UNA APROXIMACIÓN PRÁCTICA A LAS REDES NEURONALES ARTIFICIALES

• Eduardo Francisco Caicedo y Jesús Alfonso López •

Universidad del Valle
Programa Editorial

Una aproximación práctica a las Redes Neuronales Artificiales

Colección Ingeniería

El propósito general de este libro es ser una guía para que el lector interesado en trabajar con redes neuronales artificiales (RNA), esté en capacidad de solucionar problemas propios de su disciplina usando esta técnica de la inteligencia computacional. La estructura del libro se concibe desde los tipos de aprendizaje, ya que es la característica más importante que poseen las redes neuronales artificiales y en ella radica su principal fortaleza para solucionar y adaptarse a diversos problemas. En este libro se encuentran contenidos teóricos básicos que lo dejarán preparado para afrontar el estudio de libros y artículos de carácter avanzado, acompañado de problemas resueltos que afianzan el saber y el saber hacer.

Universidad del Valle

Programa Editorial

Una aproximación práctica a las Redes Neuronales Artificiales

Eduardo Francisco Caicedo Bravo
Jesús Alfonso López Sotelo

Colección Ingeniería

Caicedo Bravo, Eduardo Francisco
Una aproximación práctica a las redes neuronales artificiales / Eduardo Francisco Caicedo B., Jesús Alfonso López S. -- Santiago de Cali : Programa Editorial Universidad del Valle, 2009.
218 p. : il. ; 24 cm. -- (Colección Ciencias Naturales y Exactas)
Incluye bibliografía e índice.
1. Inteligencia artificial 2. Redes neurales (Informática) I. López S., Jesús Alfonso II. Tít. III. Serie.
006.3 cd 21 ed.
A1237772

CEP-Banco de la República-Biblioteca Luis Ángel Arango

Universidad del Valle
Programa Editorial

Título: *Una aproximación práctica a las redes neuronales artificiales*
Autores: Eduardo Francisco Caicedo Bravo y Jesús Alfonso López Sotelo
ISBN: 978-958-670-767-1
ISBN PDF: 978-958-765-510-0
DOI: 10.25100/peu.64
Colección: Ingeniería
Primera Edición Impresa diciembre 2009
Edición Digital junio 2017

Rector de la Universidad del Valle: Édgar Varela Barrios
Vicerrector de Investigaciones: Javier Medina Vásquez
Director del Programa Editorial: Francisco Ramírez Potes

© Universidad del Valle
© Eduardo Francisco Caicedo Bravo y Jesús Alfonso López Sotelo

Diseño de carátula: Anna Echavarria. Elefante

Este libro, o parte de él, no puede ser reproducido por ningún medio sin autorización escrita de la Universidad del Valle.

El contenido de esta obra corresponde al derecho de expresión del autor y no compromete el pensamiento institucional de la Universidad del Valle, ni genera responsabilidad frente a terceros. El autor es el responsable del respeto a los derechos de autor y del material contenido en la publicación (fotografías, ilustraciones, tablas, etc.), razón por la cual la Universidad no puede asumir ninguna responsabilidad en caso de omisiones o errores.

Cali, Colombia, junio de 2017

ÍNDICE GENERAL

Introducción 9

Capítulo 1
Generalidades sobre redes neuronales artificiales 13
Introducción 13
Breve reseña histórica 14
De la neurona biológica a la neurona artificial.....17
La neurona biológica 20
La neurona artificial 21
Procesamiento matemático en la neurona artificial.....22
Red neuronal artificial 23
Arquitecturas de redes neuronales artificiales.....25
Redes monocapa 25
Redes multicapa 26
Redes *feedforward*.....27
Redes recurrentes 27
El aprendizaje en las redes neuronales artificiales.....28
Aprendizaje supervisado 29
Aprendizaje no-supervisado 31
Ejemplo de procesamiento de la información en una red neuronal 31
Nivel de aplicación.....32

Capítulo 2
Redes neuronales perceptron y adaline.....37
Introducción 37
Red neuronal perceptron 38

Arquitectura de un perceptron 38
Algoritmo de aprendizaje 40
Red neuronal adaline 43
Arquitectura 43
Algoritmo de aprendizaje 45
Limitaciones del perceptron 48
Aproximación práctica 50
Construcción de un perceptron usando MATLAB® 50
Solución de la función lógica AND con un perceptron 51
Exportando la red neuronal a simulink 56
Solución de la función lógica AND con UV-SRNA 56
Clasificador lineal con UV-SRNA 57
Reconocimiento de caracteres usando el Perceptron 58
Reconocimiento de caracteres con UV-SRNA 64
Filtro adaptativo usando una red adaline 66
Filtrado de señales biomédicas 69
Filtrado de señales de voz 71
Proyectos propuestos 72

Capítulo 3
Percetron multicapa y algoritmo backpropagation 75
Introducción 75
Arquitectura general de un perceptron multicapa 76
Entrenamiento de un MLP 77
Nomenclatura del algoritmo backpropagation 78
Algoritmo backpropagation: regla delta generalizada 79
Pasos del algoritmo backpropagation 85
Algoritmo gradiente descendente con alfa variable 85
Pasos del algoritmo gradiente descendente con alfa variable 86
Algoritmos de alto desempeño para redes neuronales MLP 87
Algoritmo de aprendizaje del gradiente conjugado 88
Algoritmo de aprendizaje levenberg marquardt 93
Consideraciones de diseño 99
Conjuntos de aprendizaje y de validación 99
Dimensión de la red neuronal 100
Velocidad de convergencia del algoritmo 101
Funciones de activación 102
Pre y pos-procesamiento de datos 102
Regularización 103
Aproximación Práctica 107
Solución del problema de la función XOR con MATLAB® 107
Aprendizaje de una función seno con MATLAB® 110
Aprendizaje de la función silla de montar con MATLAB® 112

Solución del problema de la XOR con UV-SRNA.....116
Identificación de sistemas usando redes neuronales MLP 119
Pronóstico de consumo de energía (demanda).....126
Aplicación a la clasificación de patrones (el problema de IRIS) 131
Proyectos propuestos.....135

Capítulo 4
Red neuronal de hopfield.....137
Introducción 137
Memoria autoasociativa bidireccional (BAM) 138
Arquitectura de la BAM.....138
Memoria autoasociativa 139
Procesamiento de información en la BAM 140
Modelo discreto de hopfield.....141
Procesamiento de aprendizaje 141
Principio de funcionamiento 142
Concepto de energía en el modelo discreto de hopfield.....143
Ejemplo de procesamiento 143
Modelo continuo de hopfield 147
Modelo continuo de hopfield de una neurona 148
Función de energía para el modelo continuo de hopfield.....151
Aproximación práctica 152
Red tipo hopfield con MATLAB® 152
Proyectos propuestos.....154

Capítulo 5
Mapas auto-organizados de kohonen 157
Introducción 157
El modelo bioinspirado de kohonen.....159
Arquitectura de la red.....160
Algoritmo de aprendizaje 163
Consideraciones iniciales 163
Modelo matemático.....163
Ejemplo 165
Principio de funcionamiento 171
Aproximación práctica 172
Capacidad para reconocer grupos de patrones de un mapa de kohonen 172
Capacidad de autoorganización de los mapas de kohonen usando MATLAB® 174
Capacidad de autoorganización de los mapas de kohonen usando UV-SRNA.....178
clasificación de patrones usando mapas de kohonen 182

Proyectos propuestos.....185

Capítulo 6
Red neuronal de base radial (RBF) 187
Introducción 187
El problema de interpolación 187
Redes de base radial 191
Arquitectura de una red de base radial 192
Entrenamiento de la red RBF 194
Diferencias entre las redes MLP y RBF 196
Aproximación práctica 201
Ejemplo de interpolación exacta con MATLAB® 201
Aprendizaje de la función XOR 203
Aprendizaje de una función de una variable 203
Identificación de la dinámica de un sistema con una red RBF.....206
Proyectos propuestos 211

Bibliografía 213

INTRODUCCIÓN

El propósito general de este libro es ser una guía para que el lector interesado en trabajar con Redes Neuronales Artificiales (RNA), esté en capacidad de solucionar problemas propios de su disciplina usando esta técnica de la Inteligencia Computacional. Es imposible que un único libro cubra todos los posibles tipos de redes neuronales que existen en la literatura, por esta razón, llevaremos a cabo una revisión de las principales arquitecturas de red y de sus características generales con el fin de que el lector quede en capacidad de comprender y utilizar no solo las redes que vamos a presentar, sino aquellas que por razones propias del problema a solucionar, exijan la utilización de un modelo no cubierto en este libro. En el presente libro presentamos un modelo de neurona artificial y tipos de conectividad que se mantienen para todas las arquitecturas de redes estudiadas y, por este motivo, consideramos que pueden extenderse hacia otras arquitecturas no vistas en el presente texto.

Hemos concebido la estructura de este libro desde el punto de vista de los tipos de aprendizaje, pues consideramos que ésta es la característica más importante que poseen las redes neuronales artificiales, y en ella radica su principal fortaleza para solucionar y adaptarse a diversos problemas. En este libro se encuentran contenidos teóricos básicos que lo dejarán preparado para afrontar el estudio de libros y artículos de carácter avanzado, acompañado de problemas resueltos que afianzan el saber y el saber hacer.

En el primer capítulo, iniciamos con una breve revisión histórica de la evolución de las RNA con el fin de mostrar los principales desarrollos científicos que han enriquecido este apasionante campo del saber. Presentamos el modelo artificial de una neurona inspirado en el funcionamiento de la neurona biológica y, a partir de este modelo artificial, enfatizamos en las

características que potencian su aplicabilidad. Hacemos una revisión de las arquitecturas monocapa, multicapa y recurrente de las redes neuronales artificiales, así como de los procesos de aprendizaje supervisado y no–supervisado. Finalizamos con un ejemplo que ilustra el procesamiento de los datos llevado a cabo en la red.

En el capítulo dos, estudiamos las redes neuronales tipo *Perceptron y Adaline,* las arquitecturas, los principales algoritmos de aprendizaje y su aplicabilidad. Es importante detenerse en las limitaciones inherentes al *Perceptron* con el fin de visualizar la introducción de estructuras de red más complejas y que las superen. Como en todos los capítulos siguientes, el libro propone una aproximación práctica para solucionar problemas usando MATLAB® y UV-SRNA, siendo esta última una herramienta desarrollada en la Universidad del Valle. Entre los problemas prácticos solucionados tenemos el de filtrado de ruido usando *Adaline*, clasificación y el reconocimiento de caracteres con un *Perceptron*. Al finalizar el capítulo se proponen algunos proyectos para ser resueltos por el lector.

De las limitaciones observadas en el *Perceptron*, relacionadas fundamentalmente con su imposibilidad para solucionar problemas no lineales, surgen el Perceptron Multicapa (MLP) y el algoritmo de Backpropagation, temas que son ampliamente discutidos en el capítulo tres. Partimos de la arquitectura general de un MLP y del algoritmo básico de aprendizaje conocido como Regla Delta Generalizada, para luego presentar algunos algoritmos que buscan mejorar el proceso de aprendizaje como el del Gradiente Conjugado y el de Levenberg Marquardt. Fruto de la experiencia en diversas aplicaciones se entrega al diseñador de soluciones con redes neuronales artificiales, un amplio apartado de recomendaciones de tipo práctico para dimensionar la red, seleccionar los conjuntos de datos, las funciones de activación, pre y posprocesar los datos y regularizar el aprendizaje en la red. El MLP es ampliamente utilizado en diversas aplicaciones por lo que este capítulo presenta ejemplos de identificación de sistemas dinámicos, pronóstico de demanda de energía eléctrica, reconocimiento y clasificación de patrones.

El Modelo de Hopfield es un buen ejemplo de red neuronal dinámica, cuyo estudio se plantea en el capítulo cuatro, a partir de las memorias asociativas y autoasociativas para luego, adentrarnos en el Modelo Discreto de Hopfield, a través de su procedimiento de aprendizaje y principio de funcionamiento. Continuaremos con el Modelo Continuo de Hopfield para finalizar con la aproximación práctica donde veremos paso a paso como se construye y simula en MATLAB® este tipo de red.

En el capítulo cinco estudiaremos los Mapas Auto-organizados de Kohonen como un ejemplo representativo del aprendizaje no supervisado de las redes neuronales artificiales; partimos del modelo bio-inspirado de Kohonen, para estudiar la arquitectura de la red, su algoritmo de aprendiza-

je y el principio de funcionamiento. En las aplicaciones prácticas se enfatiza en la capacidad que tiene esta red para auto-organizarse dependiendo de la estructura de los datos que usemos para su entrenamiento, con el fin de mostrar su aplicabilidad en el campo del reconocimiento y clasificación de patrones.

En el último capítulo estudiaremos las Redes de Base Radial (RBF) que pertenecen al grupo de redes neuronales cuyo aprendizaje se considera híbrido, pues en ellas se manifiestan los entrenamientos supervisado y no-supervisado. Para continuar con la uniformidad del libro, igualmente veremos su arquitectura, los algoritmos de aprendizaje y su aplicabilidad. Llevaremos a cabo un último trabajo práctico donde las RBF serán aplicadas en problemas de clasificación y reconocimiento de patrones y en la aproximación de funciones.

Para finalizar, tenemos el gusto de presentar a la comunidad este libro que corresponde a más de una década de trabajo académico y de investigación de los autores en el Grupo Percepción y Sistemas Inteligentes de la Universidad del Valle, e invitamos al lector a adentrarse en el maravilloso mundo de las Redes Neuronales Artificiales que hace parte de la línea de la Inteligencia Computacional, sabiendo que encontrará una poderosa herramienta de procesamiento de datos y señales para solucionar una amplia gama de problemas complejos.

Los Autores

PÁGINA EN BLANCO
EN LA EDICIÓN IMPRESA

Capítulo 1

GENERALIDADES SOBRE REDES NEURONALES ARTIFICIALES

Introducción

Muchos de los desarrollos del hombre se deben a su capacidad para explicar y emular funciones que son realizadas por seres vivos, por ejemplo, se puede citar el radar, el cual surge como una emulación de la forma como un murciélago es capaz de detectar los objetos que están en su camino, sin necesidad de verlos, gracias a la emisión de una onda ultrasónica, su posterior recepción de la señal de eco y procesamiento, con el fin de detectar obstáculos en su vuelo con una rapidez y precisión sorprendentes. Como el mencionado, existen muchos ejemplos más en la naturaleza que han ins-pirado diversos inventos: el helicóptero, el avión, el submarino, para citar algunos.

Aunque el hombre ha sido capaz de emular funciones de los animales, para él siempre ha sido un sueño poder conocer e imitar, la llamada por muchos la máquina perfecta: el cerebro humano.

Las redes neuronales artificiales (RNA) surgen como un intento para emular el funcionamiento de las neuronas de nuestro cerebro. En este sentido las RNA siguen una tendencia diferente a los enfoques clásicos de la inteligencia artificial que tratan de modelar la inteligencia humana buscando imitar los procesos de razonamiento que ocurren en nuestro cerebro.

En este primer acercamiento, pretendemos introducir los conceptos iniciales y básicos asociados a las RNA. Empezamos con un recorrido por el proceso evolutivo que los científicos han llevado a cabo en la construcción de este apasionante campo del saber. Luego, mostramos como pasar del modelo de la neurona biológica, al modelo de la neurona artificial, que se seguirá utilizando a lo largo de este libro. Establecemos las características más relevantes de las RNA cuando emulan el proceso de aprendizaje que

ocurre en nuestro cerebro. Finalmente, presentamos algunos ejemplos prácticos, donde las RNA han sido aplicadas con éxito.

Breve reseña histórica

Emular redes neuronales de manera artificial no es un desarrollo reciente, se hicieron algunos intentos antes del advenimiento de los computadores, pero su verdadero desarrollo tuvo lugar cuando las simulaciones por computador fueron factibles por su capacidad de procesamiento y bajo costo. Luego de un periodo inicial de entusiasmo, las redes neuronales cayeron en un periodo de frustración y desprestigio, durante esta etapa el soporte económico y computacional era muy limitado y sólo unos pocos investigadores consiguieron logros de algún nivel de importancia. Estos pioneros fueron capaces de desarrollar una tecnología que sobrepasara las limitaciones identificadas en algunas publicaciones de Minsky y Papert en 1969, que sembraron un desencanto y frustración general en la comunidad científica.

Actualmente, las redes neuronales ocupan un sitio preponderante en el ámbito del procesamiento de señales con técnicas adaptativas. Acerquémonos a algunas etapas en las que puede resumirse la historia de las redes neuronales artificiales:

1. ***El concepto de Neurona***: A finales del siglo XIX, el científico español Santiago Ramón y Cajal logra describir por primera vez los diferentes tipos de neuronas en forma aislada. Al mismo tiempo plantea que el sistema nervioso estaba constituido por neuronas individuales, las que se comunicaban entre sí a través de contactos funcionales llamados sinapsis (teoría de la neurona). La hipótesis de este investigador se oponía a la de otros científicos de su época que concebía al sistema nervioso como una amplia de red de fibras nerviosas conectadas entre sí formando un continuo, en clara analogía con el sistema circulatorio.

Santiago Ramón y Cajal
(1852-1934). Médico español que obtuvo el Premio Nobel en 1906 por descubrir los mecanismos que gobiernan la morfología y los procesos conectivos de las células nerviosas, una nueva y revolucionaría teoría que se empezó a ser llamada la «doctrina de la neurona» .

Cursó la carrera de Medicina en Zaragoza, a donde toda su familia se trasladó en 1870, allí se centró en sus estudios universitarios con éxito. El año 1875 marcó el inicio del doctorado y de su vocación científica. Ganó la cátedra de Anatomía Descriptiva de la Facultad de Medicina de Valencia en 1883, donde pudo estudiar la epidemia de cólera que azotó la ciudad el año 1885. En 1887 se trasladó a Barcelona para ocupar la cátedra de Histología creada en la Facultad de Medicina de la Universidad de Barcelona. En 1888, definido por Cajal como "mi año cumbre", descubre los mecanismos que gobiernan la morfología y los procesos conectivos de las células nerviosas de la materia gris del sistema nervioso cerebroespinal.

Su teoría fue aceptada en 1889 en el Congreso de la Sociedad Anatómica Alemana, celebrado en Berlín. Su esquema estructural del sistema nervioso como un aglomerado de unidades independientes y definidas, pasó a conocerse como «doctrina de la neurona» y en ella destaca la ley de la polarización dinámica, modelo capaz de explicar la transmisión unidireccional del impulso nervioso. Su trabajo y su aportación a la neurociencia se verían reconocidos, finalmente, en 1906, con la concesión del Premio Nobel de Fisiología y Medicina.

2. ***Primeros intentos:*** McCulloch y Pitts (1943) desarrollaron algunos modelos de redes neuronales basados en su conocimiento de neurología, estos modelos se basaban en neuronas simples, consideradas como dispositivos binarios con umbrales fijos. Los resultados de sus modelos fueron funciones lógicas elementales tales como "*a* ***or*** *b*" y "*a* ***and*** *b*". En 1949, Donald Hebb en su libro *The Organization of Behavior*, presenta el principio del aprendizaje no supervisado, conocido como la Regla de Hebb.
3. ***Tecnología emergente y promisoria***: No solo la neurociencia influía en el desarrollo de las redes neuronales, también los físicos y los ingenieros contribuían al progreso de las simulaciones de redes neuronales. Rosenblatt (1958) revitalizó fuertemente el interés y la actividad en esta área cuando diseñó y desarrolló el modelo de red neuronal que denominó Perceptron. El Perceptron tiene dos niveles, una de entrada cuya función es recibir la información del exterior y un nivel de

salida o de procesamiento que es la que se encarga de hacer el procesamiento de los datos entregados por el nivel de entrada para generar así la salida de la red neuronal. Este sistema es capaz de asociar unas entradas dadas a una salida determinada. Otro sistema fue el ADALINE (ADAptive LInear Element) el cual fue desarrollado en 1960 por Widrow y Hoff (de la Universidad de Stanford). El ADALINE fue un dispositivo electrónico analógico hecho de componentes simples, con un método de aprendizaje diferente al del Perceptron, empleando la regla de aprendizaje basada en mínimos cuadrados (LMS –*Least Mean Square*).

4. ***Periodo de frustración y desprestigio***: En 1969 Minsky y Papert presentaron un trabajo, en cual se recalcaron las limitaciones del Perceptron para solucionar problemas complejos, esencialmente aquellos que no son linealmente separables. Estos autores, además generalizaron las limitaciones de un Perceptron monocapa a las redes neuronales multicapa, cuando plantearon: *"...nuestro intuitivo juicio es que la extensión (a sistemas multicapa) es una tarea estéril"*. El resultado de las afirmaciones de este trabajo fue disminuir el interés de los investigadores en la simulación de redes neuronales; lo que generó un desencanto de los investigadores en el área y trajo como resultado un periodo en el cual prácticamente se dejó de hacer nuevas propuestas.
5. ***Innovación***: Aunque el interés por las redes neuronales era mínimo, varios investigadores continuaron trabajando en el desarrollo de métodos computacionales basados en neuromorfología para problemas de identificación y clasificación de patrones. Durante este periodo se generaron varios paradigmas, entre los cuales podemos mencionar a: Steve Grossberg y Gail Carpenter quienes desarrollaron la teoría de la resonancia adaptativa, ART (Adaptive Resonance Theory) (1976), Anderson y Kohonen (1982) quienes desarrollaron técnicas para aprendizaje asociativo, Hopfield (1984) quien desarrolló una red neuronal haciendo un símil energético. Paul Werbos (1982) desarrolló y usó el método de aprendizaje conocido como Backpropagation, destacando que varios años después de popularizarse este método, es actualmente el más utilizado en las arquitecturas multicapa con mayor nivel de aplicación práctica. En esencia una red Back-Propagation es un Perceptron con múltiples capas, con diferentes funciones de activación en las neuronas artificiales y con una regla de aprendizaje más robusta y confiable.
6. ***Resurgimiento***: Durante el final de la década del setenta y principios de los ochenta, fue importante el resurgimiento del interés en el campo de las redes neuronales. Varios factores han influenciado este movimiento, tales como la aparición de libros y conferencias que han dado a conocer las bondades de esta técnica a personas de diferen-

tes áreas. Se introdujeron cursos en los programas académicos de las principales universidades europeas y americanas. El financiamiento a proyectos de investigación en redes neuronales en Europa, Estados Unidos y Japón que han dado origen a una gran variedad de aplicaciones comerciales e industriales.

7. ***Lo Actual***: Se han realizado progresos muy significativos en el campo de las RNA, lo suficientes como para atraer una gran atención e interés en generar nuevos campos de aplicación. Ya se encuentran comercialmente circuitos integrados basados en RNAs y las aplicaciones desarrolladas resuelven problemas cada vez más complejos. En la actualidad ha surgido un nuevo tipo de máquinas de aprendizaje, como ejemplo las máquinas con vectores de soporte, con grandes capacidades para hacer procesamiento de datos, que aunque no son bioinspiradas como las redes neuronales, pues su aprendizaje se basa en métodos estadísticos, han impactado fuertemente en diversos campos del conocimiento.

De la neurona biológica a la neurona artificial

Una neurona es una célula viva y está constituida por los mismos elementos que conforman las células biológicas. En general una neurona consta de un cuerpo celular más o menos esférico de 5 a 10 micras de diámetro, del que sale una rama principal el axón, y varias ramas más cortas denominadas dendritas. A su vez el axón puede producir ramas en torno a su punto de arranque, y con frecuencia se ramifica extensamente cerca de su extremo. Figura 1.1.

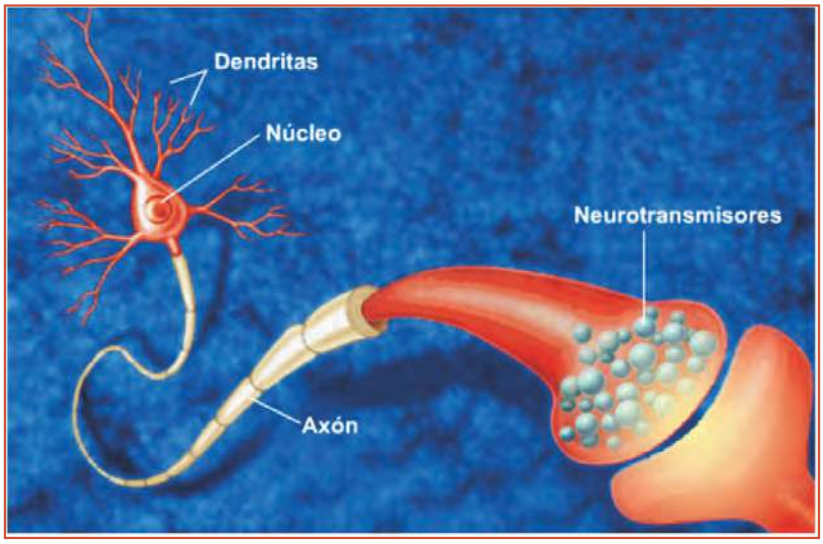


Fig. 1.1 La neurona biológica

Una de las características que diferencian a las neuronas del resto de células vivas, es su capacidad de comunicación. En términos generales, las dendritas y el cuerpo celular reciben señales de entrada, el cuerpo celular las combina e integra y emite señales de salida. El axón transporta esas señales a sus terminales, los cuales se encargan de distribuir la información a un nuevo conjunto de neuronas. Por lo general una neurona recibe información de miles de otras neuronas y, a su vez, envía información a miles de neuronas más. Se estima que en el cerebro humano existen del orden de 10^{15} conexiones. El procesamiento de información de esta máquina maravillosa es en esencia paralelo, en la Tabla 1.1 mostramos un análisis comparativo entre un computador secuencial tipo Von Neumann y un sistema biológico neuronal.

Tabla 1.1. Paralelo entre un computador secuencial y el sistema biológico neuronal

Característica	**Computador secuencial**	**Sistema biológico neuronal**
Unidad de Procesamiento	Compleja Alta velocidad Una sola unidad	Simple Baja velocidad Muchas unidades

Memoria	Separada del procesador Localizada Direccionable aleatoriamente	Integrada dentro del procesador Distribuida Direccionable por contenido
Procesamiento de los Datos	Centralizado Secuencial Instrucciones almacenadas en Programas	Distribuido Paralelo Capacidad de aprendizaje
Confiabilidad	Muy vulnerable ante fallos	Robusto ante fallos
Ambiente de operación	Bien definido	Puede ser ambiguo

John von Neumann zu Margitta
(1903-1957) Matemático húngaro que realizó importantes contribuciones a la Física, Matemáticas, Economía y Ciencias de la Computación. A los 23 años recibió su doctorado en matemáticas de la Universidad de Budapest.

Tras publicar junto a Oskar Morgenstern el libro *Theory of games and economic behavior* ('Teoría de juegos y comportamiento económico'), es considerado como el padre de la Teoría de los Juegos. Además es reconocido su trabajo en el Proyecto Manhattan que dio origen a la bomba atómica.

Es considerado como el pionero del computador digital moderno, y de hecho, una de las arquitecturas de computadores más utilizadas lleva su nombre. Junto con Eckert y Mauchly, en la Universidad de Pennsylvania, desarrollaron el concepto de programa almacenado en memoria, que permitió la ejecución de las instrucciones sin tener que volverlas a escribir. El primer computador en usar este concepto fue el llamado EDVAC (Electronic Discrete-Variable Automatic Computer), desarrollado por Von Neumann, Eckert y Mauchly. Los programas almacenados dieron a los computadores flexibilidad y confiabilidad, haciéndolos más rápidos y menos sujetos a errores que los programas mecánicos.

La neurona biológica

las señales que se encuentran en una neurona biológica son de naturaleza eléctrica y química. La señal generada por la neurona y transportada a lo largo del axón es eléctrica, mientras la señal que se transmite entre los terminales del axón de una neurona y las dendritas de las neuronas siguientes es de origen químico. Concretamente se realiza mediante neurotransmisores que fluyen a través de una región especial, llamada sinapsis que está localizada entre los terminales del axón y las dendritas de las neuronas siguientes, tal como vemos en la Figura 1.2.

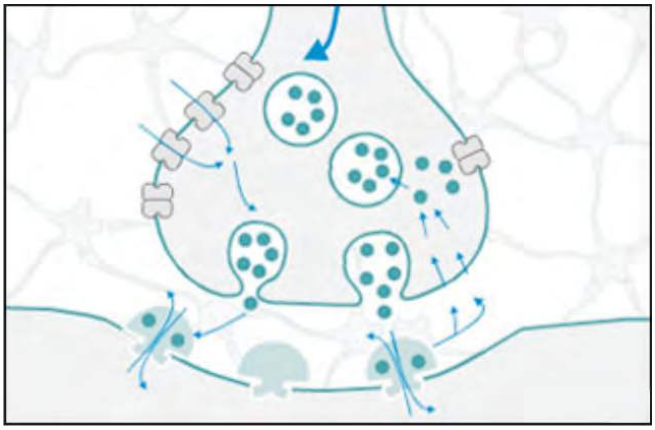


Fig. 1.2 Neurotransmisores en una Sinapsis

Para establecer una similitud directa entre la actividad de una neurona biológica y una artificial, analicemos los siguientes aspectos funcionales:

1. ***Los elementos de proceso (neuronas) reciben las señales de entrada***
   Una de las características de las neuronas biológicas, y a la que deben su gran capacidad de procesamiento y realización de tareas de alta complejidad, es que están altamente conectadas con otras neuronas de las cuales reciben un estímulo de algún evento que está ocurriendo o cientos de señales eléctricas con la información aprendida. Esta información al llegar al cuerpo de la neurona, afecta su comportamiento y puede afectar una neurona vecina o algún músculo.
2. ***Las señales pueden ser modificadas por los pesos sinápticos***
   La comunicación entre una neurona y otra no es por contacto directo. La comunicación entre neuronas se hace a través de lo que se ha denominado sinapsis. Las sinapsis es una espacio que está ocupado por unas sustancias químicas denominadas neurotransmisores. Estos neurotransmisores son los que se encargan de bloquear o dejar pasar las señales que provienen de las otras neuronas.

3. Los elementos de proceso suman las entradas afectadas por las sinapsis
Las neuronas van recibiendo las señales eléctricas provenientes de las otras neuronas con las que tienen contacto. Estas señales se acumulan en el cuerpo de la neurona para definir qué hacer.

4. Bajo una circunstancia apropiada la neurona transmite una señal de salida
Si el total de la señal eléctrica que recibe la neurona es suficientemente grande, se puede vencer el potencial de acción, lo cual permite que la neurona se active o por el contrario permanezca inactiva.

5. La salida del elemento de proceso puede ir a muchas neuronas
Al activarse una neurona, está en capacidad de transmitir un impulso eléctrico a las neuronas con las cuales tiene contacto. Este nuevo impulso, por ejemplo, actúa como entrada para otras neuronas o como estímulo en algún músculo.

La neurona artificial

A partir de los aspectos funcionales de la neurona biológica, vamos a proponer un modelo de neurona artificial como el ilustrado en la figura 1.3.

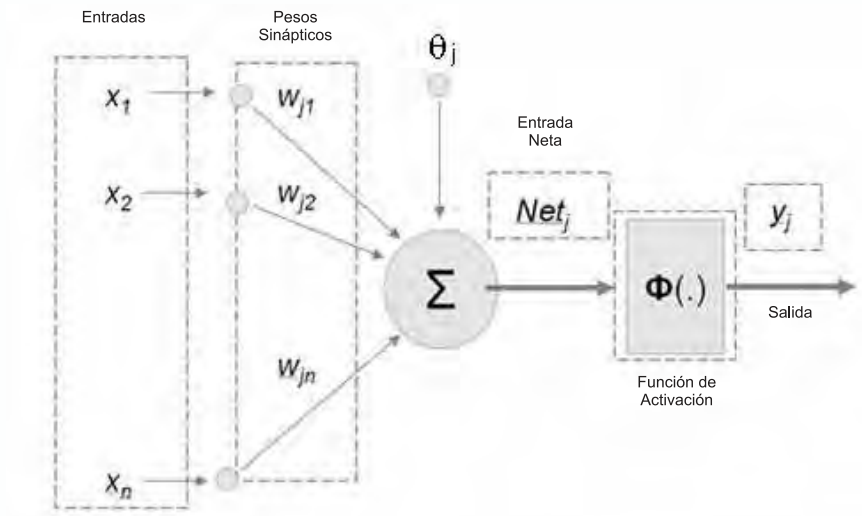


Fig. 1.3 Modelo de Neurona Artificial

Como en el caso de la neurona biológica, la neurona artificial recibe unas entradas de estímulo que pueden provenir del sistema sensorial externo o de otras neuronas con las cuales posee conexión. Para el caso del modelo que proponemos en la ilustración, la información que recibe la neurona la definimos con el vector de entradas $X= [x_1, x_2, \ldots x_n]$.

La información recibida por la neurona es modificada por un vector *w*

de pesos sinápticos cuyo papel es el de emular la sinapsis existente entre las neuronas biológicas. Estos valores se pueden asimilar a ganancias que pueden atenuar o amplificar los valores que se desean propagar hacia la neurona. El parámetro θ_j se conoce como el *bias* o umbral de una neurona, cuya importancia veremos más adelante.

Los diferentes valores que recibe la neurona, modificados por los pesos sinápticos, los sumamos para producir lo que hemos denominado la entrada neta. Esta entrada neta es la que va a determinar si la neurona se activa o no.

La activación o no de la neurona depende de lo que llamaremos Función de Activación. La entrada neta la evaluamos en esta función y obtenemos la salida de la red. Si, por ejemplo, esta función la definimos como un escalón unitario, la salida será 1 si la entrada neta es mayor que cero, en caso contrario, la salida será 0.

Aunque no hay un comportamiento biológico que indique la presencia de algo parecido en las neuronas del cerebro, el uso de la función de activación es un artificio para poder aplicar las RNA a una gran diversidad de problemas reales. De acuerdo a lo mencionado la salida y_j de la neurona se genera al evaluar la neta en la función de activación.

Podemos propagar la salida de la neurona hacia otras neuronas o puede ser la salida de la red, que de acuerdo a la aplicación tendrá una interpretación para el usuario.

Procesamiento matemático en la neurona artificial

En una neurona artificial el cálculo de la entrada neta se puede representar con la ecuación 1.1, y en forma vectorial se representa con la ecuación 1.2

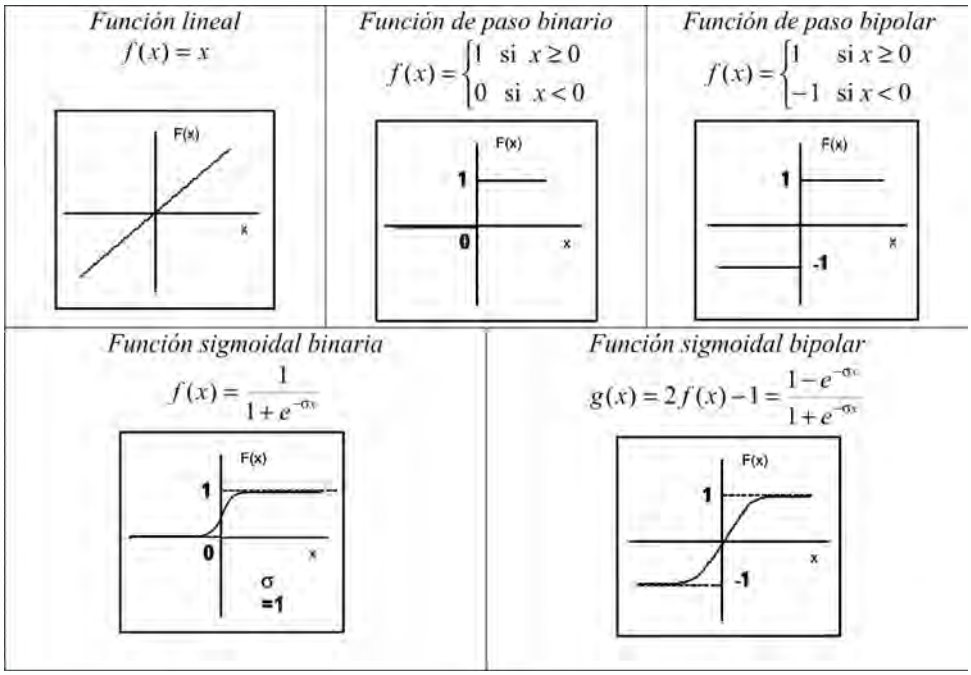
$$Net_j = \sum_{i=1}^{N} x_i w_{ji} + \theta_j \qquad [1.1]$$

$$\begin{aligned} Net_j &= w_1 x_{j1} + w_2 x_{j2} + \ldots + w_i x_{ji} + \ldots + w_N x_{jN} + \theta_j \\ Net_j &= \mathbf{w}^{\mathbf{T}} \mathbf{X}_j + \theta_j \end{aligned} \qquad [1.2]$$

La salida de la neurona artificial está determinada por una función de activación (*Fact)*, tal como se aprecia en la ecuación 1.3.

$$y_j = Fact_j(Net_j) \qquad [1.3]$$

La función de activación generalmente es del tipo escalón, lineal o sigmoidal según se presenta en la Figura 1.4.



Red neuronal artificial

La neurona artificial por si sola posee una baja capacidad de procesamiento y su nivel de aplicabilidad es bajo, su verdadero potencial radica en la interconexión de las mismas, tal como sucede en el cerebro humano. Esto ha motivado a diferentes investigadores a proponer diversas estructuras para conectar neuronas entre si, dando lugar a las redes neuronales artificiales. En la literatura encontramos múltiples definiciones, de las cuales queremos destacar las siguientes, que se ajustan muy bien al concepto de red que seguiremos a lo largo de este libro.

La Agencia de Investigación de Proyectos Avanzados de Defensa (DARPA), define una red neuronal artificial como un sistema compuesto de muchos elementos simples de procesamiento los cuales operan en paralelo y cuya función es determinada por la estructura de la red y el peso de las conexiones, donde el procesamiento se realiza en cada uno de los nodos o elementos de cómputo.

Según Haykin, una red neuronal es un procesador paralelo masivamente distribuido que tiene una facilidad natural para el almacenamiento de conocimiento obtenido de la experiencia para luego hacerlo utilizable. Se parece al cerebro en dos aspectos:

1. El conocimiento es obtenido por la red a través de un proceso de aprendizaje.
2. Las conexiones entre las neuronas, conocidas como pesos sinápticos, son utilizadas para almacenar dicho conocimiento.

Kohonen, las define como redes de elementos simples (usualmente adaptativos) masivamente interconectados en paralelo y con organización jerárquica, las cuales intentan interactuar con los objetos del mundo real del mismo modo que lo hace el sistema nervioso biológico.

En síntesis se puede considerar que una red neuronal artificial es un sistema de procesamiento de información que intenta emular el comportamiento con las redes neuronales biológicas. Las redes neuronales artificiales han sido desarrolladas como generalizaciones de modelos matemáticos del conocimiento humano o de la biología neuronal, con base en las siguientes consideraciones:

1. El procesamiento de información se realiza en muchos elementos simples llamados neuronas.
2. Las señales son pasadas entre neuronas a través de enlaces de conexión.
3. Cada enlace de conexión tiene un peso asociado, el cual, en una red neuronal típica, multiplica la señal transmitida.
4. Cada neurona aplica una función de activación (usualmente no lineal) a las entradas de la red (suma de las señales de entrada pesadas) para determinar su señal de salida.

La distribución de las neuronas dentro de una red neuronal artificial se realiza formando niveles de un número de neuronas determinado. Si un conjunto de neuronas artificiales reciben simultáneamente el mismo tipo de información, lo denominaremos capa. En una red podemos diferenciar tres tipos de niveles:

- **Entrada**: Es el conjunto de neuronas que recibe directamente la información proveniente de las fuentes externas de la red.
- **Oculto**: Corresponde a un conjunto de neuronas internas a la red y no tiene contacto directo con el exterior. El número de niveles ocultos puede estar entre cero y un número elevado. En general las neuronas de cada nivel oculto comparten el mismo tipo de información, por lo que formalmente se denominan Capas Ocultas. Las neuronas de las capas ocultas pueden estar interconectadas de diferentes maneras, lo que determina, junto con su número, las distintas arquitecturas de redes neuronales.

- **Salida**: Es el conjunto de neuronas que transfieren la información que la red ha procesado hacia el exterior.

En la figura 1.5, se puede apreciar la estructura de capas de una red neuronal artificial con varios niveles.

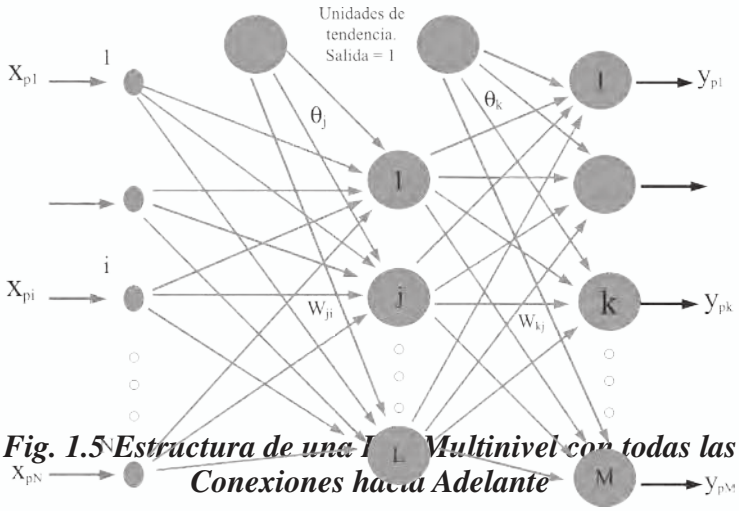


Fig. 1.5 Estructura de una Red Multinivel con todas las Conexiones hacia Adelante

Arquitecturas de redes neuronales artificiales

La arquitectura de una red neuronal artificial es la forma como se organizan las neuronas en su interior y está estrechamente ligada al algoritmo de aprendizaje usado para entrenar la red. Dependiendo del número de capas, definimos las redes como monocapa y multicapa; y si tomamos como elemento de clasificación la forma como fluye la información, definimos las redes como Feedforward y Recurrentes. En este libro conservamos el Anglicismo Feedforward, para las redes cuya información fluye en un solo sentido desde el nivel de entrada hacia la capa de salida.

Redes monocapa

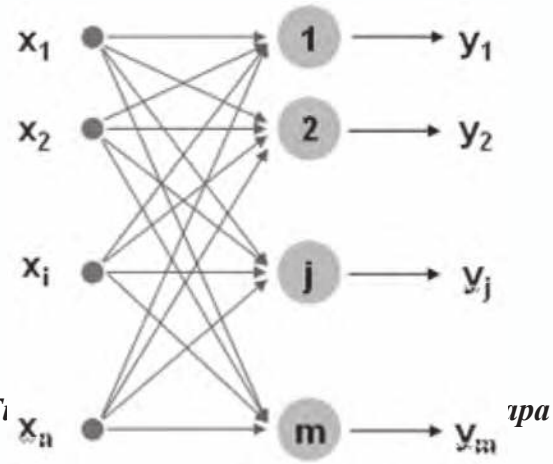
En la figura 1.6, observamos dos niveles de neuronas, el primero asociado al vector de entrada, pero no hay ningún tipo de procesamiento de estos datos, por esta razón no se considera formalmente como una capa, únicamente sirve de interfaz entre los datos de entrada y las siguientes capas de neuronas.

Este primer nivel tiene la misma dimensión del vector de entrada, la

información entra al mismo nivel y los datos son transferidos al siguiente nivel, modificados por los pesos sinápticos. Como las neuronas de este nivel reciben el mismo tipo de información lo denominamos capa y, a su vez, corresponde a la salida de la red, la llamaremos Capa de Salida. Notemos que en esta arquitectura solo disponemos de una capa de procesamiento, de ahí su nombre arquitectura monocapa.

Observemos que hay conectividad total entre el nivel de entrada y la capa de salida, pues todas las neuronas de entrada están conectadas con todas las neuronas de salida, por ejemplo, la neurona de entrada ***i-ésima*** se conecta a las ***m*** neuronas de salida.

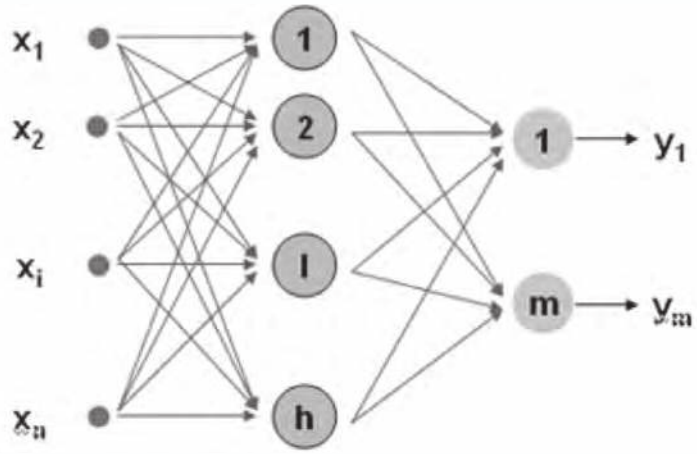
La capa de salida tiene ***m*** neuronas, por lo que luego del entrenamiento, la red neuronal establecerá una relación matemática de un espacio ***n*** dimensional a uno ***m*** dimensional.



Redes multicapa

En este caso, la red tiene un nivel de entrada con ***n*** neuronas y una capa de salida de ***m*** neuronas; cuyo comportamiento es similar al que describimos en la red monocapa. La diferencia sustancial, es que incluimos una nueva capa intermedia entre la entrada y la salida, a esta capa la denominaremos Capa Oculta, que está conformada por ***h*** neuronas.

Como en el caso anterior, la información fluye en una única dirección, de la entrada a la capa oculta y finalmente, a la capa de salida, además existe conectividad total entre ellas. En este ejemplo, presentamos una única capa oculta, pero una red puede tener más de una capa intermedia. ¿Por qué el nombre de oculta? Simplemente porque ésta no tiene contacto con los datos que modelan el mundo real, es decir, los datos de entrada y salida.



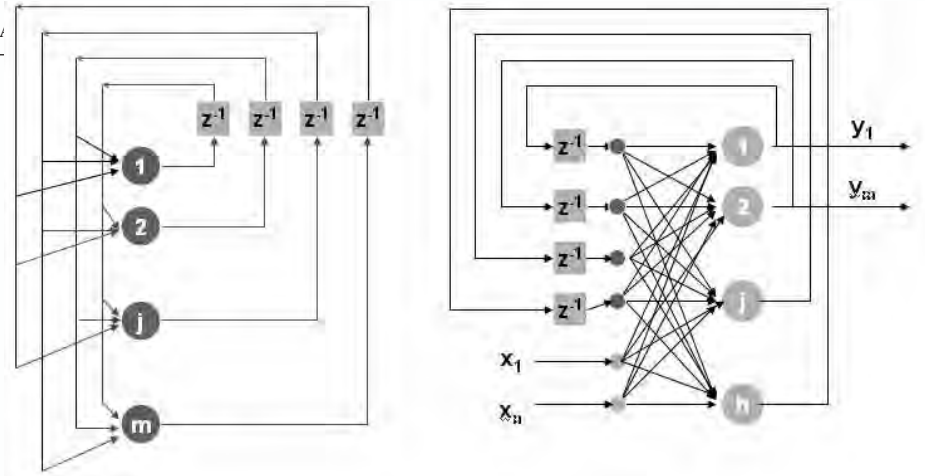
Redes feedforward

En este tipo de red neuronal artificial, la información fluye en un único sentido desde las neuronas de entrada a la capa o capas de procesamiento, para los casos de redes monocapa y multicapa, respectivamente; hasta llegar a la capa de salida de la red neuronal. En la figura 1.6, la información fluye desde la neurona ***i-ésima*** de entrada hacia la ***j-ésima*** de salida. En la figura 1.7, la información proviene de la neurona ***i-ésima***, pasa por la neurona ***l-ésima*** y finaliza en las neuronas de salida.

Redes recurrentes

En este tipo de red neuronal, la información no siempre fluye en un sentido, puesto que puede realimentarse hacia capas anteriores a través de conexiones sinápticas. Este tipo de red neuronal puede ser monocapa o multicapa. En este caso, presentamos dos ejemplos de redes neuronales recurrentes monocapa. La salida de cada neurona es pasada por una unidad de retardo y, luego, llevada hacia todas las neuronas, excepto a sí misma. Observe en la figura 1.8.a, que hay conexión desde la neurona ***j-ésima***, hacia las ***m*** neuronas, menos a la misma neurona ***j-ésima***.

En la figura 1.8.b, hemos definido un grupo de neuronas que luego de recibir la información que proviene de las unidades de retardo, llevan su señal de estímulo hacia todas las neuronas de salida. A este conjunto de neuronas, se le suele llamar neuronas de contexto.



a) Monocapa *b) Monocapa con unidades de contexto*

Fig. 1.8 Redes Neuronales Recurrentes

El aprendizaje en las redes neuronales artificiales

El concepto de aprendizaje lo asociamos normalmente, en nuestra vida cotidiana, al proceso de formación que llevamos a cabo en las aulas de clase; adicionalmente, lo podemos asociar al resultado que nos dejan las diversas experiencias que tomamos de a nuestro diario vivir y a la manera como éstas nos condicionan frente a los diferentes estímulos que recibimos del entorno. La Real Academia de la Lengua Española define el aprendizaje como: *"Adquirir el conocimiento de algo por medio del estudio o de la experiencia"*.

Otro punto de vista, es mirar el aprendizaje como un proceso que nos permite apropiar un conocimiento, alguna habilidad, construir actitudes o valores, gracias al estudio, la experiencia o la enseñanza. Este proceso da origen a cambios permanentes, que son susceptibles de ser medidos y que, generalmente, modifican el comportamiento del individuo.

Biológicamente, se acepta que la información memorizada en el cerebro está más relacionada con los valores sinápticos de las conexiones entre las neuronas que con ellas mismas; es decir, el conocimiento se encuentra en las sinapsis. En el caso de las redes neuronales artificiales, se puede considerar que el conocimiento se encuentra representado en los pesos de las conexiones entre las neuronas. Todo proceso de aprendizaje implica un cierto cambio en éstas. En realidad se puede decir que se aprende modificando los pesos sinápticos de la red neuronal artificial.

El aprendizaje para la red neuronal artificial, es la habilidad para aprender del entorno y mejorar su desempeño, es un proceso interactivo que permite ajustar los pesos sinápticos. Según Mendel McClare, el aprendizaje en las redes neuronales artificiales, se puede definir como: *"Un proceso mediante el cual los parámetros libres de una red neuronal artificial son*

adaptados a través de un proceso de estimulación del ambiente en el cual está embebida la red. El tipo de aprendizaje está determinado por la forma como se cambian los parámetros en el proceso".

Teniendo en cuenta lo mencionado en la ecuación 1.4, planteamos una expresión general para modelar el proceso de aprendizaje en las redes neuronales artificiales.

$$w(t+1) = w(t) + \Delta w(t) \qquad [1.4]$$

Donde,

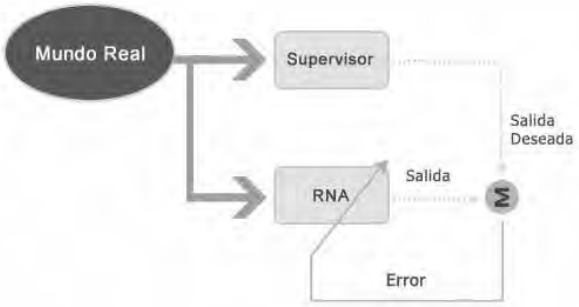
$w(t+1)$	:	Valor actualizado del peso sináptico
$w(t)$	:	Valor actual del peso sináptico
$\Delta w(t)$	:	Variación del peso sináptico

Es importante precisar, que la variación del peso sináptico depende del algoritmo o regla de aprendizaje que se esté utilizando para modificar los pesos sinápticos de la red neuronal artificial. En general, en este libro estudiaremos dos tipos de aprendizaje: el Supervisado y el No–Supervisado.

Aprendizaje supervisado

El aprendizaje Supervisado, cuyo esquema presentamos en la de la figura 1.9, se caracteriza porque el proceso de entrenamiento es controlado por un agente externo llamado ***supervisor o maestro.*** En la figura, el mundo real corresponde al problema a modelar que se representa mediante un conjunto de datos de entrada y salida. El Supervisor opera como un "maestro" que guía el aprendizaje en la red, y conoce las salidas deseadas correspondientes a las respectivas entradas.

Los datos de entrada se le presentan al supervisor y a la red de neuronal simultáneamente, el supervisor propone la salida deseada para ser comparada con la salida de la red neuronal artificial. Para que la labor del supervisor sea exitosa se define en él, un error de entrenamiento como la diferencia entre la salida deseada o esperada y la salida que produce la RNA (Ecuación 1.5).



$$error = d - y \qquad [1.5]$$

Donde:

y : Salida de la red neuronal artificial
d : Salida deseada

Generalmente, en los procesos de aprendizaje se usa un conjunto de patrones de entrenamiento, que está conformado por un conjunto de vectores de entrada $\boldsymbol{X}$ y su correspondiente conjunto de vectores de salida $\boldsymbol{D}$.

$\boldsymbol{X} = \{\boldsymbol{x}_1, \boldsymbol{x}_2,...,\boldsymbol{x}_p,...,\boldsymbol{x}_P\}$ *Conjunto de vectores de entrada*
$\boldsymbol{D} = \{\boldsymbol{d}_1, \boldsymbol{d}_2,...,\boldsymbol{d}_p,...,\boldsymbol{d}_P\}$ *Conjunto de vectores de salida*

Donde cada patrón de entrenamiento está constituido por la pareja ordenada de vectores $\{\boldsymbol{x}_p, \boldsymbol{d}_p\}$, los cuales los podemos definir así:

$\boldsymbol{x_p} = \{x_{p1}, x_{p2}, \ldots, x_{pN}\}$ *Elementos de la entrada de un patrón de entrenamiento*
$\mathbf{d_p} = \{d_{p1}, d_{p2}, \ldots, d_{pM}\}$ *Elementos de la salida de un patrón de entrenamiento*

Por lo regular, la red debe aprender todo el conjunto de patrones de entrenamiento y, por esta razón, no se puede entrenar con un error local, sino que el aprendizaje se hace en términos de un *error global* E_p, que lo calculamos con base en la ecuación 1.6 y define el error cuadrático promedio. Se debe entender el error global, como el error que produce la red en sus diferentes neuronas de salida antes todos los patrones de aprendizaje que se estén utilizando para el proceso de entrenamiento.

$$E_p = \frac{1}{2P}\sum_{p=1}^{P}\sum_{j=1}^{M}(d_{pj} - y_{pj})^2 \qquad [1.6]$$

Donde,

M : Número de neuronas en la capa de salida
P : Números de patrones de entrenamiento

Aprendizaje no-supervisado

En este caso el vector de datos que describe el problema, se le presenta directamente a la red, pero ahora ya no hay un supervisor o maestro que guía el aprendizaje. En este caso los pesos de la red se calculan en función de la caracterización que se haga de la entrada que la red neuronal artificial esté recibiendo, de acuerdo a un objetivo específico que nos permite obtener el conocimiento que queremos representar con la red. El esquema de este tipo de aprendizaje lo presentamos en la figura 1.10.

