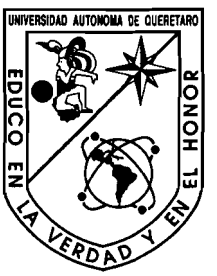
México

UNIVERSIDAD AUTÓNOMA DE QUERÉTARO
FACULTAD DE INFORMÁTICA



ANÁLISIS Y DESARROLLO DE UNA RED NEURONAL DISTRIBUIDA

TESIS

QUE PARA OBTENER EL TITULO DE:

INGENIERO EN COMPUTACIÓN

PRESENTA:

GABRIELA JIMÉNEZ MONTOYA

DIRIGIDA POR:
M.C. MA. TERESA GARCÍA RAMÍREZ

Santiago de Querétaro, Qro., Mayo del 2007

Universidad Autónoma de Querétaro
Facultad de Informática
Ingeniería en Computación

ANÁLISIS Y DESARROLLO DE UNA RED NEURONAL DISTRIBUIDA

TESIS

Que para obtener el TITULO de:

INGENIERO EN COMPUTACIÓN

Presenta
GABRIELA JIMÈNEZ MONTOYA

Dirigida por: M.C. Teresa García Ramírez

SINODALES

M.C. Ma. Teresa García Ramírez Presidente	_____ Firma
Dr. Efrén Gorrostieta Hurtado Secretario	_____ Firma
M.C. Miguel Ángel Ceballos Balderas Vocal	_____ Firma
L.I. Jesús Martín Jaramillo Morales Suplente	_____ Firma
M.C. Rosa Maria Romero González Suplente	_____ Firma

_____

I.S.C Alejandro Santoyo Rodríguez
Director de la Facultad

Centro Universitario
Santiago de Querétaro, Qro.
Mayo del 2007

RESUMEN

Una de las diversas técnicas que emplea la Inteligencia Artificial para conseguir un comportamiento inteligente de forma artificial, son las llamadas redes neuronales. En la actualidad las redes neuronales se utilizan en la investigación y la industria, debido sus modelos artificiales y simplificados del cerebro humano capaces de adquirir conocimiento a través de la experiencia. Sin embargo, los modelos modernos de cómputo no procesan de manera aceptable estos modelos artificiales, debido a esto se proponen la construcción de arquitecturas de entornos distribuidos con el uso de componentes middelware basados en los estándares COM/DCOM, Java RMI, o CORBA. Los sistemas desarrollados con estas tecnologías adquieren parámetros que hoy en día son importantes en el desarrollo de diversas aplicaciones dentro de la inteligencia artificial para el proceso de aprendizaje. Entre otras cosas los sistemas pueden reconfigurarse fácilmente, obtienen una alta escalabilidad y se convierten en sistemas dinámicos, flexibles y abiertos en el uso de tecnologías de software. En este trabajo de tesis se presenta el diseño y desarrollo de una red neuronal distribuida basada en el Perceptron de Rossenblatt de tres capas con alimentación hacia adelante utilizando el estándar de CORBA, para el reconocimiento de los dígitos manuscritos de la base de datos MNIST. El funcionamiento del sistema se crea en un entorno uníarquitectural utilizando el sistema operativo Windows XP, el lenguaje de programación C++, y el middelware VisiBroker. El diseño de las interfaces del sistema distribuido contempla el contenido de los dos conceptos en el modelo de comunicación de los sistemas distribuidos, cliente y servidor. El servidor realiza las funciones de la capa de reacción y los clientes las funciones de la capa asociativa de las dos capas de las tres que componen el Perceptron de Rossenblatt. Finalmente se presentan diferentes pruebas al sistema y se explican gráficas de tiempos y resultados de aprendizaje.

(**Palabras clave:** Perceptron Distribuido, CORBA, Middelware, Sistemas Distribuidos, Redes Neuronales)

DEDICATORIA

A Dios, A mis padres, hermanos, sobrinos y Erik

AGRADECIMIENTOS

*A **Dios** por la vida que me ha tocado vivir y las metas que me ha dejado conseguir. Gracias Dios por este sueño logrado.*

*A las personas mas importantes en mi vida, **mis Padres y Hermanos** a los que siempre tuve en los momentos difíciles, de alegría, de angustia, a ellos les debo decir gracias infinitamente por siempre creer en mi y por darme la herencia mas grande que pueda yo obtener de su parte, su amor y comprensión. Gracias **Mamita, Papito, yolas, beto y hugo** por que juntos hemos logrado una meta más que sin sus sacrificios, trabajo y dedicación no hubiese llegado a su fin. Los Amo.*

*A **Erik** por su compañía, cariño, comprensión, amistad y sobre todo por su Amor, gracias Enano por permitirme ser parte de tu vida y sobre todo gracias por ayudarme a conocer el significado de esfuerzo, dedicación, entrega y valor. Gracias por estar siempre a mi lado y por terminar una meta más juntos. Te Amo*

*A mis **Sobrinitos** Miriam, Adrián, Santi y Fer, gracias por vivir y llenar de alegría mi vida.*

*A mis **Sinodales** M.C. Teresa García Ramírez, Dr. Efrén Gorrostieta Hurtado, M.C. Miguel Ángel Ceballos Balderas, L.I. Jesús Martín Jaramillo Morales, M.C. Rosa Maria Romero González, por sus sabios y atinados consejos por que sin estos no hubiese sido posible este trabajo.*

*A mis **maestros** Alberto, Ricardo, Adriana, Tere, Martín Jaramillo, por darme la oportunidad de conocerlos y de trabajar con uds. Gracias por que quedan en mi corazón como grandes maestros y como grandes amigos.*

*A mis **Amigos** a todos ellos les doy las gracias por estar conmigo en esta etapa de nuestras vidas, recuerden nunca los olvidare. A **Jez y Krlo** mis súper amigos gratzie por la confianza, el apoyo y la amistad, Valen 1000 los quiero mucho. A **Sor, Be y Sandy** mis amiguísimas del alma, cada una de Uds. tiene su propio espacio en mi corazón, gracias eternamente. A **Lenin, Gus, Ayipey e Iván**, gracias por el trabajo en equipo, por el apoyo, pero sobre todo por su Amistad, los quiero.*

*Al grupo de **Robótica** por cada uno de los momentos de adrenalina que vivimos juntos, por el compañerismo, por el trabajo en equipo pero sobre todo por el apoyo que siempre hubo entre todos por sacar adelante nuestros sueños e ideales. Gracias a todos por que nunca nadie venció nuestros objetivos.*

*A **CONCYTEQ** por la oportunidad que me ha brindado para desarrollarme profesionalmente, y sobre todo por el apoyo que siempre me ha brindado. En especial a le agradezco a la D.G. Alicia Arriaga, por esa confianza que me ha brindado.*

*En especial a **Rafa [QEPD]** mi maestro y amigo, donde quiera que te encuentres gracias, gracias por la confianza, por el apoyo, por la amistad, por tus enseñanzas y por tu dedicación. Te quiero Mucho y te extrañamos mucho, siempre vivirás en mi corazón.*

*A **todos** los que hicieron posible este trabajo, mil gracias.*

*Atte: **Gabillas**☺*

ÍNDICE DE CONTENIDO

RESUMEN I

DEDICATORIA II

AGRADECIMIENTOS III

ÍNDICE DE CONTENIDO V

ÍNDICE DE FIGURAS VII

ÍNDICE DE TABLAS IX

ACRÓNIMOS X

PARTE I: 1

ANTECEDENTES 1

CAPÍTULO 1: INTRODUCCIÓN 2

CAPÍTULO 2: TRABAJOS RELACIONADOS 4

CAPÍTULO 3: REDES NEURONALES 6

3.1 INTRODUCCIÓN BIOLÓGICA 6
3.2 HISTORIA DE LAS REDES NEURONALES ARTIFICIALES 7
3.3 MODELO GENERAL DE UNA NEURONA ARTIFICIAL 8
3.4 MODELO ESTÁNDAR DE NEURONA ARTIFICIAL 10
3.5 ESTRUCTURA DE UN SISTEMA NEURONAL ARTIFICIAL 12
3.6 MODELOS DE REDES NEURONALES ARTIFICIALES 14
3.6.1.1 Perceptron Simple 15
3.6.1.2 Adalina 16
3.7 VENTAJAS DE LAS REDES NEURONALES 17
3.8 APLICACIONES DE LAS REDES NEURONALES 17

CAPÍTULO 4: PERCEPTRON 19

4.2 FUNCIÓN DEL PERCEPTRON 19
4.3 ALGORITMO DE APRENDIZAJE DEL PERCEPTRON SIMPLE 21

CAPÍTULO 5: SISTEMAS DISTRIBUIDOS 26

5.1 HISTORIA DE SISTEMAS DISTRIBUIDOS 26
5.2 CARACTERÍSTICAS DE LOS SISTEMAS DISTRIBUIDOS 26
5.3. DESVENTAJAS DE UN SISTEMA DISTRIBUIDO 27
5.4 APLICACIONES DE SISTEMAS DISTRIBUIDOS 27
5.5 HERRAMIENTAS PARA CONSTRUIR OBJETOS DISTRIBUIDOS 27
5.5.1 DCOM (Distributed Component Object Model) *27*
5.5.2 Java RMI (Remote Method Invocation) *28*

CAPÍTULO 6: CORBA 31

6.1 CORBA 31
6.1.1 Especificaciones y terminología CORBA *31*
6.1.2 El lenguaje IDL *34*
6.1.3 Stubs y Skeletons *35*
6.1.4 ORB *35*

6.1.5 GIOP *36*
6.1.2 Referencias a objetos *36*
6.3 VENTAJAS DE CORBA 37
CAPÍTULO 7: PLANTEAMIENTO DEL PROBLEMA 38
7.1 OBJETIVOS GENERALES Y ESPECÍFICOS 38
7.1.1 Objetivos Generales *38*
7.1.2 Objetivos específicos *38*
7.2 HIPÓTESIS QUE SUSTENTA EL TRABAJO 38
7.3 IMPORTANCIA DEL TEMA 39
7.4 JUSTIFICACIÓN 39
PARTE II: 41
DESARROLLO 41
CAPÍTULO 8: DESARROLLO DE LA RED NEURONAL 42
CAPÍTULO 9: DESARROLLO DE LA RED NEURONAL DISTRIBUIDA 61
9.1 DESARROLLO DEL SERVIDOR 61
9.2 DESARROLLO DEL CLIENTE 72
9.3 DISEÑO Y CONFIGURACIÓN DE LA RED 76
PARTE III: 78
RESULTADOS 78
CAPÍTULO 10: PRUEBAS DEL PERCEPTRON 79
CAPÍTULO 11: CONCLUSIONES 89
CAPÍTULO 12: TRABAJO A FUTURO 91
BIBLIOGRAFÍA 93

ÍNDICE DE FIGURAS

Fig. Descripción **Pág.**

FIGURA 1. ESTRUCTURA NEURONAL BIOLÓGICA 7
FIGURA 2. MODELO GENÉRICO DE UNA NEURONA ARTIFICIAL 10
FIGURA 3. MODELO DE UNA NEURONA ARTIFICIAL ESTÁNDAR 11
FIGURA 4. ESTRUCTURA JERÁRQUICA DE UN SISTEMA BASADO EN EL SISTEMA ARTIFICIAL NEURONAL 13
FIGURA 5. CLASIFICACIÓN DE LOS ANS POR EL TIPO DE APRENDIZAJE Y ARQUITECTURA 14
FIGURA 6. PERCEPTRON SIMPLE 15
FIGURA 7. NEURONA LINEAL DE LA RED ADALINA 16
FIGURA 8. NEURONA Y PERCEPTRON 20
FIGURA 9. PERCEPTRON CON UMBRAL AJUSTABLE, IMPLEMENTADO COMO UN PESO ADICIONAL 21
FIGURA 10. CLASIFICACIÓN DE PATRONES LINEALMENTE SEPARABLES 22
FIGURA 11. PERCEPTRON APRENDIENDO A CLASIFICAR OBJETOS 24
FIGURA 12. APLICACIÓN DISTRIBUIDA RMI. 30
FIGURA 13. COMMON OBJECT REQUEST BROKER ARCHITECTURE (CORBA). 33
FIGURA 14. CONTENIDO DE UNA REFERENCIA A OBJETO. 37
FIGURA 15. CLASES DEL PERCEPTRON 43
FIGURA 16. REPRESENTACIÓN GRÁFICA DE LOS ATRIBUTOS DE LA CLASE NEURONA .44
FIGURA 17. REPRESENTACIÓN GRÁFICA DE LOS ATRIBUTOS DE LA CLASE CAPA 49
FIGURA 18. REPRESENTACIÓN GRÁFICA DE LOS ATRIBUTOS DE LA CLASE CAPA 54
FIGURA 19. FORMA PRINCIPAL DEL PERCEPTRON NO DISTRIBUIDO 60
FIGURA 20. INICIO DE UN SERVIDOR CORBA 62
FIGURA 21. ASPECTO DEL ASISTENTE PARA CREAR UN NUEVO SERVIDOR 63
FIGURA 22. ARCHIVOS STUB Y SKELETON DEL SERVIDOR 65
FIGURA 23. SELECCIÓN DE CORBA IMPLEMENTATION 65
FIGURA 24. SELECCIÓN DEL OBJETO DE IMPLEMENTACIÓN CORBA 66
FIGURA 25. INTERFAZ DEL SERVIDOR DISTRIBUIDO 71
FIGURA 26. SELECCIÓN DE LA OPCIÓN CLIENTE CORBA 72
FIGURA 27. SELECCIÓN DEL MODULO IDL 73
FIGURA 28. ARCHIVOS STUB Y SKELETON DEL CLIENTE 73

