

T  
519.232  
LAB



**ESCUELA SUPERIOR POLITECNICA DEL LITORAL**

**Instituto de Ciencias Matemáticas**

**“USO DE REDES NEURONALES EN EL ANALISIS DE SERIES DE TIEMPO”**

*Tesis de Grado*

Previa a la obtención del título de:

**INGENIERO EN ESTADISTICA INFORMATICA**

Presentada por:

José Miguel Laborde Navas



**GUAYAQUIL - ECUADOR**

**AÑO**

**1999**



\*D-19580\*

## AGRADECIMIENTO

Agradezco a Dios por darme la capacidad y la fuerza que necesité todo este tiempo. A mis padres y hermana por haberme apoyado pacientemente a lo largo de toda mi carrera universitaria hasta estas últimas instancias, Al M.Sc. Gaudencio Zurita por haber compartido su experiencia y sabiduría conmigo y con mis compañeros. Al M.Sc. Carlos Jordán por haberme dado las ideas claves en la elaboración de esta tesis.



## DEDICATORIA

A mi Familia, y a la que vendrá.



**TRIBUNAL DE GRADUACION**

4.2) =

---

**M.Sc. Carlos Jordán**  
Director de Tesis

---

**Ing. Félix Ramírez**  
Presidente

---

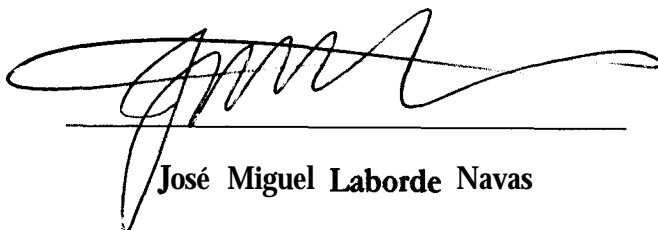
**Phd. Holger Capa**  
Vocal

---

**Ing. Adelaida Albán**  
Vocal

## **DECLARACION EXPRESA**

“La responsabilidad del contenido de esta Tesis de Grado, me corresponden exclusivamente; y el patrimonio intelectual de la misma a la ESCUELA SUPERIOR POLITECNICA DEL LITORAL”



Handwritten signature of José Miguel Laborde Navas, consisting of a stylized, cursive script with a large initial 'J' and 'M'.

**José Miguel Laborde Navas**

## **RESUMEN**

En el presente trabajo “USO DE REDES NEURONALES EN EL ANÁLISIS DE SERIES DE TIEMPO”, hemos hecho una labor investigativa en una de las ramas de la Inteligencia Artificial como lo son los Sistemas Neuronales Artificiales. Lo que se desea como última instancia es hacer predicciones de una serie de tiempo con esta herramienta computacional sin hacer el modelamiento por Análisis de Series de Tiempo.

En la primera parte de la tesis presentamos de forma muy general los aspectos concernientes a la teoría del Análisis de Series Temporales. Los aspectos que cubriremos son en su mayoría los vistos en el curso impartido en la carrera Ingeniería en Estadística Informática. Para esto nos hemos apoyado en el clásico “Time Series Analysis” de Box y Jenkins. Advertiremos que la presencia del primer capítulo sirve para justificar el diseño neuronal que hagamos posteriormente.

En el segundo capítulo hemos cubierto lo concerniente a redes neuronales. Nos hemos extendido en muchos puntos debido a que este tipo de conocimiento no es de uso común en nuestro medio y al principio puede causar cierta desconfianza debido a las grandes ambiciones que se plantean las redes neuronales. Haremos una descripción biológica, resumen histórico, descripción de una red en particular con su forma de aprender, mencionaremos que nos dicen los expertos con respecto a su experiencia, y al final haremos una aproximación de la red que predice series de tiempo.



El tercer capítulo trata sobre la familiarización que debemos tener con software diseñado para modelar redes neuronales. Con una forma en lo posible pedagógica, destacaremos las características de un paquete computacional neuronal más importantes para poder hacer predicciones de una serie de tiempo. Para esto utilizaremos un paquete llamado *Trajan 3.0*, pero pudimos haber utilizado cualquier otro de haber dispuesto del mismo.

En el cuarto capítulo aplicamos los conocimientos vistos en los 3 anteriores capítulos para efectuar las predicciones de una serie de tiempo; envolviendo diseño, y análisis de los resultados, utilizando datos reales y de uso muy conocido.

Finalizaremos con nuestras conclusiones y recomendaciones, en las que involucraremos toda nuestra experiencia asimilada a lo largo del desarrollo de esta tesis, con el fin de que se haga más sencilla la réplica de este trabajo por parte de las personas interesadas en el tema.



## INDICE GENERAL

	<b>Pág.</b>
<b>RESUMEN</b>	II
<b>INDICE GENERAL</b>	IV
<b>INDICE DE FIGURAS</b>	VI
<b>INDICE DE TABLAS</b>	VIII
<b>INDICE DE CUADROS</b>	VIII
<b>INTRODUCCION</b>	9
<b>CAPITULO 1</b>	
<b>1 Análisis De Series De Tiempo</b>	11
1.1 Introducción.	11
1.1.1 Predicción de una serie de tiempo.	12
1.2 Procesos estacionarios y funciones de autocorrelación.	14
1.3 Modelos estacionarios lineales.	17
1.4 Procesos no estacionarios lineales.	19
1.5 Identificación, estimación, chequeo y diagnóstico del modelo.	21
1.6 Modelos estacionales.	26
1.7 Observaciones.	28
<b>CAPITULO II</b>	
<b>2 Sistemas Neuronales Artificiales</b>	29
2.1 Introducción.	29
2.2 Descripción desde el punto de vista biológico.	31
2.3 Descripción de los ANS.	35
2.4 El perceptrón .	39
2.4.1 Modelo de Frank Rosenblatt.	39
2.4.2 Problema XOR.	42
2.5 Modelo de redes de propagación hacia atrás.	46
2.5.1 Introducción.	46
2.5.2 Descripción de la estructura BPN.	47
2.5.3 Ley de Aprendizaje.	49
2.5.3.1 Regla delta generalizada.	49
2.5.3.1.1 Obtención de los pesos de la capa de salida.	52
2.5.3.1.2 Cambio de pesos en la capa oculta.	56
2.5.4 Criterios prácticos y de convergencia.	57
2.5.4.1 Datos de Entrenamiento.	58
2.5.4.2 Dimensionamiento de la red.	59
2.5.4.3 Pesos y parámetros de velocidad de aprendizaje.	60
2.5.5 Estructura BPN que resuelve el problema de predicciones en series de tiempo.	62
<b>CAPITULO III</b>	
<b>3 Uso de un paquete computacional de redes neuronales.</b>	65
3.1 Introducción.	65
3.2 Un ejemplo: programa de Karsten Kutza en C++.	66
3.3 Un software sofisticado de redes neuronales que permite hacer predicciones de series de tiempo: Trajan 3.0	67



3.3.1	Sobre el archivo de datos de entrenamiento.	68
3.3.2	Arquitectura Multicapa.	69
3.3.3	Con respecto al aprendizaje de la red.	71
3.3.4	Proyección de series de tiempo.	74
3.3.5	Observaciones	76
<b>CAPITULO IV</b>		
<b>4</b>	<b>Aplicación y análisis de los modelos empleando datos de una serie temporal.</b>	<b>77</b>
4.1	Introducción.	77
4.2	Predicción del número anual de manchas solares.	78
4.2.1	Modelización de la serie por un proceso estacional.	78
4.2.2	Predicción de la serie de <b>Wölfer</b> mediante una BPN.	82
4.2.3	Comparación de las predicciones de manchas solares.	83
4.3	Predicción del número mensual de pasajeros de aerolíneas en EEUU.	85
4.3.1	Predicción mediante un modelo SARIMA.	85
4.3.2	Diseño y uso de una BPN para predecir el número mensual de pasajeros.	87
4.3.3	Comparación entre valores reales, predicciones SARIMA y predicciones BPN.	91
4.4	Observaciones Finales.	93
	<b>Conclusiones y recomendaciones.</b>	<b>95</b>
	<b>ANEXOS</b>	<b>100</b>
	Anexo 1: Código del programa Bpn de Karsten Kutza.	101
	Anexo 2: Corrida completa del programa Bpn de Karsten Kutza	111
	<b>BIBLIOGRAFIA</b>	<b>118</b>

**INDICE DE FIGURAS**

		<b>Pág.</b>
<b>Figura 1.1</b>	Serie de tiempo con predicciones en intervalos de confianza	13
<b>Figura 1.2</b>	Serie temporal estacionaria	15
<b>Figura 1.3</b>	Función de autocorrelación estimada	17
<b>Figura 1.4</b>	Recta $xt = a + bt$ .	19
<b>Figura 1.5</b>	Recta después de la diferenciación	20
<b>Figura 1.6</b>	Autocorrelaciones sobresaliendo de la banda.	22
<b>Figura 1.7</b>	Autocorrelaciones parciales (1 fuera de la banda).	23
<b>Figura 1.8</b>	Predicciones de una serie de tiempo.	25
<b>Figura 1.9</b>	Serie Estacional de orden 12.	26
<b>Figura 1.10</b>	Serie estacional con una pendiente notoria.	27
<b>Figura 2.1.</b>	Célula nerviosa	32
<b>Figura 2.2.</b>	Sinápsis 0 unión sináptica	33
<b>Figura 2.3.</b>	Unidad básica de procesamiento o neuronal artificial	36
<b>Figura 2.4.</b>	Red Neuronal Básica	37
<b>Figura 2.5.</b>	Fotoperceptrón	40
<b>Figura 2.6.</b>	El perceptrón de F. Rosenblatt	41
<b>Figura 2.7.</b>	Un problema de clasificación de patrones linealmente separables	43
<b>Figura 2.8.</b>	Problema XOR (O-Exclusiva)	44
<b>Figura 2.9.</b>	Un perceptrón multicapa que resuelve a XOR	45
<b>Figura 2.10.</b>	Red de Propagación hacia Atrás (general)	47
<b>Figura 2.11.</b>	Estructura BPN al detalle	50
<b>Figura 2.12.</b>	Superficie de Error (supuesta) en base a pesos y su gradiente	52
<b>Figura 2.13.</b>	Función Sigmoide	55
<b>Figura 3.1.</b>	Definiendo el tipo de datos para una serie temporal en Trajan3.0	68
<b>Figura 3.2.</b>	Creando una red en Trajan 3.0	70
<b>Figura 3.3.</b>	Ilustración de Trajan 3.0 de una red MLP	71
<b>Figura 3.4.</b>	Entrenamiento por BPN en Trajan 3.0	72
<b>Figura 3.5.</b>	Gráfico del error de verificación y entrenamiento al entrenar la red por <b>Back Propagation</b> en Trajan 3.0	72
<b>Figura 3.6.</b>	Proyección de Serie de Tiempo en Trajan 3.0	75
<b>Figura 4.1.</b>	Serie ajustada del número anual de manchas solares de Wölfer	79
<b>Figura 4.2.</b>	Autocorrelaciones serie manchas solares	79
<b>Figura 4.3.</b>	Autocorr. Parciales serie manchas solares	79
<b>Figura 4.4.</b>	Gráfico de las predicciones de la serie de manchas solares	80
<b>Figura 4.5.</b>	Predicciones de la BPN y SARIMA en comparación con los datos reales "SUNSPOTS"	83
<b>Figura 4.6.</b>	Gráfico de la serie de pasajeros (132 observaciones)	86
<b>Figura 4.7.</b>	Definiendo los datos como "input/output" en Trajan 3.0	87

<b>Figura 4.8.</b>	Diseño y creación de la red predictora en Trajan 3.0	<b>88</b>
<b>Figura 4.9.</b>	Cambiando la función de salida (de sigmoide a lineal) en la última capa de la red	89
<b>Figura 4.10.</b>	Resultados después de correr 10000 “epochs” en Trajan 3.0	<b>90</b>
<b>Figura 4.11.</b>	Gráfico de las 132 observaciones, predicciones y simulaciones	92
<b>Figura 4.12.</b>	Gráfico de las 12 últimas observaciones vs predicciones BPN y proceso SARIMA	92

### INDICE DE TABLAS

		Pág.
<b>Tabla 4.1</b>	Media cuadrática del error de Predicción de manchas solares	<b>84</b>
<b>Tabla 4.2</b>	Predicciones de la manchas solares	<b>84</b>
<b>Tabla 4.3</b>	Predicciones del número de pasajeros	<b>91</b>
<b>Tabla 4.4</b>	MCE de las predicciones sobre los 132 casos	<b>92</b>

### INDICE DE CUADROS

		Pág.
<b>Cuadro 1.1</b>	Corrida de un modelo ARMA en Systat 7.0	<b>23</b>
<b>Cuadro 1.2</b>	Corrida de un modelo AR en Systat 7.0	<b>24</b>
<b>Cuadro 4.1</b>	Corrida del modelo SARIMA(2,0,0) (0,0,2) II en Systat 7.0	<b>81</b>
<b>Cuadro 4.2</b>	Corrida de BPN para manchas solares	<b>82</b>

## INTRODUCCION

Predecir eventos en el futuro es una idea ambiciosa para una persona que no tenga la habilidad psíquica para hacerlo. Sin embargo el hombre junto con su imaginación ilimitada a creado estructuras matemáticas que le permiten incursionar en el ámbito predictivo fuera de lo esotérico. Hermann Hankel decía: *“En la mayoría de las ciencias, una generación derriba lo que otra ha construido, y lo que una ha establecido otra lo destruye. Solamente en matemáticas, cada generación construye un nuevo piso sobre la vieja estructura”*. La base de la estadística es la matemática, y la base del análisis de series de tiempo es estadística – matemática, por lo que ahora tenemos una gran “estructura” teórica que nos permite hacer predicciones de variables aleatorias (que evidentemente corresponden a eventos concretos).

Pero tal y como era de esperarse en esta época de adelantos tecnológicos y de microtecnología, con computadores personales tan poderosos, capacidad de almacenamiento y velocidad de transmisión de datos tan altos, el campo de la inteligencia artificial se ha desarrollado a gran escala. Uno de los aspectos de la IA (inteligencia artificial) es el aprendizaje por modelos conexionistas: Las redes neuronales artificiales.

Vamos a intentar recrear la capacidad predictiva que tiene el modelamiento de series de tiempo mediante su análisis estadístico – matemático con la ayuda de las redes neuronales. Es de esperarse que las nuevas generaciones empleen otro tipo más adelantado de modelo “inteligente” con que llevar a cabo esta tarea, ya que como dijo Hankel todas las ciencias derrumban sus propias teorías, menos la matemática, y la IA no es la excepción. De todas

formas la IA y particularmente la Redes Neuronales han estado en desarrollo por más de medio siglo y a estas alturas, cuando estamos a punto de entrar al nuevo milenio, las computadoras y el software parecen estar estandarizandose cada vez mas. Lo que sí parece estar en constante cambio (mejoramiento) es la velocidad de procesamiento y la capacidad de almacenamiento de datos, lo cual es beneficioso para todo aspecto del mundo moderno.



## **CAPITULO 1**

### **1 Análisis de Series de Tiempo.**

#### ***1.1 Introducción.***

En este primer capítulo vamos a presentar de una forma muy general lo que son las series de tiempo. El propósito de hacer referencia hacia la teoría del análisis de series de tiempo tiene que ver directamente con el modelo de red neuronal que vayamos a escoger para hacer predicciones, es decir (y como veremos al final del capítulo 2), que necesitamos repasar ciertos aspectos de las series de tiempo para poder entender bien nuestro problema y luego poder hacer un buen planteamiento a partir de la naturaleza de los datos que se tenga.

Como dijimos, en este capítulo nos vamos a dedicar a describir el problema del análisis de series de tiempo y la naturaleza de los datos. Veremos de una forma muy breve (ya que el

describir toda la teoría del análisis de series de tiempo escapa del contexto de esta tesis) lo que son autocorrelaciones, datos estacionarios y no estacionarios, modelos lineales, además de describir lo que son datos estacionales. Una vez que comprendamos y tengamos claro nuestro problema podremos tener más elementos de juicio además de herramientas de diseño para sugerir un modelo solución.

Si el lector desea conocer más acerca del análisis de series de tiempo, puede consultar la bibliografía. Aquí se ha usado la base teórica del curso de series de tiempo impartido en la carrera, y casi en consecuencia, utilizamos un texto que se ha convertido en un clásico en el tema como lo es "Time Series Analysis" de Box y Jenkins, que en su tercera edición cuenta con la coautoría Reinsel.

### **1.1.1 Predicción de una Serie de Tiempo.**

¿Cuál es el objeto de hacer predicciones de una serie de tiempo? ¿En qué campos estas son requeridas? Cuan confiables pueden estas llegar a ser?

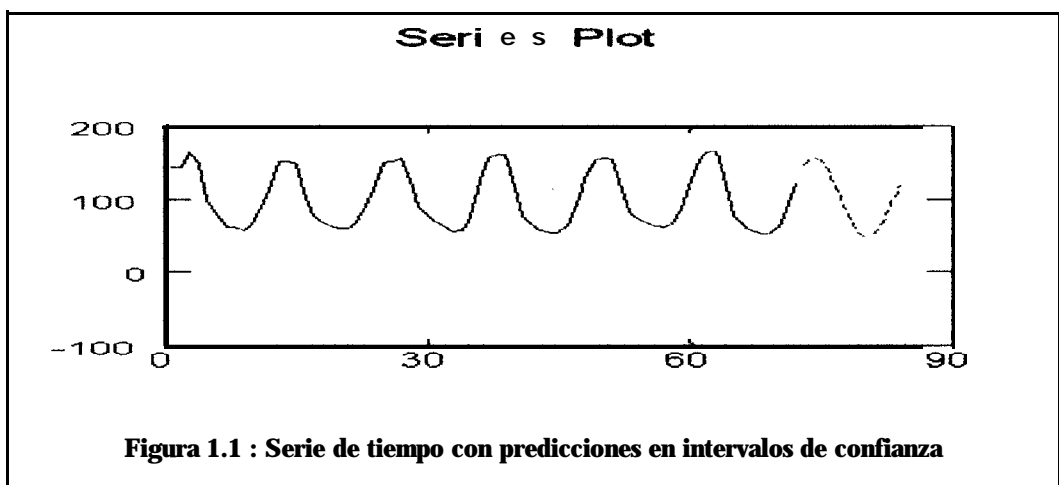
Veamos un ejemplo. Supongamos que un almacén posee datos periódicos de sus ventas. El jefe de ventas sabe que estas varían según la época del año, pero no tiene forma de saber como describir esta dependencia en los datos de su trabajo. Una de sus tareas es de prever la cantidad de productos que se van a vender para así mantener un buen inventario y sacar el mejor provecho a el trabajo de esta temporada. ¿Cómo se supone que esta persona sabrá dar una buena respuesta en base a su pura intuición? A no ser que el jefe de ventas tenga poderes paranormales, este no tendrá forma de contestar a sus superiores. El análisis de series de tiempo sirve para dar una respuesta en base a los datos anteriores que se tienen.

El uso en el tiempo  $t$  de observaciones disponibles de una serie temporal para predecir sus valor en un tiempo futuro  $t + l$  puede proveer una base para el planeamiento económico y



de negocios, planeación de producción, control de inventario, etc. Supongamos que las observaciones están disponibles en intervalos de tiempo discretos y equiespaciados; es decir, si las ventas  $x_t$  en el actual mes  $t$  y las ventas  $x_{t-1}, x_{t-2}, x_{t-3}, \dots$  de los meses previos pueden ser usadas para predecir las ventas para los tiempos posteriores  $l = 1, 2, 3, \dots, 12$  meses hacia adelante. Denotemos por  $x_t(l)$  a la predicción hecha desde el origen  $t$  de las ventas  $x_{t+l}$  en algún tiempo futuro  $t + l$ , que es, en el tiempo posterior  $l$ . La función  $x_t(l), l = 1, 2, 3, \dots$  la cual provee las predicciones desde el origen  $t$  para todos los tiempos posteriores, la llamaremos función predictora al origen  $t$ . Nuestro objetivo es obtener una función predictora que sea tal que la media cuadrática de las desviaciones  $x_{t+l} - x_t(l)$  entre los valores verdaderos y predichos sea tan pequeña como sea posible para cada tiempo futuro  $l$ .

Además de calcular las mejores predicciones, es también necesario especificar la precisión, tal que, por ejemplo, los riesgos asociados con decisiones basadas sobre las predicciones puedan ser calculados. La precisión de las predicciones pueden ser expresadas al calcular **intervalos de confianza** sobre cada una de las predicciones. Al escoger un nivel de confianza se garantiza que las predicciones **estarán** dentro del intervalo con probabilidad igual a la confianza. Veamos la figura 1.1. En esta se advierte una serie temporal la que posee una secuencia de predicciones con un nivel de confianza del 95%.



## 1.2 Procesos Estacionarios y Funciones de Autocorrelación.

“Un proceso estocástico es un fenómeno estadístico que evoluciona a través del tiempo de acorde a leyes probabilísticas” <sup>1</sup>. Sabiendo que un proceso estocástico es una secuencia de variables aleatorias que corresponden o provienen de un conjunto de eventos hacia un conjunto de valores en el espacio de números reales los que dependen del tiempo, vamos entonces a definir lo que es un proceso estocástico estacionario (en adelante obviaremos la palabra “estocástico” para referimos a procesos de esta naturaleza). Es oportuno mencionar que en la estadística nos podemos encontrar con muchos otros tipos de procesos los que son objeto de estudio en cursos enteros dedicados exclusivamente a cada uno de ellos. No todos tienen fines prácticos, pero en particular, las series de tiempo evidencian muchas aplicaciones en la vida real.

Un proceso estacionario  $(x_t) t \in T$  (donde  $T$  es un espacio de tiempos que pueden ser números enteros, cardinales, etc.) es aquel en que se asume que este se mantiene en “equilibrio estadístico”. Existen dos clases de procesos estacionarios: los estrictamente estacionarios y los débilmente estacionarios. Si asumimos que el espacio probabilístico en que el proceso se desenvuelve posee una función de distribución  $L$  entonces definimos:

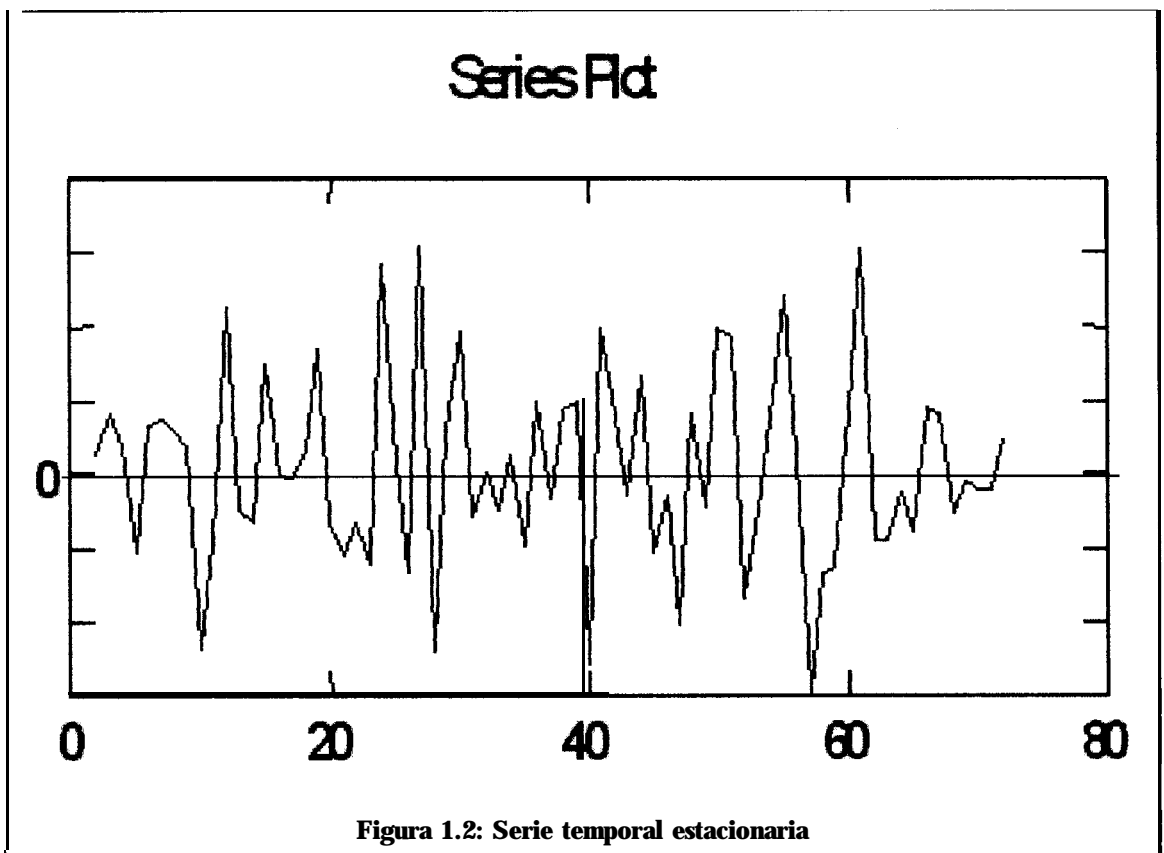
- ✓ Def. 1:  $(x_t) t \in T$  se dice estrictamente estacionario si  $L(x_{t_1}, x_{t_2}, \dots, x_{t_k}) = L(x_{t_1+h}, x_{t_2+h}, \dots, x_{t_k+h})$ , para diferentes tiempos  $t_1, t_2, \dots, t_k$  y  $h=1, 2, 3, \dots$ , en donde  $h$  es el cambio en el origen del tiempo en  $t, t_2, \dots, t_k, t_1+h, t_2+h, \dots, t_k+h \in T$ .
- ✓ Def. 2:  $(x_t) t \in T$  se dice débilmente estacionario si:
  - i.  $E(x_t^2) < \infty \forall t \in T$ .
  - ii.  $E(x_t) = m \forall t \in T$ .
  - iii.  $Cov(x_{t+h}, x_{s+h}) = Cov(x_t, x_s)$ .

---

<sup>1</sup> Definición dada por el texto de Box y Jenkins

En este caso la definición 2 se interpreta como aquel proceso que posee una **varianza** finita y una media constante, así como que la **covarianza** del proceso no depende del tiempo sino del salto que se de, es decir, la autocovarianza es independiente del origen temporal.

Por lo general será raro encontrarnos con casos de estricta estacionariedad. Para fines prácticos nos referiremos a procesos débilmente estacionarios como procesos estacionarios (obviaremos la palabra “débilmente”). Un proceso que no cumpla con alguno o con ninguno de los tres puntos de la definición 2, se dice un proceso no estacionario. Una serie de tiempo es un proceso que puede o no ser estacionario. En la figura 1.2 vemos una serie temporal que se advierte como estacionaria. Visualmente observamos que la variación de los datos es finita y que estos oscilan en una sola línea paralela a el eje temporal.



Es posible demostrar que la parte iii de la definición 2 es lo mismo que decir (si y solo si)

$$\text{Cov}(x_t, x_{t+h}) = \gamma(h)$$

La cual es la autocovariancia de orden  $h$  del proceso en cuestión. Esto quiere decir que la covarianza se transforma en una función que solo depende del salto de orden  $h$  (lag  $h$  en inglés). Si no se da ningún salto ( $h=0$ ), entonces tendríamos la varianza de  $x_t$ . Además, sabemos que:

$$\text{Cov}(x_t, x_{t+h}) = E[(x_t - \mu)(x_{t+h} - \mu)]$$

Por lo que se define la autocorrelación de orden  $h$ :

$$\rho_h = \frac{E[(x_t - \mu)(x_{t+h} - \mu)]}{\sqrt{E[(x_t - \mu)^2]E[(x_{t+h} - \mu)^2]}}$$

Y dado que la varianza es la misma sin importar el salto de orden  $h$ , entonces:

$$\rho_h = \frac{E[(x_t - \mu)(x_{t+h} - \mu)]}{\sigma_x^2}$$

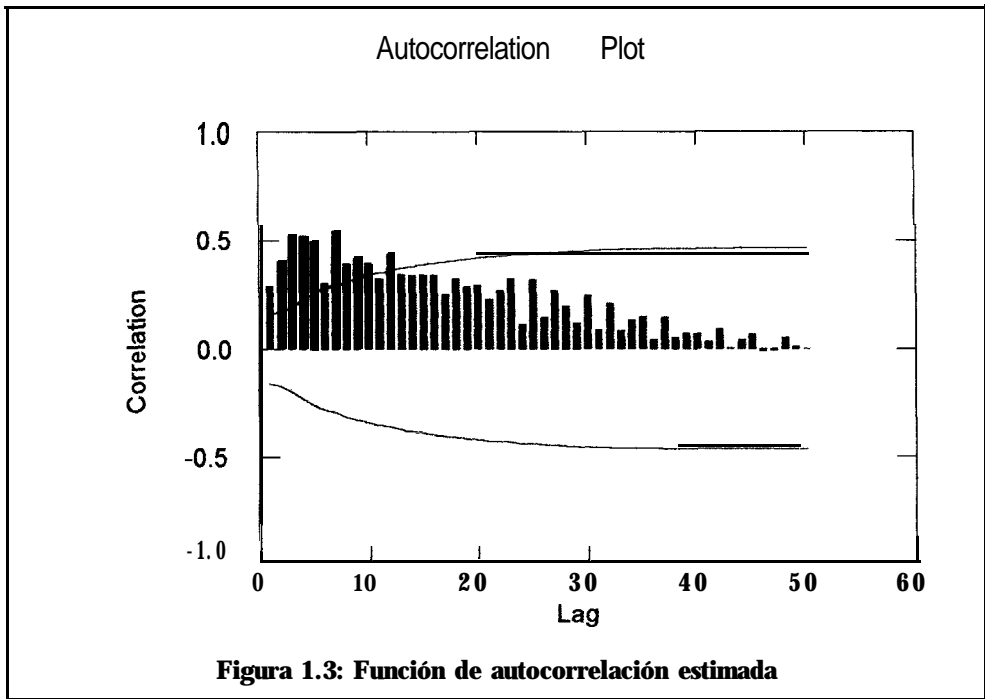
Y siendo  $\sigma_x^2 = \gamma_0$  entonces tenemos finalmente:

$$\rho_h = \frac{\gamma_h}{\gamma_0}$$

Lo que conlleva a que  $\rho_0 = 1$ .

En la práctica, se puede observar como las autocorrelaciones de una serie de tiempo tienden a cero cuando  $h$  tiende al infinito. En la figura 1.3 se puede apreciar la función de autocorrelación (estimada) graficada para diversos ordenes  $h$ . Los datos que generan esta estimación son una serie cualquiera tomada con fines de únicamente ejemplificar el caso.

Las formas de estimar esta función se pueden consultar en la bibliografía aunque es necesario recalcar que este tipo de detalles escapan al alcance de esta tesis.



**Figura 1.3: Función de autocorrelación estimada**

En la práctica, si las autocorrelaciones estimadas están tan próximas las unas con las otras se infiere que los datos son en realidad no estacionarios. Así, la figura 1.3 es también un ejemplo de un caso de no estacionariedad.

### 1.3 Modelos *Estacionarios Lineales*.

De aquí en adelante solo haremos referencias teóricas hacia definiciones necesarias para explicar en términos generales el análisis de series de tiempo, mas no profundizaremos en el tema debido a razones ya antes mencionadas.

Un modelo de serie temporal es lineal si es que se puede este proceso expresar en función de una suma de coeficientes por ruidos blancos o errores aleatorios con la suma de

coeficientes (en valor absoluto) al cuadrado finita. Esto es, dado un proceso  $(x_t)_{t \in T}$ , este se dice lineal si se puede expresar como:

$$x_t = \sum_{j=0}^{\infty} \psi_j u_{t-j} \quad \text{Dado que} \quad \sum_{j=0}^{\infty} |\psi_j|^2 < \infty$$

El subíndice de el ruido blanco  $u$ , indica el retardo en el tiempo en el que se incurre al expresar el valor actual de  $x$  en el tiempo  $t$ . Los coeficientes  $\psi_j$  son números condicionados a ser absolutamente sumables para todos sus valores existentes.

