l-1)} + \omega_{N_l2}^{(l-1)} \cdot y_2^{(l-1)} + \dots + \omega_{N_lN_{l-1}}^{(l-1)} \cdot y_{N_{l-1}}^{(l-1)} \end{pmatrix} = \Omega^{(l-1)} \cdot \vec{y}^{(l-1)} \tag{2.1}$$

donde $\Omega^{(l-1)}$ es la matriz de pesos (de dimensiones $N_l \times N_{l-1}$) dada por

$$\Omega^{(l-1)} = \begin{pmatrix} \omega_{11}^{(l-1)} & \omega_{12}^{(l-1)} & \dots & \omega_{1N_{l-1}}^{(l-1)} \\ \omega_{21}^{(l-1)} & \omega_{22}^{(l-1)} & \dots & \omega_{2N_{l-1}}^{(l-1)} \\ \dots & & & \\ \omega_{N_l1}^{(l-1)} & \omega_{N_l2}^{(l-1)} & \dots & \omega_{N_lN_{l-1}}^{(l-1)} \end{pmatrix} \tag{2.77}$$

e $\vec{y}^{(l)}$ es el vector de salida de la capa l-ésima. La matriz de pesos entre la capa de entrada ($l = 1$) y la primera capa oculta ($l = 2$), por ejemplo, tiene dimensiones $N_o \times N_e$, y la matriz de pesos entre la capa de oculta y la capa de salida, $N_s \times N_o$.

$$\vec{y}^{(l)} = g(\vec{a}^{(l)}) = g(\Omega^{(l-1)} \cdot \vec{y}^{(l-1)}) \tag{2.2}$$

Más aún, se pueden definir los vectores

$$\nabla_y E(\Omega) = \begin{pmatrix} \frac{\partial E(\Omega)}{\partial y_1^{(l)}} \\ \frac{\partial E(\Omega)}{\partial y_2^{(l)}} \\ \dots \\ \frac{\partial E(\Omega)}{\partial y_{N_l}^{(l)}} \end{pmatrix} \tag{2.78}$$

de forma que, en el caso más simple de un perceptrón de tres capas, las ecuaciones de actualización de los pesos de la capa oculta en forma matricial serían:

$$\Delta\Omega^{(l-1)} = -\alpha \cdot \nabla_{\Omega^{(l-1)}} E(\Omega) = -\alpha \cdot [\nabla_{y^{(l)}} E(\Omega) \odot \frac{dg}{d\vec{a}^{(l)}}(\vec{a}^{(l)})] \cdot (\vec{y}^{(l-1)})^T \tag{2.79}$$

donde $\odot$ es el producto Hadamard[i] de dos matrices e $(\vec{y}^{(l-1)})^T$ es el vector de entradas traspuesto (puesto en forma de vector fila). Finalmente, $\nabla_{\Omega^{(l-1)}} E(\Omega)$ sería

$$\nabla_{\Omega^{(l-1)}} E(\Omega) = \begin{pmatrix} \frac{\partial E(\Omega)}{\partial \omega_{11}^{(l-1)}} & \frac{\partial E(\Omega)}{\partial \omega_{12}^{(l-1)}} & \dots & \frac{\partial E(\Omega)}{\partial \omega_{1N_{l-1}}^{(l-1)}} \\ \frac{\partial E(\Omega)}{\partial \omega_{21}^{(l-1)}} & \frac{\partial E(\Omega)}{\partial \omega_{22}^{(l-1)}} & \dots & \frac{\partial E(\Omega)}{\partial \omega_{2N_{l-1}}^{(l-1)}} \\ \dots & \dots & \dots & \dots \\ \frac{\partial E(\Omega)}{\partial \omega_{N_l 1}^{(l-1)}} & \frac{\partial E(\Omega)}{\partial \omega_{N_l 2}^{(l-1)}} & \dots & \frac{\partial E(\Omega)}{\partial \omega_{N_l N_{l-1}}^{(l-1)}} \end{pmatrix} \tag{2.80}$$

y, en el caso de funciones de activación sigmoideas por ejemplo,

$$\frac{dg}{d\vec{a}^{(l)}}(\vec{a}^{(l)}) = \begin{pmatrix} g(a_1) \cdot (1 - g(a_1)) \\ g(a_2) \cdot (1 - g(a_2)) \\ \dots \\ g(a_{N_l}) \cdot (1 - g(a_{N_l})) \end{pmatrix} \tag{2.81}$$

[i] Dadas dos matrices $A_{m\times n}$ y $B_{m\times n}$, el producto de Hadamard se define como la matriz $C_{m\times n}$ tal que sus elementos c_{ij} se obtienen como el producto de a_{ij} y b_{ij}.

Capítulo 3

Los mapas autoorganizados de Kohonen

Objetivo: El objetivo del presente módulo es dar una introducción a las redes autoorganizadas de Kohonen . La estructura de este módulo consiste en una introducción formal del algoritmo de actualización de los pesos en su versión más sencilla, una sección dedicada al significado teórico del algoritmo y, finalmente, una sección dedicada a variantes del método y su significación teórica. Hay que aclarar aquí que sólo vamos a tratar de mapas autoorganizados. El término más general de red de Kohonen se puede referir a cualquiera de las arquitecturas y algoritmos desarrollados por el investigador en Neurocomputación Teuvo Kohonen, por ejemplo, VQ (*Vector Quantization en inglés)*, *LVQ (*Learning Vector Quantization en inglés) o los propios mapas autoorganizados.

3.1. Introducción. Algoritmos de agrupamiento en clases o *clustering*.

Uno de los problemas más habituales en los que se emplean redes neuronales es el de la clasificación no supervisada, que consiste en separar un conjunto de datos en clases sin ningún conocimiento adicional al margen de los propios datos. Este tipo de problemas aparecen con frecuencia en el ámbito de la minería de datos con interesantes aplicaciones en el ámbito empresarial. Un ejemplo sencillo sería el de una base de datos en la que se almacenan una serie de variables relativas a clientes de una empresa. La empresa podría querer agrupar los clientes en clases homogéneas (perfiles) de forma que pudiera ofrecer a cada cliente productos adecuados al perfil de su clase, en lugar de bombardearle con un gran número de ofertas entre las que se perderían las verdaderamente interesantes para el cliente.

En lo que sigue, vamos a describir una serie de algoritmos competitivos de entrenamiento no supervisado. Se denominan competitivos porque, dado un vector de entradas, las neuronas de la red van a *competir* entre ellas de acuerdo con un cierto criterio, de forma que el vector de entradas será asignado a la neurona que sea más adecuada

de acuerdo con ese criterio. Se denominan no supervisados porque no conocemos *a priori* las clases ni qué vectores de entrada pertenecen a qué clases, sino que todo ello emergerá como resultado del entrenamiento de la red.

El primero de los métodos no es estrictamente un método conexionista pero se puede implementar con una red neuronal. Se trata de las medias-k (*k-means* en inglés) y su implementación conexionista consiste en una red con una primera capa de neuronas de entrada seguida de una capa en la que cada neurona va ser interpretada como una clase. Por lo tanto, en la segunda capa habrá que crear tantas neuronas como clases deseemos (esto es, se trata de una decisión previa a la construcción de la red). Supongamos para fijar conceptos que tenemos una capa de entrada de M neuronas para la entrada de vectores $\vec{x}$ de M componentes $x_1, x_2, ..., x_M$ y supongamos también que la capa de clasificación tiene N_c neuronas que representan las N_c clases en las que queremos agrupar los vectores de entradas. Cada neurona de la segunda capa recibe conexiones de todas las neuronas de la capa de entrada por lo que su vector de pesos tendrá M componentes.

Para cada vector $\vec{x}$ de entradas el algoritmo de las medias-k comienza por buscar la neurona j cuyo vector de pesos $\vec{\omega}_j$ se encuentre más próximo al vector de entrada. ¿Cómo se mide la 'proximidad' de dos vectores? Ése es precisamente el criterio al que hacíamos referencia anteriormente cuando explicábamos qué es un algoritmo competitivo. En general, la proximidad se mide definiendo primero una métrica, es decir, una forma de medir distancias. Una forma sencilla y habitual de definir distancias es la euclídea, estudiada en los primeros cursos de matemática elemental. En este caso, el vector de pesos más próximo al vector de entrada será el de la neurona j si el valor de

$$\|\vec{x} - \vec{\omega}_j\| = \sqrt{(x_1 - \omega_{1j})^2 + (x_2 - \omega_{2j})^2 + ... + (x_M - \omega_{Mj})^2} \tag{3.1}$$

es mínimo, donde $\|\vec{x} - \vec{\omega}_j\|$ es la distancia euclídea entre los dos vectores.

Si los vectores de entrada $\vec{x}$ están normalizados[a], la métrica euclídea es equivalente a encontrar el vector que forma un ángulo mínimo con el vector de entrada. De esta forma, la neurona j cuyo vector de pesos se encuentra más próximo al vector de entrada será aquélla para la cual $\vec{x} \cdot \vec{\omega}_j$ tenga el valor máximo, porque

$$\vec{x} \cdot \vec{\omega}_j = \|\vec{x}\| \cdot \|\vec{\omega}_j\| \cdot \cos\theta = \cos\theta \tag{3.2}$$

y, por lo tanto, el producto escalar de dos vectores de longitud unidad es igual al coseno del ángulo que forman y éste es máximo cuando el ángulo es mínimo.

Supongamos entonces que, dado el vector de entrada $\vec{x}$, a éste se le asigna la neurona 'ganadora' j de acuerdo con el criterio de la distancia euclídea y supongamos también que a dicha neurona ya se le asignaron antes N vectores de entrada (si $\vec{x}$ es el primer vector que se le presenta a la red, entonces $N = 0$). Entonces, el nuevo vector de pesos $\vec{\omega}_i^{n+1}$ vendrá dado en función del valor anterior $\vec{\omega}_i^n$ por la fórmula

$$\vec{\omega}_i^{n+1} = \frac{N}{N+1} \cdot \vec{\omega}_i^n + \frac{1}{N+1} \cdot \vec{x} \tag{3.3}$$

que, como se puede comprobar, da como resultado final de cada ciclo que los vectores $\vec{\omega}_i$ correspondan al valor medio de los vectores de entrada asignados a la neurona i. A

[a] Un vector se normaliza a la unidad dividiéndolo por su módulo.

medida que se modifican los vectores $\vec{\omega}_i$ de las neuronas clasificadoras, la asignación de los vectores de entrada puede variar y un vector $\vec{x}$ que inicialmente fuera asignado a la neurona i por la proximidad (en sentido euclideo) a su vector de pesos, puede posteriormente ser asignado a otra neurona como resultado de la modificación de $\vec{\omega}_i$ por el algoritmo.

Se puede demostrar que esta forma de modificar los pesos de la red asegura la convergencia a una clasificación óptima si medimos el error de la clasificación como la suma de los cuadrados de las diferencias entre los vectores de entrada y los vectores de pesos de sus neuronas ganadoras correspondientes.

$$E = \sum_{n=1}^{N_{\mathcal{A}}} (\vec{x}[n] - \vec{\omega}[n])^2 \tag{3.4}$$

donde $N_{\mathcal{A}}$ es el número de vectores de entrada que queremos agrupar en clases, $\vec{x}[n]$ es el vector n-ésimo de ese conjunto de $N_{\mathcal{A}}$ vectores y $\vec{\omega}[n]$ es el vector de pesos de la neurona ganadora correspondiente a $\vec{x}[n]$.

Ejemplo 1:

Supongamos que queremos agrupar en dos clases los siguientes vectores de entrada:

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}
20	38	18	15	23	39	42	18	35	40
110	1000	70	70	110	1100	900	90	1200	1100

Los valores de la primera componente de cada vector podría ser la edad de un cliente de banco y la segunda los ingresos mensuales promedio del último año. Obviamente se puede deducir a simple vista que hay dos clases, una compuesta por mayores de 35 años con ingresos superiores a 900 euros mensuales y otra de menores de 23 con ingresos inferiores a 110 euros mensuales.

Si quisieramos agrupar en clases los 10 vectores de la tabla mediante una red neuronal que implementase las medias-k ésta tendría que tener una arquitectura de dos neuronas en la capa de entrada (correspondientes a las dos componentes de cada vector de entrada) y dos neuronas en la capa de clasificación (dado que queremos agrupar dichos vectores en sólo dos clases).

Inicializamos los pesos de las neuronas de la capa de clasificación según el método más habitual con los dos primeros vectores de entrada:

$$\begin{array}{lll} \text{Neurona 1} & \omega_{11} = 20 & \omega_{21} = 110 \\ \text{Neurona 2} & \omega_{12} = 38 & \omega_{22} = 1000 \end{array}$$

Presentamos a la red el primer vector de entrada y buscamos la neurona ganadora, calculando su distancia euclidea con los vectores de pesos. Respecto a la neurona 1 la distancia euclídea será $\sqrt{(20-20)^2+(110-110)^2} = 0$ y respecto a la neurona 2, $\sqrt{(20-38)^2+(110-1000)^2} = 890{,}2$. Por lo tanto la neurona ganadora es la primera (como cabía esperar dada la inicialización de los pesos) y sus nuevos no cambian porque $N = 0$ en la ecuación 3.3 (es el primer vector de entrada asignado a la neurona 1).

Presentando el segundo vector de entrada, tendremos que respecto a la neurona 1 la distancia euclídea será $\sqrt{(38-20)^2+(1000-110)^2} = 890{,}2$ y respecto a la neurona

2, 0 (de nuevo como cabía esperar). Por lo tanto la neurona ganadora es, en esta ocasión la segunda y sus nuevos pesos tampoco cambian porque $N = 0$ en la ecuación 3.3 (es el primer vector de entrada asignado a la neurona 2).

Presentando el tercer vector de entrada, tendremos que respecto a la neurona 1 la distancia euclídea será $\sqrt{(18-20)^2+(70-110)^2} = 40{,}0$ y respecto a la neurona 2, $\sqrt{(18-38)^2+(70-1000)^2} = 930{,}2$. Por lo tanto la neurona ganadora es la primera y sus nuevos pesos se calculan según la ecuación 3.3

$$\omega_{11} = \tfrac{1}{2}\cdot 20 + \tfrac{1}{2}\cdot 18 = 19 \quad \omega_{21} = \tfrac{1}{2}110 + \tfrac{1}{2}70 = 90$$

Si repetimos el proceso para todos los vectores de entrada iríamos obteniendo los siguientes valores de los pesos:

Vector de entrada	Neurona ganadora	ω_{11}	ω_{21}	ω_{12}	ω_{22}
x_1	1	20	110	38	1000
x_2	2	10	110	38	1000
x_3	1	19	90	38	1000
x_4	1	17.7	83.3	38	1000
x_5	1	19.0	90.0	38	1000
x_6	2	19.0	90.0	38.5	1050
x_7	2	19.0	90.0	39.7	1000
x_8	1	18.8	90.0	39.7	1000
x_9	2	18.8	90.0	38.5	1050
x_10	2	18.8	90.0	38.8	1060

que, como se puede comprobar, coincide con las medias de los vectores de entrada para cada clase. Repetimos el proceso hasta que un nuevo ciclo de actualizaciones no varíe los pesos más allá de un margen prefijado.

Otra forma sencilla de realizar la clasificación no supervisada con algoritmos competitivos es la denominada de Cuantificación de Vectores (o *Vector Quantization* en inglés) diseñada por Teuvo Kohonen. Es muy similar a la de las medias-k pero la ecuación de actualización de los pesos no depende de N (el número de vectores de entrada previamente asignados a la neurona ganadora) sino de una velocidad de aprendizaje α elegida *a priori*:

$$\Delta\vec{\omega}_i = \vec{\omega}_i^{n+1} - \vec{\omega}_i^{n} = \alpha\cdot(\vec{x} - \vec{\omega}_i) \tag{3.5}$$

o lo que es lo mismo

$$\vec{\omega}_i^{n+1} = (1-\alpha)\cdot\vec{\omega}_i^{n} + \alpha\cdot\vec{x} \tag{3.6}$$

En este caso, modificamos el peso de la neurona ganadora haciendo que se aproxime más al vector de entrada porque le sumamos una fracción α del vector diferencia entre el vector de entrada y el vector de pesos de la neurona ganadora $(\vec{x} - \vec{\omega}_i)$.

3.2. Los mapas autoorganizados de Kohonen. El algoritmo.

Una red autoorganizada de Kohonen consta de dos capas: la primera pasiva que sólo recibe las entradas y las transmite sin procesar a todas las neuronas de la siguiente capa y, la segunda que habitualmente tiene dos dimensiones y que es la responsable del procesamiento de las entradas de acuerdo con el algoritmo que describimos a continuación.

Sea $\vec{x}$ el vector de N_e componentes compuesto por las entradas a la red neuronal. Cada neurona de la segunda capa estará conectada con todas la neuronas de la capa de entrada. Llamemos ω_{ji} al peso de la sinapsis que realiza la neurona i de la primera capa con la neurona j de la segunda. Por lo tanto, la neurona j tendrá un vector de pesos de N_e componentes dado por $\vec{\omega}_j$ compuesto por ω_{ji} con $i = 1, 2, ..., N_e$.