Fig. 1.10 Aprendizaje No-Supervisado

Ejemplo de procesamiento de la información en una red neuronal

El procesamiento de la información que realiza una red neuronal artificial, generalmente, consiste en recibir la información de entrada e ir propagándola por toda la estructura de la red neuronal hasta generar la salida de la misma

El vector de entrada lo denotaremos por $\boldsymbol{x}$ **,** y está conformado por los diferentes valores que va recibir la red neuronal. Este vector se propagará por las diferentes neuronas y capas que la conforman hasta generar la salida de la red neuronal.

En este ejemplo presentaremos el procesamiento necesario para generar la salida de la red neuronal mostrada en la figura 1.11. El vector de entrada esta compuesto por tres componentes $\boldsymbol{x} = [x_1, x_2, x_3]$. El objetivo es generar la expresión para el vector de salida $\boldsymbol{y^o} = [\, y_1^o, y_2^o]$

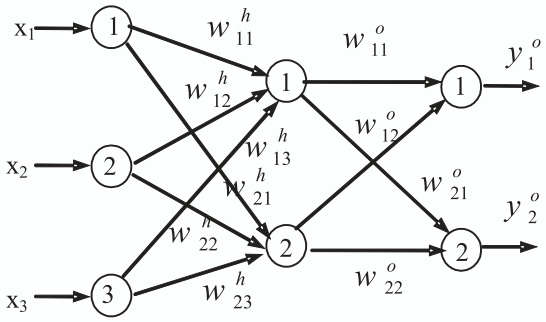


Fig. 1.11 Estructura de conexión entre neuronas

En el anterior esquema la entrada neta de la neurona de la capa oculta es:

$$Net_j^h = \sum_{i=1}^{3} w_{ji}^h x_i$$
$$Net_j^h = w_{j1}^h x_1 + w_{j2}^h x_2 + w_{j3}^h x_3 \qquad [1.7]$$

y su salida:

$$y_j^h = Fact\left(Net_j^h\right) \qquad [1.8]$$

donde,

$$y_1^h = Fact\left(w_{11}^h\, x_1 + w_{12}^h\, x_2 + w_{13}^h\, x_3\right)$$
$$y_2^h = Fact\left(w_{21}^h\, x_1 + w_{22}^h\, x_2 + w_{23}^h\, x_3\right)$$

Empleando las ecuaciones 1.7 y 1.8, la salida se puede expresar como:

$$Net\,{}_1^o = w_{11}^o y_1^h + w_{12}^o y_2^h \qquad y_1^o = Fact\left(Net\,{}_1^o\right) \qquad [1.9]$$

$$Net\,{}_2^o = w_{21}^o y_1^h + w_{22}^o y_2^h \qquad y_2^o = Fact\left(Net\,{}_2^o\right) \qquad [1.10]$$

Nivel de aplicación

Las redes neuronales artificiales, con su inherente capacidad para extraer el conocimiento a partir de datos complejos e imprecisos, sin requerir un modelo *a priori* del problema, se han convertido en una herramienta útil para el procesamiento de series de tiempo, o para el reconocimiento o cla-

sificación de patrones. Esta capacidad está asociada al aprendizaje que la red hace del problema, convirtiéndose en una técnica emergente que supera ampliamente a las técnicas computacionales tradicionales o determinísticas en la solución de problemas complejos. Una red entrenada puede ser vista como un experto en el manejo de la información que se le ha dado para analizar, este experto puede ser utilizado para proporcionar proyecciones ante nuevas situaciones de interés.

Entre las características y ventajas representativas de las redes neuronales artificiales tenemos:

- *Capacidad de aprendizaje*. Las redes neuronales artificiales son capaces de aprender a partir de un conjunto de ejemplos que sean representativos o intenten modelar el problema a solucionar. Ésta es una de las características más poderosas, pues permite que la red neuronal se adapte a nuevas situaciones siempre y cuando exista la información necesaria para realizar el proceso de entrenamiento. Como en el caso de nosotros lo seres humanos, la calidad de la información o datos que le entreguemos a la red, va a ser fundamental para garantizar la calidad del aprendizaje, en otras palabras, nosotros garantizamos que "la red aprende lo que le enseñemos".
- *Capacidad de generalización*. Cuando se usan las redes neuronales artificiales se necesita que ellas sean capaces de extender su respuesta adecuada a eventos o datos que no han sido utilizados en la fase de aprendizaje, a esta característica la denominaremos: generalización. Esta propiedad de nosotros los humanos, nos permite extender un conocimiento adquirido bajo ciertas circunstancias, a situaciones si-milares pero no exactas, también la tienen las redes neuronales, si los datos son representativos del problema. Lo que buscamos es que la red aprenda la relación entre la entrada y la salida usando algunos ejemplos y, luego, a la red ya entrenada, si le suministramos un conjunto de entradas del mismo contexto del problema, la salida generada debe ser la correcta.
- *Capacidad para extraer características esenciales los datos*. Algunos tipos de redes neuronales artificiales, son capaces de extraer la información relevante que hay en ciertos datos y rechazar aquella información que no sea importante. Esta característica se ha aplicado con mucho éxito en problemas asociados a la minería de datos.
- *Capacidad de asociación*. Esta es una característica muy importante de la inteligencia de nosotros los seres humanos e incluso, podemos atrevernos a decir que es más elevada que la capacidad de aprender a partir de ejemplos. ¿Qué entendemos por asociación? Cuando establecemos una relación entre dos conceptos. Por ejemplo, si decimos "aprendizaje en redes neuronales", para todos nosotros, esperamos

que inmediatamente se genere una asociación al concepto de modificación de pesos sinápticos. Si decimos esta misma frase en un auditorio que no conozca de Redes Neuronales, muy seguramente no le dirá nada o no la asociarán a un conocimiento. Más adelante aprenderemos sobre un tipo de aprendizaje que trata de emular esta característica y sobre la red, ya entrenada, que es capaz de asociar.

- *Capacidad de agrupación*. Hemos venido evolucionando en las diferentes formas de aprender que presenta el ser humano, iniciamos con un aprendizaje basado en la repetición de una serie de ejemplos de entrada y salida que modelan el conocimiento al aprender, esto podría asociarse a la memorización, pero les garantizamos que las redes neuronales artificiales son capaces de generalizar. Luego vimos que al igual que los seres humanos, las redes neuronales artificiales son capaces de separar la información relevante de la irrelevante en un conjunto de datos. Pasamos al conocimiento adquirido por asociación y ahora presentamos la facilidad que tenemos para agrupar conocimiento. Por ejemplo, tenemos los libros de nuestra comunidad y queremos organizarlos en una biblioteca… ¿Qué pasa si pedimos a diferentes grupos que los organicen? Cada uno de ellos buscará algún criterio para hacerlo; pues bien, veremos cómo las redes neuronales artificiales emulando nuestra capacidad, podrán, igualmente solucionar problemas de asociación por sectores o grupos de información, algo que la literatura se conoce como *clustering* o agrupamiento.
- *Aprendizaje Adaptativo.* Hemos visto que las redes neuronales artificiales, al igual que las naturales, aprenden a realizar algunas tareas gracias a un proceso de entrenamiento utilizando ejemplos lo suficientemente ilustrativos para ello, por lo que no se requiere de modelos a priori, del sistema o problema que queremos solucionar. Pero el nivel de emulación no se limita a este importante hecho, sino que la redes neuronales artificiales son capaces de modificar su propia estructura de pesos sinápticos para adaptarse a nuevas situaciones que se presenten en el sistema y modifiquen su desempeño. Visto desde otra perspectiva, una misma arquitectura de red neuronal puede aprender diferentes problemas modificando su matriz de pesos para adaptarse a las diferentes tareas que le pedimos sean solucionadas.
- *Auto-Organización*. En el caso concreto de los modelos de redes neuronales artificiales propuestos por Teuvo Kohonen, su arquitectura se puede modificar completamente dependiendo de la forma como estén organizados los datos o del objetivo a cumplir. De alguna manera, la red neuronal trata de seguir (imitar) la estructura de los datos que le presentemos a su entrada sin necesidad de que le impongamos una salida predeterminada.
- *Tolerancia a fallos.* Las redes neuronales artificiales son tolerables

a fallos en los datos (ruido, distorsiones, datos incompletos) debido a que la red no almacena el conocimiento de una manera localizada, sino que él mismo se encuentra distribuido en toda la estructura de la red.

- *Operación en Tiempo Real.* Las redes neuronales deben su fortaleza a su capacidad para procesar información en paralelo, en implementaciones hardware de redes neuronales artificiales se puede realizar este tipo de procesamiento, lo que permite un cálculo de la información a muy altas velocidades, suficientes para cumplir las restricciones de tiempo real, de la gran mayoría de las aplicaciones en las cuales se utilizan las redes neuronales artificiales. De igual manera, en implementaciones software, hemos visto que el procesamiento de una neurona es simple y de rápida ejecución; aunque, en este tipo de uso no podemos garantizar el paralelismo, de igual manera por su simplicidad, podemos asegurar que la ejecución total de una red, se hace en un tiempo lo suficientemente corto como para garantizar las restricciones de tiempo real para aplicaciones típicas, como de filtrado, control, identificación de sistemas, entre otras.

Los computadores digitales actuales, superan al hombre en su capacidad de cálculo numérico y el manejo de símbolos relacionales. Sin embargo, el hombre puede solucionar problemas mucho más complejos de percepción (por ejemplo, reconocer a un amigo entre un tumulto desde un simple vistazo de su cara o al escuchar su voz, incluso por el modo de caminar; definir la habitabilidad de un aula a partir de sensaciones de temperatura, ruido, humedad, iluminación, etc.) a muy altas velocidades y sin necesidad de concebir un complejo modelo matemático o computacional. La respuesta está en la arquitectura del sistema neuronal biológico que es completamente diferente a la arquitectura del computador tradicional von Neumann, como se indicó en la Tabla 1.1. Estas diferencias, le brindan un factor de diferenciación y aumento del desempeño a las redes neuronales artificiales en una gran variedad de campos de aplicación como mencionados a continuación:

- *Reconocimiento y Clasificación de Patrones.* Esta es una de las grandes fortalezas de las Redes Neuronales Artificiales y, al igual, que las naturales, tienen una especial facilidad para solucionar problemas de reconocimiento y clasificación de patrones, en especial se puede encontrar una gran cantidad de problemas resueltos utilizando las redes tipo Perceptron Multicapa.
- *Categorización de Patrones* ("*clustering*"). Otra área donde las redes neuronales han sido utilizadas con éxito es el agrupamiento o clustering. Este tipo de aplicación es muy útil en minería de datos donde se necesitan encontrar grupos de patrones con características similares.

- *Procesamiento de señales.* Otro campo de aplicación de las redes neuronales es el procesamiento de señales como, por ejemplo, la voz. En este caso la red se puede usar para realizar alguna parte del proceso de extracción de características o como una etapa clasificadora
- *Optimización*. Aunque la principal fortaleza de las redes neuronales no es la solución de este tipo de problemas y existen otras técnicas de inteligencia computacional para ello, como los Algoritmos Genéticos, se han utilizado algunos modelos de RNA, como las redes neuronales de Hopfield y Kohonen, para resolver problemas clásicos de optimización en ingeniería, v.gr. el problema del Agente Viajero.
- *Control.* En las industrias hay procesos donde han sido usadas con éxito las redes neuronales para el modelado y control de los mismos.
- *Medicina.* Las redes neuronales se han usado para la extracción de características, a partir de una imagen o como un clasificador para determinar la pertenencia a una clase del patrón representado por una imagen.
- *Gestión financiera*. En este caso se ha aprovecha las capacidad de las redes neuronales para modelar series de tiempo, lo cual nos permite utilizarlas para hacer predicciones de la misma. En este contexto las redes neuronales se han usado, por ejemplo, para modelar el comportamiento de variables económicas, como el valor de la acción de una empresa.
- *Robótica.* La robótica móvil es un área donde se puede sacar partido de la capacidad de aprendizaje de las redes neuronales. Básicamente, para ayudarle al robot a tomar decisiones en ambientes cambiantes.
  En este momento se ha dado solución a problemas que se caracterizan por tener altos niveles de incertidumbre que con métodos tradicionales jamás se habría obtenido. Soluciones tan novedosas e interesantes como la reconstrucción cráneofacial para la identificación de hombres, música neurocomputacional, sistemas de detección de virus en computadores, identificación de usuarios en cajeros automáticos desde la imagen del iris de los ojos, reconocimiento de emisores en comunicaciones, diagnóstico de hepatitis, recuperación de telecomunicaciones ante fallas en el software, interpretación de palabras chinas, detección de minas submarinas, análisis de texturas, reconocimiento de objetos tridimensionales, reconocimiento de textos manuscritos, entre otros.

Capítulo 2

REDES NEURONALES PERCEPTRON Y ADALINE

Introducción

El Perceptron fue el primer modelo de RNA presentado a la comunidad científica por el psicólogo Frank Rosenblatt en 1958. Como es natural despertó un enorme interés en la década de los años sesenta, debido a su capacidad para aprender a reconocer patrones sencillos con una superficie de separación li-neal, razón por la cual fue también objeto de severas críticas que terminaron por dejar en el olvido la propuesta de Rosenblatt. La estructura del Perceptron es supremamente sencilla: en su entrada posee varias neuronas lineales que se encargan de recibir el estímulo externo de la red y a la salida presenta una única capa de neuronas, con función de activación binaria o bipolar lo cual trae como consecuencia que la salida de cada neurona esté entre dos posibles valores.

De manera casi simultánea al desarrollo del Perceptron, Bernard Widrow y su estudiante Marcian Hoff presentaron su red neuronal ADALINE, esta red es similar al Perceptron pero en vez de tener una función de activación binaria o bipolar posee una función de activación lineal. La red ADALINE, a pesar de tener las mismas limitaciones de la red Perceptron, fue un gran adelanto en el desarrollo de las redes neuronales pues el mecanismo de aprendizaje que utiliza es basado en elementos de la teoría de optimización; específicamente se usa la propuesta del gradiente descendente para la deducción de la regla de entrenamiento. El uso del gradiente descendente es la base de los algoritmos de aprendizaje que hoy se usan para entrenar redes neuronales más sofisticadas que las presentadas en este capítulo. Uno de los aspectos más interesante de la red ADALINE es su aplicación en problemas de filtrado de señales, pues su estructura se ajusta a la de un filtro adaptativo.

Red neuronal perceptron

Arquitectura y funcionamiento

El Perceptron es una red monocapa pues la capa de entrada, al no realizar procesamiento alguno sobre la señal de estímulo, no se tiene en la cuenta y por tanto, esta red posee solo una capa de procesamiento que es la misma de salida. Posee conectividad total, ya que la neurona de la capa de salida está conectada a todas las unidades o neuronas de entrada, tal como lo observamos en la figura 2.1. Además, por lo general se le implementan unidades de tendencia o umbral para que la superficie de separación no se quede anclada en el origen del espacio *n-dimensional* en donde se esté realizando la separación lineal, ya que existen algunos problemas de clasificación de patrones que sin el término de umbral no tendrían solución. La función de activación que utiliza la neurona de salida es la tipo escalón binario [0,+1] o bipolar [-1,+1].

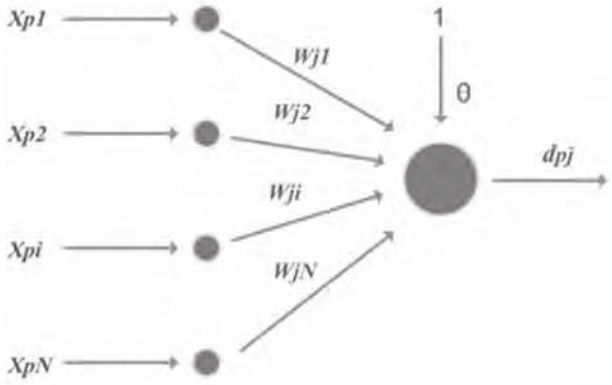


Fig. 2.1 Arquitectura de un Perceptron

Para comprender de mejor manera el funcionamiento de un Perceptron, veamos apoyados en la figura 2.2, el comportamiento de esta red cuando la dimensión del espacio de entrada es igual a dos, esto significa que la red tendrá dos entradas y por simplicidad se supondrá una sola neurona de procesamiento.

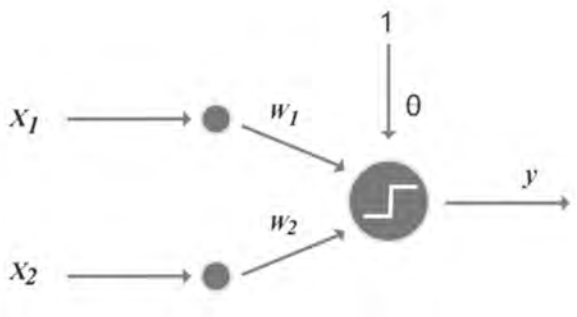


Fig. 2.2 Perceptron para trabajar puntos en dos dimensiones

La única neurona de salida del Perceptron realiza la suma ponderada de las entradas, se suma el umbral y se pasa el resultado a una función de activación binaria o bipolar. Debido a la naturaleza de la función de activación la salida de la red tiene solo dos opciones para la señal de salida *y*, 1 ó 0. Si generamos la salida de la red para todos los posibles valores del espacio de entrada y, si "pintamos" de negro el semiplando donde la salida es 1, o de blanco si la salida es 0, el plano quedará dividido como mostramos en la figura 2.3.

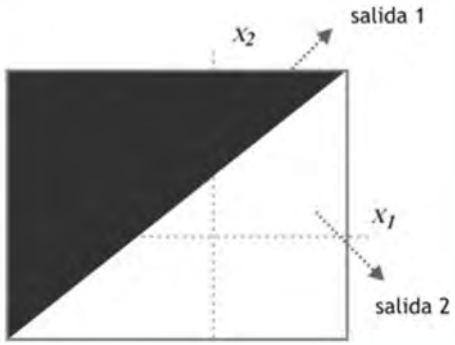


Fig. 2.3 Semiplanos generados por un Perceptron

Para analizar matemáticamente cuál es la superficie de separación que genera el Perceptron en el plano cartesiano, tomemos la expresión de la entrada neta,

$$Neta = x_1 * w_1 + x_2 * w_2 + \theta \tag{2.1}$$

La superficie de separación se tiene exactamente cuando la entrada neta es igual a cero,

$$x_1 * w_1 + x_2 * w_2 + \theta = 0 \tag{2.2}$$

Teniendo esta condición, reorganicemos la expresión de la siguiente manera,

$$x_2 = -\frac{w_1}{w_2} x_1 - \frac{\theta}{w_2} \tag{2.3}$$

La superficie de separación generada por el Perceptron corresponde a una línea recta y, este comportamiento, se puede extender a problemas donde tenemos tres entradas y el espacio de trabajo será tridimensional. En este caso la superficie de separación corresponderá a un plano con la siguiente ecuación,

$$x_3 = -\frac{w_1}{w_3} x_1 - \frac{w_2}{w_3} x_2 - \frac{\theta}{w_3} \tag{2.4}$$

Extendamos el análisis a problemas donde la dimensión del vector de patrones de entrada del Perceptron es *n,* ahora la superficie de separación lineal es de dimensión (*n-1)* que denominaremos *hiperplano.* Cuando entrenamos un Perceptron modificaremos sus pesos, lo cual trae como consecuencia la variación de la superficie de separación que está generando.

En la clasificación de patrones podemos aplicar directamente los conceptos de Perceptron, por ejemplo, tenemos un conjunto de datos con dos clases que representamos en el plano de la figura 2.4 con círculos negros y grises. El perceptron propuesto para resolver esta tarea genera una línea que separa las dos clases adecuadamente.

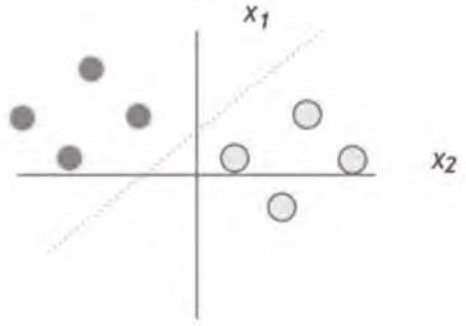


Fig. 2.4 Regiones de separación de un Perceptron

Algoritmo de aprendizaje

Antes de pasar a describir los pasos del algoritmo de aprendizaje del Perceptron, vamos a presentar la nomenclatura utilizada.

N Número de neuronas en la capa de entrada.
x_{pi} Componente *i-ésima* de la entrada correspondiente al *p-ésimo* patrón.
$Neta_{pj}$ Entrada neta de la neurona *j-ésima* de la única capa de procesamiento del Perceptron.
w_{ji} Valor del peso de la conexión entre la *j-ésima* neurona de la capa de procesamiento y la *i-ésima* neurona de la capa de entrada.
θ_j Valor del umbral o tendencia para la *j-ésima* neurona de la capa de procesamiento.
y_{pj} Valor de salida de la *j-ésima* neurona de la capa de procesamiento.
d_{pj} Valor de salida deseado para la *j-ésima* neurona de la capa de procesamiento.
e_p Valor del error para el *p-ésimo* patrón de aprendizaje.
α Tasa de aprendizaje.

Paso 1
Asignamos los valores iniciales de forma aleatoria a los pesos w_{ji} y al umbral θ_j. Se sugiere que estos valores estén en un rango entre -1 y +1.

Paso 2
Presentamos a la red el vector de entrada $\boldsymbol{x_p}$ y especificar el vector de salida deseada $\boldsymbol{d_{pj}}$.

$$\boldsymbol{x_p} = \{x_{p1}, x_{p2}, \ldots, x_{pN}\} \tag{2.5}$$

Paso 3
Calculamos los valores netos procedentes de las entradas para las unidades de la capa de salida.

$$Neta_{pj} = \sum_{i=1}^{N} w_{ji} x_{pi} + \theta_j \tag{2.6}$$

Paso 4
Calculamos la salida de la red.

$$y_{pj} = f_j\,(Neta_{pj}) \tag{2.7}$$

Paso 5
Actualizamos los pesos de la capa de salida.

$$w_{ji}(t+1) = w_{ji}(t) + \alpha\left[d_{pj} - y_{pj}\right] x_{pi} \tag{2.8}$$

Paso 6

Calculamos el error del *p-ésimo* patrón.

$$e_p = \frac{1}{2}\sum_{j=1}^{M}(d_{pj} - y_{pj})^2 \qquad (2.9)$$

Paso 7

Si el error es diferente de cero para cada uno de los patrones aprendidos volver al paso 2.

La velocidad de convergencia la ajustamos con el valor de parámetro de aprendizaje (α), normalmente dicho parámetro debe ser un número pequeño (del orden de 0.05 a 0.25), aunque este valor dependerá de las características del problema que estemos resolviendo.

Vamos a proponer como un ejemplo del proceso de aprendizaje, la solución de la función lógica OR usando un Perceptron. Los patrones de entrada para el entrenamiento de la red se presentan en la siguiente tabla.

Tabla No 2.1. Patrones de entrenamiento

Patrón	x_1	x_2	*d (OR)*
$\mathbf{p_1}$	0	0	0
$\mathbf{p_2}$	0	1	1
$\mathbf{p_3}$	1	0	1
$\mathbf{p_4}$	1	1	1

Cuya representación en forma matricial viene dada por,

$$X = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix}$$

$$D = \begin{bmatrix} 0 & 1 & 1 & 1 \end{bmatrix}$$

La salida de la red se produce de acuerdo a la siguiente condición:

$$\text{Si} \quad w_1 x_1 + w_2 x_2 \geq 0 \longrightarrow y = 1$$

$$\text{Si} \quad w_1 x_1 + w_2 x_2 < 0 \longrightarrow y = 0$$

Si seguimos los pasos del algoritmo, empezamos por elegir los valores de los pesos, *bias* y factor de aprendizaje:

θ=0.5, w_1=-0.5, w_2=-0.5, α=1

Se deben evaluar los cuatro patrones de entrada 0 0, 0 1, 1 0, 1 1; sin embargo, para este ejemplo, sólo evaluaremos el patrón 1,16, que de acuerdo a la tabla 2.1 tiene salida igual a 1.

- Con el patrón 1,16, la suma ponderada se calcula así,

$$Net = w_1 x_1 + w_2 x_2 + \theta = (-0.5)(1) + (-0.5)(1) + 0.5 = -0.5$$

la salida de la red es $y = f(Net) = 0$

Calculamos el error $e(t) = d(t) - y(t) = 1 - 0 = 1$

Se procede con la modificación de los pesos,

$$\mathbf{w}(t+1) = \mathbf{w}(t) + \Delta w = \begin{cases} w_1(t+1) = -0.5 + (1.0)(1.0-0)(1.0) = 0.5 \\ w_2(t+1) = -0.5 + (1.0)(1.0-0)(1.0) = 0.5 \\ \theta(t+1) = 0.5 + (1.0)(1.0-0) = 1.5 \end{cases}$$

Con estos nuevos valores de pesos y *bias* se vuelven a calcular las salidas como se hizo con el patrón inicial 1 1. El procedimiento continua de manera iterativa hasta que los pesos tengan los valores adecuados para modelar la función lógica OR.

Red neuronal adaline (adaptative linear element)

Arquitectura

La red ADALINE es similar al Perceptron, pero en vez de tener una función de activación binaria o bipolar posee una función de activación lineal, esto se puede observar en la figura 2.5

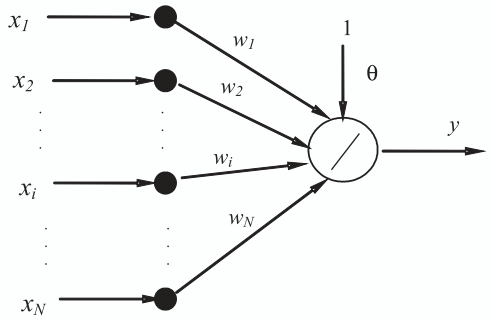


Fig. 2.5. Estructura de una red tipo ADALINE

El modelo del ADALINE fue desarrollado por Bernard Widrow y su estudiante Marcian Hoff a comienzos de la década de los sesenta. Como la neurona de esta red tiene una función de activación lineal, el valor de salida será igual al valor de la entrada neta que la neurona esté recibiendo, que lo podemos expresar con la ecuación 2.10.

$$y = \sum_{i=1}^{N} w_i x_i + \theta \tag{2.10}$$

Es importante recalcar que la red tipo ADALINE puede tener más de una neurona en su capa de procesamiento, dando origen a un sistema con N entradas y M salidas, como el de la figura 2.6.

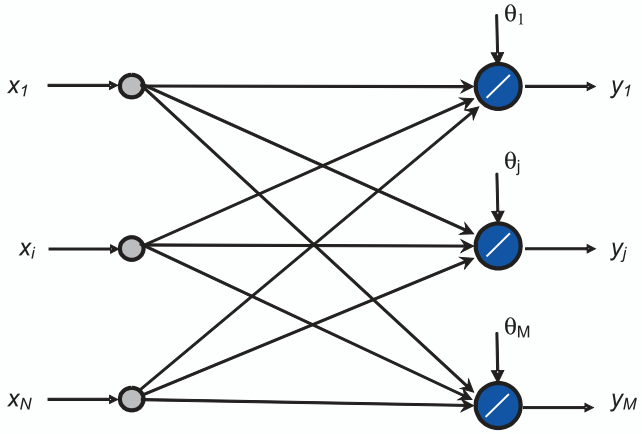


Fig. 2.6 Estructura de una red tipo ADALINE

Algoritmo de aprendizaje

Supongamos que nos encontramos en un sistema montañoso muy complejo en un lugar a gran altura y totalmente cubierto por neblina. Estamos perdidos, pero sabemos que en el valle hay una población donde podremos obtener ayuda, ¿qué estrategia utilizamos para llegar a dicha población? Muy seguramente estamos de acuerdo en que lo mejor es empezar a bajar y siempre mantener esta tendencia, por ejemplo, podríamos seguir el cause de un río, pues esta corriente de agua nos garantiza que siempre caminaremos paulatinamente hacia un lugar más bajo.

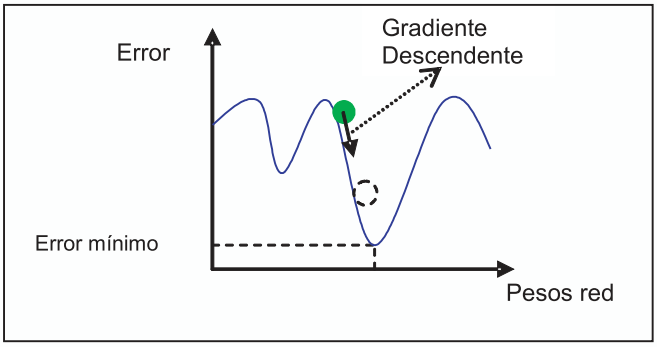


Fig. 2.7 Gradiente Descendente

Este es el principio del algoritmo utilizado en el proceso de aprendizaje de la red ADALINE, en donde, partiendo desde un punto aleatorio en la superficie de error, buscaremos un punto donde el error sea mínimo, siguiendo una trayectoria descendente, como se ilustra en la figura 2.7. Esta técnica de búsqueda se denomina Algoritmo de Gradiente Descendente.

Si recordamos del capítulo 1, la ecuación de actualización de pesos para una red neuronal la podemos definir en términos del peso en el instante t y de la variación del peso sináptico,

$$w(t+1) = w(t) + \Delta w(t) \tag{2.11}$$

donde,

$w(t+1)$	:	Valor actualizado del peso sináptico
$w(t)$	:	Valor actual del peso sináptico
$\Delta w(t)$	:	Variación del peso sináptico

Si variamos los pesos de la red en un factor proporcionar al gradiente negativo del error de la red respecto a los pesos, logramos que los pesos

así generados disminuyan el índice de desempeño (J) que para este caso lo definimos en términos del error cuadrático promedio en la ecuación 2.12 y para una red ADALINE de una neurona de salida, con P patrones de entrenamiento.

$$J = \frac{1}{2P}\sum_{p=1}^{P}(d_p - y_p)^2 \tag{2.12}$$

p : *p-ésimo* patrón del conjunto de entrenamiento
d_p : Valor deseado de salida para el *p-ésimo* patrón
y_p : Valor de salida de la red ADALINE para el *p-ésimo* patrón

El objetivo de algoritmo es encontrar el conjunto de pesos que hagan que J sea mínimo; si hacemos la variación de los pesos proporcional al gradiente negativo, el cambio en el peso de la *i-ésima* conexión la representamos por la ecuación 2.13, donde α es el factor de aprendizaje de la red.

$$\Delta w_i = -\alpha \frac{\partial J}{\partial w_i} \tag{2.13}$$

Si en la ecuación anterior aplicamos la regla de la cadena para obtener la derivada, queda en términos de la derivada del índice de desempeño respecto del error, la derivada del error respecto de la salida de la red ADALINE y la derivada de la salida de la red respecto del peso de la *i-ésima* conexión.

$$\Delta w_i = -\alpha \frac{\partial J}{\partial e}\cdot\frac{\partial e}{\partial y}\cdot\frac{\partial y}{\partial w_i} \tag{2.14}$$

Tras evaluar las derivadas parciales de la ecuación 2.14, obtenemos los siguientes valores,

$$\frac{\partial J}{\partial e} = e = (d_p - y_p)$$
$$\frac{\partial e}{\partial y} = -1$$
$$\frac{\partial y}{\partial w_i} = x_i$$

Estos valores de las derivadas parciales los reemplazamos en la ecuación 2.14 para obtener el valor final de la variación del el peso de la *i-ésima* conexión.

$$\Delta w_i = -\alpha e_p(-1)x_i$$
$$\Delta w_i = \alpha e_p x_i \tag{2.15}$$

Retomando la ecuación 2.11 y 2.15, la expresión para la actualización del peso de la *i-ésima* conexión está dada por la ecuación 2.16.

$$w_i(t+1) = w_i(t) + \Delta w_i$$
$$w_i(t+1) = w_i(t) + \alpha e_p x_i \tag{2.16}$$

Con base en las ecuaciones obtenidas, proponemos un algoritmo de aprendizaje para la red ADALINE, cuyos pasos los describimos a continuación.

Paso 1
Asignamos los valores iniciales de forma aleatoria a los pesos w_{ji} y el umbral θ_j.
Asignamos un valor al parámetro de aprendizaje α.
Definimos un valor para el error mínimo aceptado en el entrenamiento de la red.

Paso 2
Mientras la condición de parada sea falsa, ejecutamos los pasos 3 al 7.

Paso 3.
Para cada patrón de entrenamiento $\{\boldsymbol{x_p}, \boldsymbol{y_p}\}$ ejecutamos los pasos 4 al 5

Paso 4.
Calculamos la salida de la red ADALINE

$$y_{pj} = Neta_j = \sum_{i=1}^{N} w_i x_{pi} + \theta_j \tag{2.17}$$

Paso 5.
Calculamos el error e_p en la salida de la red ADALINE

$$e_p = \frac{1}{2}\sum_{j=1}^{M}(d_{pj} - y_{pj})^2 \tag{2.18}$$

Paso 6.
Actualizamos los pesos

$$w_i(t+1) = w_i(t) + \alpha e_p x_i \qquad (2.\ 19)$$

$$\theta_j(t+1) = \theta_j(t) + \alpha e_p \qquad (2.20)$$

Paso 7.
Calculamos el error global de la red y si es menor que un valor mínimo detenemos el algoritmo. En caso contrario, retornamos al paso 2.

Limitaciones del perpectrón

Como hemos mencionado a lo largo de este capítulo, uno de los principales inconvenientes que tiene el Perceptron es su incapacidad para separar regiones que no sean linealmente separables, como se observa en la figura 2.8. Sin embargo, Rosenblatt ya intuía que un Perceptron multicapa sí podía solucionar este problema pues, de esta manera, se podían obtener regiones de clasificación mucho más complejas, debido a una estructura, como la de la figura 2.9, que incluye además de las capas de entrada y salida, una o más capas internas u ocultas.

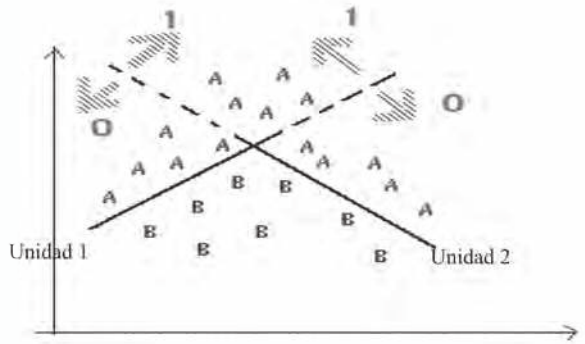


Fig. 2.8 Regiones de separación lineales que genera un Perceptron

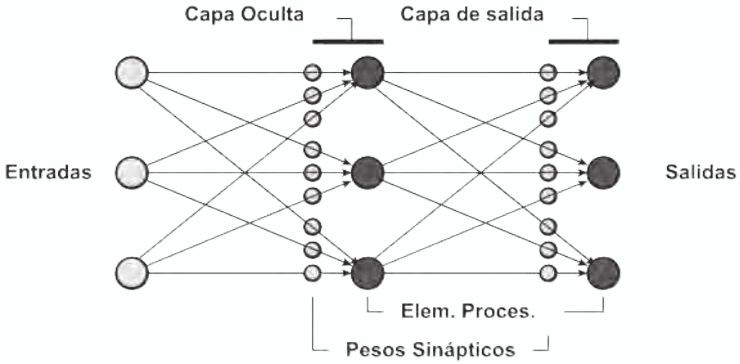
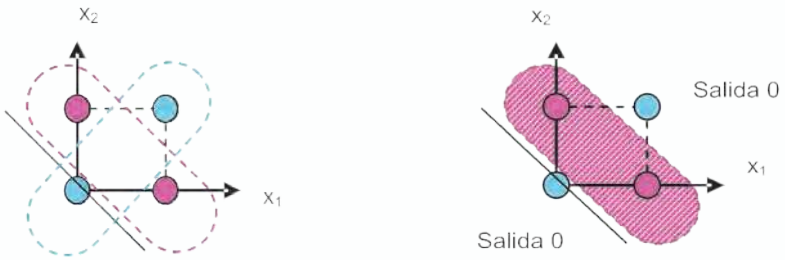


Figura 2.9 Estructura de una Red Neuronal Multicapa

Para ilustrar mejor este planteamiento observe la figura 2.10, donde se muestra que un Perceptron no está en capacidad de solucionar la función lógica XOR, descrita en la Tabla 2.1, ya que no es posible separar linealmente su salidas. Esta situación nos permite comprender de alguna manera los planteamientos de Minsky y Paper, pues se muestra con un problema relativamente sencillo las limitaciones de este tipo de red neuronal.

Tabla 2.1 Función Lógica XOR

Patrón	x_1	x_2	*X OR*
$\mathbf{x^1}$	0	0	0
$\mathbf{x^2}$	0	1	1
$\mathbf{x^3}$	1	0	1
$\mathbf{x^4}$	1	1	0



a) Representación en el espacio de entradas para la función XOR

b) Clasificación sin separación lineal

Fig. 2.10 Solución de la XOR con un Perceptron

Una arquitectura de Perceptron multinivel con dos neuronas en el nivel de entradas, una única capa oculta con dos neuronas y la respectiva neurona de la capa de salida, como se ilustra en la siguiente figura 2.11, con adecuado entrenamiento puede resolver el problema de separación no lineal planteado por la función XOR.

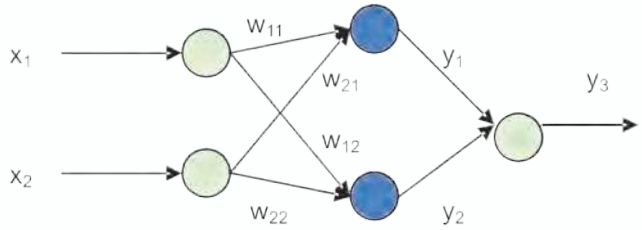


Fig. 2.11 Perceptron multicapa propuesto para la solución la función XOR

Lo anterior es cierto y ya lo vislumbraba Rosenblatt, pero en ese entonces se tenían algunos interrogantes sin respuesta ¿Cómo entrenar un Perceptron multicapa? ¿Cómo evaluar el error en las capas ocultas si no hay un valor deseado conocido para las salidas de estas capas?

Este inconveniente es inherente a la forma como opera el algoritmo de aprendizaje del Perceptron pues, para poder variar los pesos, lo hace con base en el error de salida y, por tanto, se necesita conocer la salida deseada de la capa a la cual se la va a realizar dicha operación. En el Perceptron sin capas ocultas, este problema no existe pues al ser el algoritmo de aprendizaje del Perceptron del tipo supervisado, las salidas de la red están definidas. Si el Perceptron es multicapa es imposible conocer con anterioridad el valor de la salida de una de las capas ocultas, motivo por el cual el algoritmo de aprendizaje que posee el Perceptron se hace inutilizable en este tipo de redes neuronales.

Una solución a este problema la planteó formalmente, y por primera vez,Werbos, y se denominó algoritmo de aprendizaje Backpropagation que se estudiará en el siguiente capítulo.

Aproximación práctica

Construcción de un perceptron usando MATLAB®

En este proyecto proponemos generar un Perceptron de dos entradas usando los bloques de *Simulink* de la herramienta de redes neuronales del MATLAB®. Usando los bloques de *Simulink* de esta aplicación como presentamos en la figura 2.12, podemos representar el funcionamiento básico de un Perceptron. En el diagrama de simulación, el bloque *dotprod* está en la biblioteca *weight functions* y el bloque *hardlim* está en la biblioteca

transfer functions.

Para simular los pesos y la entrada se usan bloques *constant* (biblioteca *sources*) y para visualizar la salida de la red y la neta se usan bloques *display* (biblioteca *sinks*).

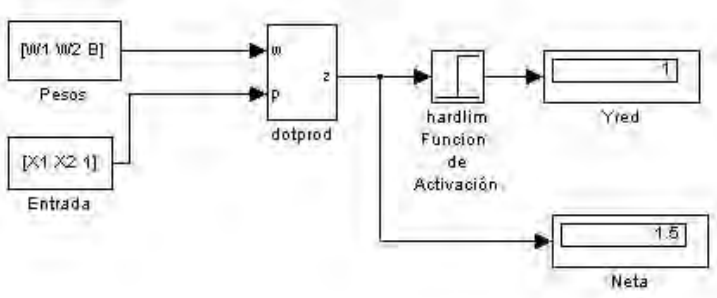


Fig. 2.12 Simulación de un Perceptron de dos entradas en Simulink

Con el diagrama construido es posible simular otros tipos de redes neuronales sencillas, por ejemplo, cambiando la función de activación a lineal, sigmoidal o tangente sigmoidal.

5.2 Solución de la función lógica and con un perceptron

Como vimos anteriormente, el Perceptron es una red neuronal artificial que está en capacidad de realizar separaciones lineales; veamos entonces, como se puede solucionar un problema de estos con la ayuda de la herramienta de redes neuronales de MATLAB®.

La función lógica AND se define como:

x_1	x_2	y
0	0	0
0	1	0
1	0	0
1	1	1

Sigamos cuidadosamente los siguientes pasos y podremos resolver este problema usando MATLAB®:

Definición del problema

Definir el problema que una red neuronal va a resolver es proporcionarle a la misma un conjunto de parejas ordenadas de entradas y salidas para que la red "*aprenda"* los llamados patrones de entrenamiento o aprendizaje de la red. En MATLAB® esto se hace definiendo dos matrices una para las entradas y otra para las salidas donde cada patrón de aprendizaje se define por columnas, los comandos necesarios para lo anterior son:

```
>> % Definición de la función lógica AND
>> X=[0 0 1 1 ;
      0 1 0 1 ] ;
>> D=[0 0 0 1] ;

>>%Para ver la gráfica de estos patrones se usa el comando
plotpv

>> plotpv(X,D)
```

Al ejecutar esta serie de comandos usted podrá obtener como resultado algo similar a lo presentado en la figura 2.13.

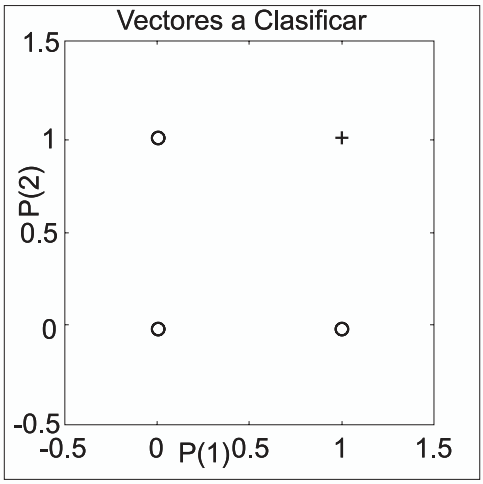


Fig. 2.13 Patrones a clasificar

Como se puede observar en la figura 2.13, MATLAB® grafica los puntos dados en el vector *X* y le asigna un símbolo para la clasificación dependiendo de la salida deseada, en este caso:

- Para salida deseada cero (0): **o**
- Para salida deseada uno (1): **+**

Inicialización de la red neuronal

Con base en los patrones de entrenamiento que definen el problema a resolver, procedemos a inicializar la red neuronal. Para el caso del Perceptron utilizamos la función de MATLAB® *initp.* En el comando de ayuda de MATLAB® *(help)* podemos encontrar una descripción completa de cada una de las funciones utilizadas.

```
% Generación de un perceptron

>> red = newp([0 1;0 1],1)
```

donde,

red	:	Objeto donde se va almacenar la red creada por el MATLAB®
[0 1;0 1]	:	Rango del valor de la entrada de la red neuronal, el número de filas de esta matriz lo utilizará MATLAB® para definir el número de entradas que tiene la red neuronal.
1	:	Número de neuronas que tiene la red neuronal en la capa de salida

Ahora procedemos a generar una matriz de pesos iniciales para la red. Este paso no es indispensable, pero permite generar un Perceptron con una superficie de separación conocida.

```
>> red.iw{1,1}=[1 1];
>> red.b{1}=0.5;
>> Pesos=red.iw{1,1};
>> Bias=red.b{1};

>> % Comando para graficar la línea que el Perceptron define para separar las regiones
>> plotpc(Pesos,Bias)
```

Con el último comando ejecutado, adicionamos la recta de separación de las regiones de clasificación que se tiene en un instante determinado de acuerdo con los pesos de la red en ese momento. Tal como lo mostramos en la figura 2.14, la clasificación no es correcta, pues los parámetros que definen la recta fueron asignados de manera arbitraria.

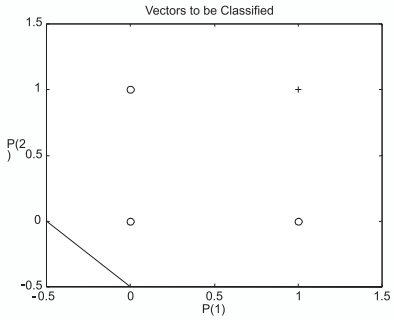


Fig. 2.14 Patrones a clasificar y la recta clasificadora inicial

Entrenamiento de la red

El entrenamiento de la red lo realizamos con el comando *train* el cual se implementa con la regla de aprendizaje tipo Perceptron. En MATLAB® el entrenamiento se realiza ejecutando el siguiente comando:

```
>> red = train(red,X,D)
```

donde,

red	:	Red a ser entrenada por el MATLAB®.
X	:	Vector con los patrones de entrada para el aprendizaje.
D	:	Vector con los patrones de salida deseada para el aprendizaje.

Cuando ejecutamos este comando en MATLAB®, iniciamos el entrenamiento y éste grafica la evolución del error al progresar las iteraciones, tal como se muestra en la figura 2.15.

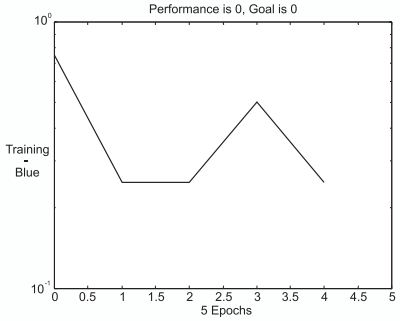


Fig. 2.15 Evolución del error durante el entrenamiento de la red

Una vez entrenada la red, ejecutamos los siguientes comandos para visualizar en la figura 2.16, la recta que separa las dos clases de salidas, comprobando con ello que la red ha realizado correctamente la tarea.

```
>> figure;
>> Pesos=red.iw{1,1};
>> Bias=red.b{1};
>> plotpv (X,D)
>> plotpc (Pesos, Bias)
```

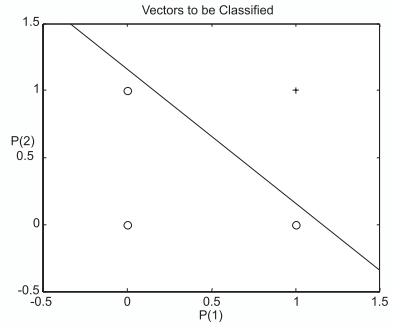


Fig. 2.16 Patrones a clasificar y la recta clasificadora final

Validación de la red

Luego de tener una red entrenada, procedemos a validar si el comportamiento de la misma es correcto o no, usando el comando *sim*. Podemos corroborar que el comportamiento de la red es el adecuado y dar por finalizado el entrenamiento.

```
>> in_prueba=[0;0];            % Patrón de prueba
>> % Prueba de la red ante el patrón de prueba, W,b son los pesos
>>% y el bias de la red entrenada

>> a = sim(red, in_prueba)
>>a =
>> 0

>> % Otro patón de prueba
>> in_prueba=[1;1];
>> a = sim(red, in_prueba)
>>a =
>>1
```

Exportando la red neuronal a simulink

En la sección anterior entrenamos y validamos nuestra primera red neuronal artificial, en particular un Perceptron. MATLAB® almacena esta red como un bloque funcional que puede ser exportado al ambiente de trabajo de ***simulink*** para verificar el comportamiento de la misma de una manera completamente gráfica. Para llevar a cabo esta tarea usamos el comando *gensim* y la herramienta nos responde en su ambiente gráfico con el diagrama de bloques de la figura 2.17.

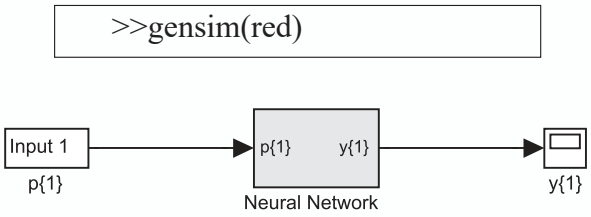


Fig. 2.17 Red neuronal generada en el Simulink

MATLAB® nos entrega el diagrama de la figura 2.17, pero lo podemos modificar como en la figura 2.18, separando las dos entradas y, con este nuevo esquema, procedemos a simular el comportamiento de la red neuronal diseñada. El valor de los bloques de entrada se puede modificar para verificar el comportamiento de la red ante las diferentes combinaciones de la función lógica AND.

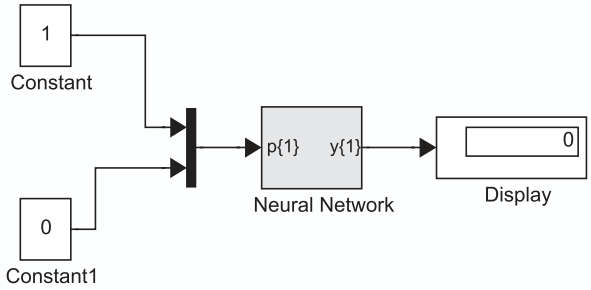


Fig. 2.18 Diagrama para verificar el comportamiento de la red neuronal

Solución de la función lógica and con uv-srna

En el Grupo de Percepción y Sistemas Inteligentes de la Escuela de Ingeniería Eléctrica y Electrónica, se desarrolló la herramienta UV-SRNA, para el diseño y simulación de Redes Neuronales Artificiales. Esta aplicación junto con su herramienta se ayuda se entrega en el CD disponible con este libro.

Para iniciar el proceso de aprendizaje con UV-SRNA, debemos crear un

archivo texto con los patrones que se desean usar para el proceso de entrenamiento que se almacenan con la extensión **.pat*, que para el caso de la función lógica **AND,** el archivo puede tener la estructura que definimos en la Tabla 2.1.

Tabla No. 2.2 Codificación de los patrones de entrenamiento para la función lógica AND

Datos en el archivo	Significado
4	Número de patrones de entrenamiento
2	Número de entradas de cada patrón
1	Número de salidas de cada patrón
0 0 0	Patrón No. 1
0 1 0	Patrón No. 2
1 0 0	Patrón No. 3
1 1 1	Patrón No. 4

Una vez creado el archivo, lo leemos desde UV-SRNA usando en la interfaz gráfica la opción *leer patrones* del menú de *archivo.*

Con los patrones en el ambiente de UV-SRNA procedemos a inicializar la red, para esto solo es necesario oprimir el botón de *inicializar*, de la interfaz gráfica.

Una vez inicializada la red procedemos a llevar a cabo el entrenamiento. Esto se logra oprimiendo el botón *entrenar*, e inmediatamente observamos la evolución del error con una clara tendencia decreciente, hasta que converge a cero.

Para finalizar, verifiquemos si el entrenamiento de la red fue adecuado, utilizando en la interfaz de usuario la opción de validación, que se puede hacer directamente por teclado o por archivo previamente construido.

Clasificador lineal con uv-srna

En el ejercicio anterior construimos un Perceptron completamente y lo aplicamos en un problema de clasificación de patrones, ahora, ejecutemos una demostración que nos permite resolver problemas de clasificación lineal en un conjunto de puntos en un plano. Como ejemplo típico, nuevamente usaremos el problema de la función lógica AND. Para ello debemos ejecutar la demostración que está dentro de las “demos” del Perceptron de UV-SRNA y cargar el conjunto de patrones disponible para la función AND.

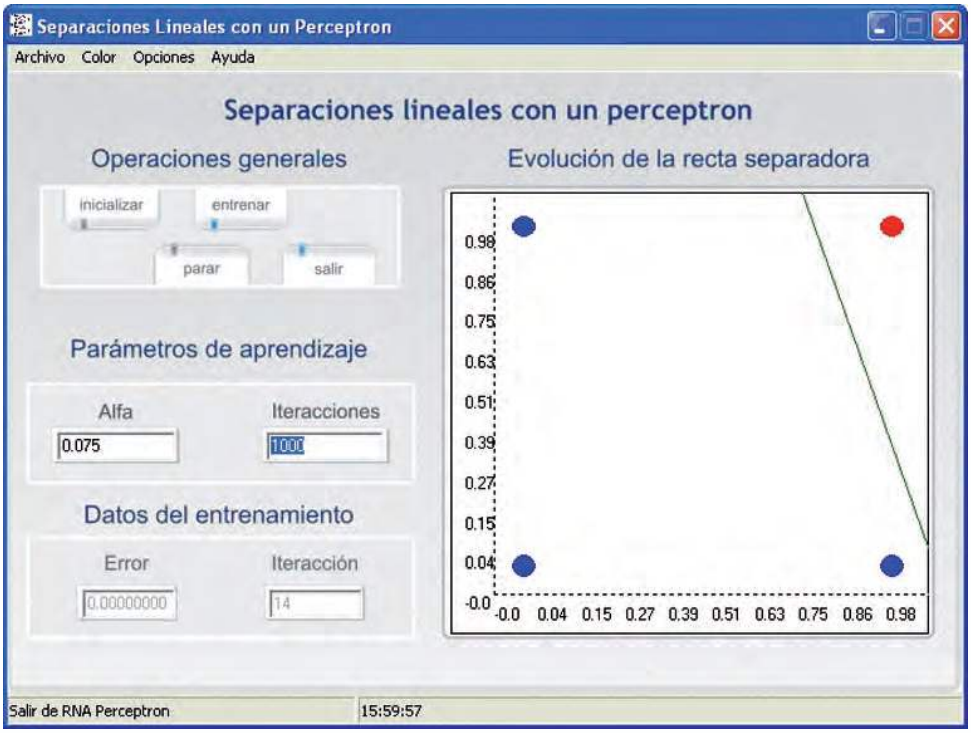


Fig. 2.19 Interfaz gráfica con el usuario del módulo de separaciones lineales de UVSRNA

Al oprimir el botón "inicializar" observemos la recta generada por el Perceptron no clasifica correctamente los datos, ver figura 2.19, pues la asignación de pesos de la red en esta fase es aleatoria. Cuando oprimimos el botón "entrenar", la recta va cambiando a medida que transcurre el proceso de aprendizaje hasta que se consiga separar adecuadamente los datos.