Existen 3 clases de procesos estacionarios lineales que son de trascendental importancia en el análisis de series de tiempo:

- ✓ Modelos Autorregresivos
- ✓ Modelos Media Móvil
- ✓ Modelos Autorregresivos - Media Móvil

En el primer caso, un modelo autorregresivo  $AR(p)$  es aquel proceso lineal en que  $x_t$  se puede expresar mediante las observaciones  $x_{t-h}$  de tiempos anteriores  $t-h$  mas un término de error o ruido blanco en el instante  $t$ . En cambio, un modelo media móvil  $MA(q)$  es aquel en que  $x_t$  es expresado mediante una combinación lineal de ruidos blancos  $u_{t-h}$  retardados en tiempos  $t-h$ . El tercer modelo es el caso en que  $x_t$  es expresado mediante una parte autorregresiva más una parte media móvil. Este modelo mixto  $ARMA(p, q)$  es el más general de los procesos lineales estacionarios. Entonces en general tendremos:

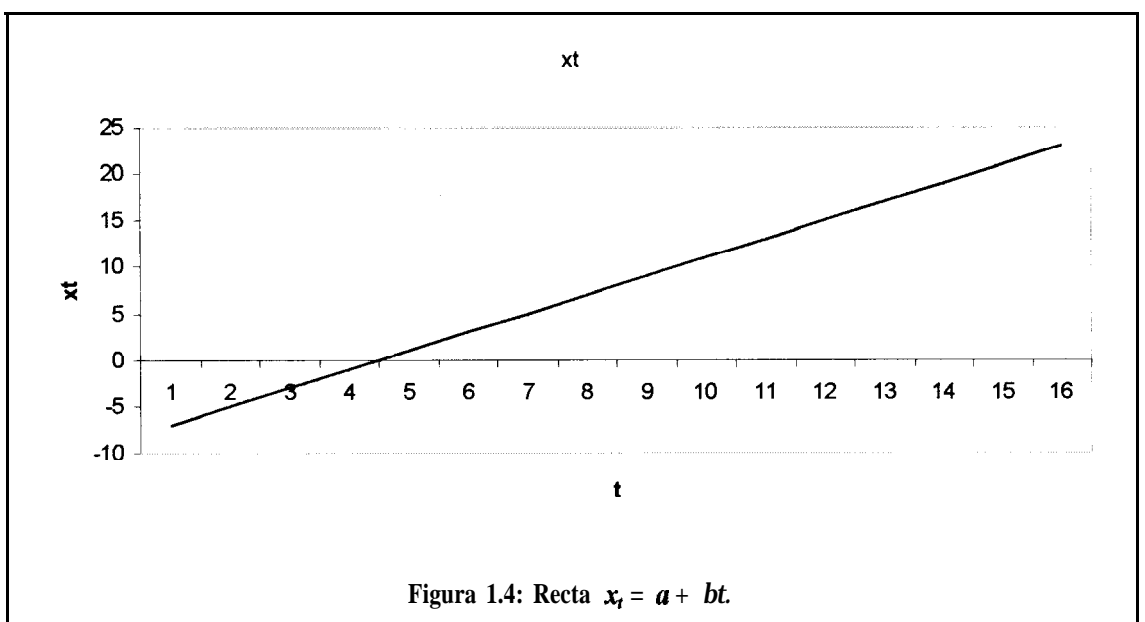
$$x_t = \phi_1 x_{t-1} + \phi_2 x_{t-2} + \dots + \phi_p x_{t-p} + u_t - \theta_1 u_{t-1} - \theta_2 u_{t-2} - \dots - \theta_q u_{t-q}$$

En donde  $p$  y  $q$  son el número de parámetros y retardos de cada parte. Existen diversas y complejas formas (que son en realidad necesarias para el análisis) de expresar un  $ARMA(p, q)$  pero no es de vital importancia el describirlas en este caso.

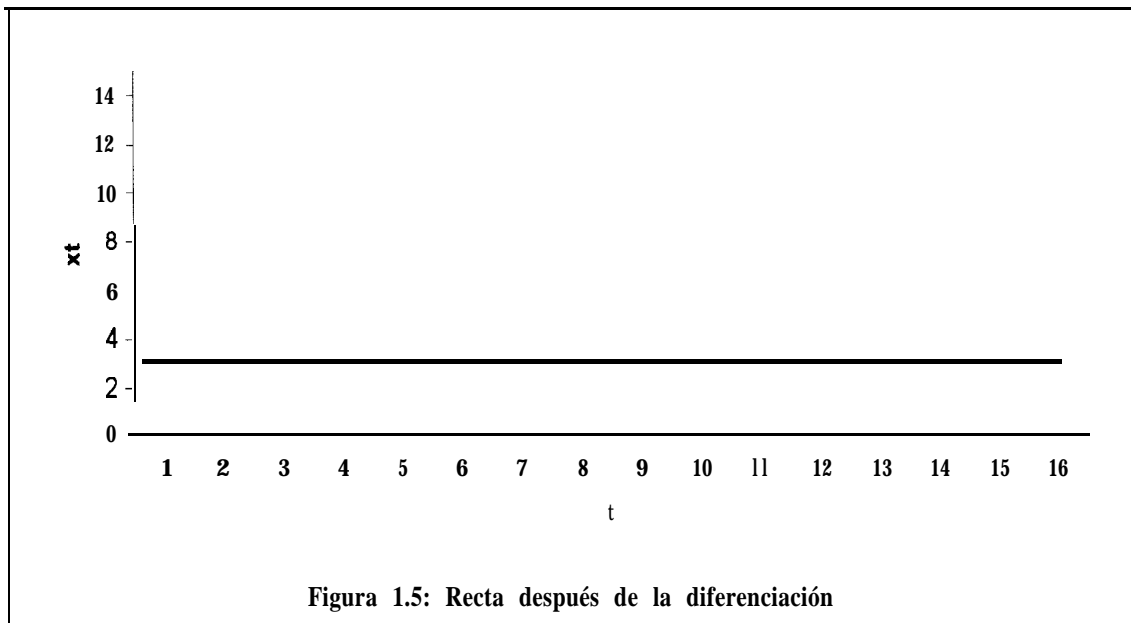
#### 1.4 Procesos No *Estacionarios* Lineales.

Un proceso no estacionario lineal es aquel en que no se cumple que la serie tenga una media fija, que la *varianza* tampoco lo sea, o que ninguna de las dos se cumpla. Por definición, no podemos modelar una serie no estacionaria y por tanto lo que se hace es modificar la serie para que se convierta en estacionaria. A esta modificación se la denomina *diferenciación*, la que consiste en aplicar (multiplicar) polinomios de retardo al modelo en la parte autorregresiva pero de manera pura (sin la multiplicación por algún parámetro).

Un modelo  $ARIMA(p, d, q)$  (Autoregressive Integrated Moving Average en inglés) es el más general de los modelos no estacionarios en los que pueden existir combinaciones entre medias móviles puras con diferenciación o autor-regresivos diferenciados. El parámetro  $d$  indica el de retardo en el tiempo aplicado sobre el modelo ARMA original. En otras palabras, si  $d=1$ , entonces la nueva variable  $y_t = x_t - x_{t-1}$  es un proceso nuevo diferenciado 1 vez; si  $d=2$ ,  $w_t = y_t - y_{t-1}$  es un proceso que ha sido diferenciado 2 veces con respecto a  $x_t$ . Por lo general, los datos se hacen estacionarios con una sola diferenciación, pero no siempre es el caso.



La idea tras la diferenciación se esclarece tras el siguiente ejemplo: supongamos que  $(x_t)$   $t \in T$  es un proceso no estacionario tal que  $x_t = a + bt$ . Entonces, como observamos en la figura 1.4, el proceso tiene una pendiente  $b$  (lo cual significa no estacionariedad ya que la media no es constante) y una elevación natural  $a$ .



Definamos entonces  $y_t$  tal que  $y_t = x_t - x_{t-1}$ , por lo tanto  $y_t = (a + bt) - [a + b(t-1)] = b$ .

Esto nos lleva a la conclusión de que  $y_t$  se vuelve estacionario porque  $b$  es una constante y el valor esperado de una constante es también una constante. Este ejemplo no justifica el método pero lo ilustra. Veamos la figura 1.5. Observemos como quedan los datos a partir de la diferenciación.

A partir de la diferenciación se puede proceder a hacer las estimaciones del caso ya que el nuevo modelo es estacionario; sin embargo vale la pena mencionar que bajo propiedades que se pueden consultar en la bibliografía, un ARMA no es invertible.



### **1.5 Identificación, Estimación, Chequeo y Diagnóstico del Modelo.**

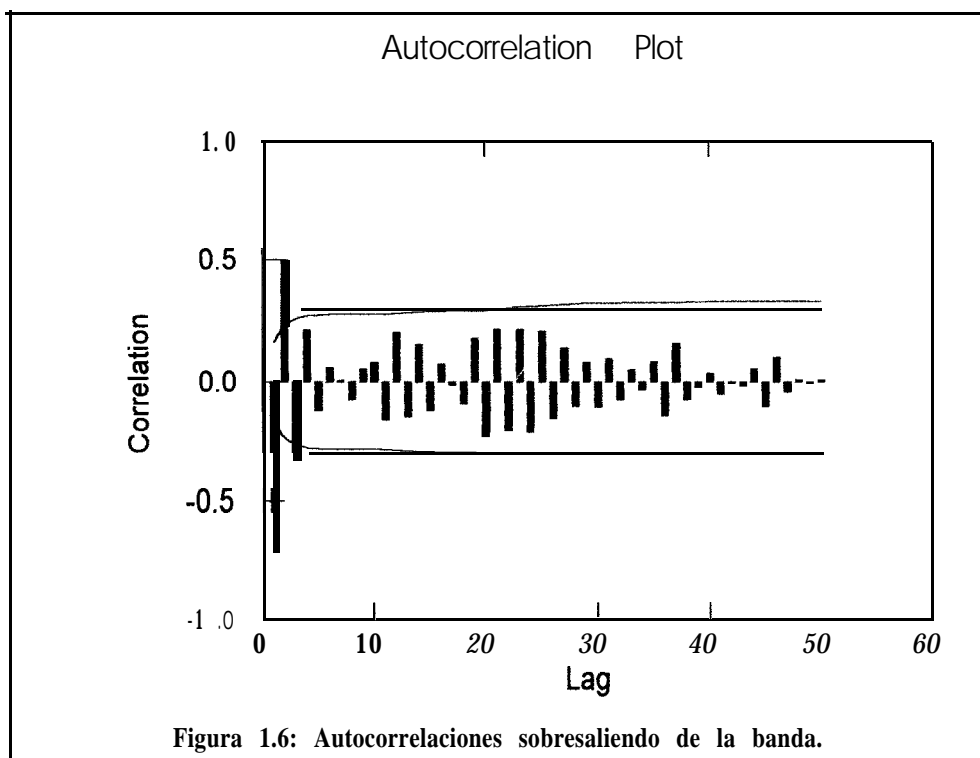
En esta sección nos vamos a enfocar únicamente en el sentido práctico de la modelización de datos de series temporales. Para esto es necesario saber que existen formas de estimar las autocorrelaciones así como la **varianza** de las mismas, lo que por lógica nos dice que podemos construir intervalos de confianza para estas. Gracias a que se pueden estimar las autocorrelaciones también se puede, a partir de estas, estimar los parámetros que uno supone posee el modelo. Modelar una serie de tiempo no es como modelar en regresión lineal. En regresión lineal el propio método de mínimos cuadrados arroja un modelo a partir de los datos de que se disponga. En el análisis de series de tiempo la persona propone un modelo el que se identifica a partir de la observación de las autocorrelaciones y sus intervalos de confianza. Una vez identificado el modelo, se procede a hacer las estimaciones (las que se explican en detalle en la **bibliografía**); luego se debe verificar si este modelo se ajusta bajo pruebas de hipótesis y se da un diagnóstico. Si el modelo ajusta y predice correctamente, entonces hemos llegado a lo que necesitábamos, de lo contrario debemos repetir el procedimiento desde el inicio al proponer un modelo distinto.

El primer criterio que tenemos que tomar en cuenta para la identificación del modelo es que existen 2 tipos de correlaciones con las que se trata en este análisis: las autocorrelaciones y las autocorrelaciones parciales. Las autocorrelaciones (AC) sirven para identificar si el modelo es o no estacionario y además para detectar cuantos parámetros tiene la parte media móvil, es decir nos da el orden  $q$  inicial de la parte MA que debemos de sugerir. Las autocorrelaciones parciales (ACP) sirven para detectar un valor inicial de el número de parámetros que posee la parte autoregresiva del modelo, es decir las ACP sugieren el orden  $p$  inicial de la parte AR. El parámetro  $d$  se lo obtiene contando el número de veces que se han diferenciado los datos



iniciales. En general  $p + d + q \leq 5$ , por lo que se deben probar algunos modelos desde el primer vistazo hacia las correlaciones ya que en general nos encontraremos con valores grandes  $d$  y  $q$  que harán que esto no se cumpla.

Para objeto de identificación de los valores iniciales de  $p$  y  $q$ , las autocorrelaciones y las autocorrelaciones parciales se deben de **graficar** junto con sus intervalos de confianza así como se observa en la figura 1.1. Si las AC y las ACP son estadísticamente nulas, entonces quedaran por dentro de las bandas, de lo contrario serán significativas. El orden es determinado por el valor de  $h$  correspondiente a el último valor que sobresalga fuera de las bandas. Los órdenes de los modelos se van aumentando o reduciendo de 1 en uno y de un solo lado a la vez (sea de  $p$  en AR o de  $q$  en MA). Nunca se debe de modificar el modelo cambiando los órdenes al mismo tiempo. Veamos la figura 1.6. Aquí se observa como sobresalen de la banda las autocorrelaciones de orden 1, 2 y 3.



Veamos ahora la figura 1.7. Podemos observar como sobresale la ACP de orden 1 de manera notable.

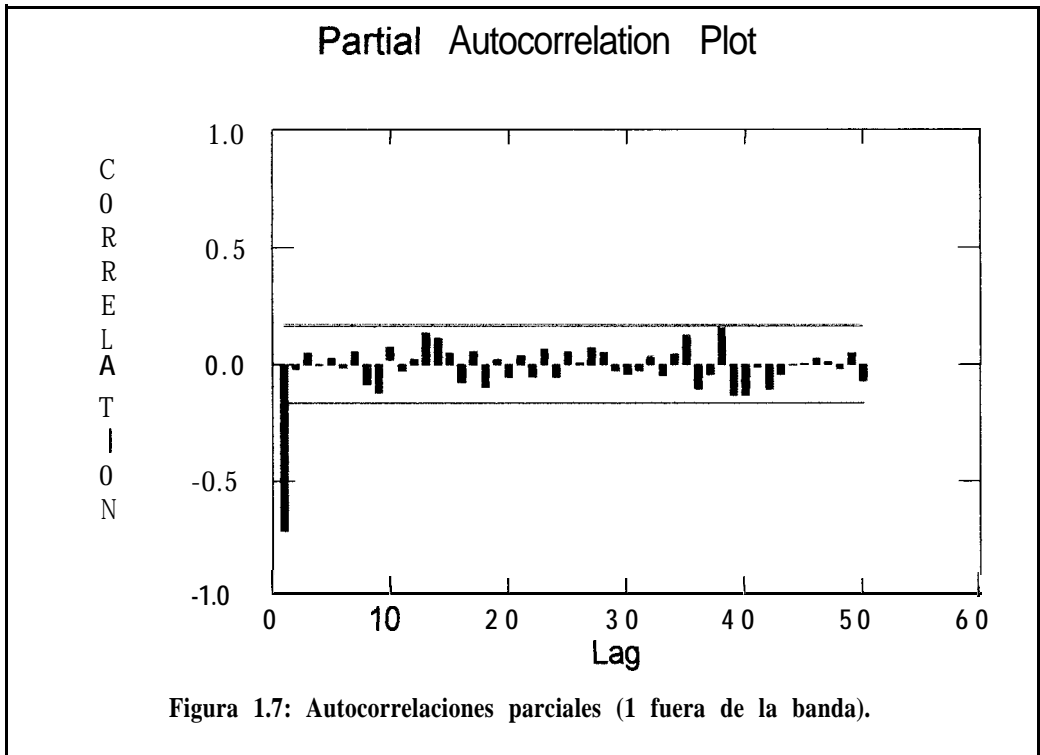


Figura 1.7: Autocorrelaciones parciales (1 fuera de la banda).

Podemos entonces sugerir un modelo inicial ARMA(1,3) para que sea estimado mediante métodos numéricos que no vamos a explicar ya que en la actualidad los paquetes estadísticos se encargan de generar todas estas estimaciones. Ahora veamos el siguiente cuadro proveniente de el modelo sugerido hacia una salida en un programa estadístico:

		Final value of MSE is 1.038			
Index	Type	Estimate	A. S.E.	Lower <95%>	Upper
1	AR	-0.655	0.182	-1.013	<b>-0.296</b>
2	MA	0.078	<b>0.199</b>	-0.315	0.470
3	MA	-0.073	0.154	-0.377	0.231
4	MA	0.014	0.118	-0.218	0.247

**Cuadro 1.1: Corrida de un modelo ARMA en Systat 7.0**

En este cuadro podemos darnos cuenta que al sugerir un modelo ARMA(1, 3) los 3 parámetros de la parte media móvil son estadísticamente nulos, es decir que los parámetros se encuentran dentro de un intervalo de confianza del 95% en donde el cero está incluido. Por lo tanto el siguiente modelo que se debe probar es un AR(1), ya que el sentido común nos lo dice. Veamos el siguiente cuadro. Aquí vemos que efectivamente el parámetro  $\phi_{1,1}$  (estimado) tiene un valor de -0.722 y una media cuadrática del error muy chica ( $MSE = 1.020$ ).

<b>Final value of MSE is</b>		1.020			
<b>Index</b>	<b>Type</b>	<b>Estimate</b>	<b>AS.E</b>	<b>Lower &lt;95%&gt;</b>	<b>Upper</b>
1	AR	-0.722	0.057	-0.835	-0.609
<b>Cuadro 1.2: Corrida de un modelo AR en Systat 7.0</b>					

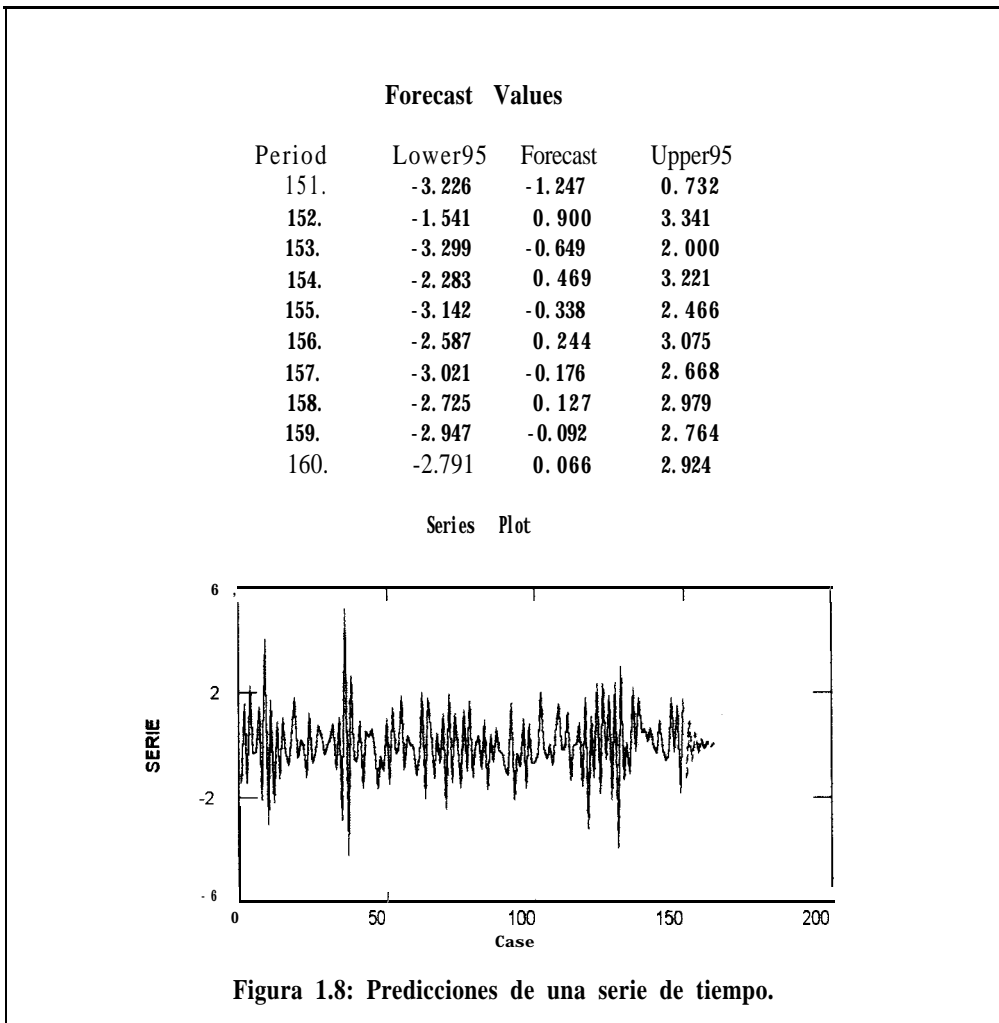
Lo curioso del caso es que los datos que se usaron para este ejemplo fueron en realidad simulados de un proceso AR(1) el que tenía un  $\phi_{1,1} = -0.8$ , el que se aproxima bastante a nuestro resultado de la estimación. Además podemos ver que el parámetro no se anula.

En la figura 1.8 se muestran unas predicciones en base al modelo estimado así como una tabla con estos valores. El modelo estimado definitivo es el siguiente:

$$\hat{x}_t = -0.722x_{t-1} + u_t$$

Vérsus el modelo que originalmente simulamos:

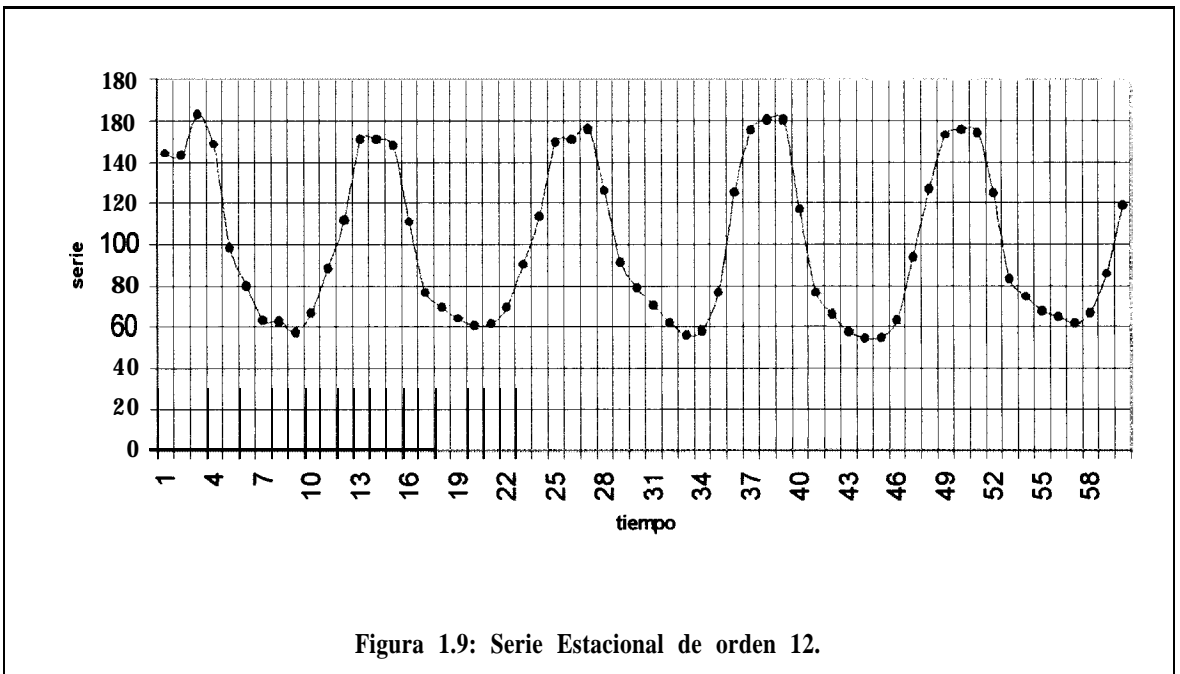
$$x_t = -0.8x_{t-1} + u_t$$



Existen también criterios en base a las predicciones y los errores generados en estas, así como pruebas de hipótesis para chequear la idoneidad del modelo, pero no las presentaremos en esta tesis ya que nuestra intención es dar una idea general de este tipo de análisis. En todo caso, a estas alturas ya estamos casi terminando de comprender la idea general tras el análisis de series de tiempo. En la siguiente sección veremos un tipo de serie temporal de vital importancia como para complementar nuestro conocimiento general de la naturaleza de los datos.

## 1.6 Modelos Estacionales.

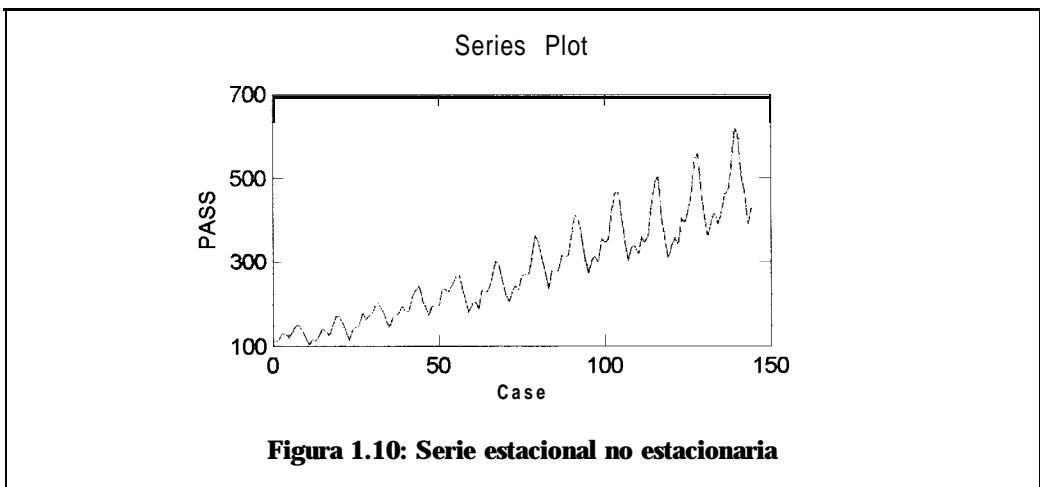
Existe un tipo de datos marcadamente importantes entre las series de tiempo. Son los de tipo estacional, que significa que la variable se encuentra correlacionada con un salto periódico de si misma sobre un horizonte de tiempo equiespaciado. En otras palabras, “se dice que una serie exhibe un comportamiento periódico con periodo  $s$ , cuando ocurren similitudes en la serie cada  $s$  intervalos básicos de tiempo”<sup>2</sup>. Para terminar de comprender la idea basta con observar la serie de la figura 1.9. Aquí se observa una serie estacional cada 12 meses (que es el caso que con mayor frecuencia ocurre). Podemos darnos cuenta que los máximos y mínimos ocurren de manera equiespaciada.



Se da el caso de que los datos, además de estacionalidad, tengan una pendiente que haga que los mínimos y máximos se repitan de forma creciente o decreciente. Por lo tanto lo que se debe hacer primero es una diferenciación por no estacionaridad, y luego observar las nuevas

<sup>2</sup> Definición dada por el texto de Box y Jenkins

autocorrelaciones para ver si se necesita diferenciar estacionalmente. La diferenciación estacional consiste de obtener un nuevo proceso  $\tilde{x}_t = x_t \cdot x_{t-s}$ , que genera un salto de orden  $s$ . Por lo tanto podemos inferir que la idea es la misma que en el caso de la diferenciación por no estacionaridad pero con la intención de dar saltos mayores que **1** según el comportamiento estacional de la serie. La figura 1.10 muestra una serie que eventualmente necesitará ambos tipos de diferenciación.



**Figura 1.10: Serie estacional no estacionaria**

El modelo más general que abarca todos los anteriores se denomina Seasonal Autoregressive Integrated Moving Average (multiplicativo) o **SARIMA( $p, d, q$ ) x ( $P, D, Q$ ) $_s$** , en donde  $p, d$  y  $q$ , son como ya sabemos y  $P, D$  y  $Q$  representan el orden que posee las partes estacionales AR, por diferenciación estacional, y MA estacional respectivamente.

Se escapan muchos otros detalles sobre estacionalidad y su respectiva modelización, estimación, chequeo y diagnóstico, pero para objetivos comparativos, de criterio práctico, y de acuerdo con la intención de esta tesis (que evidentemente se orienta a las redes neuronales), son de segunda y tercera importancia.

### **1.7 Observaciones.**

Una vez conscientes de lo que está implicado en el análisis de series de tiempo, ya tenemos elementos de juicio para ser combinados con las herramientas que se presentan en el siguiente capítulo y así poder distinguir bien cada problema. Adelantándonos un poco en materia, advertimos que es necesario conocer la naturaleza de los datos para así poder construir un modelo de redes neuronales que se ajuste al problema sin caer en muchas generalidades que, como es de conocimiento común, restan precisión y añaden errores a la solución de los problemas.

Al final del siguiente capítulo se utilizarán los criterios de este para construir una arquitectura adecuada de red neuronal.



## CAPITULO II

### 2 Sistemas Neuronales Artificiales (ANS).

#### 2.1 *Introducción.*

¿Qué mejor modelo para una máquina inteligente que la propia mente del hombre? La inteligencia humana se basa en el funcionamiento de una interconexión masivamente paralela de células con características de procesamiento de estímulos electro-químicos sustancialmente distintas al resto de células del cuerpo humano. Es la estructura de esta interconexión y dicho tipo de estímulos lo que hace posible que una persona tenga la habilidad de reconocer patrones tales como letras y números de una forma instantánea. ¿Y por qué es que a un computador le tomaría mucho más tiempo el hacer esta “simple tarea” de manera algorítmica? La respuesta es que la tendencia en el ámbito computacional siempre fue la de seguir pasos uno a uno, y si se quería resolver este tipo de problemas se tenía que hacer búsquedas gigantescas cuya

convergencia no se garantizaba con computadores tan lentos y espaciosos. Es decir, que el funcionamiento en paralelo se convertía en prohibitivo debido a las limitaciones espacio-tiempo de las que en la última década se ha logrado superar. Por eso en la actualidad el uso de las redes neuronales en una amplia gama de situaciones se ha acentuado.

Pero, ¿qué es una red neuronal? Una red neuronal es aquella simulación computacional de el funcionamiento del cerebro y su estructura en base a la conexión de elementos que se encuentran en un espacio lógico. A estos elementos se los denomina neuronas, tales como las células del cerebro. Las redes neuronales eliminan el problema de búsqueda ya que estas no almacenan información: los sistemas neuronales artificiales (ANS) aprenden de la información, por lo tanto no hay búsqueda sino más bien hay una respuesta en correspondencia a una entrada.

Para resolver nuestro problema de **análisis** de series de tiempo vamos a usar ANS (siglas genéricas en inglés: Artificial Neural Systems) que aprenderán a predecir (que es en realidad lo que el análisis de series de tiempo quiere lograr al arrojar un modelo matemático) eventos en el futuro que se corresponden a valores de variables aleatorias con rangos de números reales. El uso de redes neuronales en este **ámbito**, presenta una alternativa hacia el análisis de series de tiempo discutido en el capítulo anterior.