Para cada vector $\vec{x}$ de entrada, el algoritmo comienza por buscar la neurona j cuyo vector de pesos $\vec{\omega}_j$ se encuentre más próximo al vector de entrada. Supongamos, como en la sección anterior, que escogemos la distancia euclídea como métrica con la que seleccionar la neurona con el vector de pesos más próximo al vector de entrada. Llamemos a esa neurona, neurona ganadora. Una vez hallada, el algoritmo va a modificar sus pesos y los de las neuronas que se encuentren próximas a ella (en su vecindad). Ésta es una diferencia importante respecto al método de retropropagación del error en el que todos los pesos de todas las capas son modificados simultáneamente. Aquí sólo se modifican los vectores de pesos de la neurona ganadora y los de las que se encuentran en sus alrededores. En los mapas autoorganizados, cada vector de entrada define una neurona ganadora cuyos pesos serán modificados así como los de las neuronas de su vecindad y ésta es la diferencia fundamental respecto a los algoritmos de agrupamiento en clases vistos en la sección anterior. Una vez introducidos todos los vectores de entrada, se procede a repetir el ciclo hasta que la presentación de los vectores de entrada no produzca cambios apreciables en la topología de la red (es decir, en los valores de los pesos).

Como se ha mencionado antes, los términos *proximidad* o *alrededores* implican la definición de una forma de medir distancias (una *métrica*). En párrafos anteriores, el concepto *proximidad* estaba relacionado con un espacio de vectores: se trataba de dar respuesta a la cuestión de cómo elegir el vector de pesos más próximo al vector de entrada. En este caso, nuestro espacio es la segunda capa de la red, que hemos supuesto bidimensional. Dada una neurona de esa capa (la neurona ganadora) cuyos pesos vamos a modificar, se plantea la cuestión de hasta dónde se extenderá su influencia o, dicho de otro modo, qué neuronas de sus alrededores verán también sus pesos modificados y en qué medida. Si llamamos Λ_j a la vecindad de la neurona j , podemos definir Λ_j como el conjunto de las neuronas que se encuentran a una distancia menor o igual a d_{max}. Como vemos, al tratar el tema de la proximidad inmediatamente aparece el concepto distancia. De nuevo, podemos decir que existen muchas formas de definir la distancia, pero aquí, por simplicidad, vamos a recurrir de nuevo a la distancia euclídea. Para ello, definamos el vector posición de una neurona de la segunda capa de un mapa de Kohonen $\vec{r}_{u,v}$ como el vector de componentes (u, v) donde u y v son las coordenadas fila y columna de la neurona en la segunda capa que hemos supuesto bidimensional. Por ejemplo, la segunda neurona de la segunda fila del mapa tendrá como vector posición $\vec{r}_{22} = (2, 2)$ y la quinta neurona de la sexta fila, $\vec{r}_{65} = (6, 5)$. De

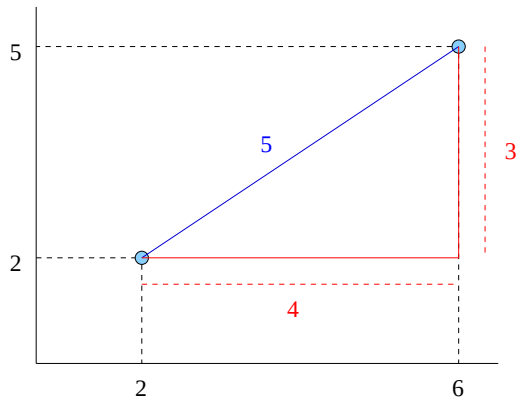


Figura 3.1: Distancias euclídea y de Manhattan para las posiciones $u = 2, v = 2$ y $u = 6, v = 5$.

acuerdo con la definición de distancia euclídea

$$\|\vec{r}_{65} - \vec{r}_{22}\| = \sqrt{(6-2)^2 + (5-2)^2} = 5 \tag{3.7}$$

Otra métrica que podríamos haber utilizado es la distancia de Manhattan que se define como la suma de los valores absolutos de las diferencias en cada coordenada. Para dos vectores $x = (x_1, x_2, ..., x_n)$ e $y = (y_1, y_2, ..., y_n)$ de n componentes tenemos

$$d_{Manhattan} = \|\vec{x} - \vec{y}\|_{Manhattan} = \sum_{i=1}^{n} |x_i - y_i| = |x_1 - y_1| + |x_2 - y_2| + ... + |x_n - y_n| \tag{3.8}$$

En el ejemplo anterior, la distancia de Manhattan entre $\vec{r}_{22}$ y $\vec{r}_{6,5}$ sería $|2-6| + |2-5| = 4+3 = 7$. La distancia de Manhattan es siempre mayor o igual a la distancia euclídea que, en dos dimensiones corresponde a la distancia más corta entre dos puntos, como se puede comprobar en la figura 3.1.

Emplearemos la métrica de Manhattan en alguno de los ejemplos que se incluyen al final de este capítulo. Sin embargo en lo que sigue seguiremos empleando la distancia euclídea para medir proximidades.

Una vez definida la vecindad, vamos a estudiar cómo modificar los pesos de la conexiones. La fórmula más simple de actualización de los pesos de la neurona ganadora y de las que se encuentran próximas a ella es:

$$\Delta\vec{\omega}_i = \alpha \cdot (\vec{x} - \vec{\omega}_i) \tag{3.9}$$

donde i es un subíndice que representa a la neurona ganadora y al conjunto de neuronas próximas a ella. α en esta versión simplificada es, como en el capítulo anterior, una constante que regula la magnitud del cambio en los pesos (la velocidad de aprendizaje). Si deseamos que la actualización se realice sin cambios bruscos daremos a α un valor pequeño, próximo a cero. Como veremos más adelante, α no suele ser constante sino que depende del ciclo de actualización en el que nos encontramos y de la proximidad a la neurona ganadora. Por lo general, α será mayor durante los primeros ciclos de actualización e irá disminuyendo a medida que el proceso avance. Asimismo, para un ciclo dado, α toma valores altos y positivos para la neurona ganadora y para

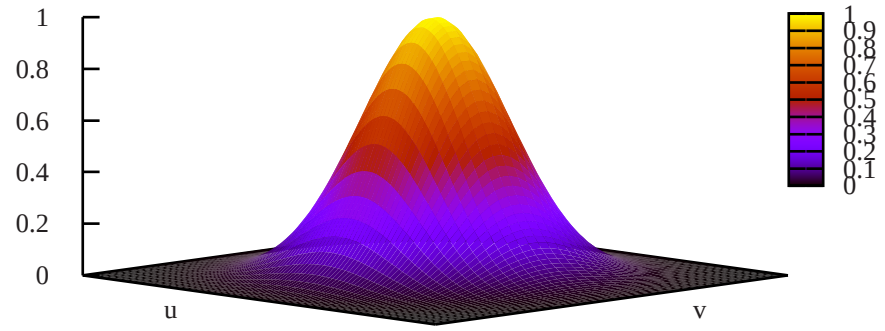


Figura 3.2: Forma del kernel gaussiano dado por la ecuación 3.12 para $n = 1$, $\alpha_0(1) = 1{,}0$ y $\sigma(1) = 3$

neuronas en su vecindad inmediata y va disminuyendo a medida que nos alejamos de ella pudiendo incluso llegar a ser negativa.

Puesto que las neuronas alejadas de la ganadora no experimentan ningún cambio en los pesos, podemos resumir el algoritmo con la ecuación

$$\Delta\vec{\omega}_i = \begin{cases} \alpha \cdot (\vec{x} - \vec{\omega}_i) & \text{si } i \in \Lambda_j \\ 0 & \text{si } i \notin \Lambda_j \end{cases} \tag{3.10}$$

donde Λ_j representa la vecindad de la neurona j. El primer refinamiento posible ya ha sido sugerido en la sección anterior. Consiste en modificar α para que, en lugar de ser constante, dependa del ciclo de actualización. Por ejemplo, podríamos definir $\alpha(n)$ como $\frac{1}{n}$, donde n representa el ciclo en el que se va a realizar la actualización. O, de forma más general, como

$$\frac{A}{B+n} \tag{3.11}$$

con A y B constantes. Con esta definición, la modificación de los pesos, que será la misma para la neurona ganadora y para cualquiera de las que estén en su vecindad, irá disminuyendo a medida que se vayan completando ciclos de actualización.

Hay otras definiciones funcionales posibles y su adecuación sólo puede ser evaluada experimentalmente. La única restricción que impone el método sobre la definición funcional es que $\alpha(n)$ ha de ser una función definida positiva y monótonamente decreciente.

El segundo refinamiento posible consiste en hacer α dependiente de la distancia a

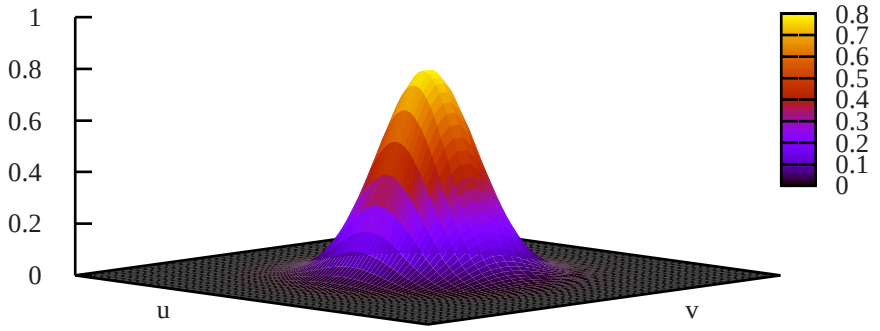


Figura 3.3: Forma del kernel gaussiano dado por la ecuación 3.12 para $n = 10$, $\alpha_0(10) = 0{,}8$ y $\sigma(10) = 2$

la neurona ganadora. Una opción habitual en este caso es la elección de una forma funcional gaussiana como

$$\alpha(n, \|\vec{r}_{uv} - \vec{r}_{u_g v_g}\|) = \alpha_0(n) \cdot \exp(-\frac{\|\vec{r}_{uv} - \vec{r}_{u_g v_g}\|^2}{2\sigma^2(n)}) \tag{3.12}$$

donde $\vec{r}_{u_g v_g}$ es el vector posición de la neurona ganadora, $\alpha_0(n)$, la velocidad de aprendizaje, es una función monótona decreciente (como $\frac{1}{n}$ o la definida por la ecuación 3.11) que hará disminuir el valor de $\alpha(n, \|\vec{r}_{uv} - \vec{r}_{u_g v_g}\|)$ a medida que se vayan completando ciclos y $\sigma(n)$ también es una función monótona decreciente que hace que el radio dentro del cual la actualización de los pesos es significativa (en relación con la actualización de los pesos de la neurona ganadora) también disminuya a medida que avanzamos ciclos.

Las figuras 3.2-3.4 proporcionan una intuición visual de la forma de $\alpha(n, \|\vec{r_{uv}} - \vec{r_{u_g v_g}}\|)$ dada por la ecuación 3.12 y de su evolución con el paso de los ciclos .

Podemos comprobar cómo, a medida que avanzan los ciclos, el máximo del kernel gaussiano disminuye debido a la función[b] α_0 y el radio del kernel (que mide el tamaño de la región de influencia) también disminuye al disminuir $\sigma(n)$.

[b] que, en este caso vendría dada por la ecuación

$$\alpha_0(n) = \frac{36}{35 + n} \tag{3.13}$$

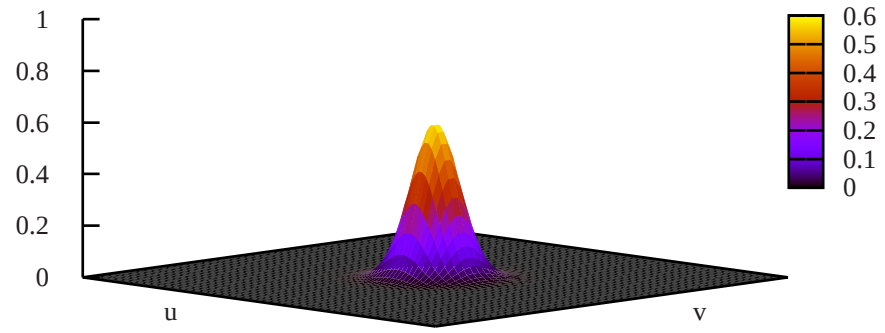


Figura 3.4: Forma del kernel gaussiano dado por la ecuación 3.12 para $n = 25$, $\alpha_0(25) = 0{,}6$ y $\sigma(25) = 1$

Ejemplo 2: Autoorganización en una dimensión

Se desea crear un mapa autoorganizado en una sola dimensión de diez componentes a partir del conjunto de vectores del cuadro 3.1 suponiendo que la vecindad de la neurona ganadora está compuesta en todos los ciclos de actualización de pesos por todas aquellas neuronas que se encuentran a una distancia de manhattan igual a uno (lo que para este mapa unidimensional representa las neuronas inmediatamente a la derecha o izquierda de la ganadora) y que la velocidad de aprendizaje α es constante e igual a 1 para la neurona ganadora y 0.5 para el resto de neuronas en su vecindad. Supóngase asimismo que en cada ciclo los vectores se presenta a la red de forma aleatoria.

Si representamos los vectores bidimensionales de entrada como puntos del plano obtendremos la gráfica de la figura 3.2.

Dadas las características del mapa, la red neuronal estará compuesta por una capa de entrada con dos neuronas (puesto que los vectores de entrada tienen dos componentes) y una capa de clasificación de diez neuronas. Cada vector de entrada, al final del proceso de entrenamiento activará la neurona cuyo vector de pesos se encuentre más próximo al vector de entrada y, dadas las características de los mapas de Kohonen, esperamos que vectores próximos en el espacio de entradas activen neuronas próximas en la capa de clasificación. Para inicializar la red vamos a asignar a todos los pesos de la capa de clasificación el valor $0{,}5$.Por lo tanto, si representamos los vectores de pesos en una gráfica, obtendremos un único punto situado en las coordenadas $(0{,}5, 0{,}5)$. Podríamos haberles asignado valores aleatorios en el intervalo $[-1, 1]$ o en el intervalo $[0, 1]$ y el resultado final no habría sido diferente, pero entonces la gráfica inicial de los pesos mostraría, no un único punto, sino 10 puntos distribuidos por el intervalo elegido.

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}
0	0.0159	0.0317	0.0476	0.0634	0.0792	0.0951	0.1108	0.1266	0.1423
1.0000	0.9999	0.9995	0.9989	0.9980	0.9969	0.9955	0.9938	0.9920	0.9898
x_{11}	x_{12}	x_{13}	x_{14}	x_{15}	x_{16}	x_{17}	x_{18}	x_{19}	x_{20}
0.1580	0.1736	0.1893	0.2048	0.2203	0.2358	0.2511	0.2665	0.2817	0.2969
0.9874	0.9848	0.9819	0.9788	0.9754	0.9718	0.9679	0.9638	0.9595	0.9549
x_{21}	x_{22}	x_{23}	x_{24}	x_{25}	x_{26}	x_{27}	x_{28}	x_{29}	x_{30}
0.3120	0.3271	0.3420	0.3569	0.3717	0.3863	0.4009	0.4154	0.4298	0.4441
0.9501	0.9450	0.9397	0.9341	0.9284	0.9224	0.9161	0.9096	0.9029	0.8960
x_{31}	x_{32}	x_{33}	x_{34}	x_{35}	x_{36}	x_{37}	x_{38}	x_{39}	x_{40}
0.4582	0.4723	0.4862	0.5000	0.5137	0.5272	0.5406	0.5539	0.5671	0.5801
0.8888	0.8815	0.8738	0.8660	0.8580	0.8497	0.8413	0.8326	0.8237	0.8146
x_{41}	x_{42}	x_{43}	x_{44}	x_{45}	x_{46}	x_{47}	x_{48}	x_{49}	x_{50}
0.5929	0.6056	0.6182	0.6306	0.6428	0.6549	0.6668	0.6785	0.6901	0.7015
0.8053	0.7958	0.7861	0.7761	0.7660	0.7557	0.7453	0.7346	0.7237	0.7127
x_{51}	x_{52}	x_{53}	x_{54}	x_{55}	x_{56}	x_{57}	x_{58}	x_{59}	x_{60}
0.7127	0.7237	0.7346	0.7453	0.7557	0.7660	0.7761	0.7861	0.7958	0.8053
0.7015	0.6901	0.6785	0.6668	0.6549	0.6428	0.6306	0.6182	0.6056	0.5929
x_{61}	x_{62}	x_{63}	x_{64}	x_{65}	x_{66}	x_{67}	x_{68}	x_{69}	x_{70}
0.8146	0.8237	0.8326	0.8413	0.8497	0.8580	0.8660	0.8738	0.8815	0.8888
0.5801	0.5671	0.5539	0.5406	0.5272	0.5137	0.5000	0.4862	0.4723	0.4582
x_{71}	x_{72}	x_{73}	x_{74}	x_{75}	x_{76}	x_{77}	x_{78}	x_{79}	x_{80}
0.8960	0.9029	0.9096	0.9161	0.9224	0.9284	0.9341	0.9397	0.9450	0.9501
0.4441	0.4298	0.4154	0.4009	0.3863	0.3717	0.3569	0.3420	0.3271	0.3120
x_{81}	x_{82}	x_{83}	x_{84}	x_{85}	x_{86}	x_{87}	x_{88}	x_{89}	x_{90}
0.9549	0.9595	0.9638	0.9679	0.9718	0.9754	0.9788	0.9819	0.9848	0.9874
0.2969	0.2817	0.2665	0.2511	0.2358	0.2203	0.2048	0.1893	0.1736	0.1580
x_{91}	x_{92}	x_{93}	x_{94}	x_{95}	x_{96}	x_{97}	x_{98}	x_{99}	x_{100}
0.9898	0.9920	0.9938	0.9955	0.9969	0.9980	0.9989	0.9995	0.9999	1.0000
0.1423	0.1266	0.1108	0.0951	0.0792	0.0634	0.0476	0.0317	0.0159	0.0000

Cuadro 3.1: Conjunto de entrenamiento del enunciado.