FIGURA 29. SELECCIÓN DEL NOMBRE DE LA INTERFASE....74
FIGURA 30. INTERFAZ DEL CLIENTE DISTRIBUIDO....75
FIGURA 31. RED NEURONAL DISTRIBUIDA....77
FIGURA 32. GRÁFICA DE APRENDIZAJE MAQUINA 1....81
FIGURA 33. GRÁFICA DE TIEMPOS MAQUINA 1....82
FIGURA 34. GRÁFICA DE APRENDIZAJE MAQUINA 2....83
FIGURA 35. GRÁFICA DE TIEMPOS MAQUINA 2....84
FIGURA 36. GRÁFICA DE APRENDIZAJE MAQUINA 3....85
FIGURA 37. GRÁFICA DE TIEMPOS MAQUINA 3....86
FIGURA 38. GRÁFICA COMPARATIVA – TIEMPO DE APRENDIZAJE....87
FIGURA 39. GRÁFICA COMPARATIVA – PORCENTAJE DE APRENDIZAJE....88

ÍNDICE DE TABLAS

Tabla **Descripción** **Pág.**

TABLA 1. VALORES DEL VECTOR DESEADOS 55
TABLA 2. RELACIÓN DE ARCHIVOS Y SU CONTENIDO 57
TABLA 3. CONFIGURACIÓN COMPLETA DE LA RED 76
TABLA 4. CARACTERÍSTICAS DE LOS EQUIPOS 80
TABLA 5. TABLA DE RESULTADOS DE LA MÁQUINA 1 81
TABLA 6. TABLA DE RESULTADOS DE LA MÁQUINA 2 83
TABLA 7. TABLA DE RESULTADOS DE LA MÁQUINA 3 85
TABLA 8. TABLA DE RESULTADOS DE LA RED NEURONAL DISTRIBUIDA 86

ACRÓNIMOS

Acrónimo	Descripción
ANS	Artificial Neuronal Systems
IA	Inteligencia Artificial
DCOM	Distributed COM
CORBA	Common Object Request Broker Architecture
OMG	Object Management Group
ORB	*Object Request Broker*
DII	*Dynamic Invocation Interface*
DSI	*Dynamic Skeleton Interface*
OMA	Object Management Group
GIOP	General Inter-ORB
CDR	Common Data Representation
IOR	Interoperable Object Reference
COM	Component Object Model
LAN	Local Area Network
WAN	Wide Area Network
TCP/IP	Transmission Control Protocol/Internet Protocol
UDP	User Datagram Protocol
IPX/SPX	Internet work Packet Exchange/Sequenced Packet Exchange
NetBIOS	Network Basic Input Output System
JVM	Java Virtual Machine
JRMP	Java Remote Method Protocol
IIOP	Internet Inter-ORB Protocol
MLP	Multilayer Perceptron
DAI	Distributed Artificial Intelligence
MNIST	Base de Datos de imágenes de dígitos manuscritos
LIRA	Limited Receptive Area classifier

PARTE I: ANTECEDENTES

CAPÍTULO 1: INTRODUCCIÓN

El reconocimiento de patrones es una de las áreas más importantes en el estudio de la inteligencia artificial y tiene numerosas aplicaciones prácticas de la vida real. Por otro lado, la combinación de la inteligencia artificial y los sistemas distribuidos dan como resultado lo que se denomina DAI (Distributed Artificial Intelligence) que ofrece sistemas con características importantes para un desempeño óptimo de los sistemas de cómputo.

En el presente trabajo de tesis, se diseña y desarrolla una red neuronal basada en el Perceptron de Rossenblatt de tres capas con alimentación hacia adelante para el reconocimiento de dígitos manuscritos de la base de datos MNIST, la cual cuanta con un conjunto de 60,000 muestras para la fase de entrenamiento y 10,000 para la fase de identificación del proceso de aprendizaje de este tipo de redes. Los parámetros de este tipo de red hacen que su proceso de entrenamiento tarde tiempos considerables para la convergencia hacia un estado de aprendizaje adecuado. Por esta razón se utilizan técnicas de sistemas distribuidos empleando el estándar CORBA como herramienta para el desarrollo de objetos distribuidos y mejorar el tiempo que se consume para la convergencia hacia un estado aceptable. Aunque como se detallara en el presente trabajo una de las características del estándar CORBA es la independencia a los productos en los trabajos, en esta investigación se desarrolla un sistema distribuido utilizando una arquitectura unitaria en todos los puntos involucrados; el sistema operativo de Microsoft Windows XP, el middelware de Borland VisiBroker y el lenguaje de programación C++ utilizando como herramienta de desarrollo Borland C++ Builder. La presente propuesta mejora la arquitectura de tiempo en el proceso de aprendizaje del perceptron debido a las características del proceso ya que se puede obtener un cierto grado de paralelismo en la ejecución del mismo.

Este trabajo de tesis generó un desarrollo de software del perceptron de Rossenblatt de una manera no distribuida y distribuida, teniendo cada uno de estos los mismos parámetros de entrenamiento con la finalidad de tener comparaciones reales en tiempo de aprendizaje.

El diseño de las interfaces del sistema distribuido contempla el contenido de los conceptos en el modelo de comunicación de los sistemas distribuidos, que son cliente y servidor, en cada uno de los módulos del sistema.

La tesis está ordenada de la siguiente forma: en el capítulo 2 se hace referencia al trabajo relacionado y se describen algunos sistemas similares al desarrollado. En el capítulo 3 se describen los diferentes tipos de redes neuronales. En el capítulo 4 se describe el tipo red perceptron, con el cual se desarrollo la aplicación. En el capítulo 5 se describen las características de los sistemas distribuidos. En el capítulo 6 se describe la tecnología distribuida utilizada para el desarrollo de la aplicación, CORBA. En el capítulo 7 se describe el problema que se trata en el presente trabajo de tesis. En el capítulo 8 se presenta el desarrollo de la red neuronal simple. En el capítulo 9 se presenta el desarrollo de la red neuronal distribuida. Finalmente en los capítulos 10,11 y 12 se concretan los resultados, conclusiones, y trabajo a futuro.

CAPÍTULO 2: TRABAJOS RELACIONADOS

El concepto de sistemas de cómputo distribuido en conjunto con otras tecnologías ha provocado que surjan nuevas aplicaciones en los últimos años. La reducción de costos es una de las ventajas principales en el desarrollo de aplicaciones con tecnologías de sistemas distribuidos, esto gracias al aprovechamiento de infraestructuras y componentes estándar. Además, el estándar CORBA (Common Object Request Broker Arcuitecture) [OMG, 2000] vence el problema de heterogeneidad que se presenta en los sistemas desarrollados con el uso de diferentes tecnologías de software, lenguajes de programación, sistemas operativos o hardware.

En [KUSSUL, 2002] Kussul (et. al) desarrollaron un nuevo clasificador de dígitos manuscritos llamado LIRA (*Limited Receptive Area Classifier*) basado en los principios del perceptron de Rosenblatt utilizando la base de datos MNIST y proponiendo cuatro cambios en la estructura del perceptron y de los algoritmos de entrenamiento y reconocimiento. En la modificación del perceptron simple de Rosenblatt se incluyeron 10 neuronas en la capa de reacción. En la primera serie de experimentos utilizaron la regla simple del ganador de la selección. La neurona de la capa de Reacción teniendo la excitación mas alta determina la clase a reconocer, utilizando esta regla obtuvieron un 99.21% de aprendizaje, después de esto modifican la regla de selección del ganador y mejoraron su porcentaje de aprendizaje a un 99.37%. La segunda modificación fue realizada en el proceso de entrenamiento donde proponen que la neurona ganadora tenga excitación, y que el competidor más cercano también tenga excitación. La tercera modificación que proponen es referente con los pesos, donde las conexiones entre la capa asociativa y reacción del perceptron de Rosenblatt pueden ser negativas o positivas y ellos proponen conexiones positivas para las líneas que pertenecen al objeto y conexiones negativas para las líneas que no pertenecen al objeto. La

cuarta modificación fue conectar una neurona a la capa asociativa con neuronas de la capa Sensorial aleatoriamente no seleccionada de toda la capa Sensorial si no de un rectángulo con dimensiones menores. Con lo anterior ellos obtienen un tiempo de entrenamiento de 45 horas generando un 99.37% de aprendizaje.

En [KUSSUL, 2003] Kussul (et. al), proponen un nuevo clasificador neuronal para el reconocimiento de dígitos manuscritos. El clasificador se basa en la extracción de características con la técnica de codificación de permutación. El desempeño del clasificador fue probado con la base de datos MNIST y se obtuvo un porcentaje de error del 0.54%. En su trabajo proponen un nuevo algoritmo, el cual funciona mejor que sus predecesores debido a que sus nuevos elementos (extractor de características y codificador) se incluyen dentro de la estructura de la red. Un dígito manuscrito es la entrada del extractor de características, la salida del extractor de características se aplica a la entrada del codificador, el cual produce un vector binario que se presenta a la capa de Reacción del Perceptron.

En la actualidad el desarrollo de reconocimiento de patrones mediante redes neuronales adaptadas a componentes distribuidos están siendo desarrolladas por diferentes investigadores. [Aldabas, 2002] Emiliano Aldabas, presenta el análisis del reconocimiento de patrones mediante redes neuronales y una aplicación desarrollada en MATLAB 5.0, utilizando un perceptron multicapa entrenado mediante el algoritmo propagación hacia atrás, donde su patrón de imágenes son los dígitos del 0 al 9, el punto y el guión; la red neuronal dispone de 35 entradas debido a que cada dígito está pintado en una matriz de 7 x 5, utilizando 12 muestras distintas de los patrones, los resultados obtenidos, en dicha simulación, han sido casi ideales.

CAPÍTULO 3: REDES NEURONALES

Las redes neuronales artificiales ANS (Artificial Neuronal System) tienen dos direcciones de estudio, la primera como modelo del sistema nervioso y los fenómenos cognitivos, y la segunda como herramienta para resolver problemas prácticos. Por esta razón las redes neuronales son consideradas sistemas de procesamiento (hardware o software), que copian de forma esquemática la estructura neuronal del cerebro humano para tratar de reproducir sus capacidades. Las ANS son capaces de aprender de la experiencia a partir de señales o datos provenientes del exterior, dentro de un marco de computación paralela y distribuida.

3.1 Introducción Biológica

En el año de 1888 Santiago Ramón y Cajal demostraron que el sistema nervioso estaba compuesto por una red de aproximadamente 100 mil millones de células individuales interconectadas (neuronas), que podían presentarse en diversas formas, aunque la mayoría presentaba un aspecto similar al que se muestra en la figura 1, con un cuerpo celular o soma, del que surge un denso árbol de ramificaciones compuesto por dendritas y una fibra tubular denominada axón.