En este capítulo nos introduciremos hacia lo que son las redes neuronales haciendo analogías y diferencias con respecto a la estructura de la mente humana para así poder entender las ventajas y limitaciones que tenemos a la mano. Luego de explicar brevemente la estructura neuronal del cerebro, haremos la introducción hacia una red neuronal de forma general. En este modelo ANS general definiremos los componentes principales que integran una red. Lo siguiente será presentar la neurona artificial creada por Frank Rosenblatt (uno de los mayores investigadores en el campo neuronal artificial que creó el perceptrón) y las

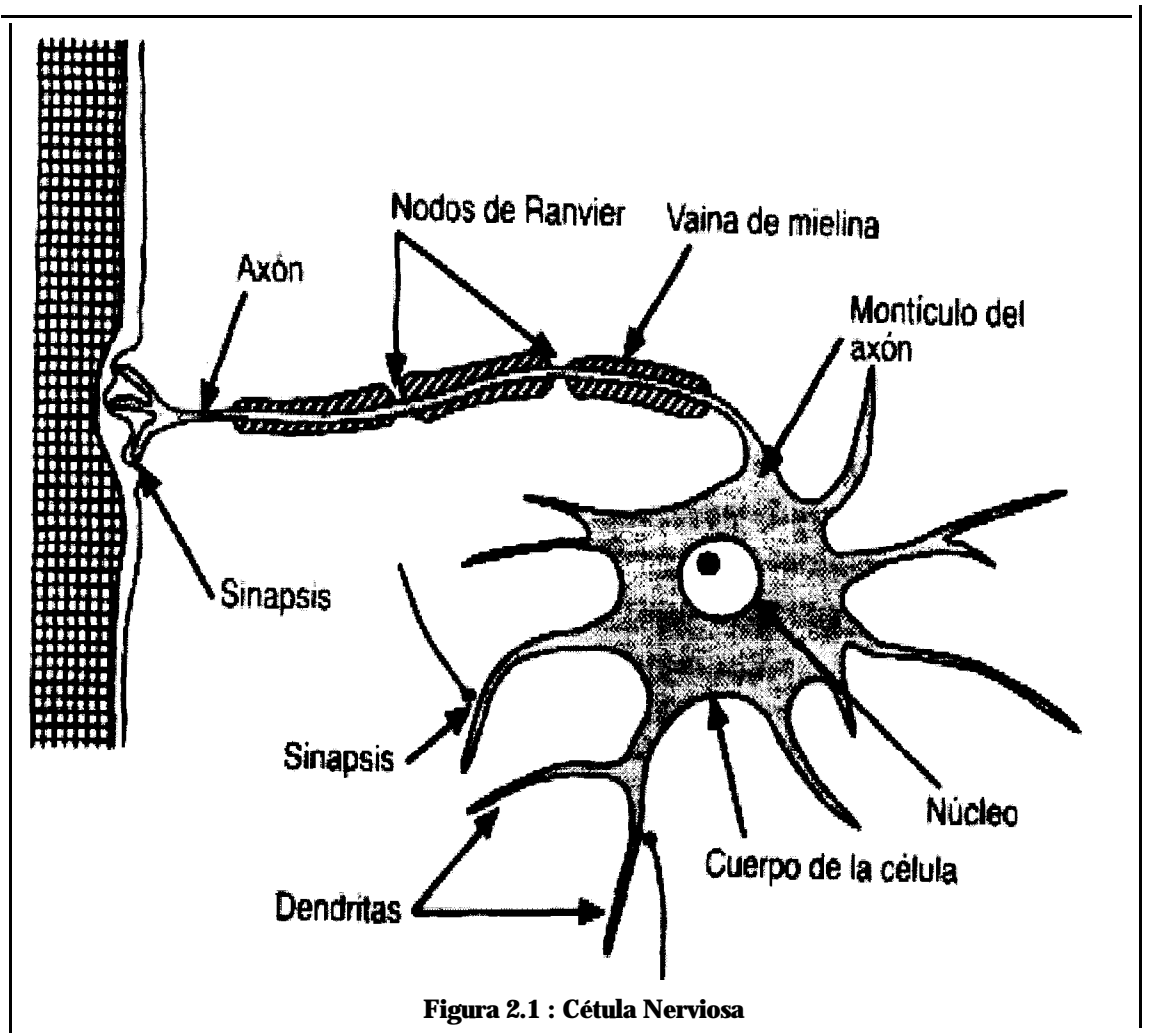
desventajas de usarla al exponer el problema XOR. Tendremos también que presentar un modelo multicapa llamado Red de Propagación hacia Atrás (BPN) conociendo su ley de aprendizaje paso a paso. Por último, presentaremos el modelo BPN que resuelve nuestro problema de predicción.

## **2.2 Descripción desde el punto de vista biológico.**

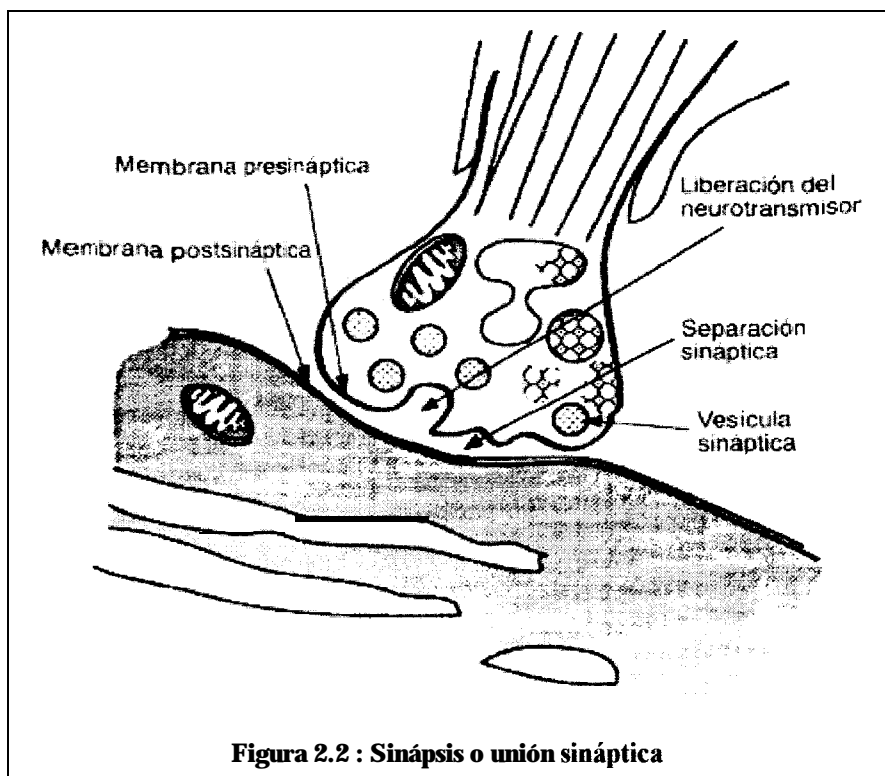
Las células nerviosas del cerebro se llaman neuronas. Esta es la unidad fundamental del que dependen los procesos en el cerebro. Una neurona consta (Fig. 2.1) de un cuerpo celular con su núcleo que es donde se procesan los impulsos electroquímicos provenientes de otras neuronas. La neurona posee dendritas, que son receptores de los impulsos (vías de entrada). El axón de la neurona es la parte que arroja impulsos hacia otras neuronas (vía de salida) .

El axón es único en cada neurona. El axón de muchas neuronas está rodeado por una membrana que se denomina vaina de mielina. Los nodos de Ranvier interrumpen periódicamente la vaina de mielina a lo largo del axón . El axón, al final, se divide y se conecta con dendritas de otras células nerviosas mediante una unión llamada sinapsis. La transmisión a través de esta unión tiene naturaleza química y la cantidad de señal transmitida (impulso) depende de la cantidad de químicos liberados por el axón y receptados por las dendritas. Los químicos que sirven como objeto de transmisión se llaman neurotransmisores. Estos neurotransmisores están almacenados en vesículas (que son una especie de contenedores) próximas a la membrana presináptica. Estas sustancias se liberan en la separación sináptica y se difunden hacia la membrana postsináptica, en las que son absorbidas posteriormente.

Al hablar de membranas presinápticas y postsinápticas nos referimos al **final** del axón de una neurona y el comienzo de una dendrita de otra neurona respectivamente. Esta unión sináptica posee una eficiencia que es lo que se modifica cuando el cerebro aprende.



La figura 2.2 muestra la forma que tiene esta unión entre dos neuronas. Obsérvese que la eficiencia a la que nos referimos es la permeabilidad existente para enviar neurotransmisores de una neurona a otra. La actividad generada en un impulso puede ser de naturaleza excitatoria o inhibitoria. Esto dependerá de la carga iónica que posean los neurotransmisores, pero la excitación o inhibición de la neurona dependen de la permeabilidad de la sinapsis para dejar pasar la carga iónica. Como nos podemos dar cuenta la sinapsis no es un ente por sí solo. La sinapsis es el punto de encuentro o bien de separación entre dos neuronas.



McCulloch y Pitts (1943) fueron los primeros investigadores en “tratar al cerebro como un organismo computacional” y proporcionaron una teoría para explicar como se combinaban los aspectos biológicos que hemos mencionado para dar al cerebro la inmensa capacidad que posee. Una de las suposiciones que hacía la teoría de McCulloch-Pitts era que las neuronas se

comportaban de forma binaria, su actividad estaba presente o no lo estaba del todo. Esto permitió que se pudiera usar lógica de predicados para describir la actividad de las redes. Así, se pudo describir de forma computacional la actividad cerebral aunque esta teoría no lo diga todo ni se haya comprobado su total veracidad. Pero lo importante es que se pudo entender la manera en que el cerebro funcionaba desde un punto de vista matemático y por ende simbólico. La abstracción hecha por esta teoría dio paso a otros investigadores para que pudieran modelar redes con distintos propósitos.

La forma en que aprendemos debe ser el otro aspecto de importancia que nos concierne para poder entender el comportamiento neuronal. Las personas no nacemos sabiendo y debemos de tener alguna manera de aprender. En 1949 en el libro *Organization and Behaviour*, Hebb plantea una ley de aprendizaje del sistema neuronal. La principal suposición dice textualmente:

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

Los cambios de origen metabólico a los cuales hace referencia Hebb eran, según el, el aumento en el área sináptica. Luego se supo que la eficiencia dependía de la permeabilidad de la sinapsis (es decir de la cantidad de neurotransmisores que se dejaba pasar) de la que discutimos anteriormente en esta sección.

Para simplificar, Hebb planteaba una ley de aprendizaje que decía que el sistema neuronal aprendía por familiarización de la actividad neuronal debido a constantes estímulos.

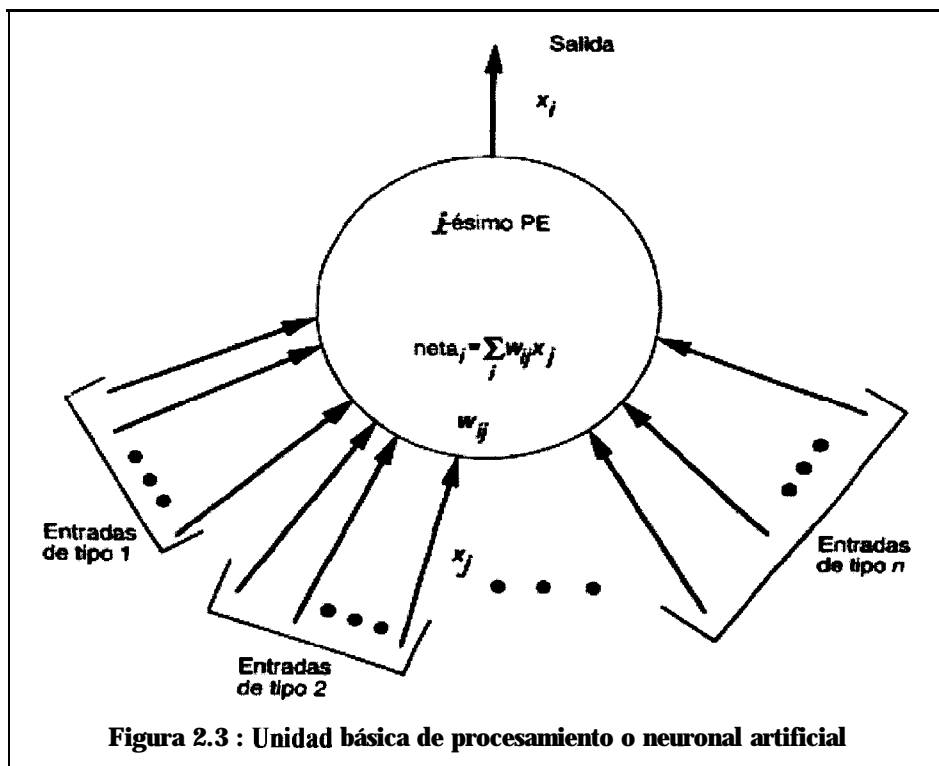
### 2.3 Descripción del los ANS.

En una red neuronal artificial, la unidad análoga a la neurona biológica se denomina elemento de procesamiento (PE: Processing Element), en lugar de llamarle neurona artificial, pero es indistinto llamarlas unidades o nodos de igual manera. Un nodo tiene cierto número de entradas (que vendrían a ser las dendritas), estas entradas son numéricas en naturaleza y son combinadas usualmente por una suma ponderada. Las ponderaciones que esta suma posee son análogas a la permeabilidad sináptica de las células nerviosas ya que de la modificación de estos pesos depende la “inteligencia” de la red. La combinación de la entrada es luego modificada por una función de transferencia. Esta función de transferencia puede ser de tipo umbral: es decir que produce el paso de información sólo si la combinación de la actividad pasa cierto nivel; pero también puede ser una función continua de la combinación de las entradas. La respuesta de la función de transferencia sirve como la salida para el PE, que a su vez sirve de entrada para la o las siguientes unidades.

Veamos la figura 2.3. Todos los nodos están numerados y en la figura aparece el  $i$ -ésimo nodo. Las entradas que provienen del  $j$ -ésimo nodo hacia el  $i$ -ésimo nodo se las puede escribir vectorialmente como  $X = (x_0, x_1, \dots, x_n)^T$ . El peso de la conexión procedente del  $j$ -ésimo nodo y que llega al  $i$ -ésimo nodo se denota por  $w_j$ . Todo nodo calcula un valor neto entre todas las entradas al sumar ponderadamente las entradas  $x_j$  por los pesos de conexión  $w_j$ . Escribimos entonces:

$$neta_i = \sum x_j w_{ij}$$

para todas las  $j$  conexiones que llegan al nodo  $i$ . Los pesos pueden tener signos positivos o negativos dependiendo de la necesidad de excitación o inhibición de la neurona.



Lo siguiente es transformar la entrada *neta*, mediante la función de salida o transferencia  $f$ . Cada nodo tiene su función de transferencia, y la respuesta que esta genera es única por cada nodo. Es decir que no pueden haber dos salidas distintas para un mismo PE, pero la salida puede ser transferida hacia cuantos nodos se desee. Esto es análogo con la división del axón, que lleva un mismo impulso que es repetido hacia diferentes dendritas de otras neuronas. Entonces tenemos la nueva salida del nodo  $i$ :

$$x_i = f_i(neta_i)$$



Una red neuronal consiste de muchos elementos procesadores unidos de la manera mostrada. La manera usual de organizar las unidades se hace mediante capas. Una red típica consiste de una secuencia de capas con conexiones totales hacia capas subsiguientes. Por lo general existen dos capas que interactúan con el medio externo: La capa de entrada en donde se le presentan los datos a la red, y la capa de salida la cual provee de una respuesta total de la red determinada para cierta entrada. Las capas que se encuentran en el intermedio de estas se denominan capas ocultas. La figura 2.4 muestra esta red típica con una capa oculta.

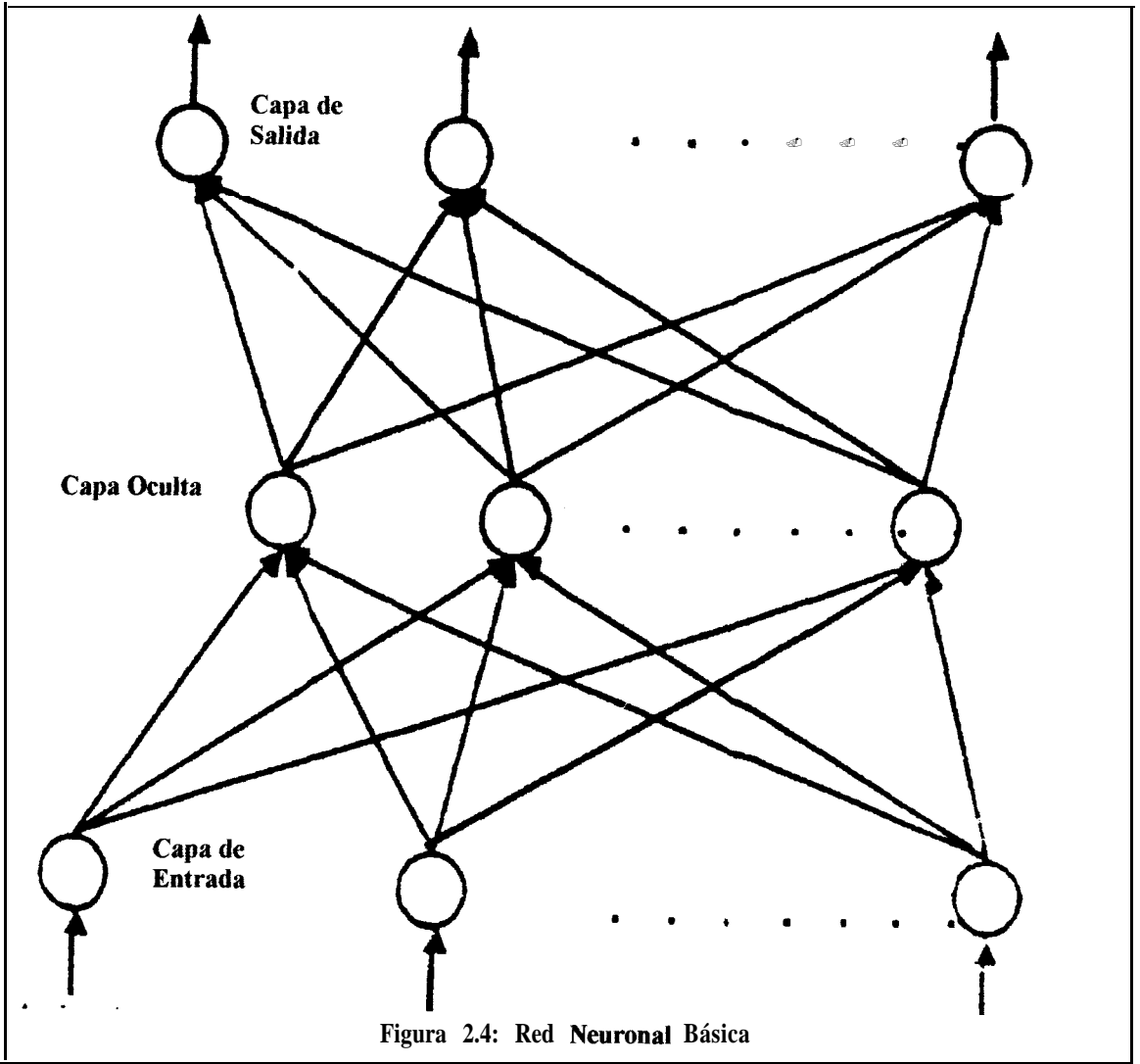


Figura 2.4: Red Neuronal Básica

Las redes neuronales se distinguen en base a su estructura y su ley de aprendizaje.

**Aprendizaje** es el proceso de adaptación o modificación de los pesos de conexión en respuesta a estímulos presentados en la capa de entrada y, si se quiere, de la capa de salida. La regla de aprendizaje especifica cómo las ponderaciones se adaptan en respuesta hacia un ejemplo de entrenamiento. El entrenamiento de una red podría tomar muchos ejemplos así como tan solo uno. Una vez entrenada la red se desea que esta genere respuestas para distintos propósitos de aplicación (dependiendo del modelo de red empleado). Se define como **recordara** la forma en que la red procesa estímulos presentados en la capa de entrada y crea una respuesta en la capa de salida. Aunque el **recordar** es una parte integral del aprendizaje, la red también recuerda cuando no se está entrenando si no cuando se la está utilizando en una aplicación.

Aunque hemos hecho muchas analogías con respecto al sistema nervioso, es necesario advertir que estas suposiciones no deben de ser tomadas de forma literal cada vez que advirtamos un modelo de redes neuronales artificiales. Podría darse el caso que nos llevemos decepciones por creer que las redes neuronales poseen las mismas restricciones del sistema nervioso. Hay que ser claro al decir que los ANS son gigantescamente más limitados que una red neuronal biológica como lo mencionan Freeman y Skapura (ver bibliografía). Lo que sucede con las analogías es que nos sirven para entender la intención que tienen las redes neuronales de poder ser buenas fuentes de inteligencia artificial.

## 2.4 El Perceptrón.

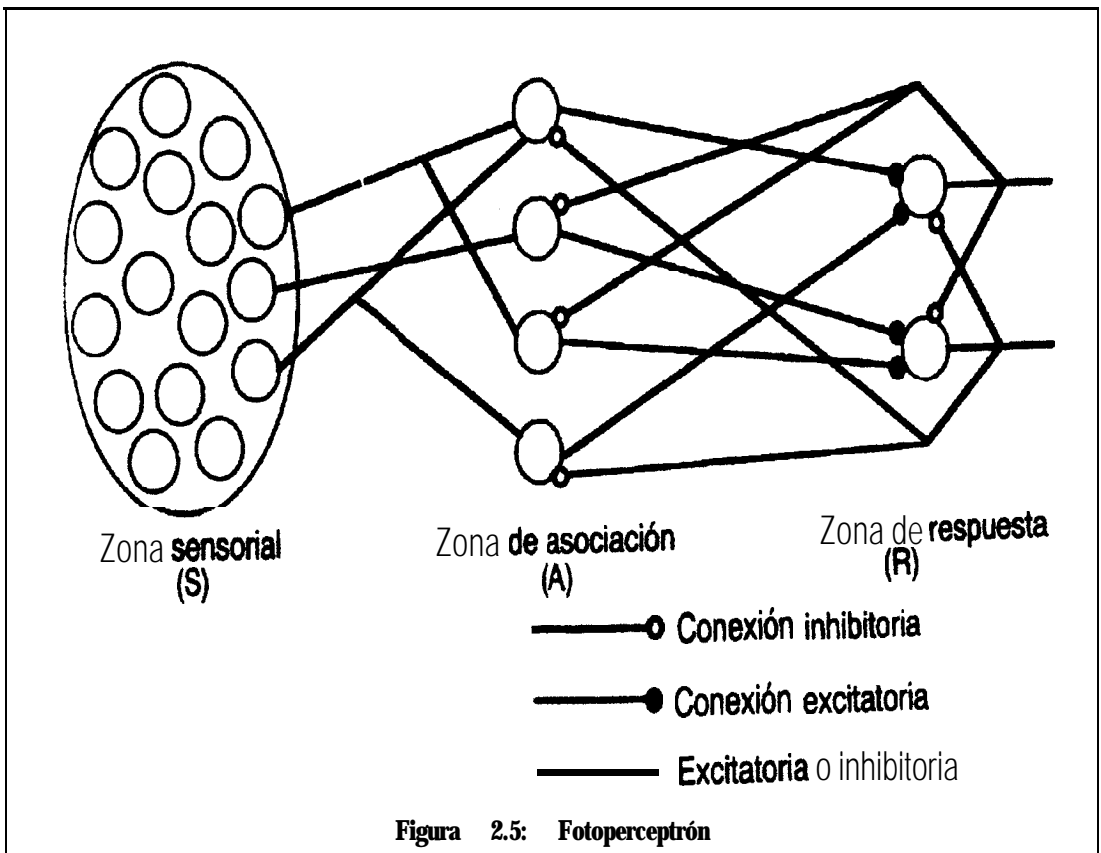
En esta sección vamos a discutir acerca de uno de los primeros modelos de neuronas que surgieron a mediados de siglo. Era de mucho interés la investigación sobre las máquinas inteligentes hasta que limitaciones de tipo computacional y de no convergencia hizo desaparecer, o más bien interrumpir, casi por completo y por varias décadas la investigación de modelos conexionistas.

### 2.4.1 Modelo de Frank Rosenblatt.

En 1958 fue editado un libro llamado *“The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain”* escrito por Frank Rosenblatt, en el que se presentaba un modelo neuronal basado en la aleatoriedad y que prometía ser revolucionario debido a su capacidad de aprender a calcular y a distinguir y clasificar patrones suficientemente distintos. Rosenblatt le llamó *Perceptrón* a su unidad de procesamiento básica, los que son de interés para nosotros por su trascendencia histórica en el campo de la investigación neuro-computacional. El perceptrón fue diseñado específicamente para que este pudiera reconocer patrones visuales y luego los clasificara en distintos grupos (aunque para la época esto suene un tanto ambicioso).

De la manera en que Rosenblatt concibió el perceptrón, tenemos un dispositivo que percibe señales visuales y arroja una clasificación de esta. Nace así el *fotoperceptrón*, que es una red que consta de 3 capas. Veamos la figura 2.5; en esta se advierten tres capas, dos de las

cuales interactúan con el mundo externo. La primera capa sirve de buffer, la que recepta y codifica las señales pixeladas de un sensor rectangular. Estas señales son transferidas linealmente hacia la capa intermedia que sirve de asociación aleatoria con la capa de entrada (también se puede asociar en forma total, lo que es interconexión entre y sobre todos los nodos implicados). Las señales son entonces transferidas por una función umbral de salida hacia la tercera capa llamada capa de respuesta o capa de perceptrones. La capa de respuesta contiene verdaderos perceptrones cuyos pesos no son fijos como en las capas anteriores: los pesos se modificaban en esta capa con el fin de que los perceptrones aprendieran a clasificar el patrón de entrada.



El perceptrón, como unidad de proceso, se describe tal y como se observa en la figura 2.6. La diferencia con respecto del modelo general de PE mostrado en la sección anterior es que el nodo posee una entrada  $x_0=1$ , la que provee una tendencia inicial sobre el nodo  $i$ -ésimo al multiplicarse por su respectivo peso inicial de tendencia  $w_0$ . La tendencia provista puede ser de naturaleza inhibitoria ( $w_0 < 0$ ) o excitatoria ( $w_0 > 0$ ). Otra característica particular del perceptrón es que posee una función de transferencia de tipo umbral. La función umbral hace que el perceptrón se dispare sólo si la suma ponderada es mayor que un valor umbral  $\theta_i$  para el  $i$ -ésimo nodo, de lo contrario el perceptrón se inhibir-a. Es decir que el perceptrón es de naturaleza binaria: si esta excitado arrojará el valor 1 y si esta inhibido arrojará el valor 0.

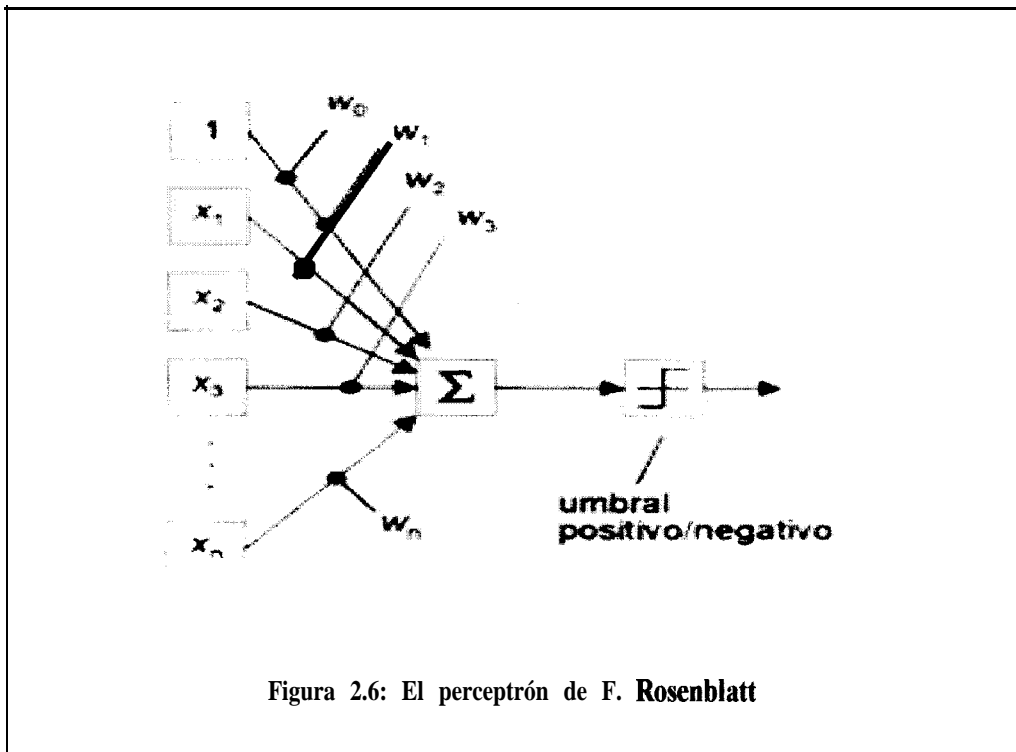


Figura 2.6: El perceptrón de F. Rosenblatt

Más claramente tendremos definida la siguiente función de salida para el  $i$ -ésimo perceptrón:

Dado un conjunto de entradas  $x_0, x_1, \dots, x_n$  donde  $x_0=1$ , y un conjunto de pesos  $w_0, w_1, \dots, w_n$  donde  $w_0$  es el peso de tendencia inicial, sea la entrada neta del nodo

$$neta_i = \sum_j x_j w_{ij}$$

donde  $j$  recorre los nodos antecesores al nodo  $i$  más la tendencia, entonces dado un valor umbral  $\theta_i \in \mathbb{R}$ , la función de transferencia  $f_i$  es umbral si:

$$f_i(neta_i) = \begin{cases} 0, & neta_i < \theta_i \\ 1, & neta_i > \theta_i \end{cases}$$

Por lo general (pero no necesariamente como veremos después) los valores umbrales  $\theta_i$  son 0.

El perceptrón se entrena mediante el “*método de descenso por el gradiente*”, que es un caso particularizado del método de búsqueda heurística de “subir *la colina*” (*steepest-ascent hill climbing*) o “*escalada por la máxima pendiente*”, utilizado en Inteligencia Artificial, con el que se intenta encontrar los pesos  $w_{ij}$  que resuelven el problema de correcta clasificación. En la sección siguiente analizaremos con mayor detalle la regla de aprendizaje del perceptrón al ser presentado el problema XOR.

### 2.4.2 Problema XOR.

Antes de entrar de lleno al problema XOR es necesario definir la forma en que el perceptrón aprende. Un perceptrón es capaz de separar linealmente una región para clasificar una entrada dada y definir si esta pertenece a un lado de la región o a su complemento. Veamos un ejemplo en dos dimensiones: Dadas dos entradas  $x_1, x_2$ , y sus pesos correspondientes, entonces

$$neta = w_1 x_1 + w_2 x_2$$

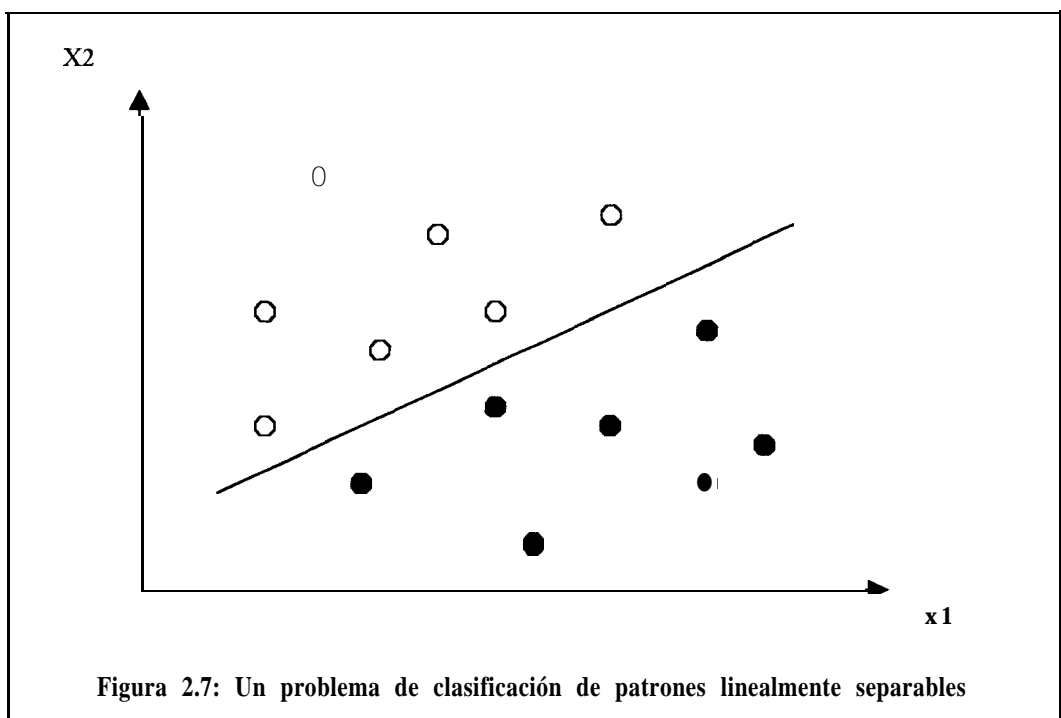
y dado un valor umbral  $\theta$  tendremos lo siguiente:

$$f(\text{neta}) = \begin{cases} 1, & w_1 x_1 + w_2 x_2 > \theta \\ 0, & w_1 x_1 + w_2 x_2 < \theta \end{cases}$$

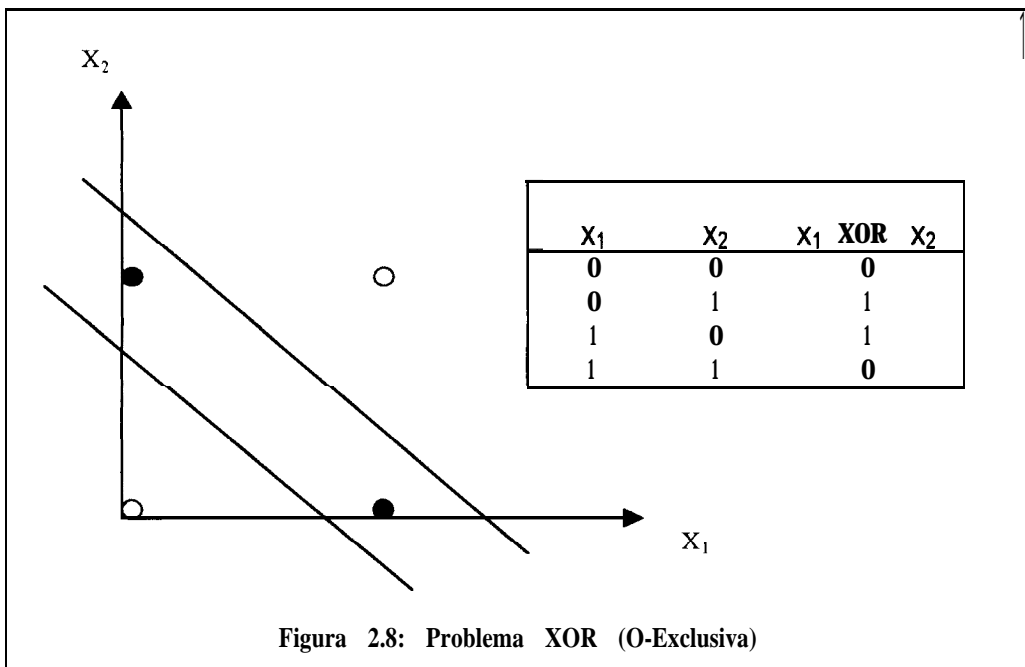
La ecuación de la recta

$$e = w_1 x_1 + w_2 x_2$$

divide a el plano  $x_1, x_2$  en exactamente dos regiones en las cuales se podrá decidir si la activación del perceptrón clasifica en 0 o en 1 a el par de entradas dado. Un problema es linealmente separable si existe una recta que pueda dividir exactamente en dos regiones al plano formado por el conjunto de entradas. El aprendizaje consiste en hallar los pesos que hagan que la recta se ajuste a la clasificación deseada. La recta en este caso se denomina espacio solución para el conjunto de pesos. Si nos encontráramos en múltiples dimensiones la recta se convierte en un hiperplano solución. Veamos la figura 2.7. Aquí podemos advertir dos clases de puntos. La recta proporciona la clasificación adecuada para estos puntos.