Comenzamos a ejecutar el algoritmo de creación del mapa introduciendo un vector (seleccionado aleatoriamente de entre los que componen la tabla 3.1) en la capa de entrada. Cualquiera que sea el vector seleccionado, tendremos que las activaciones de la capa de salida serán todas iguales porque todos los vectores de pesos son iguales y, por lo tanto, no hay neurona ganadora (todas están igualmente próximas al vector de entrada). Supongamos que el vector seleccionado es el primero de la tabla. Entonces, la distancia euclídea entre ese vector y el vector de pesos de cualquier neurona de la capa de clasificación será

$$d = \sqrt{(x_1 - \omega_{j1})^2 + (x_2 - \omega_{j2})^2} = \sqrt{(1{,}0 - 0{,}5)^2 + (0{,}0 - 0{,}5)^2} = 0{,}7071 \quad (3.14)$$

Para este caso (inhabitual porque se suelen inicializar los pesos de forma aleatoria) en el que hay más de una neurona ganadora (en este caso todas son ganadoras) el

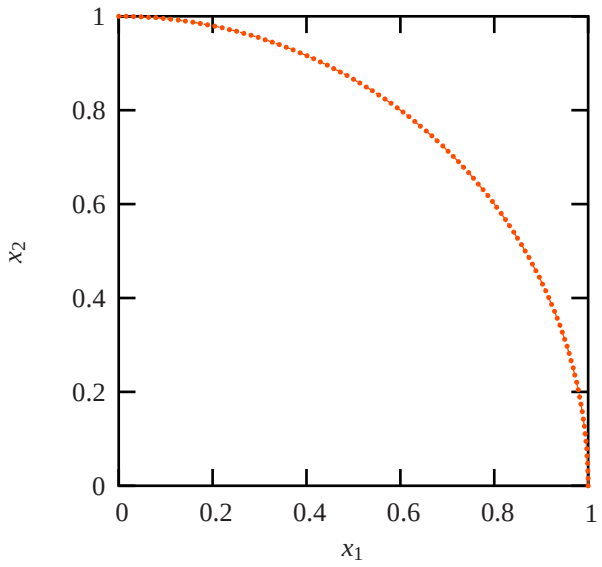


Figura 3.5: Conjunto inicial de entrenamiento del mapa unidimensional.

algoritmo considera ganadora a la primera neurona según su ordenación espacial. La actualización de los pesos de la neurona ganadora y su vecindad será según la ecuación 3.10

$$\Delta\omega_{11} = \alpha \cdot (x_1 - \omega_{11}) = 1{,}0 \cdot (0{,}0 - 0{,}5) = -0{,}5 \qquad \Delta\omega_{12} = \alpha \cdot (x_2 - \omega_{12}) = 1{,}0 \cdot (1{,}0 - 0{,}5) = 0{,}5$$
$$\Delta\omega_{21} = \alpha \cdot (x_1 - \omega_{21}) = 0{,}5 \cdot (0{,}0 - 0{,}5) = -0{,}25 \qquad \Delta\omega_{22} = \alpha \cdot (x_2 - \omega_{22}) = 0{,}5 \cdot (1{,}0 - 0{,}5) = 0{,}25$$

donde la única neurona en la vecindad de la ganadora es la segunda neurona de la capa de clasificación. Cualquier otra neurona de esa capa se encuentra a una distancia de Manhattan superior a 1.

Los pesos de las neuronas del mapa unidimensional tendrán entonces los siguientes pesos:

$\vec{\omega}_1$	$\vec{\omega}_2$	$\vec{\omega}_3$	$\vec{\omega}_4$	$\vec{\omega}_5$	$\vec{\omega}_6$	$\vec{\omega}_7$	$\vec{\omega}_8$	$\vec{\omega}_9$	$\vec{\omega}_{10}$
0	0.25	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
1.0	0.75	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5

A continuación se introduce en la red un nuevo vector extraido aleatoriamente de la tabla. Supongamos que ese vector es el número 17 de componentes (0.2511,0.9679). Entonces la distancia al vector de pesos de la primera neurona clasificadora será

$$d = \sqrt{(x_1 - \omega_{j1})^2 + (x_2 - \omega_{j2})^2} = \sqrt{(0{,}2511 - 0{,}0)^2 + (0{,}9679 - 1{,}0)^2} = 0{,}2531 \tag{3.15}$$

y el resto de distancias se encuentran resumidas en el cuadro 3.2.

$\vec{\omega}_1$	$\vec{\omega}_2$	$\vec{\omega}_3$	$\vec{\omega}_4$	$\vec{\omega}_5$	$\vec{\omega}_6$	$\vec{\omega}_7$	$\vec{\omega}_8$	$\vec{\omega}_9$	$\vec{\omega}_{10}$
0.25	0.22	0.53	0.53	0.53	0.53	0.53	0.53	0.53	0.53

Cuadro 3.2: Distancias del vector de entrada 17 a los distintos vectores de pesos.

Vemos que la neurona cuyo vector de pesos está más próximo al vector de entrada es la segunda y que su vecindario comprende a la primera y tercera neuronas (todas ellas señaladas en la tabla con un fondo distinto).Si volvemos a aplicar el algoritmo de actualización de pesos obtendremos

$$\begin{aligned}
\Delta\omega_{11} &= \alpha \cdot (x_1 - \omega_{11}) = 0{,}5 \cdot (0{,}25 - 0{,}0) = 0{,}125 \\
\Delta\omega_{12} &= \alpha \cdot (x_2 - \omega_{12}) = 0{,}5 \cdot (0{,}97 - 1{,}0) = -0{,}015 \\
\Delta\omega_{21} &= \alpha \cdot (x_1 - \omega_{21}) = 1{,}0 \cdot (0{,}25 - 0{,}25) = 0{,}00 \\
\Delta\omega_{22} &= \alpha \cdot (x_2 - \omega_{22}) = 1{,}0 \cdot (0{,}97 - 0{,}75) = 0{,}22 \\
\Delta\omega_{31} &= \alpha \cdot (x_1 - \omega_{31}) = 0{,}5 \cdot (0{,}25 - 0{,}5) = -0{,}125 \\
\Delta\omega_{32} &= \alpha \cdot (x_2 - \omega_{32}) = 0{,}5 \cdot (0{,}97 - 0{,}5) = 0{,}235
\end{aligned}$$

donde recordamos que el valor de α depende de la proximidad a la neurona ganadora y vale 1 para dicha neurona, 0,5 para neuronas a una distancia de Manhattan igual a uno y 0 para el resto (que por lo tanto no se actualizan).

$\vec{\omega}_1$	$\vec{\omega}_2$	$\vec{\omega}_3$	$\vec{\omega}_4$	$\vec{\omega}_5$	$\vec{\omega}_6$	$\vec{\omega}_7$	$\vec{\omega}_8$	$\vec{\omega}_9$	$\vec{\omega}_{10}$
0.125	0.25	0.375	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.985	0.97	0.735	0.5	0.5	0.5	0.5	0.5	0.5	0.5

Cuadro 3.3: Valor de los pesos tras la presentación de un nuevo vector de entrada a la red.

Los pesos que obtendríamos como resultado de esta actualización se recogen en el cuadro 3.3.Ténganse en cuenta las posibles discrepancias debidas al redondeo.

El proceso continua hasta calcular las actualizaciones de los vectores de pesos correspondiente a los 98 vectores de entrada restantes, lo que completa el primer ciclo o época. Tras la realización de 100, 200 y 500 ciclos, los 10 vectores de pesos, que inicialmente eran iguales $(0{,}5, 0{,}5)$, tendrán valores distribuidos según la figura 3.6.Vemos que efectivamente, la capa de clasificación ha distribuido sus pesos por todo el espacio de entrada y, dado que los vectores de entrada se repartían uniformemente por dicho espacio, los vectores de pesos se distribuyen asimismo de forma casi uniforme. Vemos también que vectores de entrada próximos van a activar neuronas ganadoras próximas entre sí en la capa de clasificación.

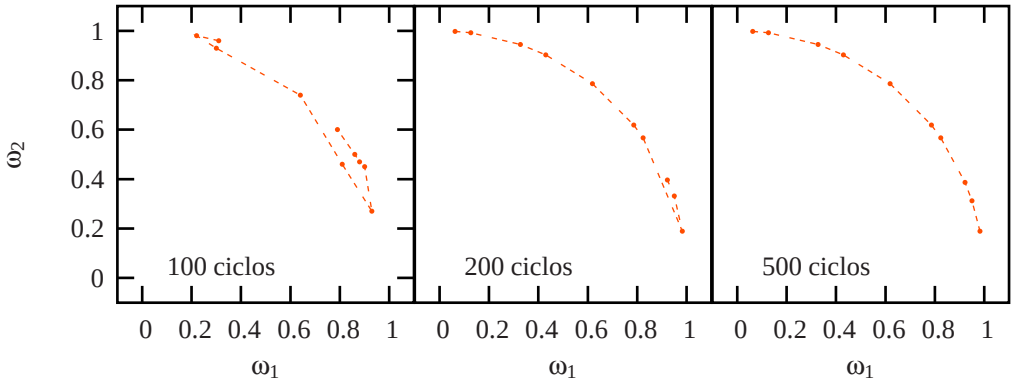


Figura 3.6: Evolución de los vectores de pesos de las 10 neuronas del mapa unidimensional del ejemplo.

Ejemplo 3:

Como ampliación del ejemplo anterior vamos a considerar la creación de un mapa bidimensional de dimensiones 5×6 (30 neuronas en total, ver figura 3.7) a partir del conjunto de 200 vectores de entrada representado en la figura 3.8. Tendremos por lo tanto treinta vectores de pesos (uno para cada neurona del mapa) de dos componentes cada uno (una para cada componente del vector de entrada). Supondremos, como en el caso anterior, que los treinta vectores han sido inicializados con el mismo valor de sus componentes. Sean estos valores $-0{,}0017$ y $-0{,}0084$ por ejemplo (valores obtenidos aleatoriamente)[c].

Vamos a introducir también un valor de α dependiente del ciclo y de la distancia euclídea a la neurona ganadora. Para calcular el valor de α en el ciclo n de la neurona que ocupa la fila u y columna v cuando la introducción de un vector de entrada ha dado lugar a que la neurona que se encuentra en la fila u_g y columna v_g fuera la neurona ganadora, tenemos

$$\alpha(n, \|\vec{r}_{uv} - \vec{r}_{u_g v_g}\|) = \frac{1}{2+n} \cdot \exp\left(-\frac{\|\vec{r}_{uv} - \vec{r}_{u_g v_g}\|^2}{2(\sigma(n))^2}\right) \tag{3.16}$$

donde, recordando la notación empleada anteriormente, $\vec{r}_{u_g v_g}$ es el vector posición en la capa bidimensional de clasificación de la neurona ganadora, $\|\vec{r}_{uv} - \vec{r}_{u_g v_g}\|$ es la distancia entre la neurona cuyos pesos vamos a actualizar y la neurona ganadora, la velocidad de aprendizaje durante el ciclo de actualizaciones n viene dada por $\alpha_0(n) = \frac{1}{2+n}$ y $\sigma(n)$ está relacionada con la anchura de la distribución gaussiana durante el ciclo n y viene dada por $\frac{1}{n}$.

No vamos a listar los 200 vectores de entrenamiento como hicimos en el ejemplo anterior por motivos de espacio. Baste para ilustrar los primeros pasos del cálculo enumerar tres vectores $x_1 = (-0{,}5, -0{,}57)$, $x_2 = (0{,}0, 0{,}43)$ y $x_3 = (-0{,}42, 0{,}5)$ de los 200 que forman el conjunto de entrenamiento a partir del cual vamos a crear el mapa.

[c]Como ya comentamos en el ejemplo anterior, lo habitual es inicializar los vectores de pesos con valores aleatorios y diferentes entre sí. Aquí mantendremos la igualdad inicial por motivos de sencillez de cálculo

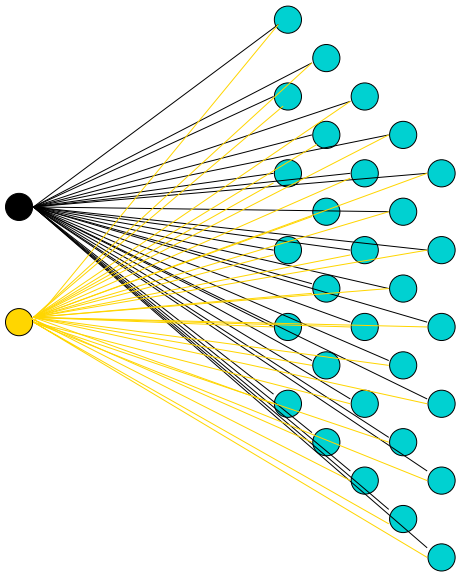


Figura 3.7: Arquitectura del mapa autoorganizado bidimensional del ejemplo.

Vamos a suponer que estos tres vectores han sido los tres primeros seleccionados aleatoriamente de la muestra.

A partir de todos estos datos, vamos a iniciar el proceso de actualización de los pesos (que, como hemos visto, inicialmente son iguales para todas las neuronas de la capa de clasificación). Para el primer ciclo ($n = 1$) y el primer vector de entrada $x_1 = (-0{,}5, -0{,}57)$ tendremos que todas las neuronas se encuentran a la misma distancia puesto que todos sus vectores de pesos son iguales:

$$\|\vec{x}_1 - \vec{\omega}_{uv}\| = \sqrt{(-0{,}5 + 0{,}0017)^2 + (-0{,}57 + 0{,}0084)^2} = 0{,}7508 \tag{3.17}$$

Siguiendo el mismo criterio que en el ejemplo anterior, en el caso de que los vectores de pesos de dos o más neuronas se encuentren a la misma distancia del vector de entrada consideraremos neurona ganadora a la primera en orden topológico (la que se encuentre en la fila más próxima a la primera y, en caso de que dos neuronas de la misma fila se encuentren a la misma distancia, la que se encuentre en la columna más próxima a la primera). En nuestro caso se trata de la que se encuentra en la primera fila y primera columna. Por lo tanto, $\vec{r}_{u_g v_g} = \vec{r}_{11}$

$$\Delta\vec{\omega}_{u_g v_g} = \alpha \cdot (\vec{x} - \vec{\omega}_{u_g v_g}) = \frac{1}{3} \cdot \exp\left(-\frac{\|\vec{r}_{11} - \vec{r}_{11}\|^2}{2 \cdot (1)^2}\right) \cdot (\vec{x} - \vec{\omega}_{u_g v_g}) \tag{3.18}$$

La ecuación vectorial anterior se descompone en dos, una para cada componente de $\vec{\omega}_{u_g v_g}$

$$\Delta\omega_{u_g v_g 1} = \frac{1}{3} \cdot 1{,}0 \cdot (-0{,}5 + 0{,}0017) = -0{,}1661$$
$$\Delta\omega_{u_g v_g 2} = \frac{1}{3} \cdot 1{,}0 \cdot (-0{,}57 + 0{,}0084) = -0{,}1872$$

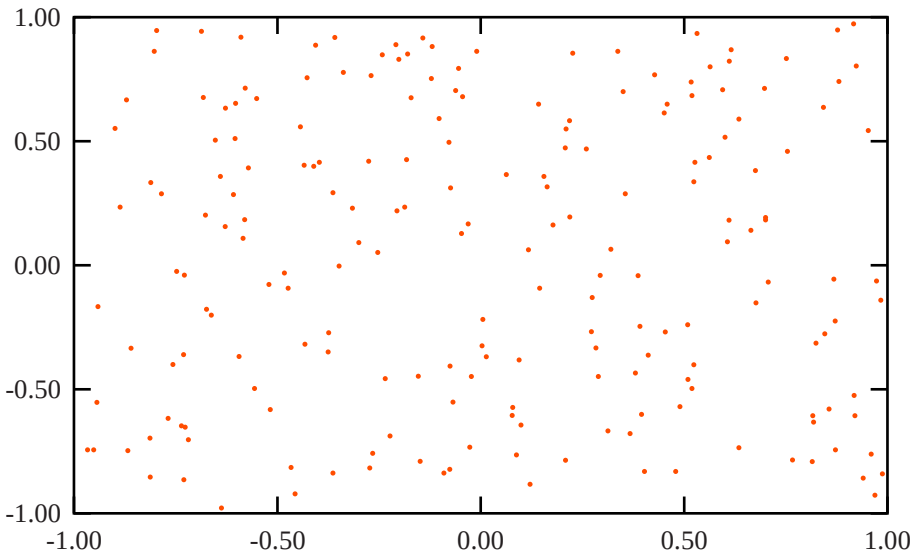


Figura 3.8: Conjunto de entrenamiento para el ejemplo de mapa bidimensional autoorganizado.