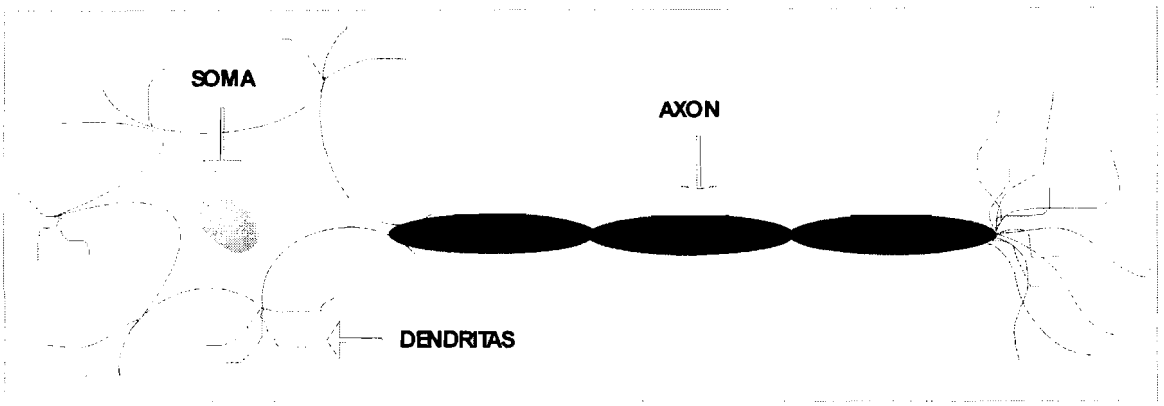


Figura 1. Estructura Neuronal Biológica

Las neuronas funcionalmente constituyen sencillos procesadores de información. Las dendritas poseen un canal de entrada de información, el soma un órgano de cómputo, y el axón un canal de salida.

En los diferentes tipos de neuronas la forma de enviar y recibir información es diferente, es decir, las neuronas intermedias envían información a otras neuronas, las neuronas motoras lo hacen directamente al músculo y las neuronas receptoras o sensoras reciben información del exterior en lugar de recibirla de otras neuronas, [BONIFACIO, 2002].

3.2 Historia de las Redes Neuronales Artificiales

En un principio, los esfuerzos para la construcción de máquinas "inteligentes" estuvieron dirigidos a la obtención de autómatas, en el sentido de máquinas que realizaran alguna función típica de los seres humanos. Hoy en día se continúa estudiando en ésta misma línea, con resultados sorprendentes; existen maneras de realizar procesos similares a los inteligentes y que podemos encuadrar dentro de la llamada Inteligencia Artificial (IA).

A pesar de disponer de herramientas y lenguajes de programación diseñados expresamente para el desarrollo de sistemas inteligentes, existe un

enorme problema que limita los resultados que se pueden obtener debido a que estos sistemas se implementan en computadoras basadas en la filosofía de Von Neumann, y que se apoyan en una descripción secuencial del proceso de tratamiento de la información. Estas máquinas son capaces de realizar tareas mecánicas de forma rápida, como por ejemplo cálculo, ordenación o control, pero son incapaces de obtener resultados aceptables cuando se trata de tareas como reconocimiento de formas, voz, etc. siguiendo la línea secuencial antes expuesta.

Los primeros resultados sobre mecanismos simuladores de la mente humana los obtuvieron Warren McCulloch y Walter Pitts en el año de 1943, y para las dos siguientes décadas los investigadores que obtuvieron diferentes resultados fueron Ashby en 1952, Marvin Minsky y Selfridge en 1961, Block en 1962 y Frank Rossenblatt en el mismo año. Los modelos generados por las investigaciones sobre redes neuronales en la década de los setenta eran muy pobres en sus algoritmos computacionales. En la actualidad, el interés en dichas redes ha vuelto a resurgir debido a la aparición de computadoras digitales mucho más rápidas en las que se pueden realizar simulaciones de redes mucho más completas, así como construir computadoras paralelas o sistemas distribuidos, aunque la razón principal radica en el descubrimiento de nuevas arquitecturas para redes neuronales y algoritmos de aprendizaje mucho más potentes. Estas arquitecturas no se han creado con la idea de crear el funcionamiento humano sino para recibir alguna inspiración acerca de hechos conocidos sobre el funcionamiento del cerebro [RICH, 2002].

3.3 Modelo general de una neurona artificial

Una neurona es un dispositivo simple de cálculo que proporciona una salida a partir de un vector de entrada procedente del exterior o de otras neuronas. Cada neurona consta de los elementos que se muestran en la figura 2, y se listan a continuación:

- *Conjunto de entradas* $x_j(t)$, pueden ser digitales (discretas) o analógicas (continuas).
- *Pesos sinápticos de la neurona* i, w_{ij}, que representan la intensidad de interacción entre cada neurona presináptica j y la neurona postsináptica i.
- *Regla de propagación* $\sigma(w_{ij}, x_j(t))$, que proporciona el valor del potencial postsináptico $h_i(t) = \sigma(w_{ij}, x_j(t))$ de la neurona i en función de sus pesos y entradas. Esta regla de propagación permite obtener, a partir de las entradas y los pesos, el valor del potencial postsináptico h_i de la neurona. La función más habitual de esta regla es lineal y se basa en la suma ponderada de las entradas con los pesos sinápticos $h_{ij}(t) = \sum w_{ij} x_j$.
- *Función de Activación* $F_i(a_i(t-1), h_i(t))$, que proporciona el estado de activación actual $a_i(t) = F_i(a_i(t-1), h_i(t))$ de la neurona i, en función de su estado anterior $a_i(t-1)$ y de su potencial postsináptico actual.
- *Función de salida* $F_i(a_i(t))$, que proporciona la salida actual $y_i(t) = F_i(a_i(t))$ de la neurona i en función de su estado de activación.

Por lo tanto la operación de la neurona i puede expresarse como:

$$y_i(t) = F_i\left(f_i\left|a_i(t-1), \sigma(w_{ij}, x_j(t))\right|\right)$$

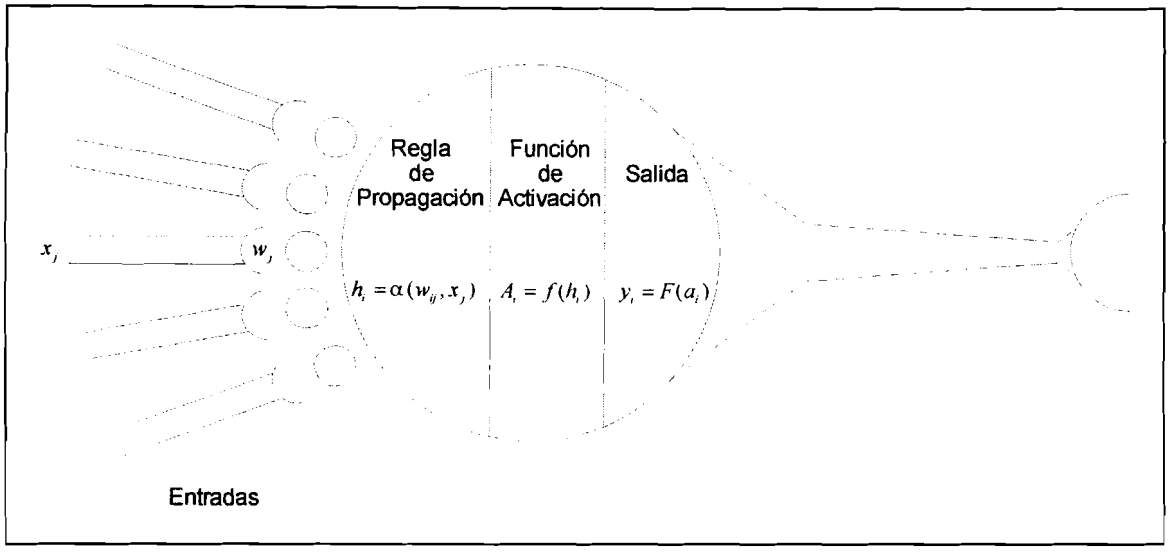


Figura 2. Modelo Genérico de una Neurona Artificial

3.4 Modelo estándar de neurona artificial

En la practica generalmente suele utilizarse un modelo de neurona mas simple que se denomina modelo estándar, dicho modelo considera que la regla de propagación es la suma ponderada y que la función de salida es la identidad. Los elementos de los que consta una neurona estándar se muestran en la figura 3:

- Conjunto de entradas $x_j(t)$ y pesos sinápticos w_{ji}.
- Regla de propagación $h_i(t) = \sigma(w_{ij}, x_j(t)) = \sum w_{ij} x_j$.
- Función de activación $y_i(t) = F_i(a_i(t))$ que representa simultáneamente la salida de la neurona y su estado de activación.

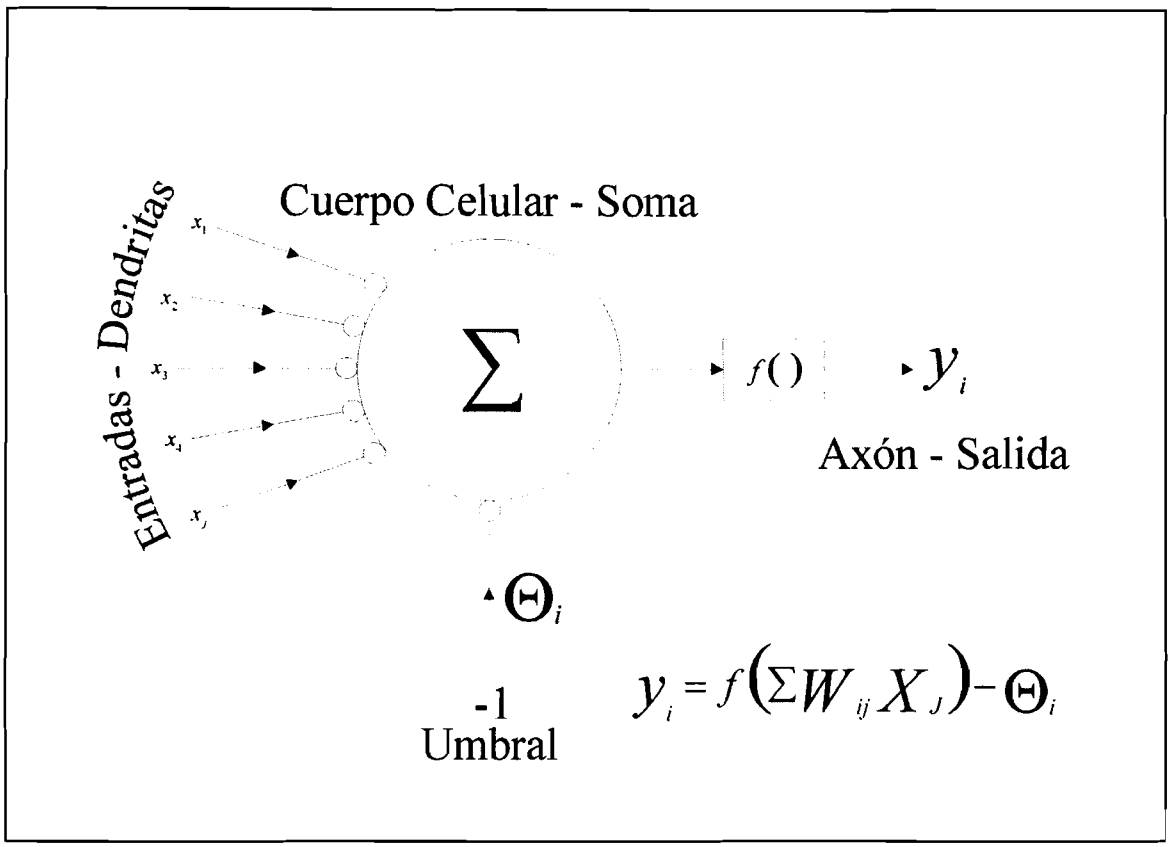


Figura 3. Modelo de una Neurona Artificial Estándar

Al conjunto de pesos de la neurona se le añade un parámetro adicional Θ_i, el cual se denomina umbral y que se resta al potencial postsináptico, la función de activación queda de la siguiente manera:

$$(\Sigma W_{ij} X_j) - \Theta_i$$

En conclusión, el modelo general de la neurona estándar queda de la siguiente forma:

$$y_i(t) = F_i(\Sigma_j W_{ij} X_j) - \Theta_i$$

Si se considera que las entradas de la neurona estándar son digitales $(x_i = \{0,1\})$ y la función de activación es de tipo escalón H definida entre 0 y 1 se obtiene:

$$y_i(t) = H_i(\Sigma_j W_{ij} X_j) - \Theta_i$$

Como $H(x) = 1$ cuando $x \geq 0$, y $H(x) = 0$ cuando $x < 0$, entonces se obtiene:

$$y_i = \begin{cases} 1, si\ \Sigma_j W_{ij} X_j \geq \Theta_i \\ 0, si\ \Sigma_j W_{ij} X_j < \Theta_i \end{cases}$$

Es decir, si el potencial de membrana supera un valor umbral &i (umbral de disparo), entonces la neurona se activa, si no lo supera la neurona no se activa. A este modelo de neurona es llamado perceptron original.