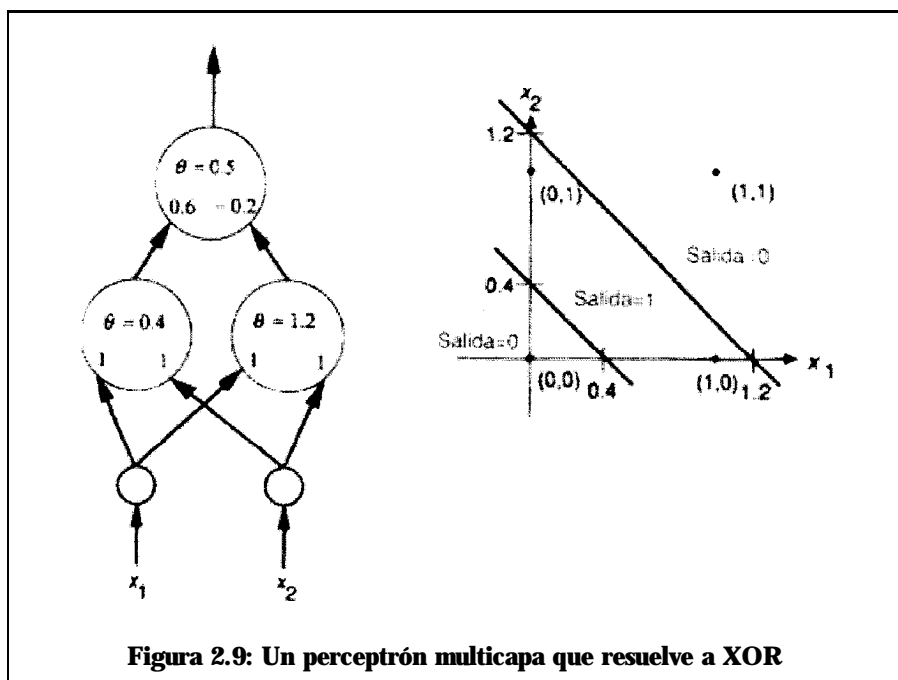
Minsky y Papert, dos investigadores del tema, expusieron las limitaciones del perceptrón al descubrir que el perceptrón por sí solo no podía dar soluciones a problemas que no fueran linealmente separables. La mayoría de los problemas del mundo real no gozan de separabilidad lineal por lo tanto la teoría de Rosenblatt se veía amenazada por las verdades de Minsky y Papert.



El problema XOR (Exclusive-OR) o O-Exclusiva, es un problema muy sencillo que no puede ser resuelto por un perceptrón. Al perceptrón llegan 2 entradas que pueden ser 0 ó 1 y la salida debe ser 0 si ambas entradas son iguales, y 1 si ambas entradas son distintas. Como se observa en la figura 2.8, no existe una recta que pueda separar correctamente una clase de la otra.



Frank Rosenblatt garantizaba, con *su teorema de convergencia del perceptrón*, que se encontrará un espacio solución con los pesos que separen correctamente la región para cualesquiera conjunto de entradas si el problema fuese linealmente separable. El problema XOR puede ser resuelto por un perceptrón de dos capas, pero no se ha demostrado que el teorema se pueda generalizar para perceptrones multicapas. Por lo tanto, Minsky y Papert concluyeron que si por un lado estaba garantizada la convergencia hacia un estado solución para un número finito de iteraciones de un solo perceptrón para un problema linealmente separable, y si por otro lado un perceptrón multicapa puede resolver un problema no separable linealmente, entonces el uso del perceptrón no provocaba ya más interés debido a que la mayoría de los problemas no gozan de separabilidad lineal y además no se garantizaba (y por tanto no se recomendaba su uso) que un perceptrón multicapa encontrara, en un número finito de iteraciones, el conjunto de pesos del espacio solución del problema. La figura 2.9 muestra un perceptrón multicapa que resuelve el problema XOR.



**Figura 2.9: Un perceptrón multicapa que resuelve a XOR**

## **2.5 Modelo de Redes de Propagación hacia Atrás (BPN).**

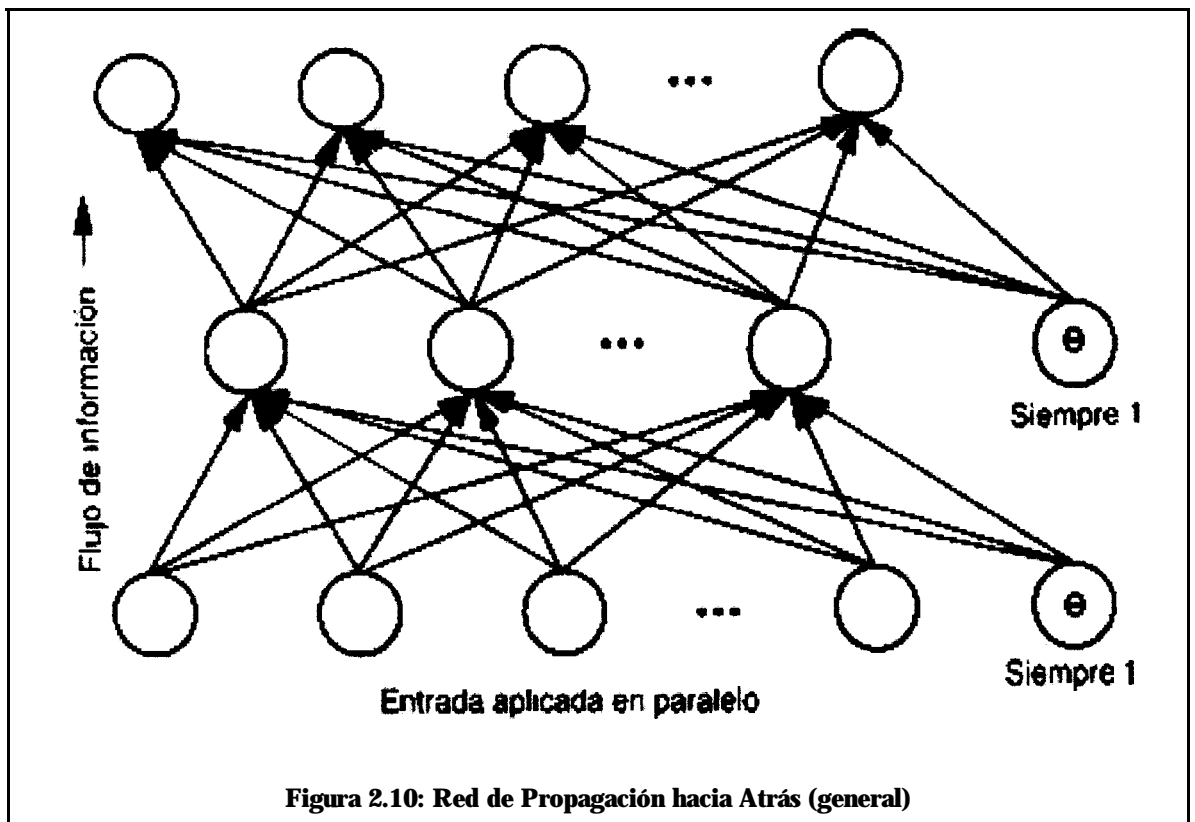
### **2.5.1 Introducción.**

Una red de perceptrones es capaz de entrenar las unidades de la capa de salida a aprender a clasificar patrones de entrada, dado que las clases son linealmente separables. Clases de mayor complejidad que no gozan de separabilidad lineal pueden ser separadas por una red multicapa. Sin embargo, si la salida incurre en un error, ¿cómo determinar que PE o interconexión ajustar? Este es el problema de “asignamiento de crédito” (propuesto por Minsky y Papert en su libro “Perceptrones”). La Propagación hacia Atrás (Back-Propagation) resuelve este problema al asumir que todos los elementos de procesamiento y conexiones tienen algo de culpa en la generación de una respuesta errónea. La responsabilidad por el error es asignada al propagar el error de salida al revés hacia las conexiones de la capa anterior. Este proceso se repite hasta que se llega hasta la capa de entrada. El nombre de “Redes de Propagación hacia Atrás” o BPN (Back-Propagation Networks) se deriva de este método de distribución de la “culpabilidad” por los errores.

Se conoce que los problemas de reconocimiento de tramas o patrones complejos así como los de proporcionamiento de funciones no triviales que devuelvan una buena aproximación para el mismo, son el uso frecuente que se les da a las BPN. Debido a que el problema de predicción de series temporales es una de estas situaciones, es por eso que presentamos aquí la descripción de la estructura BPN así como su regla de aprendizaje. Así como en el reconocimiento de tramas, las predicciones se dan en un entorno ruidoso, el cual es fácilmente resuelto por una BPN que aprenda a predecir.

### 2.5.2 Descripción de la estructura BPN.

La figura 2.10 muestra de una forma general el grafo de una red de propagación hacia atrás. De esta figura se advierten 3 tipos de capas como ya habíamos comentado en secciones anteriores: La capa de entrada, en donde se aplica un vector de entrada; las capas intermedias; finalmente la capa de salida. Las entradas se aplican en paralelo y las salidas son arrojadas de igual manera. Puede existir en todas las capas un nodo que contenga una tendencia (cuya entrada es la unidad como ya hemos mencionado), es decir, un peso de tendencia puro. Cada capa esta totalmente conectada con la subsiguiente. Las flechas indican la dirección en que la información viaja hacia la capa de salida.



**Figura 2.10: Red de Propagación hacia Atrás (general)**

Luego de que se ha aplicado un conjunto de entrada sobre la primera capa de la red a ser estimulada, esta señal se va propagando a través de todas las capas superiores hasta generar una salida. La señal de salida se compara entonces con la salida deseada, y se calcula una señal de error para cada unidad de salida. Los errores se transmiten hacia atrás para todas las unidades de las capas intermedias desde las unidades de la capa de salidas. El error que asume cada unidad es apenas una fracción del total y es proporcional a su contribución en el proceso en cuestión. Las ponderaciones de las conexiones son actualizadas en base a la proporción de error correspondiente. Este proceso se lleva a cabo hasta que la salida final converja hacia la verdadera salida dada, que complementa el conjunto de entrenamiento, es decir que el conjunto de entrenamiento en una BPN es la unión entre el conjunto de entrada y el conjunto de salida deseado.

### **2.5.3 Ley de Aprendizaje.**

La ley de aprendizaje o regla de aprendizaje que rige a las BPN es la llamada Regla Delta Generalizada o GDR (Generalized Delta Rule). Este es un algoritmo que provee de un método matemático para que se lleve a cabo lo descrito en la subsección anterior.

#### **2.5.3.1 Regla Delta Generalizada (paso a paso).**

Entre la entrada y salida de una BPN existe una relación necesaria para usar GDR. Con el objeto de entender la relación entre la entrada y la salida de la red, debemos primero

definir lo que es una red de correspondencia. Se dice que una red neuronal es una **red de correspondencia** si se demuestra capaz de calcular alguna relación funcional entre su entrada y su salida. Tal y como se pudiera decir que dado un número positivo  $x$ , siendo la salida su logaritmo natural, se establezca por medio de la red, la correspondencia  $x \rightarrow \ln(x)$ . Es evidente que para un cálculo tan sencillo no se necesita una red neuronal, pero que tal si la relación funcional fuera demasiado intrínseca (y no se diga multidimensional) y que sólo se disponga de ejemplos con valores de entrada y salida, mas no de una relación funcional anticipadamente prevista. Cuando sólo se dispone de valores de correspondencia en forma de ejemplos, la red aprende a descubrir la relación en base a estos.

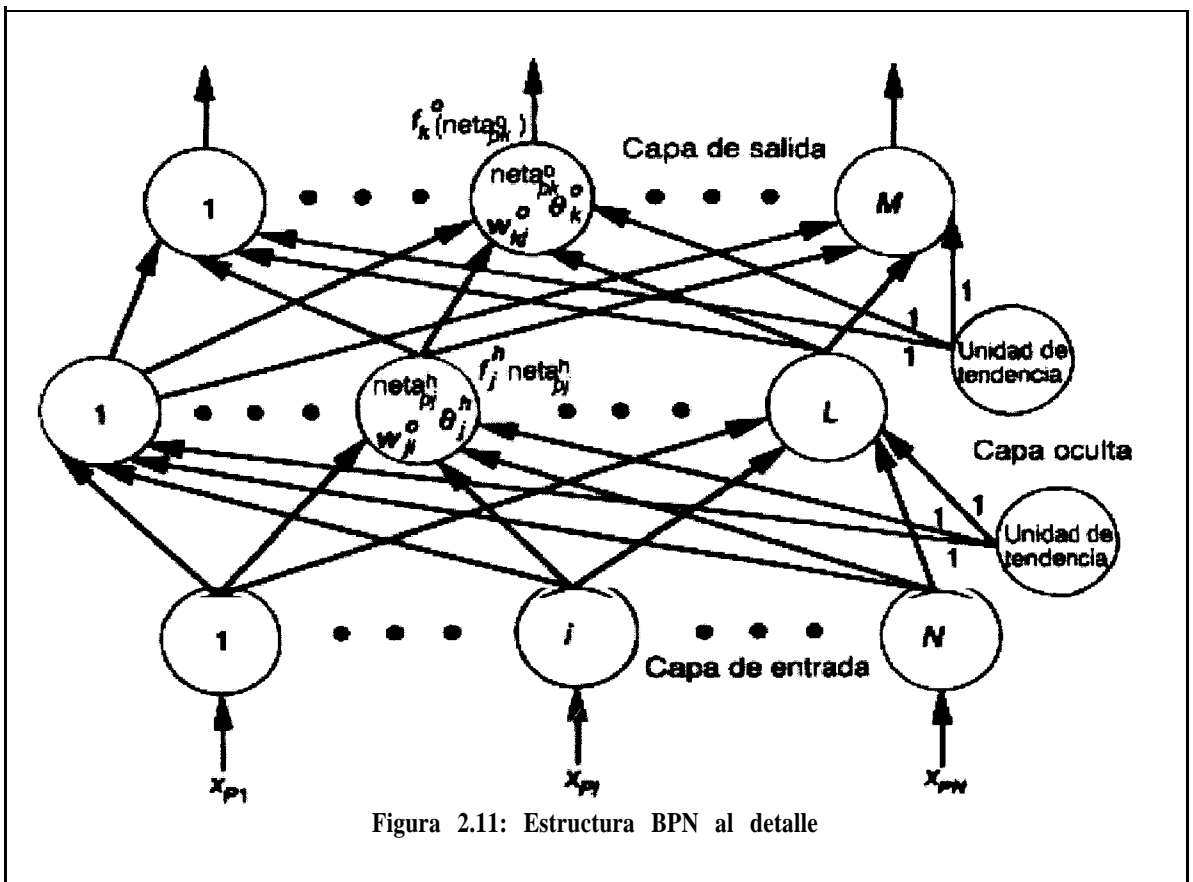
Ahora podemos suponer que tenemos un conjunto de  $P$  pares de vectores  $(x_p, y_p)$ ,  $(x_2, y_2), \dots, (x_p, y_p)$ , los que son ejemplos de una correspondencia funcional  $y = \Phi(x) : x \in \mathbf{R}^N$ , y  $y \in \mathbf{R}^M$ . Quisieramos hacer que la BPN aprenda una aproximación de  $y = \Phi(x)$ . El método que a continuación se describe proporciona una buena aproximación a la correspondencia funcional dados conjuntos de entrenamiento seleccionados de forma *adecuada y suficiente*. Freeman y Skapura proporcionan una *definición de adecuada y suficiente*; veremos estos criterios en una sección posterior. Lo que a continuación vamos a hacer es describir un proceso iterativo en el que se trata de encontrar un conjunto de pesos que minimize el error global arrojado por la red. Como podemos inferir, parece que nos encontramos frente un caso más generalizado de el método de mínimos cuadrados **comunmente** utilizado en regresión lineal.

Observemos la figura 2.11, en la cual se aprecia una red de tres capas. Esta arquitectura se asemeja mucho a la versión general de una red neuronal cualesquiera que mostramos en una sección anterior. Existen  $N$  nodos de entrada para que en efecto ingrese el  $p$ -ésimo vector de entrada  $\mathbf{X}_p = (x_{p0}, x_{p1}, \dots, x_{pN})^t$ , en donde  $x_{p0} = 1$  (*por el peso de tendencia*); luego las salidas de esta capa

se propagan hacia la capa intermedia (u oculta) que posee  $L$  nodos. Entonces se calcula la entrada neta de la  $j$ -ésima unidad oculta:

$$neta_j^h = \sum_{i=1}^N w_{ji}^h x_{pi} + \theta_j^h \quad (2.1)$$

En donde  $w_{ji}^h$  es el peso de aquella conexión proveniente del  $i$ -ésimo nodo de entrada, y  $\theta_j^h$  representa la tendencia pura (con entrada =1) de la capa. La letra  $h$ , como supraíndice, representa magnitudes de la capa que permanece oculta.



La salida de este nodo (en total  $M$  salidas de la red) es:

$$i_{pj} = f_j^h(neta_{pj}^h) \quad (2.2)$$

Análogamente, para la capa de salida tenemos las siguientes sumas y salidas:

$$neta_{pk}^o = \sum_{j=1}^L w_{kj}^o i_{pj} + \theta_k^o \quad (2.3)$$

$$o_{pk} = f_k^o(neta_{pk}^o) \quad (2.4)$$

Y de la misma manera el supraíndice “o” representa las magnitudes de la capa de salida.

A partir de este conjunto de ecuaciones podemos detallar el siguiente algoritmo básico que entrena a la red:

1. Un vector de entrada es aplicado a la red y luego se calculan los valores de salida.
2. Se determina un término de error en base a la comparación de las salidas resultantes con respecto a las verdaderas.
3. Se decide en qué dirección (positiva o negativa) debe cambiar cada uno de los pesos y así poder disminuir el error.
4. Se calcula la magnitud que debe cambiar cada ponderación.
5. Las conexiones son aplicadas sobre los pesos.
6. Todos los pasos del 1 al 5 se repiten con todos los vectores de entrenamiento hasta que el error para cada uno de ellos se reduzca a una cantidad mínima.

La regla Delta (GDR) establece el siguiente criterio de modificación sobre los pesos con el objeto de que la red aprenda de los vectores de entrenamiento:

$$w(t+1)_i = w(t)_i + 2\mu e_k x_{ki} \quad (2.5)$$

En donde  $\mu$  es una constante positiva,  $x_{ki}$  es la  $i$ -ésima componente del  $k$ -ésimo vector de entrenamiento y  $e_k$  es la diferencia entre la salida obtenida y el valor correcto,  $e_k = (d_k - y_k)$ .

Tal y como lo presentan Freeman y Skapura (salvo algunos errores de notación en su texto) vamos a ser fieles en presentar la derivación de la regla delta paso a paso. Para esto ellos dividen la derivación del método, para una red con una capa oculta, en la actualización de los

pesos de la capa de salida y, por otro lado, la actualización de la capa oculta; con esto que daría completa la descripción detallada de GDR.

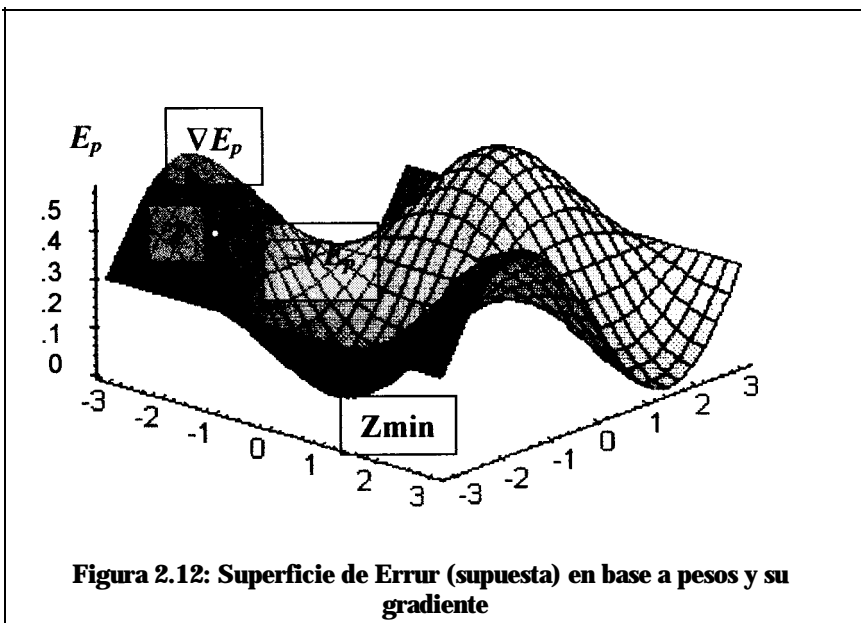
### 2.5.3.1.1 Obtención de los pesos de la capa de salida.

Definamos el error de una sola unidad de salida como  $\delta_{pk} = (y_{pk} - o_{pk})$ , en donde el subíndice  $p$  se refiere al  $p$ -ésimo vector de entrenamiento, y  $k$  se refiere a la  $k$ -ésima unidad de salida. Como se dijo al principio  $y_{pk}$  es el valor de salida deseado y  $o_{pk}$  es la salida que se obtiene del  $k$ -ésimo nodo. Como en el método de mínimos cuadrados (salvo la constante que se utiliza por conveniencia al derivar) deseamos minimizar la siguiente expresión:

$$E_p = \frac{1}{2} \sum_{k=1}^M \delta_{pk}^2 \quad (2.6)$$

En pasos subsiguientes se justificara la presencia del factor  $1/2$ .

Como el proceso es iterativo, deseamos saber en que sentido debemos cambiar los pesos. Para esto se calcula el valor negativo de el gradiente de  $E_p$ ,  $-\nabla E_p$ , respecto a los pesos  $w_{kj}$ . Luego se ajustan los valores de los pesos de manera que se reduzca el error total.





En la figura 2.12 se presenta la superficie supuestamente generada por la ecuación 2.6 en la que se asume un conjunto de pesos compuesto por apenas dos elementos. Observese lo compleja que se toma la superficie y cual es la idea tras el criterio del gradiente. Si estamos en el punto  $\zeta$  desearemos llegar al punto  $\zeta_{min}$ , el cual hace que  $E_p$  sea mínimo, por lo tanto el par de pesos al que se llegue harán que el error para este vector de entrenamiento sea el menor posible. Intuitivamente nos damos cuenta de que, ya que  $\nabla E_p$  proporciona la dirección del ascenso más pronunciado, entonces  $-\nabla E_p$  proporcionará el **descenso más pronunciado** y por tanto será la dirección que debemos seguir para llegar al mínimo.

Tenemos entonces de la ecuación 2.6 y de la definición de  $\delta_{pk}$ ,

$$E_p = \frac{1}{2} \sum_k (y_{pk} - o_{pk})^2 \quad (2.7)$$

Y

$$\frac{\partial E_p}{\partial w_{kj}^o} = -(y_{pk} - o_{pk}) \frac{\partial f_k^o}{\partial (neta_{pk}^o)} \frac{\partial (neta_{pk}^o)}{\partial w_{kj}^o} \quad (2.8)$$

en donde utilizando la ecuación 2.4 (salida  $o_{pk}$ ), se hace uso de la regla de la cadena para las derivadas parciales. Notece que el factor  $1/2$  desaparece al derivar. Para no evaluar las derivadas de  $f_k^o$ , las escribiremos por el momento como  $f_k^{o'}(neta_k^o)$ , siendo el último factor de la ecuación 2.8:

$$\frac{\partial (neta_{pk}^o)}{\partial w_{kj}^o} = \left( \frac{\partial}{\partial w_{kj}^o} \sum_{j=1}^L w_{kj}^o i_{pj} + h_k^o \right) = d_{pj} \quad (2.9)$$

Al combinar las ecuaciones 2.8 y 2.9, nos queda el gradiente negativo:

$$-\frac{\partial E_p}{\partial w_{kj}^o} = (y_{pk} - o_{pk}) f_k^{o'}(neta_{pk}^o) d_{pj} \quad (2.10)$$

Con respecto a la magnitud del cambio de peso, se considera que será proporcional al gradiente negativo. Entonces, la actualización de los pesos de la capa de salida están dados por:

$$w_{kj}^o(t+1) = w_{kj}^o(t) + \Delta_p w_{kj}^o(t) \quad (2.11)$$

en donde

$$\Delta_p w_{kj}^o = \eta(y_{pk} - o_{pk}) f_k^o(neta_{pk}^o) d_{pj} \quad (2.12)$$

Encontramos un nuevo término  $\eta$  que multiplica la anterior expresión: se denomina **parámetro de velocidad de aprendizaje y es** positivo y menor que 1. En una sección posterior profundizaremos acerca de este factor.

Recordemos ahora que en los perceptrones la función de salida era alguna de tipo umbral, pero en este caso es evidente que  $f_k^o$  debe ser derivable. Entonces, en este punto se hace necesario **definir** la nueva función de salida. Según Freeman y Skapura existen dos formas de la función de salida que son interesantes (aunque hay autores que mencionan otras funciones que no analizaremos aquí):

$$\longrightarrow f_k^o(neta_{jk}^o) = neta_{jk}^o$$

$$\longrightarrow f_k^o(neta_{jk}^o) = (1 + e^{-neta_{jk}^o})^{-1}$$

Sobre la primera función sólo se puede acotar que es la identidad (en este punto de la tesis no se disponen de argumentos que resalten el uso de esta función, aunque posteriormente se puede dar que la experiencia nos haga concluir algo destacable sobre el tema). En el segundo caso, a esta función se la denomina sigmoide y viene a ser una especie de función umbral suavizada debido a su forma (figura 2.13). Debido a que el uso de esta función es más común, lo es también el hecho de que los datos se representen de forma binaria, pero nada hemos encontrado que impida que una serie temporal sea llevada a intervalos entre 0 y 1 para ser representadas con la sigmoide. Parecería más lógico usar para nuestro problema la función lineal, pero eso dependerá **esencialmente** del programa que usemos para resolverlo.

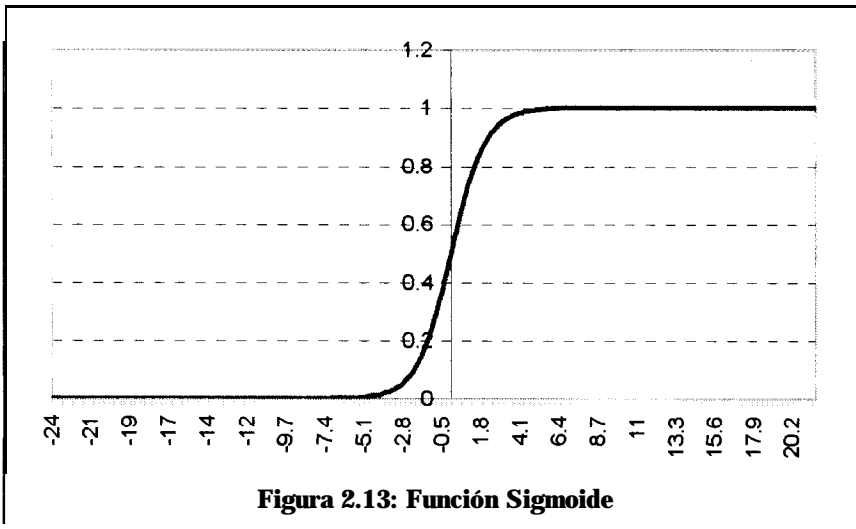
Resolviendo las derivadas en cada caso tendremos  $f_k^{o'} = 1$ , y  $f_k^{o'} = o_{pk}(1 - o_{pk})$  respectivamente; en sendos casos tenemos:

$$w_{kj}^o(t+1) = w_{kj}^o(t) + \eta(y_{pk} - o_{pk}) d_{pj} \quad (2.13)$$

en la salida lineal, y

$$w_{kj}^o(t+1) = w_{kj}^o(t) + \eta(y_{pk} - o_{pk})o_{pk}(1-o_{pk})d_{pj} \quad (2.14)$$

para la salida sigmoide.



Se desea resumir las ecuaciones de actualización de pesos al definir una magnitud

$$\begin{aligned} \delta_{pk}^o &= (y_{pk} - o_{pk})f_k^{\prime}(neta_{pk}^o) \\ &= \delta_{pk}f_k^{\prime}(neta_{pk}^o) \end{aligned} \quad (2.15)$$

Con esto se puede escribir la ecuación de actualización de pesos en la forma

$$w_{kj}^o(t+1) = w_{kj}^o(t) + \eta\delta_{pk}^o d_{pj} \quad (2.16)$$

sin importar la forma funcional ( $f_k^o$ ) que tenga la salida de cada nodo.

Recordemos ahora el método de mínimos cuadrados. En él se pretendería conocer todos los vectores de entrenamiento antes de hacer algún cambio a los pesos de la red, pero computacionalmente este criterio sería costoso en espacio y memoria. Para que se entienda mejor, el error a minimizar si siguiéramos una analogía completa con el método de mínimos cuadrados sería

$$E_p = \sum_{p=1}^P E_p \quad (2.17)$$

siendo  $P$  el número de ejemplos vectoriales que se poseen. Más allá de lo que digan los expertos, parece lógico que las iteraciones se hagan por cada vector mas no por toda la matriz de datos – de todas formas Freeman y Skapura recomiendan iterar por cada trama de ejemplos.

### 2.5.3.1.2 cambio de pesos en la capa oculta.

Para dar la respectiva “culpabilidad” a los pesos de la capa oculta sobre el error  $E_p$ , fijemonos primero de que forma están relacionados estos pesos con el error total:

$$\begin{aligned} E_p &= \frac{1}{2} \sum_k (y_{pk} - o_{pk})^2 \\ &= \frac{1}{2} \sum_k (y_{pk} - f_k^o(\text{net}a_{pk}^o))^2 \\ &= \frac{1}{2} \sum_k (y_{pk} - f_k^o(\sum_j w_{kj}^o i_{pj} + \theta_k^o))^2 \end{aligned}$$

Veamos en las ecuaciones 2.1 y 2.2 que  $i_{pj}$  depende de los ponderadores de la capa oculta, por lo tanto debemos de obtener el gradiente de  $E_p$  con respecto a los pesos de supraindice  $h$ . De esta forma y utilizando la regla de la cadena tendremos :

$$\begin{aligned} \frac{\partial E_p}{\partial w_{ji}^h} &= \frac{1}{2} \sum_k \frac{\partial}{\partial w_{ji}^h} (y_{pk} - o_{pk})^2 \\ &= - \sum_k (y_{pk} - o_{pk})^2 \frac{\partial o_{pk}}{\partial (\text{net}a_{pk}^o)} \frac{\partial (\text{net}a_{pk}^o)}{\partial i_{pj}} \frac{\partial i_{pj}}{\partial (\text{net}a_{pj}^h)} \frac{\partial (\text{net}a_{pj}^h)}{\partial w_{ji}^h} \end{aligned} \quad (2.18)$$

Al resumir este resultado, es posible verificar que llegaremos a:

$$\frac{\partial E_p}{\partial w_{ji}^h} = - \sum_k (y_{pk} - o_{pk})^2 f_h^{o'}(\text{net}a_{pk}^o) w_{kj}^o f_j^{h'}(\text{net}a_{pj}^h) x_{pi} \quad (2.19)$$

Por lo tanto debemos de **actualizar** los pesos de la capa oculta proporcionalmente al valor negativo de la ecuación 2.19, de tal manera que:

$$\Delta_p w_{ji}^p = \eta f_j^{h'}(\text{net}a_{pj}^h) x_{pi} \sum_k (y_{pk} - o_{pk})^2 f_h^{o'}(\text{net}a_{pk}^o) w_{kj}^o \quad (2.20)$$

siendo  $\eta$  el parámetro de velocidad de aprendizaje del que se hizo referencia en la sección anterior.