y, por lo tanto, los nuevos valores de ω_{11} serán

$$\omega_{111} = -0{,}0017 - 0{,}1661 = -0{,}1678$$
$$\omega_{112} = -0{,}0084 - 0{,}1872 = -0{,}1956$$

Como vemos, el nuevo vector de pesos se ha aproximado al vector entrada pero no coincide con él debido a que α_0 tiene un valor inicial menor que 1. Una vez actualizada la neurona ganadora resta calcular los cambios en los vectores de pesos de las neuronas de su vecindad. Empezamos por las neuronas que se encuentran a una distancia de Manhattan igual a 1. Para la neurona de coordenadas $\mathbf{r}_{12}$ tendremos

$$\begin{aligned}
\Delta\omega_{121} &= \frac{1}{3}\cdot\exp\left(-\frac{\|(1,1)-(1,2)\|^2}{2\cdot(1{,}0)^2}\right)\cdot(-0{,}5+0{,}0017)\\
&= \frac{1}{3}\cdot 0{,}6065\cdot(-0{,}5+0{,}0017) = -0{,}1007\\
\Delta\omega_{122} &= \frac{1}{3}\cdot\exp\left(-\frac{\|(1,1)-(1,2)\|^2}{2\cdot(1{,}0)^2}\right)\cdot(-0{,}57+0{,}0084)\\
&= \frac{1}{3}\cdot 0{,}6065\cdot(-0{,}57+0{,}0084) = -0{,}1135
\end{aligned}$$

y por lo tanto

$$\omega_{121} = -0{,}0017 - 0{,}1007 = -0{,}1024$$
$$\omega_{122} = -0{,}0084 - 0{,}1135 = -0{,}1219$$

Para la neurona de coordenadas $\vec{r}_{21}$ tendremos

$$\begin{aligned}
\Delta\omega_{211} &= \frac{1}{3}\cdot\exp(-\frac{\|(1,1)-(2,1)\|^2}{2\cdot(1{,}0)^2})\cdot(-0{,}5+0{,}0017)\\
&= \frac{1}{3}\cdot 0{,}6065\cdot(-0{,}5+0{,}0017) = -0{,}1007\\
\Delta\omega_{212} &= \frac{1}{3}\cdot\exp(-\frac{\|(1,1)-(2,1)\|^2}{2\cdot(1{,}0)^2})\cdot(-0{,}57+0{,}0084)\\
&= \frac{1}{3}\cdot 0{,}6065\cdot(-0{,}57+0{,}0084) = -0{,}1135
\end{aligned}$$

y por lo tanto

$$\begin{aligned}
\omega_{211} &= -0{,}0017-0{,}1007 = -0{,}1024\\
\omega_{212} &= -0{,}0084-0{,}1135 = -0{,}1219
\end{aligned}$$

A continuación, actualizamos los vectores de pesos de las neuronas que se encuentran a una distancia de Manhattan igual a 2: $\vec{r}_{13}, \vec{r}_{22}$ y $\vec{r}_{31}$. Para $\vec{r}_{13}$ tenemos

$$\begin{aligned}
\Delta\omega_{131} &= \frac{1}{3}\cdot\exp(-\frac{\|(1,1)-(1,3)\|^2}{2\cdot(1{,}0)^2})\cdot(-0{,}5+0{,}0017)\\
&= \frac{1}{3}\cdot 0{,}1353\cdot(-0{,}5+0{,}0017) = -0{,}0225\\
\Delta\omega_{132} &= \frac{1}{3}\cdot\exp(-\frac{\|(1,1)-(1,3)\|^2}{2\cdot(1{,}0)^2})\cdot(-0{,}57+0{,}0084)\\
&= \frac{1}{3}\cdot 0{,}1353\cdot(-0{,}57+0{,}0084) = -0{,}0253
\end{aligned}$$

y por lo tanto

$$\begin{aligned}
\omega_{131} &= -0{,}0017-0{,}0225 = -0{,}0242\\
\omega_{132} &= -0{,}0084-0{,}0253 = -0{,}0337
\end{aligned}$$

Con estos cálculos damos por terminado el ejemplo puesto que no resta ya más que repetir el procedimiento iniciado hasta aquí. A medida que vayamos considerando neuronas a mayores distancias de la ganadora, el factor exponencial de α será cada vez menor hasta llegar a un punto en que haga despreciable el valor de $\Delta\vec{\omega}$ con lo que vemos efectivamente que no hace falta definir explícitamente la vecindad de una neurona. En el segundo ciclo $n = 2$ habrá que tener en cuenta que $\alpha_0(2) = \frac{1}{4} = 0{,}25$ y que $\sigma(2) = \frac{1}{2} = 0{,}5$ con lo que los valores de $\Delta\vec{\omega}$ serán una fracción todavía menor de $\|\vec{x}-\vec{\omega}\|$.

Simulación sugerida:

A continuación recomendamos la realización de la siguiente práctica con el applet java incluido en la página web `http://www.ia.uned.es/asignaturas/sbc2/sbc2/applets/som/som.php`. La descripción de la práctica que haremos aquí es exactamente igual de válida para su realización con JavaNNS o SOM_PAK tras el estudio del capítulo 4, aunque en ese caso la definición de los vectores de entrada tendrá que ser mediante un editor de texto y no interactivamente como en el applet.

1. En primer lugar, cree puntos en el espacio de entrada con el botón izquierdo del ratón. Hágalo de forma que los puntos se agrupen en tres o cuatro regiones pequeñas con 5 o 10 puntos. Inicialice los pesos de la red aleatoriamente y comience el entrenamiento de la red. Reduzca progresiva y alternativamente el radio de la vecindad y la velocidad de aprendizaje hasta que los cambios sean inapreciables.

2. ¿Podría explicar por qué se agrupan los vectores de pesos en las mismas zonas que los vectores de entradas? ¿Qué ocurre cuando disminuye el radio de la vecindad? ¿Y cuando disminuye la velocidad de aprendizaje? Pasee el curso por distintas regiones intermedias del espacio de entrada y anote dónde se encuentran las neuronas ganadoras para esos puntos y sus activaciones ¿Podría describir la relación final entre los vectores de entrada, los vectores de pesos y la posición en el mapa de cada neurona ganadora?

3. Repita el proceso pero introduciendo como datos de entrada puntos que formen una curva como la del ejemplo 2. ¿Qué ocurre ahora? ¿Es una buena idea hacer mapas bidimensionales en este caso? ¿Por qué?

4. Repita de nuevo el proceso pero introduciendo como datos de entrada puntos que se distribuyan uniformemente en el espacio de entrada como en el ejemplo del mapa bidimensional. ¿Qué ocurre ahora? ¿Es una buena idea hacer mapas bidimensionales en este caso? ¿Por qué?

5. Intente ahora representar una agrupación de datos más compleja como, por ejemplo, una elipse con un grupo de puntos en su centro. ¿Cómo representa la red el espacio de entradas? ¿Por qué en unos casos lo hace mejor que en otros?

Recuerde que la utilidad de los mapas de Kohonen es representar espacios de entrada de alta dimensión en mapas de una o dos dimensiones.

Capítulo 4

Herramientas de ayuda en el desarrollo de redes neuronales

Objetivo: En este tema vamos a explicar la diferencia entre las redes neuronales artificiales, tal y como fueron concebidas en un principio, y sus realizaciones prácticas más habituales. Como veremos, éstas pierden algunas de las ventajas y propiedades que les conferían sus definiciones iniciales a cambio de una mayor flexibilidad, al ser simuladas mediante ordenadores secuenciales Von Neumann. Aprenderemos el manejo de un programa de simulación de redes neuronales de libre distribución y su utlización en aplicaciones prácticas. Asimismo, veremos de forma superficial algunos tipos de neurosimuladores que abarcan otras áreas de la neurocomputación más alejadas de las aplicaciones prácticas y los aspectos más básicos de las implementaciones físicas.

4.1. Concepto de simulador de redes neuronales

En capítulos anteriores hemos repasado y analizado los componentes básicos de una red neuronal y disponemos ya de una batería conceptual que nos permitirá abordar aspectos prácticos de la implementación de redes neuronales y de algoritmos de aprendizaje tanto supervisados como no supervisados de los parámetros de una red neuronal (los pesos y polarizaciones). Aquí, pretendemos acercar al alumno a una herramienta particular de ayuda al desarrollo de redes neuronales con la que le resultará sencillo aplicar los conocimiento teóricos que se abordan en los siguientes capítulos a ejemplos concretos y sencillos que se le irán presentando.

El nombre de dicha herramiento es JavaNNS, heredera de SNNS (Stuttgart Neural Network Simulator). En realidad, ambos sólo se diferencian en la interfaz gráfica, puesto que el núcleo del sistema, escrito en lenguaje c, es el mismo. Como su propio nombre indica, se trata de un simulador de redes neuronales desarrollado inicialmente en la Universidad de Stuttgart. En ocasiones, a este tipo de herramientas de ayuda al desarrollo de redes neuronales se las conoce como neurosimuladores, aunque para hablar con propiedad, un neurosimulador es una cosa muy distinta: se trata de un programa con el que puede modelar el funcionamiento de un neurona biológica me-

diante la resolución simultánea de un conjunto de ecuaciones diferenciales que describen el comportamiento de los canales iónicos de una neurona real, su potencial de membrana o las concentraciones de diferentes sustancias en el citoplasma neuronal. Evidentemente, no es ése el tipo de herramientas que nos serán de utilidad en el desarrollo de este curso. La utilidad de los neurosimuladores se pone de manifiesto en otros ámbitos de la computación neuronal más próximos a la neurofisiología y que, en muchos casos, sirven de inspiración en última instancia para modelos artificiales de neurona. Al lector interesado en este tipo de herramientas le recomendamos la lectura de los documentos que aparecen en la página web de la asignatura bajo la solapa de Neurosimuladores. Nosotros en lo que sigue nos circunscribiremos a los simuladores de redes neuronales y, en particular, al que se menciona al comienzo del párrafo, es decir, a JavaNNS.

Los simuladores de redes neuronales son programas modulares que comprenden bibliotecas de componentes de forma que el usuario no necesita programar los algoritmos de cálculo más usuales sino que, bien de forma gráfica, bien en línea de comandos, selecciona la arquitectura de la red neuronal que desea construir (el número de capas, el tipo de neuronas que componene cada una de ellas y su conectividad) y el algoritmo de aprendizaje que considera apropiado, y el simulador realiza todo el proceso tanto de inicialización como de aprendizaje de la red, almacenando sus configuraciones en ficheros externos que pueden ser modificados o reutilizados en cualquier momento. Obviamente, cada simulador tiene unas especificaciones particulares que nos obligan a dar a nuestros patrones de entrada/salida un formato determinado pero, en conjunto, el beneficio que obtenemos al ahorrarnos la implementación de modelos de red/aprendizaje tan comunes resulta muy superior al esfuerzo de adaptación de los datos.

4.2. Java Neural Network Simulator (JavaNNS)

4.2.1. Instalación

JavaNNS puede ser descargado desde la página web de la Universidad de Tübingen [a] Para ejecutar JavaNNS es necesario instalar previamente el entorno de ejecución Java (Java Runtime Environment, JRE) o el kit de desarrollo de software (Software Development Kit, SDK). Ambos se pueden descargar gratuitamente de la página web de Sun Microsystems[b]. [c]

JavaNNS se distribuye en forma de fichero comprimido zip para plataformas Windows, o tar.gz para sistemas operativos basados en unix (Linux o Solaris). Una vez descomprimido, el fichero genera los siguientes ficheros y carpetas:

---

[a] http://www-ra.informatik.uni-tuebingen.de/software/JavaNNS/welcome_e.html

[b] http://java.sun.com/j2se/1.4.2/download.html

[c] Si el alumno tiene alguna dificultad instalando cualquiera de los dos, póngase en contacto con el equipo docente de la asignatura en la dirección de correo electrónico habitual.

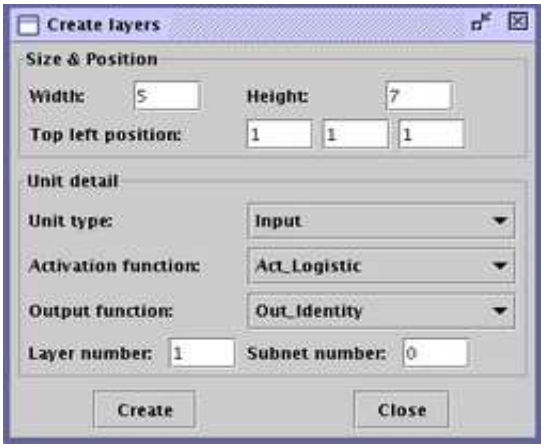


Figura 4.1: create-layers

1. JavaNNS.jar - el archivo java que contiene las clases del interfaz de usuario
2. examples - carpeta de ejemplos
3. manual - carpeta con el manual

Los usuarios de un sistema operativo basado en el kernel de Linux, pueden ejecutar JavaNNS mediante la orden java -jar JavaNNS.jar en línea de comandos; los usuarios de Windows, simplemente pulsando dos veces el ratón en el icono del fichero JavaNNS.jar.

4.2.2. Creación de una red neuronal

La primera vez que se ejecuta, JavaNNS crea una copia del núcleo del simulador en la carpeta que el usuario elija mediante la ventana de diálogo correspondiente. La ubicación de la librería queda almacenada en el fichero JavaNNS.properties en el directorio personal del usuario. Una vez realizada esta operación una única vez (la primera), el sistema operativo arranca la interfaz gráfica basada en Java. Lo primero que vemos es el espacio de trabajo vacío salvo por una ventana de título default <1>. En esta ventana se visualizarán las redes neuronales que creemos y, por supuesto, podemos aumentar o disminuir su tamaño de igual manera que se haría con un entorno de ventanas de los habituales. A la izquierda de la ventana nos encontramos con la escala de color que se empleará para mostrar la magnitud de distintas variables que podemos representar. Por defecto éstas son la activación de las neuronas (units) y los valores de los pesos que conectan unas con otras (links).

Crear una red neuronal consiste en JavaNNS en definir la arquitectura: seleccionar un número de capas, la dimensión o número de neuronas de cada una de ellas y, finalmente, la conectividad entre capas e *intra* capa. Dicho proceso se realiza mediante la ventana de diálogo Create layers que se abre al seleccionar en la pestaña Tools sucesivamente la opción Create y Layers (ver figura 4.1).

En dicha ventana, podemos seleccionar la anchura (Width) y la altura (Height) de cada

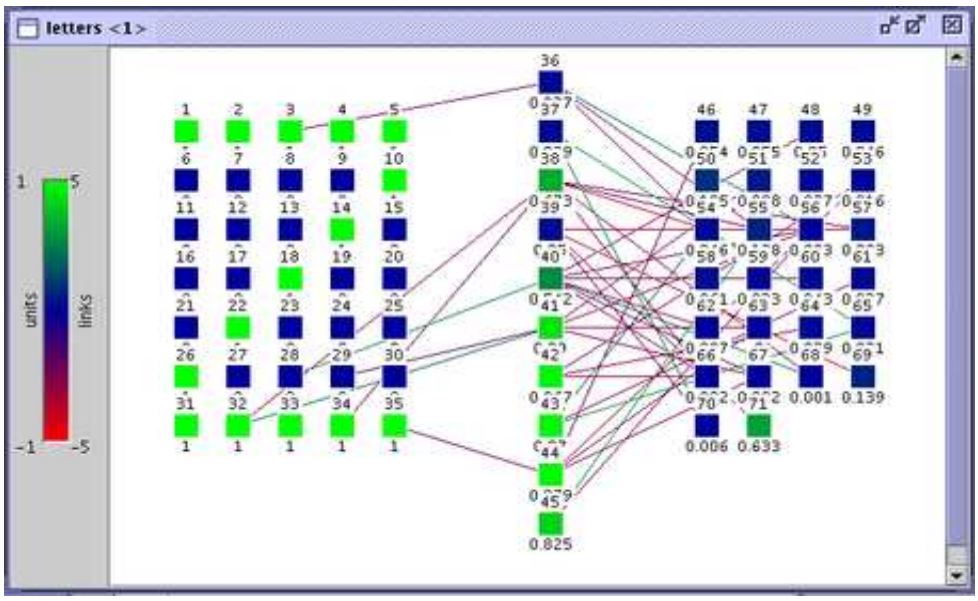


Figura 4.2: Ventana de Display con la red neuronal de ejemplo de JavaNNS.

capa de la red además del tipo de neuronas que la componen (Unit type), la función de activación (Activation function) y la función de salida (Output function). En la misma ventana se pueden especificar otras opciones en las que, por el momento, no vamos a entrar. Supongamos que deseamos crear una red neuronal para clasificar una imagen binaria bidimensional como una de las 26 letras mayúsculas del alfabeto anglosajón. Supongamos también que la imagen bidimensional tiene 7 filas y 5 columnas. Entonces, espacio de entrada de la red tendrá 35 neuronas sensoras que, para que sea más intuitivo, dispondremos también como una matriz bidimensional de 7 filas y 5 columnas. El espacio de salida lo representaremos como una capa de 26 neuronas, una para cada letra del alfabeto. La red completa estará compuesta de tres capa: la primera será la capa de entrada bidimensional, la segunda, una capa oculta y la tercera la capa de salida. La figura 4.2 muestra la ventana de Display de la red después del proceso de creación que describimos a continuación.