3.5 Estructura de un Sistema Neuronal Artificial

Un sistema neuronal artificial está compuesto por millones de neuronas agrupadas en capas constituyendo un sistema con funcionalidad propia. Varias capas constituyen una red neuronal y esta red neuronal a su vez está constituida por interfaces de entradas y salidas, más los módulos convencionales adicionales y necesarios para constituir un sistema global de proceso. La figura 4, muestra una estructura jerárquica de un sistema artificial neuronal.

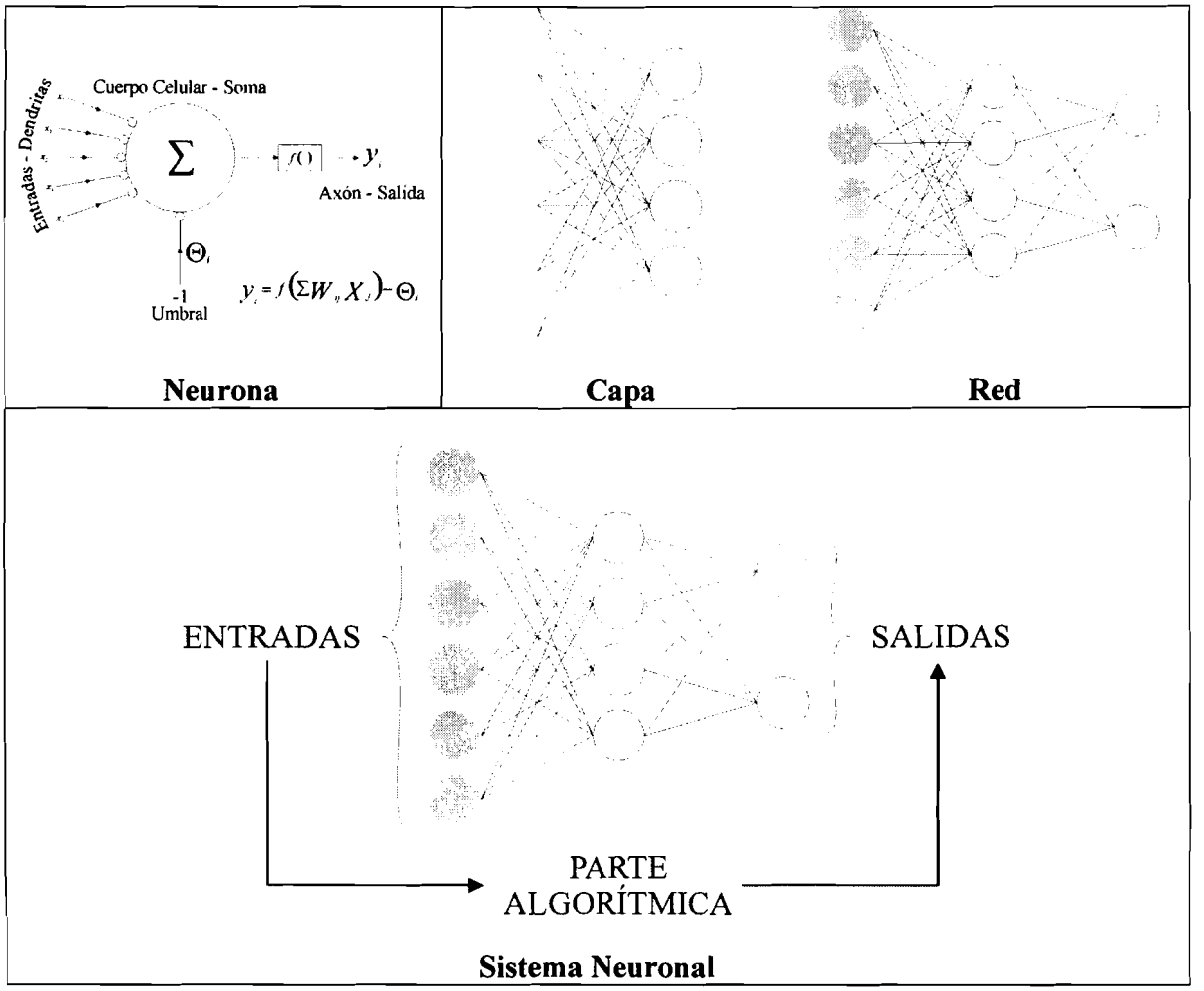


Figura 4. Estructura Jerárquica de un Sistema Basado en el Sistema Artificial Neuronal

Un sistema neuronal esta constituido por los siguientes elementos:

- Un conjunto de procesadores elementales o neuronas artificiales.
- Un patrón de conectividad o arquitectura.
- Una dinámica de activaciones.
- Una regla o dinámica de aprendizaje.
- El entorno donde opera.

3.6 Modelos de Redes Neuronales Artificiales

En esta sección, se describirán diferentes modelos de redes neuronales para posteriormente dedicar un análisis profundo a la red que se utiliza en el presente trabajo de tesis.

Existen diferentes características que nos pueden ayudar a clasificar un modelo neuronal, sin embargo, solo dos de ellas se consideran importantes, la primera es por el tipo de aprendizaje y la segunda la arquitectura de la red.

La clasificación por tipo de aprendizaje se muestra en la figura 5, esta clasificación se subdivide en híbridos, supervisados, no supervisados y reforzados. Cada una de estas subclasificaciones se puede dividir de acuerdo a la topología de red.

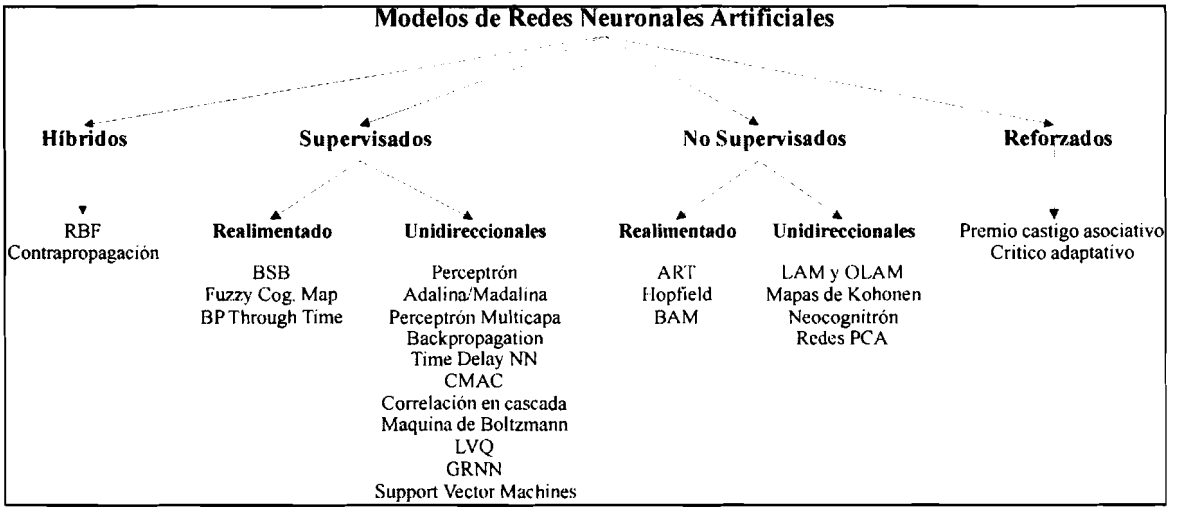


Figura 5. Clasificación de los ANS por el tipo de Aprendizaje y Arquitectura

De los modelos del grupo de redes neuronales supervisadas trataremos especialmente los casos del Perceptron sirnple y Adalina. Es importante destacar que estos modelos representan la historia de las redes neuronales, debido a su gran interés desde los años en que fueron propuestas.

3.6.1.1 Perceptron Simple

Introducido por Frank Rossenblatt a finales de los cincuenta, la estructura se inspira en las etapas de procesamiento de los sistemas sensoriales de los animales, en los cuales la información va atravesando capas de neuronas sucesivamente para realizar un procesamiento progresivo de alto nivel.

El Perceptron simple es un modelo unidireccional, compuesto por dos capas de neuronas, una de entradas o sensorial y otra de salidas como lo muestra la figura 6. La operación puede ser expresada de la siguiente manera con n neuronas de entrada y m de salida [BONIFACIO, 2002]:

$$y_i(t) = f\left(\sum w_{ij} x_j - \Theta_i\right), \forall_i 1 \le i \le 0$$

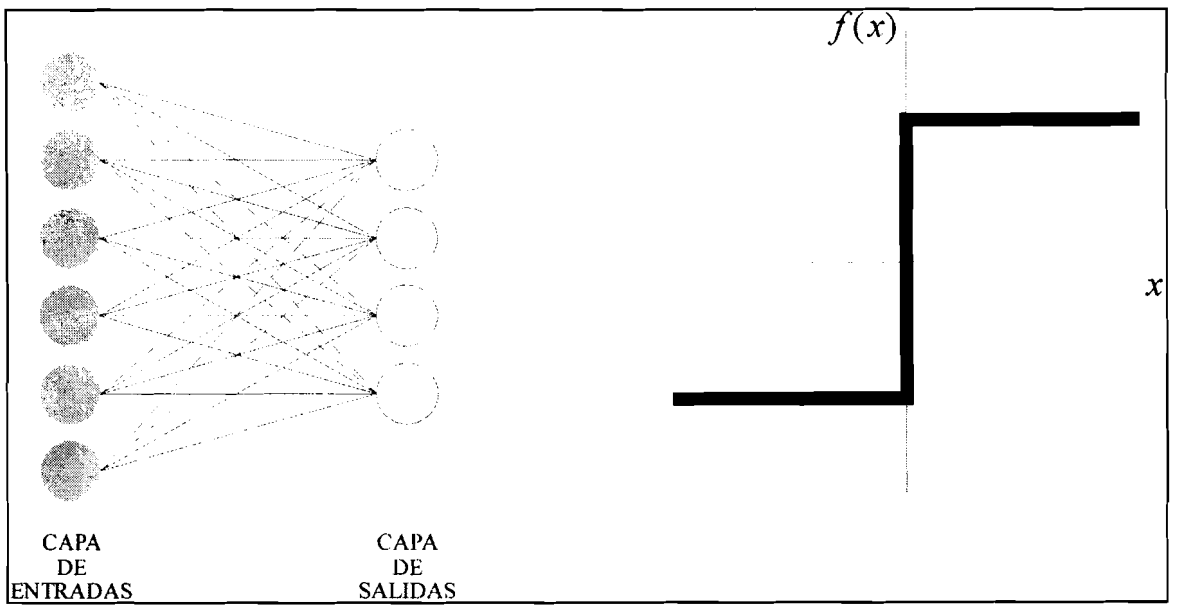


Figura 6. Perceptron Simple

3.6.1.2 Adalina

Este modelo fue introducida por Widrow en 1959, cuyo nombre proviene de ADAptive Linear Neuron. Modelo que utiliza una neurona semejante a la del Perceptron, la diferencia es que Adalina obtiene una respuesta lineal como se representa en la figura 7, cuyas entradas pueden ser continuas. Además Adalina incorpora un parámetro denominado umbral que no es un umbral de disparo como en el perceptron, sino un parámetro que proporciona un grado de libertad adicional.

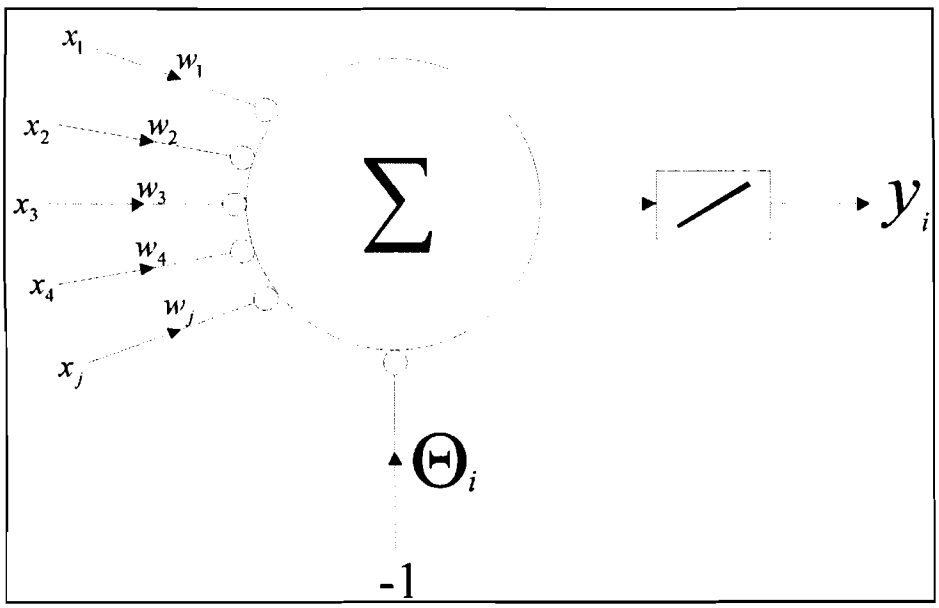


Figura 7. Neurona lineal de la Red Adalina

La diferencia más importante entre el Perceptron simple y la Adalina reside en la regla de aprendizaje que implementan.