Un ejercicio interesante, es verificar el efecto que tiene el valor de α en la velocidad de convergencia de la red. Evaluemos qué efectos tiene el realizar este experimento con valores de α igual a 0.07 y 0.007. Notaremos que en el segundo caso el proceso de aprendizaje es mucho más lento.

Reconocimiento de caracteres usando el perceptron

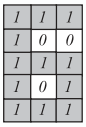
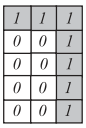
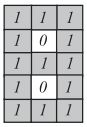
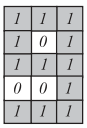
El objetivo de este proyecto es el de entrenar un Perceptron en MATLAB® que esté en capacidad de identificar los números del 0 al 9 donde cada número se define con una matriz de dimensión 5x3. Por ejemplo el número 2 sería representado así,

1	1	1
0	0	1
1	1	1
1	0	0
1	1	1

Para resolver el problema, verifiquemos todos los patrones correspondientes a cada número y asignémosle el respectivo valor de salida. En nuestro caso, definiremos por patrón un vector de entrada de 15 elementos y un vector de salida de 4 elementos, correspondientes al número binario equivalente. El vector de patrones de entrada queda definido como se muestra en la tabla 2.3, donde cada número decimal se codifica con un vector de 15 bits.

Tabla 2.3 Vector de Patrones correspondiente a los caracteres decimales

Caracter	**Vector de Patrones**														
	B_1	**B_2**	**B_3**	**B_4**	**B_5**	**B_6**	**B_7**	**B_8**	**B_9**	**B_{10}**	**B_{11}**	**B_{12}**	**B_{13}**	**B_{14}**	**B_{15}**
1 1 1 1 0 1 1 0 1 1 0 1 1 1 1	1	1	1	1	0	1	1	0	1	1	0	1	1	1	1
0 0 1 0 0 1 0 0 1 0 0 1 0 0 1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1
1 1 1 0 0 1 1 1 1 1 0 0 1 1 1	1	1	1	0	0	1	1	1	1	1	0	0	1	1	1
1 1 1 0 0 1 0 1 1 0 0 1 1 1 1	1	1	1	0	0	1	0	1	1	0	0	1	1	1	1
1 0 1 1 0 1 1 1 1 0 0 1 0 0 1	1	0	1	1	0	1	1	1	1	0	0	1	0	0	1
1 1 1 1 0 0 1 1 1 0 0 1 1 1 1	1	1	1	1	0	0	1	1	1	0	0	1	1	1	1

	1	1	1	1	0	0	1	1	1	1	0	1	1	1	1
	1	1	1	0	0	1	0	0	1	0	0	1	0	0	1
	1	1	1	1	0	1	1	1	1	1	0	1	1	1	1
	1	1	1	1	0	1	1	1	1	0	0	1	1	1	1

La salida deseada de entrenamiento de la red la representaremos con el código binario del número que le presentamos a la entrada. Es decir, si a la red le llegan como entrada los 15 bits correspondientes al número 0, esperamos que la red genere el código binario del cero expresado en 4 bits (0 0 0 0); o en otro caso, si le llegan como entrada los 15 bits correspondientes al número 1, ella generará el código binario del uno expresado en 4 bits (0 0 0 1) y así con el resto de patrones.

Teniendo en cuenta lo anterior, presentamos en la tabla 24, los patrones de entrenamiento para los números decimales y, en seguida, el código en MATLAB® para entrenar la red neuronal.

Tabla 2.4 Codificación de los patrones de entrenamiento para el reconocimiento de caracteres decimales en MATLAB®

	Patrón de Entradas X															Patrón Salidas D			
	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8	B_9	$\mathbf{B_{10}}$	$\mathbf{B_{11}}$	$\mathbf{B_{12}}$	$\mathbf{B_{13}}$	$\mathbf{B_{14}}$	$\mathbf{B_{15}}$	$\mathbf{2^3}$	$\mathbf{2^2}$	$\mathbf{2^1}$	$\mathbf{2^0}$
0	1	1	1	1	0	1	1	0	1	1	0	1	1	1	1	0	0	0	0
1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	0	1
2	1	1	1	0	0	1	1	1	1	1	0	0	1	1	1	0	0	1	0
3	1	1	1	0	0	1	0	1	1	0	0	1	1	1	1	0	0	1	1
4	1	0	1	1	0	1	1	1	1	0	0	1	0	0	1	0	1	0	0
5	1	1	1	1	0	0	1	1	1	0	0	1	1	1	1	0	1	0	1
6	1	1	1	1	0	0	1	1	1	1	0	1	1	1	1	0	1	1	0
7	1	1	1	0	0	1	0	0	1	0	0	1	0	0	1	0	1	1	1
8	1	1	1	1	0	1	1	1	1	1	0	1	1	1	1	1	0	0	0
9	1	1	1	1	0	1	1	1	1	0	0	1	1	1	1	1	0	0	1

```
% Rec_Carac.m
% Programa que reconoce los caracteres decimales

% Patrones de entrada con los valores que debe tener X
Xaux=[1 1 1 1 0 1 1 0 1 1 0 1 1 1 1;
      0 1 0 0 1 0 0 1 0 0 1 0 0 1 0;
      1 1 1 0 0 1 1 1 1 1 0 0 1 1 1;
      1 1 1 0 0 1 1 1 1 0 0 1 1 1 1;
      1 0 1 1 0 1 1 1 1 0 0 1 0 0 1;
      1 1 1 1 0 0 1 1 1 0 0 1 1 1 1;
      1 0 0 1 0 0 1 1 1 1 0 1 1 1 1;
      1 1 1 0 0 1 0 0 1 0 0 1 0 0 1;
      1 1 1 1 0 1 1 1 1 1 0 1 1 1 1;
      1 1 1 1 0 1 1 1 1 1 0 1 0 0 1 ];
X=Xaux';
% Salida deseada de la red neuronal
Daux=[0 0 0 0;
      0 0 0 1;
      0 0 1 0;
      0 0 1 1;
      0 1 0 0;
      0 1 0 1;
      0 1 1 0;
      0 1 1 1;
      1 0 0 0;
      1 0 0 1 ];
 D=Daux';
red=newp(minmax(X),4);                % Definición de la red
red.iw{1,1}=rand(4,15);               % Definición aleatoria de los
                                      pesos
red.b{1}=rand(4,1);
red.trainParam.show=1;                % Evolución de la red en
                                      cada iteración
disp('los pesos inciales son:')       % Pesos asignados por la
                                      función random
Pesos=red.iw{1,1}
Bias=red.b{1}
pause(2);
red = train(red,X,D)                  % Entrenamiento de la red
disp ('para validar la red, digite el % Validación de la red, se pide
 vector de patrones de entrada')      digitar un patrón
                                      % para verificar el funcionamiento
                                      de la red
```

```
disp('Número de 10 binarios entre [ ]')
X1=input('X1=')              % Patrón de entrada
Y = sim(red, X1');           % Simulación de la red

disp ('el número resultante, en binario, leído de arriba para abajo es:')
Y
```

Como ejemplo presentemos dos casos particulares, en donde en el primero introducimos como valor de validación el número 7 y, en el segundo, el número 9. Para cada caso presentamos la gráfica de evolución en el entrenamiento y el patrón introducido para el proceso de validación.

Primera ejecución

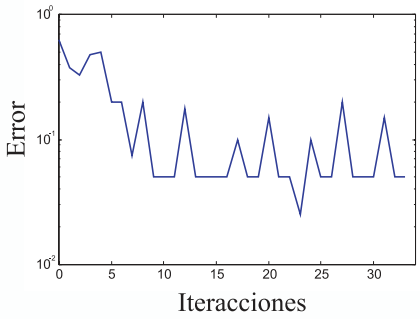


Fig. 2.20 Evolución del error de entrenamiento

Patrón de Entrada Para Validación	Vector de Entrada X1	Valor de Salida Y
1 1 1 0 0 1 0 0 1 0 0 1 0 0 1	**[1 1 1 0 0 1 0 0 1 0 0 1 0 0 1]**	**[0 1 1 1] → 7**

Segunda ejecución:

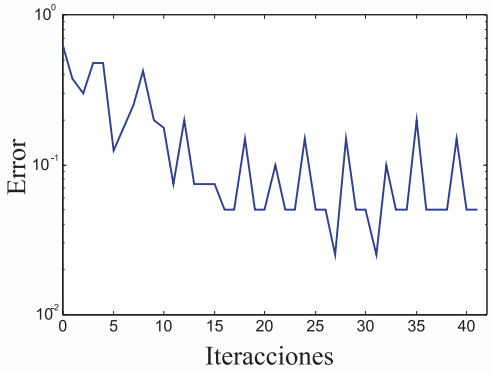
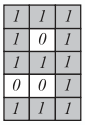


Fig. 2.21 Evolución del error de entrenamiento

Patrón de Entrada Para Validación	Vector de Entrada X1	Valor de Salida Y
	**[1 1 1 1 0 1 1 1 1 0 0 1 1 1 1]**	**[1 0 0 1]→ 9**

Teniendo la red entrenada, se puede exportar a la herramienta de simulación Simulink® asociado a MATLAB®, para verificar su comportamiento de una manera completamente gráfica como se observa en la figura 2.22.

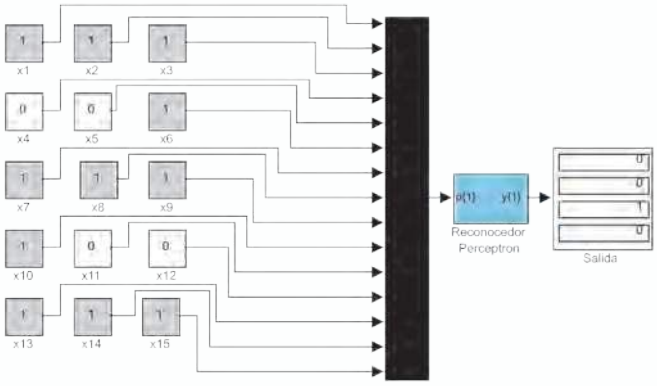


Fig. 2.22 Reconocedor de Caracteres Implementado en Simulink®

Reconocimiento de caracteres con uvsrna

En UV-SRNA se ha diseñado un módulo especial para verificar la capacidad de un Perceptron en el reconocimiento de caracteres, que se encuentra dentro de las demostraciones del Perceptron, cuya interfaz con el usuario se muestra en la figura 2.23

Fig. 2.23 Interfaz de Usuario del Módulo de Reconocimiento de Caracteres de UVSRNA

Reconocimiento de los números del 0-9 con uvsrna

Con esta aplicación podemos realizar el reconocimiento de los números del 0 al 9. En primera instancia creamos un archivo texto con los patrones que se desean usar para el proceso de entrenamiento, que debe almacenarse con la extensión *.pat., siguiendo el formato que se presenta en la tabla 2.5.

Tabla No.2.5 Ejemplo de codificación de los patrones de entrenamiento para el reconocimiento de números

Datos en el archivo	Significado
10	Número de patrones de entrenamiento
35	Número de entradas de cada patrón
4	Número de salidas de cada patrón
1 1 1 1 1 1 0 0 0 1 1 0 0 0 1 1 0 0 0 1 1 0 0 0 1 1 0 0 0 1 1 1 1 1 1 0 0 0 0	Patrón No. 1. Que corresponde al número cero
0 0 1 0 0 0 1 1 0 0 1 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 1	Patrón No. 2 Que corresponde al número uno
1 1 1 1 1 0 0 0 0 1 0 0 0 0 1 1 1 1 1 1 1 0 0 0 0 1 0 0 0 0 1 1 1 1 1 0 0 1 0	Patrón No. 3 Que corresponde al número dos

Una vez creado el archivo, lo cargamos desde UV-SRNA usando la opción "*leer patrones*", para luego inicializar la red y proceder a su entrenamiento. Emerge la ventana de la figura 2.24 que nos muestra como evoluciona el error y, muy seguramente, alcanzaremos un valor igual a cero, considerando así que el entrenamiento se ha finalizado.

Fig. 2.24 Evolución del error de entrenamiento en UVSRNA

Para validar el entrenamiento de la red, el módulo de reconocimiento de caracteres de UVSRNA dispone de una matriz 7x5 para dibujar los caracteres que vamos a reconocer. Una vez dibujado el caracter, presionamos el botón *"validar"* y la red toma esta entrada y esperamos que la salida, en binario, corresponda al dígito seleccionado.

Filtro adaptativo usando una red adaline

En la figura 2.25 vemos un problema clásico en procesamiento digital de señales, donde una fuente de ruido interfiere con la señal que contiene la información válida para la aplicación. La solución implica el diseño de un filtro para eliminar la señal no deseada. La red ADALINE puede usarse como filtro y por su capacidad de aprendizaje, puede adaptarse progresivamente a medida que va recibiendo la señal y de esa manera elimina el ruido indeseado y nos entrega una señal limpia, sin interferencias.

Fig. 2.25 Figura Esquema de Filtrado adaptativo con una red adaline

La arquitectura general de la red *ADALINE* utilizada para filtrar esta señal, figura 2.26, toma la *k-ésima* muestra de la señal contaminada y *N* muestras anteriores.

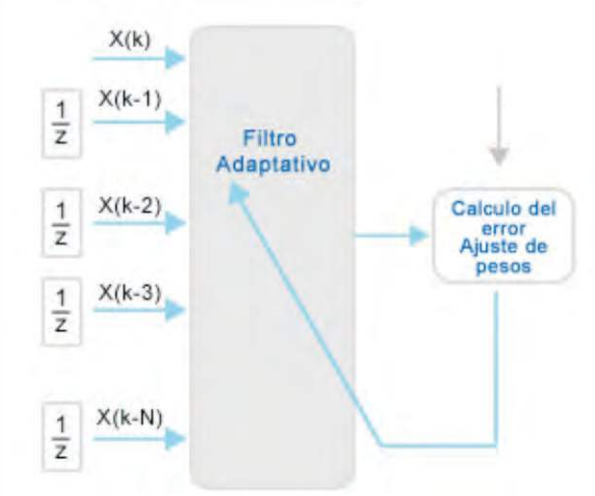


Fig. 2.26 Esquema de un Filtro Adaptativo

En la figura 2.27, tenemos una señal senoidal contaminada con una señal de alta frecuencia, pretendemos con la red *ADALINE*, eliminar la señal de alta frecuencia. Para ello programamos en *MATLAB*® una red que tiene como entradas, la muestra *k-ésima* y cinco retardos más de la señal. Como el filtro es de naturaleza adaptativa, a medida que la señal se va presentando al mismo tiempo, éste va modificando sus pesos y la convergencia va mejorando paulatinamente, en las figura 2.28 se puede apreciar esta convergencia visualizando el desempeño del filtro durante la sintonización.

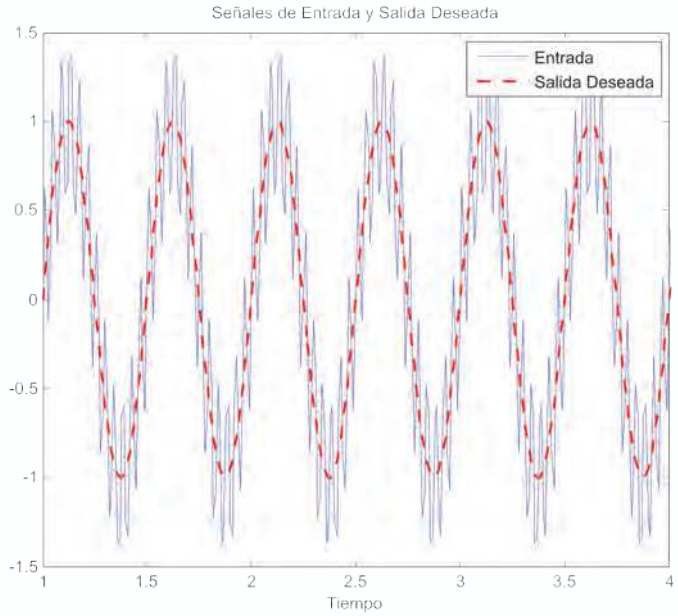


Fig. 2.27 Señal Contaminada con Ruido

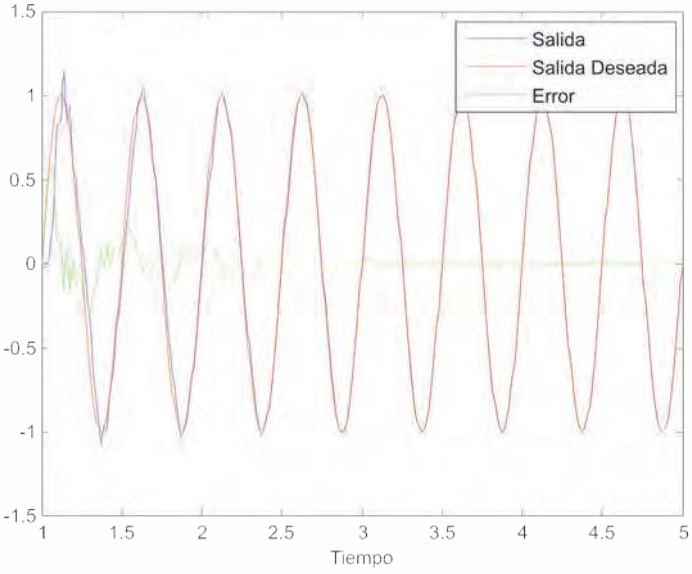


Fig. 2.28 Señal Filtrada y Evolución del Error

El código en MATLAB® que permite diseñar el filtro para la anterior aplicación es el siguiente:

```
time = 1:0.01:5;
X = sin(2*pi*2*time)+0.5*sin(2*pi*24*time);
P = con2seq(X);
Taux = sin(2*pi*2*time);
T = con2seq(Taux);
Figure
plot(time, cat(2,P{:}),time,cat(2,T{:}),'--')
title('Señales de Entrada y Salida Deseada')
xlabel('Tiempo')
legend({'Entrada','Salida Deseada'})
% Creamos una red ADALINE con 5 retardos
net = newlin([-2 2],1,[0 1 2 3 4 5],0.1);
net.biasConnect=0; % El peso de tendencia o umbral no se usa para
esta aplicación.

[net,Y,E,Pf]=adapt(net,P,T);
figure
plot(time,cat(2,Y{:}),'b', ...
   time,cat(2,T{:}),'r', ...
   time,cat(2,E{:}),'g')
legend ({'Salida','Salida Deseada','Error'})
```

Filtrado de señales biomédicas

Normalmente las señales biomédicas que provienen del cuerpo humano se encuentran contaminadas por una señal de ruido de 60Hz, producto de la inducción que hace el mismo cuerpo ante la presencia de los campos magnéticos producidos por las redes de distribución eléctrica. Por lo que es necesario un procesamiento automático que nos permita filtrar la señal de 60 Hz dejando la señal biomédica sin contaminación alguna, tal como vemos en la figura 2.29.

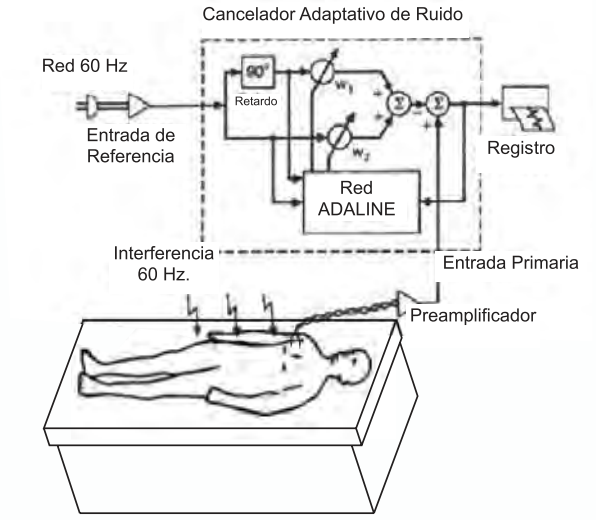


Fig. 2.29 Filtrado de una Señal biomédica

La aplicación que diseñamos y cuyo programa MATLAB®, *Filtrado_ Biomedico.m*, se encuentra en el CD que se adjunta a este libro, se utiliza para filtrar la señal proveniente de un electro cardiógrafo (ECG). En la figura 2.30, presentamos el esquema del filtro diseñado que elimina la componente inducida de 60 Hz., y en la figura 2.31 observamos las señales de entrada, salida deseada y error.

Fig. 2.30 Esquema del Filtrado planteado

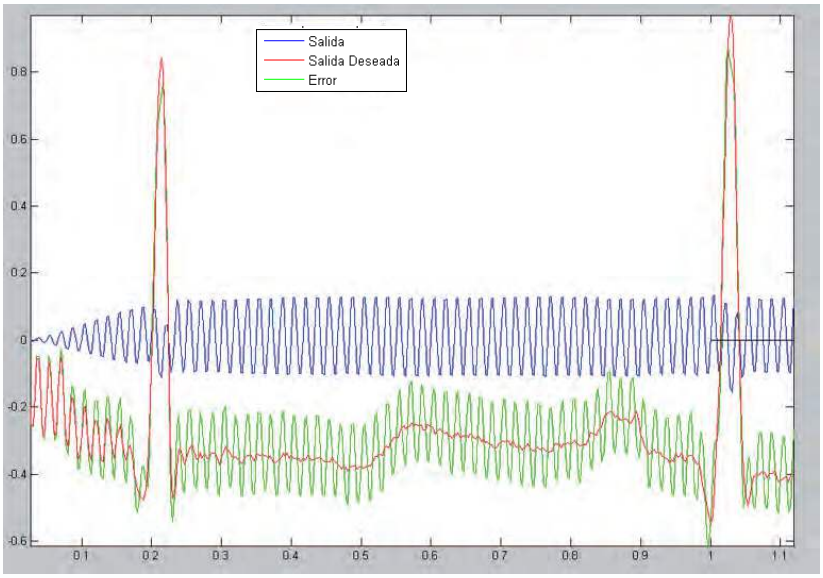


Fig. 2.31 Señales en el proceso de filtrado de una señal ECG

Filtrado de señales de voz

Cuando grabamos un concierto es imposible separar la música del ruido que hacen los espectadores, como resultado tenemos una señal de la voz del cantante contaminada con las voces y los aplausos de los asistentes. Esta fuente de ruido la podemos conocer, pero no sabemos como se ve afectada la señal de la voz del intérprete que queremos procesar. En la figura 2.32, presentamos un esquema con el que podemos eliminar el efecto del ruido en nuestra grabación utilizando un filtro adaptativo dise-ñado con una red ADALINE. Vale la pena destacar que la señal de error será la señal de la voz filtrada pues en este caso, la red ADALINE se encargará de aprender como la fuente de ruido afecto la voz original; cuando la red aprenda lo anterior, al restar de la señal de voz contaminada la salida de la red, tendremos la voz filtrada tal como lo observamos en la figura 2.33. La aplicación la diseña-mos para ser ejecutada en MATLAB®, *Filtrado_voz.m*, y se encuentra en el CD que se adjunta a este libro.

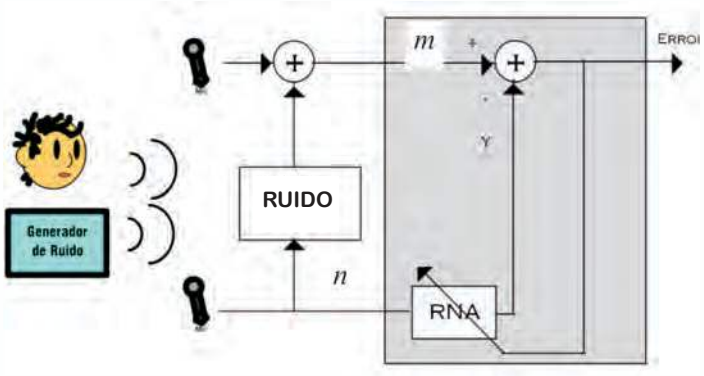


Fig. 2.32 Esquema para filtrar una señal de voz con una red ADALINE

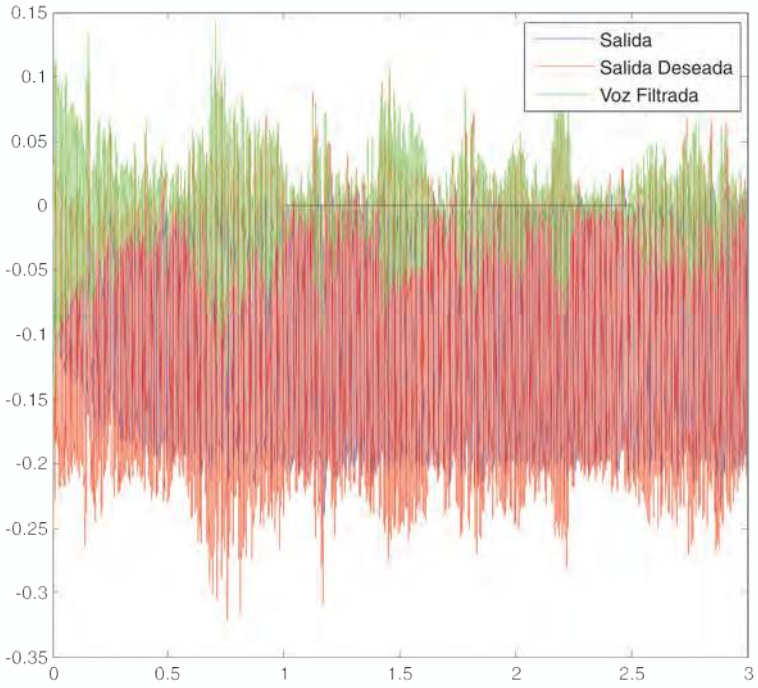


Fig. 2.33 Señales en el proceso de filtrado de voz

Proyectos propuestos

1. Realice la siguiente clasificación usando un Perceptron y con ayuda del *toolbox de redes neuronales* del MATLAB®.

X1	X2	D
-0.5	-1.0	0
1.0	1.0	1
1.0	0.5	1
-1.0	-0.5	0
-1.0	-1.0	0
0.5	1.0	1

2. Repita el ejercicio anterior con UV-SRNA.
3. Teniendo los siguientes puntos en el plano realice el procedimiento necesario tanto en MATLAB® como en UVSRNA para resolver la clasificación indicada. La clase A la puede codificar como 0 y la clase B la puede codificar como 1.

X	Y	Familia
-1	-1	A
-0.5	-0.5	A
-1	0	A
0	-1	A
1	1	B
0.5	0.5	B
1	0	B
0	1	B

4. Cree un programa en MATLAB® que implemente el algoritmo de entrenamiento tipo Perceptron para resolver el problema de la función lógica OR.
5. Intente en MATLAB® y en UVSRNA solucionar el problema de la función lógica XOR. ¿Qué conclusiones obtiene de los resultados alcanzados?.
6. Entrenar una red neuronal tipo Perceptron tanto en MATLAB® como en UVSRNA que sirva para reconocer las vocales.
7. Entrenar una red neuronal tipo Perceptron tanto en MATLAB® como en UVSRNA que sirva para reconocer el código hexadecimal.
8. Entrenar una red neuronal tipo Perceptron tanto en MATLAB® como en UVSRNA que sirva para reconocer las letras del nombre de algún integrante de su familia.
9. Entrenar una red neuronal tipo Perceptron tanto en MATLAB® como en UVSRNA que sirva para reconocer cuatro figuras geométricas sencillas (un cuadrado, un triángulo, un rectángulo y un rombo).

PÁGINA EN BLANCO
EN LA EDICIÓN IMPRESA

Capítulo 3

PERCEPTRON MULTICAPA Y ALGORITMO BACKPROPAGATION

Introducción

Debido a la imposibilidad de solucionar problemas de clasificación no lineales que presenta el Perceptron propuesto por Rosenblatt, y redes algo más evolucionadas como el Adaline, el interés inicial que las redes neuronales artificiales habían despertado, decayó fuertemente y sólo quedaron unos pocos investigadores trabajando en el desarrollo de arquitecturas y algoritmos de aprendizaje capaces de solucionar problemas de alta complejidad. Esta situación fue ampliamente divulgada por Misky y Papert en su libro *Perceptrons*; lo que significó prácticamente el olvido científico de la propuesta de Rosenblatt.

Ronsenblatt, sin embargo, ya intuía la manera de solucionarlo, el autor del *Perceptron*, percibía que si incluía una capa de neuronas entre las capas de entrada y salida, podía garantizar que la red neuronal artificial sería capaz de solucionar problemas de este tipo y mucho más complejos como los que se presentarán a lo largo de este capítulo.

¿Pero cuál fue la mayor dificultad para poner en práctica la nueva propuesta? La modificación de los pesos sinápticos asociados a la capa de salida, se hace con el error entre la salida esperada y la salida de la red. Pero, ¿cómo calcular el error de la capa intermedia para modificar los pesos sinápticos de esta nueva capa, que llamaremos capa oculta? Justamente este interrogante quedó por varios años sin respuesta y fue la razón fundamental por la que las redes neuronales artificiales quedaran casi en el olvido.

A mediados de la década de los setenta, Paul Werbos en su tesis doctoral propone el Algoritmo Backpropagation, que permite entrenar al Perceptron

multicapa y posibilita su aplicación en la solución de una gran variedad de problemas de alta complejidad como lo veremos a lo largo de este capítulo.

Paul J. Werbos
Científico norteamericano que obtuvo su doctorado en la Universidad de Harvard en 1974, reconocido en el mundo de las redes neuronales artificiales, porque en su Tesis fue el primero en describir el algoritmo *Backpropagation* para el entrenamiento del Perceptron multicapa. Una ampliación de esta información puede encontrarse en su libro *The Roots of Backpropagation.* Por este aporte, fue galardonado por la IEEE con el *Neural Network Pioneer Award.* Actualmente, trabaja para la *National Science Foundation*.

Arquitectura general de un perceptron multicapa

En la figura 3.1 presentamos la estructura del Perceptron Multicapa (MLP de sus siglas en inglés *Multi Layer Perceptron*), que a diferencia del Perceptron y del Adaline, posee al menos tres niveles de neuronas, el primero es el de entrada, luego viene un nivel o capa oculta y, finalmente, el nivel o capa de salida. Podemos proponer más de una capa oculta, pero en general no lo recomendamos pues se aumenta fuertemente la complejidad computacional del algoritmo de aprendizaje y para la gran mayoría de las aplicaciones prácticas que propondremos en este libro, es suficiente un MLP con una única capa oculta.

En las redes neuronales artificiales, el término conectividad se refiere a la forma como una neurona de una capa cualquiera está interconectada con las neuronas de la capa previa y la siguiente. Para el MLP, la conectividad es total porque si tomamos una neurona del nivel de entrada, ésta estará conectada con todas las neuronas de la capa oculta siguiente, una neurona de la capa oculta tendrá conexión con todas las neuronas de la capa anterior y de la capa siguiente. Para la capa de salida, sus neuronas estarán conectadas con todas las neuronas de la capa oculta previa, para mayor claridad observemos la figura 3.1. Por lo general, se le implementan unidades de tendencia o umbral con el objetivo de hacer que la superficie de separación no se quede anclada en el origen del espacio *n*-dimensional en donde se esté realizando la clasificación.

La función de activación que utilizan las neuronas de una red MLP suele ser lineal o en la mayoría de los casos sigmoidal.

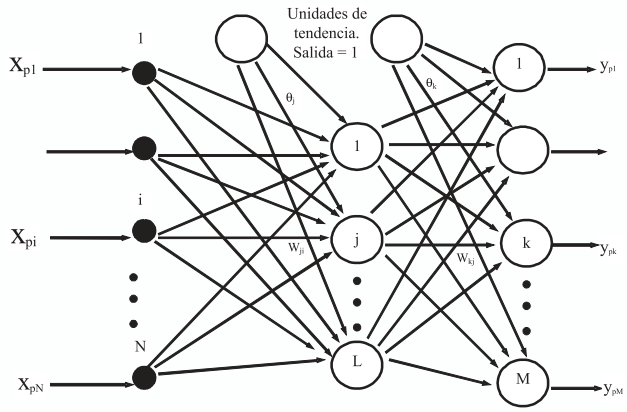


Fig. 3.1 Arquitectura General de un MLP

David Rumelhart

A este científico norteamericano se debe en gran parte, el resurgimiento de las Redes Neuronales Artificiales, cuando en 1986, publica su libro "*Parallel distributed processing: Explorations in the microstructure of cognition"*, en el cual difunde ampliamente el algoritmo de entrenamiento de un *Perceptron* multicapa. Con este algoritmo se da respuesta al interrogante que Papert y Minsky, le plantearon a Rosenblatt, de cómo entrenar la capa oculta del MLP, si no se tenían datos esperados de salida para esta capa.

Entrenamiento de un MLP

Muchos autores traducen al español el nombre de este algoritmo, quizá la mejor acepción para el término Backpropagation es la de *retropropagación,* pues nos da una idea conceptual de la esencia del algoritmo, justamente de propagar el error de la capa de salida hacia las capas ocultas; sin embargo, el nombre anglosajón se ha aceptado en la mayoría de las publicaciones en español y lo usaremos en este libro.

En el capítulo 2, vimos que uno de los principales inconvenientes que tiene el Perceptron es su incapacidad para separar regiones que no son linealmente separables y lo ilustramos al aplicarlo en la solución de un problema simple como la separación de las salidas de una función lógica XOR. Sin embargo, Rosenblatt ya intuía que un Perceptron multicapa sí podía solucionar este problema pues, de esta manera, se podían obtener regiones de clasificación mucho más complejas. Sin embargo, persistían algunas interrogantes sin respuesta ¿Cómo entrenar un Perceptron multicapa? ¿Cómo

evaluar el error en las capas ocultas si no hay un valor deseado conocido para las salidas de estas capas?

Una respuesta a estos interrogantes la planteó formalmente Werbos, cuando formuló el algoritmo de aprendizaje *Backpropagation*. Algoritmo que debe su amplia difusión y uso a David Rumelhart. Como el error de la capa de salida es el único que puede calcularse de forma exacta, el algoritmo propone propagar hacia atrás este error para estimar el error en las salidas de las neuronas de las capas ocultas, con el fin modificar los pesos sinápticos de estas neuronas.

Antes de profundizar matemáticamente en el algoritmo de aprendizaje Backpropagation, revisemos su funcionamiento de manera conceptual:

- Seleccionamos un conjunto de patrones claramente representativos del problema a solucionar, con los cuales vamos a entrenar la red. Esta fase es fundamental pues dependiendo de la calidad de los datos utilizados para el entrenamiento, será la calidad de aprendizaje de la red.
- Aplicamos un vector de entrada a la red y calculamos la salida de las neuronas ocultas, propagamos estos valores hasta calcular la salida final de la red.
- Calculamos el error entre el valor deseado y la salida de la red.
- Propagamos el error hacia atrás, es decir, estimamos el error en la capa oculta con base en el error de la capa de salida.
- Modificamos los pesos de la capa de salida y de las capas ocultas con base en una estimación del cambio de los pesos Δw en cada una de las capas que a su vez, depende del cálculo del error de la capa de salida y de la estimación del error en las capas ocultas realizado en los pasos anteriores.
- Verificamos la condición de parada del algoritmo ya sea por que el error calculado en la salida es inferior al impuesto por el problema bajo análisis o por ejemplo, porque hemos superado un número determinado de iteraciones, en cuyo caso consideraremos que es necesario hacer ajustes al diseño de la red, pues la solución no converge, o simplemente que debemos ampliar el número máximo de iteraciones a realizar
- En caso de no cumplir con ninguna de las condiciones de parada, volvemos a presentarle a la red un patrón de entrenamiento.

Nomenclatura del algoritmo backpropagation

Antes de formular matemáticamente el algoritmo, con la ayuda de la figura 3.1, definamos la notación que seguiremos a lo largo de este capítulo.

$\boldsymbol{x_p}$	Patrón o vector de entrada
x_{pi}	Entrada *i-ésima* del vector de entrada x_p
N	Dimensión del vector de entrada
P	Número de ejemplos, vectores de entrada y salidas diferentes.
L	Número de neuronas de la capa oculta: h
M	Número de neuronas de la capa de salida, dimensión del vector de salida
w_{ji}^{h}	Peso de interconexión entre la neurona *i-ésima* de la entrada y la *j-ésima* de la capa oculta.
θ_j^{h}	Término de tendencia de la neurona *j-ésima* de la capa oculta.
$Neta_{pj}^{h}$	Entrada neta de la *j-ésima* neurona de la capa oculta
i_{pj}	Salida de la *j-ésima* neurona de la capa oculta
f_j^{h}	Función de activación de la *j-ésima* unidad oculta
w_{kj}^{o}	Peso de interconexión entre la *j-ésima* neurona de la capa oculta y la *k-ésima* neurona de la capa de salida.
θ_k^{o}	Término de tendencia de la *k-ésima* neurona de la capa de salida.
$Neta_{pk}^{o}$	Entrada neta de la *k-ésima* neurona de la capa de salida.
y_{pk}	Salida de la *k-ésima* unidad de salida
f_k^{o}	Función de activación de la *k-ésima* unidad de salida $o \in \Re^m; x \in \Re^n$
d_{pk}	Valor de salida deseado para la *k-ésima* neurona de la capa de salida.
e_p	Valor del error para el *p-ésimo* patrón de aprendizaje.
α	Taza o velocidad de aprendizaje
δ^{o}_{pk}	Término de error para la *k-ésima* neurona de la capa de salida.
δ^{h}_{pj}	Término de error para la *j-ésima* neurona de la capa oculta *h*
f'^{h}_{j}	Derivada de la función de activación de la *j-ésima* neurona de la capa oculta.
f'^{o}_{k}	Derivada de la función de activación de la *k-ésima* neurona de la capa de salida.

Algoritmo backpropagation: regla delta generalizada

El objetivo buscado con el aprendizaje en las redes neuronales, multicapa, es el de poder establecer una transformación matemática $\Phi(x)$ que relacione adecuadamente los pares ordenados de ejemplos de entradas de excitación a la red y su correspondiente salida deseada. La calidad de la estimación va a depender, fundamentalmente, del número de ejemplos disponibles del problema bajo observación y que la muestra sea lo suficientemente representativa. A continuación vamos a describir las etapas de este algoritmo.

Procesamiento de datos hacia adelante "feedforward"

Como primer paso estimulamos la red neuronal con el vector de entrada:

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

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

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

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

$$i_{pj}^{\ h} = f_j^h(Neta_{pj}^{\ h}) \tag{3.3}$$

Una vez calculadas las salidas de las neuronas de la capa oculta, éstas se convierten en las señales de excitación de las neuronas de la capa de salida y así podemos calcular la entrada neta de la *k-ésima* neurona de la capa de salida:

$$Neta_{pk}^{\ o} = \sum_{j=1}^{L} w_{kj}^{\ o} i_{pj}^{\ o} + \theta_k^{\ o} \tag{3.4}$$

Con base en la función de activación de la *k-ésima* neurona de la capa de salida podemos calcular la salida estimada por la red neuronal ante el estímulo de entrada:

$$y_{pk} = f_k^{\ o}(Net_{pk}^{\ o}) \tag{3.5}$$

Error en las capas oculta y de salida

Antes de continuar con el cálculo del error en las capas oculta y de salida, recordemos como modifica la Regla Delta o LMS (*Least Mean Square*) los pesos sinápticos de una red neuronal con base en la ecuación 3.6.

$$\begin{aligned} w_i(t+1) &= w_i(t) + 2\alpha e_p x_i \\ e_p &= d - y \end{aligned} \tag{3.6}$$

En la Regla Delta, e_p se define como el error generado por una única neurona o elemento de procesamiento. Con el algoritmo *Backpropagation* calculamos el error global, considerando todas las unidades de procesamiento y sumando el aporte al error de cada una de las neuronas,

$$E_p = \frac{1}{2}\sum_{p=1}^{P}\sum_{k=1}^{M} e_{pk}^2 \tag{3.7}$$

El aporte unitario al error global e_{pk} se define como el error de la *k-ésima* neurona y se calcula con la ecuación 3.8, donde d_{pk} representa la salida deseada.

$$e_{pk} = (d_{pk} - y_{pk}) \tag{3.8}$$

La búsqueda del mínimo en la superficie de error se fundamenta en el cálculo de su ***gradiente descendente*** ∇E_p. Para efectos de la deducción de este algoritmo vamos a considerar que el análisis lo hacemos considerando *p-ésimo* patrón de aprendizaje, por lo que en la ecuación 3.7, eliminamos los términos correspondientes a la sumatoria desde *p=1* hasta *P*.

En un primero paso vamos a calcular la derivada del error global respecto del peso w_{kj}^{o}. Para efectos de este procedimiento, sustituimos en la ecuación 3.7 el valor de e_{pk} que obtuvimos en la ecuación 3.8. y procedemos al cálculo de esta derivada.

$$E_p = \frac{1}{2}\sum_{k=1}^{M}(d_{pk} - y_{pk})^2$$

$$\frac{\partial E_p}{\partial w_{kj}^{o}} = \frac{\partial}{\partial w_{kj}^{o}}\left[\frac{1}{2}\sum_{k=1}^{M}(d_{pk} - y_{pk})^2\right]$$

$$\frac{\partial E_p}{\partial w_{kj}^{o}} = \frac{\partial}{\partial w_{kj}^{o}}\left[\frac{1}{2}\sum_{k=1}^{M}(d_{pk} - f_k^{o}(Neta_{pk}^{o}))^2\right]$$

$$\frac{\partial E_p}{\partial w_{kj}^{o}} = -(d_{pk} - f_k^{o}(Neta_{pk}^{o}))\cdot f_k^{o'}\frac{\partial Neta_{pk}^{o}}{\partial w_{kj}^{o}}$$

Ahora calculemos la derivada de la entrada neta, usando la ecuación 3.4,

$$\frac{\partial Neta_{pk}^{o}}{\partial w_{kj}^{o}} = \frac{\partial}{\partial w_{kj}^{o}}\left[\sum_{j=1}^{L} w_{kj}^{o} i_{pj}^{h} + \theta_k^{o}\right] = i_{pj}^{h}$$

Con base en este resultado, podemos calcular el gradiente que corresponde a la derivada del error global respecto del peso w_{kj}^{o},

$$\frac{\partial E_p}{\partial w_{kj}^{\ o}} = -(d_{pk} - y_{pk}^{\ o}) \cdot f_k^{\ o'}(Neta_{pk}^{\ o}) \cdot i_{pj}^h$$

En el capítulo 2, vimos que la intención de aplicar concepto del gradiente descendente es buscar paulatinamente un punto de error mínimo siempre guiados por la valor negativo de la derivada del error global, por lo que la expresión que utilizaremos corresponde al negativo del gradiente del error $(-\nabla E_p)$.

$$-\nabla E_p = -\frac{\partial E_p}{\partial w_{kj}^{\ o}} = (d_{pk} - y_{pk}^{\ o}) \cdot f_k^{\ o'}(Neta_{pk}^{\ o}) \cdot i_{pj}^h \tag{3.9}$$

El proceso de entrenamiento de la red busca como objetivo fundamental modificar el peso $w_{kj}^{\ o}$, esta modificación se realiza con base en la ecuación 3.10.

$$w_{kj}^{\ o}(t+1) = w_{kj}^{\ o}(t) + \Delta w_{kj}^{\ o}(t) = w_{kj}^{\ o}(t) + \alpha\left(-\nabla E_p\right) \tag{3.10}$$

Sustituimos el valor del gradiente en esta ecuación y obtenemos la ecuación 3.11 para la modificación de los pesos,

$$w_{kj}^{\ o}(t+1) = w_{kj}^{\ o}(t) + \alpha(d_{pk} - y_{pk}^{\ o}) f_k^{\ o'}(Neta_{pk}^{\ o}) i_{pj}^h \tag{3.11}$$

De esta expresión inferimos que la función de activación f_k^o debe ser derivable, por consiguiente, tanto en las capas ocultas como de salida, generalmente escogemos una función de activación lineal o sigmoidal. En el primer caso, la derivada es uno, ecuación 3.12 y, en el segundo caso, el resultado lo entrega la ecuación 3.13.

$$f_k^{\ o}(Neta_{pk}^{\ o}) = Neta_{pk}^{\ o} \rightarrow f_k^{\ o'} = 1 \tag{3.12}$$

$$f_k^{\ o}(Neta_{pk}^{\ o}) = \frac{1}{1+e^{-Net_{pk}^{\ o}}} \tag{3.13}$$

$$f_k^{\ o'} = f_k^{\ o}(1 - f_k^{\ o})$$

$$f_k^{\ o'} = y_{kp}^{\ o}(1 - y_{pk}^{\ o})$$

Luego de calcular las derivadas dependiendo del tipo de función de activación que escojamos, la ecuación 3.11 se cambia por la 3.14, si la función de activación es lineal,

$$w_{kj}^{\ o}(t+1) = w_{kj}^{\ o}(t) + \alpha(d_{pk} - y_{pk}^{\ o}) i_{pj}^h \tag{3.14}$$

Para el caso de una función de activación sigmoidal, resulta la ecuación 3.15.

$$w_{kj}^{\ o}(t+1) = w_{kj}^{\ o}(t) + \alpha(d_{pk} - y_{pk}^{\ o}) y_{kp}^{\ o}(1 - y_{pk}^{\ o}) i_{pj}^{h} \tag{3.15}$$

Con el fin de simplificar las expresiones 3.14 y 3.15 en una única ecuación para facilitar su representación en el algoritmo, definimos el término del error en las neuronas de la capa de salida, con base en la ecuación 3.16.

$$\begin{aligned} \delta_{pk}^{\ o} &= (d_{pk} - y_{pk}^{\ o}) f_k^{o'}(Neta_{pk}^{\ o}) \\ \delta_{pk}^{\ o} &= e_{pk} f_k^{o'}(Neta_{pk}^{\ o}) \end{aligned} \tag{3.16}$$

Por lo que la ecuación de modificación de pesos la unificamos en la expresión 3.17

$$w_{kj}^{\ o}(t+1) = w_{kj}^{\ o}(t) + \alpha \delta_{pk}^{o} i_{pj}^{h} \tag{3.17}$$

Actualización de pesos de las capas ocultas

Para la capa oculta no es posible calcular el error directamente, ya que no se conocen las salidas deseadas de esta capa. Sin embargo, si usamos el concepto de retropropagación, observamos que existe una manera de estimar este error de la capa oculta partiendo del error en la capa de salida. Veamos como podemos actualizar los pesos de la capa oculta y, en particular, el peso $w_{ji}^{\ h}$. Retomemos en la ecuación 3.18, el error global en la capa de salida para el patrón p-ésimo.

$$E_p = \frac{1}{2} \sum_{k=1}^{M} (d_{pk} - y_{pk})^2 \tag{3.18}$$

Ahora podemos calcular el gradiente descendente $-\nabla E_p$ con respecto a, $w_{ji}^{\ h}$

$$\frac{\partial E_p}{\partial w_{ji}^{\ h}} = \frac{\partial}{\partial w_{ji}^{\ h}} \left[\frac{1}{2} \sum_{k=1}^{M} (d_{pk} - y_{pk})^2 \right] \tag{3.19}$$

$$\frac{\partial E_p}{\partial w_{ji}^{\ h}} = -\sum_{k=1}^{M} (d_{pk} - y_{pk}) \cdot \frac{\partial y_{pk}^{\ h}}{\partial w_{ji}^{\ h}} \tag{3.20}$$

Apliquemos la regla de la cadena sucesivamente para el cálculo de la derivada interna,

$$\frac{\partial E_p}{\partial w_{ji}^{\ h}} = -\sum_{k=1}^{M}(d_{pk} - y_{pk}) \cdot \frac{\partial y_{pk}^{\ h}}{\partial Neta_{pk}^{\ o}} \cdot \frac{\partial Neta_{pk}^{\ o}}{\partial i_{pj}} \cdot \frac{\partial i_{pj}^{h}}{\partial Neta_{pj}^{\ h}} \cdot \frac{\partial Neta_{pj}^{\ h}}{\partial w_{ji}^{\ h}} \quad (3.21)$$

$$\frac{\partial E_p}{\partial w_{ji}^{\ h}} = -\sum_{k=1}^{M}(d_{pk} - y_{pk}) \cdot f_k^{\ o}(Net_{pk}^{\ o}) \cdot w_{kj}^{\ o} \cdot f_j^{\ h'} \cdot x_{pi} \quad (3.22)$$

En la ecuación 3.22 calculamos el gradiente del error global respecto de los pesos de la capa oculta y con base en este valor, definamos la expresión para la modificación de los pesos de la capa oculta,

$$w_{ji}^{\ h}(t+1) = w_{ji}^{\ h}(t) + \Delta w_{ji}^{\ h}(t) = w_{ji}^{\ h}(t) + \alpha\left(-\frac{\partial E_p}{\partial w_{ji}^{\ h}}\right) \quad (3.23)$$

Al reemplazar en esta ecuación la expresión del gradiente, obtenemos la ecuación 3.24 que nos permite actualizar los pesos de la capa oculta.

$$w_{ji}^{\ h}(t+1) = w_{ji}^{\ h}(t) + \alpha\left(\sum_{k=1}^{M}(d_{pk} - y_{pk}) \cdot f_k^{\ o}(Neta_{pk}^{\ o}) \cdot w_{kj}^{\ o} \cdot f_j^{\ h'} \cdot x_{pi}\right) \quad (3.24)$$

De manera similar a lo planteado en la capa de salida, definamos en la ecuación 3.25 el término del error en la capa oculta,

$$\delta_{pj}^{\ h} = f_j^{\ h'}(Neta_{pj}^{\ h})\sum_{k=1}^{M}\delta_{pk}^{\ o} w_{kj}^{\ o} \quad (3.25)$$

En la ecuación 3.25 observamos que el término de la sumatoria, representa matemáticamente el concepto de *Backpropagation*, ya que el error de la capa oculta está dado en función de los pesos sinápticos y de los términos de error de la capa de salida. Finalmente, la modificación de los pesos sinápticos de la capa oculta la realizaremos con base en la ecuación 3.26.

$$w_{ji}^{\ h}(t+1) = w_{ji}^{\ h}(t) + \alpha\delta_{pj}^{\ h} x_{pi} \quad (3.26)$$

Pasos del algoritmo backpropagation

1. Inicializamos los pesos del MLP.
2. Mientras la condición de parada sea falsa ejecutamos los pasos 3 a 12
3. Aplicamos un vector de entrada $\boldsymbol{x_p} = [x_{p1}, x_{p2}, \ldots, x_{pi}, \ldots, x_{pn}]^T$.
4. Calculamos los valores de las entradas netas para la capa oculta.

$$Neta_{pj}^{\ h} = \sum_{i}^{N} w_{ji}^{\ h} x_{pi} + \theta_j^{\ h}$$

5. Calculamos la salida de la capa oculta.

$$i_{pj}^{\ h} = f_j^h(Neta_{pj}^{\ h})$$

6. Calculamos los valores netos de entrada para la capa de salida.

$$Neta_{pk}^{\ o} = \sum_{j=1}^{L} w_{kj}^{\ o} i_{pj}^{\ o} + \theta_k^{\ o}$$

7. Calculamos las salidas de la red.

$$y_{pk} = f_k^{\ o}(Neta_{pk}^{\ o})$$

8. Calculamos los términos de error para las unidades de salida.

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

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

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

10. Actualizamos los pesos en la capa de salida.

$$w_{kj}^{\ o}(t+1) = w_{kj}^{\ o}(t) + \alpha \delta_{pk}^{o} i_{pj}^{h}$$

11. Actualizamos pesos en la capa oculta.

$$w_{ji}^{\ h}(t+1) = w_{ji}^{\ h}(t) + \alpha \delta_{pj}^{\ h} x_{pi}$$

12. Verificamos si el error global cumple con la condición de finalizar.

$$E_p = \frac{1}{2} \sum_{p=1}^{P} \sum_{k=1}^{M} \left(d_{pk} - y_{pk}\right)^2$$

Algoritmo gradiente descendente con alfa variable

Uno de los inconvenientes que presenta el algoritmo básico de gradiente descendente con *Backpropagation* es tener el parámetro de aprendizaje α fijo. Este parámetro cuyo valor depende de la aplicación que estemos solucionando, también puede variar en el proceso de aprendizaje con el fin de

modificar el tamaño de la variación de los pesos Δw_i, para acelerar la convergencia del algoritmo de aprendizaje. Una estrategia efectiva para variar el parámetro de aprendizaje es el incrementarlo o disminuirlo en cada iteración, dependiendo de la manera como evolucione el error de entrenamiento, con base en la regla descrita en la ecuación 3.27.

$$\alpha_{k+1} = \begin{cases} \rho\alpha_k, si & E_p(w_{k+1}) < E(w_k) \\ \sigma\alpha_k, si & E_p(w_{k+1}) \geq E(w_k) \end{cases} \tag{3.27}$$

Donde, $\rho>1$ y $0<\sigma<1$.

Los parámetros ρ, σ y α_0 son iniciados de manera heurística. Generalmente ρ es definido con un valor cercano a uno (por ejemplo ρ =1.1), para evitar incrementos exagerados en el error de entrenamiento y σ con un valor tal que reduzca α rápidamente para de esta manera abandonar valores elevados de la razón de aprendizaje (por ejemplo σ = 0.5).