El procedimiento para crear la primera capa será introducir el valor 5 en el campos de anchura y 7 en el de altura de la capa y seleccionar el tipo Input en el menú desplegable que aparece al pinchar con el ratón sobre el campo a la derecha de Unit type. Por ahora, dejemos como funciones de activación y salida las que figura por defecto en la ventana, es decir, Act_Logistic, y Out_Identity. Al pulsar con el ratón sobre la opción Create aparece automáticamente en la ventana que representa la red neuronal, la matriz de neuronas de entrada que acabamos de definir. La ventana de creación de capas se modifica automáticamente para indicarnos que la posición de la esquina superior izquierda de la próxima capa se habrá desplazado (para evitar que las distintas capas se superpongan en la ventana de visualización) y que dicha próxima capa (la segunda) tendrá, como cabía esperar, el número de orden (Layer number) 2. Si queremos visualizar las capas en vertical, modificaremos los valores que aparecen en la ventana Create layers para que figure 1 en el campo de anchura y 10 en el de altura (valores de la segunda capa). Además, hemos de cambiar el tipo de neurona a Hidden (oculta).

Al pulsar de nuevo Create aparece en la ventana de visualización de la red una nueva capa cuyas neuronas se disponen en alineamiento vertical. Si hubiéramos invertido los valores de anchura y altura la capa se desplegaría en horizontal. Finalmente, repetimos el procedimiento para la capa de salida seleccionando los valores 1 para la anchura y 26 para la altura y Output para el tipo de neurona. Ya podemos pulsar Close y en la ventana de visualización podremos contemplar la red neuronal recién creada.

Sólo resta conectar las neuronas. Para ello, seleccionamos de nuevo la pestaña Tools y, en ella, Create y Connections. Así habremos abierto una ventana denominada Create links que nos ofrece tres posibilidades: Connect selected units (conectar unidades seleccionadas), Connect feed-forward (conectividad hacia adelante) y Auto-associative (para redes autoasociativas). En nuestro ejemplo, seleccionaremos la conectividad hacia adelante lo que al pulsar sobre Connect con el ratón hará que aparezcan en la ventana de visualización de la red enlaces que van de la neurona de entrada a todas las de la capa oculta, y de cada una de las neuronas de la capa oculta a la neurona de salida. La opción feed-forward crea todas las posibles conexiones entre una capa y la siguiente.

4.2.3. Entrenamiento

Antes de comenzar a describir el proceso de especificación del modo de aprendizaje de la red, vamos a cargar el fichero que contiene el conjunto de entrenamiento de la red. Para ello seleccionamos, como suele ser habitual, el menú desplegable titulado File la opción Open. En la ventana de selección de ficheros que aparece, descendemos al directorio examples (ejemplos en inglés) y pinchamos dos veces en el fichero letters.pat. Ese fichero, que incluimos a continuación, fue creado con las especificaciones que se han de seguir para crear nuestros propios ficheros de patrones.

```
SNNS pattern definition file V3.2
generated at Mon Apr 25 18:08:50 1994


No. of patterns : 26
No. of input units : 35
No. of output units : 26

# Input pattern 1:
0 1 1 1 0
1 0 0 0 1
1 0 0 0 1
1 1 1 1 1
1 0 0 0 1
1 0 0 0 1
1 0 0 0 1
# Output pattern 1:
1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0
# Input pattern 2:
1 1 1 1 0
```

```
1 0 0 0 1
1 0 0 0 1
1 1 1 1 0
1 0 0 0 1
1 0 0 0 1
1 1 1 1 0
# Output pattern 2:
0 1 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0
```

Es muy importante atenerse a las especificaciones de este fichero, muy especialmente en lo que se refiere a la cabecera. Ésta consta de ocho líneas como las mostradas (algunas de ellas en blanco) que especifican el número de patrones que contiene el fichero(No. of patterns), el número de entradas (No. of input units) y el de salidas (No. of output units). Las líneas que comienzan con una almohadilla son comentarios y no se ajustan a ninguna regla salvo la mencionada. A continuación se incluyen alternativamente los valores de entrada y salida en el orden en que se hayan numerado las neuronas de la red. Las líneas se pueden terminar en cualquier punto para dar mayor legibilidad al fichero. En este caso, las entradas se presentan en bloques de siete de forma que se pueden entrever en los patrones de entrada las matrices correspondientes a las letras A y B mayúsculas. Una vez cargados los patrones estamos ya en disposición de seleccionar el algoritmo de entrenamiento que deseemos en el Panel de Control (Control Panel) que se activa en el menú desplegable de la pestaña Tools.

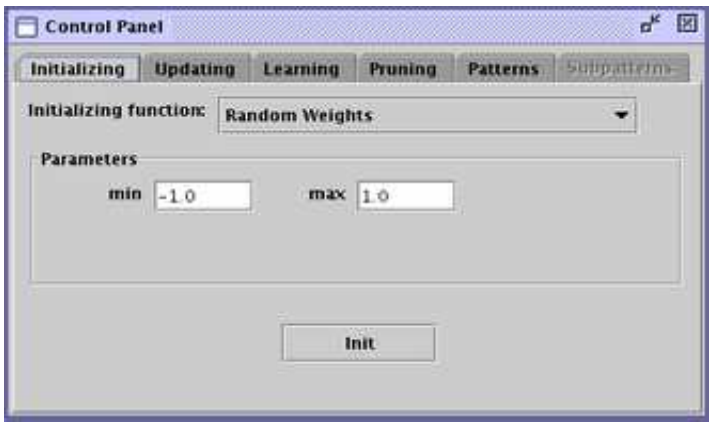


Figura 4.3: init

El Panel de Control consta a su vez de seis pestañas algunas de las cuales pueden estar inactivas (no seleccionables): Initializing que permite elegir el modo de inicialización de los pesos; Updating, que define el orden en que se actulizan los pesos; Learning, que selecciona el algoritmo de aprendizaje y los parámetros que los caracterizan; Pruning, que permite definir un mecanismo de poda que elimina selectivamente sinapsis; Patterns, que nos permite definir qué fichero de patrones de los que hemos abierto con

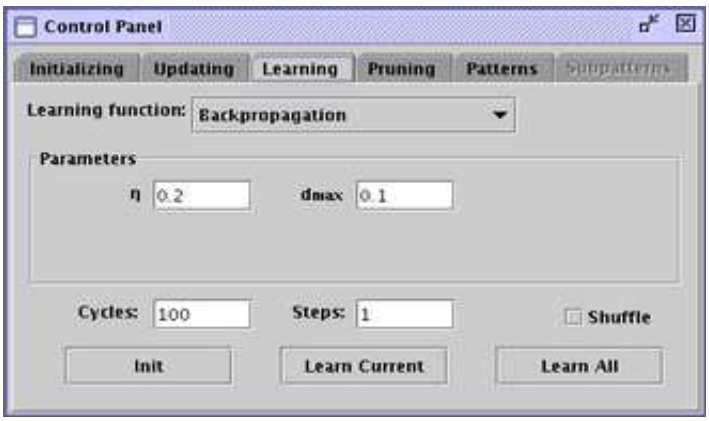


Figura 4.4: learn

JavaNNS vamos a emplear como conjunto de entrenamiento o validación y, finalmente, Subpatterns que se activará cuando hayamos definido subpatrones en el conjunto de entrenamiento (para saber más sobre subpatrones os recomendamos leer el manual de usuario de SNNS).

Para nuestro ejemplo, vamos a dejar seleccionada la función de inicialización Random Weights (Pesos Aleatorios) entre -1.0 y 1.0, la función de actualización Topological Order (Orden Topológico) que actualiza los pesos según el número de orden de la neurona, el algoritmo de aprendizaje Backpropagation (Retropropagación) con parámetros $\eta = 0{,}2$ (velocidad de aprendizaje) y $d_{max} = 0{,}1$ (la máxima diferencia entre el patrón objetivo y el resultado de la red que se considera como despreciable y que, por lo tanto, no se propaga hacia atrás para recalcular los pesos). Cada función de aprendizaje tiene sus propios parámetros y el usuario de JavaNNS debe leer el manual para conocer todos los algoritmos a su disposición y la selección de parámetros óptima para la tarea que desea resolver. En esta misma pestaña de aprendizaje vamos a fijar el valor de la variable Cycles (que nos permite seleccionar el número de veces que se le va a presentar a la red el conjunto entero de patrones de entrenamiento o ciclos) en 150, mientras que la variable Steps (Pasos) que representa el número de patrones que se van a procesar al pulsar el botón Learn Current (aprende el patrón seleccionado actualmente) la dejaremos en su valor por defecto. La opción Shuffle, cuando es seleccionada, nos permite reordenar los patrones aleatoriamente (barajarlos) en cada ciclo de presentación. No vamos a podar ningún peso y, en la pestaña de Patrones dejaremos seleccionado el fichero letters en el campo Training set (Conjunto de entrenamiento). La pestaña, además, nos permite añadir nuevos patrones a los contenidos en el fichero leido y modificar o borrar los ya existentes así como eliminar o añadir nuevos ficheros de patrones y seleccionar una función de procesado de las salidas (Remapping function) que se puede utilizar para re-escalar los valores de salida de los ejemplos. Si se desea realizar preprocesado de los patrones de entrada (ver sección 2.2.1 del capítulo 2) JavaNNS ofrece varias posibilidades entre las que se encuentran la normalización de la longitud euclídea, el cambio de escala lineal (en cuyo caso solicita la pendiente y la ordenada en el origen de la recta) y la umbralización, pero todo ello mediante la utilización de **vistra** y de manera previa a la incorporación del fichero de patrones a JavaNNS. Sin embargo, **vistra** es un proyecto antiguo y desatendido que en la mayoría de los sistemas actuales tendrá problemas para compilar. Por ello, los usuarios de JavaNNS escriben sus propias rutinas de preprocesado como parte del proceso de

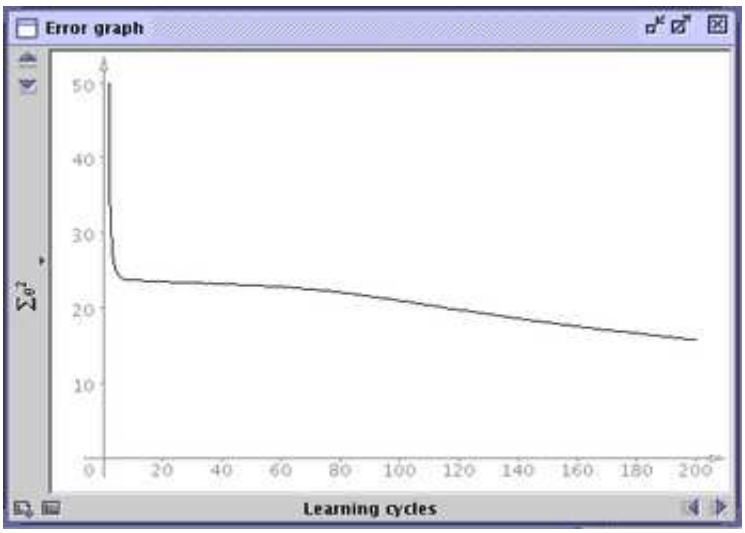


Figura 4.5: error

creación del fichero de patrones.

Antes de volver a la pestaña Learning para comenzar el aprendizaje pulsando la opción Learn All (Aprender todos los patrones), vamos a abrir una ventana gráfica que nos permita visualizar en todo momento la diferencia entre la salida deseada de la red y la salida real. Para ello, seleccionamos en la pestaña View del Menú Principal la opción Error Graph. Con ello aparece la ventana Error Graph que, en su margen izquierdo, presenta otro menú desplegable con las distintas opciones para el eje vertical.

1. Error Cuadrático Total o Summed Squared Error (SSE)

$$SSE = E(\Omega) = \frac{1}{2} \cdot \sum_{n=1}^{N_{\mathcal{A}}} \|\vec{t}[n] - \vec{y}[n]\|^2 = \frac{1}{2} \cdot \sum_{n=1}^{N_{\mathcal{A}}} \sum_{k=1}^{N_s} (t_j - y_k)^2 \qquad (4.1)$$

   que se corresponde con la definición de la fórmula 2.5

2. Error Cuadrático Medio o Mean Squared Error (MSE) definido como

$$MSE = \frac{SSE}{N_{\mathcal{A}}} \qquad (4.2)$$

   donde $N_{\mathcal{A}}$ es el número de patrones. o como

$$MSE = \frac{SSE}{N_s} \qquad (4.3)$$

   donde N_s es el número de neuronas de salida de la red.

Dicha ventana ofrece además la posibilidad de modificar los rangos representados en ambos ejes mediante las cabezas de flecha, borrar las curvas trazadas o superponer una cuadrícula.

Una vez abierta la ventana de representación gráfica del error, volvemos al Panel de Control y pulsamos el botón Learn All en la pestaña Learning, con lo que el núcleo

de JavaNNS ejecuta 150 ciclos de aprendizaje. Como ejercicio, proponemos al lector que varíe la longitud total del aprendizaje bien pulsando varias veces seguidas el botón Learn All, bien cambiando el valor de Cycles y que describa las curvas obtenidas para, por lo menos, dos funciones de aprendizaje: BackPropagation y BackProp_Momentum. ¿Por qué detenemos el aprendizaje cuando aún podríamos disminuir más el error? ¿Cuál de las dos funciones de aprendizaje hace disminuir el error más rápidamente?

4.3. Herramientas de JavaNNS

JavaNNS es la interfaz más moderna de SNNS pero no incorpora todas las herramientas que éste proporciona. Para aquellos alumnos interesados en la creación automatizada de redes neuronales mediante scripts o en el empleo de dichas redes como funciones integrables en c, resumimos aquí una guía de uso de las herramientas batchman y snns2c de SNNS. Para disponer de estas herramientas se deberá instaladar previamente SNNS.

4.3.1. Batchman

Batchman se ejecuta desde línea de comandos con una orden equivalente a unix-prompt>batchman -f file.bat -l file.log. El fichero file.bat estará compuesto de sentencias de control de tipo IF-THEN-ELSE-ENDIF, FOR-TO-DO-ENDFOR, WHILE-DO-ENDWHILE o REPEAT-UNTIL, y de llamadas a funciones del núcleo como las que se enumeran a continuación

Selección de parámetros	Funciones de redes	Funciones de patrones	Funciones especiales
setInitFunc()	loadNet()	loadPattern()	pruneNet()
setLearnFunc()	saveNet()	setPattern()	pruneTrainNet()
setUpdateFunc()	saveResult()	delPattern()	
setPruningFunc()	initNet()		pruneNetNow()
setRemapFunc()	trainNet()		delCandUnits()
setActFunc()	resetNet()		execute()
setCascadeParams()	jogWeights()		print()
setSubPattern()	jogCorrWeights()		exit()
setShuffle()	testNet()		setSeed()
setSubShuffle()			
setClassDistrib()			

El significado de las funciones es evidente a partir de su nombre. Los parámetros que se le han de pasar a cada llamada al núcleo se encuentran en el manual de uso de SNNS y sería demasiado largo detallar aquí todas las posibilidades. Como con cualquier otra duda, recomendamos al alumno interesado que consulte con el equipo docente cualquier duda que pueda surgir al respecto. Aquí solo vamos a incluir un ejemplo comentado de fichero de comandos.

```
# cargar red neuronal del fichero encoder.net
loadNet("encoder.net")
```

```
# cargar fichero de patrones
loadPattern("encoder.pat")
# seleccionar función de inicialización de pesos
setInitFunc("Randomize_Weights", 1.0, -1.0)
# Inicializar pesos
initNet()

# Continuar el entrenamiento de la red mientras el error sumado supere un umbral
#y no se hayan superado los 1000 ciclos
while SSE > 6.9 and CYCLES < 1000 do
# Imprimir el valor del error sumado cada 10 ciclos de entrenamiento
  if CYCLES mod 10 == 0 then
    print ("cycles = ", CYCLES, "  SSE = ", SSE) endif
  trainNet()
endwhile

# Guardar en encoder.res (crearlo si no existe) todos los patrones de entrada y sali
saveResult("encoder.res", 1, PAT, TRUE, TRUE, "create")
# Guardar la red en el estado actual en encoder.trained.net
saveNet("encoder.trained.net")


print ("Cycles trained: ", CYCLES)
print ("Training stopped at error: ", SSE)
```

4.3.2. snns2c

El ejecutable snns2c genera, a partir de una red neuronal dada, una función escrita en ANSI c que pueder ser llamada desde cualquier programa en dicho lenguaje. Dado un comando como snns2c red.net la salida consiste en dos ficheros red.c y red.h que, compilados e incluidos en un programa c mediante la línea

```
#include "red.h"
```

proporcionan una función definida por int red(float *entrada, float *salida, int init) donde entrada y salida son dos punteros a vectores con los valores de entrada y salida a la red neuronal e init es un entero requerido por algunos tipos de red neuronal. De nuevo, recomendamos al alumno la lectura del manual o enviar al equipo docente las dudas que puedan surgir relativas al empleo de snns2c.