La operación del modelo de Adalina queda de la siguiente forma:

$$y_i(t) = \sum w_i x_j - \Theta_i \forall_i 1 \le i \le m$$

3.7 Ventajas de las Redes Neuronales

Debido a su constitución y a sus fundamentos, las redes neuronales artificiales presentan un gran número de características semejantes a las del cerebro. Por ejemplo, son capaces de aprender de la experiencia, de generalizar de casos anteriores a nuevos casos, de abstraer características esenciales a partir de entradas que representan información irrelevante, etc. Esto hace que ofrezcan numerosas ventajas y que este tipo de tecnología se esté aplicando en múltiples áreas. Algunas de las ventajas son las siguientes: aprendizaje adaptativo, auto-organización, tolerancia a fallos, y operación en tiempo real entre otras.

3.8 Aplicaciones de las Redes Neuronales

Las redes neuronales son una tecnología computacional emergente que puede utilizarse en un gran número y variedad de aplicaciones, que van desde los temas comerciales hasta aspectos militares.

Hay muchos tipos diferentes de redes neuronales, cada uno tiene una aplicación particular apropiada. Separándolas por las distintas disciplinas y enunciando sólo algunas de las posibles aplicaciones en las que pueden trabajar, algunos ejemplos son los siguientes: en *biología* se utilizan para comprender más acerca del cerebro y otros sistemas; en *empresas* para el reconocimiento de caracteres escritos, identificación de candidatos para posiciones específicas, optimización de plazas, horarios en líneas de vuelo, explotación de bases de datos, evaluación de probabilidad de formaciones geológicas y petrolíferas y síntesis de voz desde texto; en *medio ambiente* para analizar tendencias, patrones y previsión del tiempo; en *finanzas* para prevenir la evolución de los precios, valoración del riesgo de los créditos, identificación de falsificaciones e interpretación de firmas; en *manufactura* para la automatización de procesos con el uso de robots industriales y sistemas de control (visión artificial y sensores de presión, temperatura, gas, etc.) control de producción en líneas de proceso,

inspección de calidad, filtrado de señales; en *medicina* con analizadores del habla para la ayuda de audición de sordos profundos, diagnóstico y tratamiento a partir de síntomas y/o de datos analíticos (encefalograma, etc.), monitorización en cirugía, predicción de reacciones adversas a los medicamentos, lectoras de Rayos X y entendimiento de la causa de ataques epilépticos; en aspectos *bélicos* para la clasificación de las señales de radar, creación de armas inteligentes y optimización del uso de recursos escasos.

Las aplicaciones de las redes neuronales se pueden dividir en las siguientes categorías de acuerdo a la complejidad del problema y del comportamiento de la red. Estas complejidades pueden estar sintetizadas en reconocedores de patrones y memorias asociativas, transformadores de patrones y derivadores dinámicos [RICH, 2002].

CAPÍTULO 4: PERCEPTRON

La red neuronal Perceptron es una de las herramientas centrales del presente trabajo de tesis, esta parte del documento se dedicará a la descripción detallada del modelo de red. Como ya se mencionó en el capítulo anterior el Perceptron fue descubierto por Frank Rossenblatt en la década de los cincuentas. El primer diseño del Perceptron se denominó Perceptron Simple, inspirado en los sistemas sensoriales de los animales como por ejemplo el de la visión. El segundo modelo del Perceptron fue creado a finales de la década de los ochenta, agregándole capas intermedias.

4.2 Función del Perceptron

El perceptron simple es un modelo unidireccional, compuesto por dos capas de neuronas, una de entradas y otra de salidas. La estructura de operación de este tipo de red es la siguiente:

$$y_i(t) = f\left(\sum w_{ij} x_j - \Theta_i\right), \forall_i 1 \le i \le 0$$

La estructura del perceptron simple se muestra en la Figura 6, donde las neuronas de entrada no realizan ningún procesamiento de datos, solamente envían la información. La función de activación de las salidas es de tipo escalón.
Un Perceptron simple imita una neurona tomando la suma ponderada de sus entradas y enviando a la salida un 1 si la suma es más grande que un valor umbral ajustable, la figura 8, muestra la representación de una neurona y de un perceptron.

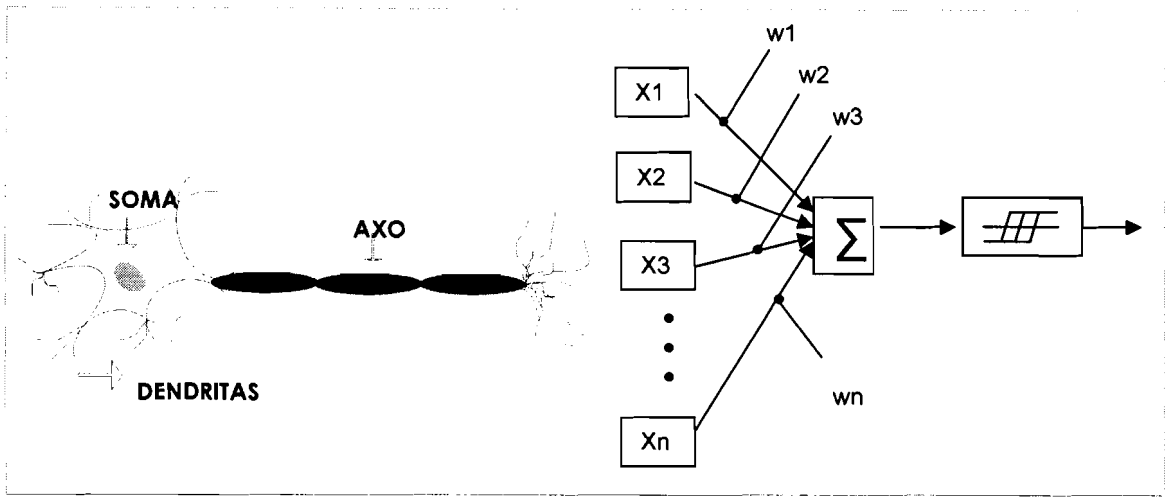


Figura 8. Neurona y Perceptron

Las entradas $(x_1, x_2, x_3, \ldots, x_n)$ y los pesos de conexión $(w_1, w_2, w_3, \ldots, w_n)$ que se muestran en la figura anterior normalmente son valores reales. Si la presencia de alguna característica x_i tiende a causar la activación del perceptron, el peso w_i será positivo; si la característica x_i inhibe al perceptron, el peso w_i será negativo. El perceptron se compone de los pesos, el procesador de sumas y el procesador del umbral ajustable. El aprendizaje es un proceso en el cual se modifican los valores de los pesos y del umbral. Resulta conveniente considerar el umbral exactamente como otro peso w_0, ya que se puede ver este peso como la propensión a que se dispare el perceptron independientemente de sus entradas, dicha consideración se muestra en la figura 9, donde el perceptron se dispara si la suma ponderada es mayor de cero [RICH, 2002].

Un Perceptron calcula una función binaria de su entrada, sin embargo, algunos se pueden combinar para calcular funciones más complejas, entrenándose mediante pares de entrada/salida, hasta que aprendan a calcular la función correcta. La propiedad en el aprendizaje del perceptron que resulta interesante es que cualquiera que sea el cálculo, el perceptron puede aprender a calcularlo.

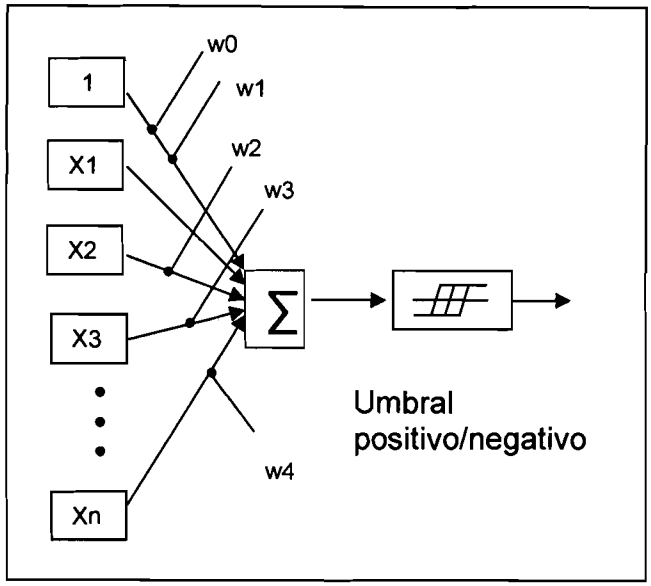


Figura 9. Perceptron con umbral ajustable, implementado como un peso adicional

4.3 Algoritmo de aprendizaje del Perceptron Simple

El algoritmo de aprendizaje del perceptron es un algoritmo de búsqueda, que comienza en un estado inicial aleatorio y termina encontrando un estado de solución. El espacio de búsqueda simplemente consiste en todas las posibles asignaciones de valores reales a los pesos del perceptron y a la estrategia de búsqueda es un descenso por el gradiente.

La importancia del perceptron radica en su carácter de dispositivo entrenable, debido a que el algoritmo introducido por Frank Rossenblatt permite que el Perceptron determine automáticamente los pesos sinápticos que clasifican un conjunto de patrones etiquetados.

El teorema de convergencia del perceptron de Frank Rossenblatt garantiza que el Perceptron encontrará un estado de solución, es decir aprenderá a clasificar cualquier conjunto de entradas linealmente separables; en otras palabras

el teorema muestra que en el espacio de pesos no existen mínimos locales que no se correspondan con el mínimo global [RICH, 2002].

Los programas de clasificación de patrones, a menudo combinan diferentes características para determinar la categoría correcta en la que se debe situar un estimulo dado. Algunas veces cuando se diseñan estos programas es difícil conocer a priori los pesos que se deben conceder a cada una de las características que se van a utilizar. Una manera de saber si se está utilizando el peso necesario a cada una de ellas, es iniciando con alguna estimación de los valores correctos y, entonces, permitir que el programa los modifique en base a su propia experiencia. Las características adecuadas para predecir éxitos globales, aumentaran sus pesos y las que no los disminuirán [RICH, 2002].

Un ejemplo de la aplicación de este algoritmo se basa en la figura 10, en la que se presenta un problema de clasificación de patrones linealmente separables, ya que se puede dibujar una línea recta que separa una clase de objetos de otra. Se requiere entrenar un Perceptron de tal modo que origine un 1 en la salida si piensa que la entrada pertenece a los objetos blancos y 0 si piensa que la entrada pertenece a los objetos negros.