Pasos del algoritmo gradiente descendente con α variable

1. Definir los pesos iniciales. ***Condición_parar*** =Falsa
2. Mientras ***Condición_parar*** = Falsa ejecutar los pasos 3 a 12.
3. Aplicamos un vector de entrada
   $\boldsymbol{x_p = [x_{p1}, x_{p2}, ..., x_{pi}, ..., x_{pn}]^T}$.
4. Calculamos los valores de las entradas netas para la capa oculta.
   $$Neta_{pj}^{\ h} = \sum_{i}^{N} w_{ji}^{\ h} x_{pi} + \theta_j^{\ h}$$
5. Calculamos la salida de la capa oculta.
   $$i_{pj}^{\ h} = f(Neta_{pj}^{\ h})$$
6. Calculamos los valores netos de entrada para la capa de salida.
   $$Neta_{pk}^{\ o} = \sum_{j=1}^{L} w_{kj}^{\ o} i_{pj}^{\ o} + \theta_k^{\ o}$$
7. Calculamos las salidas de la red.
   $$y_{pk} = f_k^{\ o}(Neta_{pk}^{\ o})$$
8. Calculamos el error para las unidades de salida.
   $$\delta_{pk}^{\ o} = (d_{pk} - y_{pk}) f_k^{\ o'}(Neta_{pk}^{\ o})$$

9. Estimamos el error para las unidades ocultas.

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

10. Calculamos el error global de salida.

$$E_p = \frac{1}{2P}\sum_{p=1}^{P}\sum_{k=1}^{M}\left(d_{pk} - y_{pk}\right)^2$$

11. Hacemos $\boldsymbol{E_{prev} = E_p}$
12. Si $\boldsymbol{E_{prev} < E_{min}}$ entonces
    Condición_parar = Verdadera y salir
    Sino
    Condición_parar = Falsa
13. Actualizamos los pesos en la capa de salida.

$$w_{kj}{}^{o}(t+1) = w_{kj}{}^{o}(t) + \alpha\delta_{pk}{}^{o} i_{pj}$$

14. Actualizamos los pesos en la capa oculta.

$$w_{ji}{}^{h}(t+1) = w_{ji}{}^{h}(t) + \alpha\delta_{pj}{}^{h} x_{pi}$$

15. Calculamos el error $\boldsymbol{E_p}$
16. Si $\boldsymbol{E_p < E_{prev}}$ entonces

    $E_{prev} = E_p$

    $\alpha_{k+1} = \rho\alpha_k$, *donde* $\rho > 1$

    Volvemos al paso 3
    sino

    $\alpha_{k+1} = \sigma\alpha_k$, *donde* $0 < \sigma < 1$

    Volvemos al paso 13

ALGORITMOS DE ALTO DESEMPEÑO PARA REDES NEURONALES MLP

El algoritmo *backpropagation* basado en el gradiente descendente resulta ser un método de entrenamiento lento para ciertas aplicaciones donde requerimos una alta velocidad de convergencia. En esta sección presentaremos dos nuevos algoritmos de alto desempeño para el aprendizaje de redes MLP que emplean técnicas de optimización numérica estándar:

- Algoritmo de Aprendizaje del Gradiente Conjugado
- Algoritmo de Aprendizaje de Levenberg Marquardt

Algoritmo de aprendizaje del gradiente conjugado

En esta sección, presentamos una de las más populares y efectivas familias de algoritmos de optimización multivariable para el aprendizaje de redes tipo MLP. El algoritmo del Gradiente Conjugado es un método avanzado de aprendizaje supervisado fuera de línea para la red tipo MLP. Usualmente funciona mejor que el algoritmo *Backpropagation* y lo podemos utilizar en el mismo tipo de aplicaciones. Este algoritmo lo recomendamos para cualquier red neuronal con un gran número de pesos (más de varios centenares) y/o múltiples neuronas de salida.

Lo que buscamos con los algoritmos de aprendizaje para redes MLP es minimizar la función de error o índice de desempeño de la red, considerando una superficie cuadrática *n-dimensional*, de tal manera que al finalizar el aprendizaje se encuentre un vector de pesos cuyos elementos, al reemplazarse en la función de error, entreguen el mínimo deseado. En el método original del gradiente descendente calculamos el gradiente en un punto o vector inicial de pesos y buscamos el mínimo moviéndonos hacia la dirección definida por el negativo del gradiente, entonces, en el nuevo punto que encontramos el gradiente es perpendicular a la dirección de búsqueda inicial o del gradiente anterior. En la mayoría de los casos tal como observamos en la figura 3.2, el nuevo punto que encontramos no es muy diferente al anterior y se pierde un gran esfuerzo computacional minimizando en esta dirección.

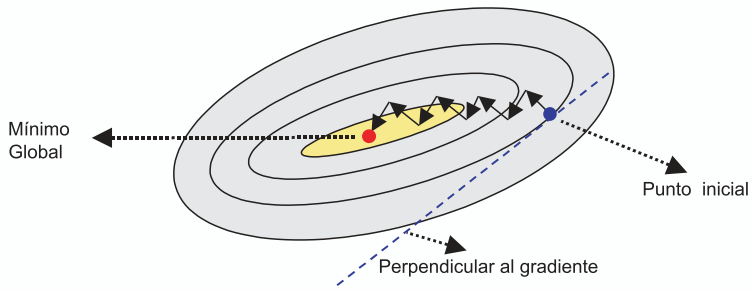


Fig. 3.2 Evolución del Gradiente

Aunque la función decrezca rápidamente alrededor del negativo del gradiente, esto no necesariamente garantiza una mayor velocidad de convergencia. En el algoritmo de gradiente conjugado buscamos minimizar el error sobre una dirección que es conjugada a la que en este momento tenemos, lo cual produce generalmente una convergencia más rápida, que si la búsqueda la hacemos únicamente en la dirección del gradiente descendente. Con el fin de ilustrar este método, consideremos dos vectores $\boldsymbol{r}$ y $\mathbf{s}$ que son conjugados. Esto quiere decir que al moverse a lo largo de uno de ellos, por ejemplo por $\boldsymbol{r}$, el cambio en el gradiente de la función es perpendicular al vector $\boldsymbol{s}$, que matemáticamente se representa con la ecuación 3.28.

$$\mathbf{r}^{T}\mathbf{Hs} = 0 \tag{3.28}$$

Donde ***H*** es la Matriz *Hessiana* y se define en la ecuación 3.29

$$\mathbf{H} = \begin{bmatrix} \frac{\partial^2 y}{\partial^2 w_1} & & \frac{\partial^2 y}{\partial w_1 w_j} \\ . & & . \\ . & \frac{\partial^2 y}{\partial w_i w_j} & . \\ . & & .. \\ \frac{\partial^2 y}{\partial w_i w_1} & & \frac{\partial^2 y}{\partial^2 w_n} \end{bmatrix} \tag{3.29}$$

De la expresión 3.28 deducimos que para obtener un vector conjugado, debemos calcular la Matriz Hessiana, que implica una alta carga computacional, por lo que propondremos un método alternativo que no implique este cálculo. Antes de explicarlo, veamos gráficamente en las curvas de nivel de la función de error de la figura 3.3, la ventaja de minimizar una función de error utilizando esta idea de movernos en una dirección conjugada al gradiente.

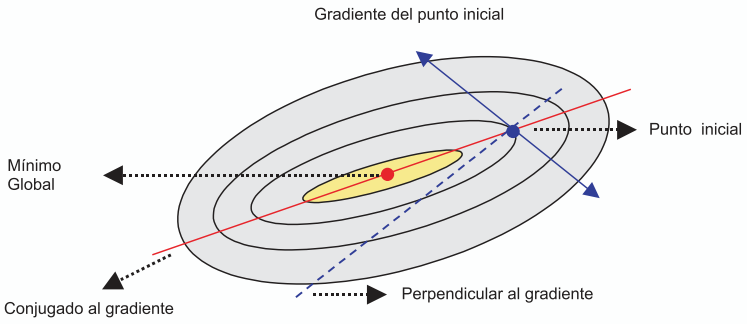


Fig. 3.3 Curvas de nivel de la función de error

Como se muestra en la figura 3.3, si en lugar de calcular una dirección perpendicular a la dirección previa, se calcula una dirección conjugada y se minimiza a lo largo de ésta, se llega más rápido al mínimo global de la función que si seguimos la dirección del gradiente. Por lo que podemos extender esta idea ilustrado en dos dimensiones a un caso de *n* dimensiones, donde para minimizar una función cuadrática de *n* variables (pesos de la red), se requieren ***n*** minimizaciones lineales en direcciones que son mutuamente conjugadas.

Actualización de pesos y determinación de la dirección conjugada

La Matriz *Hessiana* juega un papel importante en el momento de determinar la dirección conjugada. Pero en problemas de optimización complejos como el entrenamiento de una red neuronal para ser aplicada en el reconocimiento de voz o de caracteres manuscritos, es difícil o imposible calcularla. Por esta razón, propondremos un método que permite calcular esta dirección conjugada al gradiente sin requerir el cálculo formal de esta matriz.

Consideremos a $\boldsymbol{g}_i$ como el vector con dirección negativa al gradiente de la función de error en la iteración *i*-ésima. Sea $\boldsymbol{h}_i$ la dirección de búsqueda en esta iteración. Entonces se desarrolla una línea de búsqueda para determinar la distancia óptima para moverse a lo largo de la dirección de búsqueda común, luego el vector de pesos de la red es actualizado de acuerdo con la ecuación 3.30.

En este nuevo algoritmo, el cambio en los pesos lo calculamos con la ecuación 3.30, donde ***k(t)*** es el factor de multiplicación resultado de la búsqueda y ***h(t)*** es el vector de la dirección en la cual vamos a llevar a cabo el proceso de minimización e incluye el aporte de la dirección conjugada del gradiente.

$$\mathbf{w}(t+1) = \mathbf{w}(t) + \mathbf{k}(t)\ \mathbf{h}(t) \tag{3.30}$$

Para inicializar el algoritmo, igualamos $\boldsymbol{g}(0)$ y ***h(0)*** al negativo del gradiente en el punto inicial.

$$\mathbf{g}(0) = \mathbf{h}(0) = -\overline{\nabla} E(0) \tag{3.31}$$

En cada paso debemos considerar la evolución del algoritmo llevando un registro del gradiente y la dirección de búsqueda del paso anterior. Cada vector de dirección de búsqueda lo calculamos como una combinación lineal del vector gradiente en el instante actual y el vector de dirección de búsqueda previa.

$$\mathbf{h}(t) = -\mathbf{g}(t) + \gamma \mathbf{h}(t-1) \tag{3.32}$$

Donde γ es un nuevo parámetro variante en el tiempo y, para su ajuste, existen varias reglas que nos permiten determinar su valor en términos de ***g(t)*** y ***g(t-1)***. Las diferentes versiones que se tienen del algoritmo del gradiente conjugado son distinguidas por la manera como calculamos la constante γ de las cuales presentaremos tres variaciones.

La primera, es la regla de actualización Fletcher-Reeves cuya actualización de γ se hace con la ecuación 3.33, que corresponde a la razón de

cambio de la norma cuadrada del gradiente actual a la norma cuadrada del gradiente previo. Este algoritmo basado en el Gradiente Conjugado requiere sólo un poco más de memoria que otros algoritmos más simples, así que ofrecen una buena opción para redes con un gran número de neuronas, conexiones y por ende pesos sinápticos.

$$\gamma = \frac{\mathbf{g}^T(t)\mathbf{g}(t)}{\mathbf{g}^T(t-1)\mathbf{g}(t-1)} \tag{3.33}$$

Otra forma de calcular la constante γ es la Regla de actualización Polak-Ribiére, que al igual que con el algoritmo Fletcher-Reeves, la dirección de búsqueda en cada iteración es determinada por la ecuación (3.32) y la constante γ es calculada con la ecuación 3.34.

$$\gamma = \frac{\big(\mathbf{g}(t)-\mathbf{g}(t-1)\big)^T\mathbf{g}(t)}{\mathbf{g}^T(t-1)\mathbf{g}(t-1)} \tag{3.34}$$

Este es el producto interno entre el cambio en el gradiente previo y el gradiente actual dividido por la norma cuadrada del gradiente previo. Los requerimientos de memoria para Polak-Ribiére (cuatro vectores) son ligeramente mas grandes que para Fletcher-Reeves (tres vectores).

La última regla para calcular la constante γ la presentamos en la ecuación 3.35. La propusieron Hestenes y Steifel y corresponde al producto interno entre el cambio en el gradiente previo y el gradiente actual, dividido por el producto interno entre el cambio del gradiente previo y la dirección de búsqueda en el instante anterior.

$$\gamma = \frac{\big(\mathbf{g}(t)-\mathbf{g}(t-1)\big)^T\mathbf{g}(t)}{\big(\mathbf{g}(t)-\mathbf{g}(t-1)\big)^T\mathbf{h}(t-1)} \tag{3.35}$$

Una vez calculamos el valor de γ podemos encontrar con la ecuación 3.32 la nueva dirección de búsqueda $\boldsymbol{h}(t)$. Para calcular el tamaño de la variación de peso en la dirección de búsqueda $\boldsymbol{h}(t)$ utilizamos una rutina de minimización lineal. En los algoritmos de aprendizaje hasta ahora estudiados, el valor de la variación de peso Δw es proporcional al parámetro de aprendizaje α, el cual es generalmente constante. En el algoritmo de gradiente conjugado no tenemos un parámetro de aprendizaje, sino que el valor de Δw se ajusta en cada iteración usando una rutina de minimización lineal.

Finalmente, el algoritmo del Gradiente Conjugado requiere de algunos cálculos empleados en el Algoritmo Backpropagation para determinar el gradiente de la función de error con respecto a todos los pesos de la red. A diferencia del algoritmo *Backpropagation* que presentamos en la sección

3.2, en este algoritmo la actualización de los pesos se lleva a cabo en un entrenamiento por lotes, es decir, en cada iteración todos lo patrones deben ser presentados a la red para determinar el error cuadrático promedio (MSE) de la ecuación 3.36, cada vez que se requiera evaluar la función de error durante el proceso de aprendizaje.

$$E_p = \frac{1}{2P}\sum_{p=1}^{P}\sum_{k=1}^{M}\left(d_{pk} - y_{pk}\right)^2 \tag{3.36}$$

La actualización de pesos por lotes tiene como ventaja incrementar la rata de aprendizaje de una red MLP, ya que en cada iteración se presentan todos los patrones de entrada antes de realizar la actualización.

Pasos del algoritmo gradiente conjugado

1. Definimos los parámetros iniciales:
   Condición_parar = Falsa
   Valores iniciales de los pesos y bias de la red $\boldsymbol{w}$.
   Factor de aprendizaje α.
   Error mínimo deseado $\boldsymbol{E_{min}}$
2. Mientras la *Condición_parar* sea falsa ejecutamos los pasos 3 a 7
3. Presentamos todos los patrones de entrada a la red
   $x_p = \{x_1, x_2, ..., x_i, ..., x_n\}$.
4. Calculamos la entrada neta para las neuronas de la capa oculta.
   $$Neta_{pj}^{h} = \sum_{i=1}^{N} w_{ji}^{h} x_{pi} + \theta_j^{h}$$
5. Calculamos la salida de la capa oculta.
   $$i_{pj}^{h} = f(Neta_{pj}^{h})$$
6. Calculamos la entrada neta para las neuronas de la capa de salida.
   $$Neta_{pk}^{o} = \sum_{j=1}^{L} w_{kj}^{o} i_{pj}^{o} + \theta_k^{o}$$
7. Calculamos la salida de la red neuronal.
   $$y_{pk} = f_k^{o}(Neta_{pk}^{o})$$
8. Calculamos el gradiente para las capas de salida y oculta en la primera iteración; en las siguientes usamos la calculada en el paso 17.

9. Inicializamos el gradiente y la dirección de búsqueda para la primera iteración, para las dos capas, en las siguientes iteraciones se omite este paso

$$\mathbf{h}(0) = -\mathbf{g}(0)$$

10. Calculamos el error global promedio de la red.

$$E_p = \frac{1}{2P}\sum_{p=1}^{P}\sum_{k=1}^{M}(d_{pk} - y_{pk})^2$$

11. Si $E_p > E_{min}$ ejecutamos los pasos 12 a 18.
En caso contrario *Condición_parar* = *Verdadera* y salir.
12. Calculamos el **k** *(t)* próximo usando una rutina de minimización lineal *mlin*

$$\mathbf{k}(t) = mlin\left(\mathbf{w}(t), \mathbf{g}(t), E_p\right)$$

13. Actualizamos los pesos de la red para cada una de las capas.

$$\mathbf{w}(t+1) = \mathbf{w}(t) + \mathbf{k}(t)\,\mathbf{h}(t)$$

14. Calculamos la regla de actualización por cualquiera de las variantes.

Flecher -Reeves $\quad \gamma = \dfrac{\mathbf{g}^T(t)\mathbf{g}(t)}{\mathbf{g}^T(t-1)\mathbf{g}(t-1)}$

Polak -Ribiére $\quad \gamma = \dfrac{\left(\mathbf{g}(t) - \mathbf{g}(t-1)\right)^T \mathbf{g}(t)}{\mathbf{g}^T(t-1)\mathbf{g}(t-1)}$

Hestenes -Steifel $\quad \gamma = \dfrac{\left(\mathbf{g}(t) - \mathbf{g}(t-1)\right)^T \mathbf{g}(t)}{\left(\mathbf{g}(t) - \mathbf{g}(t-1)\right)^T \mathbf{h}(t-1)}$

15. Calculamos la siguiente dirección de búsqueda para cada capa.

$$\mathbf{h}(t) = -\mathbf{g}(t) + \gamma \mathbf{h}(t-1)$$

16. Normalizamos la dirección de búsqueda $nor_h(t) = \dfrac{\mathbf{h}(t)}{\|\mathbf{h}(t)\|}$

17. Volvemos al paso 2

Algoritmo de aprendizaje levenberg marquardt

En esta sección presentamos una alternativa al método de Gradiente Conjugado para acelerar la fase de aprendizaje de una red neuronal artificial, en honor a sus inventores se ha denominado: Algoritmo Levenberg

Marquardt. Debido a la gran carga computacional que implica su ejecución, no recomendamos el uso de este algoritmo cuando se vaya a entrenar una red con alto número de conexiones. Para redes con una arquitectura con pocas neuronas y conexiones, este algoritmo será más rápido y efectivo que el Algoritmo del Gradiente Conjugado.

Una de las grandes fortalezas que presenta este algoritmo es la combinación de dos estrategias de minimización, el método de gradiente descendente y el método de Newton. Vamos a revisar, rápidamente, este segundo método y en la ecuación recursiva 3.37 presentamos como se localiza un valor mínimo de una función de una variable *f(x)*, utilizando la primera y segunda derivada.

$$x_{\min}(t+1) = x_{\min}(t) - \frac{f'(x_{\min}(t))}{f''(x_{\min}(t))} \tag{3.37}$$

Con base en esta ecuación podemos inferir la ecuación 3.38, donde vamos a minimizar el error global $\boldsymbol{E}_p$ en el espacio de los pesos sinápticos representado por la matriz $\boldsymbol{W}$.

$$W_{\min}(t+1) = W_{\min}(t) - \frac{E_p'}{E_p''} \tag{3.38}$$

La segunda derivada del error global corresponde a la Matriz Hessiana $\boldsymbol{H}$ y la primera derivada la conocemos como el vector gradiente $\boldsymbol{g}$. El vector gradiente y la matriz Hessiana de la función de error los podemos calcular utilizando la regla de la cadena. Así, para una neurona de la capa de salida el vector gradiente está compuesto por las derivadas parciales del error con respecto a cada uno de los pesos $\boldsymbol{w}_i$ de la red, el elemento ***(i,j)*** de la matriz Hessiana lo calculamos con las segundas derivadas parciales del error con respecto a los pesos $\boldsymbol{w}_i$ y $\boldsymbol{w}_j$.

Recordemos la expresión para calcular el error para una neurona de salida de Perceptron multicapa,

$$e = (d - y)^2 \tag{3.39}$$

La aplicación directa de la regla de la cadena sobre esta expresión genera el siguiente resultado:

$$\frac{\partial e_p}{\partial w_{kj}^{\,o}} = -2(d_{pk} - y_{pk})\frac{\partial y_{pk}}{\partial w_{kj}^{\,o}} \tag{3.40}$$

$$\frac{\partial^2 e_p}{\partial w_{kj}^{\ o} \partial w_{ji}^{\ h}} = 2\left[\frac{\partial y_{pk}}{\partial w_{kj}^{\ o}} \frac{\partial y_{pk}}{\partial w_{ji}^{\ h}} - (d_{pk} - y_{pk}) \frac{\partial^2 y_{pk}}{\partial w_{kj}^{\ o} \partial w_{ji}^{\ h}}\right] \tag{3.41}$$

Las ecuaciones 3.40 y 3.41, nos permiten calcular las componentes del vector gradiente y de la matriz Hessiana.

Los factores que componen la expresión para la segunda derivada parcial del error, es decir, la derivada parcial de la salida con respecto a los pesos, son producidos en el algoritmo gradiente descendente con *Backpropagation*.

En la ecuación 3.41 es posible eliminar el segundo término, considerando que éste se encuentra multiplicado por el error. Es razonable asumir que los errores generados en cada iteración del entrenamiento son completamente independientes y distribuidos alrededor de cero. Algunas veces el error será positivo y otras negativo. Esta situación mantiene un balance que permite cancelar el segundo término de la ecuación 3.41. Estos errores producidos por la aproximación de la matriz Hessiana no afectarán la precisión de los resultados. Su efecto se reflejará en un proceso de convergencia más lento, haciendo más confiable la tendencia hacia el antiguo método de Gradiente Descendente.

La obtención del gradiente y la matriz Hessiana, después de realizar la aproximación para esta última, requieren sólo del cálculo de la derivada de la salida de cada neurona de la última capa con respecto a cada uno de los pesos. De acuerdo con la teoría aplicada al método de aprendizaje gradiente descendente con *Backpropagation*, es posible calcular las expresiones para la derivada teniendo en cuenta el tipo de pesos:

$$\frac{\partial y_{pk}}{\partial w_{kj}^{\ o}} = f_k^{\ o'}(Neta_{pk}^{\ o}) \cdot i_{pj}^{h} \tag{3.42}$$

$$\frac{\partial y_{pk}}{\partial w_{ji}^{\ h}} = f_j^{\ h'}(Neta_{pj}^{\ h}) w_{ji}^{\ h} \cdot x_{pi} f_k^{\ o'}(Neta_{pk}^{\ o}) \tag{3.43}$$

Asumiendo una red de dos capas de procesamiento, la primera ecuación determina la derivada de la salida con respecto a uno de los pesos que conectan la capa oculta con la capa de salida, y la segunda representa la derivada de la salida con respecto a uno de los pesos que conectan las neuronas de entrada con la capa oculta, ***f*** representa la función de activación, ***i*** corresponde a la salida de la capa oculta.

La matriz ***H*** no la calculamos exactamente sino que recurrimos a su estimación, por lo que debemos establecer un mecanismo de control para garantizar la convergencia del algoritmo. En primer lugar se prueba la ecuación

del método de Newton, si al evaluarla el algoritmo no converge sino que el valor del error comienza a crecer, eliminamos este valor e incrementamos el valor de λ en la ecuación 3.44, con el fin de minimizar el efecto de la matriz **H** en la actualización de los pesos. Si λ es muy grande, el efecto de la matriz **H** prácticamente desaparece y la actualización de pesos se hace esencialmente con el algoritmo de gradiente descendente. Si el algoritmo tiene una clara tendencia hacia la convergencia disminuimos el valor de λ con el fin de aumentar el efecto de la matriz **H** y de esta manera garantizamos que el algoritmo se comporta con un predominio del Método de Newton.

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

$$\mathbf{W}(t+1) = \mathbf{W}(t) - (\mathbf{H} + \lambda \mathbf{I})^{-1}\mathbf{G} \tag{3.44}$$

Selección del parámetro λ

Existe un último aspecto a tratar antes de mostrar los pasos del algoritmo Levenberg Marquardt, se trata de la selección del valor del parámetro λ**.** Algunos autores recomiendan escoger un valor inicial fijo tal como λ=0.001; sin embargo, tal valor puede resultar lo suficientemente grande para algunas aplicaciones, y muy pequeño para otras. Un método para optimizar este parámetro es buscar en la diagonal principal de la matriz ***H*** el valor más grande y tomarlo como el valor inicial de λ**.**

El parámetro λ debe reducirse, si existe una tendencia a bajar el error producido por los pesos estimados en la ecuación 3.44, en caso contrario debemos incrementar el valor de λ**.** En el algoritmo utilizaremos ρ como factor de reducción y σ como factor de incremento.

Solución de la ecuación para determinar la variación de pesos Δ

De acuerdo con la ecuación 3.44, la variación de pesos Δ la definimos con la ecuación 3.45.

$$\mathbf{W}(t+1) = \mathbf{W}(t) - \Delta \tag{3.45}$$

El cálculo directo de Δ empleando está ecuación es lento y puede resultar inestable debido al cálculo de la inversa de la matriz ***H.*** Para solucionar estos inconveniente proponemos calcular Δ con la ecuación 3.46.

$$(\mathbf{H} + \lambda \mathbf{I})\Delta = \mathbf{G} \tag{3.46}$$

Bajo condiciones ideales estas dos ecuaciones son matemáticamente equivalentes, y uno de los métodos de álgebra lineal más indicados para

resolver el sistema es conocido como *Backsustitution*. La ventaja del método es que asegura que la matriz de coeficientes será no singular previendo siempre una solución aceptable.

Pasos del algoritmo levenberg marquardt

En el algoritmo Levenberg Marquardt actualizamos el vector de pesos de la red MLP en un aprendizaje por lotes, al igual que en el Algoritmo Gradiente Conjugado. La matriz Hessiana y el vector gradiente también los calculamos para cada patrón y promediados por el número total de éstos, como resultado de la realización de un aprendizaje por lotes.

1. Inicializamos los parámetros de la red:
   Definimos los valores iniciales de los pesos y *bias* de la red
   Definimos el valor inicial del parámetro de aprendizaje α
   Definimos el error mínimo deseado $\boldsymbol{E}_{min}$
   Definimos la ***Condición_parar*** = *Falsa* y *RESET* = *Falso*
2. Si *RESET* = *Falso* ejecutamos los pasos 3 a 13, sino vamos al paso 14.
3. Presentamos todos los patrones de entrenamiento a la red.
   $x_p = \{x_1, x_2, ..., x_i, ..., x_p\}$
4. Calculamos los valores de las entradas netas para la capa oculta.

$$Neta_{pj}^{h} = \sum_{i=1}^{N} w_{ji}^{h} x_{pi} + \theta_{j}^{h}$$

5. Calculamos la salida de la capa oculta.

$$i_{pj}^{h} = f(Neta_{pj}^{h})$$

6. Calculamos los valores netos de entrada para la capa de salida.

$$Neta_{pk}^{o} = \sum_{j=1}^{l} w_{kj}^{o} i_{pj}^{o} + \theta_{k}^{o}$$

7. Calculamos la salida de la red neuronal.

$$y_{pk} = f_{k}^{o}(Neta_{pk}^{o})$$

8. Calculamos el error global de la red y lo almacenamos en la variable. $\boldsymbol{E}_{prev}$ (Error previo)

$$Eprev = E = \frac{1}{2P} \sum_{p=1}^{P} \sum_{k=1}^{M} (d_{pk} - y_{pk})^2$$

9. Calculamos el gradiente para el conjunto de patrones.

$$\mathbf{G}_{save} = -\nabla E(0)$$

10. Calculamos la matriz ***H***.

$$\mathbf{H}_{save} = \mathbf{H}$$

11. Si $E_{prev} > E_{min}$ (el error previo es mayor al error mínimo deseado), ejecutamos los pasos 12 a 18, sino hacemos ***Condición_parar*** = Verdadera y salimos.
12. Inicializamos el parámetro λ.

$$\lambda = \mathbf{H}_{ijmax}$$

$$i = j$$

13. Inicializamos indicador RESET a verdadero y volver a paso 2.
14. Si RESET = verdadero, guardamos el vector gradiente y la matriz ***H***.

$$\mathbf{H} = \mathbf{H}_{save}$$

$$\mathbf{G} = \mathbf{G}_{save}$$

15. Incrementamos cada elemento de la diagonal de **H** en λ.

$$\mathbf{H} = \mathbf{H} + \lambda\mathbf{I}$$

16. Resolvemos el sistema de ecuaciones para Δ.

$$(\mathbf{H} + \lambda\mathbf{I})\Delta = \mathbf{G}$$

17. Adicionamos Δ al actual vector de pesos para actualización.

$$\mathbf{W}(t+1) = \mathbf{W}(t) - \Delta$$

18. Calculamos el error ***E***, el gradiente ***G*** y la matriz ***H*** con el nuevo vector de pesos.

$$E = \frac{1}{2P}\sum_{p=1}^{P}\sum_{k=1}^{M}(d_{pk} - y_{pk})^2$$

19. Si $E_{prev} <= E_{min}$, hacer ***Condición _ parar*** = Verdadero, en caso contrario ***Condición _ parar*** = Falsa

20. Si $E < E_{prev}$ hacemos $E_{prev} = E$, en caso contrario hacemos los siguientes ajustes y volvemos al paso 14:
    RESET = verdadero

$$\lambda = \lambda * \rho \quad \text{donde } \rho<1$$

21. Si $E > E_{prev}$ hacemos los siguientes ajustes y volvemos al paso 14.
RESET = falso
$\lambda = \lambda * \sigma$
$\mathbf{H}_{save} = \mathbf{H}$ donde $\sigma > 1$
$\mathbf{G}_{save} = \mathbf{G}$

Consideraciones de diseño

Antes de empezar a utilizar las redes MLP en algunas aplicaciones, muy seguramente nos surgen algunas preguntas: ¿Cómo entrenamos una red? ¿Cómo seleccionamos su arquitectura? ¿Cuántas neuronas definimos en cada capa? ¿Cómo saber que la red aprendió nuestro problema? ¿Cómo garantizar que la solución es satisfactoria? La respuesta a estas preguntas es difícil de dar y como todo en Ingeniería, a lo largo de la experimentación iremos ganando criterios para diseñar correctamente una red neuronal artificial para la solución de un problema específico. De todas maneras, en este apartado revisaremos cada una de estas preguntas y trataremos de dar una respuesta satisfactoria o al menos algunas recomendaciones o sugerencias.

Conjuntos de aprendizaje y de validación

El conjunto de datos que disponemos para describir el fenómeno o problema a solucionar debe ser representativo de toda la información, es decir, debemos evitar que tenga algún tipo de tendencia. En principio, destacamos dos grupos, el primero, que llamamos de entrenamiento y lo utilizaremos para esta primera fase de ejecución del algoritmo de aprendizaje, se suele utilizar entre el 50% y 70% de los datos. En el segundo grupo tenemos los datos que utilizaremos para verificar el grado de aprendizaje. Recomendamos que sea totalmente diferente a los datos utilizados en la fase de entrenamiento. Se suele utilizar entre el 30% y 50% de los datos del conjunto total disponible. Cuando estudiemos los mecanismos para mejorar la capacidad de generalización del aprendizaje de las redes neuronales artificiales, introduciremos un nuevo conjunto denominado datos para validación, el cual será explicado ampliamente.

En la figura 3.4 presentamos un ejemplo de datos de entrenamiento y validación para una aplicación donde la red neuronal aprenderá la dinámica de una función $y = f(t)$. En éste a partir de un conjunto de muestras tomamos las parejas ordenadas (t_p, y_p). Para el entrenamiento tomamos los puntos en azul para el entrenamiento y los puntos en rojo para el proceso de validación del aprendizaje.

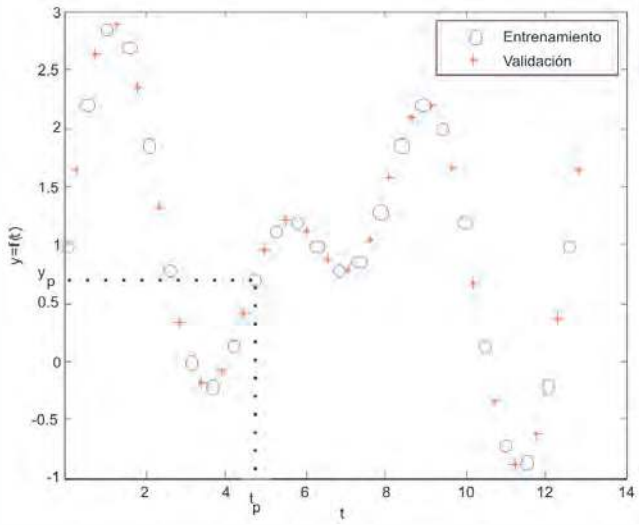


Fig. 3.4. Conjunto de datos de entrenamiento y validación

Dimensión de la red neuronal

Cuando nos enfrentamos en el diseño de la arquitectura de un MLP ante la pregunta de ¿Cuántas neuronas elegir?, la respuesta para las capas de entrada y salida es sencilla ya que el número de neuronas en dichas capas lo definimos con base en el problema a solucionar, pues debe coincidir con la dimensión de los vectores de entrada y salida que lo describen. Por ejemplo, queremos diseñar un clasificador de caracteres numéricos (0 al 9) en código de siete segmentos y salida en código BCD. ¿Cuántas neuronas escoger en la entrada? Para todos es claro que debemos disponer de siete neuronas para saber si se activa o no cada uno de los siete segmentos con los que representamos los dígitos decimales. Para la salida el problema claramente nos define que debemos escoger cuatro neuronas.

La dificultad se centra en la capa oculta, ya que la definición del número de neuronas queda supeditada a la experticia del diseñador. Existen dos tendencias, empezar desde un muy alto número de neuronas y evaluar su funcionamiento y, luego, paulatinamente, ir decreciendo el número de neuronas hasta el instante en el cual, el error de aprendizaje y verificación se mantengan en el mínimo requerido. La otra es empezar por un número pequeño de neuronas en la capa oculta, por ejemplo, igual al 50% de la dimensión del vector de entrada y, depués, empezar a crecer el tamaño de esta capa oculta verificando si su desempeño mejora; llegará un momento en el cual, aunque incrementemos el número de neuronas en la capa oculta, los errores no van a decrecer sustancialmente, consideramos que en este instante hemos llegado a un buen diseño de red. Si este desempeño no es satisfactorio debemos revisar la arquitectura de la red, el algoritmo de aprendizaje o la

calidad de los datos.

Finalmente, surge la pregunta ¿Cuántas capas ocultas debe tener un MLP?, para una gran cantidad de aplicaciones de alta complejidad, tres capas son suficientes, es decir recomendamos utilizar solo una capa oculta. El utilizar más de una capa oculta, aumenta drásticamente la carga computacional de la red, sobretodo en los algoritmos de entrenamiento de segundo orden. En general recomendamos aumentar el tamaño de la capa oculta y no incrementar el número de capas ocultas, por ejemplo, si tenemos una red neuronal con diez neuronas en la capa oculta, computacionalmente es mejor, incrementar a 20 neuronas en esta capa y no generar una nueva capa oculta con diez neuronas más.

Por supuesto, que hay aplicaciones de alta complejidad, como por ejemplo el reconocimiento y clasificación de patrones (reconocimiento y clasificación de imágenes o caracteres manuscritos), en las cuales es muy importante generar superficies de decisión altamente complejas y una segunda capa oculta es muy útil en este caso.

Velocidad de convergencia del algoritmo

El Algoritmo *Backpropagation* encuentra un valor mínimo de error (local o global) navegando con el gradiente descendente por la superficie de error. La velocidad de convergencia se controla con el valor de parámetro **α**, normalmente dicho parámetro debe ser un número pequeño, en el rango de 0.05 a 0.25. Un valor de α muy pequeño trae como consecuencia un aprendizaje seguro, pero puede representar un alto número de iteraciones y tardar mucho el tiempo de ejecución de este algoritmo. En contraposición, un valor de α muy alto, puede generar oscilaciones en el aprendizaje.

Una forma de acelerar la convergencia en el proceso de aprendizaje y tal como lo muestran las ecuaciones 3.47 y 3.48, además de utilizar el valor de los pesos del instante *t*, utilizamos una fracción del valor de los pesos del instante anterior *t-1*, que la controlaremos con el valor del parámetro β. Con esto queremos emular al *momentum* físico considerando la tendencia en el aprendizaje que traía la red en el instante anterior. El valor de β es recomendable tomarlo como positivo y menor que la unidad.

Para los pesos de la capa de salida, se tiene la siguiente expresión:

$$w_{kj}^{o}(t+1) = w_{kj}^{o}(t) + \alpha\delta_{pk}{}^{o} i_{pj} + \beta(w_{kj}^{o}(t) - w_{kj}^{o}(t-1)) \quad (3.47)$$

Para los pesos de la capa oculta se tiene la siguiente expresión:

$$w_{ji}^{h}(t+1) = w_{ji}^{h}(t) + \alpha\delta_{pj}{}^{h} x_{pi} + \beta(w_{ji}^{h}(t) - w_{ji}^{h}(t-1)) \quad (3.48)$$

Funciones de activación

En la mayor cantidad de problemas resueltos con redes MLP, la función de activación a utilizar en las capas ocultas es del tipo sigmoidal, pues le garantiza la capacidad de procesamiento no-lineal. Para la capa de salida, vamos a depender del tipo de problema que estemos solucionando; por ejemplo, en el modelado e identificación de sistemas, en la capa de salida es preferible utilizar una función de activación lineal, con el fin de no saturar la salida de la red, hecho que se presentaría si usamos una función de activación sigmoidal.

Si de lo que se trata es de solucionar problemas de clasificación y reconocimiento de patrones, lo recomendable es utilizar una función de activación sigmoidal en su salida con el fin de garantizar una salida selectiva, similar a una salida binaria.

Pre y pos-procesamiento de datos

Consideremos un problema donde nos proponen diseñar un sistema para estimar el valor del dólar en nuestro país, que naturalmente debe ser solucionado aplicando redes neuronales artificiales. La primera pregunta que nos surge es: ¿Qué parámetros de entrada debemos considerar como datos para entrenar esta red?

Lo usual es que tomemos valores anteriores del valor del dólar al día que vamos a realizar la estimación. Por ejemplo, podríamos tomar los valores correspondientes a dos o tres días anteriores. Pero esto normalmente es insuficiente y la experiencia nos ha mostrado que debemos considerar otros parámetros, entre los que destacamos:

- *Tasa de interés*, es el porcentaje de utilidad que normalmente el sector financiero concede a los diferentes tipos de depósitos que hacen los usuarios con sus ahorros.
- *Masa de dinero disponible*, es el dinero en efectivo disponible en el país para realizar transacciones financieras.

Observemos las magnitudes que tienen los valores que estamos manejando aplicándolas para el caso colombiano. Los valores anteriores del dólar están dados en miles de pesos, por ejemplo $2.400, la tasa de interés actualmente es del orden del 7% (0.07) y la masa de dinero es del orden de miles de millones de pesos.

Si observamos la entrada a la capa oculta, una neurona tendría que sumar estos valores ponderados por los pesos sinápticos. Claramente se ve que el parámetro Masa de Dinero anularía los efectos de los valores anteriores del dólar y estos a su vez anularían el valor de la Tasa de Interés, para evitar esta situación es fundamental normalizar los datos de entrada.

Con esta técnica de pre-procesamiento buscamos que tanto los datos de

entrada como de salida queden en el intervalo [-1, +1]. La expresión 3.49 normaliza un valor *x,* a partir del valor máximo del conjunto de datos x_{max} y del valor mínimo x_{min}.

$$x_n = 2\left(\frac{x - x_{\min}}{x_{\max} - x_{\min}}\right) - 1 \tag{3.49}$$

Otra forma de normalizar los datos es utilizando la técnica de Normalización Gaussiana que viene definida por la ecuación 3.50, donde se busca que los datos de entrada y salida tengan una desviación estándar igual a 1 y el valor medio igual a 0.

$$x_n = \left(\frac{x - \mu}{\sigma}\right) \tag{3.50}$$

donde,

μ : Valor medio de los datos

σ : Desviación estándar de los datos

Luego de entrenar la RNA con los datos normalizado y siguiendo con el ejemplo de la predicción del valor del dólar, la salida de la red tal vez no es fácil de interpretar pues sus valores están en el rango de –1 a +1. Necesariamente debemos volver a los valores originales para cuando la red esté interactuando con el usuario, lo cual lo podemos hacer con la siguiente expresión, donde *x* corresponde al valor en la escala original y x_n es el valor normalizado.

$$x = 0.5(x_n + 1)(x_{\max} - x_{\min}) + x_{\min} \tag{3.51}$$

Regularización

En algunas aplicaciones veremos que las redes neuronales artificiales pueden caer en un problema que se conoce como sobre-entrenamiento, en donde la red no es capaz de responder adecuadamente ante datos de entrada diferentes a los datos que se utilizaron para el proceso de aprendizaje, pero que hacen parte del problema que se quiere solucionar. Visto de otra manera, la red se especializa o "memoriza" un conjunto de datos determinados con un error muy pequeño.

Este sobre-entrenamiento trae como consecuencia que el error de *test* o verificación de la red sea mucho mayor que el de entrenamiento lo cual es indeseable cuando la red está trabajando en la solución de una aplicación específica.

Surge entonces esta técnica, que llamaremos en este libro como Regula-

rización, cuyo objetivo es minimizar el fenómeno del sobre-entrenamiento y por ende sus efectos. El fenómeno del sobre-entrenamiento se hace más evidente cuando los datos de entrada están contaminados con ruido. El objetivo de la regularización es garantizar un adecuado aprendizaje evitando este problema.

En la figura 3.5 observamos un caso típico de sobre-entrenamiento, dado para una aplicación donde la red neuronal artificial debe aproximar una función cuadrática, línea a trazos en la figura, cuyos datos vienen contaminados con ruido. En el ejemplo, hemos escogido una red neuronal con muchas neuronas en la capa oculta, con el fin de minimizar el error de aprendizaje; como era de esperarse, la salida de la red, representada en línea sólida, intenta ajustarse a cada uno de los puntos del conjunto de aprendizaje, situación muy distante de la salida deseada que corresponde a la línea a trazos.

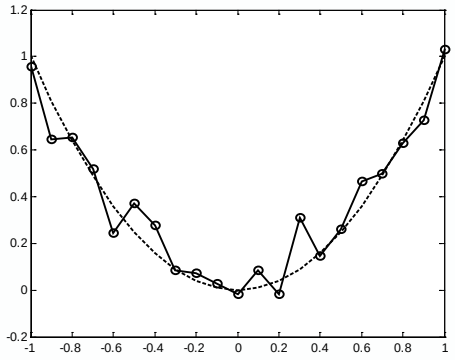


Fig. 3.5 Sobre-entrenamiento en una Red Neuronal Artificial

La solución obvia que surge para eliminar el sobre-entrenamiento es bajar la complejidad del modelo de red neuronal, reduciendo el número de neuronas en la capa oculta. Esta solución no siempre es la mejor, pues podemos caer en le fenómeno opuesto que hemos denominado sub-entrenamiento, tal como se presenta en la figura 3.6. En este caso, la salida de la red, línea sólida, trata de ajustarse a los datos de entrada a través de un modelo lineal, lo cual claramente es inadecuado.

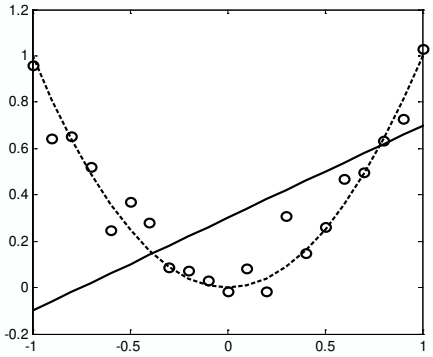


Fig. 3.6 Sub-entrenamiento en una Red Neuronal Artificial

En este libro plantearemos dos técnicas de regularización que buscan solucionar este problema:

- Regularización por parada temprana.
- Regularización por limitación de la magnitud de los pesos.

Regularización por parada temprana

Cuando definimos los conjuntos de datos para el proceso de aprendizaje de la red, habíamos mencionado dos, uno para aprendizaje y otro para *test* o verificación. Con esta técnica surge un nuevo conjunto de datos que denominaremos conjunto de datos de validación, el cual utilizaremos en la fase de entrenamiento, pero que no se toma en cuenta para la modificación de los pesos sinápticos de la red. Normalmente, dejamos entre un 20 y 30 por ciento de los datos para esta actividad de validación del proceso de aprendizaje.

En el proceso de aprendizaje ahora definiremos dos tipos de error, el primero es el de aprendizaje, cuya evolución la mostramos en la figura 3.7 con la línea azul y se define como la diferencia entre la salida de la red y el valor deseado, y este error es el que utilizamos para modificar los pesos de la red.

Con esta técnica, surge ahora, un nuevo error que llamamos de validación, cuya evolución se muestra con la línea a trazos y se define de igual manera, como la diferencia entre la salida de la red y el valor deseado, pero evaluado en un dato de entrada que pertenece al nuevo conjunto de datos de validación. Es importante aclarar que este error no lo consideraremos para la modificar los pesos sinápticos de la red, por lo que en este momento estamos observando el comportamiento de la red frente a nuevos datos que pertenecen al conjunto universo del problema, pero diferentes a los utilizados en el entrenamiento.

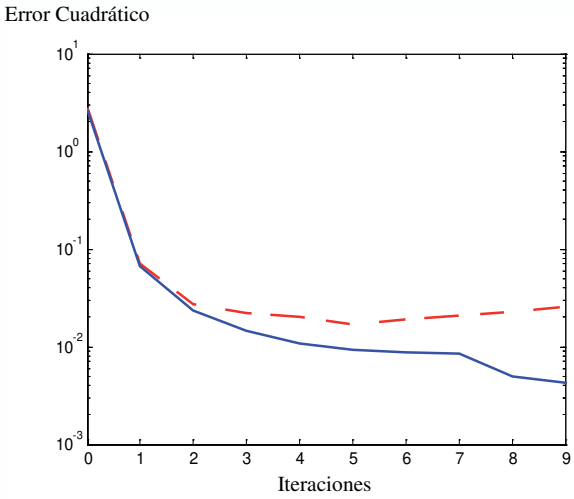


Fig. 3.7 Regularización por Parada Temprana

En la gráfica observamos que si seguimos el proceso con la tendencia marcada por la línea sólida, caeremos en un sobre-entrenamiento puesto que el error seguirá bajando pero para los datos del conjunto de entrenamiento; sin embargo, la línea a trazos nos muestra que para el conjunto de datos de validación, el error tiene una tendencia creciente.

La solución es simple y la técnica nos recomienda parar el proceso de entrenamiento de la red justo antes de que el error de validación empiece con una tendencia ascendente, por esta razón, le llamamos "Regularización por Parada Temprana", debido a que evitamos que el proceso de aprendizaje continúe bajando excesivamente el error y para ello tomamos como criterio de finalización la tendencia del error de validación.

Es bueno aclarar, que al finalizar este proceso, continuamos con la prueba final de desempeño de la red, para lo cual usamos los datos de verificación.

Regularización por limitación de la magnitud de los pesos

La experiencia nos ha mostrado que podemos garantizar una función de salida suave si logramos mantener los pesos sinápticos de la red en unos valores relativamente pequeños. Con el fin de limitar la magnitud de estos pesos redefinimos el cálculo del error así:

$$E_R = \gamma E_D + \lambda E_W \tag{3.52}$$

En la ecuación 3.52, el error se calcula con dos términos, por lo que utilizamos una nueva notación y llamaremos E_R al Error Regularizado. El primer término, ecuación 3.53, corresponde al error cuadrático promedio

que es la forma como tradicionalmente hemos estimado el error de entrenamiento, pero afectado por el parámetro γ.

$$E_D = \frac{1}{2P}\sum_{p=1}^{P}\sum_{k=1}^{M}(d_{pk} - y_{pk})^2 \tag{3.53}$$

El segundo término utilizado para estimar el error de aprendizaje regularizado, introduce la sumatoria de los pesos sinápticos de la red, ecuación 3.54, ponderados con el parámetro λ. Ya que el algoritmo de aprendizaje tiende a minimizar E_R, el resultado final es que la sumatoria de pesos igualmente tiende a ser minimizada para garantizar una suavidad en la salida de la red.

$$E_W = \sum_{n=1}^{N} w_n^2 \tag{3.54}$$

APROXIMACIÓN PRÁCTICA

Luego de haber introducido los diferentes métodos de aprendizaje aplicados al Perceptron Multicapa, proponemos una serie de proyectos cuyo seguimiento nos permitirá llevar a cabo las siguientes actividades:

- Simular redes neuronales tipo MLP con MATLAB®
- Implementar redes neuronales tipo MLP con MATLAB®
- Validar redes neuronales entrenadas con MATLAB®
- Diseñar y simular redes neuronales tipo MLP con UV-SRNA

Solución del problema de la función xor con MATLAB®

Una de las mayores dificultades que presenta el Perceptron es el no poder solucionar problemas no – lineales, razón por la cual cayeron en desuso las redes neuronales artificiales. En este capítulo hemos afirmado que al MLP solventa esta deficiencia, para evaluar esta potencialidad de las redes MLP, ejecutemos el siguiente programa escrito para MATLAB® donde se aplica el MLP en la solución de la función XOR. El programa al final nos entrega la superficie de separación no lineal cuando entrenamos una red MLP y una de las salidas la vemos en la figura 3.9.

```
function xorback(metodo,n)
% xorback.m
% Entrenamiento de una red neuronal MLP para resolver el problema de la XOR
% Además se construye la superficie de separación que genera la red entrenada
% Argumentos:
% metodo = método de aprendizaje
%          1 = Gradiente descendente
%          2 = Gradiente descendente con β variable
%          3 = Gradiente descendente con momentum
%          4 = Gradiente descendente con momentum y α variable
% n = Número de iteraciones
% Ejemplo:
%          xorback(1,50)
%          Se realizan 50 iteraciones de entrenamiento con el método del
%          gradiente descendente

close all;
x=[0 0 1 1;
   0 1 0 1];
y=[0 1 1 0];

 switch metodo
 case 1

 met_ent='traingd'
 case 2
 met_ent='traingda'
 case 3
 met_ent='traingdm'
 case 4
 met_ent='traingdx'
 otherwise
 met_ent='traingda'
 end

red=newff(minmax(x),[10 1],{'tansig','purelin'},met_ent);
red.trainParam.epochs=n;
figure;
red=train (red,x,y);
button = questdlg('Ver Superficie?',...
```

```
'Ver Superficie','Si','No','No');
if strcmp (button,'Si')

figure;hold on;axis([-0.2 1.2 -0.2 1.2]),title('Azul=Salida en Cero
Rojo=Salida en Uno')
plot(x(1,1),x(2,1),'ob','LineWidth',7);
plot(x(1,4),x(2,4),'ob','LineWidth',7);
plot(x(1,2:3),x(2,2:3),'or','LineWidth',7);

for i=-0.1:0.1:1.1
for j=-0.1:0.1:1.1
yred=sim(red,[i;j]);
if yred>0.7
plot(i,j,'or');
end;
if yred<0.3
plot(i,j,'ob');
end;
end;
end;
hold off

elseif strcmp(button,'No')
disp ('El programa ha terminado')
end
```

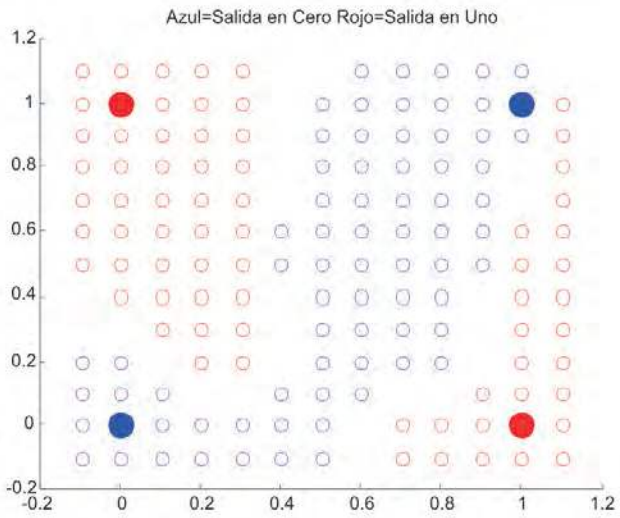


Fig. 3.9 Separación lograda por una red MLP

Aprendizaje de una función seno con MATLAB®

Las redes MLP son consideradas aproximadores universales de funciones, ya que es posible entrenar una red de este tipo para realizar una transformación matemática de un espacio *n-dimensional* a uno *m-dimensional* con un grado de precisión predefinido.

Como ejemplo de esta característica, con el programa en MATLAB® que presentamos en el recuadro, entrenaremos una red MLP que está en capacidad de aprender la dinámica de una función seno ($y = sen\ x$). Para ilustrar este ejemplo utilizaremos una red MLP de tres capas, como la presentada en la figura 3.10, es decir una neurona en la capa de entrada, un número de neuronas en la capa oculta que define el usuario y una neurona en la capa de salida. El resultado del aprendizaje de la función lo presentamos en la figura 3.11, donde observamos que la salida de la red neuronal sigue adecuadamente la dinámica de la función seno de entrada.

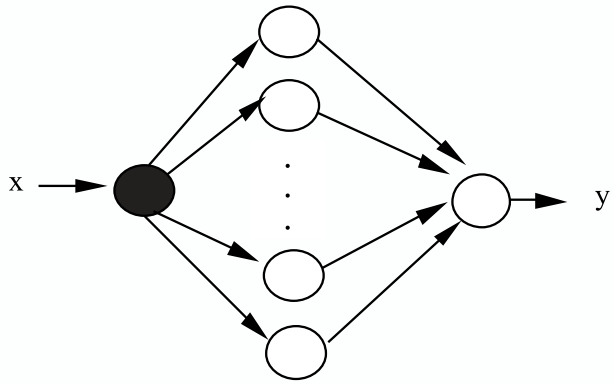


Fig. 3.10 Estructura de la red a entrenar

```
function senoback(metodo,nco,n)
% senoback.m
% Entrenamiento de una red neuronal MLP para aprender una fun-
ción       seno
% Argumentos:
% metodo = método de aprendizaje
%          1 = Gradiente descendente
%          2 = Gradiente descendente con eta variable
%          3 = Gradiente descendente con momentum
%          4 = Gradiente descendente con momentum y eta variable
% nco = Número de Neuronas en la capa oculta
% n = Número de iteraciones
% Ejemplo:
%          senoback(1,10,50)
```

```
%    Se realizan 50 iteraciones de entrenamiento con el método del
%    gradiente descendente a una red de 10 neuronas en la capa
     oculta

close all;
x=0: pi/10:2*pi;
y=sin(x);

switch metodo
case 1
met_ent='traingd'
case 2
met_ent='traingda'
case 3
met_ent='traingdm'
case 4
met_ent='traingdx'
otherwise
met_ent='traingda'
end

red=newff(minmax(x),[nco 1],{'tansig','purelin'},met_ent);
yred=sim(red,x);
figure; hold on;
plot(x,y,'+b');plot(x,yred,'or');title('Situación Inicial (+=Seno
original o=Salida de la red)')
red.trainParam.epochs=n;
figure;
red=train(red,x,y); yred=sim(red,x);
figure, hold on;
plot(x,y,'+b'); plot(x,yred,'or');title('Situación Final (+=Seno
original o=Salida de la red)')
```

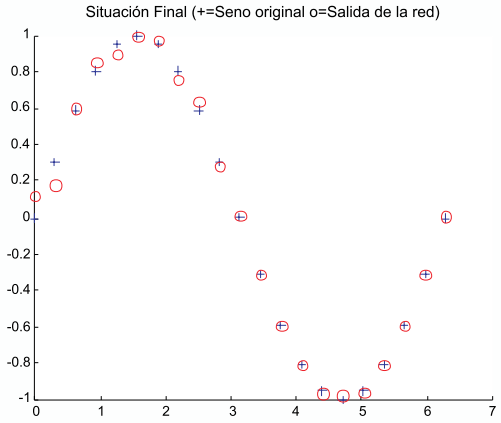


Fig. 3.11 Situación del aprendizaje de la red

Aprendizaje de la función silla de montar con MATLAB®

En este nuevo proyecto, evaluaremos la capacidad que tiene una red MLP para aprender funciones de dos variables, para lo cual tomamos como ejemplo la función Silla de Montar definida por la ecuación 3.55 y cuya superficie resultante la mostramos en la figura 3.12.

$$z = f(x, y) = x^2 - y^2 \tag{3.55}$$

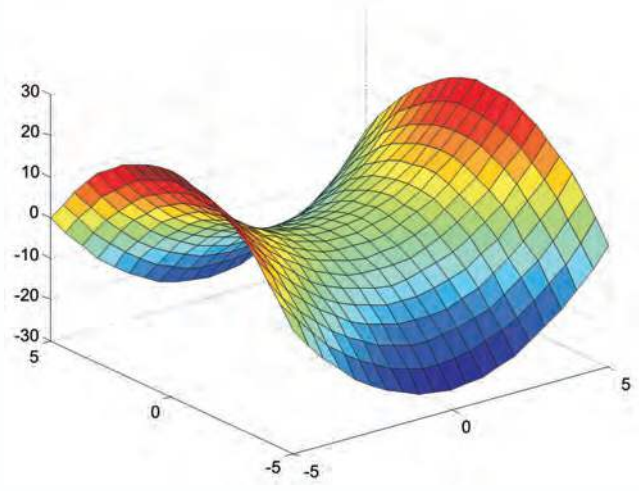


Fig. 3.12 Superficie original

En este caso la red neuronal que utilizaremos para el entrenamiento de esta función tiene la estructura de la figura 3.13 con dos neuronas de entrada, una capa oculta cuya dimensión la define el usuario como parámetro de entrada del programa del recuadro y una neurona de salida.