4.3.3. Simulación con JavaNNS

En la dirección `http://www.ia.uned.es/asignaturas/sbc2/snns` se pueden obtener dos ficheros útiles para poner en práctica los conocimientos adquiridos hasta ahora. Corresponden al ejemplo de interpolación introducido en el capítulo 2: **entre.pat** que contiene los patrones de entrenamiento y **interp.pat** con puntos intermedios para los que desearíamos obtener una interpolación. Proponemos como ejercicio la creación de la red neuronal descrita anteriormente en JavaNNS, su entrenamiento

y la posterior transformación del resultado con snns2c en una función en lenguaje c para interpolación.

4.4. SOM_PAK

SOM_PAK es un conjunto de programas para la creación y análisis de Mapas Autoorganizados (Self-Organized Maps). Como se ve en más detalle en el capítulo 3, los mapas autoorganizados son el resultado de cuantizar de manera ordenada un espacio de entrada de alta dimensión mediante un conjunto de vectores de referencia. El proceso de creación del mapa de tipo no supervisado y se emplea habitualmente para obtener agrupamientos de datos (*clustering*) complejos.

4.4.1. Instalación

El conjunto de programas de los que vamos a tratar en esta sección se pueden obtener en la página `http://www.cis.hut.fi/research/som_pak`. Los alumnos que trabajen en sistemas basados en MS-DOS deben descargar som-*.exe donde el asterisco corresponde al número de versión (p3r1 en el momento de escribir este documento). Al ejecutar este archivo se extraen automáticamente todos los ficheros necesarios para la instalación. Aquéllos que trabajen con sistemas UNIX deben descargarse el fichero som_pak-*.tar, donde, de nuevo, * representa la versión del programa (3.1 en el momento de escribir este documento). Una vez descargado el fichero, que contiene en cualquiera de sus versiones el código fuente, los makefiles y ejemplos, la instalación depende del sistema operativo. Como alternativa para aquellos alumnos que no dispongan de compilador c++, en dicha misma página se pueden encontrar dos directorios con los ficheros binarios de la última distribución compilados en MS-DOS y en Windows.

MS-DOS

Para instalar SOM_PAK en un sistema operativo basado en MS-DOS basta ejecutar el fichero

```
> som_p3r1
```

con lo que se crea el directorio som_pak.3r1. Si cambiamos a este directorio

```
> cd som_pak.3r1
```

encontraremos el fichero makefile.dos que nos ha de servir para compilar el código fuente mediante

```
> copy makefile.dos
> make
```

UNIX

En sistemas basados en UNIX, el procedimiento es similar: se extrae el fichero tar mediante

```
> tar xvf som_pak-3.1.tar
```

y se compila mediante la serie de comandos

```
> cd som_pak-3.1
> cp makefile.unix Makefile
> make
```

4.4.2. Uso y ejemplos

En lo que sigue se describe el proceso de creación de un mapa autoorganizado mediante una herramienta (vfind) que coordina los binarios de inicialización de los vectores de referencia (randinit), de entrenamiento (vsom), de calibración (vcal) y de uso (visual). Para obtener información detallada de cada uno de ellos remitimos al lector al manual de uso de SOM_PAK.

Antes de explicar el uso de vfind, es necesario exlicar el formato ASCII de los ficheros de entrada. Éstos han de ser encabezados con una línea que especifique necesariamente la dimensión de los vectores empleados para crear el mapa. Opcionalmente se pueden añadir los siguientes campos (lo que hará que vfind no pregunte por los valores correspondientes):

1. Tipo de topología: *hexa* para mapas en los que el vecindario de cada neurona está compuesto de seis neuronas o *rect* para vecindario rectangular
2. Dimensión horizontal del mapa
3. Dimensión vertical del mapa
4. Tipo de función núcleo o kernel: *bubble* para funciones constantes en la vecindad o *gaussian* para núcleos que disminuyen con la distancia según una gaussiana.

Un ejemplo de tal cabecera sería

```
3 hexa 20 10 gaussian
```

e iría seguida de un número indeterminado de vectores de tres componentes a partir del cual se constuirá el mapa. Es lícito incluir comentarios en el fichero antecedidos por el símbolo almohadilla (#). Asimismo se puede añadir al final de cada vector una etiqueta que identifique dicho vector como perteneciente a una clase determinada. Por ejemplo,

```
#primer vector
1 0 0 clase3
#segundo vector
0 0 1 clase1
#tercer vector
0 1 0 clase2
#cuarto vector
0 1 1 clase2
...
```

lo que permitiría, una vez construido el mapa, rastrear la posición en el mapa de grupos de vectores cuya naturaleza es conocida de antemano. Finalmente, se puede incorporar conocimiento parcial de ejemplos mediante el símbolo reservado *x*, por ejemplo en,

```
#enésimo vector
1 0 x
```

Dado un fichero de entrada con vectores a partir de los cuales se desea crear un mapa autoorganizado como los que se incluyen en la distribución (ex*.dat), podemos iniciar el proceso con el comando vfind lo que hará que aparezca en pantalla un breve resumen de los datos que nos van a ser requeridos y del algoritmo que vamos a emplear. Éste se compone de dos fases, una primera de ordenación y una segunda de refinamiento.

vfind solicita en primer lugar el número de mapas que se desea crear. De entre todos los mapas creados se selecciona finalmente aquél de menor error de cuantización es decir, aquél en que los vectores que activan una neurona dada se encuentran más próximos al vector de referencia. A continuación y por este orden, se pide el nombre del fichero que contiene los datos de entrada, el del que contiene los vectores de comprobación (test) y el nombre que deseamos dar al fichero de salida con el mapa autoorganizado. Después el tipo de topología (hexagonal o rectangular), el tipo de kernel (bubble o gaussian) y las dimensiones horizontal y vertical del mapa. Finalmente, los datos de cada fase de entrenamiento: el número de ciclos de entrenamiento (training length), la tasa de aprendizaje (training rate) y el radio inicial de vecindad (radius).

Una secuencia de respuestas posible en el caso del fichero de entrada ex.dat proporcionado con la distribución podría ser

```
This program will repeatedly run the initialization, training
and testing cycle for Self-Organizing Map algorithm.

In the following the training file name, the test file name
(that can be the same) and the map save file name are asked.
After them the type of map topology is asked, as well as
the type of neighborhood function. The x- and y-dimension
of the map should be integers and prefereably x-dimension
should be larger than y-dimension.

The training is done in two parts. First an ordering phase
that is usually shorter than the following converging phase.
The number of training cycles, the training rates and
the radius of the adaptation area are asked separately for
```

```
both phases. The fixed point qualifiers and weighting qualifiers
are used if the corresponding parameters were given.

The quantization error is computed for each map and
the best map (smallest quantization error) is saved to
the given file. If the verbose parameter allows the quantization
error is given for each separate trial.

After the answers have been given the training begins
and depending on the size of problem it may take a long time.

Give the number of trials: 100
Give the input data file name: ex.dat
Give the input test file name: ex.dat
Give the output map file name: ex.map
Give the topology type: hexa
Give the neighborhood type: gaussian
Give the x-dimension: 12
Give the y-dimension: 8
Give the training length of first part: 1000
Give the training rate of first part:  0.05
Give the radius in first part: 10
Give the training length of second part: 10000
Give the training rate of second part: 0.02
Give the radius in second part: 3
```

Como resultado de esta secuencia de comandos vfind genera un conjunto de 100 mapas y almacena aquél de menor error de cuantización en el fichero ex.map. El formato empleado es, de nuevo, ASCII lo que permite inspeccionar visualmente el mapa creado. Dado un mapa de dimensión $x \times y$, se listan consecutivamente los x vectores de referencia de la primera línea de neuronas y así sucesivamente hasta completar las y filas del mapa.

Si disponemos de un fichero file.dat con vectores que deseamos presentar al mapa file.map para averigüar qué neurona activa cada uno de ellos, emplearemos el programa visual con la siguiente sintaxis

```
> visual -din file.dat -cin file.map -cout file.out
```

con lo que se generará el fichero file.out con las coordenadas de las neuronas activadas por cada vector del fichero file.dat exactamente en el mismo orden en que figuran en dicho fichero.

Finalmente, dos programas muy útiles para la interpretación de los mapas obtenidos son vcal y umat. El primero es útil si existe una clasificación previa de los casos empleados en la creación del mapa. Evidentemente, no es el caso si lo que queremos es agrupar los datos o, lo que es lo mismo, definir las clases, como resultado del entrenamiento. En ocasiones, sin embargo, disponemos de algún esquema clasificatorio previo y deseamos ver qué clases están próximas en el espacio de características y qué otras están más separadas. En ese caso, podemos incluir la clase a la que pertenece cada ejemplo en una columna adicional de forma que el comando

```
> vcal -din file.dat -cin file.map -cout file.map -numlabs 1
```

calcula cuántos casos de cada clase han activado cada neurona del mapa y le asocia una etiqueta que corresponde a la clase mayoritaria.

El segundo programa, umat, genera a partir del mapa obtenido un gráfico de las distancias entre los vectores de pesos de neuronas contiguas. De esta forma, neuronas que se encuentren en el interior de una agrupación tendrán vectores de pesos similares a los de sus vecinos y las distancias serán pequeñas. Al mismo tiempo, las neuronas que codifiquen las fronteras entre grupos, tendrán vecinas con vectores de pesos muy diferentes y presentarán distancias mayores. Las distancias se codifican en niveles de gris bien según la media de los valores, bien según la mediana dependiendo del comando empleado. La sintaxis en cada caso es

```
> umat -cin file.map [-average 1]
> umat -cin file.map [-median 1]
```

o

```
> umat -cin file.map [-ps 1]
```

si preferimos la gráfica en formato Postscript en lugar de Postscript Encapsulado (que es el formato por defecto).

Capítulo 5

Aplicaciones

Objetivo: En este último capítulo pretendemos cerrar los contenidos del curso de acuerdo con el espíritu que ha guiado el diseño de la asignatura. Hemos pretendido conjugar los aspectos más teóricos del campo de la computación neuronal con una perspectiva eminentemente práctica, de forma que al final del curso el alumno esté capacitado para implementar soluciones conexionistas para el conjunto de tareas en las que este tipo de técnicas son adecuadas. Como mencionamos en el capítulo 1, éstas son básicamente las de clasificación, regresión, control y predicción. Todas ellas son diferentes caras de una misma moneda, o, dicho de otra manera más formal, son tareas que comparten un mismo esquema conceptual abstracto. Pero ésto lo veremos después de los ejemplos.

5.1. Aplicación conexionista en tarea de clasificación.

Empezamos pues con la que probablemente sea la aplicación más popular de las redes neuronales. Existen por ello, multitud de ejemplos de redes neuronales de clasificación en los más diversos ámbitos de la industria y de la ciencia. En este libro de texto hemos elegido un ejemplo del repositorio de la Universidad Carnegie Mellon de EEUU que se puede encontrar en `http://www-2.cs.cmu.edu/afs/cs/project/ai-repository/ai/areas/neural/bench/0.html` aunque existen muchos otros en la red. La razón por la que hemos elegido este ejemplo es que ilustra varios de los aspectos metodológicos fundamentales del desarrollo de una aplicación basada en conexionismo siendo el planteamiento conceptual del problema al mismo tiempo muy sencillo.

5.1.1. Planteamiento del problema

El problema al que hacemos referencia consiste en distinguir minas de rocas a partir de una serie de observaciones obtenidas mediante un instrumento denominado sónar [GS88]. El sónar envía ondas acústicas en una serie de frecuencias distintas y dependiendo del cómo rebota cada frecuencia en un objeto determinado podemos saber de qué está hecho el objeto y a qué distancia se encuentra. Las ondas rebotadas presentan

diferentes características dependiendo del ángulo que forma el haz incidente con el eje longitudinal del objeto (ángulo aspectual). En este ejemplo nos interesa distinguir la composición del material a partir de las señales recibidas por el sónar. En particular, si el objeto está compuesto de roca o de metal. En este último caso supondremos que se trata de una mina enterrada.

Partimos de la energía detectada por el sónar en 60 bandas de frecuencia diferentes y a partir de esos 60 datos pretendemos averiguar si el objeto en el que rebotaron las ondas del sónar es una mina o una roca. Como datos de partida disponemos de 208 ejemplos, 111 casos de minas y 97 de roca.

5.1.2. Preprocesado de los datos

Los datos de entrada han sido preprocesados para que los valores se encuentren en el intervalo [0,1]. Ya hemos tratado el problema de la normalización de los datos tanto de entrada como de salida en el capítulo 2. No vamos a volver aquí sobre el tema, aceptaremos como datos de partida la trasformación de Gorman y Sejnowski, y dejaremos al lector la comparación de los resultados del clasificador para distintos modelos del preprocesado.

Por otra parte, cada par entrada-salida del conjunto de datos disponible corresponde a un ángulo de incidencia entre el haz del sónar y el eje longitudinal del objeto. A este respecto, Gorman y Sejnowski realizan un estudio muy interesante que ilustra la importancia de la selección adecuada del conjunto de entrenamiento. Inicialmente, los autores dividen el conjunto de partida en 13 bloques de 16 casos cada uno, de los cuales 12 son utilizados para entrenar la red y el decimotercero para evaluar el error de clasificación. Repiten el proceso de aprendizaje 13 veces de manera que el bloque empleado para evaluar el funcionamiento de la red es siempre distinto. Los autores obtienen que para algunas particiones del conjunto de datos de partida la capacidad de clasificación de la red es bastante baja e interpretan este resultado como la consecuencia de haber incluido en el bloque de evaluación ángulos aspectuales que no estaban suficientemente representados en el bloque de entrenamiento. Es evidente que si no hemos enseñado a la red a reconocer un tipo de patrón (esto es, si ese patrón no se encontraba en el conjunto de entrenamiento), la red cometerá errores cada vez que se le presente un caso de este tipo. Ésta es en definitiva la lección que se debe aprender de el ejemplo: es fundamental que en el conjunto de entrenamiento se hallen representadas suficientemente todas las clases que queremos reconocer, a ser posible con la frecuencia en que se espera que aparezcan el el funcionamiento real de la red.

Este problema motivó la siguiente solución: los autores dividieron el conjunto de ejemplos de forma que, tanto el conjunto de entrenamiento como el de validación tuvieran una representación suficiente de todos los posibles ángulos aspectuales. Esta vez dividieron por la mitad el conjunto de ejemplos (104 para el entrenamiento y 104 para la evaluación del error) y repitieron 10 veces el entrenamiento y la evaluación de la red con esta única partición de los datos, que es la que se ofrece a los alumnos para que repitan el experimento con JavaNNS.

5.1.3. Arquitectura y aprendizaje

Los autores del artículo emplearon redes neuronales de arquitectura fija para las capas de entrada y salida puesto que la entrada eran siempre las detecciones de las ondas rebotadas en las 60 frecuencias distintas del sónar (60 neuronas de entrada) y sólo hay dos salidas posibles: roca o mina. Gorman y Sejnowski utilizaron para codificar la salida dos neuronas de forma que la activación de la primera representaba la detección de una roca y la de la segunda, una mina. Este tipo de codificación, que podría extenderse a más clases añadiendo una neurona por cada clase, se conoce con el nombre de 1-de-k ($1-of-k$ en inglés) porque las clases se codifican en el conjunto de entrenamiento mediante una única neurona de k en la capa de salida donde k es el número total de clases. En este texto, al número de neuronas de la capa de salida lo hemos llamado N_s así que deberíamos llamar a esta codificación 1-de-N_s.

Una vez fijadas por la estructura de los datos las dimensiones de las capas de entrada y salida queda por determinar la geometría intermedia de la red. Gorman y Sejnowski, a falta de conocimiento previo sobre la arquitectura más apropiada, probaron redes neuronales sin capa oculta y con una capa oculta de 2, 3, 6, 12 y 24 unidades para comparar su capacidad de clasificación mediante el conjunto de evaluación.

Cada arquitectura fue entrenada mediante el algoritmo de retropropagación del error empleando 300 ciclos de aprendizaje en cada caso y prescindiendo de la componente de momento (ver la sección 2.3 en el capítulo 2). Los pesos se inicializan con valores aleatorios entre -0.3 y +0.3 y la tasa o velocidad de aprendizaje inicial es de 2.0. Finalmente, no se consideran errores por debajo de 0.2.

5.1.4. Resumen y propuesta de ejercicio.

En esta sección hemos mostrado un ejemplo sencillo de aplicación conexionista para la resolución de una tarea de clasificación para la que se dispone de escaso o nulo conocimiento previo y sólo se tiene un conjunto de ejemplos de cada clase. Hemos aprendido la importancia de seleccionar adecuadamente el conjunto de ejemplos de entrenamiento de forma que representen estadísticamente todas las posibilidades que se le pueden presentar a la red durante las fases de evaluación y de funcionamiento real. Hemos visto que durante la fase de diseño, la escasez de conocimiento sobre la estructura de la tarea de clasificación implica tener que explorar diferentes arquitecturas para hallar la que mejor clasifica el conjunto de evaluación. A este respecto es conveniente que el lector repase los conceptos relativos al compromiso entre sesgo y varianza presentados en el capítulo 2.