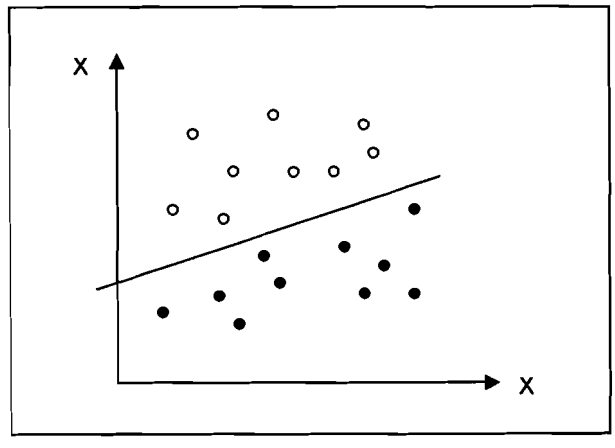


Figura 10. Clasificación de patrones linealmente separables

El cálculo que realizará el perceptron para poder resolver el problema planteado, se define $g(x)$ como la función de suma ponderada, $o(x)$ como la función de salida, entonces dichas funciones son precisadas mediante:

$$g(x)=\sum_{i=0} w_i x_i \qquad o(x)=\begin{Bmatrix}1, g(x)>0\\ 0, g(x)<0\end{Bmatrix}$$

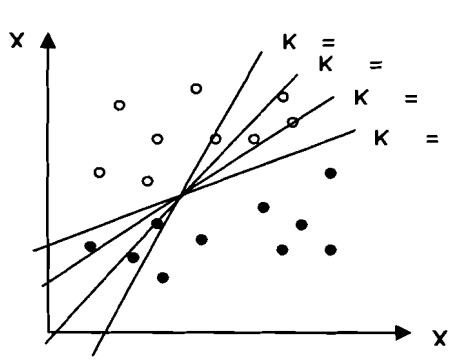
como en el problema planteado solo se tienen dos entradas x_1 y x_2 que son los objetos blancos y negros, entonces:

$$g(x)=w_o+w_1x_1+w_2x_2$$

Tomando en cuenta que la función $g(x)$ es igual a 0, entonces el perceptron no podrá decidir si debe o no dispararse, entonces un cambio en las entradas podría dar lugar a que el dispositivo decida por uno de los dos caminos.
Teniendo $g(x)=0$ se obtiene una ecuación de una recta:

$$x_2=\frac{w_1}{w_2}x_1-\frac{w_0}{w_2}$$

Esta ecuación se determina en su totalidad por los pesos w_0, w_1 y w_3, por lo que si un vector de entrada se sitúa hacia uno de los lados de la recta la salida del perceptron dará 1; y al contrario, si se desplaza hacia el otro lado la salida dará 0. Se dice que un perceptron funciona correctamente si separa dos casos de entrenamiento mediante una recta, dicha recta determina la superficie de decisión. La figura 11, muestra el aprendizaje del perceptron a partir del problema planteado en la figura anterior.



k	W0	W1	W2
10	0.41	-0.17	0.14
100	0.22	-0.14	0.11
300	-.010	-0.08	0.07
635	-0.49	-0.10	0.14

k = Número de pasos a través del entrenamiento

Figura 11. Perceptron Aprendiendo a Clasificar Objetos

Los pasos que se realizan para poder llevar a cabo el algoritmo de entrenamiento de este Perceptron, en general son los siguientes:

1. Crear un Perceptron con $n+1$ entradas y $n+1$ pesos, donde una entrada extra llamada X_0 siempre esta puesta en 1.
2. Inicializar los pesos $(w_0, w_1, ..., w_n)$ con valores reales aleatorios.
3. Iterar a través del conjunto de entrenamiento, recogiendo todos los ejemplos sin clasificar que se obtienen con el conjunto de pesos que se tienen.
4. Si todos los ejemplos se clasifican correctamente, la salida son los pesos y se termina.
5. Si no ocurre así, calcular el vector suma S de los vectores sin clasificar, donde cada vector es de la forma $(x_0, x_1, ..., x_n)$. Al calcular la suma añadir a S el vector $\vec{x}$, si $\vec{x}$ es una entrada para la que el perceptron falla al no dispararse correctamente, y añade $-\vec{x}$ si $\vec{x}$ es una entrada para la que le perceptron se dispara incorrectamente, multiplicar la suma por un vector escalar η.

6. Modificar los pesos $(w_0, w_1, ..., w_n)$ añadiendo los elementos del vector S a éstos, volver al paso 3.

CAPÍTULO 5: SISTEMAS DISTRIBUIDOS

Un sistema distribuido es una colección de computadoras autónomos enlazados por una red y soportados por aplicaciones que hacen que la colección actué como un servicio integrado [COULOURIS, 2001]. Es un sistema en el que los componentes hardware y/o software ubicados en computadores en red, se comunican y coordinan sus acciones intercambiando mensajes.

5.1 Historia de Sistemas Distribuidos

Una de las primeras caracterizaciones de un sistema distribuido fue realizada por Enslow en 1978 atribuyéndole las siguientes propiedades: está compuesto por varios recursos informáticos de propósito general, tanto físicos como lógicos, que pueden asignarse dinámicamente a tareas concretas, estos recursos están distribuidos físicamente y funcionan gracias a una red de comunicaciones; hay un sistema operativo de alto nivel que unifica e integra el control de los componentes; la distribución es transparente permitiendo que los servicios puedan ser solicitados especificando simplemente su nombre (no su localización), el funcionamiento de los recursos físicos y lógicos está caracterizado por una autonomía coordinada.

5.2 Características de los Sistemas Distribuidos

Los sistemas distribuidos presentan diversas características que se describen a continuación: cada computadora tiene su propia memoria y su propio sistema operativo, que controla los recursos locales y remotos, proporciona un sistema abierto lo cual genera facilidades de cambio y crecimiento, utiliza diversos medios de comunicación: redes, protocolos, dispositivos, etc., tiene capacidad de procesamiento en paralelo, etc.

5.3. Desventajas de un Sistema Distribuido

Las principales desventajas de los sistemas distribuidos son: la seguridad de los datos, la pérdida de mensajes, saturación en el tráfico, requerimientos de mayor control, la velocidad de propagación en ocasiones es muy lenta, los costos, la administración es más compleja, entre otras.

5.4 Aplicaciones de Sistemas Distribuidos

Los sistemas distribuidos pueden ser aplicados en diversas áreas de trabajo; tales como: sistemas comerciales, redes WAN, Aplicaciones Multimedia, etc.

5.5 Herramientas para construir objetos distribuidos

Dentro de la lista de modelos de objetos distribuidos se encuentran como exponentes más representativos COM, DCOM, Java RMI y CORBA. Utilizando estas herramientas el desarrollador no gestiona mensajes o llamadas a procedimientos individuales sino que obtiene referencias a objetos, logrando llamar a métodos para obtener y establecer propiedades.

5.5.1 DCOM (Distributed Component Object Model)

DCOM surge de COM (Component Object Model) para soportar comunicación entre objetos en distintas computadoras en una LAN (Local Área Network), WAN (Wide Área Network), o incluso en Internet. Como DCOM es una evolución lógica de COM, se pueden utilizar los componentes creados en aplicaciones basadas en COM, y trasladarlas a entornos distribuidos. DCOM maneja detalles muy bajos de protocolos de red, es por esto que un cliente llama a

los métodos del componente sin tener que preocuparse de niveles más complejos, debido a que DCOM proporciona este tipo de comunicación de una forma transparente interceptando las llamadas del cliente y reenviándolas al componente que está en otro proceso. Las librerías de DCOM proporcionan servicios orientados a objetos a los clientes y componentes, utilizando RPC (Remote Procedure Call) y un proveedor de seguridad para generar paquetes de red estándar que entienda el protocolo estándar de DCOM.

Como una extensión de COM, DCOM es completamente independiente del lenguaje. Virtualmente se puede utilizar cualquier lenguaje para crear componentes COM, y estos componentes puede ser utilizado por muchos más lenguajes y herramientas. Java, Microsoft Visual C++, Microsoft Visual Basic, Delphi, PowerBuilder, y Micro Focus COBOL interactúan perfectamente con DCOM. DCOM puede utilizar cualquier protocolo de transporte, como TCP/IP (Transmission Control Protocol/Internet Protocol), UDP (User Datagram Protocol), IPX/SPX (Internetwork Package Exchange/Sequenced Package Exchange) y NetBIOS (Network Basic Input Output System) y proporciona un marco de seguridad a todos estos protocolos [Microsoft, 1997]. Sin embargó, al ser una distribución de Microsoft, no está estandarizada ni abierta como lo está CORBA, lo que representa una desventaja.

5.5.2 Java RMI (Remote Method Invocation)

Un sistema RMI es un mecanismo que habilita un objeto sobre una máquina virtual de java (JVM.- Java Virtual Machine) para invocar métodos de un objeto en otra JVM. Las aplicaciones RMI están compuestas de dos programas separados: un servidor y un cliente. Una aplicación servidor común crea determinados objetos remotos, realiza referencias para accederlos y espera a que los clientes invoquen los métodos sobre estos objetos. Una aplicación cliente tiene una referencia a uno o más objetos en el servidor para poder invocar sus métodos. RMI proporciona los

mecanismos para que el servidor y el cliente se comuniquen e intercambien información.

Las aplicaciones utilizan uno o dos mecanismos para obtener referencias a objetos remotos. Una aplicación registra sus objetos con la facilidad de denominación del RMI, o bien, la aplicación puede pasar y regresar la referencia a los objetos como parte de su operación normal. Los detalles de comunicación entre objetos remotos se manejan por el RMI; para el programador, la comunicación remota es análoga a la invocación de cualquier método.

Dado que RMI permite que un cliente pase objetos a objetos remotos, RMI proporciona los mecanismos necesarios para cargar el código de un objeto, así como también de transmitir sus datos. RMI soporta los protocolos JRMP (Java Remote Method Protocol) e IIOP (Internet Inter-ORB Protocol). JRMP es un protocolo diseñado específicamente para RMI; IIOP es el protocolo estándar para la comunicación entre objetos CORBA. RMI sobre IIOP permite a los objetos java comunicarse con objetos CORBA que pueden ser escritos en un lenguaje diferente a java.

La Figura 12, representa una aplicación distribuida RMI que utiliza el registro para obtener una referencia a un objeto remoto. El servidor llama al registro para asociar un nombre con un objeto. El cliente busca el objeto por su nombre en el registro del servidor y entonces invoca alguno de sus métodos. La ilustración también muestra que el sistema RMI utiliza un servidor Web para cargar códigos de clases escritas en java, del servidor al cliente y del cliente al servidor, para los objetos cuando es necesario [Sun, 2005].

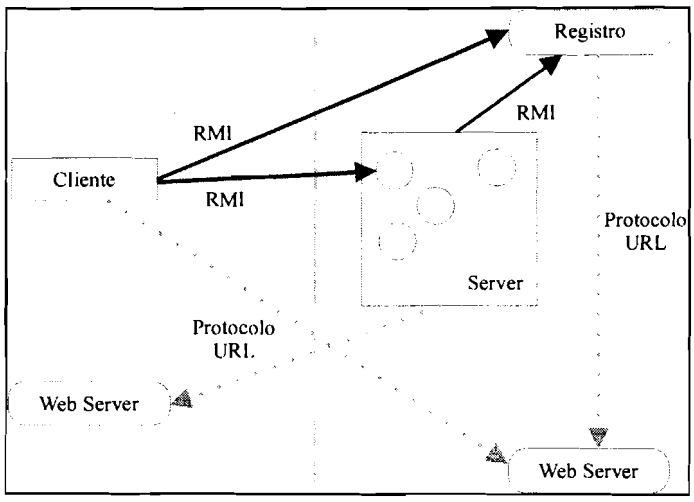


Figura 12. Aplicación distribuida RMI.

CAPÍTULO 6: CORBA

6.1 CORBA

Los problemas del cómputo distribuido heterogéneo hacen que el desarrollo y el mantenimiento de las aplicaciones de red sea una tarea difícil, debido a los numerosos detalles de bajo nivel que se deben considerar. Con el propósito de resolver esta problemática CORBA proporciona las abstracciones y los servicios para desarrollar aplicaciones distribuidas portables sin preocuparse de los detalles de bajo nivel. Para esto OMG (Object Management Group) usa dos modelos relacionados para describir como interactúan los objetos distribuidos y como se puede especificar esta interacción de forma independiente a la plataforma. El Modelo de Objetos define como se describen las interfaces de los objetos distribuidos a través de los entornos heterogéneos, los clientes utilizan los servicios de un objeto para emitir peticiones, y el modelo de referencias caracteriza las interacciones entre estos objetos, proporcionando categorías de interfaz que son agrupaciones generales de interfaces de objetos enlazadas por un ORB.

6.1.1 Especificaciones y terminología CORBA

La especificación de CORBA [Henning, 2002], [OMG, 2000], es desarrollada por el OMG en donde se proporciona un conjunto equilibrado de abstracciones flexibles y servicios concretos necesarios para proporcionar soluciones prácticas a los problemas asociados con entornos distribuidos heterogéneos. La independencia de CORBA al lenguaje de programación, a la plataforma de computación y a los protocolos de red, lo hacen adecuado para el desarrollo de nuevas aplicaciones y para su integración en sistemas distribuidos ya existentes.

Como todas las tecnologías, CORBA tiene una terminología propia que es necesario comprender, a continuación, se muestra una lista de los términos más importantes:

- Un **objeto CORBA** es una entidad "virtual" localizada por un ORB (*Object Request Broker*) y que acepta peticiones de clientes.
- Un **objeto destino**, en el contexto de una petición CORBA, es el objeto CORBA al que se le hace la petición.
- Un **cliente** es una entidad que hace una petición sobre un objeto CORBA.
- Un **servidor** es una aplicación en la que residen uno o más objetos CORBA.
- Una **petición** es una invocación de una operación a un objeto CORBA realizada por un cliente.
- Una **referencia a objeto** es un manejador que se usa para identificar, localizar y dar la dirección de un objeto CORBA.
- Un **sirviente** es una entidad de un lenguaje de programación que implementa uno a más objetos CORBA.