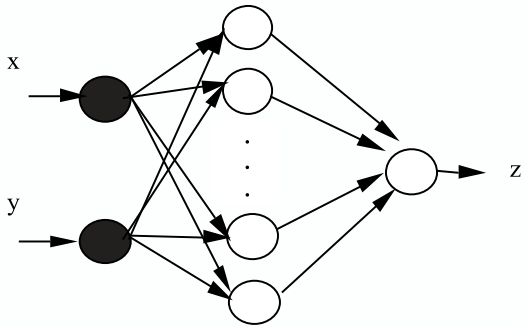


Fig. 3.13 Estructura de la red a entrenar

En las figuras 3.14 a 3.17, observamos la evolución del algoritmo de aprendizaje de la superficie definida. En la figura 3.14, vemos la salida de la red cuando inicia su proceso de entrenamiento por lo que el nivel de acercamiento a la función objetivo de silla de montar es nulo. En la figura 3.15, tomamos un punto intermedio en el proceso y notamos una clara mejoría en la aproximación a la función que hace la red neuronal. Observemos en la misma figura, con círculos representamos los patrones de entrenamiento de la función silla de montar cuya forma queremos aprender con la red, con asteriscos representamos la salida de la red en este instante de aprendizaje. Finalmente, en la figura 3.16, presentamos la salida de la red en formato de superficie, luego de 200 iteraciones, donde se observa que la superficie aprendida de la red es prácticamente igual función objetivo. En la figura 3.17, podemos apreciar en detalle la similitud de las dos funciones, tanto la objetivo cuyos patrones de aprendizaje los representamos con círculos, como la de salida de la red que la representamos con asteriscos.

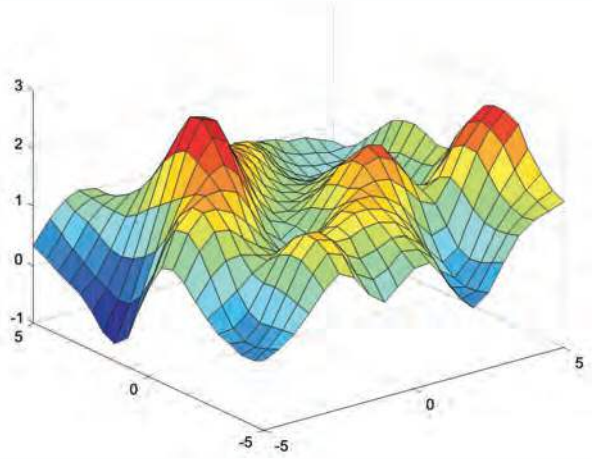


Fig. 3.14 Superficie aprendida por la red al inicio del aprendizaje

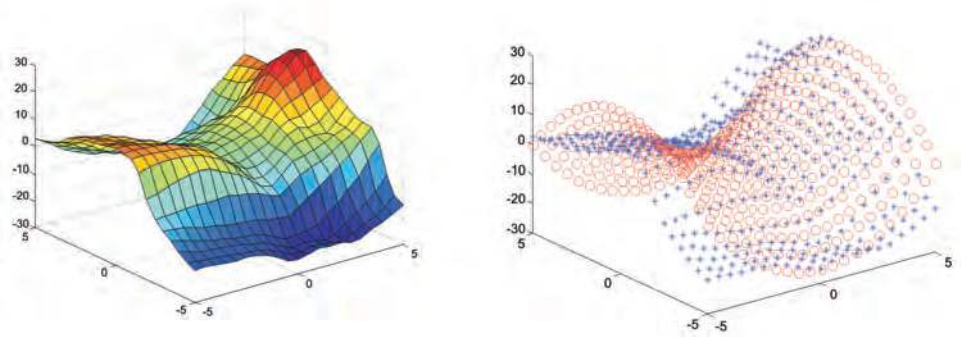


Fig. 3.15 Superficie aprendida por la red en una etapa intermedia del aprendizaje

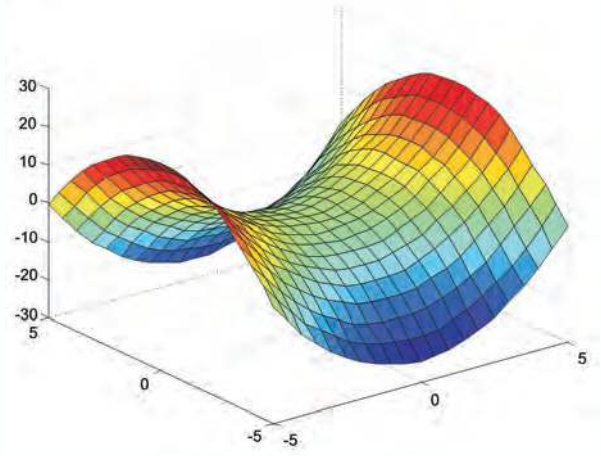


Fig. 3.16 Superficie aprendida por la red al fin del aprendizaje

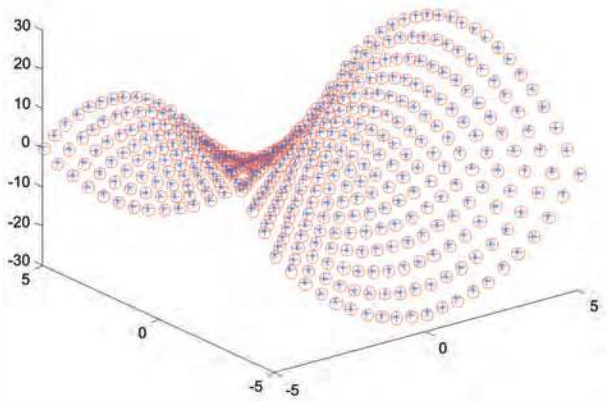


Fig. 3.17 Relación entre los patrones de aprendizaje (círculos) y la salida de la red (asteriscos)

A continuación presentamos el código en MATLAB® que permite implementar la aproximación de la función Silla de Montar usando una red neuronal tipo MLP.

```
%AproxSillaMontar.m
%Programa que permite aprender la función Silla de Montar.
close all;
x=-5:0.5:5;
y=x;
[X,Y]=meshgrid(x,y);
Z=X.^2-Y.^2;
figure;
surf(X,Y,Z);
%%
[f,c]=size(X);
N=f*c;
Xred=[reshape(X,1,N);
        reshape(Y,1,N)];
Zdred=[reshape(Z,1,N)];
red=newff(minmax(Xred),[20 1],{'tansig','purelin'},'trainlm');
red.trainparam.epochs=20;
Zredini=sim(red,Xred);
Zredinigra=reshape(Zredini,f,c);
figure;
surf(X,Y,Zredinigra)
%%
```

```
red=train(red,Xred,Zdred);
Zredfin=sim(red,Xred);
Zredfingra=reshape(Zredfin,f,c);
figure;
surf(X,Y,Zredfingra)
figure
plot3(reshape(X,1,N),reshape(Y,1,N),reshape(Z,1,N),'or');
hold on
plot3(reshape(X,1,N),reshape(Y,1,N),Zredfin,'*b');
```

Solución del problema de la xor con uv-srna

La herramienta de simulación UV-SRNA posee un módulo con el que se pueden entrenar redes tipo MLP con el algoritmo de gradiente descendente (Backpropagation), cuya interfaz la mostramos en la figura 3.18.

Fig. 3.18 Módulo Para Entrenar Redes MLP con un Algoritmo Gradiente Descendente en UV-SRNA

Si deseamos resolver el problema de la *XOR* con UV-SRNA el primer paso es crear un archivo texto con los patrones de dicha función, tal como lo mostramos en la Tabla 3.1.

Tabla 3.1 Codificación de los patrones de entrenamiento para la función lógica XOR

Datos en el archivo	*Significado*
4	Número de patrones de entrenamiento
2	Número de entradas de cada patrón
1	Número de salidas de cada patrón
0 0 0	Patrón No. 1
0 1 1	Patrón No. 2
1 0 1	Patrón No. 3
1 1 0	Patrón No. 4

Una vez creado el archivo de patrones, lo debemos cargar desde UV-SRNA, usando la opción *leer patrones de entrada* del menú de *archivo.*

Con los patrones disponibles procedemos a inicializar la red, para lo cual solo es necesario oprimir el botón de *inicializar*. Luego, procedemos a su entrenamiento oprimiendo el botón *entrenar*. Podemos observar como evoluciona el error de entrenamiento y el proceso se dará por terminado cuando el error es menor que el error mínimo aceptado. La aplicación muestra un ventana donde se visualiza la manera como cambia el error de entrenamiento, figura 3,19

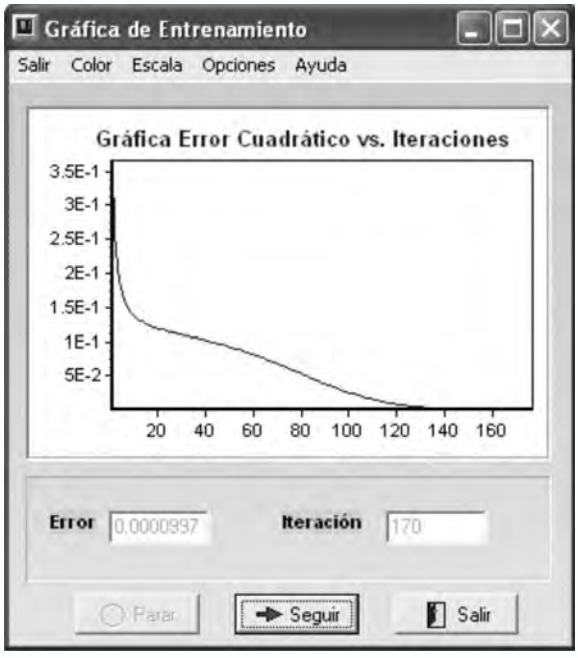


Fig. 3.19 Evolución del error de entrenamiento en UVSRNA

Una vez alcanzado el error mínimo, procedemos a validar la red, para lo cual la herramienta de simulación UV-SRNA nos provee dos formas, una por teclado y la otra por archivo.

Validación por teclado

Al seleccionar el *item validar la red por teclado* del menú *validar* se despliega una ventana como la mostrada en la figura 3.20, en ella el usuario digita unas posibles entradas para la red y usando el botón *probar* genere la salida ante dichas entradas.

Fig. 3.20 Ventana de Validación por teclado en UVSRNA

Validación por archivo

Esta opción se invoca al seleccionar la opción *validar la red por archivo* del menú validar. Para usar esta posibilidad es necesario haber creado un archivo de validación (extensión **.val*) con las entradas que se desean probar en la red neuronal. Básicamente un archivo de validación es un archivo de patrones pero sin la información de la salida deseada.

Tabla 3.2 Codificación de un archivo de validación para la función lógica XOR

Datos en el archivo	*Significado*
4	Número de patrones de validación
2	Número de entradas de cada patrón
0 0	Entrada a evaluar No. 1
0 1	Entrada a evaluar No. 2
1 0	Entrada a evaluar No. 3
1 1	Entrada a evaluar No. 4

UV-SRNA recibe este archivo y la salida generada la guarda en un archivo de salida *(*.sal*). El programa pregunta los nombres tanto del archivo de validación a usar como del archivo de salida a generar. Como el archivo de salida es un archivo texto, se puede abrir en un editor de texto cualquiera. Un ejemplo de un archivo de salida se muestra en la figura 3.21.

Fig. 3.21 Archivo de salida generado por UVSRNA

Identificación de sistemas usando un MLP

En este proyecto asumiremos un reto muy interesante que se soporta en la propiedad de las redes neuronales de aproximar una función con una

precisión predefinida, el objetivo es realizar la identificación de un sistema a partir de datos experimentales de entrada y salida, que utilizaremos como datos de entrenamiento de la red. En general, un sistema dinámico lo podemos describir como una función *f* que procesa los valores de la entrada y salida del sistema en instantes anteriores para proporcionar el muestreo en el instante *k*, es decir:

$$y(k) = f\left(y(k-1), y(k-2), \ldots, y(k-n), u(k-1), u(k-2), \ldots, u(k-m)\right) \quad (3.56)$$

El problema se reduce a entrenar una red neuronal que aprenda la función *f*, tal como se esquematiza en la figura 3.22, donde la red neuronal hace la estimación de la salida en el instante *k* con base en valores anteriores de la entrada y salida del sistema.

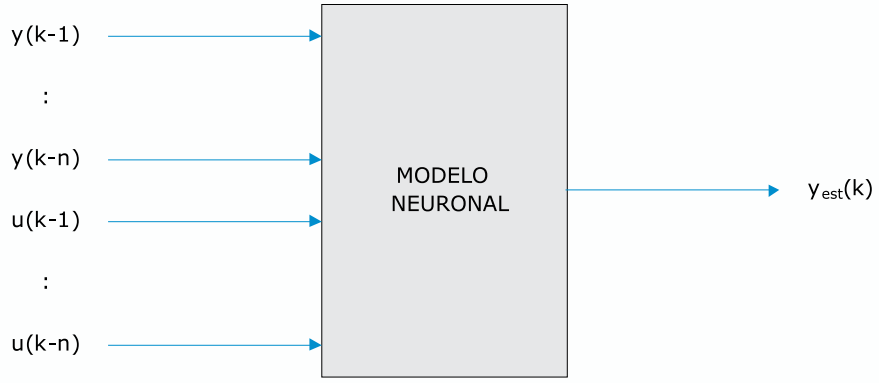


Fig. 3.22 Red Neuronal Usada para Identificar un Sistema Dinámico o Planta

Como un primer ejercicio, identifiquemos una planta conocida de primer orden que responde al modelo representado con la función de transferencia de la ecuación 3.57.

$$G(s) = \frac{1}{(s+2)} \quad (3.57)$$

Diseño del experimento y muestreo de datos

El primer paso es definir previamente el rango en el que se desea identificar la planta, para nuestro caso se diseñará un ejemplo que permita identificar el proceso propuesto en un rango de entrada entre 0 y 1.

Los datos experimentales se obtienen por simulación, suministrándole a la planta una entrada que cubre todo el rango de la entrada propuesto, esta simulación se realiza en la herramienta *Simulink* de MATLAB® con el esquema mostrado en la figura 3.22.

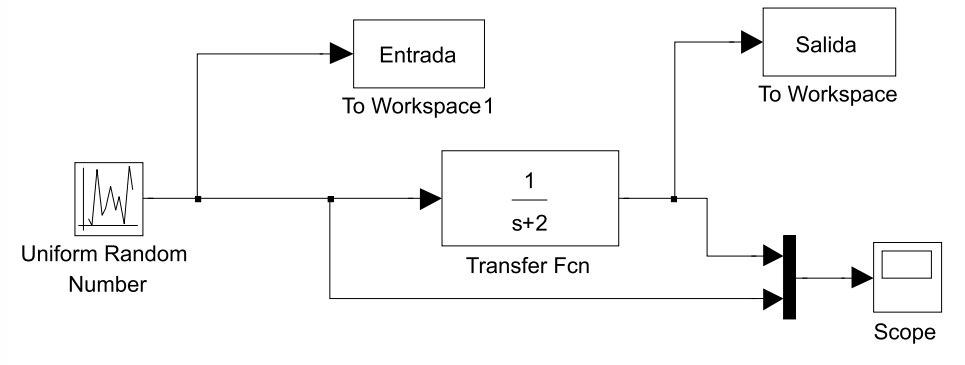


Fig. 3.23 Esquema en Simulink del experimento

El bloque de entrada es un bloque tipo *Uniform Random Number* que se encuentra en la biblioteca de *sources*.

Debemos ajustar los parámetros de estos bloques de manera adecuada, de tal manera que obtengamos datos de entrada y salida como los de la figura 3.24.

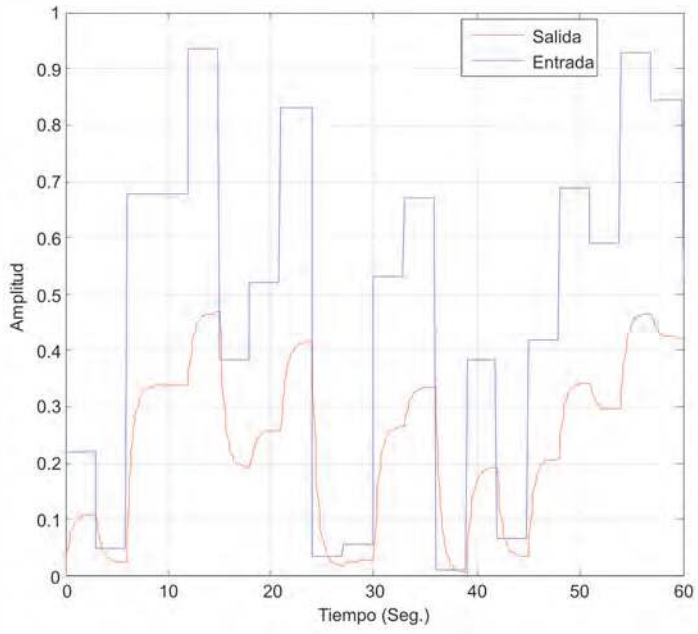


Fig. 3.24 Una posible entrada y su salida obtenida en el experimento

Recomendaciones:

- El tiempo de muestreo lo debemos ajustar entre 0.1 y 0.2 veces la constante de tiempo más rápida del sistema.
- El ancho de los escalones de entrada debe ser aproximadamente el tiempo de estabilización de la planta. Esto lo configuramos con el bloque de entrada usando el parámetro *Simple time.* Tener en cuenta que este parámetro no se debe interpretar como el tiempo de muestro que se configura en los bloques to workspace de la librería Sinks.
- La entrada del experimento debe cubrir todo el rango posible de valores de la variable de entrada del proceso.
- Por lo general la entrada del experimento se diseña para que cubra el rango de la entrada del proceso ± el 20%.

Modelo a usar y estimación de parámetros (entrenamiento de la red)

Con los datos experimentales, procedemos a definir una arquitectura de red neuronal que sirve para identificar la planta en cuestión. Como la planta es de primer orden y al usar un modelo ARX, se necesitan como entrada de la Red Neuronal un retardo de la entrada y un retardo de la salida para obtener la salida actual; esto se puede expresar en la ecuación 3.58.

$$y(k) = f(u(k - 1), y(k - 1))) \tag{3.58}$$

Lo anterior nos lleva a la siguiente arquitectura de Red Neuronal.

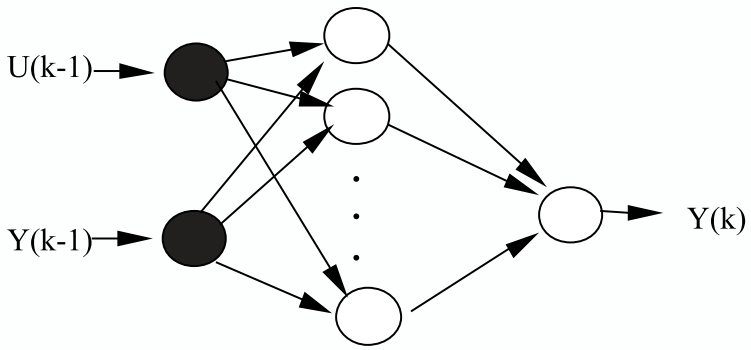


Fig. 3.25 Estructura de la RNA utilizada para la identificación

Como vemos en la figura 3.25, la red tiene dos neuronas en la capa de entrada, en la capa oculta el número de neuronas la define el usuario en la aplicación software que entregamos en el recuadro y una neurona en la capa de salida.

El paso siguiente, es organizar los datos que se obtuvieron durante el

experimento y definir los patrones de entrenamiento a utilizar en el proceso de aprendizaje de la Red Neuronal Artificial. Esto lo llevamos a cabo construyendo un programa **.m* en MATLAB®, el cual lo denominaremos *entrenar.m* y aparte de organizar los vectores para obtener las entrada que necesita la red neuronal, también realiza el entrenamiento de la red neuronal.

Creemos el archivo *entrenar.m* usando el código MATLAB® que se muestra a continuación.

```
% Inicio del archivo entrenar.m

% Entrenamiento de una red MLP para la identificación.
% de una planta lineal de primer orden.
% U debe contener los datos de entrada a la planta.
% Y debe contener los datos de salida a la planta.
% U, Y deben ser vectores columna.

% En el modelo neuronal que utilizamos, la planta se considera de
primer orden.
% esto significa que la salida actual depende de una muestra ante-
rior de la entrada % y de dos muestras anteriores de la salida así que
los datos de entrada-salida.
% deben llevarse a la forma:

%|  Entrada1 |Entrada2  |Salida |
%==========================================
===
%| U(1 )      |  Y(1)     | Y(2)  |
%| U(2 )      |  Y(2)     | Y(3)  |
%| ..................................................................
%|   U(K-1)   |  Y(K-1)  | Y(K) |
%
% Se pierde el último dato del vector de entrada U
U=Entrada;
Y=Salida;
No_Datos = length(U);
% Primera entrada de la RNA
IN_RNA_1 = [U(1:No_Datos-1)];
% Segunda entrada de la RNA
IN_RNA_2 = [Y(1:No_Datos-1)];
% Salida de la RNA
OUT_RNA  = [Y(2:No_Datos)];
```

```
% Definición de la entrada de los patrones usados para el entrenamiento
P = [IN_RNA_1 IN_RNA_2];

% Cada Columna debe representar un vector de entrenamiento
% Luego P y OUT_RNA deben tener la misma cantidad de columnas
% Para lograr esto se trasponen los vectores tal y como se han definido.

X = P';
YD = OUT_RNA';

% Inicialización de la Red Neuronal.

% Una capa oculta de 10 neuronas con función de activación tangente sigmoidal.
% Una capa de salida de 1 neurona con función de activación lineal.

Limits = minmax(X);
red = newff(Limits,[10 1],{'tansig' , 'purelin'},'trainlm');

%Ahora se procederá a entrenar la RNA.
% Parámetros de Entrenamiento:
% Frecuencia de visualización del progreso del entrenamiento (en iteraciones).
red.trainParam.show = 20;
% Número máximo de iteraciones de entrenamiento
red.trainParam.epochs = 100;

% Error cuadrático promedio mínimo deseado.
red.trainParam.goal = (0.000000001);

% Llamada a la función de entrenamiento
red = train(red,X,YD);

% Fin del archivo entrenar.m
```

Validación del modelo obtenido con la rna

Luego de procesar el entrenamiento, se tiene en el objeto red una red neuronal que ha aprendido la dinámica de la planta, para validar este modelo se realiza un esquema en la herramienta *Simulink* de MATLAB® como el mostrado en la figura 3.26.

Primero debe obtenerse una representación en diagramas de bloques de la red neuronal que se ha acabado de entrenar, para esto se utiliza el comando *gensim* indicando el tiempo de muestreo usado para la obtención de los datos.

```
>> gensim(red,Ts)
```

donde,

red : Objeto de la red que se ha entrenado
Ts : Tiempo de muestreo seleccionado

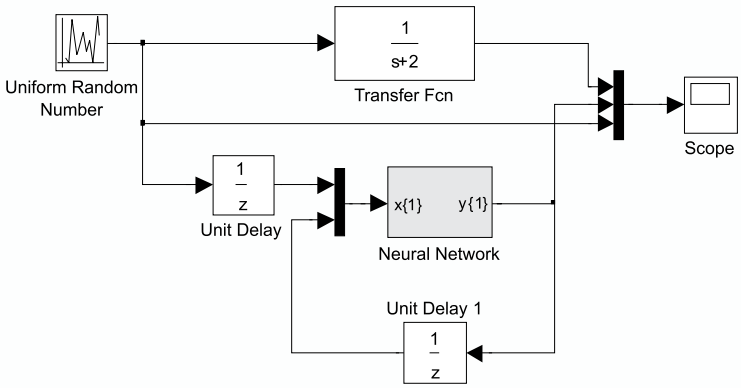


Fig. 3.26 Esquema en Simulink para validar el modelo neuronal

Cuando llevamos a cabo la simulación del sistema en Simulink, el programa nos entrega una salida como la mostrada en la figura 3.27.

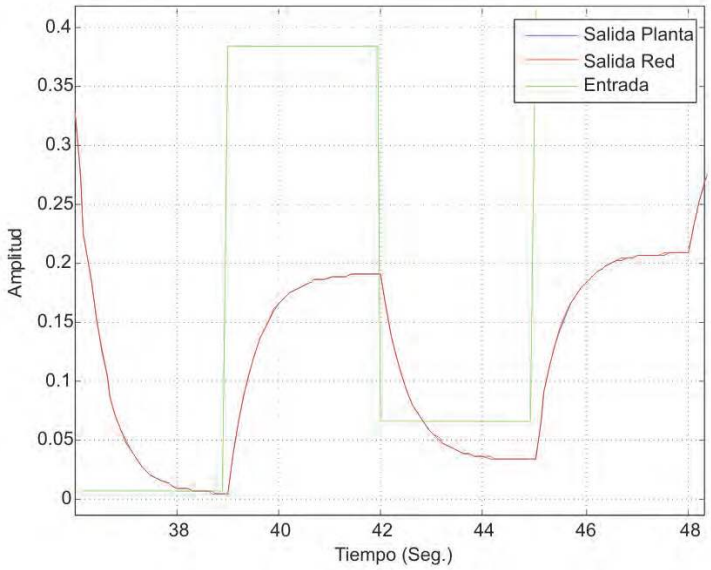


Fig. 3.27 Validación del modelo neuronal obtenido

Como podemos observar la salida de la red sigue plenamente la salida de la planta original, por lo que aseguramos que la red neuronal tipo MLP aprendió o identificó la dinámica de la planta.

Pronóstico de consumo de energía (demanda)

Un nuevo tipo de aplicación de las redes neuronales tipo MLP es en el pronóstico de series temporales, que ilustraremos un ejemplo de pronóstico de consumo de ener-gía para tres tipos de usuarios. En la Tabla 3.3 presentamos los consumos en KWH en diferentes horas del día de tres tipos básicos de consumidores de energía eléctrica; el objetivo es entrenar una red neuronal que permita pronosticar el consumo en KWH en cualquier hora del día, utilizando el simulador de redes neuronales de la Universidad del Valle, UV-SRNA.

Tabla 3.3 Demanda en KWH a diferentes horas del día

DEMANDA EN KW			
	Residencial	Comercial	Industrial
0: 00 p.m.	780	1650	5100
2: 00 a.m.	610	990	4750
4: 00 a.m.	590	990	4700
6: 00 a.m.	580	990	4850
8: 00 a.m.	710	2100	8500
10:00 a.m.	770	2990	9800
12:00 m.	790	3950	9550
14:00 p.m.	780	3900	8950
16:00 p.m.	810	3950	7450
18:00 p.m.	1450	3950	7790
20:00 p.m.	1995	3590	9980
22:00 p.m.	1380	2380	6450

Para el pronóstico proponemos una red con una neurona en la capa de entrada (la hora del día) y tres neuronas en la capa de salida para generar el consumo en los diferentes sectores.

Lo primero que tenemos que hacer es crear un archivo de patrones con la información dada en la tabla 3.3, es fundamental mencionar que debido a la naturaleza de la información es conveniente normalizar los datos de entrada y salida para que se ajusten en el rango entre 0 y 1. Para normalizar la entrada dividimos la hora entre 40, lo cual significa que en vez de pasarle a la red el 12 para indicar el medio día se le pasará (12/40) = 0.3 para representar este valor; la salida se normalizará dividiendo todos los valores entre 100, por lo tanto la red no entregará el valor 5000 sino 50. El procedimiento acabado de mencionar es lo que se conoce como escalamiento de los datos.

El archivo de patrones debe quedar como se muestra en la figura 3.28.

Fig. 3.28 Archivos de patrones

Teniendo el archivo de patrones cargado, se define una red con un capa oculta de 20 neuronas, la clave para lograr entrenar la red con este conjunto de datos es variar α cuando se perciba que el error de la red comienza a oscilar. Por ejemplo, con el α por defecto que tiene el simulador se obtuvo la curva de error mostrada en la figura 3.29.

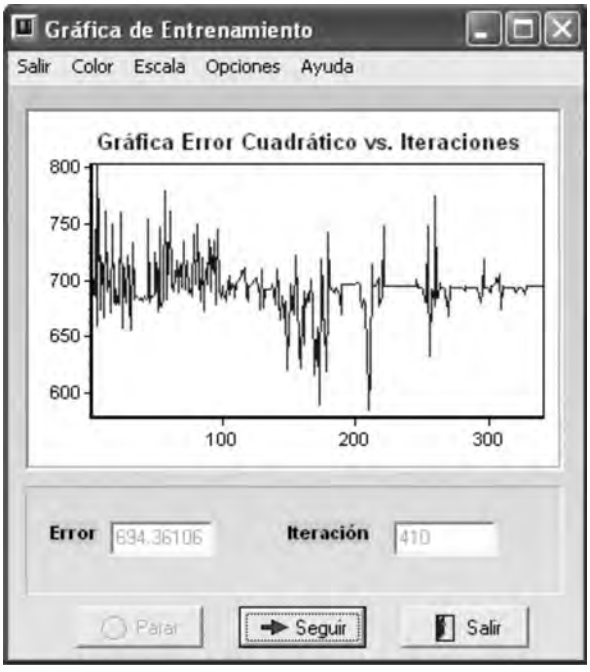


Fig. 3.29 Curva del error cuando $\alpha = 0.075$ $\beta = 0.1$

Para evitar esta oscilación detenemos el aprendizaje y hacemos $\alpha = 0.005$, el resultado mejora ostensiblemente como se observa en la figura 3.30.

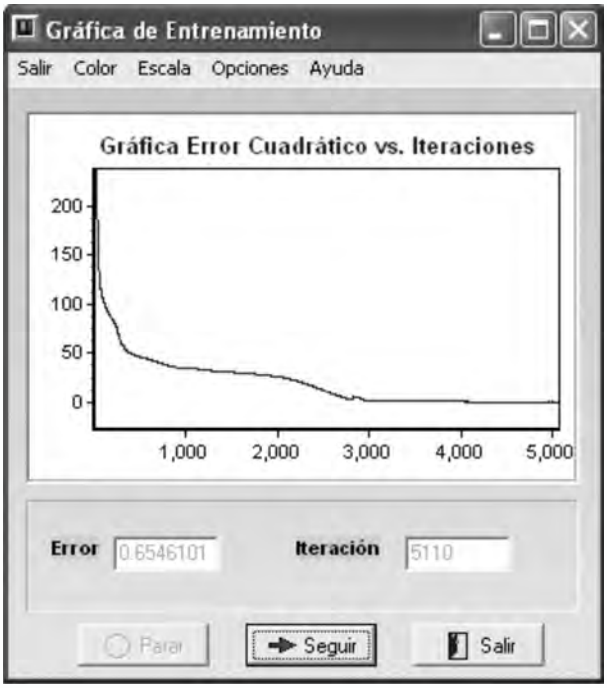


Fig. 3.30 Curva del error cuando α = 0.005 β= 0.1

Es evidente que con este nuevo de α la red comienza a disminuir el error de una manera notoria. Siguiendo el entrenamiento y teniendo en cuenta lo acabado de mencionar, valide la red, para tal fin cree el archivo de validación respectivo y use la opción de validar la red por archivo, con lo cual se genera un archivo de resultados como el mostrado en la figura 3.31.

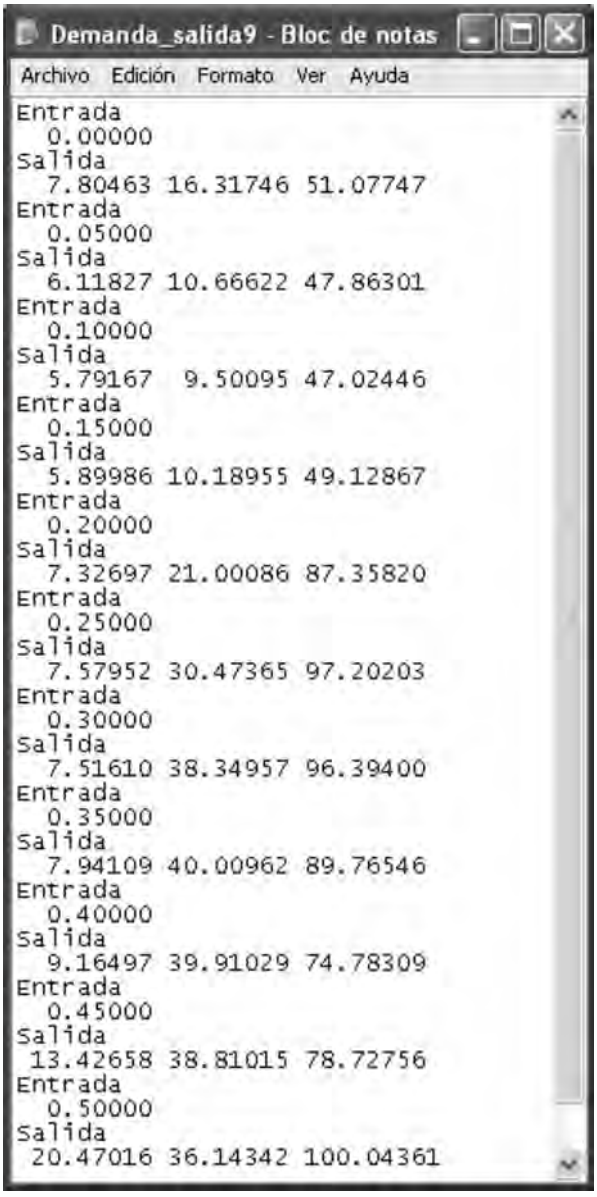


Fig. 3.31 Archivo de salida o resultados

Es necesario verificar si el entrenamiento de la red es el adecuado, con el fin de validar la red en el pronóstico de la demanda en cualquier hora, por ejemplo si deseamos saber cual es la demanda a las 23 horas escalamos este valor (17/40)=0.425, lo introducimos a la red y ésta nos genera los siguientes valores:

- 10.2335
- 39.6140
- 70.5042

Si ajustamos a la escala de salida se tiene que el pronóstico de consumo para dicha hora es:

- 1023 KWH para el sector residencial
- 13961 KWH para el sector comercial
- 7050 KWH para el sector industrial

Aplicación a la clasificación de patrones (el problema del IRIS)

El problema del IRIS es uno de los problemas mejor conocidos en la literatura de reconocimiento de patrones. El problema está definido por tres clases de especies de iris: *iris setosa canadensis, iris versicolor* y *iris virginica.* En la base de datos cada clase posee 50 ejemplos. Las características usadas para realizar la clasificación son cuatro: la longitud del sépalo en centímetros, el ancho del sépalo en centímetros, la longitud del pétalo en centímetros y el ancho del pétalo en centímetros.

El siguiente código en MATLAB® podemos entrenar una red neuronal tipo MLP con aprendizaje Levenberg-Marquardt. De los 50 ejemplos de cada una de las clases se seleccionamos el 70% de los datos para la etapa de entrenamiento, y el 30% para la validación.

```
% Ejemplo de clasificación de patrones usando la base de datos
IRIS
% El 70% de los datos los usamos para el entrenamiento
% y el 30% para validación

load iris_data2.txt;

Datos_train_C1=iris_data2(1:35,1:4);
Datos_val_C1=iris_data2(36:50,1:4);

Datos_train_C2=iris_data2(51:85,1:4);
Datos_val_C2=iris_data2(86:100,1:4);

Datos_train_C3=iris_data2(101:135,1:4);
Datos_val_C3=iris_data2(136:150,1:4);

X=[Datos_train_C1;Datos_train_C2;Datos_train_C3]';
  Yd=[ones(1,35)zeros(1,70);zeros(1,35)ones(1,35)
```

```
  zeros(1,35);zeros(1,70) ones(1,35) ];

red=newff(minmax(X),[12 3],{'tansig','logsig'},'trainlm');
red.trainparam.epochs=200;
red=train(red,X,Yd);

Xval=[Datos_val_C1;Datos_val_C2;Datos_val_C3]';
Yred1=sim(red,Xval);

figure
plot3(Datos_train_C1(:,1),Datos_train_C1(:,2),Datos_train_
C1(:,3),'*r'),hold on
plot3(Datos_train_C2(:,1),Datos_train_C2(:,2),Datos_train_
C2(:,3),'*g'),
plot3(Datos_train_C3(:,1),Datos_train_C3(:,2),Datos_train_
C3(:,3),'*b'), hold off
figure
plot3(Datos_val_C1(:,1),Datos_val_C1(:,2),Datos_val_
C1(:,3),'*r'),hold on
plot3(Datos_val_C2(:,1),Datos_val_C2(:,2),Datos_val_
C2(:,3),'*g'),
plot3(Datos_val_C3(:,1),Datos_val_C3(:,2),Datos_val_
C3(:,3),'*b'), hold off

Yred=round(Yred1);
figure; axis([4 8 2 4 1 7])
hold on
for i=1:15
if Yred(1,i)==1
plot3(Datos_val_C1(i,1),Datos_val_C1(i,2),Datos_val_
C1(i,3),'*r')
else
plot3(Datos_val_C1(i,1),Datos_val_C1(i,2),Datos_val_
C1(i,3),'*y')
end
end;

for i=1:15
if Yred(2,i+15)==1
plot3(Datos_val_C2(i,1),Datos_val_C2(i,2),Datos_val_
```

```
C2(i,3),'*g')
else
plot3(Datos_val_C2(i,1),Datos_val_C2(i,2),Datos_val_
C2(i,3),'*y')
end
end;

for i=1:15
if Yred(3,i+30)==1
plot3(Datos_val_C3(i,1),Datos_val_C3(i,2),Datos_val_
C3(i,3),'*b')
else
plot3(Datos_val_C3(i,1),Datos_val_C3(i,2),Datos_val_
C3(i,3),'*y')
end
end;

hold off
```

En la figura 3.32, podemos observar los datos utilizados para el entrenamiento, los asteriscos de diferente color representan los ejemplos de cada una de las clases usados para el proceso de aprendizaje. Para poder visualizar de manera adecuada el problema, seleccionamos las tres primeras características de las cuatro que en total tenemos a disposición.

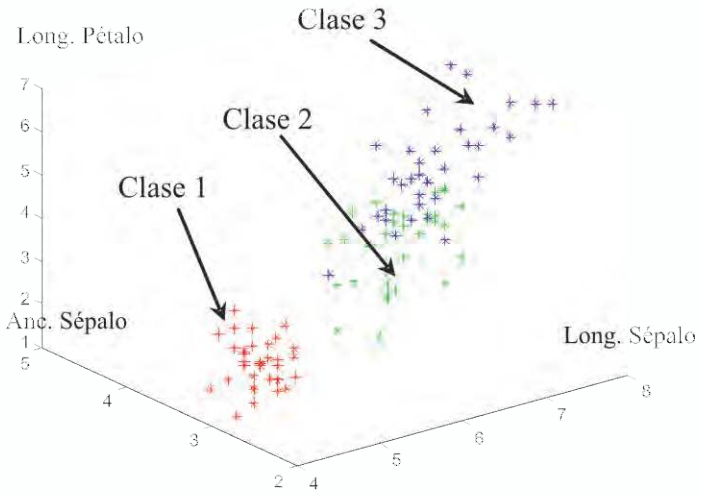


Fig. 3.32 Visualización de los datos de entrenamiento usando las tres primeras características

En la figura 3.33 podemos observar los datos utilizados para la validación.

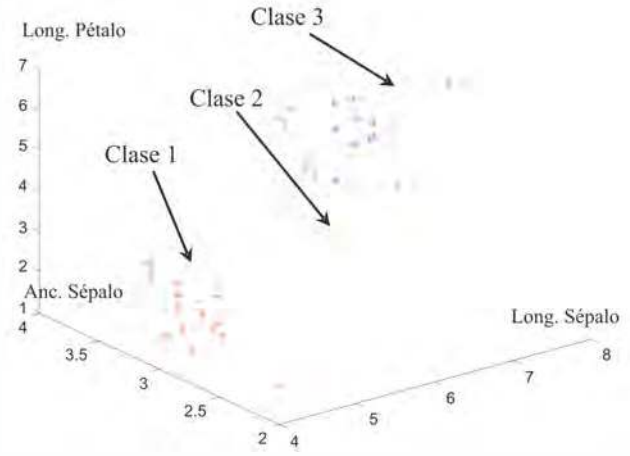


Fig. 3.33 Visualización de los datos de validación usando las tres primeras características

En la figura 3.34, podemos observar la salida de la red con los datos de validación, en donde un gran porcentaje de los ejemplos de validación fueron clasificados adecuadamente, para el ejemplo mostrado en la figura, sólo dos ejemplos fueron mal clasificados. Para distinguir dichos datos, se han representado en un color diferente a los utilizados para la definición de las clases.

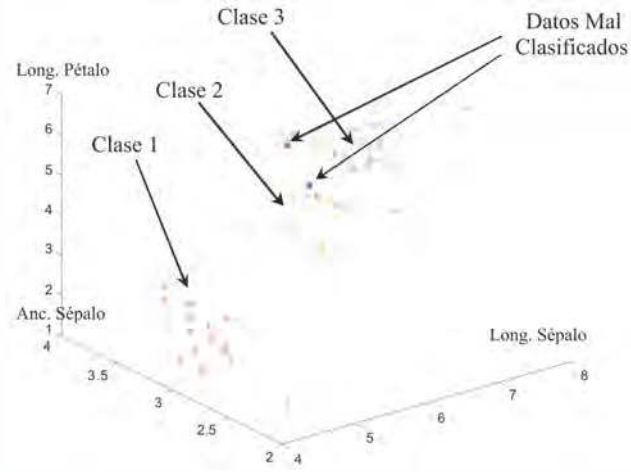


Fig. 3.34 Visualización de la clasificación realizada por la red neuronal las tres primeras características

Proyectos propuestos

1. Solucione el problema de la función XOR realizando el entrenamiento en UVSRNA.
2. Diseñe una aplicación usando UVSRNA, para que aprenda la dinámica de la función seno.
3. En el proyecto de la sección página 110, definir un número de iteraciones fijo, por ejemplo 100, y variar el método de aprendizaje para encontrar como cambia la calidad del aprendizaje de la red visualizando para tal fin la superficie de separación que la misma genera.
4. En el proyecto de la sección paginas 112 - 113, definir un número fijo de iteraciones y verificar el comportamiento de la red cuando se varia el número de neuronas en la capa oculta y el método de aprendizaje.
5. Proponga un método para evaluar la capacidad de generalización que tiene la red MLP al aprender una función, con base en el programa de la sección paginas 112 - 113.
6. Identifique la siguiente planta usando una red neuronal implementada en MATLAB®.

$$G(s) = \frac{5}{(s^2 + 4s + 10)}$$

7. Entrene con la herramienta UV-SRNA una red que sirva para encriptar palabras de 8 bits, para solucionar este problema se sugiere utilizar una red neuronal que en su entrada y salida tenga ocho neuronas y, en la capa oculta tres neuronas. Los datos de entrada y salida para el entrenamiento se sugieren en la siguiente tabla. Una vez entrenada la red la salida de la capa oculta responde al código de encriptación.

Sugerencia: Defina neuronas sigmoidales en la capa oculta y en la capa de salida. Esto hará que los códigos en la capa oculta y de salida se codifiquen con 0 y 1. Trate de lograr un error de entrenamiento de 10^{-5}

Código de Entrada	Código Encriptado (3 bits)	Código de Salida
1 0 0 0 0 0 0 0		1 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0		0 1 0 0 0 0 0 0
0 0 1 0 0 0 0 0		0 0 1 0 0 0 0 0
0 0 0 1 0 0 0 0		0 0 0 1 0 0 0 0
0 0 0 0 1 0 0 0		0 0 0 0 1 0 0 0
0 0 0 0 0 1 0 0		0 0 0 0 0 1 0 0
0 0 0 0 0 0 1 0		0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 1		0 0 0 0 0 0 0 1

8. Entrene una red MLP tanto en MATLAB® como en UV-SRNA que aprenda la función *(0.5*sin(x)+0.5*sin(2*x))* de tal forma que x esté en el rango $-\pi < x < \pi$. Para validar el entrenamiento verifique la generalización de la red entrenada.
9. Entrene una red MLP en MATLAB® que aprenda la función *Seno(x)/x* de tal forma que x esté en el rango $-10 < x < 10$. Para validar el entrenamiento verifique la generalización de la red entrenada.
10. Entrene una red neuronal tipo MLP tanto en MATLAB® como en UV-SRNA que sirva para reconocer las vocales.
11. Entrene una red neuronal tipo MLP tanto en MATLAB® como en UVSRNA que sirva para reconocer los dígitos del 0-9.
12. Entrene una red MLP en MATLAB® que aprenda la siguiente función de dos variables que se muestra en la figura 3.35. Verifique la generalización de la red entrenada.

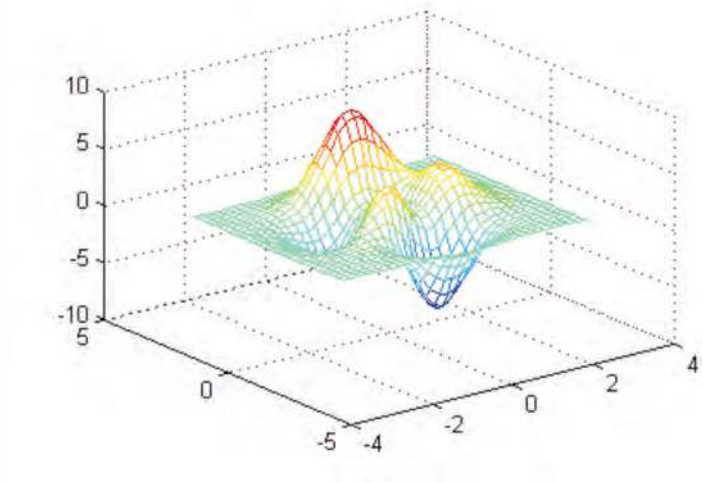


Fig. 3.35 Función de dos variables a identificar

Sugerencia: Utilice el siguiente código para generar la gráfica de la función a aprender.

```
Xini=[-3:0.2:3]; Yini=Xini;
[x,y]=meshgrid(Xini,Yini);
z= 3*(1-x).^2.*exp(-(x.^2) – (y+1).^2) ...
10*(x/5 – x.^3 – y.^5).*exp(-x.^2-y.^2) ...
1/3*exp(-(x+1).^2 – y.^2);
mesh(x,y,z)
```

Capítulo 4

RED NEURONAL DE HOPFIELD

Introducción

En este capítulo veremos un tipo de red neuronal dinámica, considerada así porque en esencia corresponde a un sistema cuyos estados dependen de la variable tiempo. Iniciaremos con una breve revisión a las memorias asociativas bidireccionales con el fin de dejar las bases necesarias para comprender el modelo de red dinámica propuesto, en 1982, por Hopfield. Abordaremos los modelos discretos y continuos propuestos por Hopfield, iniciando con la arquitectura de este tipo de red, continuaremos con el procesamiento de los datos y los respectivos algoritmos de aprendizaje.

Al final, mostraremos algunas aplicaciones en optimización de sistemas y reconocimiento y clasificación de patrones.

John Hopfield
Profesor de la Universidad de Princeton, formuló en 1982 un novedoso modelo de redes neuronales que se denominó Red de Hopfield.
Las redes dinámicas de Hopfield han sido muy usadas en modelado de sistemas no lineales, aunque se pueden aplicar a problemas de diversa índole, desde la clasificación de patrones hasta la optimización de funciones

Memoria autoasociativa bidireccional (BAM)

Una de las características más poderosas que tiene la inteligencia humana es la capacidad para asociar hechos, datos, situaciones, etc. Por ejemplo, si lanzamos una pregunta ¿quién es Rosenblatt?, ¿qué reacciones podríamos esperar?. Como es natural entre la comunidad general, muy probablemente, no se presentará una reacción especial y muy seguramente, lo asociarán al nombre de una persona de origen anglosajón. Pero entre nosotros los lectores de temas relacionados con redes neuronales artificiales, esperamos que sí se haya generado algún tipo de reacción o asociación especial; sin temor a equivocarnos, al escuchar este nombre muy seguramente a nuestras mentes se viene de inmediato el nombre de la red neuronal tipo Perceptron.
Esta maravillosa capacidad de nosotros los seres humanos de realizar asociaciones entre ideas, conceptos, cosas, etc. trataremos de emularla con este nuevo tipo de red neuronal llamada BAM.

Arquitectura de la BAM

La memoria asociativa bidireccional (BAM) presentada en la figura 4.1, está constituida por dos capas de neuronas que son los elementos básicos de procesamiento de la información. Estas capas están completamente interconectadas y el flujo de la información va desde la capa de entrada hacia la capa de salida y desde la capa de salida hacia la de entrada, por lo que los pesos son bidireccionales.

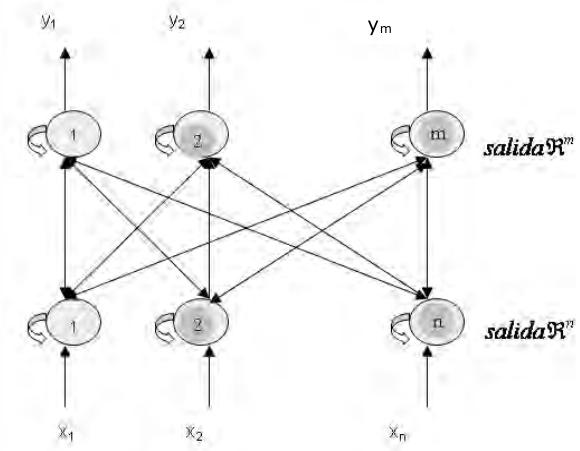


Figura 4.1 Memoria Asociativa Bidireccional

Si el problema está completamente definido, entonces conocemos todos los vectores de entrenamiento por anticipado y podemos determinar la ma-

triz de pesos ***W*** utilizando la ecuación 4.1, donde *P* es el número de patrones de entrenamiento.

$$\mathbf{W} = \mathbf{y}_1\mathbf{x}_1^T + \ldots + \mathbf{y}_i\mathbf{x}_i^T + \ldots + \mathbf{y}_P\mathbf{x}_P^T \tag{4.1}$$

Con esta expresión obtenemos la matriz ***W*** de pesos sinápticos de la red neuronal a partir de los pares ordenados de vectores de entrada y salida del conjunto de entrenamiento.

Memoria autoasociativa

Al igual que la BAM, la memoria autoasociativa está constituida por dos capas de neuronas que son los elementos básicos de procesamiento de la información (figura 4.2.). Estas capas están completamente interconectadas y el flujo de los datos va desde la capa de entrada hacia la de salida y desde la capa de salida hacia la de entrada, por lo que las conexiones y sus respectivos pesos son bidireccionales.

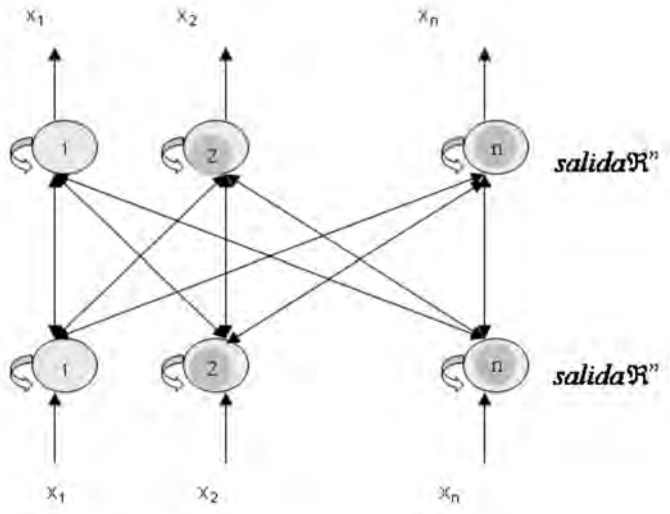


Fig. 4.2 Memoria Autoasociativa

Si conocemos todos los vectores de entrenamiento por anticipado, podemos determinar los pesos utilizando la ecuación 4.2.

$$\mathbf{W} = \mathbf{x}_1\mathbf{x}_1^T + \ldots + \mathbf{x}_i\mathbf{x}_i^T + \ldots + \mathbf{x}_P\mathbf{x}_P^T \tag{4.2}$$

Con esta expresión obtenemos la matriz ***W*** de pesos sinápticos de la red neuronal, con base en los pares ordenados de vectores de entrada y salida del conjunto de entrenamiento. Nótese que por tratarse de un proceso de

autoasociación, los vectores de entrada y salida utilizados para el entrenamiento son iguales. En este caso las dos capas son de igual dimensión por lo que la matriz de pesos $\boldsymbol{W}$ es simétrica y de dimensión *nxn.*

Una vez obtenemos la matriz de pesos, podemos utilizar la red para asociar los vectores de entrada y salida, garantizando que ésta es capaz de responder a datos contaminados con ruido.

Procesamiento de Información en la BAM

Una vez construida la matriz de pesos con base en la ecuación 4.1, la BAM puede usarse para recordar la información que almacenan los vectores del conjunto de entrenamiento. Los pasos que llevaremos a cabo para procesar la información son los siguientes:

1. Aplicamos un par de vectores $(\boldsymbol{x}_i, \boldsymbol{y}_i)$ a las neuronas de la BAM.
2. Propagamos la información de la capa $\boldsymbol{x}$ a la capa $\boldsymbol{y}$ y se actualizan las salidas de las unidades de la capa $\boldsymbol{y}$. En este caso se inicia de $\boldsymbol{x}$ hacia $\boldsymbol{y}$, sin embargo, puede ser de $\boldsymbol{y}$ hacia $\boldsymbol{x}$ y se conoce como contra-propagación.
3. Propagamos la información $\boldsymbol{y}$ actualizada en el paso anterior hacia la capa $\boldsymbol{x}$, y se actualizan estas unidades.
4. Repetimos los pasos 2 y 3 hasta que no haya cambios en las salidas de las dos capas.