Para consolidar los conceptos aprendidos a lo largo del libro y ejemplificados en este trabajo, así como para acumular destreza en el manejo de las herramientas de desarrollo de redes neuronales pedimos al alumno que reproduzca el experimento de Gorman y Sejnowski utilizando JavaNNS y el fichero de datos de nombre sonar.data que se encuentra en el repositorio cuya dirección URL aparece al comienzo de la sección. Compare las tasas de clasificación correcta obtenidas para cada arquitectura con las obtenidas por los autores del estudio. Asimismo, le pedimos que lleve a cabo las siguientes modificaciones:

1. Repita el experimento empleando retropropagación con momento, por lotes,

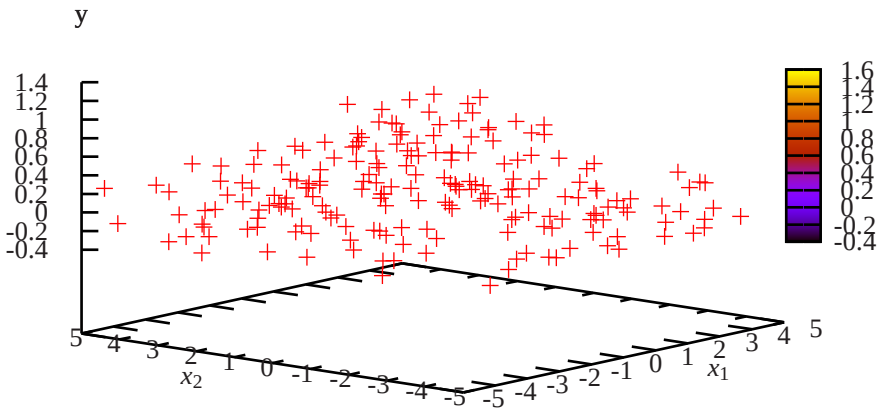


Figura 5.1: Datos de partida a partir de los cuales deseamos obtener una superficie de regresión continua.

quickpropagation, retropropagación con regularización (weight decay) y resilient backpropagation, y compare los resultados.

2. En todos los casos anteriores monitorice los valores de los distintos tipos de error en el conjunto de entrenamiento y en el de evaluación/validación y saque conclusiones sobre las diferencias.

3. Emplee un preprocesado distinto de los datos (por ejemplo, cambiando el rango a [-1,1] o empleando una distribución normal gaussiana) y compare con los resultados de Gorman y Sejnowski.

5.2. Aplicación conexionista en tarea de identificación/regresión.

Continuamos los ejemplos de aplicación de soluciones conexionistas a tareas propias de la IA con un caso de regresión. En muchas ocasiones nos encontramos con sistemas que a un conjunto de entradas responden con un conjunto de salidas y la relación entre unas y otras no es simple o lineal. Supongamos que se dispone de un conjunto de puntos como el de la figura 5.1 que representan la salida y que produce un sistema para una entrada (x_1, x_2) dada. Para fijar ideas, podríamos imaginar que el sistema es un módulo de control de un brazo robótico que mide la presión que ejerce la pinza situada en el extremo del brazo en función de la posición (x_1, x_2) de dicho extremo. Hemos supuesto que el proceso de medida de la salida está afectado por errores, como suele ser habitual en cualquier proceso de medida. La tarea para la que buscamos una solución conexionista es la de obtener la salida programada (la presión de la pinza) para una nueva posición (x_1, x_2) que no sea una de las que ya hemos medido, esto es, queremos generalizar a partir de lo ya conocido que está condensado en la gráfica 5.1. Si lo conseguimos, habremos obtenido un sistema paralelo al inicial que responde

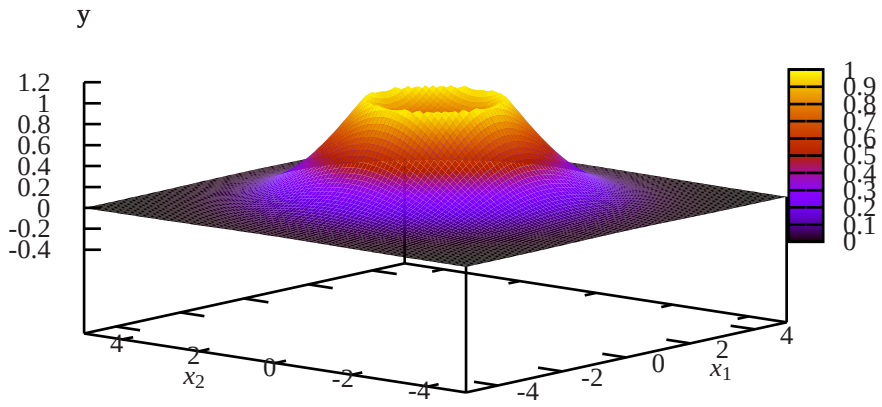


Figura 5.2: Función tridimensional a partir de la cual se han obtenido los puntos de la figura 5.1 por adición de ruido gaussiano ε de media 0 y varianza 0.5. La fórmula análitica empleada fue $\exp(-|x^2+y^2-2|/5)+\varepsilon$.

aproximadamente de la misma manera sin saber su composición ni su estructura interna. A eso podríamos llamarlo una identificación del sistema. En general los problemas son más complicados y tienen un número indeterminado de entradas y salidas.

Si nos olvidamos de que los puntos de la gráfica proceden de un sistema real (un brazo robótico) que lleva a cabo una determinada tarea, podemos abordar el problema como si se tratara de encontrar qué función ha generado los puntos suponiendo que a la función se le ha añadido ruido. Ese problema es muy general y se lo conoce como problema de regresión. En cualquier caso, como decíamos, ambas son equivalentes. En la figura 5.2 se muestra la función de la que se han obtenido los puntos de la gráfica **??** mediante la adición de ruido y en 5.3 la superposición de los puntos a la gráfica desde una perspectiva que permite apreciar la presencia del ruido.

Evidentemente, la función representada en la figura 5.2 es desconocida; se trata precisamente de recuperar esa función continua a partir de los datos de la figura 5.1 que están afectados por ruido. Nosotros la mostramos por motivos didácticos.

5.2.1. Planteamiento del problema

Disponemos del conjunto de ejemplos dado en el fichero regresion.dat accesible en la página web de la asignatura. Consta de 400 casos para los que disponemos de las entradas al sistema (dos) y la salida producida y deseamos construir una red neuronal que capture las relaciones entre las entradas y las salidas y sea capaz de reproducirlas para nuevos casos. Para ello, vamos a dividir el conjunto de datos en diez bloques de 40 ejemplos cada uno. Nueve de estos bloques compondrán el conjunto de entrenamiento y el décimo se empleará para la evaluación de la red. De los nueve primeros

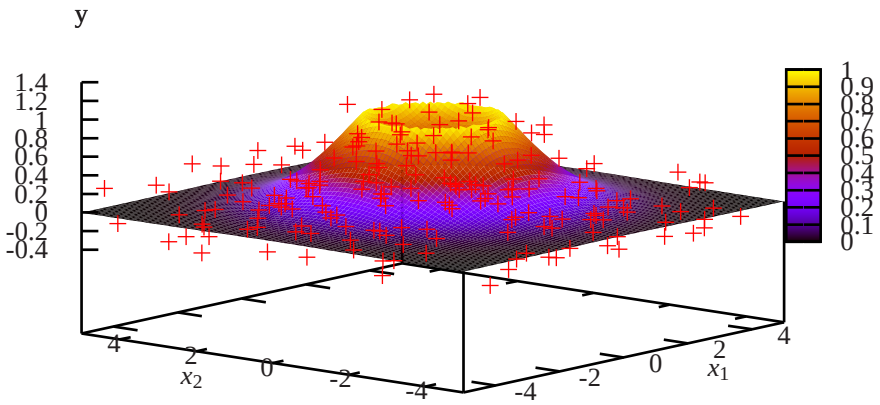


Figura 5.3: Superposición de las dos figuras anteriores.

bloques, seis se emplearán para entrenar la red y tres para la validación. A continuación repetimos el procedimiento nueve veces hasta obtener un total de diez particiones en las que el conjunto de evaluación siempre es diferente. Este proceso es lo que se conoce en inglés como *k-fold cross validation.*

5.2.2. Preprocesado de los datos

Como se ha indicado anteriormente, la definición del perceptrón multicapa hace que la normalización de los datos no sea imprescindible pero sí aconsejable para optimizar el proceso de aprendizaje. En este caso, puesto que se trata de un problema de simetría axial, existe una transformación del espacio de entrada que simplifica extraordinariamente el problema (dejamos al lector que la busque y que justifique por qué). Sin embargo, en situaciones más generales en las que se trabaja en muchas dimensiones, es difícil que podamos emplear este tipo de intuiciones geométricas. Sin más información sobre la naturaleza del problema y dado que el espacio de entrada es extraordinariamente reducido (sólo dos dimensiones) no es necesario efectuar reducción de variables mediante Análisis de Componentes Principales. Además, todos los ejemplos son completos por lo que no es necesario refinar más el preproceso de datos.

5.2.3. Arquitectura y aprendizaje

Dadas las especificaciones del problema, es evidente que la red neuronal deberá tener dos neuronas en la capa de entrada y una en la de salida. El número de capas ocultas y su composición dependerá de lo complicada que sea la función que queremos reproducir. Si se trata de una función altamente no lineal necesitaremos muchas unidades de procesamiento intermedio u ocultas y *viceversa*. Será necesario realizar varias pruebas

para seleccionar una arquitectura adecuada. Si supiésemos a priori qué tipo de función queremos que reproduzca la red neuronal quizás sería posible afinar más la arquitectura de la red. Por ejemplo, si supiésemos que la función depende, no directamente de las variables de entrada x_1 y x_2 sino de sus productos x_1^2, x_2^2 y $x_1 \cdot x_2$, podríamos añadir una capa previa de expansión de entradas que se ocupase únicamente de calcular esos productos y proporcionárselos, junto con x_1 y x_2 a la capa de entrada[a].

5.2.4. Resumen y propuesta de ejercicio

Este ejemplo reproduce un problema común en el análisis estadístico de datos. Dado un conjunto de valores de entrada y otro de salida que suponemos que representan una relación (bien porque son valores de entrada/salida de un sistema o porque simplemente corresponden a una función matemática que no podemos escribir de forma analítica), se busca obtener una red neuronal que reproduzca dicha relación. El lector deberá, a partir de la información proporcionada aquí generar una arquitectura de red neuronal capaz de reproducir los datos de los conjuntos de evaluación con valores del error cuadrático medio inferiores al 10 % en promedio para todas las particiones del método de validación cruzada. Para ello podrá emplear cualquier variante del método de retropropagación del error. Deberá asimismo representar en 3 dimensiones los valores de la función usada para generar los datos

$$f(x,y) = \frac{1}{5} \cdot \exp(-||(x^2+y^2-2||) \tag{5.1}$$

y los que obtendría con la red neuronal para valores de las variables de entrada en el rango mostrado en la figura 5.1 (entre -5 y 5) en intervalos constantes de 0.1. Finalmente, deberá representar gráficamente los errores relativos cometidos para esa misma retícula (es decir, el error absoluto[b] dividido por el valor absoluto de la función).

Finalmente, el lector debe responder a las siguientes preguntas:

1. ¿Cómo se modificaría la arquitectura de la red neuronal si, en lugar de identificar un sistema de una única salida estuviésemos tratando un sistema con múltiples salidas?
2. ¿Qué transformación del espacio de entrada permitiría resolver fácilmente el problema de identificación/regresión propuesto?
3. A la vista de la función generatri de los datos, ¿cree el lector que sería interesante expandir el espacio de entradas? ¿qué relación tendría esta expansión con la solución a la cuestión 2?
4. ¿Por qué cree que se emplea el error cuadrático medio en el aprendizaje de la red neuronal y no el error relativo?

[a] Una capa de expasión del espacio de entradas estaría compuesta de neuronas muy diferentes de los perceptrones del resto de la red. Siendo rigurosos, no sería parte de la red ni podríamos llamar a sus componentes neuronas puesto que no hay pesos adaptativos ni aprendizaje propiamente dicho sino un cálculo analítico muy sencillo en forma de productos. Este tipo de expansiones se deben incluir en el preproceso de datos para no mezclar un tipo de computación con otro.

[b] el valor absoluto de la diferencia entre el valor de la función generatriz y la predicción de la red neuronal

5.3. Aplicación conexionista en tarea de predicción.

El contenido de este texto está dirigido claramente a estudiantes de la asignatura de Sistemas Basados en el Conocimiento II de la Universidad Nacional de Educación a Distancia. Por ello y dado el contexto en el que se imparte la asignatura, se ha optado por restringir la tipología de las redes neuronales tratadas para poder hacer más hincapié en los aspectos metodológicos y prácticos del desarrollo de soluciones conexionistas para tareas en Inteligencia Artificial. Con ello, hemos pasado por alto arquitecturas y algoritmos de aprendizaje que se podrían considerar apropiados para algunas tareas de predicción de series temporales como las que que vamos a abordar en este punto (en particular arquitecturas recurrentes de Jordan o Elman). En el ejemplo que presentaremos a continuación, emplearemos de nuevo perceptrones multicapa entrenados mediante la retropropagación del error para resolver esta tarea aun cuando no sean una solución óptima en muchos casos. Si siempre es importante recordar al alumno el carácter necesariamente limitado del temario de la asignatura y las limitaciones que ello implica, es igualmente necesario ayudarle a encontrar las fuentes de información que le pueden servir para ampliar sus conocimientos. En el libro de Haykin [Hay94] se puede encontrar un capítulo dedicado a este tema y referencias completas a la bibliografía relevante del campo a mediados de los años noventa. Referencias más actualizadas se pueden obtener del equipo docente por los medios habituales.

Vamos pues a utilizar perceptrones multicapa pero de dos maneras ligeramente distintas. En la primera para la que no necesitaremos ninguna introducción teórica, la arquitectura de la red será exactamente igual a la estudiada en el capítulo 2. Supongamos que el problema que queremos solucionar consiste en predecir, a partir de una serie temporal de $p+1$ valores $x(t), x(t-\Delta t), x(t-2\cdot\Delta t), \ldots, x(t-p\cdot\Delta t)$ el valor futuro $x(t+\Delta t)$ de la variable representada. Entonces, construiremos una red neuronal en forma de perceptrón multicapa de $p+1$ entradas y una única salida que representará la predicción de la red. El conjunto de entrenamiento se construye de forma análoga. Supongamos que disponemos de una serie temporal de N ($N >> p+1$) términos a partir de la cual deseamos entrenar la red neuronal. Entonces, podemos generar un conjunto de entrenamiento con los $N-p+1$ ejemplos obtenidos al dividir la serie temporal en bloques de $p+2$ valores consecutivos. Los $p+1$ primeros corresponden a las entradas de la red y el subsiguiente a la salida. Por fijar ideas, supongamos que la serie temporal es tan simple como

$$1,2,3,4,5,6,7,8,9,8,7,6,5,4,3,2,1,2,3,4,5,6,7,8,9,8,7,6,5,4,3,2,1,\ldots \quad (5.2)$$

y que deseamos predecir el siguiente valor de la serie a partir de los últimos 3 valores (más de los necesarios). Entonces, los patrones de entrenamiento de la red neuronal serían

	$\vec{x}[1]$	$\vec{x}[2]$	$\vec{x}[3]$	$\vec{x}[4]$	$\vec{x}[5]$	$\vec{x}[6]$	$\vec{x}[7]$	$\vec{x}[8]$	$\vec{x}[9]$	...
x_1	1	2	3	4	5	6	7	8	9	...
x_2	2	3	4	5	6	7	8	9	8	...
x_3	3	4	5	6	7	8	9	8	7	...
t_1	4	5	6	7	8	9	8	7	6	...

y la arquitectura de la red vendría dada por la figura 5.3. Este tipo de soluciones requiere encontrar el número óptimo de neuronas ocultas y la dimensión del espacio de entradas óptimo para la predicción del valor futuro. Existen formas de estimar este último parámetro que se conoce como dimensión del espacio de estados pero su

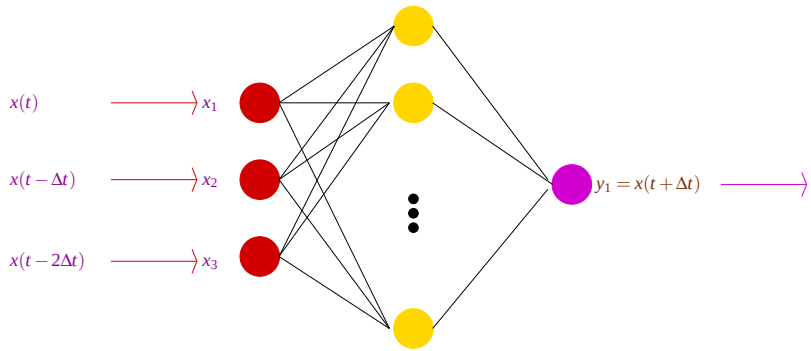


Figura 5.4: Arquitectura unidireccional de alimentación hacia adelante adaptada para el tratamiento de la variable temporal para una dimensión del espacio de entrada igual a 3.

descripción queda fuera del alcance de este libro.