El flujo de las peticiones como se ilustra en la **¡Error! No se encuentra el origen de la referencia.** Surge de la aplicación cliente, atraviesa el ORB y llega hasta la aplicación servidora de la siguiente manera:

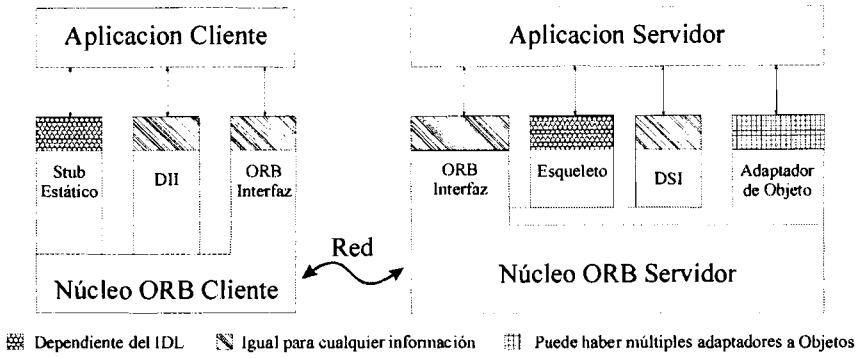


Figura 13. Common Object Request Broker Architecture (CORBA).

1. El cliente realiza peticiones usando *stubs (interfaz* entre la aplicación y el cliente ORB*)* estáticos o usando la DII (*Dynamic Invocation Interface*). En cualquier caso el cliente dirige sus peticiones al núcleo del ORB enlazado con sus procesos.
2. El ORB del cliente transmite sus peticiones al ORB enlazado con la aplicación servidora.
3. El ORB del servidor redirige la petición al adaptador de objetos que ha creado el objeto destino.
4. El adaptador de objetos dirige la petición al servidor que implementa al objeto destino. Así como el cliente, el servidor puede usar esqueletos estáticos o la DSI (*Dynamic Skeleton Interface*).
5. El servidor devuelve su respuesta a la aplicación cliente.

Para realizar una petición y obtener respuesta, son necesarios los componentes de CORBA que se describen a continuación:

1. *Lenguaje de definición de interfaces* (IDL).- permite la definición de interfaces independientes de cualquier lenguaje de programación. Una interfaz ofrecida por un objeto destino debe ser conocida por un cliente para que este pueda llamar sus operaciones.

2. Mapeo de lenguajes.- especifica como se traduce el IDL a los distintos lenguajes de programación.
3. *Adaptador de objeto*.- es un objeto interpuesto que permite a un usuario llamar peticiones de un objeto de forma transparente.
4. *Protocolos Inter-ORB*.- es una arquitectura que permite la interoperabilidad entre distintos ORB.

En la invocación de peticiones los clientes manipulan objetos enviando mensajes, el ORB envía un mensaje a un objeto cada vez que el cliente llama una operación. Para enviar un mensaje a un objeto, el cliente debe tener una referencia de ese objeto. La referencia al objeto actúa como un manejador que identifica de forma única al objeto destino y encapsula toda la información necesaria para que el ORB pueda mandar el mensaje al destino correcto. Las características de la invocación de peticiones son: la transparencia de localización, la transparencia del servidor, la independencia del lenguaje, implementación, arquitectura, sistema operativo y protocolo de transporte [Henning, 2002].

6.1.2 El lenguaje IDL

IDL no es un lenguaje de programación, si no un lenguaje descriptivo. Como su nombre lo indica se usa para describir interfaces que son tablas de métodos asociados en las que se especifica el nombre de cada uno de ellos, los parámetros que precisan y el tipo de valor de retorno. IDL es un lenguaje neutro, aunque su sintaxis es parecida a la de C/C++ y Java. Una interfaz IDL es, por lo tanto, una especie de contrato entre dos partes: un cliente y un servidor.

Conceptualmente un IDL es similar a construcciones habituales como la definición de una clase abstracta en C++ o una interfaz en Java. Las interfaces IDL se agrupan formando módulos. Un módulo es la construcción principal en un

IDL, existiendo uno por cada archivo de definición. Un módulo IDL es tan sólo una enumeración de los elementos que existen dentro de él.

Para construir una aplicación utilizando un IDL existen herramientas conocidas como compiladores IDL que traducen la descripción IDL a un determinado lenguaje, generando módulos, clases o interfaces. Un compilador IDL es dependiente del lenguaje, es decir, existen compiladores que traducen IDL a java, C++, COBOL, PASCAL, etc.

6.1.3 Stubs y Skeletons

Disponiendo del módulo IDL, el siguiente paso será la compilación, obteniendo como producto dos módulos de código dependientes del lenguaje: un stub y un skeleton.

Un skeleton es la interfaz entre una aplicación servidora y el ORB. Entrega las solicitudes procedentes del ORB a la implementación del objeto CORBA.

Un stub es la interfaz entre la aplicación y el cliente ORB. Su propósito es lograr que la petición de un cliente llegue hasta el ORB. Se logra con el acoplamiento entre el lenguaje de programación en que se escribe el cliente y el ORB. El stub crea y expide las solicitudes del cliente.

6.1.4 ORB

La función del ORB es conectar de forma transparente a los clientes con los servidores, que pueden ser desarrollados con distintos lenguajes de programación, ejecutarse sobre plataformas distintas y funcionar con diferentes sistemas operativos. Esto significa que pueden existir diferencias en tipos de datos, el orden de los parámetros en una llamada, el orden de los bytes en una palabra, etc. La tarea del ORB es efectuar los procesos conocidos como

marshaling, la nombrada traducción a un formato neutro, y unmarshaling, el paso inverso.

En caso de que el método invocado devuelva un valor de retorno, la función de ORB del cliente y servidor se invierte. El servidor realiza el marshaling de dicho valor y lo envía al ORB del cliente, que será el que realice el unmarshaling y, finalmente, proporcione el valor en formato nativo.

6.1.5 GIOP

En las últimas versiones del estándar CORBA se introdujo una arquitectura de interoperabilidad general para el ORB denominado GIOP (General Inter-ORB Protocol). GIOP es un protocolo que especifica una sintaxis de transferencia y un conjunto estándar de formatos de mensajes, para permitir que los ORB desarrollados independientemente se puedan comunicar utilizando cualquier conexión de transporte. El Protocolo IIOP define un conjunto de reglas para el formato de datos denominado CDR (Common Data Representation) utilizados en el intercambio de mensajes GIOP sobre redes que utilicen el protocolo TCP/IP.

6.1.2 Referencias a objetos

Con el objetivo de continuar con la interoperabilidad de los ORB es necesario estandarizar los formatos de las referencias a objetos. Las referencias a objetos son opacas para las aplicaciones, pero contienen información que necesitan los ORB para establecer comunicación entre los objetos. El formato de referencia a objeto estándar, denominado Referencia de Objeto Ínter operable, IOR (Inter operable Object Reference) es flexible para almacenar información de casi cualquier IIOP imaginable. Las referencias a objetos son análogas a los apuntadores de instancias a clase de C++, pero pueden denotar objetos

implementados en procesos distintos (posiblemente en otras máquinas), así como objetos implementados en el mismo espacio de direcciones del cliente, además de que son la única forma que tiene el cliente para acceder a los objetos destino. Existen diversos métodos para que los clientes puedan adquirir una referencia de objeto pero la más frecuente es recibirla como respuesta a una llamada de una operación. La Figura 14, muestra una visión conceptual del contenido de una referencia a objeto (IOR).

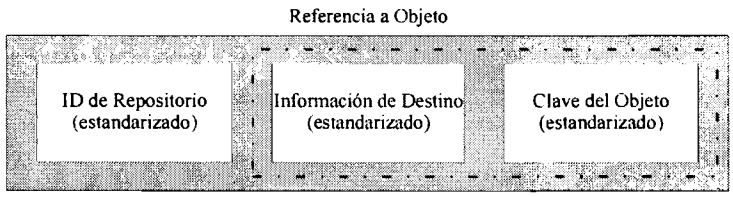


Figura 14. Contenido de una referencia a objeto.

6.3 Ventajas de CORBA

Las ventajas de CORBA frente a sus dos más claros competidores son bastantes. Los servidores y clientes CORBA pueden desarrollarse usando prácticamente cualquier lenguaje de programación, desde los más utilizados en la actualidad como ocurre con Java o C++, hasta los veteranos pero aún muy presentes, como es el caso de COBOL. Existen servicios CORBA disponibles para una treintena de plataformas, desde Dos o Linux hasta los MainFrames pasando, por todas las versiones de Windows. Se trata además de una tecnología que lleva más de una década depurándose y evolucionando.

CAPÍTULO 7: PLANTEAMIENTO DEL PROBLEMA

7.1 Objetivos Generales y Específicos

7.1.1 Objetivos Generales

Crear un sistema distribuido de aprendizaje de imágenes utilizando redes neuronales y el estándar CORBA, con el objetivo de obtener el control general de la distribución de cada una de las tareas.

7.1.2 Objetivos específicos

- Desarrollar una red tipo Perceptron, la cual permita generar las fases de aprendizaje y reconocimiento.
- Analizar tecnologías Middleware.
- Diseñar y configurar una red para satisfacer las necesidades del sistema distribuido.
- Diseñar el sistema distribuido estableciendo las funciones necesarias de las interfaces IDL.
- Mantener el sistema distribuido de una manera dinámica, es decir, éste trabajará con el número de clientes disponibles al momento.

7.2 Hipótesis que sustenta el trabajo

El uso de sistemas inteligentes en la actualidad se generaliza en diversas aplicaciones en unión con sistemas distribuidos y se considera que se pueden incrementar las potencialidades y aprovechar las ventajas de una y otra tecnología en busca de mejorar las soluciones de diversas aplicaciones.

Las fases de aprendizaje y reconocimiento de imágenes de una red neuronal es más rápido si se realiza a través de un sistema distribuido, y si éste usa tecnologías independientes de la plataforma como CORBA se facilita el desarrollo y mantenimiento del sistema.

El tipo de Red Perceptron nos permite obtener un proceso de aprendizaje robusto y adaptativo.

7.3 Importancia del tema

El realizar un sistema distribuido aplicado al aprendizaje de imágenes en un principio puede llevar a pensar que es una aplicación simple por el hecho en que el término imágenes esta generalizado. Pero si a esto le damos un enfoque real se podrían obtener aplicaciones relativamente importantes.

Entre algunas el reconocimiento de personas en los aeropuertos sobre una base de datos enorme que contenga los datos y fotos de todas las personas del país, del continente, etc., huellas digitales, reconocimiento de sexo, reconocimiento de caracteres, reconocimiento de firmas digitales, etc.

Así mismo, el crear un sistema de aprendizaje distribuido genera ventajas tanto en el proceso de aprendizaje como el proceso de identificación.

7.4 Justificación

El desarrollar un sistema distribuido utilizando el estándar CORBA para el aprendizaje de imágenes nos permite obtener un mayor porcentaje de resultados en menor tiempo. Los procesos serán divididos dentro de los clientes que se encuentren conectados lo que genera una ventaja, ya que entre más clientes

conectados estén es menor el proceso a ejecutar por cada uno, por lo cual los resultados se obtendrán en un tiempo menor.

PARTE II:
DESARROLLO

CAPÍTULO 8: DESARROLLO DE LA RED NEURONAL

El software de la red neuronal sin características distribuidas se desarrolló utilizando el lenguaje de programación C++ y el ambiente de desarrollo visual Borland C++Builder versión 6. En este capítulo se describe cada uno de los componentes, clases, métodos y tipos de datos que fueron utilizados para su desarrollo.

La aplicación esta compuesta por tres clases y el desarrollo de la descripción se guiará por las mismas. La primer clase llamada *Neurona* representa el concepto básico del perceptron de Rosenblatt, la neurona; la segunda clase llamada *Capa* esta compuesta por N objetos de clase *Neurona* y representa las capas Asociativa y Reacción del perceptron; la ultima clase llamada *Red* conformada por 2 objetos de clase *Capa* es la que controla todo el flujo de la operación de la red. La figura 15, muestra el diagrama de la estructura de las clases que componen el perceptron.