Dada la definición de  $\delta_{pk}^o$  se puede escribir

$$\Delta_p w_{ji}^p = \eta f_j^{h'} (net_{pj}^h) x_{pi} \sum_k \delta_{pk}^o w_{kj}^o \quad (2.21)$$

Todos los términos de error  $\delta_{pk}^o$  de la capa de salida provocan dependencia sobre todos los pesos de la capa oculta. Esa dependencia es lo que nos hace formar la idea de propagación hacia atrás, en la cual el cambio de pesos en una capa oculta solo puede ser provocado por un término de error propagado desde la capa de salida. El término de error para la capa oculta se define como

$$\delta_{pj}^h = f_j^{h'} (net_{pj}^h) x_{pi} \sum_k \delta_{pk}^o w_{kj}^o \quad (2.22)$$

entonces análogamente al proceso en la capa de salida tenemos:

$$w_{ji}^h(t+1) = w_{ji}^h(t) + \eta \delta_{pj}^h x_i \quad (2.23)$$

Con esta última ecuación se da por concluido el ciclo que cumple GDR.

#### 2.5.4 Criterios Prácticos y de Convergencia.

En esta sección vamos a dar brevemente ciertos criterios que nos orientaran de una mejor forma a seleccionar nuestra estructura BPN para resolver nuestro problema de predicción (aunque los criterios ciertamente sirven para ayudar a resolver cualquier problema en general en el que se escoja BPN como un buen modelo solución). Freeman y Skapura proveen tres temas sobre los cuales hay que decidir sobre un modelo de propagación hacia atrás. Primero, sobre los datos de entrenamiento; segundo, acerca del dimensionamiento de la red; y tercero, sobre los pesos y parámetros de aprendizaje.

Tengamos en cuenta que aunque estos criterios complementan la profundidad del tema, la resolución de nuestro problema va a depender del paquete computacional que utilicemos para hacerlo y, por tanto, estos criterios dependen de la forma en que se implementó este programa. De manera que los criterios de Freeman y Skapura nos deben servir para diseñar el modelo solución de red que hace predicciones mediante asignación de parámetros sobre el paquete de software que usemos.

#### 2.5.4.1 Datos de entrenamiento.

Freeman y Skapura afirman algo que nos debe parecer razonable: “la experiencia suele ser la mejor maestra”. Por eso es que no existe una definición específica para “un conjunto de vectores de ejemplo *además y suficientes*”. Lo que se dice es que en términos generales se utiliza la mayoría de los datos disponibles salvo un pequeño conjunto que se debe dejar para probar la red. También se dice que es recomendable incluir ejemplos con ruidos aunque el entorno del problema no sea ruidoso, esto ayudaría al sistema a efectivamente converger. Otro criterio dice que una red de propagación hacia atrás admite bien la generalización dentro de una misma clase ejemplos. La red tomará en cuenta las similitudes significativas entre los datos y descartará aquella información irrelevante.

Hay un criterio que en principio hace parecer como si este modelo de red neuronal no fuese adecuado para hacer predicciones: “la BPN no extrapolará bien”. ¿Qué debemos entender con respecto a este criterio? Hagamos una abstracción de lo que serían las clases en una serie de tiempo. Básicamente estamos hablando de *comportamientos distintos en tiempos distantes*. Pensemos en que si la serie fuese estacional, luego dejara de serlo o cambiara su periodo, entonces la parte que ha cambiado debe ser tratada y modelada como una

serie distinta. Y que tal si pedimos como salida un número muy grande de predicciones? Estaríamos hablando de mucha distancia en el tiempo como para que las predicciones sean confiables. Por lo tanto ellos dicen: “no entrene por completo a la red con vectores de una clase, pasando después a otra clase; la red se *olvidará* del entrenamiento original”. Por eso debemos recordar que estamos construyendo un modelo de predicción limitado y razonable, más no estamos inventando una computadora adivina.

Por último se refieren a que la función sigmoide nos obligará a aplicar una escala sobre los datos que utilizemos debido a que la salida siempre estará entre cero y uno. De todas formas sabemos que la función puede ser desplazada y modificada en su parte pendiente y que todo cambio dependerá del problema y de sus datos.

#### **2.5.4.2 Dimensionamiento de la red.**

Es importante conocer la naturaleza del problema que se desea resolver para saber cuantos nodos y cuantas capas debe tener la BPN. En general, el número de nodos en la capa de entrada es el más fácil de saber con anticipación. Este va a depender de la naturaleza del problema y por ende de los datos. En la sección 2.5.5 veremos con profundidad este enfoque sobre la naturaleza de nuestro problema. Es también un tanto evidente suponer cuantos nodos tendrá la capa de salidas: será consecuente con el vector de salida o respuesta que el problema exija.

Con respecto a las capas intermedias es menos evidente saber cuantas de estas debe de tener la BPN, y cuantos nodos debe tener cada una de ellas. Se conoce que, por lo general, con una sola capa intermedia será suficiente como para resolver una tarea, pero Freeman y Skapura aseguran que cuando se ponen de por medio más capas la red aprenderá con mayor

rapidez (si el problema lo permite). La experiencia es la única herramienta de criterio que se tiene para determinar el número de nodos que la capa intermedia debe tener; sin embargo el sentido común nos dirá que si es que no hay convergencia debemos probar con una cantidad mayor de nodos, y si hay convergencia en un primer intento, entonces debemos reducir el número de nodos hasta un tamaño óptimo. Pero para nuestro problema de predicción esto no debería ser ninguna desventaja ya que este criterio del *tanteo* sirve para redes con ciento o miles de nodos y nosotros no utilizaremos vectores de entrada tan grandes y menos en la capa de salida, que servirá como predictora.

#### **2.5.4.3 Pesos y Parámetros de velocidad de aprendizaje.**

Los valores iniciales de los pesos en la red (incluyendo los de tendencia) deben ser generados aleatoriamente al inicializar la red con valores en un intervalo uniforme y pequeño. Dependiendo de la naturaleza de los datos estos pesos podrían ser negativos también. Se puede decir por ejemplo un intervalo  $\pm 0,5$ . Se puede eliminar tranquilamente los parámetros de tendencia o “BIAS terms” como usualmente son llamados por los paquetes neuronales.

Otro parámetro es el de aprendizaje, el cual define la rapidez con que el ANS aprende; debido a que hay que asegurar el asentamiento de la red sobre un conjunto solución,  $\eta$  debe estar en el orden de 0,05 a 0,25. Si  $\eta$  es excesivamente pequeño entonces la convergencia será lenta pero con mayor grado de seguridad para una solución. Si  $\eta$  es grande la convergencia se acelera pero se incurre en el riesgo de alejarse de la solución en un solo paso al “saltarse” del mínimo.



Existe otra posibilidad para el manejo de la velocidad de convergencia de la red que es la inclusión dentro de las ecuaciones 2.23 y 2.16 **un** término de **momento** el que consiste de que luego del calculo del cambio de peso  $\Delta_p w$ , se añade una fracción del cambio anterior el cual hace que el cambio conserve la dirección previa. Se tendría entonces:

$$w_{kj}^o(t+1) = w_{kj}^o(t) + \Delta_p w_{kj}^o(t) + \alpha \Delta_p w_{kj}^o(t-1) \quad (2.24)$$

Análogamente se haría para la capa oculta, siendo  $\alpha$  el **parámetro** de momento que suele ser un valor positivo menor que 1. Esta técnica de convergencia también es opcional.

Finalmente es necesario referirse a los asentamientos sobre mínimos locales. La red parará el entrenamiento definitivamente aunque converja a un mínimo local, lo que no debería ser un problema en la práctica ya que los expertos dicen que si esto ocurre hay dos posibilidades: Que el error sea bajo por lo que no importará entonces si el mínimo es global o local y la solución será realmente satisfactorias; la segunda opción será que el error sea aun muy alto por lo que simplemente se debe de empezar nuevamente el entrenamiento con un conjunto distinto de pesos iniciales. Algo si es seguro y hasta lógico: “no hay garantia de que el mínimo alcanzado sea global o local” por lo que el control sobre el error será determinante en el entrenamiento.

### 2.5.5 Estructura BPN que Resuelve el Problema de Predicciones en Series de Tiempo.

Como a estas alturas ya nos debe de sonar muy lógico, la estructura BPN que resuelve determinado problema de predicción dependerá de los datos que tengamos. Pero, ¿qué significa esto específicamente? **Vamos** a tratar de responder esta pregunta con simple sentido común y lo más claramente posible.

En primer lugar, es saludable **graficar** la serie de datos que se tienen, para así tener la primera referencia sobre posibles tendencias, estacionalidades y comportamiento en general. Una vez **graficados** los datos, sería sugerible hacer lo mismo con las autocorrelaciones de la serie para comprobar nuestras sospechas, pero sin la intención de modelar por análisis de series de tiempo a el grupo de datos (valga la aclaración). Si descubrimos que la serie tiene estacionalidad al orden  $s$  entonces ya tendremos información sobre el número de nodos en la capa de entradas de la BPN. ¿Y cómo es esto posible? La respuesta es muy sencilla en realidad: si necesitamos un patrón de datos para la capa de entrada, es lógico que sea un grupo específicamente distinguible, y en este caso este conjunto existe y se presenta periódicamente cada  $s$  espacios temporales. También es sugerible que se tenga cierta familiaridad con los datos como para tener criterios de decisión sobre los nodos en la capas de entrada y salida: si los datos no fuesen estacionales a simple vista ¿cómo vamos a decidir cuantos nodos se tiene que poner en la capa de entadas? Cuantas predicciones debe hacer la capa de salidas? En realidad esto queda a criterio del modelador, que tiene la ventaja (según el programa que utilice) de modificar los **parámetros** de aprendizaje y demás elementos de la red, haciendo pruebas hasta encontrar un modelo que se ajuste de mejor manera.

Cómo predice la red de Propagación hacia Atrás? La idea es muy sencilla pero un tanto compleja de implementar como para poder hacerlo con cualquier tipo de datos (de ahí que los paquetes computacionales que hacen esto con **flexibilidad** son relativamente costosos). Primero se tendrá un grupo suficientemente grande de datos de una serie temporal de los que la mayoría se utilizará para entrenar la red y un pequeño subconjunto se utilizará para hacer pruebas. En la capa de entrada de la red ingresarán iterativamente los vectores de entrenamiento conformados por elementos de subconjuntos de la serie de datos originales. Esta entrada originará una salida que será el vector “predictor” de la red. Sobre este primer vector se iterará con la regla Delta hasta minimizar el error. Luego, el siguiente vector de entradas será tal que estará conformado por elementos  $x_{t+j}$  para el siguiente valor de  $t$  (y algún salto  $j$  en el tiempo) y así sucesivamente.

Algo peculiar es que los vectores de comparación con la salida se van tornando en vectores de entrada a medida que se recorre el conjunto de datos. Por ejemplo, supongamos que tenemos una serie de datos mensuales de volúmen de ventas de juguetes. Como sabemos estas ventas son mayores en la última parte del año y además año a año estas aumentan debido al crecimiento poblacional (es decir los datos tienen estacionalidad al orden 12 y una pendiente de no estacionaridad). Entonces en la capa de entrada de la BPN tendremos 12 nodos (ó 13 con el peso de tendencia de entrada 1). Haremos las iteraciones de forma tal que tomemos los primeros 12 datos y predigamos el décimotercero; comparemos e iteremos sobre el error en esa predicción. Luego tomaremos los siguientes 12 datos a partir del segundo dato, es decir que ahora el número que se predijo en la fase anterior pasa a ser el último dato del nuevo vector de entrenamiento para predecir el decimocuarto dato. Cuando hayamos terminado de recorrer el conjunto de entrenamiento, podremos hacer pruebas para medir el poder **predictivo**

de la red. Finalmente podemos utilizar la red para hacer predicciones reales de eventos que aún no ocurren. En este caso la red sólo tenía un nodo en la capa de salida.

Este proceso iterativo tiene otras implicaciones en el diseño. Los parámetros a ser modificados van desde un ajuste de los datos, pasando por la correcta elección de una función de salidas (si se utiliza la sigmoide habrá que modificar su pendiente: si es 1 esta cambiará de 0 a 1 aproximadamente en el intervalo  $[-8,+8]$  como lo muestra la figura 2.13 ), generación de pesos aleatorios, elección de términos de momento, velocidad de aprendizaje, etc. Por lo tanto si se quiere poder hacer predicciones de diferentes series de tiempo esto implicará un diseño distinto de un programa por cada caso que se nos presente o una flexibilidad a gran escala de un paquete computacional de uso profesional.

Por otra parte, con respecto al número de nodos de la capa intermedia, estos no serán más allá del 100% de la cantidad de nodos de la capa de entrada. En realidad, para empezar, deberíamos de poner igual cantidad de nodos en la capa intermedia como en la de entrada.

Por último, en la capa de salida se deben de poner tantos nodos como pasos hacia el futuro queramos predecir. Está claro que mientras mas nos alejemos del origen temporal mayor será la probabilidad de obtener grandes imprecisiones en las predicciones, por eso es sugerible que no se exagere en la cantidad de predicciones. Una cantidad proporcional a la cantidad de nodos en la capa de entrada es sugerible para la capa de salida.

Por cada una de estas modificaciones en cada capa se necesitará una estructura distinta, por lo que se debe de definir una estructura de datos distinta en el paquete neuronal y otra vez nos topamos con la inminente necesidad de flexibilidad de un paquete neuronal profesional.

## **CAPITULO III**

### **3 Uso de un Paquete Computacional de Redes Neuronales.**

#### ***3.1 Introducción.***

Un paquete neurocomputacional es un software que simula redes neuronales y su comportamiento paralelístico; este demanda computación intensa de datos matemáticos y usualmente conllevan hacia tiempos de entrenamiento excesivos en procesadores de propósito general. Sin embargo y aunque existen microprocesadores especiales para esta simulación, los microprocesadores de hoy son muy veloces y cada vez las limitaciones espacio-tiempo (tamaño del hardware y velocidad con que este opera) se desvanecen frente a la rápida escalada de la microtecnología. Por tanto con una máquina actual se podrá correr un programa de redes neuronales sin mayor problema. En nuestro caso, debido a que no se necesitan ni de miles ni de cientos de nodos para que la ANS aprenda a predecir estamos aun más lejos de mayores

inconveniencias. Seguramente para problemas a gran escala si se necesitará de computadores muy poderosos y veloces.

A continuación presentaremos dos programas de redes neuronales que específicamente son capaces de aprender a predecir mediante Propagación hacia Atrás; el primero es muy limitado ,está hecho en lenguaje C ++ y predice únicamente manchas solares anuales y, el segundo es ya un paquete neuronal completo del cual se puede obtener experiencia para modelar redes neuronales. La intención de este capítulo es familiarizarnos con el uso de un paquete de ANS que utilice BPN para poder ser capaces de asimilar cualquier programa sin caer en mayores retrasos o dificultades por su manejo. Nos enfocaremos en realidad en el segundo paquete ya que su uso es muchísimo más didáctico que el primero. Cabe señalar que este programa es realmente un “demo” y como tal tiene las limitaciones de la versión. Una versión profesional se encuentra disponible pero habría que pagar la licencia.

### 3.2 Un ejemplo: Programa de **Karsten Kutza** en **C++** .

En realidad este programa (que se anexa al final como Anexol) no permite interactividad con el usuario para nada; es tan sólo un ejemplo muy bien implementado de cuan capaz es una red neuronal de aprender a hacer nuestra tarea de predicción por medio de la propagación inversa. No se carga ningún archivo de datos ya que estos se encuentran dentro de la programación con su estructura predefinida. Lo único que hay que hacer es ejecutar el programa y esperar a que un archivo con los resultados se produzca dentro de la misma carpeta en que se encuentra el programa.

El programa se llama **Bpn** y predice el número anual de manchas solares, que es una serie de tiempo muy especial ya que no es realmente ajustable a un modelo estacional debido a que su periodicidad varía (esta serie ha sido objeto de mucha investigación). El programa tiene un arreglo con un registro de 280 datos que se ajustan en el intervalo  $[0,1]$  ya que por obvias razones la salida sigmoidea de una BPN no permitirá otro rango. Este programa utiliza además términos de momento y pesos de tendencia o “bias”.

Para aprender algo más sobre el programa tendríamos que metemos a su programación y eso en realidad no es de nuestro interés y escapa a nuestros objetivos. Lo destacable es saber que de hecho el algoritmo de propagación hacia atrás (GDR) es uno de los más utilizados con un número relativamente general de propósitos y de hecho sirve para predecir, lo que reafirma nuestro interés por las BPN.

De los resultados de este programa hablaremos en el siguiente capítulo, en el que hacemos la aplicación de modelos de redes neuronales que aprenden a predecir con regla delta.

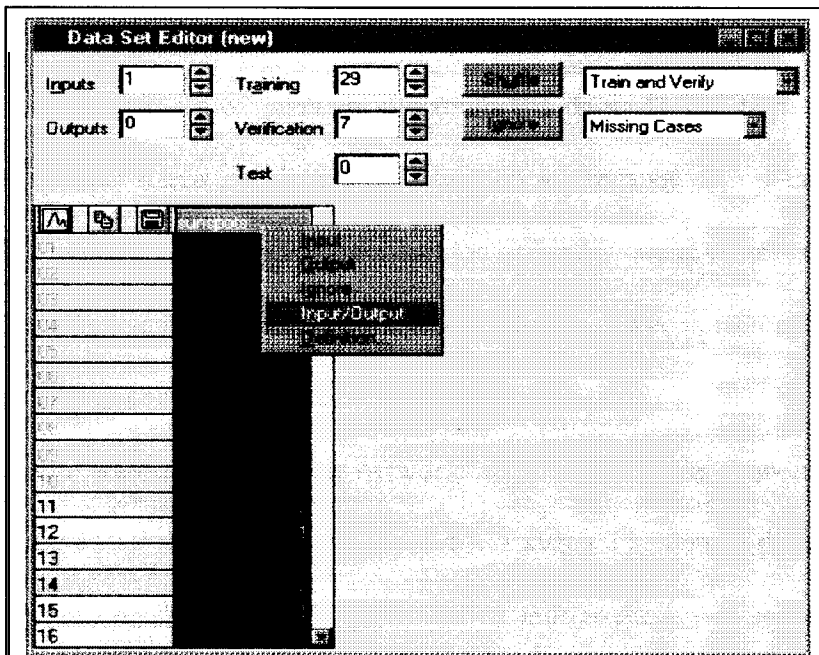
### 3.3 *Un software sofisticado de redes **neuronales** que permite hacer predicciones de series de tiempo: **TRAJAN 3.0** .*

Aunque este programa no se centra únicamente en nuestro problema, nosotros vamos a concentrarnos en describir su manejo consecuente únicamente con lo que es de nuestro interés en el contexto de esta tesis. Es **válida** esta aclaración ya que este paquete, como seguramente otros más, tienen una extensa gama de funciones, redes, algoritmos, etcétera, para una diversa cantidad de problemas para los que las redes neuronales proveen una solución. Vamos a enfocarnos en las redes de perceptrones multicapas, algoritmo de BPN, manejo de parámetros

de este tipo de aprendizaje, y todo lo consecuente con lo que hemos venido haciendo para resolver nuestro problema específico.

### 3.3.1 Sobre el archivo de datos de entrenamiento.

En Trajan 3.0 los datos con que se entrena la red se almacenan en un archivo \*.sta que se crea mediante el comando “create data set”. Este archivo es básicamente una hoja electrónica en donde en las columnas se encuentran las variables de entrada y salida, y en las filas se listan los casos observados. Para que el archivo de datos de una serie de tiempo sea considerado inteligentemente como tal, en el programa existe la opción de datos “input/output”, la que especifica que la salida de la red contiene la misma variable que la entrada. Esto debe darse debido a que en una serie de tiempo las predicciones se basan en datos de una misma variable.



**Figura 3.1: Definiendo el tipo de datos para una serie temporal en Trajan3.0**



En la figura 3.2 se ilustra como se definen los datos para que sean considerados como una serie temporal: dando un **click** derecho sobre la columna previamente seleccionada aparece la opción “**input/output**” que define a los datos como de entrada y salida. Como se puede ver también existen opciones para separar cierta cantidad de datos en tipos de entrenamiento, verificación y prueba. Según se conoce, es bueno dejar cierto subconjunto para hacer verificaciones ya que esto nos dirá cuan **efecivamente** esta aprendiendo la red. La opción “**shuffle train and verify**” **entremezcla** los datos de entrenamiento con los de verificación para que la red soporte ruido, pero es importante saber que si mezclamos el orden temporal de los datos ya no servirían de nada, por lo tanto sólo se debe hacer la mezcla entre “**train and verify**” lo que no afecta el ordenamiento de los datos.

### 3.3.2 Arquitectura Multicapa.

Según se conoce, esta **arquitectura** de red es tal vez la mas popular en uso en estos días **y Trajan 3.0** puede crear redes de perceptrones multicapa de hasta 128 capas utilizando por incumplimiento (default) sumas ponderadas y la función logística en sus operaciones básicas. Inclusive las redes multicapas de perceptrones son tan utilizadas que al crear una nueva red, la que se crea por incumplimiento (default) **es** una **MLP** (Multilayer Perceptron en Trajan 3.0). Veamos cómo se crea una red en este programa:

Sencillamente seleccionamos “new network” de el menú y nos aparecerá el recuadro que aparece en la figura 3.2 llamado “create network”. Existen 2 conjuntos distintos de diseño de la red. El primero de “time series” (que nosotros utilizaremos) y el segundo “pre/post processing” que es de un uso más general.

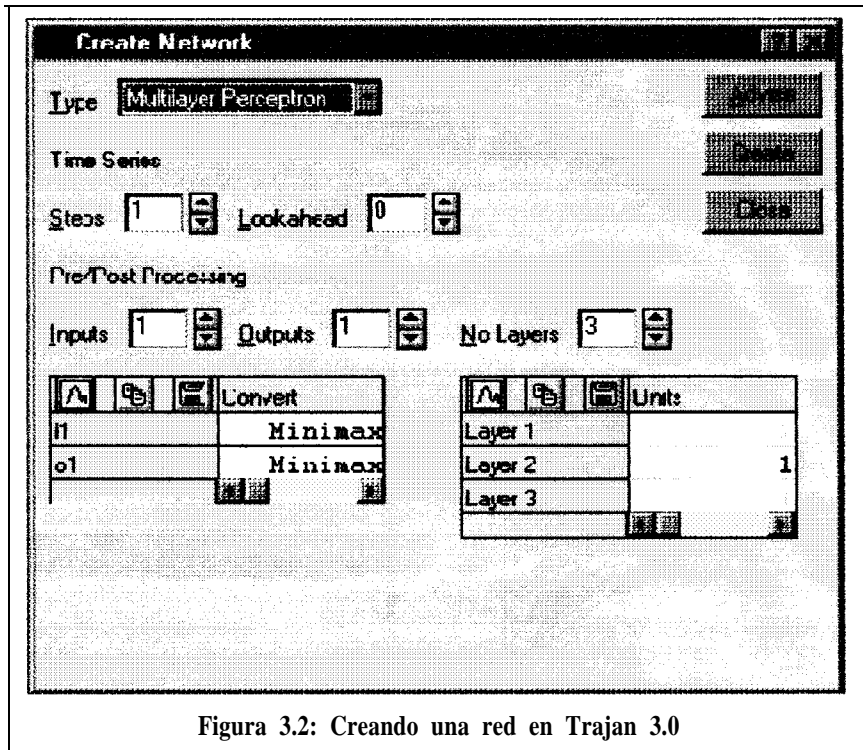


Figura 3.2: Creando una red en Trajan 3.0

Para el problema de series de tiempo seleccionamos el número de nodos de las capas de entrada y salida utilizando “steps” y “lookahead” que significan “número de pasos” y “exploración hacia delante”; algo especial que tiene este paquete es que una vez creado o cargado el archivo de datos con que se trabajará, el comando “advise” es capaz de sugerir la estructura ideal con que se debe de trabajar en MLP. “Advise” servirá de ayuda pero una vez que hayamos definido el número de nodos en las capas de entrada y salida (que requiere de un diseño propio del modelador). En realidad “advise” sugiere el número de capas ocultas y el número de nodos en cada una de ellas. Usualmente se utiliza una sólo capa oculta. El comando “create” crea automáticamente la red y una vez cargados los datos podemos empezar a entrenar la red. Una red en Trajan 3.0 tiene extensión \*.net cuya ilustración es provista por el programa una vez que esta se ha creado. En la figura 3.3 se ve como Trajan 3.0 ilustra una red neuronal con 3 capas.

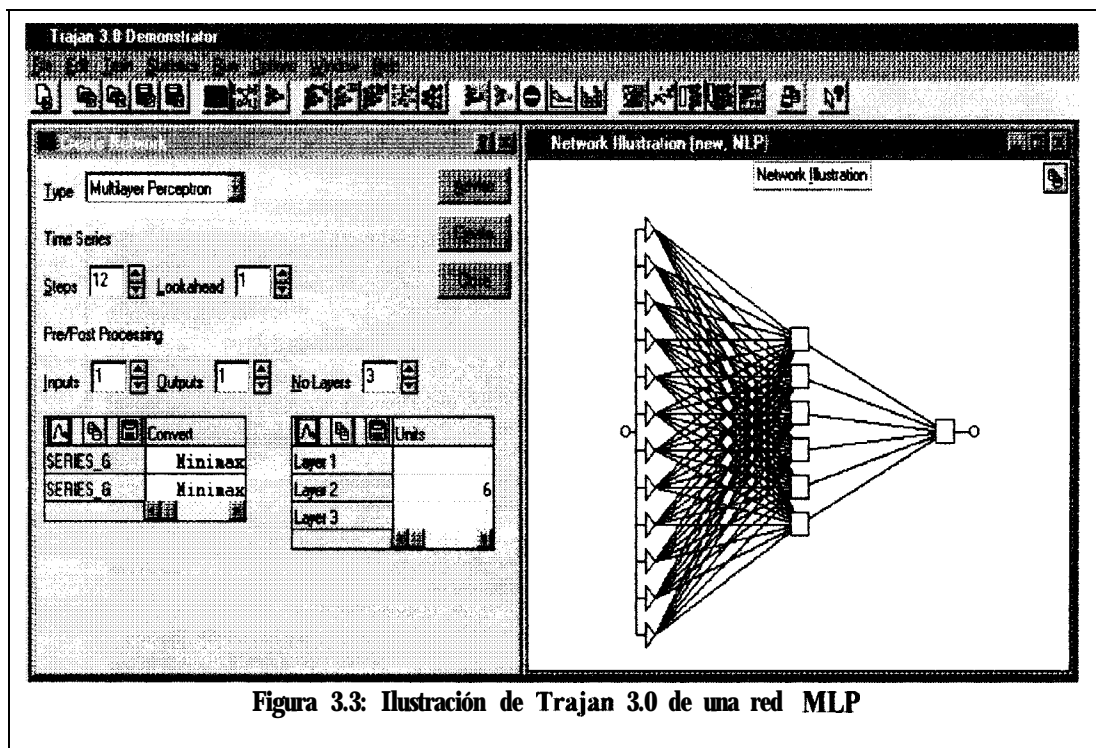


Figura 3.3: Ilustración de Trajan 3.0 de una red MLP

### 3.3.3 Con respecto al aprendizaje de la red.

Para entrenar la red de perceptrones multicapa por medio de propagación inversa (regla delta) seleccionamos del menú “train multilayer perceptrons back propagation” como lo ilustra la figura 3.4. Los parámetros que se encuentran dentro de esta opción son:

- ⊗ **Epochs:** el número de “épocas” para las cuales se correrá el algoritmo. En cada “época”, la totalidad del conjunto de entrenamiento es alimentado hacia la red y usado para ajustar los pesos y umbrales. Es un ciclo de iteraciones en realidad.
- ⊗ **Learning rate:** parámetro de velocidad de aprendizaje, el cual controla el tamaño del cambio en los pesos hechos por el algoritmo. El valor por omisión (default) es 0,6.

⊗ **Momentum:** parámetro de momento que conserva la dirección de cambio en los pesos en la medida de lo posible. Esto aumenta la velocidad de aprendizaje en las “superficies agudas” del espacio de pesos de la red. Funciona en el intervalo  $[0,1)$ .

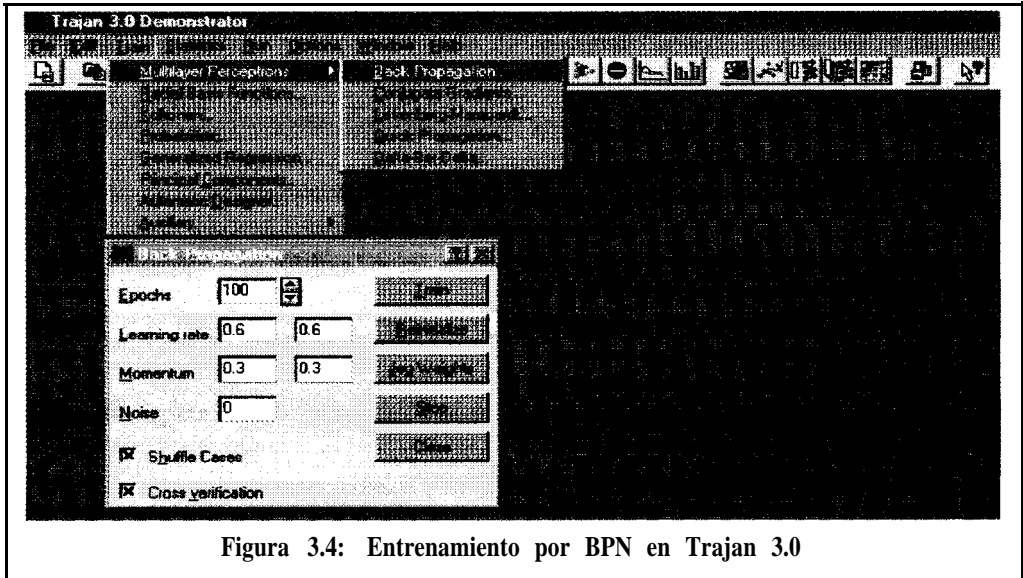


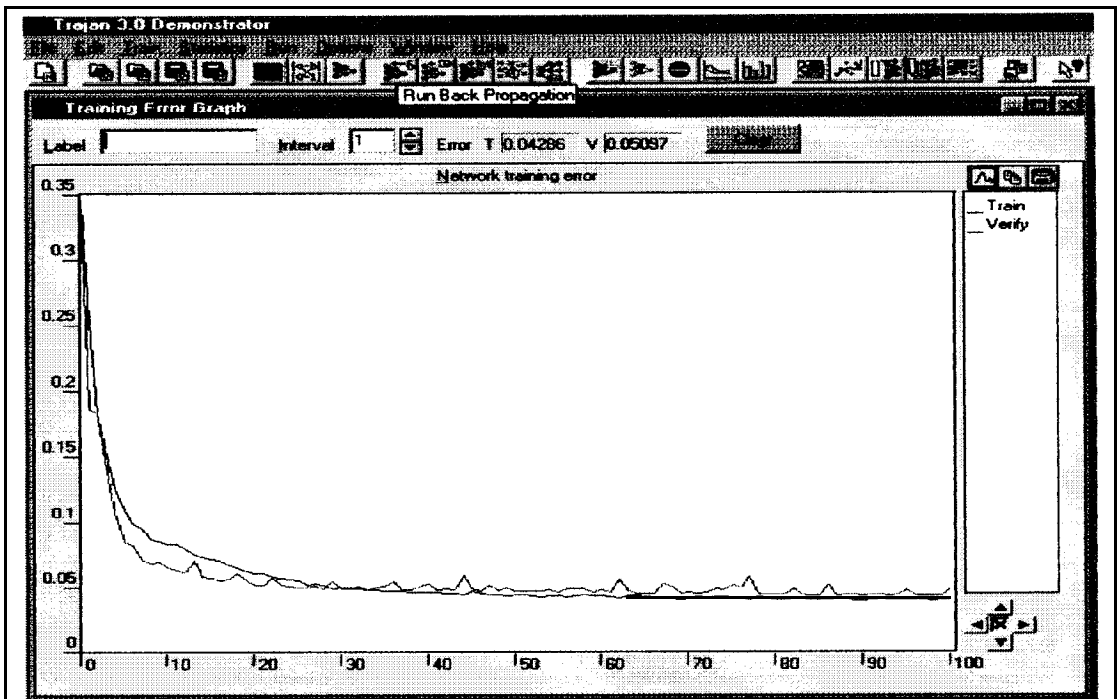
Figura 3.4: Entrenamiento por BPN en Trajan 3.0

⊗ **Shuffle cases:** esta opción sirve para que el orden en que se entrena y se verifica la red sea alterado por cada “época” de entrenamiento. Sin esta opción, el aprendizaje sufre debido a que mientras el algoritmo hace aprender casos tempranos en la “época”, luego “abandona” estos casos mientras aprende nuevos. Este “barajamiento de cartas” añade ruido al proceso de aprendizaje lo cual ayuda a escapar de mínimos locales.