Si el entrenamiento es el adecuado, el sistema convergerá a un punto que corresponderá a uno de los vectores utilizados para construir la matriz de pesos en la fase de aprendizaje. La salida para una entrada $\boldsymbol{x}$ cualquiera será el $\hat{\boldsymbol{O}}(\boldsymbol{x}_i) = \boldsymbol{y}_i$ más cercana.

Este algoritmo nos muestra la naturaleza bidireccional del procesamiento de los datos en la BAM. Para ampliar este concepto, consideremos un problema donde es necesario diseñar con este tipo de red un sistema que asocie la huella dactilar y nombre de un usuario, la diferencia con una tabla tradicional, es que si introducimos una huella dactilar contaminada con ruido, el sistema está en capacidad de recuperar el nombre correcto, en cambio una solución basada en una tabla no lo hace.

El caso inverso también el sistema puede resolverlo, por ejemplo, si introducimos el apellido "*Pérec*" a la BAM, ésta a lo largo de varias iteraciones por las conexiones de la misma sería capaz de recuperar el apellido correcto "*Pérez*" y su correspondiente huella dactilar. Una base de datos con un sistema de búsqueda tradicional no sería capaz de recuperarse de este tipo de errores.

Modelo discreto de hopfield

El modelo discreto de la red neuronal de Hopfield se asimila a una memoria autoasociativa, donde la propuesta es reducir las dos capas de neuronas de la BAM a una sola, donde la salida de estas neuronas se lleva a la entrada. En la figura 4.3, mostramos este modelo de red dinámica propuesto por Hopfield y del cual recibió su nombre. A cada una de las neuronas les llega la entrada I_i, que son transformada por la unidad de procesamiento con los pesos sinápticos w_{ij}, para generar la salida y_i.

La arquitectura corresponde a una red de una capa de neuronas o unidades de procesamiento, por lo que esta red pertenece a las denominadas redes neuronales monocapa. Esta capa está conformada por *n* neuronas y la salida de dichas neuronas constituye el estado de la red.

La salida de cada neurona es propagada hacia la entrada de cada una de las otras neuronas, pero no a si misma, por lo que no existe auto-recurrencia; es decir, la salida de una neurona no afecta la entrada de si misma.

Para representar los pesos de la red neuronal de Hopfield usaremos la notación matricial, en la cual profundizaremos más adelante.

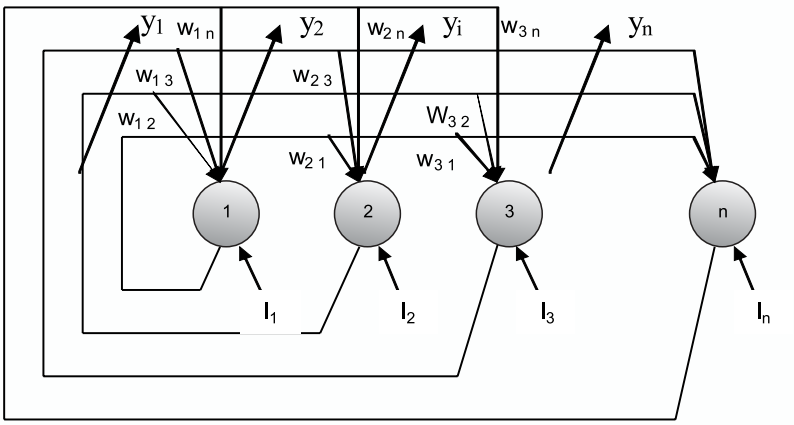


Fig. 4.3 Modelo discreto de Hopfield

Proceso de aprendizaje

En este apartado presentamos el proceso de aprendizaje de una red de Hopfield discreta. La información que esta red va a procesar es del tipo binario, pudiéndose codificar los datos como cero y uno (0,1), o como más uno y menos uno (+1,-1).

En la ecuación 4.3, presentamos la forma de calcular el peso w_{ij} para patrones codificados con valores 0 y 1, y en la ecuación 4.4 la expresión matemática para calcular el peso w_{ij} para patrones codificados con valores

-1 y 1. Donde e_i^k y e_j^k, son las componentes *i-ésima* y *j-ésima* del *k-ésimo* patrón de entrenamiento, *n* es el número de patrones a memorizar en la red neuronal.

$$w_{ij} = \begin{cases} \sum_{k=1}^{n} (2e_i^k - 1)(2e_j^k - 1) \Rightarrow 1 \le i; j \le n; i \ne j \\ 0 \Rightarrow 1 \le i; j \le n; i \ne j \end{cases} \tag{4.3}$$

$$w_{ij} = \begin{cases} \sum_{k=1}^{n} (e_i^k e_j^k) \Rightarrow 1 \le i; j \le n; i \ne j \\ 0 \Rightarrow 1 \le i; j \le n; i \ne j \end{cases} \tag{4.4}$$

Principio de funcionamiento

Como ya estamos en capacidad de calcular la matriz de pesos de una red de Hopfield discreta, ahora vamos a estudiar su funcionamiento, es decir, vamos a ver como actualizamos la salida de la red.

Como lo hemos descrito a lo largo del texto, cada una de las neuronas procesa la información de entrada calculando su entrada total como la sumatoria de las entradas por sus respectivos pesos, pero ahora adicionamos el efecto de la recurrencia que se realiza desde las salidas de las otras neuronas, tal como se muestra en la expresión 4.5 que calcula la entrada neta $Neta_i$.

Una vez calculado el valor total de entrada con la ecuación 4.5, seleccionamos de manera aleatoria una neurona de la red para calcular la salida que la misma va a generar. En el paso siguiente calculamos el valor de la entrada neta de dicha neurona. Teniendo el valor de la entrada neta calculada, actualizamos la salida de la neurona con la ecuación 4.6, para valores codificados como {-1,+1}. En estas expresiones, I_i corresponde a la entrada externa de la *i-ésima* neurona, y_j es la salida de la neurona *j-ésima*. Cuando se plantea que i≠j es porque este tipo de redes no admite la auto-recurrencia. Si las entradas están codificadas como {0,1} la ecuación 4.6 conserva su estructura, considerando que cuando la Neta es negativa, la salida es cero.

$$Neta_i = \sum_{j=1}^{N} y_j w_{ij} + I_i \quad \text{con } i \ne j \tag{4.5}$$

$$x_i(t+1) = \begin{cases} 1; Neta > 0 \\ x_i(t); Neta = 0 \\ -1; Neta < 0 \end{cases} \tag{4.6}$$

Como podemos observar si la entrada neta es igual a cero, el estado o salida de la neurona se hace igual a la salida de la neurona en el instante anterior; de esta manera, podemos visualizar una posible condición para que la salida de la red se estabilice en un valor determinado.

Concepto de energía en el modelo discreto de hopfield

La red de Hopfield es un sistema dinámico porque las salidas de la red pueden asimilarse a un vector de estado, que a medida que se procesa la información, cambia con el tiempo. Para verificar la convergencia de la misma se hace uso del criterio de estabilidad de Lyapunov.

Para efectos de mostrar el cumplimiento del criterio de estabilidad de Lyapunov, definimos en la ecuación 4.7, una función de energía para la red. Podemos verificar que a medida que el estado de una red de Hopfield evoluciona la energía de la red se hace cada vez menor. En otras palabras, la energía de la red tiende a decrecer de manera asintótica.

$$E = -\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} w_{ij}x_i x_j + \sum_{i=1}^{N}\theta_i x_i \tag{4.7}$$

Donde,

w_{ij}	Valor del peso entre la neurona *i-ésima* y la *j-ésima*
x_i	Salida de la neurona *i-ésima*
x_j	Salida de la neurona *j-ésima*
N	Número de neuronas en la capa
θ_i	Umbral de la neurona *i-ésima*

Ejemplo de procesamiento

Con el fin de aclarar el procedimiento para calcular la matriz de pesos, el funcionamiento de la red y el procedimiento para de la energía, revisemos el siguiente ejemplo donde el problema viene descrito por dos vectores de entrenamiento x_1 y x_2.

$$x_1^T = [\ 1\ \ 1\ -1\ -1]$$
$$x_2^T = [-1\ -1\ \ 1\ \ 1]$$

Si aplicamos las expresiones vistas en los apartados anteriores, podemos calcular la matriz de pesos, así:

$$\mathbf{W} = \begin{bmatrix} 1 \\ 1 \\ -1 \\ -1 \end{bmatrix} \cdot [\,1 \;\; 1 \;\text{-}1 \;\text{-}1] + \begin{bmatrix} -1 \\ -1 \\ 1 \\ 1 \end{bmatrix} \cdot [\text{-}1 \;\text{-}1 \;\; 1 \;\; 1]$$

$$\mathbf{W} = \begin{bmatrix} 0 & 2 & -2 & -2 \\ 2 & 0 & -2 & -2 \\ -2 & -2 & 0 & 2 \\ -2 & -2 & 2 & 0 \end{bmatrix}$$

Una vez calculada la matriz de pesos, podemos pasar a la fase de funcionamiento y para ello supongamos que le presentamos a la entrada de la red el patrón .

La red responderá ante el estímulo de este patrón de entrada modificando sus salidas en cada una de sus neuronas:

Para la neurona 1, calculamos la entrada neta:

$$Neta_1 = x_2 w_{12} + x_3 w_{13} + x_4 w_{14} + I_1 = 1*2 + 1*-2 + -1*-2 + 1 = 3$$

Luego, evaluamos esta entrada en la función de activación y obtenemos su respectiva salida:

$$x_1 = 1$$

Para la neurona dos, calculamos la entrada neta:

$$Neta_2 = x_1 w_{21} + x_3 w_{23} + x_4 w_{24} + I_2 = 1*2 + 1*-2 + -1*-2 + 1 = 3$$

Luego, evaluamos esta entrada en la función de activación y obtenemos su respectiva salida:

$$x_2 = 1$$

Para la neurona tres, calculamos la entrada neta:

$$Neta_3 = x_1 w_{31} + x_2 w_{32} + x_4 w_{34} + I_3 = 1*-2 + 1*-2 + -1*2 + 1 = -5$$

Luego, evaluamos esta entrada en la función de activación y obtenemos su respectiva salida:

$$x_3 = -1$$

Observemos que en este caso la salida x_3 ha cambiado su valor, por lo que para calcular la neta de la neurona cuatro se debe tener en cuenta este nuevo valor:

$$Neta_4 = x_1 w_{41} + x_2 w_{42} + x_3 w_{43} + I_4 = 1 * -2 + 1 * -2 + -1 * 2 - 1 = -7$$

Luego, evaluamos esta entrada en la función de activación y obtenemos su respectiva salida:

$$x_4 = -1$$

Finalmente, la salida de la red viene determinada por el vector $\boldsymbol{x_f}$:

$$x_f = [1 \quad 1 \quad -1 \quad -1]$$

Como podemos observar, la salida de la red converge al primer patrón que fue almacenado en la red. Ahora verifiquemos como es el comportamiento de la energía de la red, para ello retomemos el vector de entrada $\boldsymbol{x_0}$.

$$x_0 = [1 \quad 1 \quad 1 \quad -1]$$

Al iniciar la salida de la red es el mismo patrón de entrada y, por lo tanto, la energía de la red en este caso la calculamos así:

$$E = -\frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} w_{ji} x_i x_j \ , \quad i \neq j$$

Para *i=1*

$$E_1 = w_{21} x_1 x_2 + w_{31} x_1 x_3 + w_{41} x_1 x_4 = 2$$

Para *i=2*

$$E_2 = w_{12} x_2 x_1 + w_{32} x_2 x_3 + w_{42} x_2 x_4 = 2$$

Para *i=3*

$$E_3 = w_{13} x_3 x_1 + w_{23} x_3 x_2 + w_{43} x_3 x_4 = -6$$

Para *i=4*

$$E_4 = w_{14}x_4x_1 + w_{24}x_4x_2 + w_{34}x_4x_3 = 2$$

La energía total de la red la calculamos con la sumatoria de los elementos parciales anteriores:

$$E = -\frac{1}{2}\sum_{i=1}^{4} E_i = 2 + 2 - 6 + 2 = 0$$

Ahora calculemos la energía de la red para la salida final x_f, hacia donde converge la red.

$$x_f = [1 \quad 1 \quad -1 \quad -1]$$

Para i=1

$$E_1 = w_{21}x_1x_2 + w_{31}x_1x_3 + w_{41}x_1x_4 = 6$$

Para i=2

$$E_2 = w_{12}x_2x_1 + w_{32}x_2x_3 + w_{42}x_2x_4 = 6$$

Para i=3

$$E_3 = w_{13}x_3x_1 + w_{23}x_3x_2 + w_{43}x_3x_4 = 6$$

Para i=4

$$E_4 = w_{14}x_4x_1 + w_{24}x_4x_2 + w_{34}x_4x_3 = 6$$

La energía total de la red, para la salida final que implicó la convergencia, la calculamos con la sumatoria de los elementos parciales anteriores:

$$E = -\frac{1}{2}\sum_{i=1}^{4} E_i = -\frac{1}{2}(6 + 6 + 6 + 6) = -12$$

Observemos que la salida a la cual converge la red tiene una energía menor que la salida inicial. Esto es lógico pues cuando la red de Hopfield cambia su salida es por que está convergiendo a un mínimo de la función de energía.

La explicación de este hecho radica en que la red de Hopfield fue diseñada de tal forma que, por si misma, constituye un sistema estable. Lo

anterior significa que la salida de la red converge a un valor en el cual la red se estabiliza. Lo que esperamos es que esta salida a la cual converge la red, sea uno de los patrones almacenados en ella. Sin embargo, debe tenerse en cuenta que este comportamiento se garantiza si los patrones a memorizar son ortogonales. Si no se cumple esta condición la red se puede estabilizar en una salida que no fue la almacenada, pero que constituye un mínimo de la función de energía.

Modelo continuo de hopfield

Hemos estudiado con detenimiento el modelo discreto de red neuronal propuesto por Hopfield, ahora nos detendremos en el análisis de la arquitectura, la convergencia y las aplicaciones del modelo continuo de la red neuronal de Hopfield. En la figura 4.4, observamos el modelo para esta red propuesto por Hopfield, que a partir de dispositivos eléctricos y electrónicos.

En el modelo continuo de Hopfield representaremos a cada neurona con un circuito constituido por un condensador, una conductancia y un amplificador no lineal con función de transferencia sigmoidal, que genera la salida de la misma.

La salida de una neurona se lleva hacia las demás como señal de excitación o inhibición, con magnitud positiva o negativa respectivamente; por esta razón, adicionaremos al modelo un inversor a la salida de cada una de las neuronas.

La recurrencia propia de este tipo de redes, la representamos llevando la salida de cada neurona hacia la entrada de las demás. Con pequeños círculos hemos marcado las conexiones que se establecen, para el caso de una neurona que es excitada por la salida de otra, la conexión la marcamos sobre la salida positiva, en caso de existir una inhibición, la conexión la marcamos sobre la salida negativa. Esta situación en el modelo matemático lo reflejamos con la transconductancia T_{In} que representa el valor de la conexión.

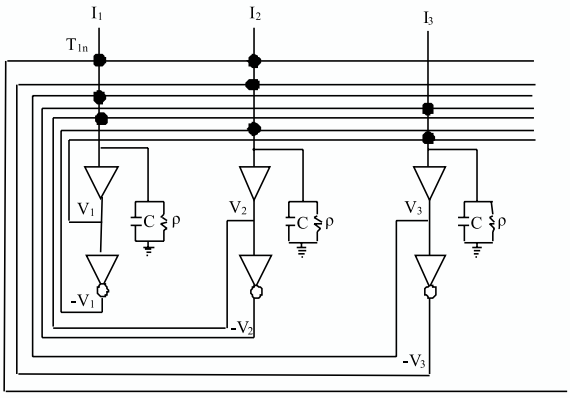


Fig. 4.4 Modelo Continuo de Hopfield

Modelo continuo de hopfield de una neurona

Vamos a detenernos en el modelo continuo que Hopfield propone para una neurona en el marco del modelo global de red neuronal. Como podemos observar de la figura 4.5, asimilamos la neurona a un nodo que recibe el aporte en corriente de las salidas de las otras neuronas a través de las transconductancias T_{ij}. Por otro lado, el nodo recibe un aporte de corriente de una fuente externa, para representar la entrada a la neurona.

El valor total de corriente se lleva al circuito paralelo conformado por el condensador y la conductancia, que representa la resistencia y la capacitancia de la membrana celular. El potencial u_i que se genera en este circuito se lleva a la entrada del amplificador no lineal, con función de transferencia sigmoidal, como una aproximación de la salida de la neurona en respuesta al potencial de activación total.

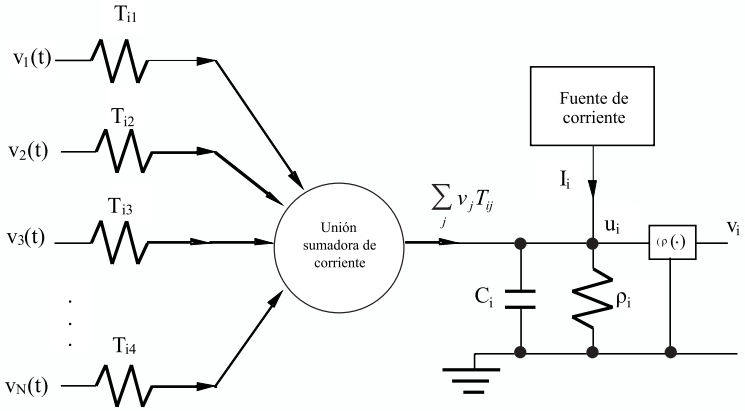


Fig. 4.5 Modelo Continuo de Hopfield de una Neurona

Si aplicamos el análisis de circuitos a este modelo continuo de Hopfield de una neurona y usando las leyes de Kirchoff, para el nodo u_i, la ecuación equivalente es la 4.8.

$$I_i + \sum_j (v_j - u_i) T_{ij} = C_i \frac{du_i}{dt} + \frac{u_i}{\rho_i} \tag{4.8}$$

Si reorganizamos la ecuación 4.8, esta expresión la podemos transformar así:

$$C_i \frac{du_i}{dt} = I_i + \sum_j v_j T_{ij} - \frac{u_i}{\rho_i} - \sum_j u_i T_{ij} \tag{4.9}$$

$$C_i \frac{du_i}{dt} = I_i + \sum_j v_j T_{ij} - u_i \left(\frac{1}{\rho_i} + \sum_j T_{ij}\right)$$

Si definimos el segundo término de la sumatoria como,

$$\frac{1}{R_i} = (\frac{1}{\rho_i} + \sum_j T_{ij}) \tag{4.10}$$

entonces podemos simplificar la expresión de la corriente total que llega a la capacitancia, así:

$$C_i \frac{du_i}{dt} = I_i + \sum_j v_j T_{ij} - \frac{u_i}{R_i} \tag{4.11}$$

Si dividimos los dos lados de la ecuación entre C_i, obtenemos la siguiente ecuación diferencial 4.12 que modela el comportamiento del potencial u_i.

$$\frac{du_i}{dt} = \frac{I_i}{C_i} + \sum_j \frac{v_j T_{ij}}{C_i} - \frac{u_i}{R_i C_i} \tag{4.12}$$

Con el fin de simplificar la anterior ecuación, vamos a definir los siguientes términos:

$$\omega_{ij} = \frac{T_{ij}}{C_i} \qquad a_i = \frac{1}{R_i C_i} \qquad b_i = \frac{1}{C_i}$$

De esta manera, vemos en la ecuación 4.13 como el cambio en el voltaje u_i queda representado en función de la entrada externa Ii, la sumatoria de las salidas de las demás neuronas y el valor del potencial u_i.

$$\frac{du_i}{dt} = b_i I_i + \sum_j \omega_{ij} v_j - a_i u_i \tag{4.13}$$

Si observamos detenidamente esta expresión, encontraremos su similitud al modelo matemático que hemos venido utilizando para representar a una neurona artificial.

La ecuación 4.13 que representa el modelo continuo de Hopfield para una neurona la podemos representar en formato matricial usando la ecuación 4.14.

$$\mathbf{\frac{dU}{dt} = WV - AU + BI} \tag{4.14}$$

Si definimos al vector de estado como vector de potenciales de activación total de las diferentes neuronas, obtenemos las siguientes expresiones:

$$\mathbf{X= U} \tag{4.15}$$
$$\mathbf{V= \Phi\ (U)=\Phi(X)} \tag{4.16}$$

Si reemplazamos los vectores de estado definidos en el modelo continuo para la neurona, obtenemos la ecuación 4.17 que representa en el espacio de estado el mode-lo continuo de Hopfield .

$$\frac{\mathbf{dX}}{\mathbf{dt}} = \mathbf{W\Phi(X) - AX + BI} \tag{4.17}$$

Para aclarar estos conceptos apliquemos la ecuación 4.17, en el caso de una red neuronal de dos estados x_1 y x_2, cuya representación matricial quedará de la siguiente manera:

$$\begin{bmatrix} \frac{dx_1}{dt} \\ \frac{dx_2}{dt} \end{bmatrix} = \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix} \begin{bmatrix} \phi(x_1) \\ \phi(x_2) \end{bmatrix} - \begin{bmatrix} a_{11} & 0 \\ 0 & a_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} b_{11} & 0 \\ 0 & b_{22} \end{bmatrix} \begin{bmatrix} I_1 \\ I_2 \end{bmatrix}$$

La figura 4.6, representa el diagrama de bloques nos muestra el modelo continuo de Hopfield de la red neuronal.

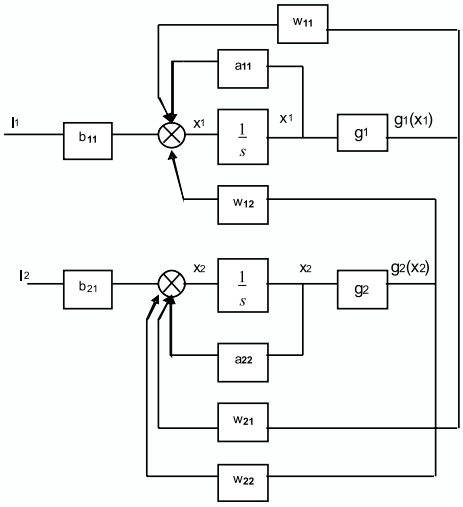


Fig. 4.6 Representación en Espacio de Estado del Modelo Continuo de Hopfield para una Red de dos Neuronas

Función de energía para el modelo continuo de hopfield

De manera similar al caso de la red discreta de Hopfield, la convergencia de la red continua de Hopfield, podemos estudiarla con base en el criterio de estabilidad de Lyapunov.

Para el modelo continuo Hopfield propuso la función de energía, ecuación 4.18, que depende de los potenciales de salida de las neuronas, la transconductancia de conexión, la entrada externa y la función de transferencia del amplificador inversor.

$$E = -\frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} T_{ij} v_i v_j + \frac{1}{\lambda}\sum_{i=1}^{n}\frac{1}{R_i}\int_0^{vi} \phi_i^{-1}(v)dv - \sum_{i=1}^{n} I_i v_i \tag{4.18}$$

Para realizar el análisis de estabilidad de la red con base en el criterio de estabilidad de Lyapunov, es necesario utilizar la derivada de la función de energía calculada con la ecuación 4.19.

$$\frac{dE}{dt} = -\sum_i C_i \left(\frac{d(\phi_i^{-1}(v_i))}{dt}\right)\left(\frac{dv_i}{dt}\right)^2 \tag{4.19}$$

Si analizamos la expresión de la derivada, vemos que ésta depende de tres términos:

- La capacitancia C_i que siempre será positiva.
- La derivada de la salida de la *i-ésima* neurona, que al estar elevada al cuadrado, igualmente siempre será positiva.
- La derivada de la inversa de la función de activación de la neurona *i-ésima*, que de igual manera, siempre será positiva.

En la figura 4.7 observamos la función de activación sigmoidal y su inversa. En la gráfica de la función inversa tracemos una recta tangente imaginaria que va recorriendo todos sus puntos, entonces podremos observar que la pendiente de esta recta siempre será positiva y por esta razón la derivada será positiva.

Como todos los términos de la sumatoria de la derivada de la función de energía son positivos, podemos concluir que esta derivada siempre será positiva, y si nos acogemos al criterio de Lyapunov, podemos estar seguros que cuando la red neuronal está en funcionamiento va a converger a un valor estable.

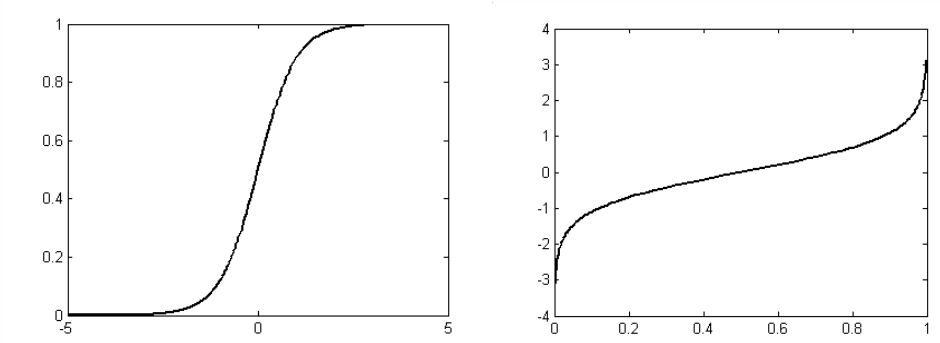


Fig. 4.7 Función de Activación Sigmoidal y su Inversa

Aproximación práctica

Red tipo hopfield con MATLAB®

En este primer proyecto quedaremos en capacidad de construir una Red de Hopfield usando MATLAB®, con base en el siguiente programa. Si aprovechamos la capacidad que tienen estas redes para memorizar datos, la utilizaremos para almacenar dos puntos en el plano.

```
% Programa que construye una red neuronal tipo HOPFIELD y
almacena dos puntos en el plano

close all;
T=[1 1;-1 -1]';

% Se grafican los valores a almacenar
figure;
plot(T(1,:),T(2,:),'*r');
hold on;
axis([-1.1 1.1 -1.1 1.1]);

% Se crea la red tipo Hopfield
red=newhop(T);

% Se verifica si los estados fueron almacenados
Ai = T;
[Y,Pf,Af] = sim(red,2,[],Ai);
Y
```

```
% Se prueba con un estado diferente
Ai = {[0.8; 0.8]};

% Se simula la red para N pasos
N=10;
[Y,Pf,Af] = sim(red,{1 N},{},Ai);
Y{1};
Evolucion=[cell2mat(Ai) cell2mat(Y)];
Inicio=cell2mat(Ai);
plot(Inicio(1,1),Inicio(2,1),'b+');
plot(Evolucion(1,:),Evolucion(2,:));

disp('Oprima una tecla para continuar');
pause;
for i=1:10
Ai = {2*rand(2,1)-1};
% Se simula la red por N pasos
N=25;
[Y,Pf,Af] = sim(red,{1 N},{},Ai);
Y{1};
EvolucionAux(:,:,i)=[cell2mat(Ai) cell2mat(Y)];
InicioAux(:,:,i)=cell2mat(Ai);
plot(InicioAux(1,1,i),InicioAux(2,1,i),'b+');
plot(EvolucionAux(1,:,i),EvolucionAux(2,:,i));
pause(1)
end;
title('Rojo = Patrones Memorizados Azul =Patrones de Prueba')

hold off;
```

En la ejecución de este proyecto propuesto es importante destacar que la red me-moriza los puntos suministrados, pero si le presentamos un patrón diferente a dichos puntos, obtenemos como resultado que la salida de la red evoluciona desde el patrón presentado hacia uno de los puntos utilizados en el entrenamiento, efecto que visualizamos en la figura 4.8.

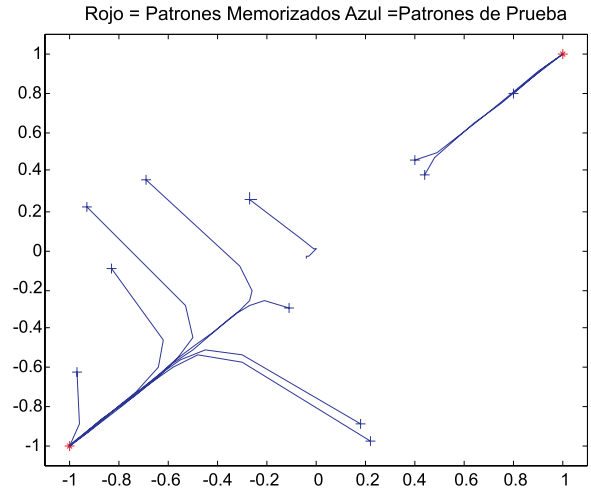


Fig. 4.8 Visualización de la evolución de la salida de la red ante diferentes patrones de entrada

Proyectos propuestos

1. Proponga y simule una red tipo Hopfield para que aprenda los siguiente puntos definidos en un espacio de cuatro dimensiones. Defina patrones con los siguientes vectores:

   P_1 =[1 1 0 0] y P_2 =[0 0 1 1].
   Verifique el comportamiento de la red ante diferentes entradas en MATLAB®

2. El objetivo ahora, es verificar como la red va minimizando su energía a medida que converge al patrón almacenado, para verificar este comportamiento, valide la red con los siguientes patrones.
   Patrón de entrada: [1 1 0 1 1]

Tabla 1 Comportamiento de la red ante el patrón de entrada [1 1 0 11]

Patrón de salida	Energía	Iteración

Patrón de entrada: [0 1 0 1 1]

Tabla 2 Comportamiento de la red ante el patrón de entrada [0 1 0 1 1]

Patrón de salida	Energía	Iteración

Patrón de entrada: [1 0 0 1 1]

Tabla 3 Comportamiento de la red ante el patrón de entrada [1 0 0 1 1]

Patrón de salida	Energía	Iteración

Patrón de entrada: [0 1 0 1 0]

Tabla 4 Comportamiento de la red ante el patrón de entrada [0 1 0 1 0]

Patrón de salida	Energía	Iteración

Patrón de entrada: [1 0 0 0 1]

Tabla 5 Comportamiento de la red ante el patrón de entrada [1 0 0 0 1]

Patrón de salida	Energía	Iteración

Patrón de entrada: [0 0 0 0 1]

Tabla 6 Comportamiento de la red ante el patrón de entrada [0 0 0 0 1]

Patrón de salida	Energía	Iteración

Patrón de entrada: [1 0 0 0 0]

Tabla 7 Comportamiento de la red ante el patrón de entrada [1 0 0 0 0]

Patrón de salida	Energía	Iteración

Patrón de entrada: [1 1 1 1 1]

Tabla 8 Comportamiento de la red ante el patrón de entrada [1 0 0 0 0]

Patrón de salida	Energía	Iteración

Realice un programa en MATLAB® que permita simular la red de Hopfield discreta. Dicho programa debe permitir visualizar la manera como cambia la energía de la red a medida que va iterando.

Capítulo 5

MAPAS AUTO-ORGANIZADOS DE KOHONEN

Introducción

Los Mapas Auto-organizados (SOM por su nombre en inglés Self-Organizing Maps) fueron presentados por Teuvo Kohonen en 1982, por lo que también reciben el nombre de Mapas Auto-organizados de Kohonen o Redes Neuronales de Kohonen, estos mapas están inspirados en la capacidad del cerebro humano de reconocer y extraer rasgos y características relevantes del mundo que los rodea. En la década de los ochenta, el profesor Kohonen propuso una red neuronal artificial capaz de aprender la estructura topológica de un conjunto de datos a través de un proceso de autoorganización de las neuronas de la red.

Desde el punto de vista bio-inspirado, esta red posee un nivel de aprendizaje de mayor evolución que el supervisado de los capítulos previos, puesto que la red, de alguna manera, es capaz de encontrar de manera autónoma la estructura de los datos, lo que nos facilita descubrir la información relevante de estos datos sin necesidad de entregarle pares ordenados donde se define una salida correspondiente a su entrada, como en el caso del aprendizaje supervisado de las redes MLP. Por primera vez en este libro nos encontramos con una red neuronal cuyo aprendizaje es No–Supervisado.

Teuvo Kohonen
Profesor Emérito de la Academia de Finlandia, en 1982 propuso los denominados Mapas Auto-organizados (SOM).
Los mapas auto-organizados se basan en la propiedad que tiene el cerebro para formar grupos de neuronas que procesan información del mismo tipo.
Los mapas auto-organizados se han aplicado a problemas diversos desde la minería de datos, hasta el procesamiento de imágenes.

La idea básica del SOM es crear una imagen de un espacio multidimensional de entrada en un espacio de salida de menor dimensión. Se trata de un modelo con dos capas de neuronas, como observamos en la figura 5.1, la primera capa es de entrada y la segunda de procesamiento. Las neuronas de la capa de entrada se limitan a recoger y canalizar la información. La capa de salida o procesamiento, está ligada a la capa de entrada a través de los pesos sinápticos de las conexiones; su principal tarea es la de realizar una proyección del espacio *n-dimensional* de entrada en una espacio *m-dimensional* de salida, conservando las características esenciales de los datos gracias a la relación de vecindad que se establece en las neuronas de la capa de salida. Al final, obtenemos un mapa autoorganizado que nos muestra un conjunto de sectores que agrupa los datos con características comunes.

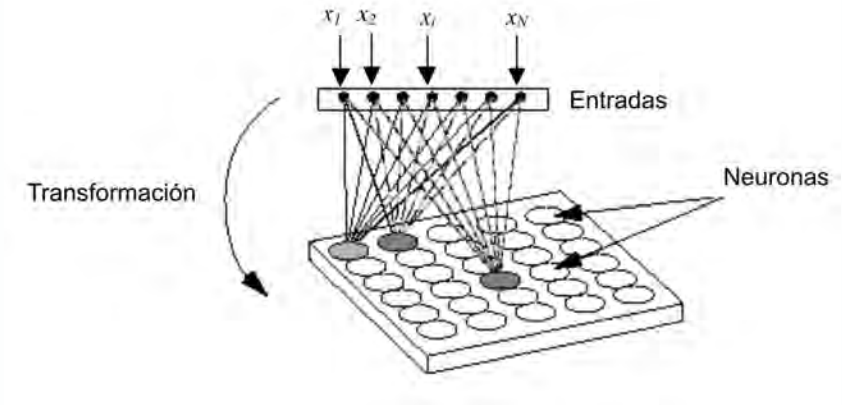


Figura 5.1 Mapa Auto Organizado

En otros tipos de redes neuronales no se tienen en cuenta las relaciones topológicas o geométricas que hay entre los nodos, y no es relevante establecer algún nivel de vecindad entre neuronas cercanas de una misma capa para establecer una relación bilateral entre ellas. En el SOM este aspecto de cercanía si lo vamos a considerar, es importante en la estructura de la red, y será fundamental en la manera como van a interactuar las neuronas.

Veremos que no va a existir la misma relación entre dos nodos cercanos y dos que están muy alejados el uno del otro, así que el objetivo será entrenar la red de manera que las salidas cercanas correspondan a entradas cercanas. En la mayor parte de los casos utilizaremos la distancia euclideana para tener una medida del nivel de relación que hay entre estos nodos.

Se puede decir que el SOM es una clase especial y única de red neuronal, ya que construye un mapa que auto-organiza la topología de las neuronas de un espacio multidimensional, de manera que las distancias entre los datos se conservan.

Estos mapas clasifican automáticamente la información de entrada permitiendo organizarla, de esta manera se pueden observar relaciones importantes entre los datos y descubrir otras relaciones subyacentes entre ellos. El SOM sirve entonces como una herramienta de agrupación de datos de dimensiones altas y es fácil de visualizar debido a su forma (usualmente) de dos dimensiones.

El modelo bioinspirado de Kohonen

El cerebro está organizado en zonas especializadas en procesar la diversa información que le llega. Si aceptamos una simplificación en los conceptos biológicos, podemos decir que en el cerebro hay una distribución predefinida de sus neuronas, orientadas a procesar información específica. De todas maneras, esto no significa que dichas neuronas no puedan procesar información de otro tipo, por ejemplo, cuando una persona pierde uno de sus sentidos. Coloquialmente se dice que un invidente desarrolla otros sentidos, pero en realidad lo que sucede es que en el cerebro ocurre una reorganización de la manera como se procesa la información sensorial proveniente de los sentidos que funcionan para suplir los datos faltantes por la visión, en este caso.

El cerebro se especializa por regiones para procesar la información proveniente de las diferentes partes del cuerpo humano. A estas regiones se les conoce como mapas sensoriales y están conformados por un conjunto de neuronas asociadas entre sí; como ejemplo podemos mencionar los siguientes:

- Mapa Somatosensorial que genera una imagen en el cerebro de la piel del cuerpo humano.
- Mapa Retinotópico que procesa la información proveniente de la visión.
- Mapa Tonotópico que procesa la información proveniente del oído.

Aunque genéticamente venimos predispuestos para que estos mapas realicen tareas como la de diferenciar sonidos, para el caso del mapa tonotópico, el aprendizaje tiene un papel fundamental en la especialización de estas zonas. Por ejemplo, cuando escuchamos una orquesta sinfónica, en general somos capaces de diferenciar los ins-trumentos de cuerda de los de viento, ninguno de nosotros confundiría el sonido de una trompeta con el de un violín; pero, ¿qué ocurre cuando se interpretan instrumentos similares, por ejemplo, un saxo tenor y un saxo barítono? Tal vez, ya no seamos capaces de diferenciarlos. Sin embargo, un estudioso de la música, a través de muchos años de entrenamiento logra no sólo diferenciarlos, sino separar las notas musicales que interpretan e identificarlas como un DO, un RE o un MI, para citar algunas.

Con esto lo que queremos mostrar es como un adecuado proceso de aprendizaje ayuda perfeccionar el trabajo realizado por estas zonas del cerebro que hemos denominado Mapas Sensoriales

En síntesis, podemos afirmar que el cerebro humano posee la capacidad inherente de formar mapas topológicos a partir de la información del mundo que lo rodea, por lo que Kohonen presentó un modelo con capacidad para formar mapas de características, cuyo objetivo es mostrar que un estímulo exterior (información de entrada), por sí sólo, suponiendo una estructura propia y una descripción funcional del comportamiento de la red, es suficiente para forzar la formación de mapas.

Arquitectura de la red

El mapa auto-organizado de Kohonen está constituido por dos niveles de neuronas, el de entrada y el de salida. Pero sólo en el nivel de salida se genera un procesamiento de información, por lo que recibe el nombre de Capa de Salida y la red pertenece al tipo Monocapa. La conectividad es total, es decir, todas las neuronas de la capa de salida reciben los estímulos de las neuronas de entrada.

Las neuronas de entrada reciben la información de los datos provenientes del exterior de la red neuronal y a través de las conexiones sinápticas envían esta información a la capa de procesamiento de la red.

Kohonen propuso inicialmente una red con una capa de salida, figura 5.2, donde las neuronas están organizadas en una fila similar a un vector, a esta estructura la denominó LVQ (Linear Vector Quantization). A través de un vector de entrada, presentamos a la red un estímulo, que es modificado por los pesos sinápticos de las conexiones y se le presentan a la capa de neuronas de procesamiento, resultando un vector de salida $\boldsymbol{y}$.

Como veremos más adelante, las neuronas compiten entre sí para activarse, ganando aquella cuyo vector de pesos sinápticos asociado, esté más

cercano al vector de entrada, es decir, se mide la distancia entre el vector de entrada y los vectores de pesos sinápticos asociados a las ***m*** neuronas de salida. Aquella neurona cuyo vector de pesos diste menos del vector de entrada, será la ganadora y modificará su vector de pesos con el fin de representar ese dato de entrada.

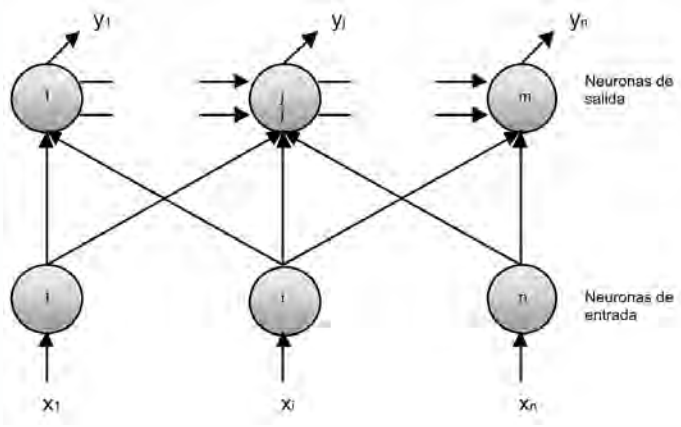


Fig. 5.2 Arquitectura de una red simple de Kohonen

La arquitectura del mapa bidimensional SOM de la figura 5.3 es similar a la del LVQ, pero las neuronas de salida se organizan en forma de matriz. A las neuronas organizadas de esta manera es lo que conocemos como el mapa de la red. En general, este tipo de red permite tomar patrones de entrada de dimensión ***n*** y visualizar los grupos que hay en los mismos usando una estructura bidimensional o de mapa.

La red neuronal tipo SOM tiene *N* neuronas en el nivel de entrada, *M* neuronas en la capa de salida, se establece una conectividad desde la capa de entrada hacia la capa de salida (*Feedforward*) y en la fase de entrenamiento, se establece en cada neurona una influencia de las neuronas vecinas, lo que se conoce como vecindad.

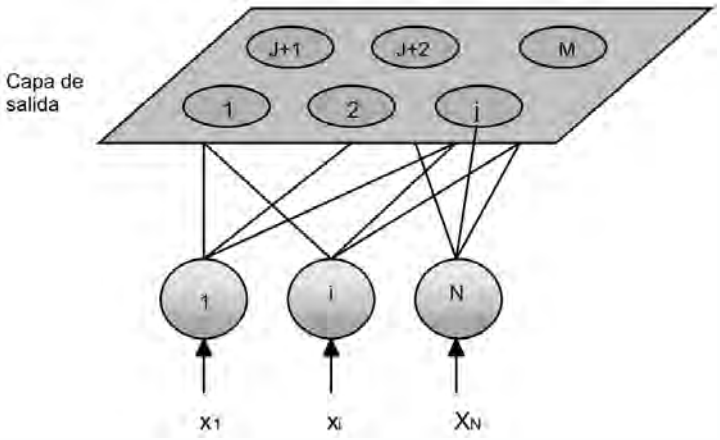


Fig. 5.3 Arquitectura de una red en dos dimensiones

Cuando Kohonen planteó su modelo de mapas auto-organizados, también tuvo en cuenta que en el cerebro, no sólo se activa ante un estímulo, una única neurona, sino que lo hace toda una región; propiedad que garantiza la robustez de nuestro cerebro y la respuesta adecuada frente a un estímulo aunque éste posea niveles de incertidumbre. Pensemos, en cómo es nuestra voz al iniciar la mañana, a media mañana o en la noche… muy seguramente es diferente, ¿coincidimos en esta apreciación? Pero de todas maneras las personas que nos conocen bien, nos reconocen, pese a que nuestra voz tiene diversos matices. Pero existe un cambio mucho más pronunciado, y es cuando hablamos por teléfono: Muchas veces nos ha pasado que a personas que conocemos poco no las reconocemos cuando nos hablan por teléfono, pero a nuestros seres queridos, los identificamos sin ningún tipo de duda. Justamente ésta es una capacidad de nuestro cerebro de responder adecuadamente a estímulos con niveles de incertidumbre y se debe, en parte, a que no hay una única neurona que responde al estímulo o que haya aprendido a reconocer un único matiz de voz, sino que hay toda una región que lo hace y a esto es lo que intentaremos emular con una Función de Vecindad en el proceso de aprendizaje.

En la figura 5.4, observamos la función de vecindad, en los espacios bidimensional y tridimensional. Se puede observar que el efecto de la vecindad es mayor mientras más cerca esté de la neurona ganadora, que está ubicada en el centro de la función de vecindad.

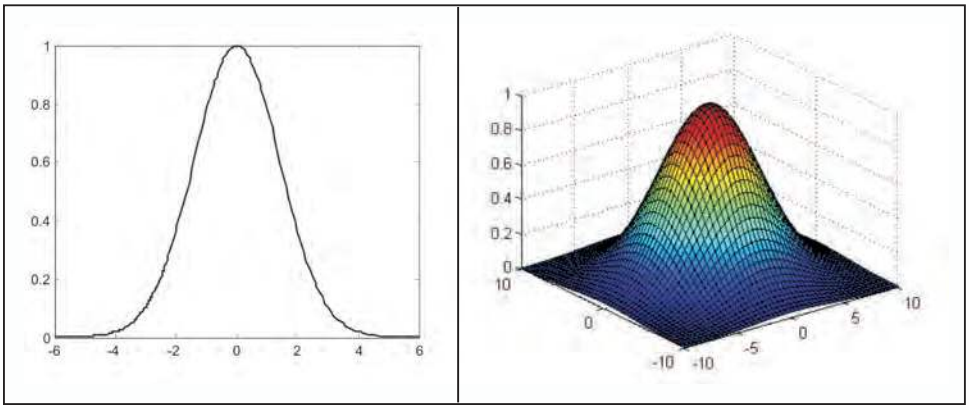


Fig. 5.4 Ejemplos de Funciones de Vecindad para Mapas Auto-Organizados

Algoritmo de aprendizaje

Consideraciones iniciales

Antes de llevar a cabo el proceso de aprendizaje de la red, debemos definir los parámetros que caracterizan la arquitectura de la red, es decir, asignar el número de neuronas (*N*) del nivel de entrada y la dimensión la capa de salida de la red (*M*). El número de neuronas de entrada es igual a la dimensión del vector de características que define el problema que se va a solucionar con la red. La capa de salida conforma el Mapa Auto-Organizado de Kohonen y, en general, las *M* neuronas del mapa se distribuyen en una matriz de dimensión M_f x M_c, donde M_f es el número de filas del mapa y M_c el número de columnas.

El siguiente paso en el proceso de entrenamiento es inicializar los pesos de la red. Estos pesos generalmente los inicializamos de manera aleatoria en un rango comprendido entre cero y uno si los datos de entrenamiento han sido normalizados. Si los datos de entrenamiento no han sido normalizados, los pesos de la red se pueden inicializar aleatoriamente alrededor de la media de dicho conjunto de datos, aunque nuestra recomendación es que los datos estén normalizados.

Modelo matemático

El aprendizaje en el modelo auto-organizado de Kohonen está regido por la ecuación 5.1 que define la variación de los pesos, $\Delta \boldsymbol{w}_r$, en este algoritmo. En donde la neurona ganadora y sus vecinas, modifican su vector de pesos sumándole una fracción de la distancia existente entre el vector de entrada y el vector de pesos en el instante *t* del algoritmo.

$$\Delta \mathbf{w}_r = \alpha(t) h_{rs}(t)(\mathbf{x} - \mathbf{w}_r) \tag{5.1}$$

donde,

$\boldsymbol{x}$	: Vector de Entrada
$\Delta \boldsymbol{w}_r$	: Variación del vector de pesos para la neurona *r-ésima*
α	: Rata de aprendizaje
$\boldsymbol{h}_{rs}$	: Función de Vecindad
$\boldsymbol{w}_r$	: Vector de pesos de la neurona *r-ésima*
$\boldsymbol{t}$	: Índice de iteración

La rata de aprendizaje $\alpha(t)$ se calcula con base en la ecuación 5.2, donde α_i y α_f las corresponden a las ratas de aprendizaje inicial y final, respectivamente; $\boldsymbol{t}_{max}$, es el número máximo de iteraciones. Con esta expresión lo que buscamos es que la rata de aprendizaje siga una forma exponencial, figura 5.5., con el fin de obtener al inicio del proceso fuertes variaciones en los pesos y, a medida que evolucione el algoritmo, el tamaño de los cambios de los pesos se atenúe para garantizar que al iniciar el algoritmo las neuronas se distribuyan rápidamente entre los datos más representativos de la base de entrenamiento y al finalizar, cuando las neuronas ya hayan aprendido la distribución de los datos, las modificaciones de los pesos sean más pequeñas con el fin de solo llevar a cabo un ajuste fino de los pesos.

$$\alpha(t) = \alpha_i \left(\frac{\alpha_f}{\alpha_i} \right)^{\frac{t}{t_{max}}} \tag{5.2}$$

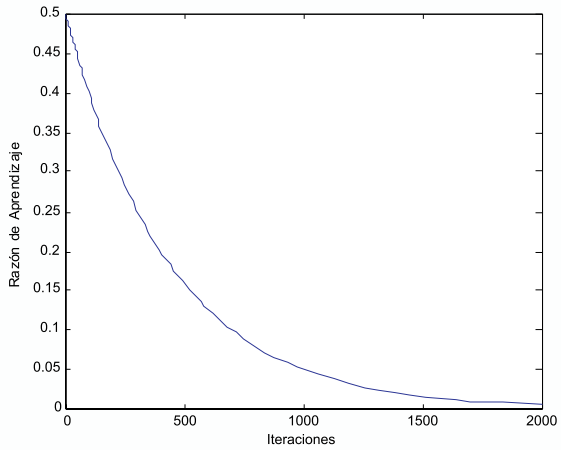


Fig. 5.5 Variación de la Razón de Aprendizaje α(t)

La función de vecindad se define con la ecuación 5.3, donde *d* es la distancia euclideana entre la neurona ganadora *s* y la neurona *r* a la cual se le modifican los pesos. El rango de vecindad $\sigma(t)$ es variable y se define con la ecuación 5.4, donde σ_i y σ_f corresponden a los rangos de vecindad inicial y final, respectivamente.

$$h_{rs}(t) = e^{\left(\frac{-d\ (r,s)^2}{2\sigma(t)^2}\right)} \tag{5.3}$$

$$\sigma(t) = \sigma_i \left(\frac{\sigma_f}{\sigma_i}\right)^{\frac{t}{t_{max}}} \tag{5.4}$$

La vecindad es una función exponencial cuya característica hace que las neuronas más alejadas de la unidad ganadora, se vean afectadas en sus pesos sinápticos en una menor proporción que las más cercanas; es importante resaltar como podemos modificar en el proceso de entrenamiento la magnitud de la región de vecindad a través del parámetro $\sigma(t)$. De hecho, cuando iniciamos el entrenamiento de la red, definimos este parámetro con un valor grande, el cual se va disminuyendo a medida que la red progresa en su aprendizaje, lo cual nos garantiza estabilidad en la convergencia del algoritmo de aprendizaje de la red.

Ejemplo

Una vez definidos los valores iniciales de los pesos, procedemos a entrenar la red neuronal, cuya explicación la haremos apoyados en un sencillo ejemplo de clasificación de patrones. En este ejemplo, un mapa auto-organizado es capaz de detectar dos grupos de patrones presentes en un espacio bidimensional como el mostrado en la figura 5.6; naturalmente, este análisis es extensible a un espacio de dimensión *N*.

En la figura, los puntos rojos y azules representan los grupos que la red debe clasificar, es decir, las dos clases de puntos que le vamos a presentar a la red en su entrenamiento. Los puntos verdes representan las posiciones aleatorias, en el espacio bi-dimensional, de las neuronas N_1 y N_2 al iniciar el proceso de aprendizaje.

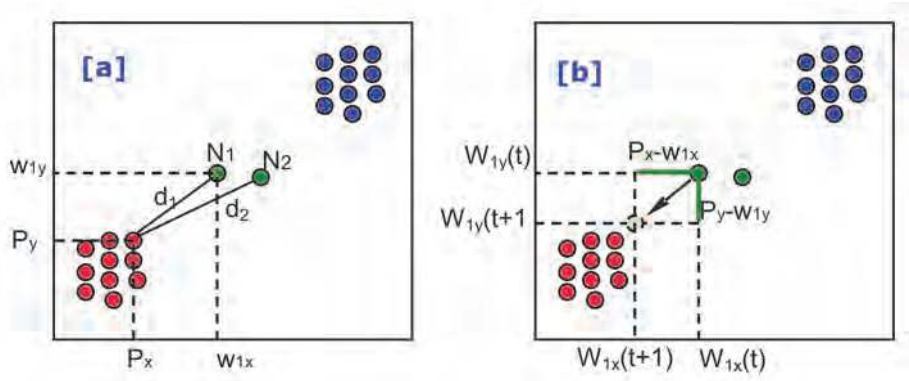


Fig. 5.6 Ilustración del algoritmo de entrenamiento de los SOM

El proceso de entrenamiento comienza seleccionando aleatoriamente alguno de los patrones de entrada, en este caso escogemos un patrón de la clase Rojo, cuyas coordenadas se representan como *(Px , Py),* figura 5.6.a. Las neuronas compiten entre sí para decidir cual de ellas es la más cercana al patrón seleccionado y para tal fin, calculamos las distancias d_1 y d_2, definidas entre el patrón y las neuronas N_1 y N_2 respectivamente. Tal como observamos en la figura, la neurona N_1 es la ganadora pues es la más cercana al patrón de aprendizaje utilizado. Los pesos de la neurona ganadora los representamos con W_{1x} y W_{1y}, cuyos valores debemos modificar con el fin de acercar esta neurona hacia el patrón seleccionado para el entrenamiento.

En la figura 5.6.b, ilustramos como la neurona ganadora modifica sus pesos para acercarse al patrón de entrenamiento presentado, lo que genera un *"movimiento"* de la neurona en el espacio de entrada. La modificación de los pesos de la neurona ganadora se lleva a cabo con base en las ecuaciones 5.5 y 5.6, que provienen de la ecuación 5.1, cuando se desglosa para cada una de las coordenadas.

$$w_{1x}(t+1) = w_{1x}(t) + \alpha(t) h_{rg} \left[P_x - w_{1x}(t) \right] \tag{5.5}$$

$$w_{1y}(t+1) = w_{1y}(t) + \alpha(t) h_{rg} \left[P_y - w_{1y}(t) \right] \tag{5.6}$$

Donde, $W_{1x}(t)$ y $W_{1y}(t)$ representan el valor de los pesos antes de la modificación; $W_{1x}(t+1)$ y $W_{1y}(t+1)$ representan el valor de los pesos después de la modificación y P_x y P_y definen las coordenadas del patrón de entre-

namiento seleccionado. En la figura también mostramos los vectores P_x-W_{1x} y P_y-W_{1y} que son los que determinan la dirección hacia donde evoluciona el nuevo peso. La neurona perdedora N_2 en este caso no sufre modificación de los pesos y el efecto final, es que se queda ubicada en sitio donde comenzó en el proceso de aprendizaje.