Esta forma de adaptar los perceptrones multicapa para la predicción de series temporales es equivalente a un modelo de autorregresión no lineal de orden p, lo que quiere decir que aproximamos $x(t+\Delta t)$ mediante

$$x(t+\Delta t) = F_{RN}(x(t), x(t-\Delta t), ..., x(t-p\cdot\Delta t). \tag{5.3}$$

Decimos que es un modelo no lineal porque la función F_{RN} que computa la red neuronal es no lineal (recordad la forma de las funciones de activación), y de autorregresión porque la función que nos permite obtener $x(t+\Delta t)$ a partir de los valores anteriores de la serie se obtiene de la propia serie temporal.

La segunda alternativa, en algunos aspectos equivalente a la anterior, consiste en emplear las redes neuronales con retardo temporal (*Time Delay Neural Networks* o TDNN en inglés). Consiste básicamente en dotar de memoria temporal a la red neuronal. Memoria de las entradas y de las activaciones que tuvieron lugar en el pasado reciente. Ello se consigue replicando las capas del perceptrón y haciendo que a cada réplica le lleguen las entradas con diversos retardos. Veámoslo con el ejemplo unidimensional de la serie anterior (figura 5.3) a la que, por fijar ideas, vamos a dotar de una capa oculta de 7 neuronas.

Para dotar de retardo a esa red neuronal de 3-7-1 neuronas vamos a permitir que la activación de la neurona de salida dependa, no sólo de la activación actual de las neuronas de la capa oculta sino de sus cinco últimos valores de activación. Para ello necesitamos replicar la capa oculta que teníamos inicialmente cuatro veces hasta tener cinco capas ocultas de siete neuronas cada una. La neurona de salida estará conectada completamente a cada una de las réplica totalizando 35 sinapsis. Igualmente, deseamos que las activaciones de las neuronas de la capa intermedia dependan de los últimos cinco valores de entrada a la red. En ese caso, y puesto que tenemos cinco réplicas de la capa oculta necesitaremos nueve neuronas en la capa de entrada de forma que la arquitectura de la red neuronal quedaría descrita por la figura 5.3 (como en el caso anterior, cada neurona de la capa de entrada representa un valor de la serie con retardos crecientes). Téngase en cuenta que, en este ejemplo, los valores del número de réplicas son arbitrarios y no están justificados por un análisis previo del problema sino escogidos para dar un ejemplo práctico de TDNN. La conectividad es total entre cada neurona de cada capa oculta y su campo receptivo en la capa de entrada compuesto

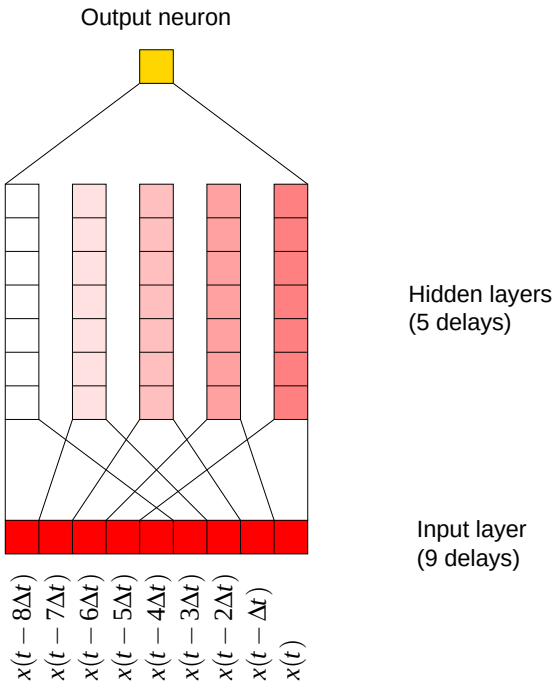


Figura 5.5: Arquitectura resumida de una red TDNN (sólo se muestran los extremos de los campos receptivos, es decir la primera y la última neurona conectadas a la capa oculta, y no todas las sinapsis por claridad).

por 5 neuronas.

5.3.1. Planteamiento del problema

La superficie del Sol (llamada fotosfera) presenta habitualmente regiones oscuras debido a la presencia de altas concentraciones de campo magnético. El número y tamaño de las regiones oscuras denominadas manchas solares varía de forma muy irregular a corto plazo (del orden de meses) pero sigue ciclos de once años en los que se pasa por una fase de mínima actividad (con pocas manchas presentes en la superficie) y otra de máxima. El ejemplo que proponemos al lector es el de predecir el número de manchas solares para los meses posteriores a diciembre del año que comienza el curso académico a partir de los datos registrados desde 1749 que puede encontrar en la página web de la asignatura como fichero de descarga **MONTHLY.PLT** (ver figura 5.3.1.

5.3.2. Preprocesado de los datos

Si se examina el fichero de datos se comprueba que está compuesto de tres columnas que contienen el año, el mes y el número promedio de manchas solares correspondientes. A pesar de ello, el problema abstracto consta únicamente de una entrada y una salida porque la coordenada temporal está desgajada en dos columnas. Una sencilla operación aritmética $x = ao + mes/12$ nos proporciona la entrada que necesitamos. En lo referente a la normalización, volvemos a una situación similar a la de los casos

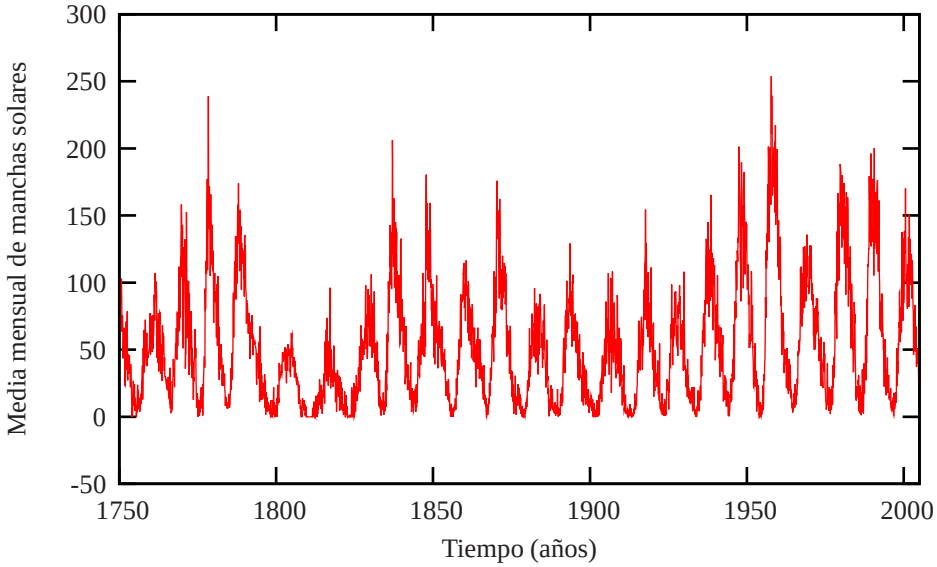


Figura 5.6: Representación gráfica del registro mensual medio de manchas solares.

anteriores: no es imprescindible pero sí puede mejorar ligeramente los resultados de la red. Así pues, podemos transformar la entrada para que se ajuste al intervalo [-0.5:0.5] por ejemplo y el valor promedio mensual de las manchas solares entre 0 y 1.

5.3.3. Arquitectura y aprendizaje

En la presentación del problema ya hemos introducido las dos arquitecturas más sencillas que podemos adoptar para resolver el problema: el perceptrón multicapa clásico o una TDNN (Time Delay Neural Network). El número de unidades de salida en ambos casos es fácilmente deducible: 1. El número de capas ocultas y sus dimensiones deben ser obtenidas por tanteo comparando los errores obtenidos en cada caso en el proceso de validación. De igual manera, lo que hemos denominado dimensión del espacio de entrada, es decir, el número de valores previos necesario para efectuar la predicción se determinará de forma aproximada probando diferentes configuraciones empezando por valores pequeños y aumentando hasta que la adición de nuevas neuronas a la capa de entrada no produzca mejoras significativas en el rendimiento de la red. No es necesario que se pruebe un número muy elevado de valores: para comparar, podemos empezar con dimensiones de entrada de 3, 6 y 12. Obtener la arquitectura de la red de esta forma refleja la falta de una fundamentación teórica de la metodología en el temario de la asignatura. De hecho, algo parecido a esta fundamentación sólo empieza a vislumbrarse en los últimos años en el campo de investigación del conexionismo y por ello hemos preferido dejarla fuera de la materia de estudio.

5.3.4. Resumen y propuesta de ejercicio

En este ejercicio, proponemos el empleo de redes neuronales para predecir el valor de una magnitud (el promedio mensual de manchas solares de los meses próximos) a partir de un registro de los valores de esa magnitud durante los últimos siglos. Para ello emplearemos un perceptrón multicapa simple y una TDNN y entrenaremos ambas redes con los mismos conjuntos de entrenamiento/validación/comprobación (que tendrán que ser creados por el lector). Asimismo, compararemos os resultados de ambas arquitecturas en términos de error absoluto tanto con el conjunto de comprobación como con los datos de medias mensuales que vayan apareciendo durante los meses que componen el segundo cuatrimestre y que se incorporarán a la página web de la asignatura. Recomendamos que se representen gráficamente los errores absolutos y relativos (ver sección 5.2.4) para predicciones cada vez más alejadas del último dato accesible para el entrenamiento (diciembre del año de comienzo del curso) y que se discuta la degradación en la bondad de las predicciones.

5.4. Paradigma unificado para tareas con aprendizaje supervisado

Lo que pretendemos al cerrar este bloque del último capítulo es volver a la formulación teórica mostrada en los capítulos 1 y 2 para ilustrar que, en el fondo, todos los ejemplos de tarea resolubles mediante un perceptrón multicapa caen bajo una clase habitualmente denominada **reconocimiento de patrones** y que las distinciones entre unas y otras se difuminan cuando formalizamos la tarea en términos abstractos. Matemáticamente se trata de un único problema que consiste en optimizar los parámetros de una función de $\mathbb{R}^{N_e}$ en $\mathbb{R}^{N_s}$ (los pesos de las sinapsis) según el valor de error elegido para evaluar el rendimiento de la red neuronal. En el plano abstracto, el primer caso se trataba de reconocer patrones y clasificarlos; en el segundo, de reconocer el patrón de funcionamiento de un sistema y reproducirlo; en el último, aprender el comportamiento de un sistema variable temporalmente, reconocer patrones de cambio presentados a la red y predecir de acuerdo con la experiencia pasada.

5.5. Aplicación conexionista en tarea de *clustering*.

Finalmente, proponemos al lector un último ejercicio para poner en práctica los conocimientos adquiridos en el área del agrupamiento en clases. Para ello, proponemos un caso clásico en Inteligencia Artificial conocido como Iris. El ejemplo está tomado del almacén de ejemplos de la Universidad de Carnegie Mellon y consiste en 150 casos que representan características de tres tipos de plantas iris. Cada caso consta de 4 características de la planta (medidas de la anchura y longitud de los pétalos y sépalos) y de una última columna con la clase a la que pertenece. Hay 50 ejemplos de cada clase y una de ellas es separable linealmente de las otras dos. El ejercicio propuesto consiste en probar diferentes mapas autoorganizados con el software sugerido (SOM_PAK) y comprobar si la agrupación generada se corresponde con las clases incluidas en el fichero de casos que, de nuevo, está disponible en la página web de la asignatura. Recomendamos emplear umat para general un gráfico que represente las particiones del

mapa y vcal para identificar las particiones.

Apéndice A

Teorema de convergencia de perceptrones

La prueba de que un perceptrón es capaz de encontrar un hiperplano de separación $\vec{\omega}^\star$ en un número finito de pasos de entrenamiento para conjuntos linealmente separables se basa en los siguientes conceptos:

1. Para simplificar la demostración vamos a normalizar la solución de forma que $||\vec{\omega}^\star|| = 1{,}0$. Esta normalización no hace que la demostración pierda generalidad puesto que $\vec{\omega}^\star$ y $\frac{\vec{\omega}^\star}{||\vec{\omega}^\star||}$ representan el mismo hiperplano de separación. Además, vamos también a normalizar los vectores de entrenamiento $\vec{x}[n]$ de forma que $||\vec{x}[n]|| = 1$ para todo n. Esto último puede parecer extraño pero tampoco resta generalidad a la demostración ya que partimos de un conjunto de datos separables **linealmente** y hace extraordinariamente más sencillos los pasos que siguen a continuación. En cualquier caso, el lector puede intentar demostrar el teorema sin estos requisitos de normalización.

2. Definamos κ como el valor absoluto del mínimo de los productos $|\vec{\omega}^\star \cdot \vec{x}[n]|$ y δ como un valor arbitrario comprendido entre 0 y κ.

3. La demostración implica la definición de una función que mide lo cerca que está una iteración $\vec{\omega}^\tau$ de la solución $\vec{\omega}^\star$. Emplearemos como función

$$G = \frac{\vec{\omega}^\tau \cdot \vec{\omega}^\star}{||\vec{\omega}^\tau||} = \cos\theta \tag{A.1}$$

   donde *theta* es el ángulo entre la solución $\vec{\omega}^\star$ y la tau-ésima iteración $\vec{\omega}^\tau$. Así pues, G es menor o igual a 1.

4. Tras cada actualización del vector de pesos debida a un vector $\vec{x}$ mal clasificado, la función G se incrementa en una cantidad no despreciable. El numerador pasa a valer

$$\vec{\omega}^{\tau+1} \cdot \vec{\omega}^\star = (\vec{\omega}^\tau + \vec{x}) \cdot \vec{\omega}^\star = \vec{\omega}^\tau \cdot \vec{\omega}^\star + \vec{x} \cdot \vec{\omega}^\star \geq \vec{\omega}^\tau \cdot \vec{\omega}^\star + \delta \tag{A.2}$$

   Si inicializamos el vector a cero ($\vec{\omega}^0 = 0$) tendremos que tras p iteraciones $\vec{\omega}^p \cdot \vec{\omega}^\star \geq p \cdot \delta$

5. Por su parte, el denominador pasará a valer

$$||\vec{\omega}^{\tau+1}|| = ||\vec{\omega}^{\tau} + \vec{x}|| = \sqrt{(\vec{\omega}^{\tau})^2 + (\vec{x})^2 + 2 \cdot \vec{\omega}^{\tau} \cdot \vec{x}} < \sqrt{(\vec{\omega}^{\tau})^2 + 1} \qquad \text{(A.3)}$$

   de forma que, tras p actualizaciones tendremos $||\vec{\omega}^{p}|| < \sqrt{p}$.

6. Resumiendo, tras p iteraciones la función G cumple las siguientes desigualdades

$$1 \geq G = \frac{\vec{\omega}^{p} \cdot \vec{\omega}^{\star}}{||\vec{\omega}^{p}||} > \delta \cdot \sqrt{p} \qquad \text{(A.4)}$$

   de donde tenemos que el número de iteraciones está acotado según $p < \frac{1}{\delta^2}$, o lo que es lo mismo, el algoritmo de aprendizaje alcanza la meta $G = 1$ en un número finito de pasos.

Bibliografía

[BCP04] M.F. Bear, B.W. Connors, and M.A. Paradiso. *Neurociencia: explorando el cerebro*. Masson-Williams & Wilkins España, 2004.

[BD62] Richard E. Bellman and Stuart E. Dreyfus. *Applied Dynamic Programming*. Princeton University Press, Princeton, New Jersey, 1962.

[Bis95] Christopher M. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, Oxford, UK, 1995.

[Fah88] Scott E. Fahlman. Faster-learning variations on back-probagation: An empirical study. In D. Tourtzky, G. E. Hinton, and Terrence J. Sejnowski, editors, *Proceedings of the 1988 Connectionist Models Summer School*, pages 38–51, San Mateo, CA, 1988. Morgan Kauffmann.

[GJ95] Zoubin Ghahramani and Michael I. Jordan. Learning from incomplete data. Technical Report CBCL-108, MIT Artificial Intelligence Laboratory, January 24 1995.

[GS88] R. P. Gorman and T. J. Sejnowski. Analysis of hidden units in a layered network trained to classify sonar targets. *Neural Networks*, 1:75–89, 1988.

[Hay94] Simon Haykin. *Neural Networks*. Macmillan College Publishing Company, New York, 1994.

[HKP91] John Hertz, Anders Krogh, and Richard G. Palmer. *Introduction to the Theory of Neural Computation*. Addison-Wesley, Reading, Massachusetts, 1991.

[Jac88] Robert A. Jacobs. Increased rates of convergence through learning rate adaptation. *Neural Networks*, 1(4):295–307, 1988.

[MP43] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:115–137, 1943.

[PNH86] D. C. Plaut, S. J. Nowlan, and G. E. Hinton. Experiments on learning back propagation. Technical Report CMU–CS–86–126, Carnegie–Mellon University, Pittsburgh, PA, 1986.

[RB93] Martin Riedmiller and Heinrich Braun. A direct adaptive method for faster backpropagation learning: The RPROP algorithm. In *Proc. of the IEEE Intl. Conf. on Neural Networks*, pages 586–591, San Francisco, CA, April 1993.

[RHW86] D.E. Rumelhart, G.E. Hinton, and R.J. Williams. Learning representations by back-propagating errors. Nature, 323:533, 1986.

[Ros62] F. Rosenblatt. *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*. Spartan Books, Washington DC, 1962.

[WH60] B. Widrow and M. E. Hoff. Adaptive switching circuits. In *1960 IRE WESCON Convention Record*, pages 96–104, New York, 1960.