⊗ **Noise:** si en este campo se pone un valor distinto de cero, entonces un ruido blanco (o error aleatorio) es añadido a cada patrón de entrada mientras dure el entrenamiento. Genera una v.a. uniforme  $U(-n, +n)$ . Sirve para ayudar a escapar de mínimos locales. Ayuda a la red a generalizar mejor y no sobreajustarse a los datos de entrenamiento.

- ☒ **Cross Verihcation:** si se desea hacer verificaciones de si la red está aprendiendo bien, se debe seleccionar esta opción para que en cada “época” se haga la respectiva verificación con el subconjunto de datos que previamente reservamos para este propósito.
- ☒ **Train:** cuando se oprime el botón de entrenar “train”, se correrá el algoritmo de aprendizaje de propagación hacia atrás por el número de “épocas” dadas o parará cuando se lo detenga con “stop”.
- ☒ **Reinitialize:** se utiliza para reinicializar la red haciendole “olvidar” todo lo aprendido al ser generados nuevos pesos aleatorios.
- ☒ **Jog Weights:** esto “remece” los pesos en un itervalo pequeñísimo con el objeto de escapar de mínimos locales y no tener que reinicializar la red.
- ☒ **Stop:** sirve para parar el entrenamiento.

Una parte fundamental en el entrenamiento es su observación desde el punto de vista de el error global por “época”. En general los paquetes computacionales calculan el RMS (Root Mean Squared Error) y los grafican. Si hemos hecho “cross verification” entonces se calculará el RMS del subconjunto de verificación también. En la figura 3.5 podemos observar como los errores por entrenamiento y verificación bajan rápidamente y empiezan a estabilizarse. En la parte superior del gráfico se muestran los últimos valores del error de entrenamiento y verificación ( $T=0.04286$ ,  $V=0.0509$ ). El gráfico está hecho dado el error versus la “época”. Una opción interesante en Trajan 3.0 con respecto al error es que se puede cambiar el intervalo en cada “época”. Errores de entrenamiento y verificación se generan cada vez que se itera en una “época”. La opción “interval” especifica cuan frecuente deben ser mostrados los errores por “época”.



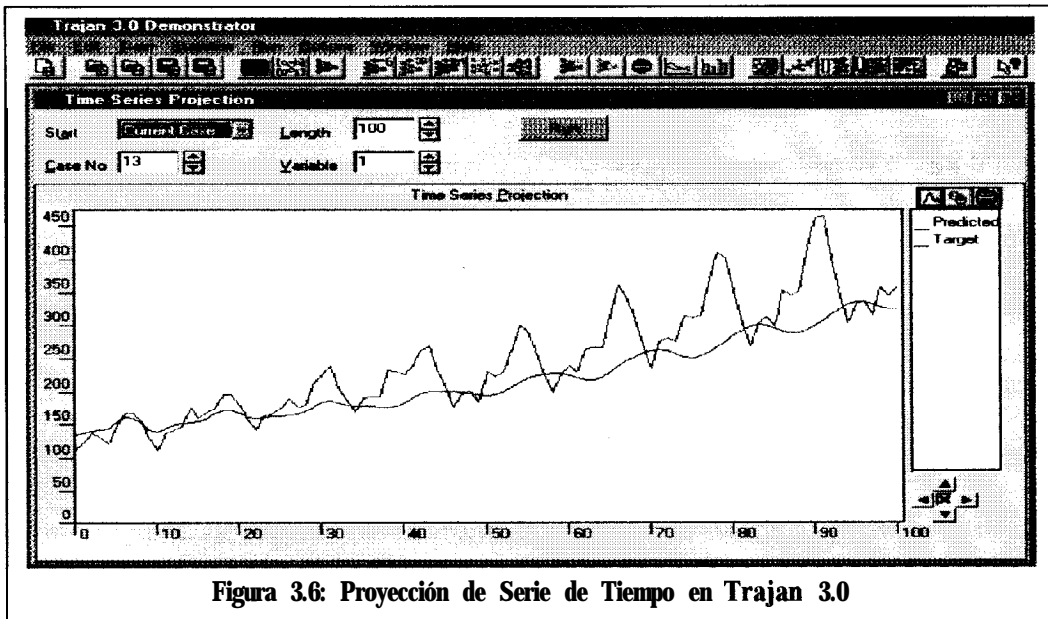
**Figura 3.5: Gráfico del error de verificación y entrenamiento al entrenar la red por Back Propagation en Trajan 3.0**

### 3.3.4 Proyección de Series de Tiempo.

En Trajan 3.0 sólo se puede hacer una sola predicción a la vez, es decir que en la capa de salida de la red habrá únicamente un nodo y en consecuencia el parámetro “Lookahead” es uno (1). Se pueden hacer predicciones que vayan más allá del espacio temporal actual, pero a medida que se avanza en el tiempo, las predicciones son menos confiables y al final como en el análisis de series de tiempo, las predicciones se desvanecen en una línea recta.

El achatamiento de las predicciones se debe a la utilización de la función sigmoide en el nodo de la capa de salida; la función tiende obviamente a limitar la salida hacia un intervalo definido por lo que para una serie de tiempo esto puede ser una desventaja debido a que puede contener datos no estacionarios (con una pendiente). Para ver el funcionamiento de la

proyección de serie de tiempo simplemente seleccionamos del menú “Run – time series” y la imagen que se muestra en la figura 3.6 aparecerá en pantalla.



**Figura 3.6: Proyección de Serie de Tiempo en Trajan 3.0**

Obviamente en la figura 3.6 vemos que la red aún no está completamente entrenada, además que la función de salida es la sigmoide para la capa de salida la que se selecciona por omisión (default) y en mejores instancias deberíamos modificarla para que nos entregue mejores predicciones. Aclaremos también que para que se ejecute la predicción se tiene que oprimir el botón “run” de la figura 3.6; además es obvio que esto se debe de hacer luego de haber cargado los datos, definido su objeto, creada la red y ser entrenada. Si la red no es entrenada antes de ejecutar esta opción, entonces se mostrará como “predicted” una línea recta reflejo de la “inexperiencia” de la red.

### 3.3.5 Observaciones.

Tengamos en cuenta que no podemos hacer demasiadas predicciones alejandonos del origen temporal debido a que una red neuronal olvida fácilmente y en el caso de las series de tiempo se alimenta mediante observaciones anteriores, por lo que llegará un momento en que se alimente únicamente de valores que ella misma ha generado. Si decidimos cambiar la función de salida de la capa de salida (que por omisión es logística) por una función lineal ( $f(x) = x$ ), mejoraremos el desempeño predictivo de la red, pero al final las predicciones se “achatarán” debido a que en la capa oculta las funciones de activación son sigmoideas. **Trajan** 3.0 tiene la opción de editar la red mediante la selección de sus capas y funciones de activación. La opción es llamada desde el menú mediante “edit – **network**” desde donde se hacen todas las modificaciones del caso.



## CAPITULO IV

### 4 Aplicación y Análisis de los Modelos Empleando Datos de una Serie Temporal.

#### 4.1 *Introducción.*

En este capítulo vamos a hacer predicciones de dos series de tiempo con redes neuronales de Propagación Hacia Atrás. La primera parte constará de las predicciones del número anual de manchas solares del programa **Bpn** de Karsten **Kutza**. Con esto evaluaremos las predicciones comparandolas con los datos originales y con una predicción hecha por análisis de series de tiempo. La segunda parte constará del diseño propio de una BPN en Trajan 3.0 que hace predicciones de el número mensual de pasajeros de aerolíneas en los Estados Unidos. Mostraremos los resultados más destacados con respecto al diseño, error, y predicción. Por último haremos la respectiva comparación de las predicciones hechas por la red neuronal en contraste con aquellas hechas por un modelo **SARIMA**.

## 4.2 Predicciones *del* número anual de manchas *solares*.

El programa *Bpn* de Karsten Kutza tiene cargado dentro de su programación un conjunto de 280 datos del número anual de manchas solares (de Wölfer) desde el año 1700 hasta 1979. Estos datos se encuentran ajustados en el intervalo  $(0, 1)$  para que no haya dificultad alguna con respecto a la función de activación de la capa de salida de la BPN. Según Box y Jenkins en su libro *Time Series Analysis* dicen que esta serie ha sido objeto de mucha investigación por su peculiar comportamiento. Box y Jenkins afirman que empíricamente esta serie puede ser modelada por un proceso AR(2) o AR(3). La diferencia es que Box y Jenkins utilizan para la modelización datos desde 1770 hasta 1869, que es un subconjunto de la serie que nosotros tenemos. Esto marca una diferencia según los resultados de nuestra propia modelización de esta serie de tiempo ya que en realidad nos encontramos frente a una serie estacional. Según los datos y las autocorrelaciones, la serie es estacional con periodo de 11 años.

### 4.2.1 Modelización de la serie por un proceso estacional.

En la figura 4.1 se puede observar el gráfico del conjunto de datos. Procederemos a modelar la serie con un subconjunto de 260 datos y reservaremos los años desde 1960 hasta 1979 para hacer predicciones y compararlas con las que arroje la red neuronal. Empezamos con el gráfico de las autocorrelaciones (fig. 4.2) y autocorrelaciones parciales (fig. 4.3). Usaremos el paquete estadístico Systat 7.0 para el análisis de series de tiempo.



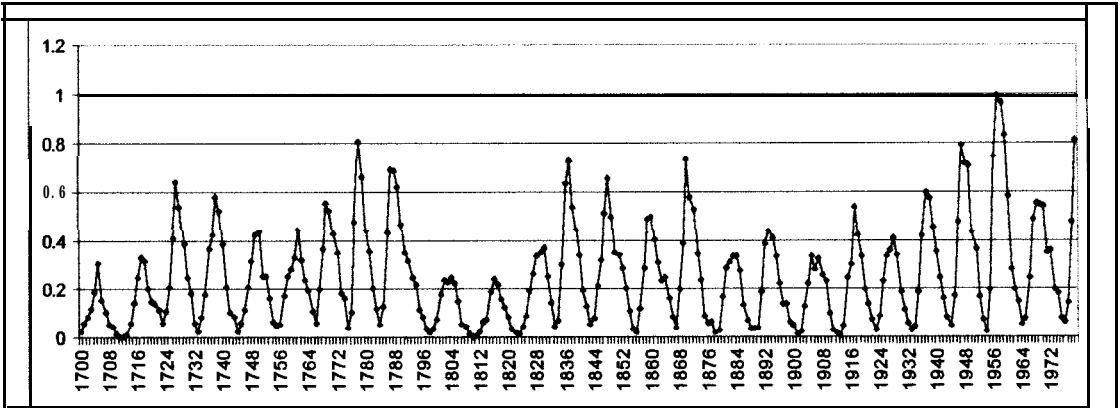


Figura 4.1: Serie ajustada del número anual de manchas solares de Wölfer

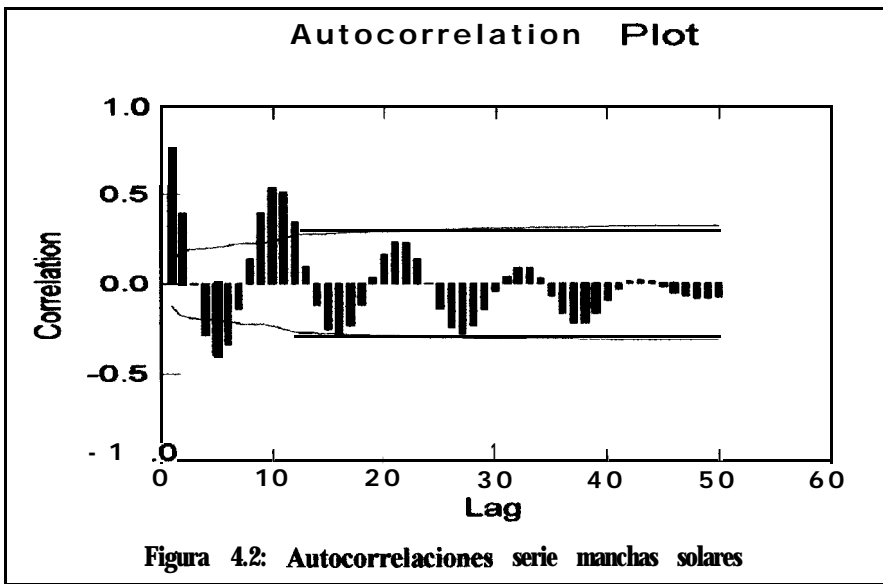


Figura 4.2: Autocorrelaciones serie manchas solares

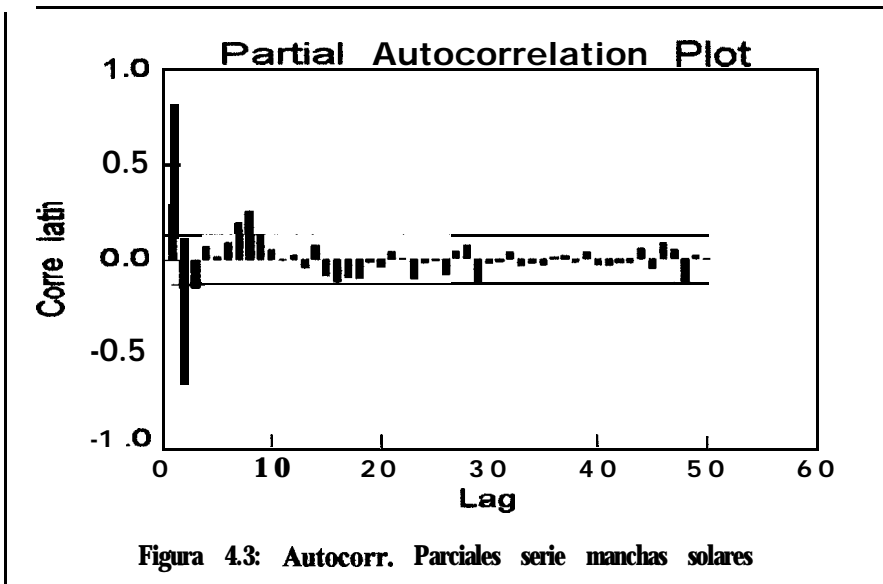
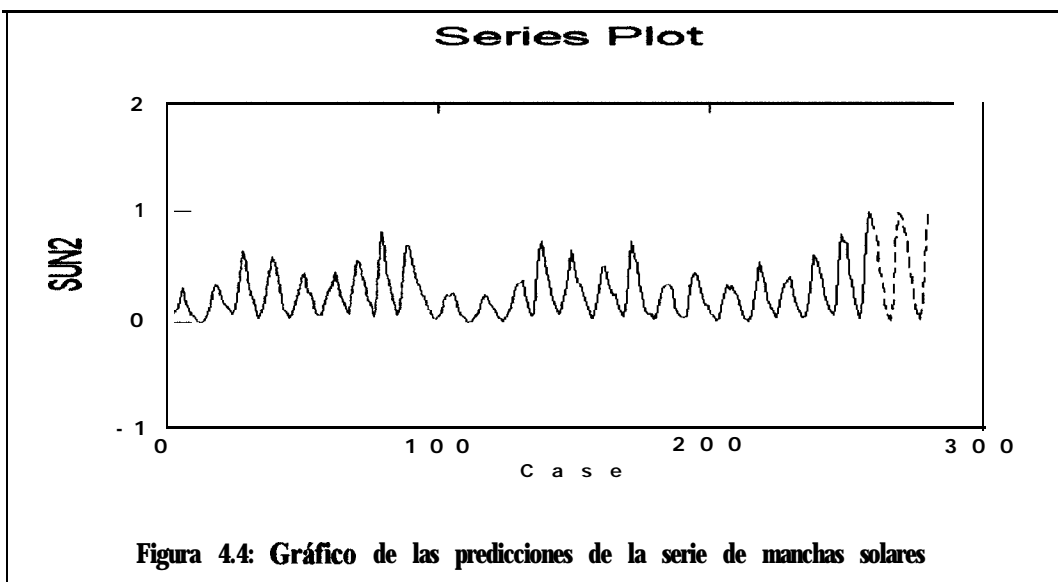


Figura 4.3: Autocorr. Parciales serie manchas solares

En el gráfico de las autocorrelaciones observamos que estas se acumulan alrededor de un periodo 11 (lag 11). Es decir, que podemos ver que cada 11 unidades temporales las autocorrelaciones se acumulan en un mismo costado de las bandas. Las dos primeras acumulaciones (de un mismo costado) resultan ser significativas. Por el lado de las autocorrelaciones parciales podemos advertir que las dos primeras son significativas, aunque la tercera sobresalga minúsculamente. Raramente la séptima y la octava sobresalen de las bandas significativamente, pero después de tomar en cuenta el criterio de los 5 parámetros y hacer varias pruebas con procesos de mayor orden, se sabe que estos órdenes quedan descartados del modelo porque no ajustan correctamente.

El modelo final al que llegamos es un  $SARIMA(2,0,0) \times (0,0,2)_{11}$ . En el cuadro 4.1 podemos observar la corrida de Systat 7.0 en el que se encuentran los valores de los parámetros del SARIMA siendo estos validos por su no nulidad estadística (el intervalo de confianza no incluye el cero); se encuentran también las 20 predicciones que necesitamos para hacer la comparación. En la figura 4.4 podemos observar el gráfico de las predicciones (en rojo punteado) hechas por Systat 7.0. Las predicciones tienen un intervalo de confianza del 95% que obviamente se agranda a medida que el tiempo transcurre.



**Figura 4.4: Gráfico de las predicciones de la serie de manchas solares**

Iteration	Sum of Squares	Parameter values			
0	.2416964D+02	.100	.100	.100	.100
1	.1111490D+02	.229	.204	-.015	.004
2	.9883390D+01	.307	.230	.025	.055
3	.3334896D+01	1.400	-.520	.166	.249
4	.3051706D+01	1.584	-.728	.134	.211
5	.2932143D+01	1.628	-.728	.244	.069
6	.2364633D+01	1.452	-.595	-.010	.062
7	.2090763D+01	1.393	-.536	-.229	-.047
8	.2049455D+01	1.358	-.515	-.242	-.239
9	.2031750D+01	1.368	-.520	-.299	-.200
10	.2029350D+01	1.372	-.524	-.334	-.203
11	.2029236D+01	1.373	-.526	-.334	-.209
12	.2029219D+01	1.374	-.527	-.334	-.210
13	.2029216D+01	1.374	-.528	-.334	-.209
14	.2029215D+01	1.374	-.528	-.334	-.209
15	.2029215D+01	1.374	-.528	-.334	-.209
16	.2029215D+01	1.374	-.528	-.334	-.209
17	.2029215D+01	1.374	-.528	-.334	-.209
18	.2029215D+01	1.374	-.528	-.334	-.209
19	.2029215D+01	1.374	-.528	-.334	-.209
20	.2029215D+01	1.374	-.528	-.334	-.209
21	.2029215D+01	1.374	-.528	-.334	-.209
22	.2029215D+01	1.374	-.528	-.334	-.209

Final value of MSE is 0.009

Index	Type	Estimate	A.S.E.	Lower	<95%>	Upper
1	AR	1.374	0.059	1.257		1.491
2	AR	-0.528	0.058	-0.642		-0.414
3	SMA	-0.334	0.076	-0.484		-0.185
4	SMA	-0.209	0.065	-0.337		-0.082

Asymptotic correlation matrix of parameters

	1	2	3	4
1	1.000			
2	-0.888	1.000		
3	0.335	-0.198	1.000	
4	0.158	-0.014	0.253	1.000

Series truncated at first missing value.

Period	Forecast Values		
	Lower95	Forecast	Upper95
261.	0.594	0.776	0.958
262.	0.166	0.474	0.783
263.	-0.021	0.374	0.770
264.	-0.286	0.161	0.608
265.	-0.411	0.062	0.535
266.	-0.474	0.010	0.494
267.	-0.301	0.187	0.674
268.	0.243	0.731	1.220
269.	0.499	0.988	1.476
270.	0.474	0.962	1.451
271.	0.340	0.829	1.318
272.	0.235	0.775	1.315
273.	-0.175	0.474	1.123
274.	-0.374	0.375	1.123
275.	-0.654	0.162	0.977
276.	-0.789	0.063	0.914
277.	-0.857	0.011	0.879
278.	-0.687	0.187	1.061
219.	-0.144	0.732	1.607
280.	0.112	0.988	1.863

Cuadro 4.1: **Corrida** del **modelo SARIMA(2,0,0)(0,0,2)<sub>11</sub>** en **Systat 7.0**

Este modelo que presentamos es, por supuesto, tentativo debido a la naturaleza de los datos y a los amplios estudios sobre transformaciones para la serie para que ajuste de manera correcta. No entraremos en más detalles sobre el análisis estocástico de esta serie.

Teniendo ya el conjunto de predicciones hechas en Systat 7.0 podemos proceder a hacer lo mismo desde el punto de vista de la inteligencia artificial.

#### 4.2.2 Predicción de la serie de Wölfer mediante una BPN.

Como ya habíamos mencionado, el programa de Kutza no permite interactividad alguna, por lo que simplemente nos queda hacerlo correr. En el cuadro 4.2 mostramos los resultados de la corrida del programa.

```
NMSE is 0.107 on Training Set and 0.170 on Test Set - stopping Training
and restoring Weights . . .
NMSE is 0.141 on Training Set and 0.138 on Test Set
```

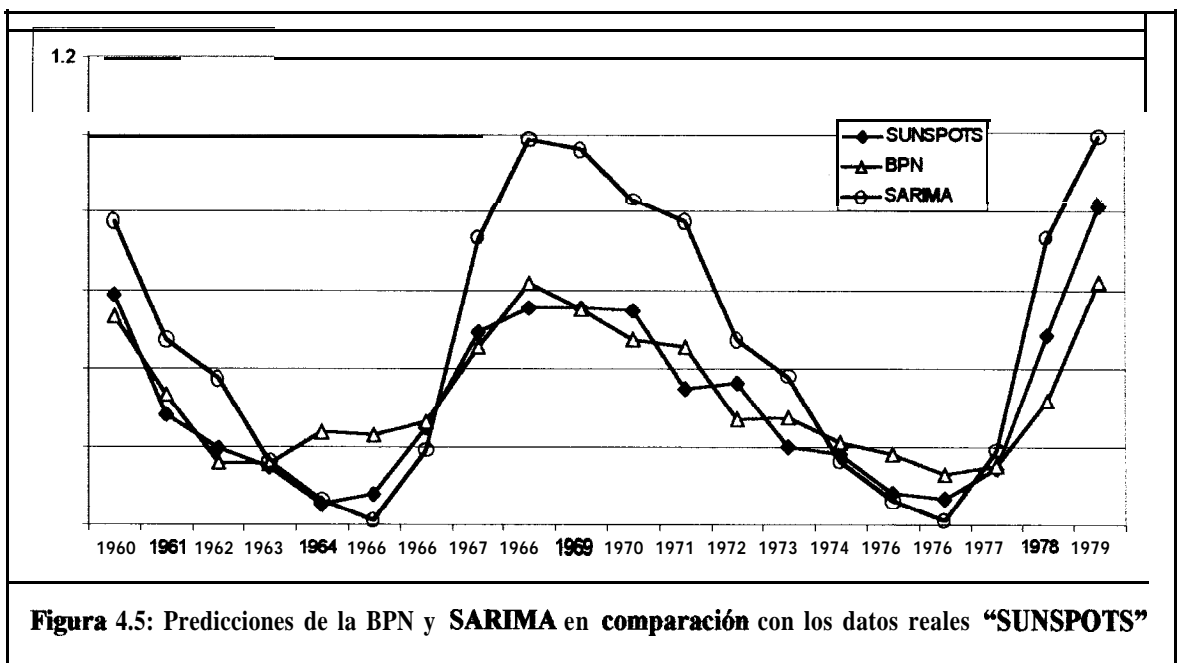
Year	Sunspots	Prediction
1960	0.572	0.532
1961	0.327	0.334
1962	0.258	0.158
1963	0.217	0.156
1964	0.143	0.236
1965	0.164	0.230
1966	0.298	0.263
1967	0.495	0.454
1968	0.545	0.615
1969	0.544	0.550
1970	0.540	0.474
1971	0.380	0.455
1972	0.390	0.270
1973	0.260	0.275
1974	0.245	0.211
1975	0.165	0.181
1976	0.153	0.128
1977	0.215	0.151
1978	0.489	0.316
1979	0.754	0.622

**Cuadro 4.2: Corrida de BPN para manchas solares**

Se ha truncado parte del texto de la corrida que se refiere al progreso del aprendizaje al reducirse los errores por entrenamiento y verificación tal y como lo hace **Trajan 3.0**. Esto lo hacemos porque este archivo que se crea al correr el programa no entraría en el cuadro 4.2 y por el momento lo interesante son las predicciones. Al **final** se anexa la corrida completa del programa.

### 4.2.3 Comparación de las predicciones de manchas solares.

Vamos a mostrar graficamente las diferencias bien marcadas que hemos encontrado y luego sacaremos una media cuadrática del error por cada conjunto de predicciones para ver cual de las dos resultó ser más efectiva.



Observese en la figura 4.5 cuan próximas a los verdaderos datos son las predicciones hechas por la red de propagación hacia atrás en contraste con el modelo SARIMA propuesto, que tiene un pico muy alto. Ya tenemos el primer resultado convincente con respecto a la

capacidad predictiva de una BPN, ahora procedamos a contabilizar el error obteniendo una varianza estimada de la diferencia de las predicciones con respecto a los datos originales:

<b>Fuente de Predicción</b>	<b>Media cuadrática del error (estimada)</b>
SARIMA	0.04900
BPN	0.00942

**Tabla 4.1: Media cuadrática del error de Predicción de manchas solares**

Podemos observar que el error cuadrático promedio de las predicciones del proceso SARIMA es casi el doble que el de las predicciones de la BPN. En la tabla 4.2 se resumen las predicciones que se han obtenido.

<b>AÑO</b>	<b>SUNSPOTS</b>	<b>BPN</b>	<b>SARIMA</b>
1960	<b>0.587</b>	0.532	0.776
1961	0.282	0.334	0.474
1962	0.196	0.158	0.374
1963	0.146	0.156	0.161
<b>1964</b>	0.053	0.236	0.062
1965	0.079	0.230	0.010
1966	0.246	0.263	0.187
1967	0.491	0.454	0.731
<b>1968</b>	0.554	0.615	0.988
<b>1969</b>	0.552	0.550	0.962
1970	0.546	0.474	0.829
1971	0.348	0.455	0.775
1972	0.360	0.270	0.474
1973	0.199	0.275	0.375
1974	0.180	0.211	0.162
1975	0.081	0.181	0.063
1976	0.066	0.128	0.011
1977	0.143	0.151	0.187
1978	0.484	0.316	0.732
1979	0.813	0.622	0.988

**Tabla 4.2: Predicciones de ta manchas solares**





### 4.3 Predicción del número mensual de pasajeros de aerolíneas en EEUU.

Esta serie de tiempo es otra de las ya muy conocidas por su utilización en el texto de Box y Jenkins. La serie es el número mensual (en miles) de pasajeros internacionales de aerolíneas en lo Estados Unidos, que es una serie estacional cada 12 meses. Tenemos datos desde el mesde enero de 1949 hasta diciembre de 1960 que hacen un total de 144 observaciones. Reservaremos las ultimas 12 observaciones para hacer la respectiva comparación, por lo tanto tendremos 132 datos para trabajar netamente.

El software Trajan 3.0 remueve las primeras 12 observaciones de el conjunto primario de datos para pre-procesamiento, por lo que las predicciones empezaran a responder desde el décimo tercer dato, es decir, desde enero de 1950. Las bservaciones son removidas sólo después de haber definido el tipo de datos de entrada y salida, y de haber diseñado la red, por lo que es evidente que si la entrada tuviera 10 nodos se removerían las 10 primeras observaciones.

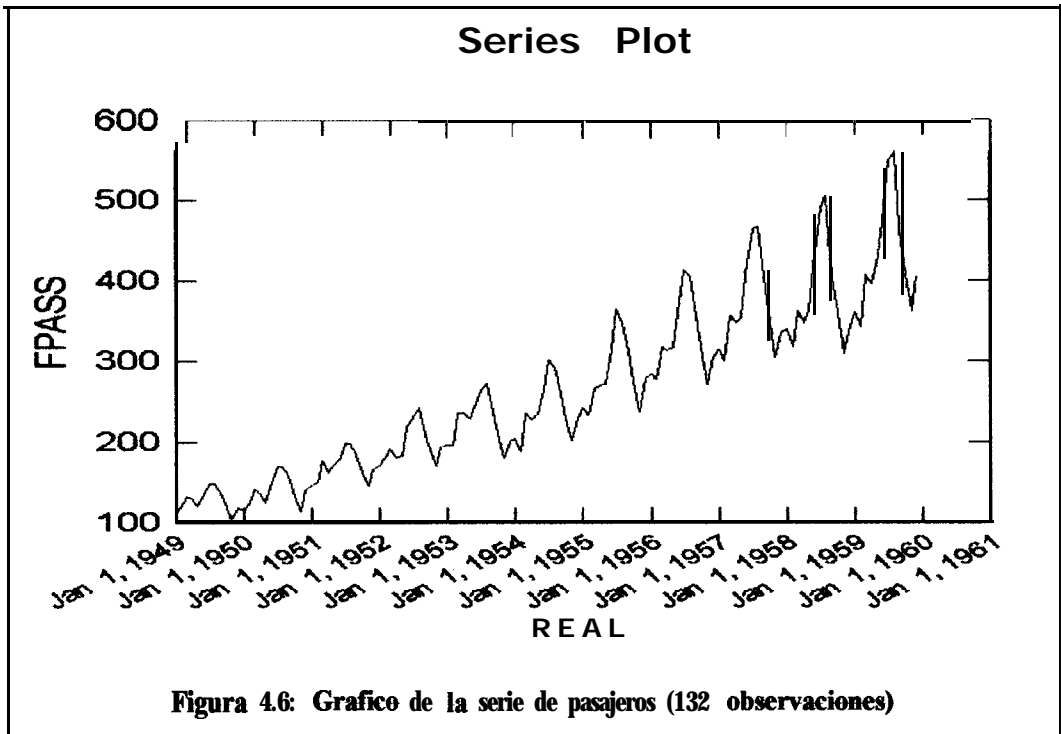
#### 4.3.1 Predicción mediante un modelo SARIMA.

En este punto no vamos a repetir el análisis que se hace en el texto de Box y Jenkins, en donde se llega a la conclusión de que el modelo que mejor ajusta es un SARIMA(0,1,1)(0,1,1)<sub>12</sub>; lo que hemos hecho es una simulación utilizando el modelo dado. En otras palabras hemos generado todos los ruidos blancos  $u_t$  que el SARIMA necesita para ajustar mediante el modelo obtenido:

$$z_t = z_{t-1} + z_{t-12} - z_{t-13} + u_t - 0.4u_{t-1} - 0.6u_{t-12} + 0.24u_{t-13} \quad (4.1)$$

en donde  $z_t = \ln(x_t)$ , que es un ajuste hecho a los datos originales  $x_t$ , y los retardos de  $z$  son producto de las diferenciaciones estacional y no estacional efectuadas (de lo contrario  $z$  sólo dependería de los ruidos blancos). Las diferenciaciones también afectan a los ruidos blancos.

En la figura 4.6 podemos observar el gráfico de la serie de datos.



**Figura 4.6: Grafico de la serie de pasajeros (132 observaciones)**

El objeto de simular la serie es simplemente obtener un conjunto de predicciones sobre todo el conjunto de datos. En otras palabras, no hemos hecho la corrida del modelo en **Systat** 7.0 sino que hemos generado las predicciones “manualmente” en una hoja electrónica mediante simulaciones. Como vemos en la ecuación 4.1 las predicciones que obtengamos serán el logaritmo natural de las observaciones por lo que, en este caso, habría que aplicarles una transformación inversa (**exponencial**) para llegar a las estimaciones adecuadas.

#### 4.3.2 Diseño y uso de una BPN para predecir el número mensual de pasajeros.

Lo primero que tenemos que hacer es cargar el archivo donde se encuentra la serie y definir la única columna como “input/output”, luego separamos 66 -observaciones para la verificación y mezclamos los subconjuntos con “shuffle train and verify”. En la figura 4.7 podemos observar cómo hacemos esto en Trajan 3.0.