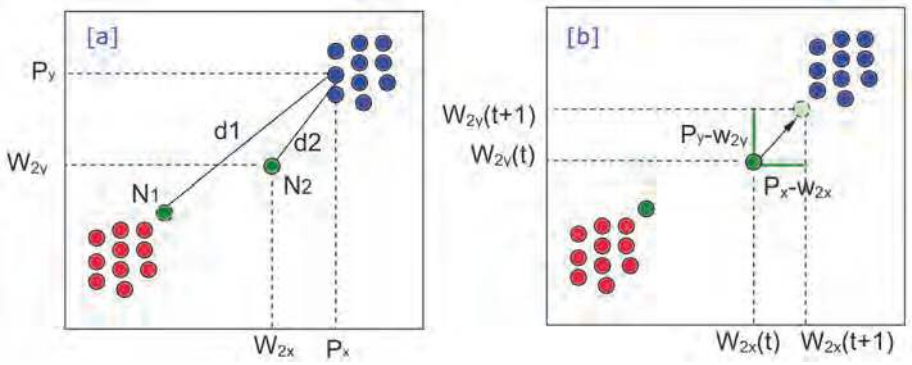


Figura 5.7 Ilustración del algoritmo de entrenamiento de los SOM

En la figura 5.7.a, presentamos las posiciones de las neuronas *N1* y *N2* luego de la primera iteración del proceso de aprendizaje, para continuar con el proceso de entrenamiento seleccionamos otro de los patrones, en este caso seleccionamos un patrón del grupo AZUL de coordenadas *Px* y *Py*. Las neuronas compiten ahora para determinar cuál está más cerca del patrón seleccionado, los valores d_1 y d_2 de la figura 5.7.a corresponden a las distancias del patrón seleccionado a las neuronas N_1 y N_2. Como se desprende de la figura, en este paso la neurona N_2 es la ganadora pues está mas cerca del patrón de aprendizaje utilizado, los pesos de la neurona ganadora se representa con W_{2x} y W_{2y} y son los que modificaremos.

En la figura 5.7.b, ilustramos como la neurona ganadora modifica sus pesos para acercarse al patrón de entrenamiento presentado, lo que genera un *"movimiento"* de la neurona en el espacio de entrada. La modificación de los pesos de la neurona ganadora se lleva a cabo con base en las ecuaciones 5.7 y 5.8.

$$w_{2x}(t+1) = w_{2x}(t) + \alpha(t) h_{rg} \left[P_x - w_{2x}(t) \right] \tag{5.7}$$

$$w_{2y}(t+1) = w_{2y}(t) + \alpha(t) h_{rg} \left[P_y - w_{2y}(t) \right] \tag{5.8}$$

Donde, $W_{2x}(t)$ y $W_{2y}(t)$ representan el valor de los pesos antes de la modificación; $W_{2x}(t+1)$ y $W_{2y}(t+1)$ representan el valor de los pesos después de la

modificación y P_x y P_y definen las coordenadas del patrón de entrenamiento seleccionado. En la figura también mostramos los vectores P_x-W_{2x} y P_y-W_{2y} que son los que determinan hacia donde va a generarse el nuevo peso. La neurona perdedora N_1 en este caso no sufre modificación de los pesos y el efecto final, es que se queda ubicada en sitio donde comenzó en el proceso de aprendizaje.

La figura 5.8 nos ilustra con claridad como las neuronas se acercan hacia los datos de entrada para intentar representar la información presente en ellos. En la primera, vemos la representación aleatoria y luego de dos iteraciones, las neuronas se han acercado lo suficiente a los datos como para representarlos en un proceso de decisión.

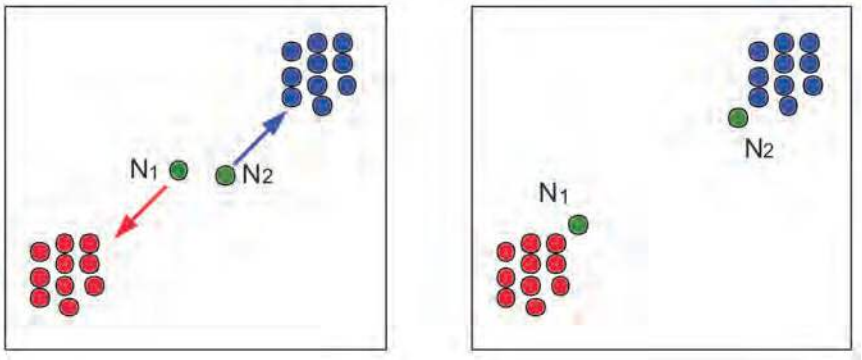


Fig. 5.8 Estado Inicial y Final de las Neuronas en el proceso de Aprendizaje del Mapa de Kohonen

En síntesis, hemos podido ver podemos ver cómo las neuronas se "mueven" en el espacio de entrada para ubicarse sobre los datos de entrada. Por ejemplo, si a un Mapa Auto-organizado de Kohonen le presentamos un conjunto de datos que tienen forma de cuadrícula, en la figura 5.9.a observamos la disposición desordenada de las neuronas de un mapa de Kohonen cuando inicia el proceso de aprendizaje y en la figura 5.9.b, vemos como las neuronas adoptan la forma cuadriculada, luego, de finalizar el proceso de aprendizaje.

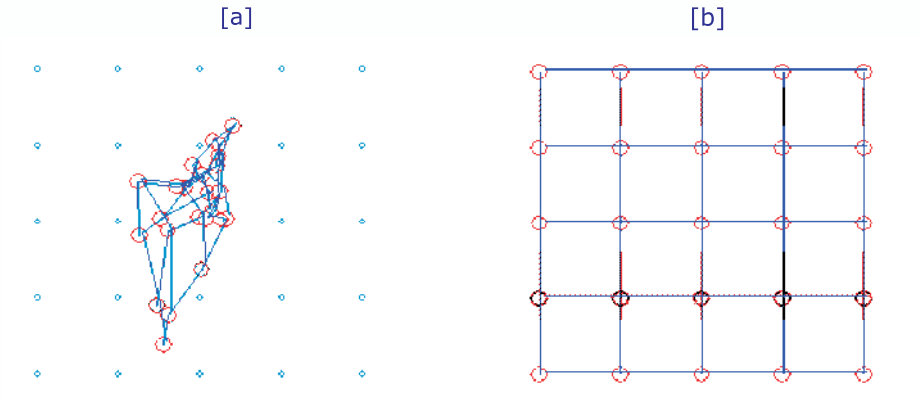


Fig. 5.9 Capacidad de auto-organización de los Mapas de Kohonen (Caso Cuadrícula)

Adicionalmente, en la figura 5.10 vemos un nuevo ejemplo de auto-organización del Mapa de Kohonen, en el cual le hemos presentado a la entrada un conjunto de datos con forma de toroide.

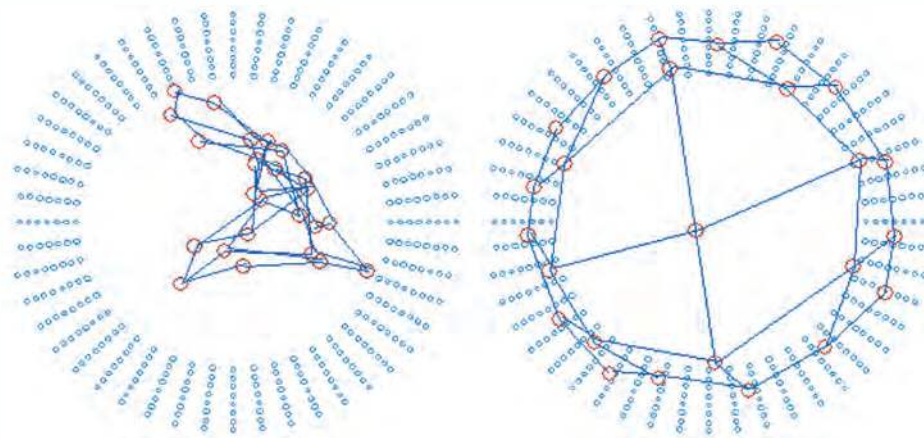


Fig. 5.10 Capacidad de auto-organización de los Mapas de Kohonen (Caso Toroide)

Pasos del algoritmo

MAPAS AUTO-ORGANIZADOS DE KOHONEN
ALGORITMO DE APRENDIZAJE

- Paso 1
  Definimos la arquitectura de la red, con N neuronas en la capa de entrada y en la capa de salida un mapa de M_f x M_c neuronas.
  Cada neurona la definimos con el vector de pesos sinápticos, w_i tiene la misma dimensión R^n del espacio de entrada, cuyos valores iniciales se asignan aleatoriamente.
  Definimos los parámetros de control: σ_i , σ_f, α_i , α_f y t_{max}

- Paso 2
  Seleccionamos un vector de entrada x de manera aleatoria, tal que pertenezca al conjunto de patrones de entrenamiento.

- Paso 3
  Determinamos el índice de la neurona ganadora s con base en la mínima distancia entre el vector de entrada y los vectores de pesos de las neuronas.

$$s = \arg\min \left\| \mathbf{x} - \mathbf{w}_i \right\|$$

- Paso 4
  Modificamos los pesos de la neurona *r-ésima* de acuerdo con:

$$\Delta \mathbf{w}_r = \alpha(t) h_{rs}(t)(\mathbf{x} - \mathbf{w}_r)$$

  donde,

$$h_{rs}(t) = e^{\left(\frac{-d_1(r,s)^2}{2\sigma(t)^2}\right)}, \quad \alpha(t) = \alpha_i \left(\frac{\alpha_f}{\alpha_i}\right)^{\frac{t}{t_{max}}}, \quad \sigma(t) = \sigma_i \left(\frac{\sigma_f}{\sigma_i}\right)^{\frac{t}{t_{max}}}$$

- Paso 5
  Incrementamos el parámetro t

  $t=t+1$

- Paso 6
  Si $t < t_{max}$ retornamos al Paso 2.

PRINCIPIO DE FUNCIONAMIENTO

Una vez hemos finalizado el proceso de entrenamiento de la red neuronal artificial, es común utilizarla para clasificar un conjunto de datos y conocer a que clase pertenece un patrón de entrada. En este tipo de ejemplos, iniciamos presentando el patrón de entrada a la red, lo que origina una competencia entre las diferentes neuronas de la capa de salida, que intentan determinar cual de las neuronas está a menor distancia del patrón presentado, en el espacio *N* dimensional. Cuando el algoritmo define la neurona ganadora, inmediatamente podemos determinar a que clase pertenece el patrón de entrada y, entonces, se finaliza el proceso de clasificación.

Si formalizamos matemáticamente el procedimiento, al Mapa de Kohonen le presentamos el patrón de entrada *k-ésimo* $\mathbf{x}^k=[x_1^k, x_2^k,..., x_i^k,..., x_N^k]$, para calcular la mínima distancia respecto del vector de pesos w_i. La neurona que tenga la mínima distancia, se constituye en la neurona ganadora y definirá la clase a la cual pertenece el patrón de entrada.

Retomemos el ejemplo de la sección 4.3, donde entrenamos una Red de Kohonen con un conjunto de datos que tiene dos clases definidas. En la Figura 5.11.a, vemos el estado final del entrenamiento, donde las neuronas N_1 y N_2, se ubican en los conjuntos de datos rojo y azul, respectivamente. Si le presentamos a la red un nuevo patrón de coordenadas P_x y P_y, representado en la figura como un círculo de color amarillo, la neurona ganadora será la N_1. En la figura 5.11.b, observamos como el patrón presentado ahora es asignado a la clase rojo debido a que la Neurona N_1 fue la ganadora.

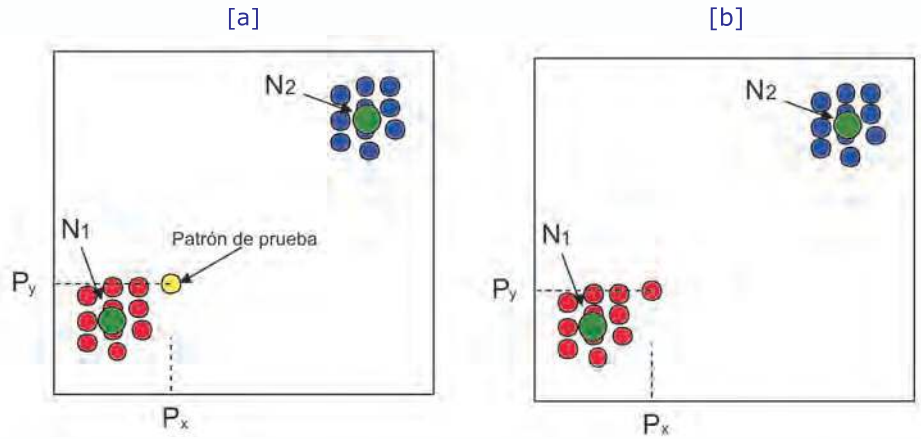


Fig. 5.11 Principio de Funcionamiento del Mapa de Kohonen

Aproximación práctica

Capacidad de reconocer grupos de patrones de un mapa de kohonen

En este primer proyecto tenemos un conjunto de datos conformado por dos clases, el objetivo es diseñar y entrenar una red competitiva para que clasifique estos puntos, para lo cual presentamos el siguiente programa escrito en MATLAB®.

```
%SomClasificador.m
% Este programa nos permite clasificar un conjunto de puntasen el
plano

close all;
clear;

% Generamos el conjunto de patrones
px1=0.05*randn(1,20);
py1=0.05*randn(1,20);
px2=0.05*randn(1,20)+1;
py2=0.05*randn(1,20)+1;

p = [px1 px2; py1 py2];

% Visualizamos los patrones a ser clasificados.
figure
plot(p(1,:),p(2,:),'ob');
hold on;

% Creamos el Mapa de Kohonen
red = newsom(minmax(p),2);

disp('Pesos y Bias Iniciales')
Wini = red.IW{1}
Bini=red.b{1}

W(:,:,1) = red.IW{1};
B(:,:,1)=red.b{1};

% Graficamos las neuronas en la figura
plot(W(:,1),W(:,2),'or');
title('Posición Inicial de las Neuronas')
```

```
hold off;
disp('Oprima una Tecla');
pause;

% Ejecutamos el proceso de aprendizaje de la red
red.trainParam.epochs = 25;
for i=1:20
red = train(red,p);
W(:,:,i+1) = red.IW{1};
B(:,:,i+1)=red.b{1};
end;

% Visualizamos los pesos y umbrales de la red
disp('Pesos y Bias Finales')
Wfin = red.IW{1}
Bfin=red.b{1}

disp('Vectores')
a = sim(red,p)
disp('Indices')
ac = vec2ind(a)

% Graficamos la evolución del proceso de aprendizaje
figure
plot(p(1,:),p(2,:),'ob');
hold on;
X1=W(1,1,:);
X1a=reshape(X1,1,21);
Y1=W(1,2,:);
Y1a=reshape(Y1,1,21);
plot(X1a,Y1a,'or');
plot(X1a,Y1a);

X2=W(2,1,:);
X2a=reshape(X2,1,21);
Y2=W(2,2,:);
Y2a=reshape(Y2,1,21);
plot(X2a,Y2a,'or');
plot(X2a,Y2a);
title('Evolución de las Neuronas')
hold off;
```

En la figura 5.12, observamos la ejecución de este programa y cómo las neuronas del Mapa de Kohonen parten de un sitio aleatorio para con migrar hacia los dos conjuntos de datos a medida que llevamos a cabo el proceso de aprendizaje para cumplir con el objetivo de clasificarlos.

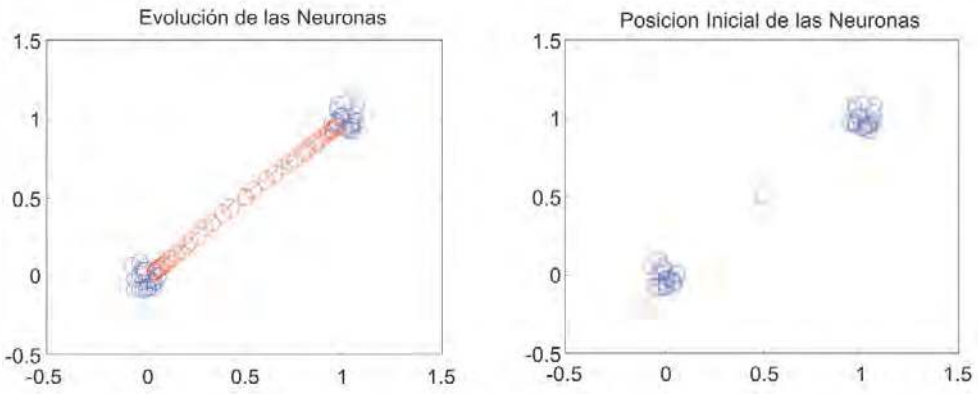


Fig. 5.12 Clasificación de Patrones usando un Mapa de Kohonen

Capacidad de autoorganización de los mapas de kohonen usando MATLAB®

Ejemplo 1: puntos en forma de arco

Con el fin de mostrar la capacidad de autoorganización de los Mapas de Kohonen, vamos a ejecutar el siguiente programa, por medio del cual le presentamos a la entrada de la red un conjunto de puntos que en el plano siguen la forma de un arco, en seguida llevamos a cabo su entrenamiento y visualizaremos el resultado del aprendizaje que hace esta red.

```
%SomArco.m
% Programa que ilustra la capacidad de auto-organización de
los Mapas de Kohonen
% Las neuronas al finalizar seguirán la forma de arco que se
definió en los puntos.
close all;
clear;

% Definimos los patrones de entrenamiento
  angles = 0:0.5*pi/15:0.5*pi;
   P = [sin(angles); cos(angles)];
```

```
% Creamos el Mapa de Kohonen
   net=newsom([0 1;0 1],[10]);

% Visualizamos los datos de entrenamiento y el estado inicial de las
% neuronas de la red
 figure;
  plot(P(1,:),P(2,:),'og');
  hold on;
  plotsom(net.iw{1,1},net.layers{1}.distances);
  disp('Oprima una tecla para continuar');
  pause;

% Definimos los parámetros de entrenamiento
  net.trainParam.epochs=1000;
  net.trainParam.show=100;
% Entrenamos la red
  net=train(net,P);

% Visualizamos los datos de entrenamiento y el estado final de las
% neuronas de la red
  figure;
  plot(P(1,:),P(2,:),'og');
  hold on;
  plotsom(net.iw{1,1},net.layers{1}.distances);
```

En la figura 5.13, vemos el resultado de ejecutar el anterior programa, en donde al iniciar tenemos que todas las neuronas del Mapa de Kohonen están centradas en el punto (0.5, 0.5) del plano. A medida que el programa ejecuta el proceso de entrenamiento, podemos observar como las neuronas paulatinamente buscan los datos de entrada y van asumiendo la forma de arco, hasta que al finalizar obtendremos un resultado como el de la figura, en donde las neuronas han modificado su estructura inicial para asumir la forma de arco que le presentan los datos.

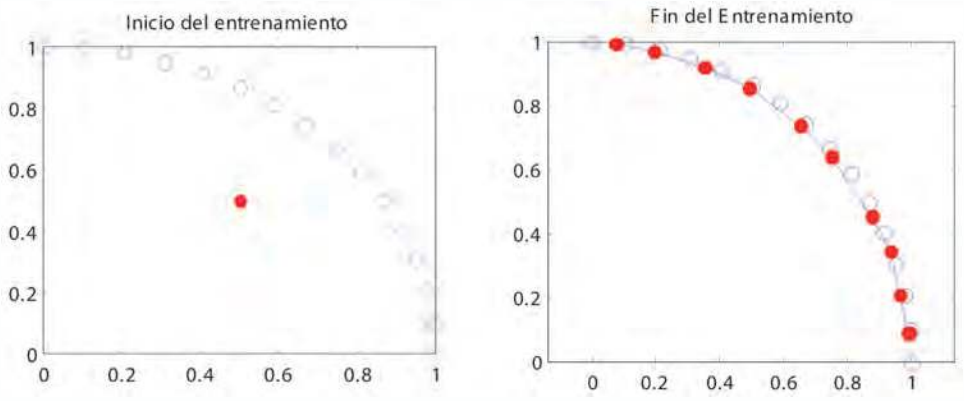


Fig. 5.13 Resultado del proceso de aprendizaje

Ejemplo 2: puntos en forma de cuadrícula

Como un segundo ejemplo para mostrar la capacidad de auto-organización de los Mapas de Kohonen, vamos a ejecutar el siguiente programa, por medio del cual le presentamos a la entrada de la red un conjunto de puntos que en el plano siguen la forma de una cuadrícula, para luego llevar a cabo su entrenamiento y visualizar el resultado del aprendizaje que hace esta red.

```
%SomCuadricula.m
% Programa que ilustra la capacidad de auto-organización de los
Mapas de Kohonen.
% Las neuronas al finalizar seguirán la forma de cuadrícula
definida en los puntos.

close all;
clear;

% Definimos los patrones de entrenamiento
cont=0;
for i=0:0.2:1
  for j=0:0.2:1
     cont=cont+1;
     X(cont)=i;
     Y(cont)=j;
  end;
end;
  P=[X;Y];
```

```
%Creamos el Mapa de Kohonen
net=newsom([0 1;0 1],[5 5],'gridtop');

% Visualizamos los datos de entrenamiento y el estado inicial de
las
% neuronas de la red
   figure;
  plot(P(1,:),P(2,:),'ob');
  hold on;
  plotsom(net.iw{1,1},net.layers{1}.distances);
  disp('Oprima una tecla para continuar');
  pause;

% Definimos los parámetros de entrenamiento
  net.trainParam.epochs=2000;
  net.trainParam.show=100;

%Entrenamos la red
  net=train(net,P);

% Visualizamos los datos de entrenamiento y el estado final de las
% neuronas de la red
  figure;
  plot(P(1,:),P(2,:),'ob');
  hold on;
  plotsom(net.iw{1,1},net.layers{1}.distances);
```

En la figura 5.14, vemos el resultado de ejecutar el anterior programa, en donde al iniciar tenemos que todas las neuronas del Mapa de Kohonen están centradas en el punto (0.5, 0.5) del plano. A medida que el programa ejecuta el proceso de entrenamiento podemos observar como las neuronas paulatinamente buscan los datos de entrada y van asumiendo la forma de cuadrícula, hasta que al finalizar obtendremos un resultado como el de la figura, en donde las neuronas han modificado su estructura inicial para asumir la forma de cuadrícula que le presentan los datos.

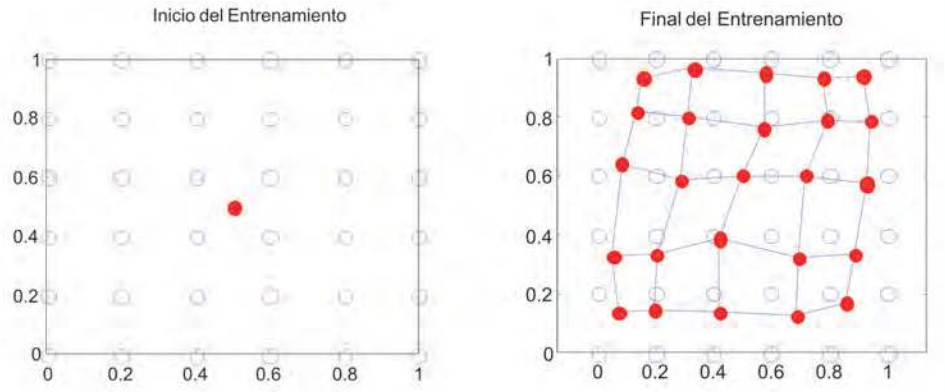


Fig. 5.14 Resultado del proceso de aprendizaje

Capacidad de autoorganización de los mapas de kohonen usando uv-srna

Vamos a ilustrar la capacidad de auto-organización de los Mapas de Kohonen, usando el simulador de redes neuronales UV-SRNA que tiene desarrollado un módulo para simular este tipo de redes competitivas. La interfaz gráfica con el usuario de este módulo lo presentamos en la figura 5.15.

Fig. 5.15 Interfaz de usuario del Móulo RNA Kohonen de UVSRNA

Para seguir la misma metodología propuesta en la sección 6.2, proponemos que el Mapa de Kohonen aprenda una configuración de puntos en forma de arco, por lo creamos un archivo de patrones como el mostrado en la

figura 5.16, que contiene 16 patrones de dos coordenadas, cuyos valores debemos listar tal como ilustra la figura con el fin de ser leídos por UV-SRNA.

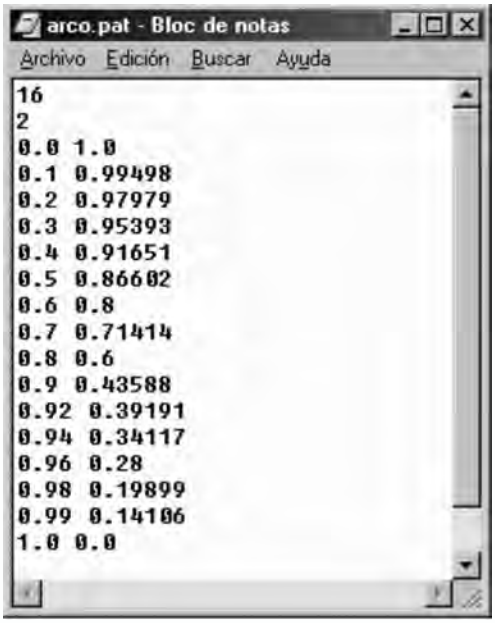


Fig. 5.16 Archivos de patrones.

Una vez editado el archivo de patrones procedemos a invocarlo desde UV-SRNA. Por defecto UV-SRNA crea un Mapa de Kohonen de 25 neuronas organizadas en una matriz de 5x5; pero debido a que los puntos usados definen una curva y no una región en el plano, debemos cambiar la arquitectura de la red a un Mapa de Kohonen de 1x10 neuronas, usando la interfaz de la figura 5.17.

Fig. 5.17 Interfaz para definir la arquitectura de la red

Luego de definir la arquitectura de la red procedemos a su entrenamiento, usando la opción *"entrenar"* de UV-SRNA, quien nos responde mostrándonos la evoluación del proceso de entrenamiento desde un estado inicial de las neuronas, donde su organización es aleatoria, hasta que finalmente éstas adoptan la forma de arco definida por los puntos de entrenamiento. En la figura 5.18, presentamos el avance del entrenamiento cuando se han ejecutado 10 iteraciones y en la figura 5.19, el estado final de la organización de las neuronas luego de 600 iteraciones.

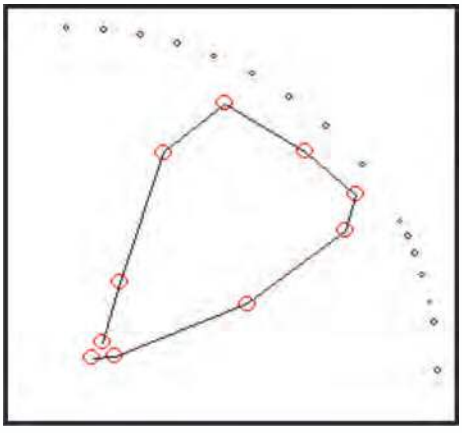


Fig. 5.18 Avance del Entrenamiento con 10 iteraciones

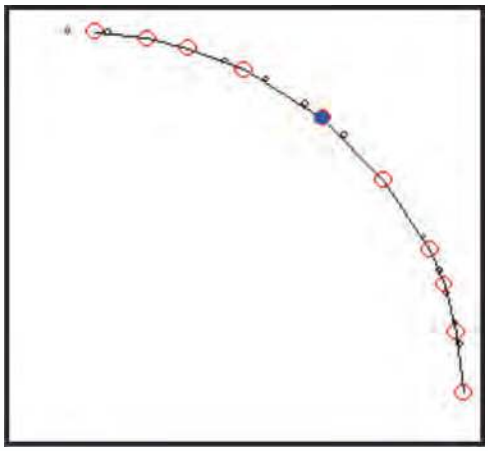


Fig. 5.19 Situación final del entrenamiento

En seguida procederemos a entrenar a un Mapa de Kohonen con un conjunto de datos en forma de cuadrícula, con base en el archivo de la figura 5.21.

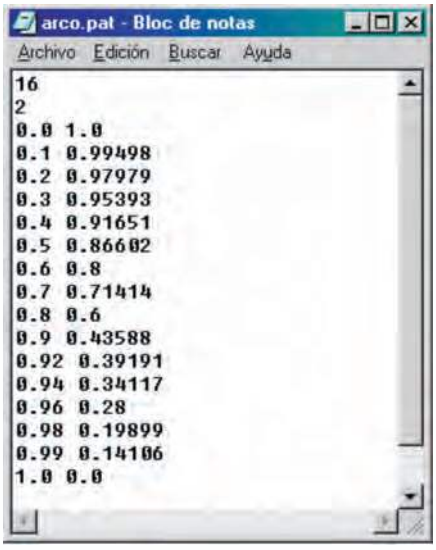


Fig. 5.20 Archivos de patrones.

Por defecto UV-SRNA crea un Mapa de Kohonen de 25 neuronas organizadas en una matriz de 5x5, que por la naturaleza de este ejercicio mantendremos y procedemos a su entrenamiento usando la opción *"entrenar"* de UV-SRNA, quien nos responde mostrándonos la evoluación del proceso de

entrenamiento desde un estado inicial de las neuronas, donde su organización es aleatoria, hasta que finalmente éstas adoptan la forma de cuadrícula definida por los puntos de entrenamiento. En la figura 5.21, presentamos el avance del entrenamiento cuando se han ejecutado 10 iteraciones y, en la figura 5.22, el estado final de la organización de las neuronas, luego, de 3000 iteraciones.

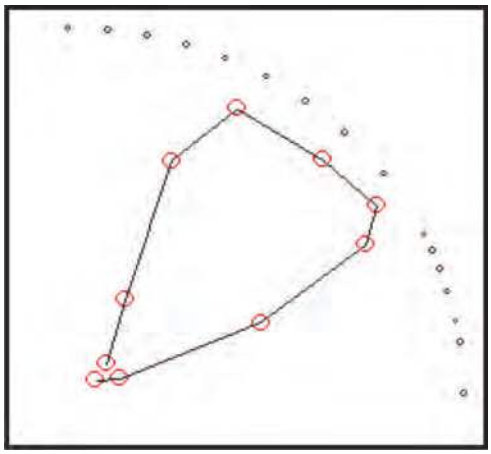


Fig. 5.21 Avance del Entrenamiento con 10 iteraciones

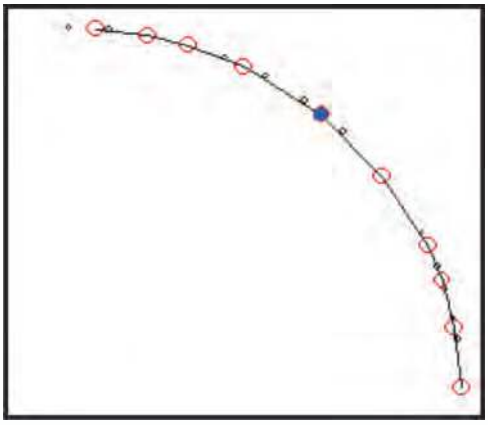


Fig. 5.22 Situación final del entrenamiento

Clasificación de patrones usando mapas de kohonen

El problema del iris fue presentado en la sección 6.7 del capítulo 4, cuando lo resolvimos utilizando una red tipo MLP, Ahora vamos a resolverlo usando un mapa auto-organizado. Para este caso presentamos al mapa todos los ejemplos disponibles para visualizar cuál es la organización que adquie-

ren la neuronas del mapa auto-organizado luego del proceso de aprendizaje. En el siguiente programa escrito para MATLAB®, presentamos los pasos necesarios para el entrenamiento de un Mapa de Kohonen.

```
%Programa que soluciona el problema de Clasificación en la
Base de Datos IRIS

close all;
clear;
clc;
Nneuronas=[10 10]
datosauxiris = load('-ascii','iris_data2.txt');
datosiris = datosauxiris(:,1:4)';
red = newsom(minmax(datosiris),Nneuronas,'gridtop','dist',0.9,
1000,0.1,0);
red.trainParam.epochs=1500;
red.trainParam.show=100;
red=train(red,datosiris);
yred=sim(red, datosiris);
indices=vec2ind(yred);
Mapa=zeros(10,10);
for (i=1:50)
     fila=floor(indices(i)/10)+1;

     columna=round(rem(indices(i),10));
     if columna==0
      columna=10;
   end;
   Mapa(fila,columna)=1;
end;

for (i=51:100)
   fila=floor(indices(i)/10)+1;
   columna=round(rem(indices(i),10));
   if columna==0
      columna=10;
   end;
   Mapa(fila,columna)=2;
end;

for (i=101:150)
   fila=floor(indices(i)/10)+1;
```

```
        columna=round(rem(indices(i),10));
        if columna==0
            columna=10;
        end;
        Mapa(fila,columna)=3;
    end;
    imagesc (Mapa(1:11,1:10)); figure(gcf)
    figure
    plotsomplanes(red)
```

En la figura 5.23, visualizamos la organización con que quedaron las neuronas del mapa de acuerdo a las tres clases definidas para este problema. En los colores rojo, amarillo y azul claro, vemos las neuronas que se activan de acuerdo a las diferentes clases. Podemos resaltar que las neuronas que definen una clase comparten una distribución espacial continua y bien definida. El color azul oscuro representa las neuronas que no se activaron.

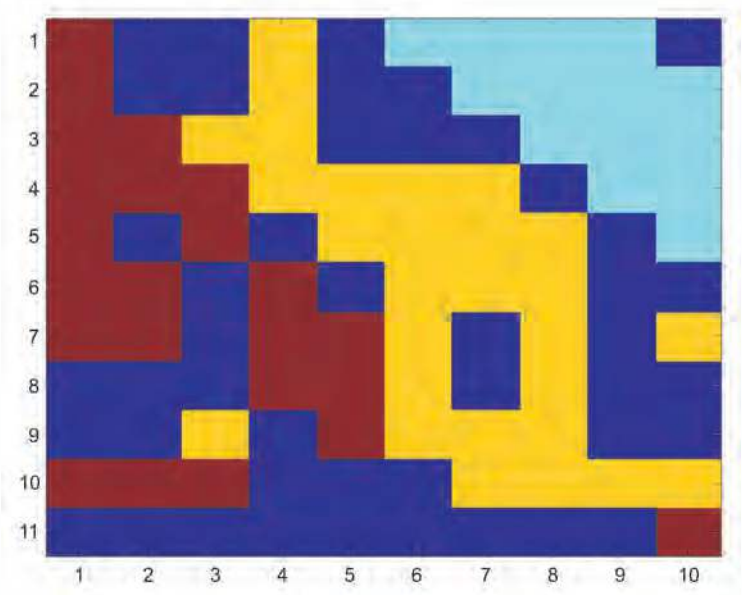


Fig. 5.23 Visualización de la categorias detectadas por el mapa auto-organizado

En la figura 5.24, mostramos la manera como cada una de las cuatro entradas afecta las diferentes zonas del mapa auto-organizado. El color rojo indica que en dicha región existe una alta influencia o valor de dicha característica. El color negro indica que la influencia de dicha característica

es nula. El color amarillo nos indica una influencia intermedia. Esta visualización es lo que se ha llamado el "*análisis de planos*" y, el mismo, nos permite detectar cuáles entradas o características tiene mayor o menor influencia en la determinación de las diferentes clases por parte del mapa auto-organizado.

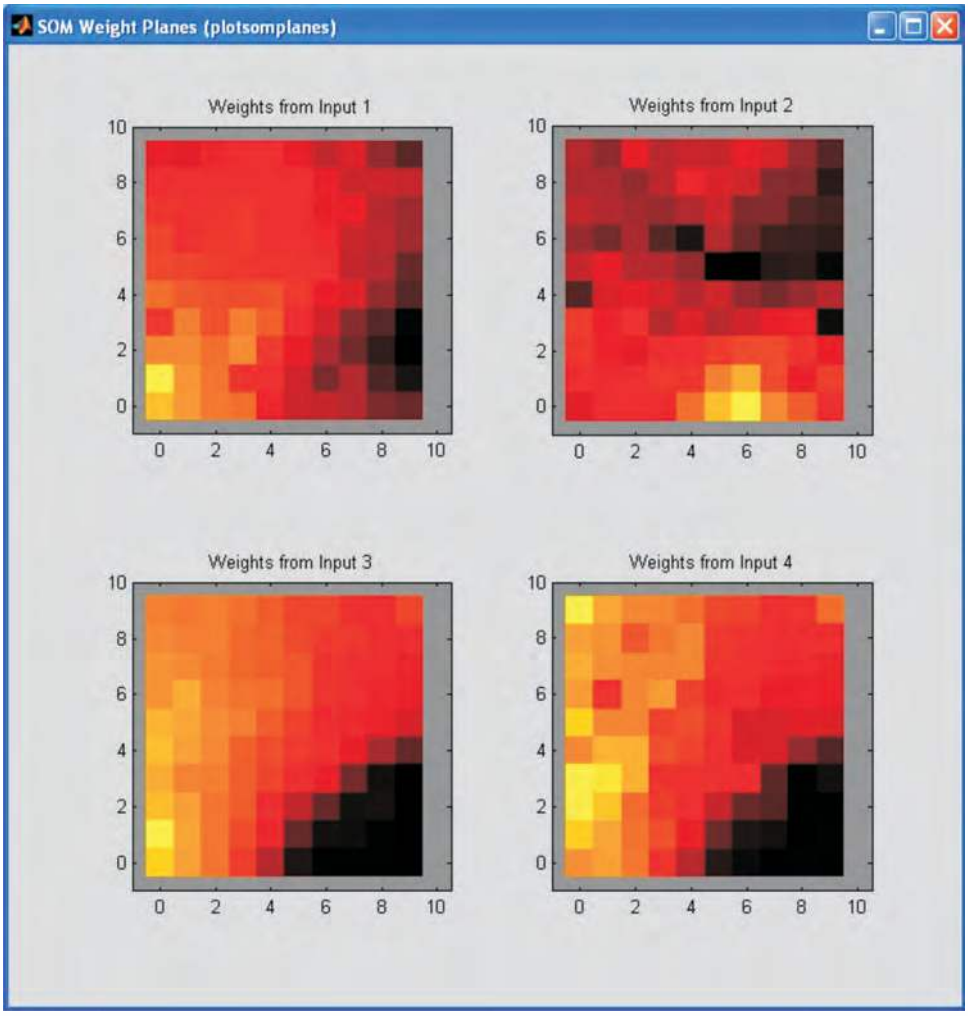


Fig. 5.24 Análisis de planos para las difentes entradas de la red neuronal

Proyectos propuestos

1. Repita el ejercicio del numeral 3.1.1 pero realizando el entrenamiento en UVSRNA.
2. Repita el ejercicio del numeral 3.1.3 pero realizando el entrenamien-

to en UVSRNA.

3. Suponga que tiene unos puntos distribuidos en un plano formando cinco grupos bien definidos como se muestra en la figura 5.9 Realice el entrenamiento en MATLAB® y en UVSRNA de un mapa auto-organizado que realice la sectorización de dichos grupos.

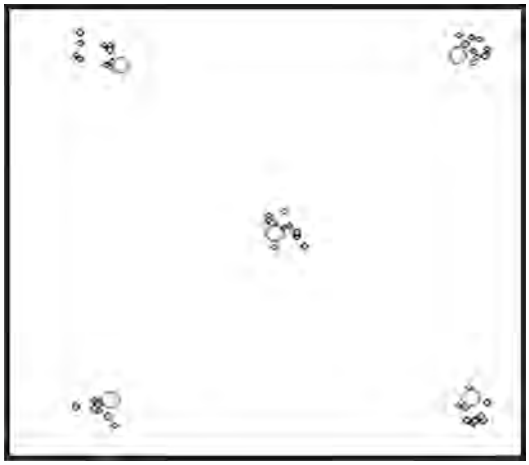


Fig. 5.25 Distribución de 5 grupos bien definidos en el plano

4. Suponga que tiene ocho grupos de bien definidos en un espacio 3D, realice la generación de patrones y el entrenamiento de la red en MATLAB® que permite sectorizar dichos patrones.
5. Suponga que tiene una distribución de puntos en el plano de forma triangular (figura 5.26). Genere dichos patrones y entrene una red en UVSRNA que se ajuste dicha distribución.

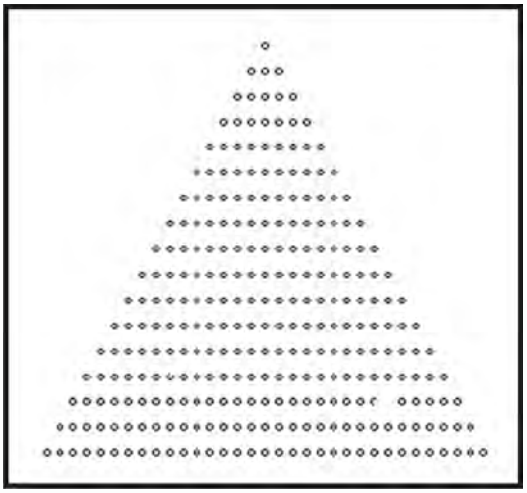


Fig. 5.26 Distribución triángular de puntos en el plano

Capítulo 6

RED DE BASE RADIAL (RBF)

Introducción

En este capítulo nos vamos a enfrentar a un nuevo tipo de redes neuronales que fueron introducidas en 1985. Desde el punto de vista de su arquitectura es una red multicapa unidireccional con aprendizaje híbrido, pues en la capa oculta se sigue un algoritmo No-Supervisado y en la de salida el aprendizaje es Supervisado.

Debido a su simplicidad y velocidad en el proceso de aprendizaje, y el alto grado de generalización, esta red ha sido utilizada en diversas aplicaciones prácticas, sobre todo en reconocimiento y clasificación de patrones. Otro campo de aplicación que ha presentado resultados promisorios es la identificación y modelado de sistemas no lineales.

Las redes de base radial conceptualmente surgen como un caso particular de una red neuronal de regularización que se ha planteado como una herramienta para solucionar problemas generales de interpolación. Por esta razón, en el presente capítulo abordaremos, primero, la definición formal del problema de interpolación, luego plantearemos la estructura de una red de regularización para, finalmente, abordar con detenimiento la red de base radial, objeto de este apartado, incluyendo su arquitectura, modelo matemático, algoritmo de aprendizaje y aplicabilidad.

El problema de interpolación

Antes de abordar las redes RBF, es muy importante revisar el concepto de interpolación. La primera acepción que encontramos de interpolación, está relacionada con la estimación o búsqueda de nuevos valores a partir de un conjunto discreto de valores conocidos.

Otra acepción estrechamente ligada con el problema de la interpolación es la aproximación de una función de alta complejidad, normalmente con un gran número de variables independientes, por una más sencilla. Para el caso de redes neuronales, dicha función además de compleja es desconocida y sólo disponemos de un conjunto de datos experimentales o de simulación que la representan. A partir de estos datos, con un entrenamiento apropiado de la red, obtendremos una función que represente adecuadamente a estos datos conocidos. El siguiente paso es verificar la capacidad de generalización de la red con el fin de garantizar que la red aproxime los datos desconocidos que pertenezcan a la superficie que deseamos modelar.

Para ilustrar el problema de interpolación, partamos de dos conjunto puntos definidos, el primero en un espacio de p dimensiones y el segundo en un espacio de dimensión uno. El problema de interpolación consiste en encontrar una transformación matemática s tal que,

$$s : \Re^p \rightarrow \Re^1 \tag{6.1}$$

donde la transformación se puede visualizar como una hiper-superficie de dimensión $p+1$.

$$\Gamma \subset \Re^{p+1} \tag{6.2}$$

Por ejemplo, si suponemos que $p=1$, debemos encontrar una superficie de dimensión 2 para representar la interpolación, en la figura 6.1 tenemos un conjunto de datos conocidos representados por círculos y vemos como la línea a trazos lleva a cabo la interpolación.

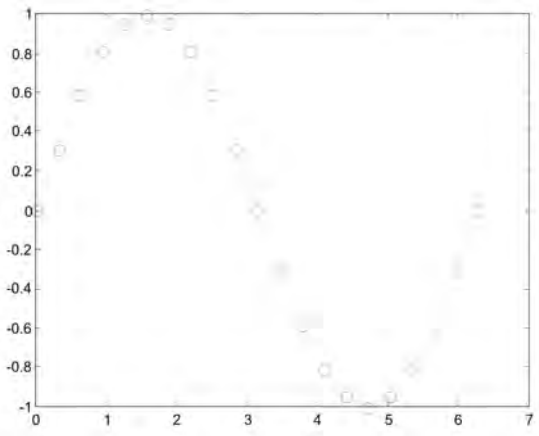


Fig. 6.1 Interpolación de un espacio de dimensión uno a otro de dimensión uno

En segundo ejemplo, supongamos que $p=2$, por lo que en este caso debemos encontrar una superficie de dimensión 3 para representar la interpolación, en la figura 6.2 partimos de un conjunto de datos conocidos representados por círculos y vemos como los asteriscos realizan la interpolación.

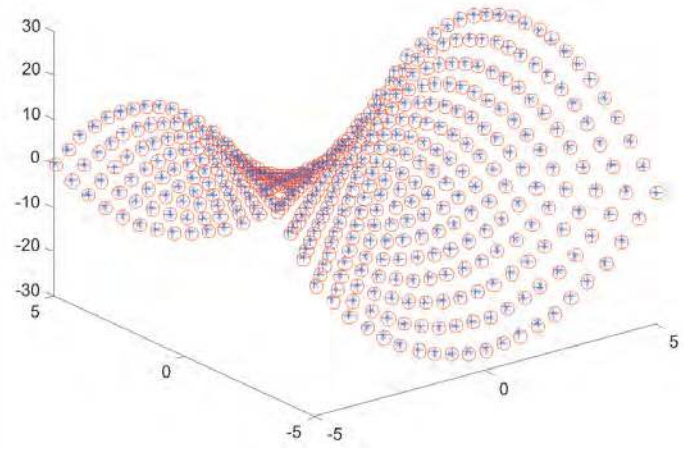


Fig. 6.2 Interpolación de un espacio de dimensión dos a otro de dimensión uno

El problema de interpolación hemos visto que se reduce a encontrar una función que aproxime al conjunto de puntos con un error mínimo. Si utilizamos redes neuronales artificiales para interpolar y el error cuadrático medio como medida de desempeño de la red, tal como lo vimos en el apartado 5.6 del Capítulo 3, la red puede entrar a memorizar los puntos de entrenamiento, que se manifiesta en una función con cambios abruptos y que tienen un mal desempeño ante puntos que estuvieron por fuera del entrenamiento.

Con el fin de evitar este problema de memorización adicionaremos a la función de error, un término de regularización que busca una función aprendida que no pase exactamente por todos los puntos pero que su comportamiento sea más suave, con el propósito de mejorar su capacidad de generalización. Sin embargo, el introducir un nuevo término de regularización trae como consecuencia que se presente múltiples soluciones al problema de interpolación.

En la ecuación 6.3, vemos como la función $\Phi(F)$ a minimizar en el proceso de aprendizaje tiene dos términos, $\Phi(E)$ que a su vez depende el Error Global de la red y $\Phi(F)$ introduce el concepto de regularización afectado por λ, con el fin de disponer un parámetro que nos permita modificar la suavidad de la función de interpolación.

$$\Phi(F)=\Phi(E)+\lambda\Phi(S) \tag{6.3}$$

En la ecuación 4, hemos sustituido la expresión del error global para mayor claridad de la expresión a minimizar.

$$\Phi(F)=\frac{1}{2}\sum_{p=1}^{P}\sum_{k=1}^{M}(d_{pk}-y_{pk})^2+\lambda\Phi(S) \tag{6.4}$$

La solución al problema planteado en la ecuación 4, se puede expresar en términos de la funciones de Green *G,* definidas en términos de la norma euclideana, tal como observamos en la ecuación 6.5.

$$F(\mathrm{x})=\sum_{i=1}^{N} w_i G(\|\mathrm{x}-\mathrm{x}_i\|) \tag{6.5}$$

Esta ecuación la podemos explicar gráficamente en forma de red neuronal, donde la función aprendida *F(x)* es igual a la sumatoria del producto de los pesos sinápticos de la red w_i por la salida de la función de Green cuando es evaluada con la norma que existe entre la entrada a la red y el centroide de la función de Green. La arquitectura presentada en la figura 6.3 se conoce como Red de Regularización, constituida por *N* entradas, una capa oculta con *L* neuronas, donde *L* es igual al número de datos disponibles en el conjunto de entrenamiento y una neurona de salida con función de activación lineal.

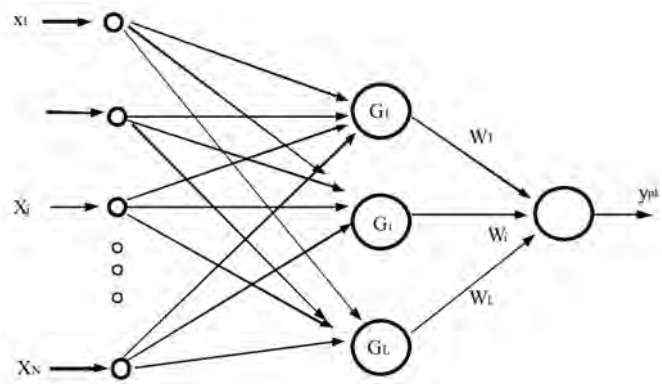


Fig. 6.3 Red de regularización

Las redes de regularización realizan una interpolación exacta, donde en la fase de entrenamiento el centro de cada una de las funciones de Green, se asocia a un dato del conjunto de entrenamiento, por lo que este parámetro ya queda definido. Para calcular los pesos w_i de la neurona de salida parti-

mos de la ecuación 6.6 que muestra la salida deseada de la red *d* en función de las salidas de las funciones de Green y de los pesos.

$$\mathbf{Gw = d} \tag{6.6}$$

Si la inversa de la matriz de funciones de Green existe, podemos obtener el vector de pesos ***w*** con la ecuación 7.

$$\mathbf{w = G^{-1}d} \tag{6.7}$$

Generalmente, las funciones de Green utilizadas son las de base radial como las mostradas en la figura 6.4.

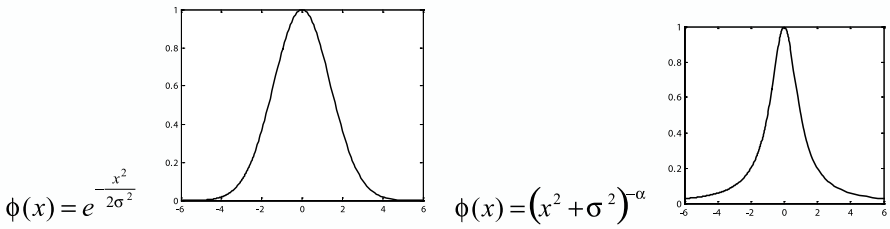


Fig. 6.4 Funciones Base Radial

REDES DE BASE RADIAL

La solución del problema de interpolación exacta usando las redes de regularización, presenta los siguientes inconvenientes:

- El número de funciones de base es igual al número de puntos en el conjunto de datos, lo que claramente es inconveniente sobre todo en aplicaciones complejas cuyo conjunto de entrenamiento suele ser muy grande.
- La interpolación exacta de datos contaminados con ruido es típicamente una función oscilatoria.
- Debido a la proliferación de funciones de Green, cuando se tienen muchos datos de entrenamiento, se puede perder capacidad de generalización.

Estos inconvenientes los podemos salvar planteando las redes de base radial (RBF) gracias a las siguientes propiedades:

- El número de funciones base (*L*) no necesita ser igual al número de datos del conjunto de entrenamiento (*P*), por lo que en la arquitectura

de red RBF, L será mucho menor que P.

- Los centros de las funciones de base no coincidirán con los datos de entrenamiento y, por consiguiente, se convierten en parámetros a modificar en la etapa de aprendizaje de la red RBF.
- El ancho de la función de base también puede ser variable.

Arquitectura de una red de base radial

A primera vista, si revisamos la figura 6.5 no existe una diferencia clara entre la arquitectura de la red neuronal RBF y la del tipo MLP que vimos en sesiones pasadas. Como en el caso de la red neuronal tipo MLP con aprendizaje supervisado, la información fluye desde la capa de entrada hacia la salida de manera unidireccional. En la capa de entrada no existe procesamiento de la información y solo sirve de interfaz entre el mundo real y la RNA, transfiriendo el vector de entrada x a la capa oculta. La capa de salida toma la información proveniente de las neuronas de la capa oculta modificada por los pesos sinápticos existente entre estas dos capas y genera la salida total de la red, a esta neurona de salida se le incluye la entrada de umbral representada por w_0.

La diferencia esencial entre estas dos redes está en la capa oculta, empezando por la función de activación ϕ que ya no es como la que habíamos visto en los otros casos, sino que es una función del tipo base radial, normalmente la función de Gauss. Por esta razón, los parámetros que caracterizan a esta capa oculta son el centro de la función de activación (*centroide*) y su respectiva desviación estándar. Más adelante nos ocuparemos de profundizar en el procesamiento de esta capa oculta.

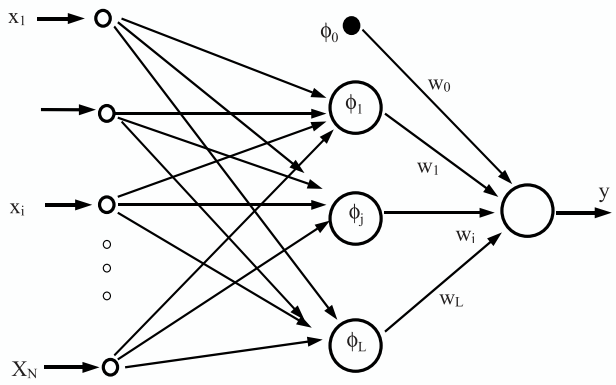


Fig. 6.5 Arquitectura de una red RBF

La capa de entrada corresponde simplemente a una interfaz entre el proceso a analizar o solucionar y la red neuronal. Toma los datos del entorno de trabajo de la red y los lleva a su interior para el procesamiento. La capa

de salida procesa los datos provenientes de la capa oculta y nos entrega la salida de la red.

En la capa oculta se establece la diferencia fundamental con la red tipo MLP, ya que no se calcula la entrada neta y aplica una función de activación sigmoide o lineal como en el caso del Perceptron multicapa, sino que opera con base en la distancia que existe entre el vector de entrada y el centroide de un elemento de la capa oculta, ecuación 8. En esta capa la función de activación es del tipo base radial, por ejemplo la función de Gauss. A esta distancia se le aplica una función de base radial, como por ejemplo la función de Gauss representada en la ecuación 9. La capa de salida procesa los datos provenientes de la capa oculta y como tiene una función de activación lineal la salida corresponde a la suma ponderada de las salidas que proporciona la capa oculta, ecuación 6.10.

$$r_j^2 = \left\| \mathbf{x} - \mathbf{c}_j \right\|^2 = \sum_i \left(x_i - c_{ji} \right)^2 \tag{6.8}$$

$$\phi(r) = e^{-r^2/2\sigma^2}$$
$$\phi(r) = \left(x^2 + \sigma^2 \right)^{-\alpha} \tag{6.9}$$

$$y(\mathbf{x}) = \sum_{j=1}^{L} \omega_j \phi_j(\mathbf{x}) + \omega_0 \tag{6.10}$$

En la figura 6.6, apreciamos el modelo de un elemento de procesamiento de la capa oculta que recibe el vector de entrada ***x*** de dimensión *N*, se calcula la norma respecto del vector centroide c_i, aplicamos la función de activación de base radial para generar la salida *y* de este elemento.