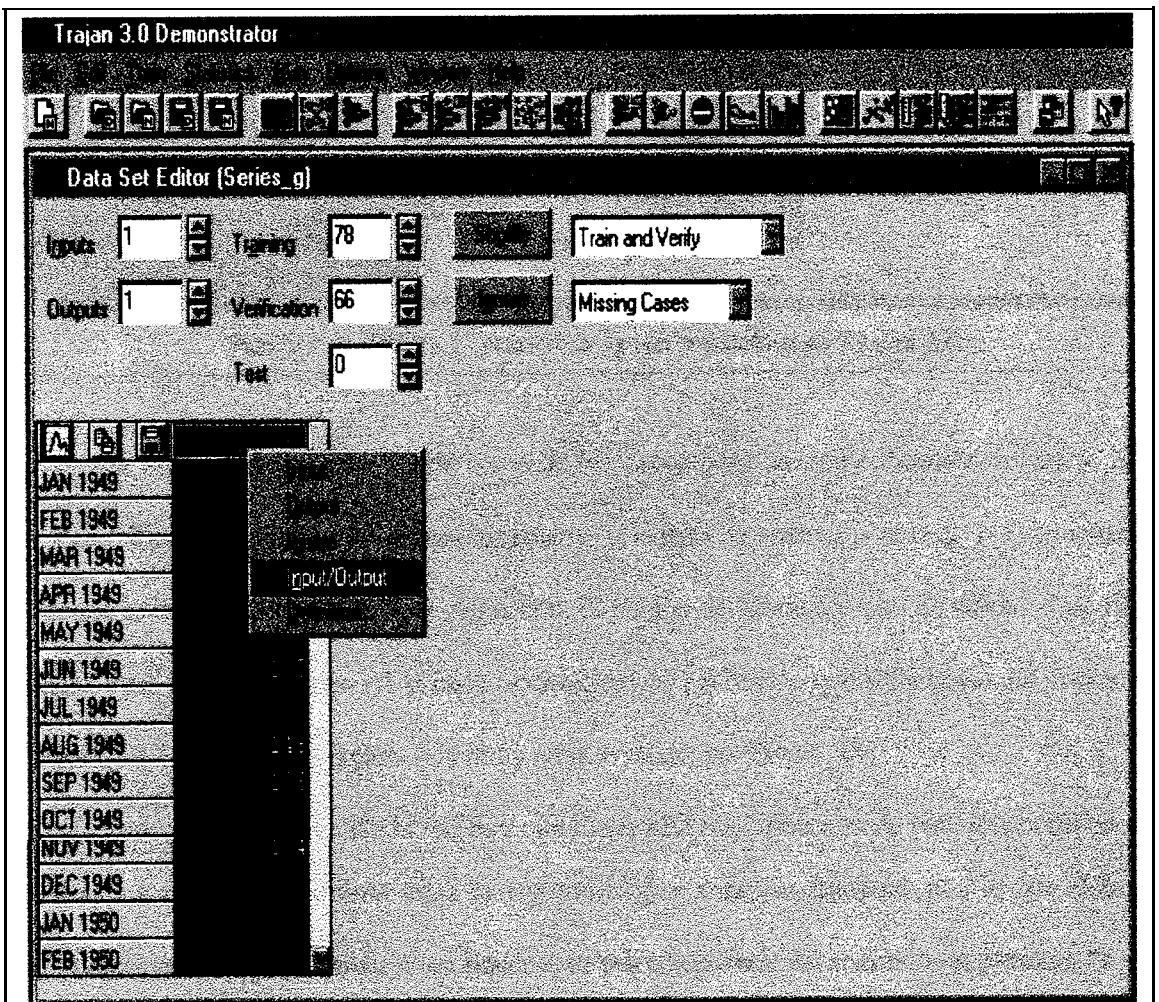


Figura 4.7: Definiendo los datos como “input/output” en Trajan 3.0

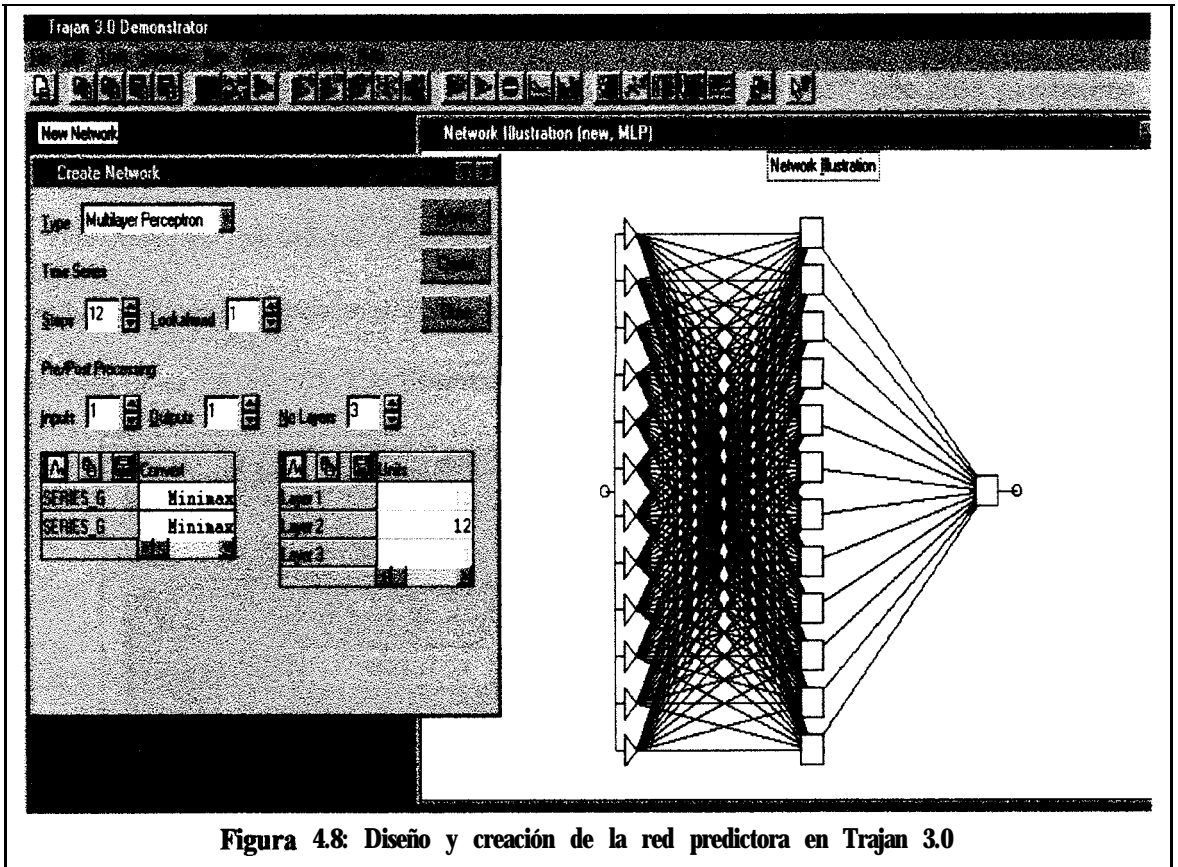
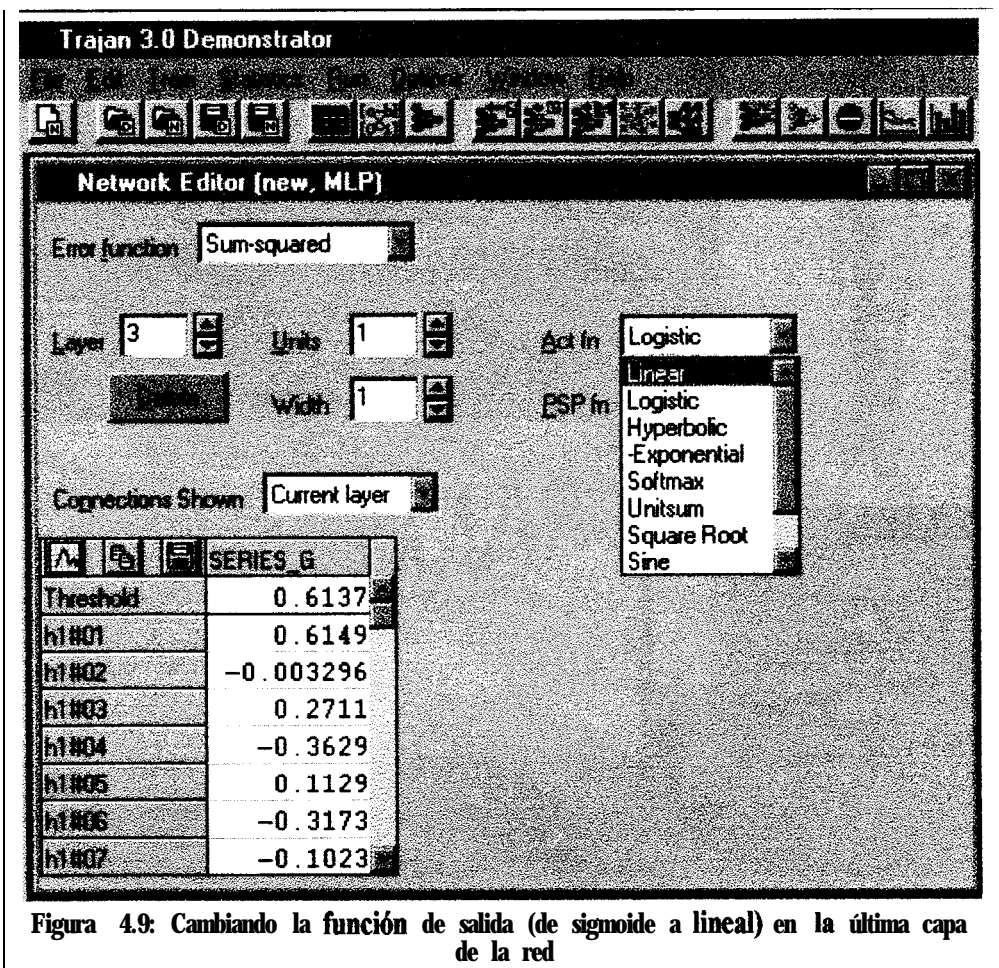


Figura 4.8: Diseño y creación de la red predictora en Trajan 3.0

Es bueno separar una buena parte de los datos para hacer verificación mientras la red aprende. Trajan 3.0 hace esto automáticamente mientras se entrena a la red. Lo siguiente que hacemos es diseñar una red de tres capas con 12 nodos de entrada y uno de salida; el comando “advise” provee de 6 nodos en la capa oculta pero es preferible que utilicemos 12 nodos. En la figura 4.8 vemos como creamos la red y su ilustración en Trajan 3.0.

Luego de haber creado la red hay que hacer una modificación de vital importancia en el diseño. La salida en la última capa es sigmoidea por omisión (default) por lo que debemos modificarla para que sea lineal. Para hacer esto seleccionamos del menú “edit network” y en la tercera capa “layer 3” modificamos de “sigmoid” a “linear”. En la figura 4.9 vemos como editamos la tercera capa en el paquete neuronal. Esta nueva salida prevendrá del “achataamiento” temprano de las predicciones.

Una vez editada la red procedemos a correr el entrenamiento por propagación hacia atrás. En la figura 4.10 mostramos la corrida de 10000 “epochs” utilizando parámetros de velocidad de aprendizaje y de momento igual a 0.02 y 0.008 respectivamente. Los parámetros de velocidad de aprendizaje y de momento que hemos utilizado son producto de varios intentos por que las predicciones sean suficientemente buenas; en otras palabras estos valores son fruto de varias corridas con distintas cantidades en los parámetros.



En la figura 4.10 también podemos observar los últimos valores de los errores por entrenamiento y verificación en el recuadro de “training error graph”. Los valores son de 0.0244 para el error de entrenamiento y 0.03732 para el error de verificación, que son bastante aceptables. Es claro que no hemos corrido las 10000 iteraciones de una sola vez (la última

corrida se observa que fueron 4000), esto es sugerible hacerlo de poco en poco para no sobreentrenar la red.

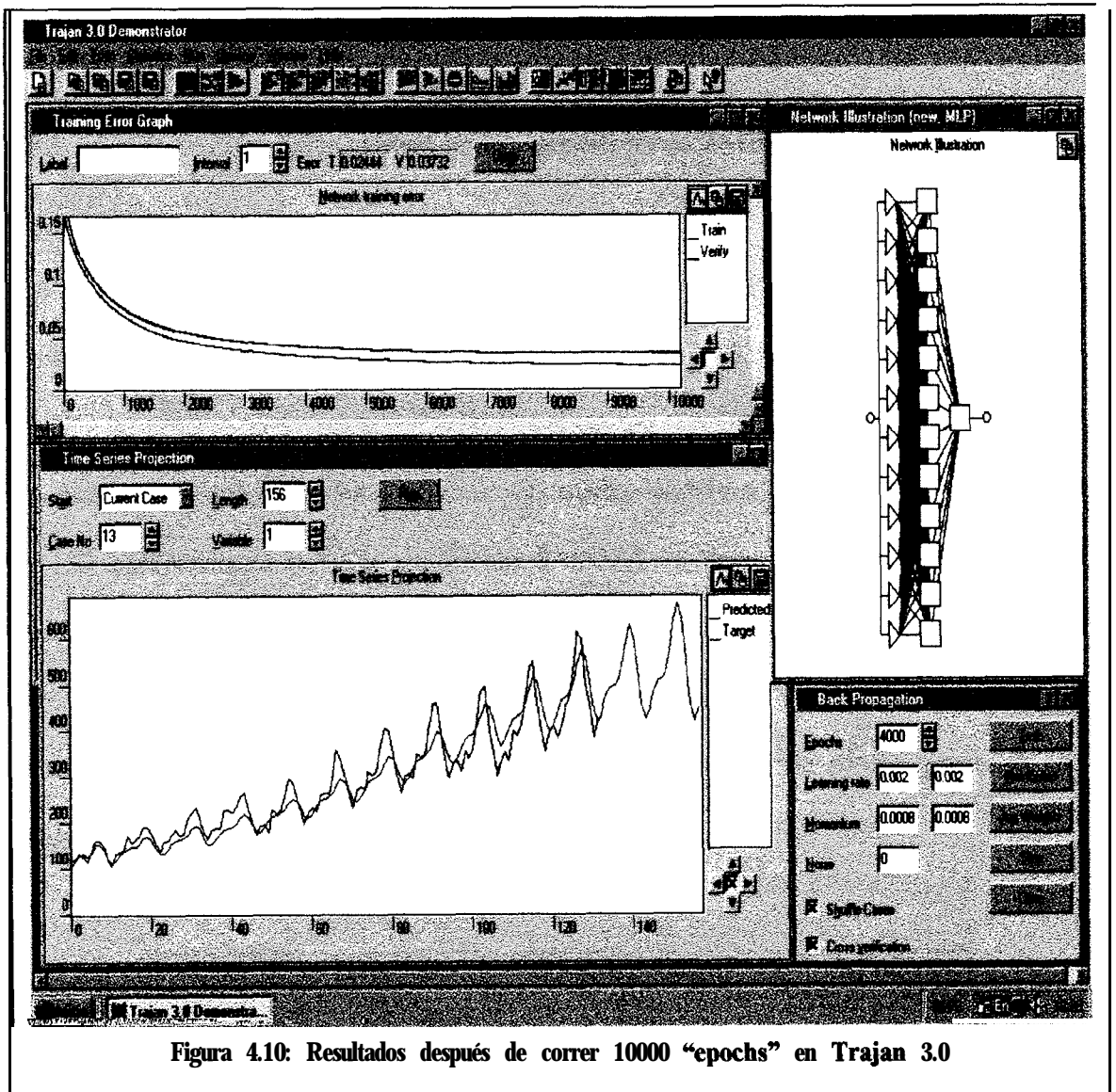


Figura 4.10: Resultados después de correr 10000 “epochs” en Trajan 3.0

También es sugerible ir haciendo verificaciones con las predicciones utilizando “run – time series” como lo muestra la figura 4.10, en la que se aprecian los últimos resultados como muy buenos. Se han corrido las predicciones desde la observación #13 hasta la #144 para el conjunto de datos reales y se han predicho 24 valores fuera de este conjunto (la línea de color

azul de 0 a 156); los 12 primeros datos no aparecen debido al pre-procesamiento de la red. Los valores predcidos fuera del conjunto de datos reales se observan como muy buenos ya que poseen la misma tendencia, pero no tenemos ningún criterio fuera de este para compararlos.

### 4.3.3 Comparación entre valores reales, predicciones SARIMA y predicciones BPN.

Los datos reales los hemos llamado “passengers” como originalmente se los llamó y de estos se puede observar los 12 últimos valores en la tabla 4.3, en donde se pueden observar también las predicciones hechas por la BPN y el proceso SARIMA en cuestión. Los resultados son muy satisfactorios aunque al obtener una media cuadrática del error de predicción sobre las 132 observaciones entre la simulación SARIMA vs Passengers, y BPN vs Passengers podemos ver que esta medida de error fue un poco mayor para BPN que para el SARIMA. Pero la diferencia es en realidad minúscula y en las figuras 4.11 y 4.12 vemos los datos casi montados los unos sobre los otros, lo que nos da una idea de mucha precisión en las predicciones.

<b>AÑO</b>	<b>SARIMA</b>	<b>PASSENGERS</b>	<b>BPN</b>
ene-60	433.957	417	398.700
feb-60	397.930	391	400.800
mar-60	464.928	419	405.600
abr-60	411.394	461	426.900
may-60	488.400	472	482.900
jun-60	531.714	535	523.900
jul-60	624.021	622	607.600
ago-60	632.262	606	611.800
sep-60	503.483	<b>508</b>	511.700
oct-60	447.593	461	434.200
nov-60	410.439	390	404.800
<b>dic-60</b>	438.123	432	401.900

**Tabla 4.3: Predicciones del número de pasajeros**

Fuente de Predicción	Media cuadrática del error (estimada)
SARIMA	13.1
BPN	15.8

Tabla 4.4: MCE de las predicciones sobre los 132 casos

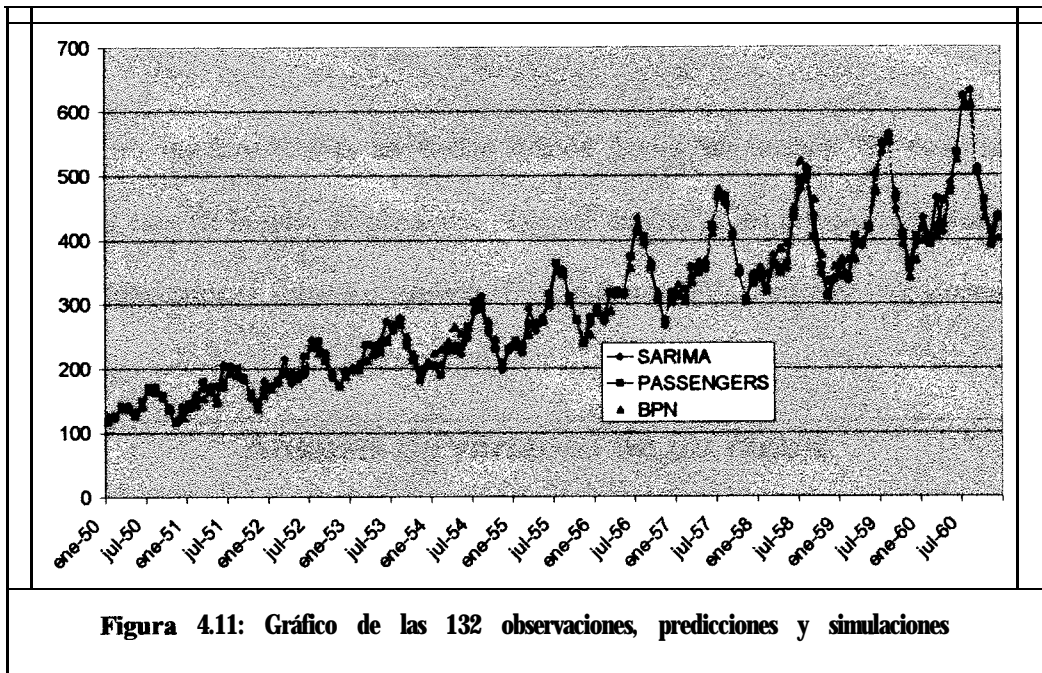


Figura 4.11: Gráfico de las 132 observaciones, predicciones y simulaciones

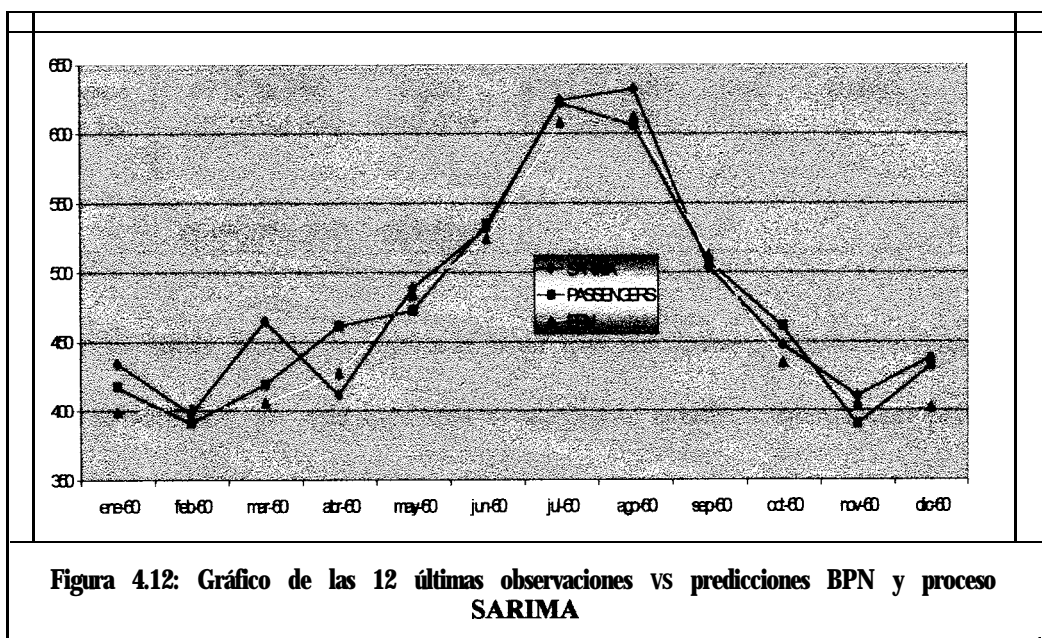


Figura 4.12: Gráfico de las 12 últimas observaciones vs predicciones BPN y proceso SARIMA



Nunca dijimos que una red neuronal sería mejor predictora que un proceso estocástico producto de un análisis de series de tiempo, pero en realidad la diferencia es tan pequeña que aunque así lo hubiésemos afirmado, este, no sería un buen contraejemplo.

#### 4.4 Observaciones Finales.

Para no quedarnos con las dudas hemos hecho un análisis de **varianza** con los resultados del caso de la serie de pasajeros. Hemos analizado los errores cuadráticos de cada una de las observaciones sobre un experimento unifactorial de 2 niveles. Los resultados son los siguientes:

<b>Análisis de varianza de un factor</b>						
<i>RESUMEN</i>						
<i>Grupos</i>	<i>Cuenta</i>	<i>Suma</i>	<i>Promedio</i>	<i>Varianza</i>		
BPN	131	32850.07	250.7638931	193992.32		
SARIMA	131	21739.8942%	165.9533915	121584.33		
<i>ANALISIS DE VARIANZA</i>						
<i>Origen de las variaciones</i>	<i>Suma de cuadrados</i>	<i>Grados de libertad</i>	<i>Promedio de los cuadrados</i>	<i>F</i>	<i>Probabilidad</i>	<i>Valor crítico para F</i>
Entre grupos	471129.78%	1.000	471129.78%	2.986	0.085	6.733
Dentro de los grupos	41024964.521	260.000	157788.325			
Total	41496094.309	261.000				

Por lo tanto, siendo la hipótesis nula que los errores cuadráticos son iguales en promedio, **concluimos** que esta hipótesis es aceptable estadísticamente con probabilidad de equivocarnos de 0.085.

Por otro lado ya hemos podido verificar que una red neuronal si puede llevar la tarea de predecir una serie de tiempo mediante un buen diseño. Existen otros algoritmos de

aprendizaje con los cuales se puede hacer predicciones de series temporales, entre ellos “Lebenberg-Marquardt”, “Conjugate Gradient Descent”; y otro tipo de redes como “Generalized Regression Neural Networks”. Sin embargo las BPN con su ley de aprendizaje GDR es la forma mas sencilla de resolver este problema usando redes neuronales.

Algo curioso es que según la Ayuda de Trajan 3.0, cuando se explica la resolución del problema de predicción de una serie de tiempo, se mencionan los algoritmos “Lebenberg-Marquardt” y “Conjugate Gradient Descent”, pero no se menciona la propagación inversa que nosotros hemos utilizado. La experiencia que sacamos con estos algoritmos alternativos fue que caen rápidamente en mínimos locales y **frecuemente** hay que reinicializar la red ya que no existen múltiples parámetros que controlen la salida de estos mínimos. Por lo tanto la calidad de las predicciones dependen mas de la aleatoriedad que del propio diseño de la red de perceptrones multicapa (cuya estructura es exactamente la misma para cualesquiera de estos algoritmos de entrenamiento). Lo aconsejable entonces sería utilizar una BPN aunque con este ejemplo no hayamos demostrado nada ni generalizado (esa no es la intención) el problema. Lo intrigante es el por qué no se menciona BPN en este caso? La respuesta tal vez la tenga el número de iteraciones: mientras que en BPN nos tomó 10000 “epochs” llegar a un nivel aceptable de error, a la misma red que aprenda con “Lebenberg-Marquardt” o “Conjugate Gradient Descent” le podría tomar menos de 100 (y esta cantidad es mucho mas didáctica).

## **Conclusiones y Recomendaciones.**

De lo antes visto se puede destacar algunos puntos y hacer cierto tipo de recomendaciones para aquellos que deseen predecir datos de series temporales por medio de redes neuronales. Vamos a tratar de ser lo mas practicos y pedagógicos posibles (como lo hemos venido intentando a lo largo de esta tesis) como para que el modelador se sienta comfortable al encontrarse con este problema. Además, vamos a redundar en contestar la pregunta ...¿se puede predecir con redes neuronales?

Pues la respuesta es un SI rotundo. La red funciona como un primitivo cerebro, que aprende a advertir cual será el evento que debe corresponder en el futuro con respecto a lo que se vio en el pasado y el presente por medio de ejemplos; sin teoría, empíricamente. Una red neuronal es como un bebe recién nacido: lo que aprende es lo que capta por medio de situaciones concretas, que luego imita; nadie le explica al bebe como hablar, el bebe simplemente percibe, procesa y repite los sonidos. Así mismo hemos visto como una red

“percibe” un conjunto de valores correspondientes a eventos pasados, presentes y futuros, entonces los “procesa” y establece una correspondencia funcional que se convierte en la “repetición” de la red. La idea fue tan ambiciosa como sencilla.

Entre aquellas cosas que debemos destacar es que una BPN no fue originalmente creada para hacer predicciones de serie de tiempo sino para hacer reconocimiento de tramas; es decir (hagamos una abstracción), que se la creó con el propósito de interpolar y mas no de extrapolar. Pero sin embargo una red multicapa que aprenda con propagación inversa está diseñada para imitar el comportamiento de correspondencia funcional de un conjunto de ejemplos. Entendase bien que es esta capacidad la que hemos estado aprovechando, que ciertamente es la única que nos justifica de forma teórica.

En el diseño general de la BPN que resuelve el problema **predictivo** y que discutimos al final del capítulo 2 dijimos que en la capa de salida habrían tantos nodos como pasos hacia al futuro querramos prever. Esto es teóricamente posible pero en la práctica lo que se hace es dejar un único nodo en la capa de salida y correr la red un número de veces igual a la cantidad de valores en el futuro que se desee predecir. Esto sirve para reducir el error, ya que evidentemente sería muchísimo más ruidoso el hacerlo de un solo tajo, y por último pudieran-ros divergir del resultado que esperamos. Pero esto no fue mas que especulaciones hasta que pudimos corroborarlo con un software sofisticado y flexible como esta vez lo fue Trajan 3.0. Este programa simplemente no se “atreve” a predecir más de un solo valor al mismo tiempo. Al final se cierra un círculo de coherencia consistente con lo que se dijo sobre la capacidad de extrapolación de una BPN.

Diseñar la estructura de perceptrones multicapa es lo más sencillo del proceso. Encontrar los parámetros de aprendizaje y momento adecuados puede, sin embargo, tomar un tiempo mayor debido a que están basados en la “experiencia” del modelador. Sería raro

encontrar el caso en que estos parámetros sean los adecuados en sus valores por omisión (default), al contrario, nos encontraremos con la situación en que empezamos a “tantear” estos valores hasta que no hayan divergencias ni exceso de ruido. En nuestro caso, para la serie de pasajeros, tuvimos que disminuir los valores por omisión de estos parámetros por repetidas ocasiones por dos razones fundamentales: en primer lugar, debido a la divergencia del error (tenía mucha variación, tendía a bajar en promedio y luego subía sin control) por la que Trajan 3.0 muestra un mensaje advirtiéndolo de esto; en segundo lugar, por la consecuencia de un mal entrenamiento — las predicciones eran muy malas.

Otra de los puntos destacables es lo que respecta a la necesidad de cambiar la función de activación en la capa de salida. No es que si dejamos que una sigmoide dispare las salidas las predicciones se concentraran en el intervalo  $(0,1)$ , hay que dejar bien en claro este punto. En su momento dijimos que un software sofisticado debería ser capaz de transformar y ajustar los datos automáticamente para que las entradas y salidas puedan ser coherentes con la función sigmoide, pero es evidente que el paquete debe de devolver las inversas de las transformaciones para que puedan ser entendidas por el usuario. En otras palabras, si los datos son transformados para ser procesados, estas transformaciones deben ser “retransformadas” para volver a la naturaleza original de la serie. Trajan 3.0 utiliza la transformación “Minimax” (ver figura 3.2), la que sirve como transformación limitante de los datos, y luego retransforma hacia la naturaleza original antes de mostrar las salidas en la figura de predicción. Aunque los datos son “devueltos” hacia su naturaleza, si utilizamos la función logística o sigmoide estas salidas se verán afectadas por su limitación hacia un intervalo; por el contrario, si utilizamos la función lineal en el disparo de la salida, el efecto casi desaparece, aunque es posible que la utilización de la sigmoide en la capa intermedia se vea reflejada en un retrasado efecto aletargador del “achatamiento”.

Y es aquí donde tenemos justificativos para usar un mayor número de nodos en la capa oculta. El programa nos aconsejó usar 6 nodos en la capa oculta, sin embargo en la estructura que empleamos teníamos 12. La idea que hay detrás de esto es que, como dijimos anteriormente, puede ocurrir que el efecto de achatamiento no desaparezca por completo en la salida final debido a que en la capa intermedia utilizamos la función sigmoide, pero este efecto tiene que ver directamente con los valores de la suma ponderada en la capa final: si hay más entradas en el último nodo, la suma ponderada aumentará en valor y por lo tanto el efecto **achataador** se desvanece con la linealidad de la salida.

Es recomendable la familiarización con los datos disponibles. Esto lo hemos repetido en varias ocasiones, pero se hace necesario la conscientización de este punto al final de nuestro trabajo. De la naturaleza de los datos dependerá todo lo que eventualmente hagamos para resolver nuestro problema **predictivo**: la arquitectura, funciones de activación, parámetros de aprendizaje, parámetros de momento, cantidad de predicciones confiables. Por ejemplo, si tuviéramos una serie que respondiera a una caminata aleatoria (proceso de Markov) bastaría con poner un solo nodo en la capa de entrada y **estuvieramos** obligados a poner un solo nodo en la capa de salida. Por que? Porque en una cadena de Markov el último dato sólo depende del inmediato anterior y de ningún otro. Así pudieramos encontrar cada vez distintos casos que respondieran a diseños de redes muy particulares, por lo que nada se puede decir en concreto y lo único que hemos hecho es mostrar que una red neuronal puede ser capaz de predecir series de tiempo. Por eso es evidente que habrá series de las que **definitivamente** no se podrá encontrar un conjunto de predicciones suficientemente confiables y satisfactorias.

Consecuentemente con lo antes dicho se hace recomendable el tener una buena base del análisis de series de tiempo como soporte teórico de nuestra "consciencia" sobre la serie. Por eso no hemos dicho que es mejor el predecir (al menos actualmente) **via** redes neuronales

que **via** análisis de series de tiempo; lo que sí es cierto es que para nuestro gusto es mucho más interesante, fácil y hasta entretenido.

Por último, debemos tener en cuenta que hemos empleado un método de aprendizaje muy utilizado en redes neuronales, pero esto no quiere decir que no existan otros métodos para resolver nuestro problema (como lo discutimos al final del capítulo 4). La ciencia computacional en la rama de la inteligencia artificial siempre avanza conforme pasa el tiempo. Por eso tenemos que admitir que exista la posibilidad de que próximamente quede obsoleta nuestra forma de resolver este problema.