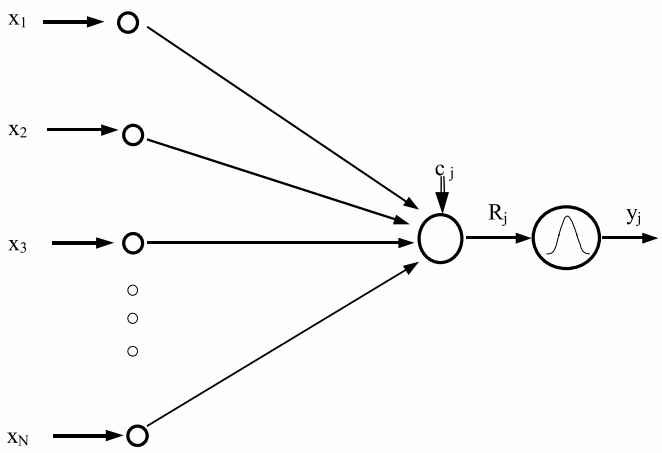


Fig. 6.6 Elemento de procesamiento de una RBF

En la figura 6.7, presentamos la arquitectura de una red *RBF* con *N* entradas, L elementos de procesamiento en la capa oculta con función de activación de base radial y un elemento de procesamiento en la salida con función de activación lineal.

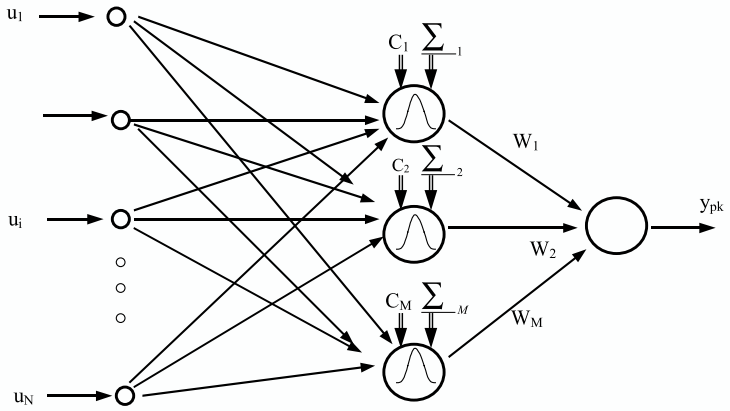


Fig. 6.7 Arquitectura de una red RBF

Entrenamiento de la red rbf

Con este tipo de red hemos planteado que el aprendizaje es híbrido por lo que incluye el aprendizaje supervisado en la capa de salida y el no-supervisado en la capa oculta; por esta razón el entrenamiento de las redes RBF lo dividiremos en dos etapas:

1. El entrenamiento de los elementos de procesamiento de la capa oculta lo realizaremos de forma No-Supervisada, de una manera muy similar a como entrenamos los mapas auto-organizados de *Kohonen*. Presentaremos los datos de entrada y los nodos de procesamiento de esta capa, las unidades de procesamiento ocultas se encargarán de dividirlos por sectores y cada una de las neuronas asumirán la descripción de los mismos en el espacio de entrada a través del vector *centroide*.
2. El entrenamiento de los elementos de procesamiento de la capa de salida lo realizaremos de forma Supervisada con el algoritmo de Corrección de Error, tal como fue descrito para el Perceptron.

Entrenamiento de la capa oculta

El aprendizaje No-Supervisado en estos elementos de procesamiento se cumple en las siguientes etapas:

1. Determinamos el valor de los centroides de los elementos ocultos y para ello utilizaremos el Algoritmo de las k-medias.
   Como esta fase es No-Supervisada no conocemos *a priori* el número de clases o sectores en que se dividen los datos, por lo que se convierte en un parámetro de diseño que depende del tipo de problema y su adecuada selección se hace de manera heurística.
   El paso siguiente es elegir los valores de los centroides C_j de los elementos de procesamiento de esta capa. La elección la podemos hacer de manera aleatoria, pero podemos tomar los primeros *L* patrones del conjunto de aprendizaje, donde *L* es el número de elementos de procesamiento ocultos escogido. Como podremos observar más adelante, el valor inicial de estos centroides es realmente irrelevante.
2. Luego de haber asignado el valor inicial de los centroides, en cada iteración *t* asignamos los patrones de aprendizaje *x* entre los *L* elementos de procesamiento ocultos. Cada patrón se asigna al elemento de cuyo centroide diste menos, con base en los siguientes pasos:
   - Tomamos el patrón *x* del conjunto de entrada y calculamos la distancia a cada uno de los centroides.
   - Verificamos cual es la menor de estas distancias.
   - El patrón de entrada se asigna al sector representado por el centroide cuya distancia haya sido mínima.
   - Repetimos este procedimiento para todos los patrones del conjunto de aprendizaje.
3. Calculamos los centroides de los nuevos sectores generados en el paso anterior, con base en la ecuación 6.11.

$$c_j = \frac{1}{p_j} \sum_{i=1}^{p_j} x_i \tag{6.11}$$

   p_j : Número de patrones que han correspondido a la neurona *j-ésima.*
4. Si hay variaciones en los centroides retornamos nuevamente al paso 2 en caso contrario, finalizamos la etapa de aprendizaje.
5. Calculamos con la ecuación 6.12 la desviación estándar σ_j por cada uno de los sectores encontrados, con el fin de determinar el radio de acción de cada una de los elementos de procesamiento.

$$\sigma_j^2 = \frac{1}{p_j} \sum_{i=1}^{p_j} \left\| x_i - c_j \right\|^2 \tag{6.12}$$

Entrenamiento de la capa de salida

El entrenamiento de los elementos de procesamiento de la capa de salida lo realizaremos de forma Supervisada con el algoritmo de Corrección de Error, tal como fue descrito para el Perceptron. Si conocemos la salida deseada d_{pk} y la salida que produce la red RBF y_{pk} podemos calcular el error global de la red con la ecuación 6.13, considerando a M como el número de elementos de procesamiento en la capa de salida y P la dimensión del conjunto de patrones de entrenamiento.

$$E_p = \frac{1}{2}\sum_{p=1}^{P}\sum_{k=1}^{M}\left(d_{pk} - y_{pk}\right)^2 \tag{6.13}$$

Donde y_{pk} es la salida del *k-ésimo* elemento de procesamiento de la red, cuyo valor determinado por la ecuación 6.10, depende de las funciones de base radial; si re-escribimos esta incluyendo en la sumatoria el término de umbral, obtenemos la ecuación 6.14 y su representación matricial en la ecuación 6.15. Es bueno recordar, que el valor del umbral ϕ_0 es igual a 1.

$$y_{pk}(\mathbf{x}) = \sum_{j=0}^{L}\omega_{kj}\phi_j(r) \tag{6.14}$$

$$\mathbf{y}(\mathbf{x}) = \mathbf{w}\boldsymbol{\Phi} \tag{6.15}$$

De la ecuación 6.15 vemos la dependencia lineal entre los pesos de la capa de salida y la salida de la red, por lo que la matriz de pesos que minimiza el error cuadrático se puede encontrar usando el método de los mínimos cuadrados. Otra forma de calcular esta matriz de pesos es utilizando métodos basados en gradiente como los algoritmos ya estudiados en el capítulo 3, de Gradiente Descendente o Algoritmo de Levenberg Marquardt.

Diferencias entre las redes MLP y RBF

La clasificación de patrones es uno de los campos de mayor aplicación de las redes neuronales artificiales, en la figura 6.8 observamos la forma como llevan a cabo este proceso las redes tipo MLP y RBF.

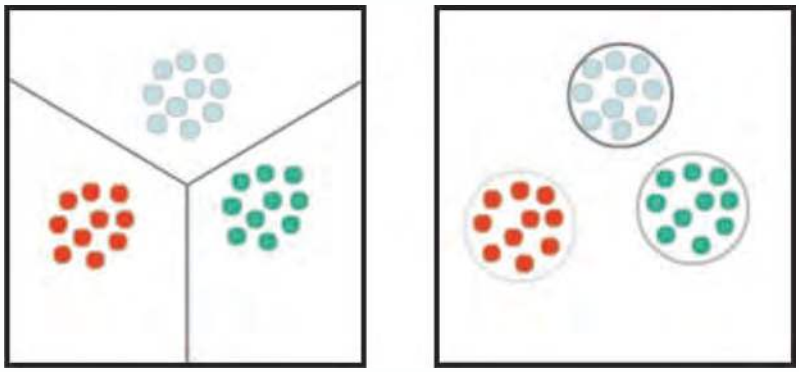


Fig. 6.8 Ejemplo de Clasificación de Patrones usando RNA

a) Clasificación usando MLP **b) Clasificación usando RBF**

Para solucionar este tipo de problemas las redes MLP de aprendizaje tipo *backpropagation*, tienen funciones de activación de respuesta en rango infinito y además las superficies que generan se pueden asimilar a hiperplanos en el espacio de decisión. Para el caso que presentamos en la figura 6.8.a, la información se representa en un espacio de dos dimensiones, la forma como éstas clasifican los datos de entrada es por medio de líneas rectas cuyas intersecciones definen semiplanos, en donde los datos pueden separarse por pertenencia o no a cada uno de ellos. De esta manera definimos tres clases de datos porque pertenecen a tres semiplanos que son delimitados por las líneas rectas.

En este ejemplo estamos limitados, por razones de dibujo, a un espacio de dos entradas; pero podríamos imaginarnos un espacio de *n* entradas, donde las superficies de separación son hiperplanos de *n-1* dimensiones, en donde igualmente se definen sub-espacios que nos separan adecuadamente las diferentes clases de datos existentes en el conjunto de entrenamiento.

Las redes tipo RBF tienen en su capa oculta funciones de activación de base radial, por lo que la separación de las clases es muy diferente al llevado a cabo por la red tipo MLP, tal como observamos en la figura 6.8.b, en ella vemos como se definen sectores alrededor de los datos de entrada, deberíamos pensar que el proceso es inverso, es decir, que los datos están distribuidos en el espacio de entrada de acuerdo a la información que albergan, y es la red la que busca esta organización y se adapta a ella a través del proceso de aprendizaje. Al final, la red RBF genera un número determinado de sectores que agrupan a los datos que poseen características similares en el espacio de entrada, llevando a cabo entonces el proceso de clasificación.

Para entender con más detalle lo acabado de exponer, veamos como una red tipo RBF resuelve el problema de la función lógica *AND* y comparemos con el resultado entregado por un Perceptron y una red tipo MLP cuando

resuelven este mismo problema.

El problema de la función lógica AND, un Perceptron lo resuelve generando una línea recta que separa los puntos con salida en cero (los azules) del punto con salida en uno (el rojo), tal como lo podemos apreciar en la figura 6.9.

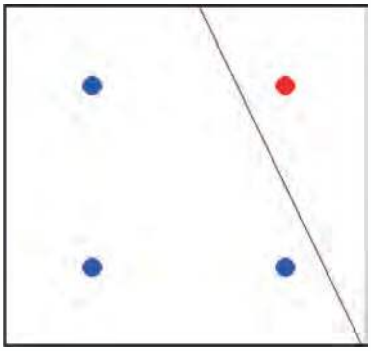


Fig. 6.9 Solución de la función AND con un Perceptron

Una red neuronal tipo MLP con funciones de activación sigmoidales en la capa oculta es capaz de generar una superficie de separación no-lineal para resolver este problema, esta superficie es de tal forma que los puntos con salida en cero queda en una región con salida cercana a cero y el punto rojo queda en una región con salida cercana a uno, tal como ilustramos en la figura 6.10.

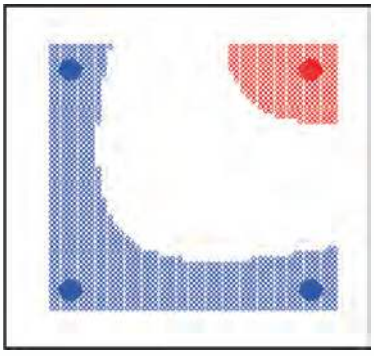


Fig. 6.10 Solución de la función AND con una red tipo MLP

Finalmente, una red tipo RBF para resolver el problema de función *AND*, como vemos en la figura 6.11 solo necesita una neurona gausiana en su capa oculta, dicha neurona se ubica en el punto con salida en uno, de tal manera que cuando a la red llega el patrón (1,1) la distancia de este patrón respecto al centro de la función de Gauss es mínima y, por ende, la salida es máxima, de tal manera que la salida de la red es máxima o en otras palabras tiende

a uno. Los otros patrones están más distantes del centro de la función de Gauss, por lo que su salida tiende a cero.

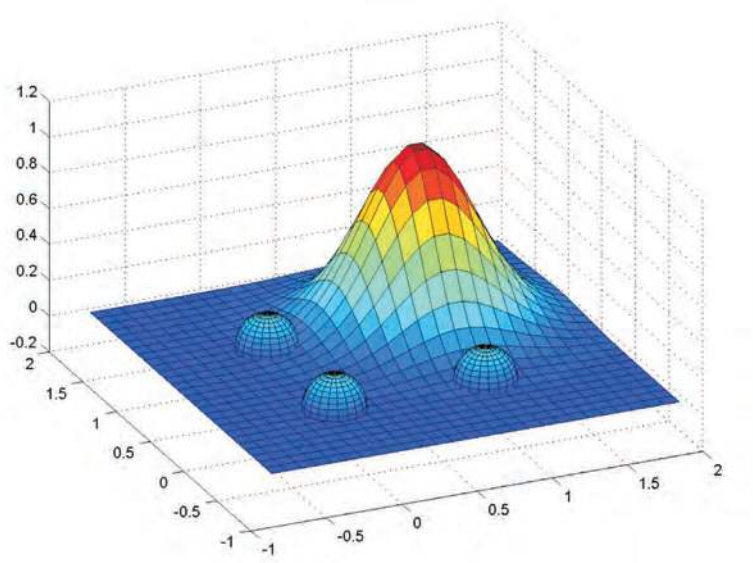


Fig. 6.11 Solución de la función AND con una red tipo RBF

Otro caso más interesante que ilustra las diferencias entre las redes neuronales tipo MLP y RBF, lo podemos ver cuando se resuelve el problema de la función lógica *XOR*. Una red tipo MLP con funciones de activación sigmoidales en la capa oculta es capaz de generar una superficie de separación no-lineal para resolver el problema de la XOR, tal como observamos en la figura 6.12, las salidas de los puntos de entrada (0,0) y (1,1) representados en el plano, se asocian a las superficies no lineales representadas en color azul que representa la salida 0; y las salidas de los puntos (1,0) y (0,1) se asocian a la superficie roja que representa la salida 1.

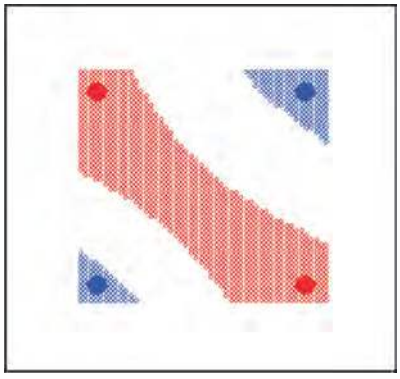


Fig. 6.12 Solución de la función XOR con una red tipo MLP

Una red tipo RBF para resolver este problema, necesita dos neuronas con función de activación gausianas en su capa oculta, dichas neuronas se ubican en los puntos con salida en uno, de tal manera que cuando a la red llega el patrón (0,1) o (1,0) la distancia de este patrón respecto al centroide de esta función es mínima y, por ende, la salida de la neurona es máxima. Los otros patrones están más alejados de los centroides y su salida tiende a cero, tal como observamos en la figura 6.13.

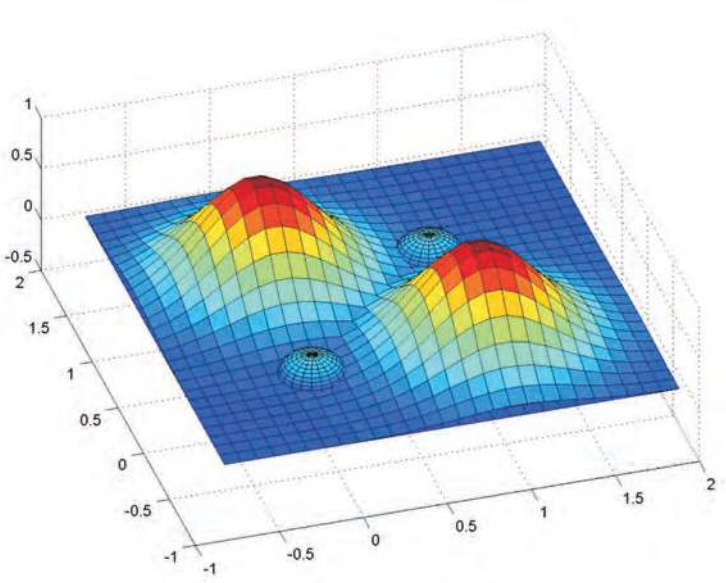


Fig. 6.13 Solución de la función XOR con una red tipo RBF

Para complementar el análisis previo, adicionalmente destacamos las siguientes diferencias que encontramos entre las redes neuronales tipo MLP y RBF:

- *Una red tipo MLP puede tener varias capas ocultas, mientras que en una RBF solo podemos definir una capa oculta.*
  Por definición de la arquitectura, las redes tipo RBF solo tienen una capa oculta debido a la naturaleza de procesamiento que tiene esta capa. En una red MLP es posible tener más de una capa oculta, de hecho en aplicaciones complejas una red MLP con dos capas ocultas puede resolver más rápidamente el problema que una MLP con una sola capa oculta.
- *El tipo de información de entrada a la capa oculta de una red RBF es diferente al que se procesa en una red MLP.*
  La información que llega a la capa oculta de una red tipo MLP es la

suma ponderada de las entradas de la red, mientras que en la red tipo RBF la capa oculta procesa la diferencia entre el vector de entrada y el vector centroide de las neuronas.

- *El procesamiento en la capa oculta en una red tipo MLP es global mientras en una red RBF es local*.
  Esta afirmación es una consecuencia del tipo de funciones de activación que estas redes tienen en su capa oculta, en una red tipo RBF la capa oculta tiene funciones de Gauss cuya activación se encuentra localizada alrededor de su centro donde la salida es máxima, mientras que las redes tipo MLP tienen funciones de activación lineales o sigmoidales en donde la salida máxima se da para un rango mucho mayor de entradas.

Aproximación práctica

Ejemplo de interpolación exacta con MATLAB®

Este capítulo lo iniciamos con una revisión a los conceptos básicos de interpolación exacta, y ahora la primera propuesta práctica es poder llevar a cabo un ejercicio para ilustrar este tipo de aplicación a partir de la función descrita en la ecuación 6.16, cuyos datos de salida están contaminados con ruido blanco centrado en cero y desviación estándar 0.5.

$$y = 0.5 + 0.4\sin(2\pi x) \tag{6.16}$$

La interpolación exacta la vamos a llevar a cabo con una red de regularización con funciones de Gauss para la activación de las neuronas de la capa oculta. Los centros de estas funciones los tomamos, inicialmente, iguales a los patrones de entrada y su desviación estándar es igual a 0.067. El código en MATLAB® que implementa la aplicación lo presentamos a continuación, y debe prestarse especial atención a la forma como se lleva a cabo la inversión de matrices.

```
% Ejemplo de interpolación exacta
close all;
clear;
figure
x=0:1/29:1;
y=0.5+0.4*sin(2*pi*x);
ruido=normrnd(0,0.05,1,length(y));
yruido=y+ruido;
plot(x,y,'--b',x,yruido,'ob');
hold on;
```

```
sigma=0.067;

cx=x;
for i=1:30
   dis=x(i)*ones(1,length(cx))- cx ;
   Phi(i,:)=exp( - ( dis./(2*sigma)).^2 );
end;

% No funciona  a pesar de ser la manera más intuitiva de hacer
la
% Operacion matricial
% W=inv(Phi)*yruido';

% La manera correcta de hacer el cálculo es con la división izqui-
erda, pues de esta
 %manera MATLAB realiza una estimación por mínimos cuadra-
dos de W

W=(Phi)\yruido';
xeval=0:1/290:1;
N=length(xeval);
for i=1:N
dis= xeval(i)*ones(1,length(cx))- cx ;
yco=exp( - ( dis./(2*sigma)).^2 );
yred(i)=(yco)*(W);
end;

plot(xeval,yred,'r'), axis([0 1 0 1.25])
hold off;
```

Finalmente, en la figura 6.14 apreciamos el resultado de hacer una interpolación exacta, donde claramente observamos que la capacidad de generalización de esta red es pobre, toda vez que la línea a trazos representa la función sin contaminación por ruido.

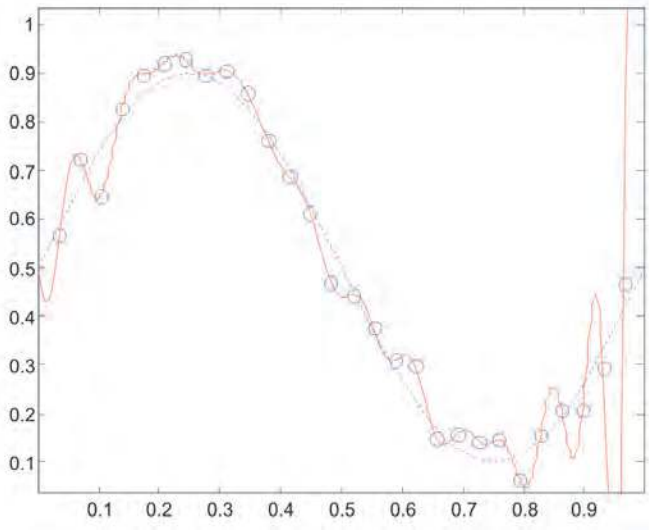


Fig. 6.14 Interpolación exacta realizada con una función de Gauss

Aprendizaje de la función xor

El código MATLAB® que presentamos a continuación nos resuelve el problema de la función lógica *XOR* usando una red RBF. El propósito es ilustrar como este tipo de red puede solucionar problemas no linealmente separables y motivar su estudio para su aplicación en problemas de mayor complejidad.

```
%Solución al problema de la XOR usando una red tipo RBF
close all;
x=[0 0 1 1;
  0 1 0 1];
yd=[0 1 1 0];
red=newrb(x,yd,0,1);
Yred=Sim(red,x);
clc;
disp('La salida de la red entrenada es')
disp(Yred);
```

Aprendizaje de una función de una variable

Con la red neuronal tipo RBF, a diferencia de la red de regularización, ya no hacemos un proceso de interpolación exacta cuando se entrena para aprender una función y con esto ganamos en capacidad de generalización. En este ejercicio de aplicación, buscamos aprender la función descrita con

la ecuación, el objetivo buscado es aprender la función que presentamos en la ecuación 6.17 y cuya descripción gráfica se aprecia en la figura 6.15.

$$f(x) = \text{sen}(x)\cos(2*x) \tag{6.17}$$

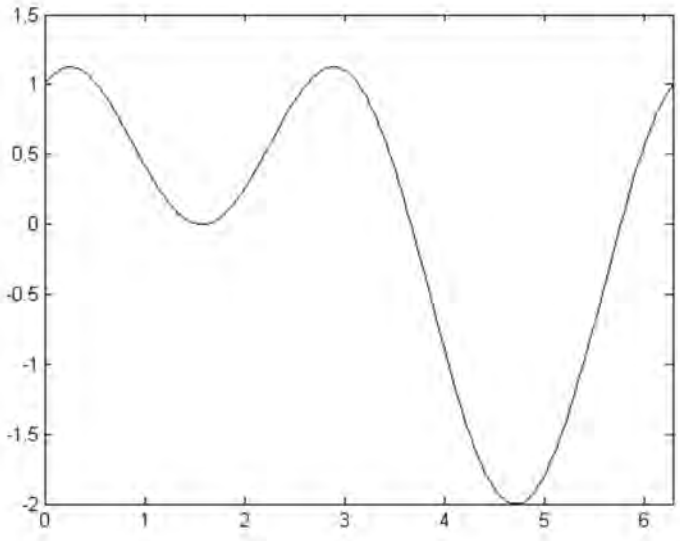


Fig. 6.15 Función a Aproximar con la red tipo RBF

A continuación presentamos un programa escrito para MATLAB® que permite alcanzar el objetivo planteado.

```
%Ejemplo de interpolación usando una red tipo RBF

close all;
x=0:0.2:2*pi;
y=sin(x)+cos(2*x);

% Parámetro de desviación estándar o ancho de la función de
activación de las neuronas de la %capa oculta.
des=1;
red=newrb(x,y,0.002,des);
xval=0:0.1:2*pi;
yred=sim(red,xval);
yval=sin(xval)+cos(2*xval);
figure;
plot(x,y,'ok',xval,yred,'k',xval,yval,'--k');
title('Azul = Original Rojo = Red')
```

Al entrenar la red con este programa, obtenemos una respuesta similar a la de la figura 6.16, donde la línea azul representa la función a aproximar y los círculos rojos corresponden a la salida de la red. Claramente se observa que el proceso de aprendizaje es satisfactorio.

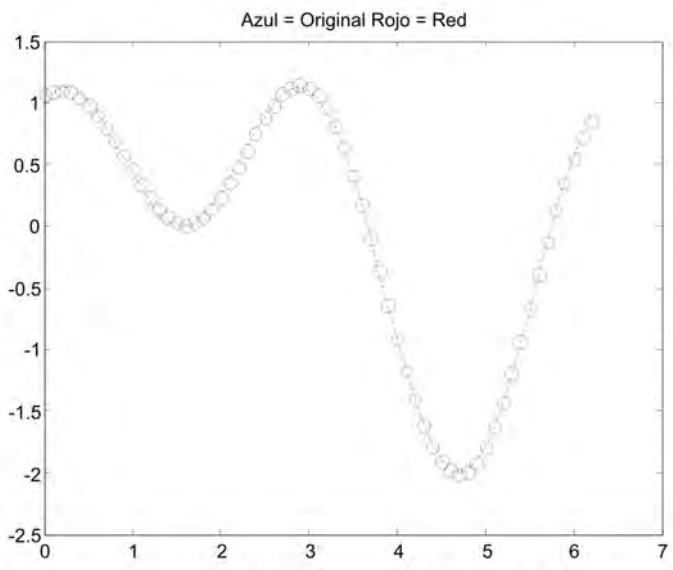


Fig. 6.16 Patrones de entenamiento y Salida de validación de la Red (des=1)

Un aspecto fundamental para la aproximación de funciones con redes RBF, es el parámetro de desviación estándar de la función de Gauss que definimos para las neuronas de la capa oculta. Si el valor de la desviación estándar es muy alto, por ejemplo 100, la salida de la red tiende a generar una interpolación excesivamente suave, tal como apreciamos en la figura 6.17.

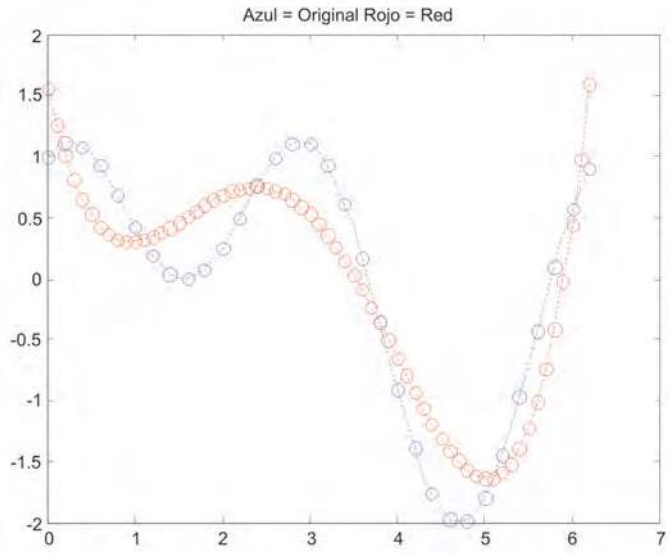
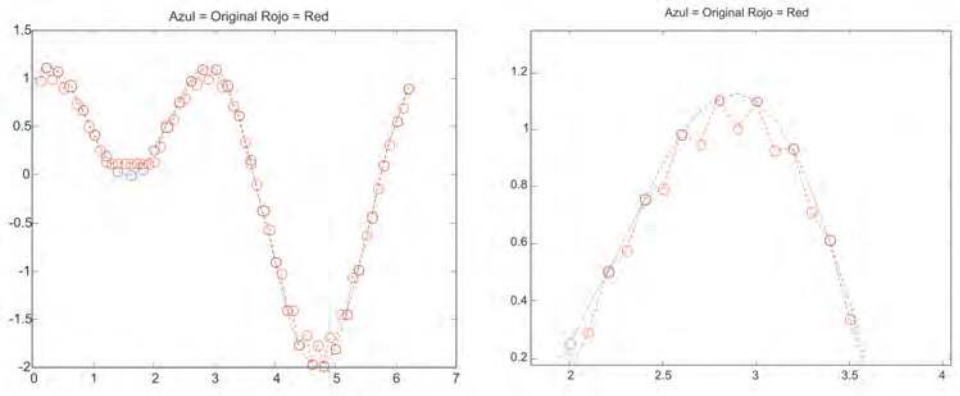


Fig. 6.17 Interpolación cuando la desviación estándar es alta

Si por el contrario el parámetro de desviación se hace muy pequeño, por ejemplo 0.1, la red presenta problemas de sobre entrenamiento y pierde capacidad de generalización, tal como se aprecia en la figura 6.18.



a) **b)**

Fig. 6.18 a) Interpolación cuando la desviación estándar es baja b) Acercamiento para visualizar el sobre-entrenamiento

Identificación de la dinámica de un sistema con una red rbf

En este proyecto asumiremos un reto muy interesante que se soporta en la propiedad de estas redes, de aproximar una función con una precisión predefinida, el objetivo es realizar la identificación de un sistema a partir

de datos experimentales de entrada y salida, que utilizaremos como datos de entrenamiento de la red. El proceso para identificar un sistema dinámico usando redes RBF es similar al utilizado con las redes tipo MLP, que describimos ampliamente en la sección 6.5 del capítulo 3.

En primera instancia vamos a identificar una planta lineal, cuya función de transferencia conocemos y está descrita con la ecuación 6.18.

$$G(s) = \frac{0.5}{(s + 0.5)} \tag{6.18}$$

Diseño del experimento y muestreo de datos

Primero es necesario definir el rango en el que se desea identificar la planta, en este ejemplo el rango de los datos de entrada es [0,1]. Usando la herramienta *simulink* de *MATLAB*® podemos simular el sistema con el esquema de la figura 6.19, obtenemos el conjunto de datos para el entrenamiento de la red.

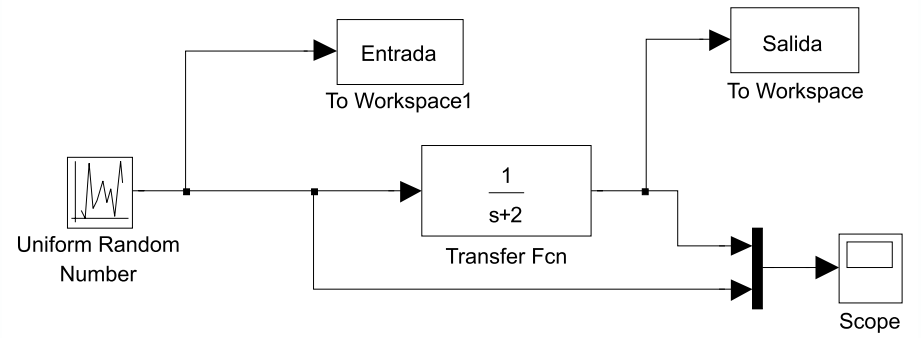


Fig. 6.19 Esquema en Simulink del experimento

El bloque de entrada es un bloque tipo *Band-Limited White Noise* que se encuentra en la biblioteca de *sources*.

El bloque *offset* es un bloque tipo *constante* también de la biblioteca *sources.*

Debemos ajustar los parámetros de estos bloques de manera adecuada, de tal manera que obtengamos datos de entrada y salida como los de la figura 6.20.

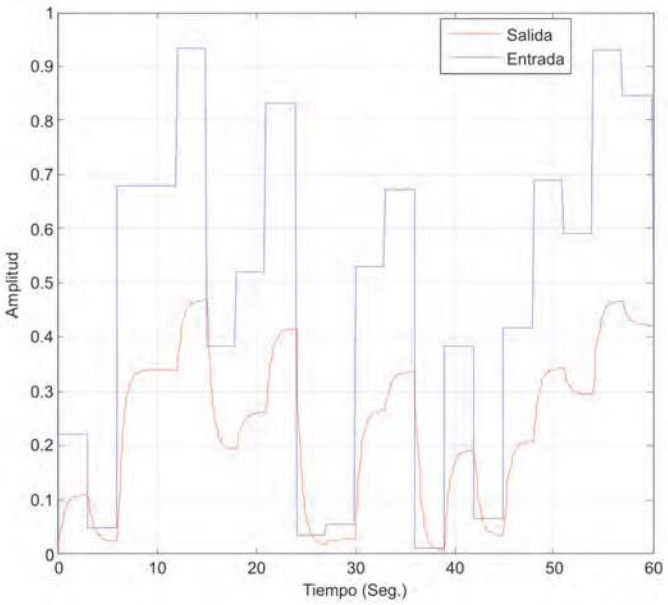


Fig. 6.20 Una posible entrada y su salida obtenida en la Simulación

Recomendaciones:

- El tiempo de muestreo lo debemos ajustar entre 0.1 y 0.2 veces la constante de tiempo más rápida del sistema.
- El ancho de los escalones de entrada debe ser aproximadamente el tiempo de estabilización de la planta.
- La entrada del experimento debe cubrir todo el rango posible de valores de la variable de entrada del proceso.
- Por lo general la entrada del experimento se diseña para que cubra el rango de la entrada del proceso ± el 20%.

Modelo a usar y estimación de parámetros (entrenamiento de la red)

Con los datos experimentales, procedemos a definir una arquitectura de red que sirva para identificar la planta en cuestión. Como la planta es de primer orden y al usar un modelo ARX, se necesitan como entrada de la red, un retardo de la entrada y un retardo de la salida para obtener la salida actual; esto es justamente lo que aplicamos en el regresor del ecuación 6.19.

$$y(k) = f(u(k-1), y(k-1))) \tag{6.19}$$

Lo anterior nos lleva a la siguiente arquitectura de red.

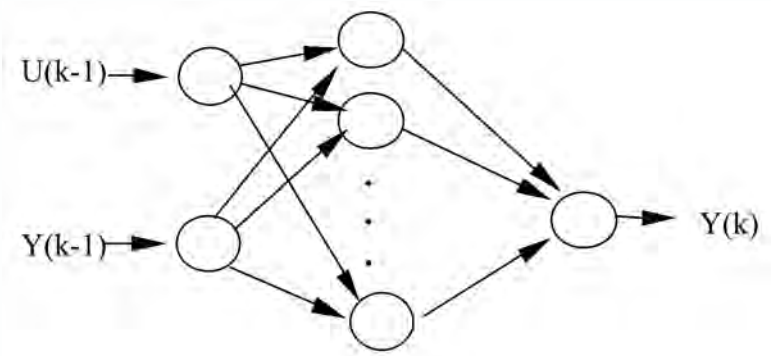


Fig. 6.21 Estructura de la red utilizada para la identificación

Como vemos en la figura 6.21, la red tiene dos elementos en la capa de entrada, en la capa oculta el número de elementos de procesamiento lo define el usuario en la aplicación software que entregamos en el recuadro, y elemento de procesamiento en la capa de salida.

El paso siguiente es organizar los datos que se obtuvieron durante el experimento y definir los patrones de entrenamiento a utilizar en el proceso de aprendizaje de la Red Neuronal Artificial. Esto lo llevamos a cabo construyendo un **.m* en MATLAB®, el cual lo denominaremos *entrenar.m* y aparte de organizar los vectores para obtener las entrada que necesita la red, también realiza el entrenamiento de la red.

Creemos el archivo *entrenar.m* usando el código MATLAB® que se muestra a continuación.

```
% Entrenamiento de una red MLP para la identificación
% de una planta lineal de primer orden
% u debe contener los datos de entrada a la planta.
% y debe contener los datos de salida a la planta.
% u,y deben son vectores columna.

%|  Entrada1 | Entrada2  | Salida |
%=====================================
====
%| U(1 )      |  Y(1)     | Y(2)  |
%| U(2 )      |  Y(2)     | Y(3)  |
%| ..........................................................................
%|   U(K-1)   |  Y(K-1)   | Y(K) |
%
% Se pierde el último dato del vector de entrada U
```

```
close all;

U=Entrada';
Y=Salida';

N=length(U);
X=[U(1:N-1);
   Y(1:N-1)];
Yd=[Y(2:N)];

red=newrb(X,Yd,0.0001,4)
```

Validación del modelo obtenido con la red

Luego de procesar el entrenamiento, se tiene en el objeto *red,* una red que ha aprendido la dinámica de la planta, para validar este modelo se realiza un esquema en la herramienta *Simulink* de MATLAB® como el mostrado en la figura 6.22.

Primero debe obtenerse una representación en diagramas de bloques de la red neuronal que se ha acabado de entrenar, para esto se utiliza el comando *gensim* indicando el tiempo de muestreo usado para la obtención de los datos.

```
>> gensim(red,Ts)
```

donde,

red : Objeto de la red que se ha entrenado
Ts : Tiempo de muestreo seleccionado

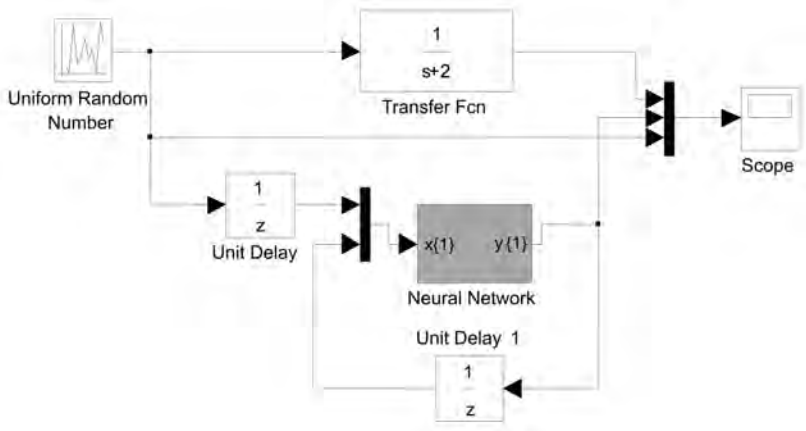


Fig. 6.22 Esquema en Simulink para validar el modelo

Cuando llevamos a cabo la simulación del sistema en *Simulink*, el programa nos entrega una salida como la mostrada en la figura 6.23. Como podemos observar la salida de la red sigue plenamente la salida de la planta original, por lo que aseguramos que la red neuronal tipo MLP aprendió o identificó la dinámica de la planta.

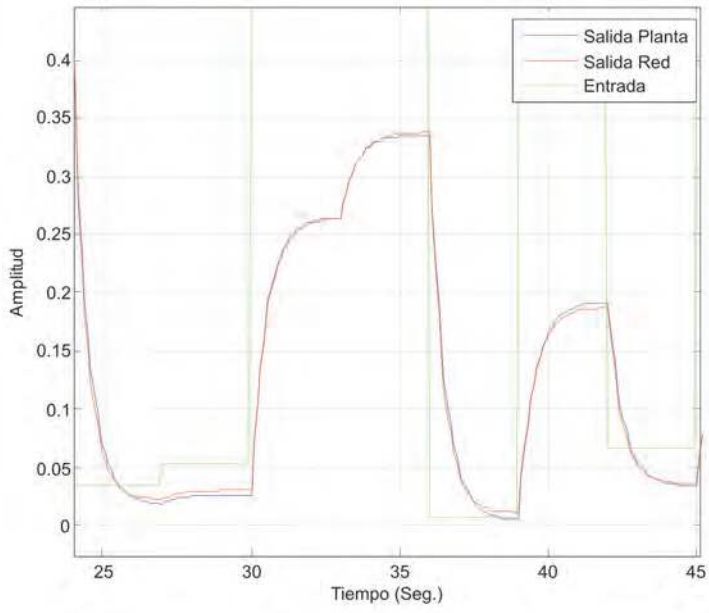


Fig. 6.23 Validación del modelo obtenido

Proyectos propuestos

1. Identifique la siguiente planta usando una red neuronal tipo RBF implemen-tada en MATLAB®. Realice una comparación de resultados con respecto a la solución obtenida con un Perceptron Multicapa en el proyecto 6 del capítulo 3 de este libro.

$$G(s) = \frac{5}{(s^2 + 4s + 10)}$$

2. Entrene una red tipo RBF tanto en MATLAB®, que aprenda la función *(0.5*sin(x)+0.5*sin(2*x))* de tal forma que x esté en el rango $-\pi < x < \pi$. Para validar el entrenamiento verifique la capacidad de generalización de la red entrenada.
3. Entrene una red tipo RBF en MATLAB® que aprenda la función

Seno(x)/x de tal forma que *x* esté en el rango $-10 < x < 10$. Para validar el entrenamiento verifique la capacidad de generalización de la red entrenada.

4. Entrene una red neuronal tipo RBF en MATLAB® que sirva para reconocer las vocales.
5. Entrene una red neuronal tipo RBF en MATLAB® que sirva para reconocer los dígitos del 0-9.
6. Entrene una red tipo RBF en MATLAB® que aprenda la siguiente función de dos variables que se muestra en la figura 6.24. Verifique la capacidad de generalización de la red entrenada.

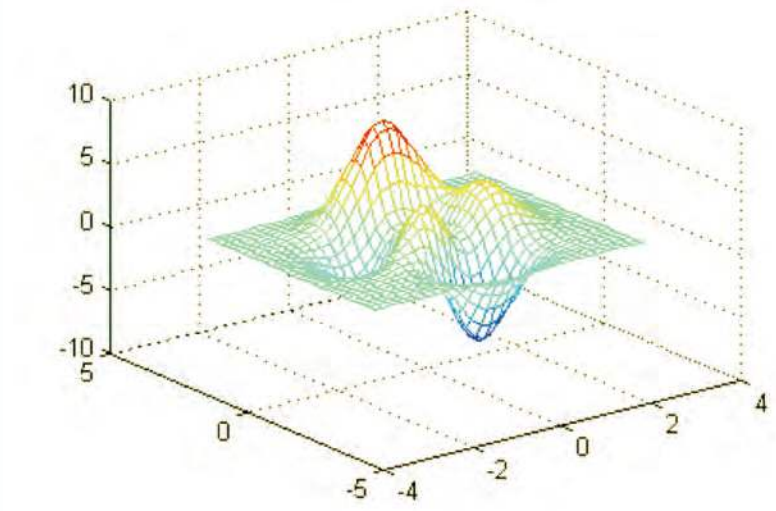


Fig. 6.24 Función de dos variables a identificar

Sugerencia: Utilice el siguiente código para generar la gráfica de la función a aprender.

```
Xini=[-3:0.2:3]; Yini=Xini;
[x,y]=meshgrid(Xini,Yini);
z= 3*(1-x).^2.*exp(-(x.^2) – (y+1).^2) ...
10*(x/5 – x.^3 – y.^5).*exp(-x.^2-y.^2) ...
1/3*exp(-(x+1).^2 – y.^2);
mesh(x,y,z)
```

BIBLIOGRAFÍA

[Barto 1983]
A.R. Barto, R.S. Sutton and C. W. Anderson, "Neuron like adaptive elements that can solve difficult learning control problems", IEEE transactions on Systems, Man, and Cybernetics. Vol. 13 No. 1983.
[Battiti 1992]
R. Battiti, "First and order methods for learning between steepest descent and Newton's method", Neural computation Vol. 4 No. 2 1992.
[Beal72]
Beale, E.M.L., "A derivation of conjugate gradients", in F.A. Lootsma, Ed., Numerical methods for nonlinear optimization, London: Academic Press, 1972.
[Bishop 1995]
Christopher M. Bishop. "Neural Networks for Pattern Recognition", Oxford University Press, USA; 1st edition, 1995.
[Bishop 2007]
Christopher M. Bishop. "Pattern Recognition and Machine Learning", Springer. 2007.
[Cybenko 1989]
G. Cybenko. "Approximation by superposition of a sigmoidal function", Math. of Control, Signals and Systems, Vol. 2 No.4 1989.
[Duda 2000]
Richard O. Duda, Peter E. Hart, David G. Stork. "Pattern Classification (2nd Edition)", Wiley-Interscience. 2000.
[Elman 1990]
J. Elman, "Finding Structure in time Cognitive", Science Vol. 14. 1990.
[Fausett 1993]
Laurene V. Fausett, "Fundamentals of Neural Networks: Architectures, Algorithms And Applications", Prentice Hall; US Ed edition, 1993.
[FlRe64]
Fletcher, R., and C.M. Reeves, "Function minimization by conjugate gradients", Computer

Journal, Vol. 7, 1964, pp. 149–154.
[FoHa97]
Foresee, F.D., and M.T. Hagan, "Gauss-Newton approximation to Bayesian regularization", Proceedings of the 1997 International Joint Conference on Neural Networks, 1997, pp. 1930–1935.
[Foresee 1997]
F. D. Foresee and M. T. Hagan, "Gauss-newton approximation to bayesian regularization", In Proceedings of the 1997 International Joint Conference on Neural Networks, 1997.
[Freeman 1993a]
J. Freeman, D. Skapura, "*Redes neuronales, Algoritmos, Aplicaciones y Técnicas de Programación*", Editorial Addison-Wesley/Diaz de Santos, 1993.
[Freeman 1993b]
J. Freeman, D. "Simulating Neural Networks with Mathematica", Addison-Wesley, 1993.
[Hagan 1994]
M. T. Hagan and Menhaj M. B., "Training feedforward networks with the Marquardt algorithm". IEEE Trans. on Neural Networks, Vol. 5 No. 6, 1994.
[Hagan 1996]
Martin T. Hagan, Howard B. Demuth, and Mark H. Beale, "Neural Network Design", PWS Publishing Company, 1996.
[Hagan 2008]
Martin T. Hagan, Howard B. Demuth, and Mark H. Beale, "MATLAB® Neural Networks Toolbox User's Guide V 6.0". The Mathworks Inc. 2008.
[Hassoun 1995]
Mohamad H. Hassoun "Fundamentals of Artificial Neural Networks", The MIT Press. 1995.
[Hastie 2003]
T. Hastie (Author), R. Tibshirani (Author), J. H. Friedman, "The Elements of Statistical Learning", Springer. 2003
[Haykin 1994]
S. Haykin, "*Neural networks a Comprehensive Foundation"*, First Edition, Prentice Hall, 1994.
[Haykin 1999]
S. Haykin, "*Neural networks a Comprehensive Foundation"*, Second Edition, Prentice Hall, 1999.
[Hebb 1949]
D. O. Hebb, "The Organization of Behavior", Wiley. 1949.
[Hilera 1996]
José Hilera y Victor Martínez. "Redes Neuronales Artificiales. Fundamentos, modelos y aplicaciones". Alfa Omega – Rama, 1996.
[Hinton 1989]
G. E. Hinton, "Connectionist learning procedures", Artificial Intelligence Vol. 40 1989.
[Hopfield 1982]
J.J. Hopfield, "Neural networks and physical system with emergent collective computational abilities", Proceedings of the National Academy of Science, Vol. 79 1982.
[Hopfield 1984]
J.J. Hopfield. "Neurons with graded response have collective computational properties like

those of two-state neurons", Proceedings of the National Academy of Science, Vol. 81 1984.
[Hornik 1989]
K. Hornik, "Multilayer feedforward networks are universal approximators", Neural Networks, No. 2, 1989.
[Hunt 1991]
K. Hunt and D. Sbarbaro, "Neural Networks for Non-Linear Internal Model Control", IEE proceedings-D, vol. 138, No. 5. UK 1991.
[Ienne 2005]
P. Ienne and G. Kuhn, "Digital Systems for Neural Networks", Digital Signal Processing Technology, volume CR57 of Critical Reviews Series, pages 314-45. SPIE Optical Engineering, Orlando, Fl. USA. 1995.
[Kecman 2001]
Vojislav Kecman. "Learning and Soft Computing. The MIT Press", 2001.
[Kohonen 2000]
Teuvo Kohonen, "Self Organizing Maps", Springer – Verlag, 2000.
[Kosko 1987]
B. Kosko, "Adaptive bidirectional associative memory", Applied Optics Vol. 26. 1987.
[Kosko 1988]
B. Kosko, "Bidirectional associative memories", IEEE transactions on Systems, Man, and Cybernetics. Vol. 18 No.1 1988.
[Kosko 1992]
B. Kosko, "Neural Networks and Fuzzy Systems", Prentice Hall 1992.
[LeCun 1998]
Y. LeCun, L. Bottou, G.B. Orr and K.-R. Muller, Efficient backprop, "Neural Networks-Tricks of the Trade", Springer Lecture Notes in Computer Sciences 1524, pp.5-50, 1998.
[Lippman 1987]
R.P Lippman,., "An introduction to computing with neural nets," IEEE ASSP Magazine, 1987, pp. 4–22.
[Lippman 1989]
R.P Lippman,., "Pattern Classification using Neural Networks," IEEE Communication Magazine, 1989, pp. 47-64.
[Looney 1997]
Carl G. Looney. "Pattern Recognition and Neural Networks, Theory and Algorithms for Engineers and Scientists", Oxford University Press, 1997.
[MacKay 1992a]
D. J. MacKay, "Bayesian interpolation", Neural Computation, Vol. 4, No. 3 1992.
[MacKay 1992b]
D. J. MacKay. "A practical bayesian framework for backpropagation networks", Neural Computation, Vol. 4, No. 3 1992.
[Masters 1993]
Timoty Masters, "Practical Neural Network Recipes in C++", Morgan Kaufmann, 1993.
[Masters 1994]
Timoty Masters, "*Signal and Image Processing with Neural Networks: A C++ Sourcebook"*, Wiley, 1994.
[Masters 1995]

Timoty Masters, “Advanced Algorithms for Neural Network: A C++ Sourcebook”, John Wiley & Sons Inc 1995.
[Martín del Brío 2007]
Bonifacio Martín del Brío, Alfredo Sanz Molina, “Redes Neuronales y Sistemas Borrosos”, Alfa Omega – Rama, 2007.
[McCulloch 1943]
W. McCulloch and W. Pitts “A logical calculus of the ideas immanent in nervous activity”, Bulletin of mathematical Biophysics Vol. 5, 1948.
[Michie 1994]
D. Michie, D.J. Spiegelhalter, C.C. Taylor (eds) “Machine Learning, Neural and Statistical Classification”, Ellis Horwood. 1994.
[Minsky 1969]
M. Minsky and S. Papert, “Perceptrons”, MIT Press. 1969.
[Moll93]
Moller, M.F., “A scaled conjugate gradient algorithm for fast supervised learning”, Neural Networks, Vol. 6, 1993.
[Nabney 2004]
Ian T. Nabney, “Netlab” Springer, 2004.
[Narendra 1990]
K. S. Narendra and K. Parthasarathy, “Identification and control of dynamical systems using neural networks”, IEEE Trans. on Neural Networks, Vol. 1 No. 1 1990.
[Neal 1996]
R. M. Neal, *“Bayesian Learning for Neural Networks”*, Springer-Verlag, 1996.
[NgWi90]
Nguyen, D., and B. Widrow, “Improving the learning speed of 2-layer neural networks by choosing initial values of the adaptive weights”, Proceedings of the International Joint Conference on Neural Networks, Vol. 3, 1990, pp. 21–26.
[Norgaard 2000]
Norgaard, M., “Neural Networks for Modeling and Control of Dynamic Systems”, Springer – Verlag, London. 2000.
[Park 1991]
J. Park and I. Sandberg, “Universal approximation using radial basis function networks”, Neural Computation, No. 3, 1991.
[Pham 1995]
X. Pham, D. LIU. “Neural Networks for Identification, Prediction and Control”, Springer-Verlag, London, UK, 1995.
[Powell77]
Powell, M.J.D., “Restart procedures for the conjugate gradient method”, Mathematical Programming, Vol. 12, 1977, pp. 241–254.
[Principe 1999]
José C. Principe, Neil R. Euliano, W. Curt Lefebvre, “Neural and Adaptive Systems: Fundamentals through Simulations”, Wiley, 1999
[Ripley 1996]
Brian D. Ripley, “Pattern Recognition and Neural Networks”, Cambridge University Press, 1996
[Rosenblatt 1961]

F. Rosenblatt, “Principles of Neurodynamcis”, Spartan Press 1961.
[Rosenblatt 1958]
F. Rosenblatt “The Perceptron: A probabilistic model for information storage and organization in the brain”. Psychological Review. Vol. 65 1958.
[RuHi86a]
Rumelhart, D.E., G.E. Hinton, and R.J. Williams, “Learning internal representations by error propagation”, Parallel Data Processing, Vol. 1, Cambridge, MA: The M.I.T. Press, 1986, pp. 318–362.
[Rumelhart 1986]
D. E. Rumelhart G. e Hinton and R. J. Williams, “Learning representations by back-propagating errors”, Nature Vol 323 1986.
[Rusell 1997]
Russell D. Reed, Robert J. Marks “II Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks”, The MIT Press. 1999.
[Shepherd 1997]
A. Shepherd Second-Order, “Methods for Neural Networks”, Springer-Verlag 1997.
[Vapnik 1995]
V. Vapnik, “The nature of Statistical Learning Theory”, Springer Verkag. 1995.
[Vries 1992]
Vries B. and J. Principe, “The gamma model: A new neural networks model for temporal processing”, Neural Network Vol 5., 1992.
[Widrow 1990]
Bernard Widrow and Hoff M. “30 years of adaptive neural networks: Perceptron, madaline and backpropagation”, Proceedings of the IEEE, Vol. 78. No. 9, 1990.
[Werbos 1990]
Paul Werbos, “Backpropagation through time: What it does and how do it”, Proceeding IEEE, vol. 78, No. 10, Oct. 1990, pp 1550-1560.

Programa Editorial

Ciudad Universitaria, Meléndez
Cali, Colombia
Teléfonos: (+57) 2 321 2227
321 2100 ext. 7687
http://programaeditorial.univalle.edu.co
programa.editorial@correounivalle.edu.co