# ANEXOS



## ANEXO 1: CODIGO DEL PROGRAMA BPN DE KARSTEN KUTZA

### Bpn

```

/*****

Network:      =====
               Backpropagation Network with Bias Terms and Momentum
               =====

Application:  Time-Series Forecasting
               Prediction of the Annual Number of Sunspots

Author:       Karsten Kutza
Date:        17.4.96

Reference:    D.E. Rumelhart, G.E. Hinton, R.J. Williams
               Learning Internal Representations by Error Propagation
               in:
               D.E. Rumelhart, J.L. McClelland (Eds.)
               Parallel Distributed Processing, Volume 1
               MIT Press, Cambridge, MA, pp. 318-362, 1986

*****/

/*****
               D E C L A R A T I O N S
*****/

#include <stdlib.h>
#include <stdio.h>

```

```

#include <math.h>

typedef int      BOOL;
typedef int      INT;
typedef double   REAL;

#define FALSE    0
#define TRUE
#define NOT
#define AND      &&
#define OR

#define MIN_REAL    -HUGE_VAL
#define MAX_REAL    +HUGE_VAL
#define MIN(x,y)    ((x)<(y) ? (x) : (y))
#define MAX(x,y)    ((x)>(y) ? (x) : (y))

#define LO          0.1
#define HI          0.9
#define BIAS

#define sqr(x)      ((x)*(x))

typedef struct {
    INT          Units;          /* A LAYER OF A NET:          */
    REAL*        output;        /* - number of units in this layer */
    REAL*        Error;         /* - output of ith unit      */
    REAL**       Weight;        /* - error term of ith unit   */
    REAL**       WeightSave;    /* - connection weights to ith unit */
    REAL**       dWeight;      /* - saved weights for stopped training */
    REAL**       /* - last weight deltas for momentum */
} LAYER;

typedef struct {
    LAYER**      Layer;         /* A NET:                    */
    LAYER*       InputLayer;    /* - layers of this net      */
    LAYER*       OutputLayer;   /* - input layer             */
    REAL         Alpha;         /* - output layer           */
    REAL         Eta;           /* - momentum factor        */
    REAL         Gain;          /* - learning rate          */
    REAL         Error;         /* - gain of sigmoid function */
} NET;

/*****
    R A N D O M S   D R A W N   F R O M   D I S T R I B U T I O N S
*****/

void InitializeRandoms()

    srand(4711);

INT RandomEqualINT(INT Low, INT High)

    return rand() % (High-Low+1) + Low;

REAL RandomEqualREAL(REAL Low, REAL High)

```

```
{
  return ((REAL) rand() / RAND_MAX) * (High-Low) + Low;
}
```

\*\*\*\*\*  
 APPLICATION - SPECIFIC CODE  
 \*\*\*\*\*/

```
#define NUM_LAYERS 3
#define N 30
#define M 1
INT Units[NUM_LAYERS] = {N, 10, M};
```

```
#define FIRST_YEAR 1700
#define NUM-YEARS 280
```

```
#define TRAIN_LWB (N)
#define TRAINUPB (179)
#define TRAIN_YEARS (TRAINUPB - TRAIN_LWB + 1)
#define TEST_LWB (180)
#define TEST_UPB (259)
#define TESTYEARS (TEST_UPB - TEST_LWB + 1)
#define EVALLWB (260)
#define EVALUPB (NUM_YEARS - 1)
#define EVAL_YEARS (EVALUPB - EVAL_LWB + 1)
```

```
REAL Sunspots_[NUM_YEARS];
REAL Sunspots [NUM-YEARS] = {
```

0.0262,	0.0575,	0.0837,	0.1203,	0.1883,	0.3033,
0.1517,	0.1046,	0.0523,	0.0418,	0.0157,	0.0000,
0.0000,	0.0105,	0.0575,	0.1412,	0.2458,	0.3295,
0.3138,	0.2040,	0.1464,	0.1360,	0.1151,	0.0575,
0.1098,	0.2092,	0.4079,	0.6381,	0.5387,	0.3818,
0.2458,	0.1831,	0.0575,	0.0262,	0.0837,	0.1778,
0.3661,	0.4236,	0.5805,	0.5282,	0.3818,	0.2092,
0.1046,	0.0837,	0.0262,	0.0575,	0.1151,	0.2092,
0.3138,	0.4231,	0.4362,	0.2495,	0.2500,	0.1606,
0.0638,	0.0502,	0.0534,	0.1700,	0.2489,	0.2824,
0.3290,	0.4493,	0.3201,	0.2359,	0.1904,	0.1093,
0.0596,	0.1977,	0.3651,	0.5549,	0.5272,	0.4268,
0.3478,	0.1820,	0.1600,	0.0366,	0.1036,	0.4838,
0.8075,	0.6585,	0.4435,	0.3562,	0.2014,	0.1192,
0.0534,	0.1260,	0.4336,	0.6904,	0.6846,	0.6177,
0.4702,	0.3483,	0.3138,	0.2453,	0.2144,	0.1114,
0.0837,	0.0335,	0.0214,	0.0356,	0.0758,	0.1778,
0.2354,	0.2254,	0.2484,	0.2207,	0.1470,	0.0528,
0.0424,	0.0131,	0.0000,	0.0073,	0.0262,	0.0638,
0.0727,	0.1851,	0.2395,	0.2150,	0.1574,	0.1250,
0.0816,	0.0345,	0.0209,	0.0094,	0.0445,	0.0868,
0.1898,	0.2594,	0.3358,	0.3504,	0.3708,	0.2500,
0.1438,	0.0445,	0.0690,	0.2976,	0.6354,	0.7233,
0.5397,	0.4482,	0.3379,	0.1919,	0.1266,	0.0560,
0.0785,	0.2097,	0.3216,	0.5152,	0.6522,	0.5036,
0.3483,	0.3373,	0.2829,	0.2040,	0.1077,	0.0350,
0.0225,	0.1187,	0.2866,	0.4906,	0.5010,	0.4038,
0.3091,	0.2301,	0.2458,	0.1595,	0.0853,	0.0382,
0.1966,	0.3870,	0.7270,	0.5816,	0.5314,	0.3462,
0.2338,	0.0889,	0.0591,	0.0649,	0.0178,	0.0314,
0.1689,	0.2840,	0.3122,	0.3332,	0.3321,	0.2730,
0.1328,	0.0685,	0.0356,	0.0330,	0.0371,	0.1862,



0.3818,	0.4451,	0.4079,	0.3347,	0.2186,	0.1370,
0.1396,	0.0633,	0.0497,	0.0141,	0.0262,	0.1276,
0.2197,	0.3321,	0.2814,	0.3243,	0.2537,	0.2296,
0.0973,	0.0298,	0.0188,	0.0073,	0.0502,	0.2479,
0.2986,	0.5434,	0.4215,	0.3326,	0.1966,	0.1365,
0.0743,	0.0303,	0.0873,	0.2317,	0.3342,	0.3609,
0.4069,	0.3394,	0.1867,	0.1109,	0.0581,	0.0298,
0.0455,	0.1888,	0.4168,	0.5983,	0.5732,	0.4644,
0.3546,	0.2484,	0.1600,	0.0853,	0.0502,	0.1736,
0.4843,	0.7929,	0.7128,	0.7045,	0.4388,	0.3630,
0.1647,	0.0727,	0.0230,	0.1987,	0.7411,	0.9947,
0.9665,	0.8316,	0.5873,	0.2819,	0.1961,	0.1459,
0.0534,	0.0790,	0.2458,	0.4906,	0.5539,	0.5518,
0.5465,	0.3483,	0.3603,	0.1987,	0.1804,	0.0811,
0.0659,	0.1428,	0.4838,	0.8127		

},

```

REAL          Mean;
REAL          TrainError;
REAL          TrainErrorPredictingMean;
REAL          TestError;
REAL          TestErrorPredictingMean;

```

```
FILE*         f;
```

```
void NormalizeSunspots()
```

```

{
    INT Year;
    REAL Min, Max;

    Min = MAX_REAL;
    Max = MIN_REAL;
    for (Year=0; Year<NUM_YEARS; Year++) {
        Min = MIN(Min, Sunspots[Year]);
        Max = MAX(Max, Sunspots[Year]);
    }
    Mean = 0;
    for (Year=0; Year<NUM_YEARS; Year++) {
        Sunspots_[Year] =
        Sunspots[Year] = ((Sunspots[Year]-Min) / (Max-Min)) * (HI-LO) + LO;
        Mean += Sunspots[Year] / NUM_YEARS;
    }
}

```

```
void InitializeApplication(NET* Net)
```

```

    INT Year, i;
    REAL Out, Err;

    Net->Alpha = 0.5;
    Net->Eta    = 0.05;
    Net->Gain   = 1;

    NormalizeSunspots();
    TrainErrorPredictingMean = 0;
    for (Year=TRAIN_LWB; Year<=TRAIN_UPB; Year++) {
        for (i=0; i<M; i++) {
            Out = Sunspots[Year+i];
            Err = Mean - Out;
            TrainErrorPredictingMean += 0.5 * sqr(Err);
        }
    }
}

```

```

}

TestErrorPredictingMean = 0;
for (Year=TEST_LWB; Year<=TEST_UPB; Year++) {
    for (i=0; i<M; i++) {
        Out = Sunspots[Year+i];
        Err = Mean - Out;
        TestErrorPredictingMean += 0.5 * sqr(Err);
    }

    f = fopen("BPN.txt", "w");
}

void FinalizeApplication()
{
    fclose(f);
}

/*****
                I N I T I A L I Z A T I O N
*****/

void GenerateNetwork(NET* Net)
{
    INT l,i;

    Net->Layer = (LAYER**) calloc(NUM_LAYERS, sizeof(LAYER*));

    for (l=0; l<NUM_LAYERS; l++) {
        Net->Layer[l] = (LAYER*) malloc(sizeof(LAYER));

        Net->Layer[l]->Units      = Units[l];
        Net->Layer[l]->Output    = (REAL*)  calloc(Units[l]+1, sizeof(REAL));
        Net->Layer[l]->Error     = (REAL*)  calloc(Units[l]+1, sizeof(REAL));
        Net->Layer[l]->Weight    = (REAL**)  calloc(Units[l]+1, sizeof(REAL*));
        Net->Layer[l]->WeightSave = (REAL**)  calloc(Units[l]+1, sizeof(REAL*));
        Net->Layer[l]->dWeight   = (REAL**)  calloc(Units[l]+1, sizeof(REAL*));
        Net->Layer[l]->Output[0] = BIAS;

        if (l != 0) {
            for (i=1; i<=Units[l]; i++) {
                Net->Layer[l]->Weight[i]      = (REAL*)  calloc(Units[l-1]+1,
                sizeof(REAL));
                Net->Layer[l]->WeightSave[i] = (REAL*)  calloc(Units[l-1]+1,
                sizeof(REAL));
                Net->Layer[l]->dWeight[i]     = (REAL*)  calloc(Units[l-1]+1,
                sizeof(REAL));
            }

            Net->InputLayer = Net->Layer[0];
            Net->OutputLayer = Net->Layer[NUM_LAYERS - 1];
            Net->Alpha      = 0.9;
            Net->Eta        = 0.25;
            Net->Gain       = 1;
        }
    }

    void RandomWeights(NET* Net)
    {

```

```

INT l,i,j;

for (l=1; l<NUM_LAYERS; l++) {
  for (i=1; i<=Net->Layer[l]->Units; i++) {
    for (j=0; j<=Net->Layer[l-1]->Units; j++) {
      Net->Layer[l]->Weight[i][j] = RandomEqualREAL(-0.5, 0.5);
    }
  }
}

void SetInput(NET* Net, REAL* Input)
{
  INT i;

  for (i=1; i<=Net->InputLayer->Units; itt) {
    Net->InputLayer->Output[i] = Input[i-1];
  }
}

void GetOutput(NET* Net, REAL* Output)
{
  INT i;

  for (i=1; i<=Net->OutputLayer->Units; itt) {
    Output[i-1] = Net->OutputLayer->Output[i];
  }
}

/*****
      SUPPORT FOR STOPPED TRAINING
*****/

void SaveWeights(NET* Net)
{
  INT l,i,j;

  for (l=1; l<NUM_LAYERS; l++) {
    for (i=1; i<=Net->Layer[l]->Units; itt) {
      for (j=0; j<=Net->Layer[l-1]->Units; j++) {
        Net->Layer[l]->WeightSave[i][j] = Net->Layer[l]->Weight[i][j];
      }
    }
  }
}

void RestoreWeights(NET* Net)
{
  INT l,i,j;

  for (l=1; l<NUM_LAYERS; l++) {
    for (i=1; i<=Net->Layer[l]->Units; itt) {
      for (j=0; j<=Net->Layer[l-1]->Units; j++) {
        Net->Layer[l]->Weight[i][j] = Net->Layer[l]->WeightSave[i][j];
      }
    }
  }
}

```

```

/*****
      P R O P A G A T I N G   S I G N A L S
*****/

```

```

void PropagateLayer(NET* Net, LAYER* Lower, LAYER* Upper)
{
  INT i,j;
  REAL Sum;

  for (i=1; i<=Upper->Units; i++) {
    sum = 0;
    for (j=0; j<=Lower->Units; j++) {
      Sum += Upper->Weight[i][j] * Lower->Output[j];

      Upper->Output[i] = 1 / (1 + exp(-Net->Gain * Sum));
    }
  }
}

```

```

void PropagateNet(NET* Net)
{
  INT l;

  for (l=0; l<NUM_LAYERS-1; l++) {
    PropagateLayer(Net, Net->Layer[l], Net->Layer[l+1]);
  }
}

```

```

/*****
      B A C K P R O P A G A T I N G   E R R O R S
*****/

```

```

void ComputeOutputError(NET* Net, REAL* Target)
{
  INT i;
  REAL Out, Err;

  Net->Error = 0;
  for (i=1; i<=Net->OutputLayer->Units; i++) {
    Out = Net->OutputLayer->Output[i];
    Err = Target[i-1]-Out;
    Net->OutputLayer->Error[i] = Net->Gain * Out * (1-Out) * Err;
    Net->Error += 0.5 * sqr(Err);
  }
}

```

```

void BackpropagateLayer(NET* Net, LAYER* Upper, LAYER* Lower)

  INT i,j;
  REAL Out, Err;

  for (i=1; i<=Lower->Units; itt) {
    Out = Lower->Output[i];
    Err = 0;
    for (j=1; j<=Upper->Units; jtt) {
      Err t= Upper->Weight[j][i] * Upper->Error[j];
    }
  }

```

```

    Lower->Error[i] = Net->Gain * Out * (1-Out) * Err;
}
}

```

```
void BackpropagateNet(NET* Net)
```

```

    INT l;

    for (l=NUM_LAYERS-1; l>1; l--) {
        BackpropagateLayer(Net, Net->Layer[l], Net->Layer[l-1]);
    }
}

```

```
void AdjustWeights(NET* Net)
```

```

    INT l,i,j;
    REAL Out, Err, dWeight;

    for (l=1; l<NUM_LAYERS; l++) {
        for (i=1; i<=Net->Layer[l]->Units; i++) (
            if (i>=Net->Layer[l]->Units-1)
                {
                    printf("Ya pase y deber;a aumentar el l pero no lo hago\n");
                }
            l++;
            for (j=0; j<=Net->Layer[l-1]->Units; j++) {
                Out = Net->Layer[l-1]->Output[j];
                Err = Net->Layer[l]->Error[i];
                dWeight = Net->Layer[l]->dWeight[i][j];
                Net->Layer[l]->Weight[i][j] += Net->Eta * Err * Out + Net->Alpha *
dWeight;
                Net->Layer[l]->dWeight[i][j] = Net->Eta * Err * Out;
            }
        }
}

```

```

/*****
                        S I M U L A T I N G   T H E   N E T
*****/

```

```
void SimulateNet(NET* Net, REAL* Input, REAL* Output, REAL* Target, BOOL
Training)
```

```

    SetInput(Net, Input);
    PropagateNet(Net);
    GetOutput(Net, Output);

    ComputeOutputError(Net, Target);
    if (Training) {
        BackpropagateNet(Net);
        AdjustWeights(Net);
    }
}

```

```
void TrainNet(NET* Net, INT Epochs)
```

```
{
    INT Year, n;
```





```

REAL Output[M];

for (n=0; n<Epochs*TRAIN_YEARS; n++) {
    Year = RandomEqualINT(TRAIN_LWB, TRAIN_UPB);
    SimulateNet(Net, &(Sunspots[Year-N]), Output, &(Sunspots[Year]), TRUE);
}

void TestNet(NET* Net)
{
    INT Year;
    REAL Output[M];

    TrainError = 0;
    for (Year=TRAIN_LWB; Year<=TRAIN_UPB; Year++) {
        SimulateNet(Net, &(Sunspots[Year-N]), Output, &(Sunspots[Year]), FALSE);
        TrainError += Net->Error;
    }

    TestError = 0;
    for (Year=TEST_LWB; Year<=TEST_UPB; Year++) {
        SimulateNet(Net, &(Sunspots[Year-N]), Output, &(Sunspots[Year]), FALSE);
        TestError += Net->Error;
    }

    fprintf(f, "\nNMSE is %0.3f on Training Set and %0.3f on Test Set",
           TrainError / TrainErrorPredictingMean,
           TestError / TestErrorPredictingMean);
}

void EvaluateNet(NET* Net)
{
    INT Year;
    REAL Output [M];
    REAL Output_ [M];

    fprintf(f, "\n\n\n");
    fprintf(f, "Year      Sunspots      Open-Loop Prediction      Closed-Loop
Prediction\n");
    fprintf(f, "\n");
    for (Year=Eval_LWB; Year<=Eval_UPB; Year++) {
        SimulateNet(Net, &(Sunspots [Year-N]), Output, &(Sunspots [Year]), FALSE);
        SimulateNet(Net, &(Sunspots_[Year-N]), Output_, &(Sunspots_[Year]), FALSE);
        Sunspots [Year] = Output [0];
        fprintf(f, "%d          %0.3f          %0.3f
%0.3f\n",
                FIRST_YEAR + Year,
                Sunspots[Year],
                Output [0],
                Output- [0]);
    }
}

```

```

/*****
                                M A I N
*****/

void main()

    NET Net;
    BOOL Stop;
    REAL MinTestError;

    InitializeRandoms();
    GenerateNetwork(&Net);
    RandomWeights(&Net);
    InitializeApplication(&Net);

    Stop = FALSE;
    MinTestError = MAX_REAL;
    do {
        TrainNet(&Net, 10);
        TestNet(&Net);
        if (TestError < MinTestError) {
            fprintf(f, " - saving Weights . ..").
            MinTestError = TestError;
            SaveWeights(&Net);
        }
        else if (TestError > 1.2 * MinTestError) {
            fprintf(f, " - stopping Training and restoring Weights . ..").
            Stop = TRUE;
            RestoreWeights(&Net);
        }
    } while (NOT Stop);

    TestNet(&Net);
    EvaluateNet(&Net);

    FinalizeApplication();

```

## ANEXO 2: CORRIDA COMPLETA DEL PROGRAMA DE KARSTEN KUTZA

NMSE is 0.879 on Training Set and 0.834 on Test Set - saving Weights . . .  
 NMSE is 0.818 on Training Set and 0.783 on Test Set - saving Weights . . .  
 NMSE is 0.749 on Training Set and 0.693 on Test Set - saving Weights . . .  
 NMSE is 0.691 on Training Set and 0.614 on Test Set - saving Weights . . .  
 NMSE is 0.622 on Training Set and 0.555 on Test Set - saving Weights . . .  
 NMSE is 0.569 on Training Set and 0.491 on Test Set - saving Weights . . .  
 NMSE is 0.533 on Training Set and 0.467 on Test Set - saving Weights . . .  
 NMSE is 0.490 on Training Set and 0.416 on Test Set - saving Weights . . .  
 NMSE is 0.470 on Training Set and 0.401 on Test Set - saving Weights . . .  
 NMSE is 0.441 on Training Set and 0.361 on Test Set - saving Weights . . .  
 NMSE is 0.423 on Training Set and 0.345 on Test Set - saving Weights . . .  
 NMSE is 0.402 on Training Set and 0.329 on Test Set - saving Weights . . .  
 NMSE is 0.385 on Training Set and 0.319 on Test Set - saving Weights . . .  
 NMSE is 0.381 on Training Set and 0.323 on Test Set  
 NMSE is 0.356 on Training Set and 0.292 on Test Set - saving Weights . . .  
 NMSE is 0.350 on Training Set and 0.279 on Test Set - saving Weights . . .  
 NMSE is 0.333 on Training Set and 0.272 on Test Set - saving Weights . . .  
 NMSE is 0.322 on Training Set and 0.258 on Test Set - saving Weights . . .  
 NMSE is 0.318 on Training Set and 0.250 on Test Set - saving Weights . . .  
 NMSE is 0.303 on Training Set and 0.244 on Test Set - saving Weights . . .  
 NMSE is 0.303 on Training Set and 0.235 on Test Set - saving Weights . . .  
 NMSE is 0.290 on Training Set and 0.238 on Test Set  
 NMSE is 0.281 on Training Set and 0.224 on Test Set - saving Weights . . .  
 NMSE is 0.275 on Training Set and 0.215 on Test Set - saving Weights . . .  
 NMSE is 0.275 on Training Set and 0.208 on Test Set - saving Weights . . .  
 NMSE is 0.264 on Training Set and 0.209 on Test Set  
 NMSE is 0.259 on Training Set and 0.207 on Test Set - saving Weights . . .  
 NMSE is 0.255 on Training Set and 0.203 on Test Set - saving Weights . . .  
 NMSE is 0.251 on Training Set and 0.202 on Test Set - saving Weights . . .  
 NMSE is 0.247 on Training Set and 0.198 on Test Set - saving Weights . . .  
 NMSE is 0.246 on Training Set and 0.203 on Test Set  
 NMSE is 0.243 on Training Set and 0.191 on Test Set - saving Weights . . .

NMSE is 0.241 on Training Set and 0.198 on Test Set  
 NMSE is 0.233 on Training Set and 0.185 on Test Set - saving Weights . . .  
 NMSE is 0.235 on Training Set and 0.184 on Test Set - saving Weights . . .  
 NMSE is 0.232 on Training Set and 0.192 on Test Set  
 NMSE is 0.224 on Training Set and 0.183 on Test Set - saving Weights . . .  
 NMSE is 0.222 on Training Set and 0.179 on Test Set - saving Weights . . .  
 NMSE is 0.219 on Training Set and 0.179 on Test Set - saving Weights . . .  
 NMSE is 0.225 on Training Set and 0.188 on Test Set  
 NMSE is 0.213 on Training Set and 0.172 on Test Set - saving Weights . . .  
 NMSE is 0.217 on Training Set and 0.171 on Test Set - saving Weights . . .  
 NMSE is 0.208 on Training Set and 0.169 on Test Set - saving Weights . . .  
 NMSE is 0.207 on Training Set and 0.170 on Test Set  
 NMSE is 0.204 on Training Set and 0.167 on Test Set - saving Weights . . .  
 NMSE is 0.203 on Training Set and 0.172 on Test Set  
 NMSE is 0.200 on Training Set and 0.170 on Test Set  
 NMSE is 0.199 on Training Set and 0.172 on Test Set  
 NMSE is 0.201 on Training Set and 0.165 on Test Set - saving Weights . . .  
 NMSE is 0.195 on Training Set and 0.164 on Test Set - saving Weights . . .  
 NMSE is 0.200 on Training Set and 0.162 on Test Set - saving Weights . . .  
 NMSE is 0.193 on Training Set and 0.164 on Test Set  
 NMSE is 0.195 on Training Set and 0.167 on Test Set  
 NMSE is 0.190 on Training Set and 0.162 on Test Set - saving Weights . . .  
 NMSE is 0.188 on Training Set and 0.155 on Test Set - saving Weights . . .  
 NMSE is 0.189 on Training Set and 0.161 on Test Set  
 NMSE is 0.189 on Training Set and 0.163 on Test Set  
 NMSE is 0.184 on Training Set and 0.155 on Test Set - saving Weights . . .  
 NMSE is 0.196 on Training Set and 0.175 on Test Set  
 NMSE is 0.183 on Training Set and 0.160 on Test Set  
 NMSE is 0.184 on Training Set and 0.164 on Test Set  
 NMSE is 0.181 on Training Set and 0.161 on Test Set  
 NMSE is 0.180 on Training Set and 0.159 on Test Set  
 NMSE is 0.187 on Training Set and 0.155 on Test Set  
 NMSE is 0.178 on Training Set and 0.159 on Test Set  
 NMSE is 0.176 on Training Set and 0.153 on Test Set - saving Weights . . .  
 NMSE is 0.177 on Training Set and 0.160 on Test Set  
 NMSE is 0.177 on Training Set and 0.153 on Test Set - saving Weights . . .  
 NMSE is 0.179 on Training Set and 0.152 on Test Set - saving Weights . . .  
 NMSE is 0.173 on Training Set and 0.152 on Test Set - saving Weights . . .  
 NMSE is 0.171 on Training Set and 0.155 on Test Set  
 NMSE is 0.174 on Training Set and 0.152 on Test Set - saving Weights . . .  
 NMSE is 0.170 on Training Set and 0.154 on Test Set  
 NMSE is 0.170 on Training Set and 0.156 on Test Set  
 NMSE is 0.170 on Training Set and 0.156 on Test Set  
 NMSE is 0.169 on Training Set and 0.151 on Test Set - saving Weights . . .  
 NMSE is 0.167 on Training Set and 0.152 on Test Set  
 NMSE is 0.167 on Training Set and 0.152 on Test Set  
 NMSE is 0.166 on Training Set and 0.156 on Test Set  
 NMSE is 0.172 on Training Set and 0.153 on Test Set  
 NMSE is 0.166 on Training Set and 0.155 on Test Set  
 NMSE is 0.172 on Training Set and 0.154 on Test Set  
 NMSE is 0.168 on Training Set and 0.151 on Test Set  
 NMSE is 0.164 on Training Set and 0.148 on Test Set - saving Weights . . .  
 NMSE is 0.164 on Training Set and 0.152 on Test Set  
 NMSE is 0.162 on Training Set and 0.148 on Test Set - saving Weights . . .  
 NMSE is 0.165 on Training Set and 0.154 on Test Set  
 NMSE is 0.161 on Training Set and 0.146 on Test Set - saving Weights . . .  
 NMSE is 0.160 on Training Set and 0.149 on Test Set

NMSE is 0.160 on Training Set and 0.149 on Test Set  
 NMSE is 0.159 on Training Set and 0.146 on Test Set - saving Weights . . .  
 NMSE is 0.160 on Training Set and 0.148 on Test Set  
 NMSE is 0.158 on Training Set and 0.145 on Test Set - saving Weights . . .  
 NMSE is 0.157 on Training Set and 0.144 on Test Set - saving Weights . . .  
 NMSE is 0.160 on Training Set and 0.141 on Test Set - saving Weights . . .  
 NMSE is 0.157 on Training Set and 0.144 on Test Set  
 NMSE is 0.159 on Training Set and 0.150 on Test Set  
 NMSE is 0.157 on Training Set and 0.144 on Test Set  
 NMSE is 0.157 on Training Set and 0.150 on Test Set  
 NMSE is 0.156 on Training Set and 0.150 on Test Set  
 NMSE is 0.154 on Training Set and 0.144 on Test Set  
 NMSE is 0.154 on Training Set and 0.146 on Test Set  
 NMSE is 0.155 on Training Set and 0.149 on Test Set  
 NMSE is 0.154 on Training Set and 0.148 on Test Set  
 NMSE is 0.152 on Training Set and 0.144 on Test Set  
 NMSE is 0.153 on Training Set and 0.145 on Test Set  
 NMSE is 0.151 on Training Set and 0.143 on Test Set  
 NMSE is 0.151 on Training Set and 0.145 on Test Set  
 NMSE is 0.152 on Training Set and 0.143 on Test Set  
 NMSE is 0.152 on Training Set and 0.147 on Test Set  
 NMSE is 0.151 on Training Set and 0.141 on Test Set - saving Weights . . .  
 NMSE is 0.154 on Training Set and 0.141 on Test Set  
 NMSE is 0.152 on Training Set and 0.147 on Test Set  
 NMSE is 0.150 on Training Set and 0.146 on Test Set  
 NMSE is 0.149 on Training Set and 0.146 on Test Set  
 NMSE is 0.150 on Training Set and 0.150 on Test Set  
 NMSE is 0.148 on Training Set and 0.143 on Test Set  
 NMSE is 0.150 on Training Set and 0.147 on Test Set  
 NMSE is 0.147 on Training Set and 0.144 on Test Set  
 NMSE is 0.153 on Training Set and 0.142 on Test Set  
 NMSE is 0.147 on Training Set and 0.144 on Test Set  
 NMSE is 0.146 on Training Set and 0.144 on Test Set  
 NMSE is 0.147 on Training Set and 0.141 on Test Set  
 NMSE is 0.145 on Training Set and 0.141 on Test Set - saving Weights . . .  
 NMSE is 0.145 on Training Set and 0.141 on Test Set - saving Weights . . .  
 NMSE is 0.145 on Training Set and 0.140 on Test Set - saving Weights . . .  
 NMSE is 0.145 on Training Set and 0.142 on Test Set  
 NMSE is 0.147 on Training Set and 0.140 on Test Set - saving Weights . . .  
 NMSE is 0.150 on Training Set and 0.140 on Test Set  
 NMSE is 0.150 on Training Set and 0.141 on Test Set  
 NMSE is 0.143 on Training Set and 0.144 on Test Set  
 NMSE is 0.143 on Training Set and 0.142 on Test Set  
 NMSE is 0.142 on Training Set and 0.141 on Test Set  
 NMSE is 0.142 on Training Set and 0.143 on Test Set  
 NMSE is 0.142 on Training Set and 0.143 on Test Set  
 NMSE is 0.142 on Training Set and 0.146 on Test Set  
 NMSE is 0.141 on Training Set and 0.143 on Test Set  
 NMSE is 0.146 on Training Set and 0.141 on Test Set  
 NMSE is 0.144 on Training Set and 0.141 on Test Set  
 NMSE is 0.140 on Training Set and 0.142 on Test Set  
 NMSE is 0.144 on Training Set and 0.148 on Test Set  
 NMSE is 0.140 on Training Set and 0.139 on Test Set - saving Weights . . .  
 NMSE is 0.140 on Training Set and 0.140 on Test Set  
 NMSE is 0.141 on Training Set and 0.138 on Test Set - saving Weights . . .  
 NMSE is 0.139 on Training Set and 0.140 on Test Set  
 NMSE is 0.138 on Training Set and 0.141 on Test Set









NMSE is 0.102 on Training Set and 0.161 on Test Set  
 NMSE is 0.104 on Training Set and 0.154 on Test Set  
 NMSE is 0.102 on Training Set and 0.160 on Test Set  
 NMSE is 0.102 on Training Set and 0.160 on Test Set  
 NMSE is 0.100 on Training Set and 0.157 on Test Set  
 NMSE is 0.105 on Training Set and 0.153 on Test Set  
 NMSE is 0.100 on Training Set and 0.155 on Test Set  
 NMSE is 0.101 on Training Set and 0.154 on Test Set  
 NMSE is 0.100 on Training Set and 0.158 on Test Set  
 NMSE is 0.107 on Training Set and 0.170 on Test Set - stopping Training  
 and restoring Weights . . .  
 NMSE is 0.141 on Training Set and 0.138 on Test Set

Year	Sunspots	Open-Loop	Prediction	Closed-Loop	Prediction
1960	0.572		0.532		0.532
1961	0.327		0.334		0.301
1962	0.258		0.158		0.146
1963	0.217		0.156		0.098
1964	0.143		0.236		0.149
1965	0.164		0.230		0.273
1966	0.298		0.263		0.405
1967	0.495		0.454		0.552
1968	0.545		0.615		0.627
1969	0.544		0.550		0.589
1970	0.540		0.474		0.464
1971	0.380		0.455		0.305
1972	0.390		0.270		0.191
1973	0.260		0.275		0.139
1974	0.245		0.211		0.158
1975	0.165		0.181		0.170
1976	0.153		0.128		0.175
1977	0.215		0.151		0.193
1978	0.489		0.316		0.274
1979	0.754		0.622		0.373

## BIBLIOGRAFIA

1. Box George E.P., Jenkins Gwilym M., Reinsel Gregory C.; "Time Series Analysis – Forecasting and Control"; 3ª edición; Pt-entice Hall; 1994.
2. Rich Elaine, Knight Kevin; "Inteligencia Artificial"; 2ª edición; Mc Graw Hill; 1994.
3. Freeman James A., Skapura David M.; "Redes Neuronales – Algoritmos, aplicaciones y técnicas de programación"; 1ª edición; Addison – Wesley Iberoamericana, S.A.; 1993.
4. NeuralWare, Inc., Technical Publications Group; "Neural Computing – A technology Hanbook for Professional II/Plus and NeuralWorks Explore?"; NeuralWare, Inc.; 1996.
5. Delgado R. J. Alberto; "Elementos de Redes Neuronales y Algoritmos Genéticos"; 1ª edición; **Neurales** Ltda.; 1992.
6. Russel Ingrid F.; " **Article** – Neural Networks"; Department of Computer Science, University of Hartford, CT 06117, [irussell@uhavax.hartford.edu](mailto:irussell@uhavax.hartford.edu).
7. Trajan Software Ltd.; "Trajan Neural Networks Help"; Version 3.0; 1998.
8. <http://www.trajan-software.demon.co.uk>
9. <http://www.neuralware.com>
10. <http://www.statsoft.com>