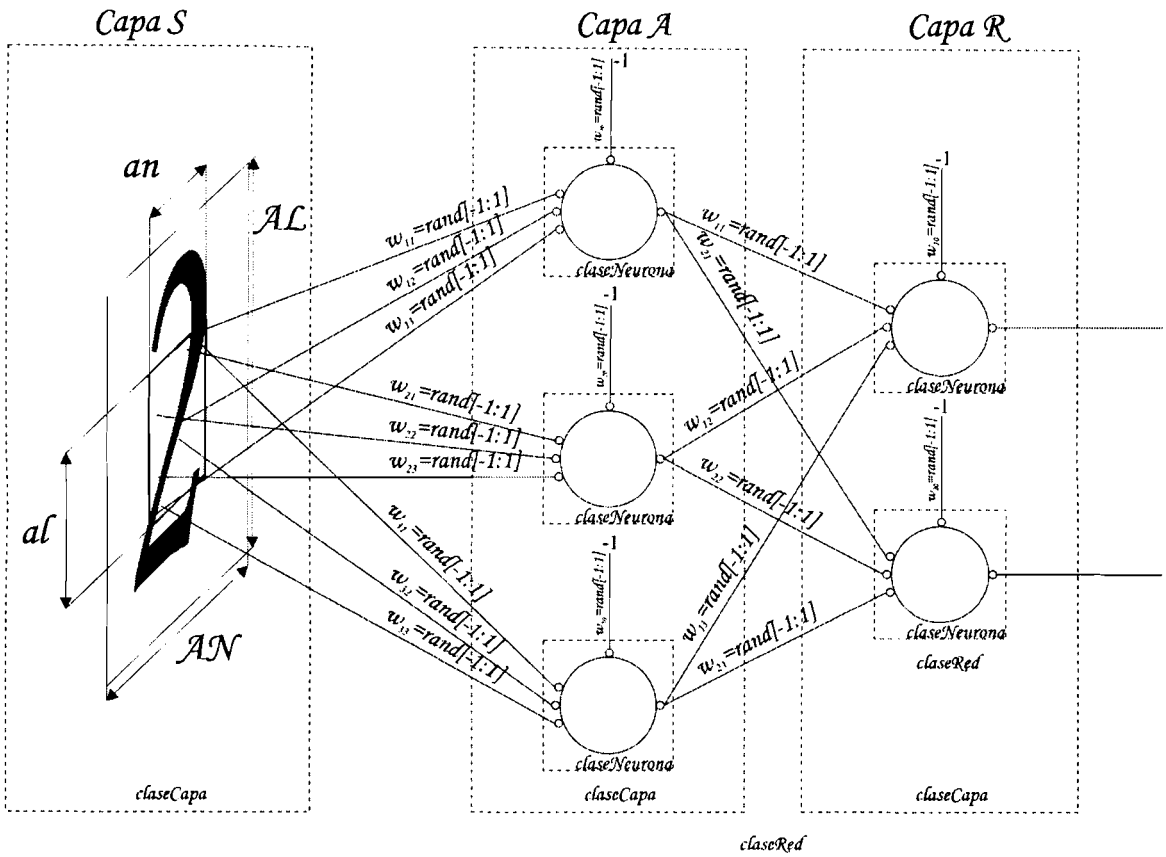


Figura 15. Clases del Perceptron

Clase Neurona

La clase ***Neurona*** esta compuesta por métodos y atributos para manejar y representar el concepto neurona, en la figura 16 se muestra la representación gráfica de los atributos de esta clase.

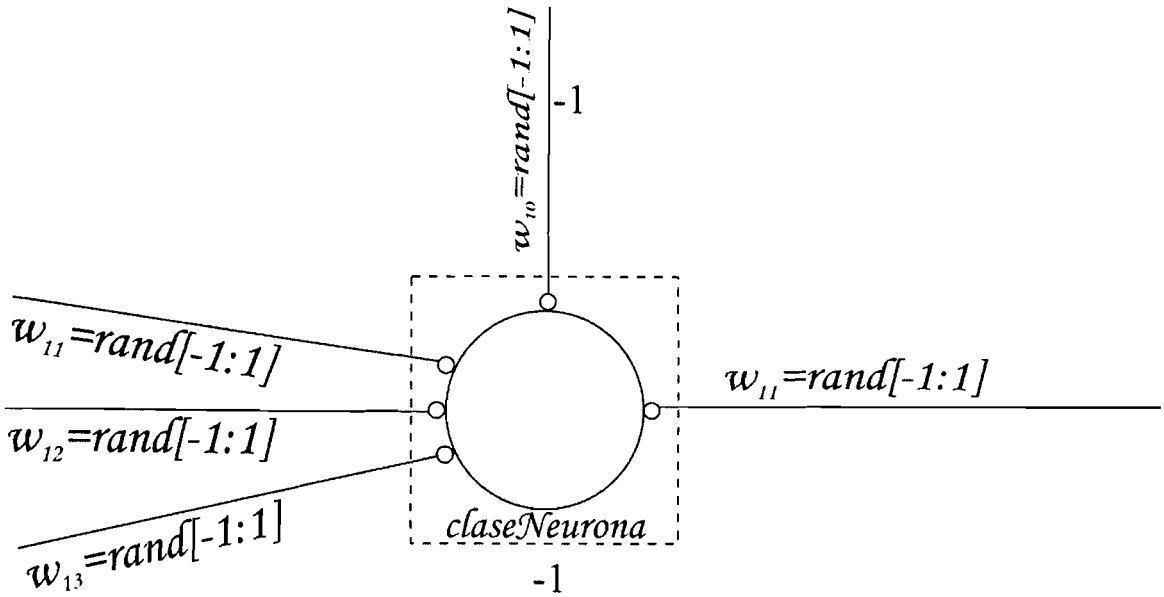


Figura 16. Representación gráfica de los atributos de la Clase Neurona

Las funciones que realiza esta clase son la asignación aleatoria de las coordenadas ***X*** y ***Y*** de la imagen a las neuronas de la capa Asociativa, la asignación de pesos aleatorios en cada una de las líneas de entrada de las neuronas, la asignación aleatoria del umbral para cada neurona y la inicialización de la salida para cada neurona.

Debido a que todos los atributos de esta clase se declaran como privados se crearon métodos para que las clases externas obtengan los valores de los atributos de las coordenadas, los pesos, los umbrales y las salidas.

Los atributos privados que se declaran en esta clase se enlistan a continuación y se da una pequeña descripción de cada uno.

El atributo ***coordenadasX*** es un apuntador de tipo entero que se utiliza para manejar un vector con los valores de las coordenada X de las ***n*** líneas de la neurona ***m*** que pertenecen a la capa Asociativa. En las neuronas de la capa de Reacción, este valor no se utiliza.

El atributo ***coordenadaY*** es un apuntador de tipo entero que se utiliza para manejar un vector con los valores de las coordenada Y de las ***n*** líneas de la neurona ***m*** que pertenecen a la capa Asociativa. En las neuronas de la capa de Reacción, este valor no se utiliza.

El atributo ***pesos*** es un apuntador de tipo flotante que se utiliza para manejar un vector con los valores de los pesos de las ***n*** líneas de la neurona ***m*** de las capas Asociativa y de Reacción del Perceptron.

El atributo **salida** es un valor de tipo flotante que representa la salida de la neurona ***m*** de las capas Asociativa y de Reacción del Perceptron.

El atributo **umbral** es un valor de tipo flotante que representa el peso del umbral de la neurona ***m*** de las capas Asociativa y de Reacción del Perceptron. El valor de entrada para los umbrales de cada neurona de las dos capas se estable como constante con un valor de -1.

La función que realiza cada método de esta clase se describe a continuación así como los parámetros de cada uno de ellos.

El método **AsignacionMemoriaNeuronas** realiza la asignación de memoria a los vectores para manejar las coordenadas ***X***, ***Y*** y los pesos de las ***n*** líneas de las ***m*** neuronas de la capa Asociativa. Para la capa de Reacción solo se asigna memoria al vector que maneja los pesos. Recibe dos parámetros, el primero es un valor entero que representa el número de líneas de cada neurona; para la capa Asociativa este valor es el número de líneas que se asignará de forma aleatoria en la imagen, para la capa de Reacción representa el número de neuronas de la capa Asociativa. El segundo parámetro es un valor de tipo booleano que define el tipo de capa, para la capa Asociativa tendrá un valor true y para la capa de Reacción

tendrá un valor false; este valor se utiliza para determinar si se asigna memoria al apuntador de las coordenadas X, Y.

El método **ObtenerCoordenadaX** se utiliza para obtener el valor de la coordenada X de la i-ésima línea de la j-ésima neurona. Recibe un solo parámetro de tipo entero que representa la línea de la que se quiere obtener la información. El número de la neurona de la línea no se utiliza aquí ya que este valor se maneja en la clase *Capa* que se describirá mas adelante.

El método **AsignarCoordenadaX** se utiliza para asignar el valor de la coordenada X de la i-ésima línea de la j-ésima neurona. Recibe dos parámetros, el primero es de valor entero que representa la i-ésima línea de la neurona a la que se le asignará la coordenada y el segundo es de tipo entero que representa el valor de la coordenada en X que se asignará a la línea. Para la capa asociativa este valor se selecciona aleatoriamente dentro del rango [0, (AN – an)], donde AN es el ancho de las imágenes de los dígitos, para las muestras de la base de datos MNIST es de 28, y ***an*** es el ancho de la ventana en la que se asignaran las líneas de las neuronas de la capa Asociativa siendo una constante de valor 12. Para la capa de reacción este método no es utilizado.

El método **ObtenerCoordenadaY** se utiliza para obtener el valor de la coordenada Y de la i-ésima línea de la j-ésima neurona. Recibe un solo parámetro de tipo entero que representa la línea de la que se quiere obtener la información. El número de la neurona de la línea no se utiliza aquí ya que este valor se maneja en la clase Capa que se describirá mas adelante.

El método **AsignarCoordenadaY** se utiliza para asignar el valor de la coordenada Y de la i-ésima línea de la j-ésima neurona. Recibe dos parámetros, el primero es de valor entero que representa la i-ésima línea de la neurona a la que se le asignará la coordenada y el segundo es de tipo entero que representa el

valor de la coordenada en Y que se asignará a la línea. Para la capa asociativa este valor se selecciona aleatoriamente dentro del rango [0, (AL – al)], donde AL es el alto de las imágenes de los dígitos, para las muestras de la base de datos MNIST es de 28, y ***al*** es el alto de la ventana en la que se asignaran las líneas de las neuronas de la capa Asociativa siendo una constante de valor 12. Para la capa de reacción este método no es utilizado.

El método **ObtenerPeso** se utiliza para obtener el peso de la i-ésima línea de la j-ésima neurona de las capas Asociativa y de Reacción del Perceptron. Recibe un solo parámetro que representa el número de la línea de la que se quiere obtener información. El número de la neurona de la línea no se utiliza aquí ya que este valor se maneja en la clase *Capa* que se describirá mas adelante.

El método **AsignarPeso** se utiliza para asignar el peso de la i-ésima línea de la j-ésima neurona de las capas Asociativa y de Reacción del Perceptron. Recibe dos parámetros, el primero indica el número de la línea a la que se le asignará el valor del peso y el segundo representa el valor flotante aleatorio que será asignado. Para ambas capas este valor toma el rango [-1, 1] con un formato de un dígito y tres decimales.

El método **ObtenerUmbral** se utiliza para obtener el valor del umbral de la j-ésima neurona de las capas Asociativa y de Reacción del Perceptron. No recibe parámetros.

El método **AsignarUmbral** se utiliza para asignar el valor del umbral de la j-ésima neurona de las capas Asociativa y de Reacción del Perceptron. Recibe un parámetro de tipo flotante que representa el valor que se asignará al umbral de la neurona. Para ambas capas este valor toma el rango [-1, 1] con un formato de un dígito y tres decimales.

El método **ObtenerSalida** se utiliza para obtener el valor de la salida de la j-ésima neurona de las capas Asociativa y de Reacción del Perceptron. No recibe parámetros.

El método **AsignarSalida** se utiliza para asignar el valor de salida de la j-ésima neurona de las capas Asociativa y de Reacción del Perceptron. Recibe un parámetro de tipo entero que representa el valor que se asignará a la salida de la neurona. Para la capa Asociativa se asigna el valor -1 o 1 que se determina de la función de activación de cada neurona de esta capa. Para la capa de Reacción se le asignan valores de 1 o 0, 1 significa que la imagen pertenece a la clase de esa neurona y 0 que no.

Clase Capa

La clase ***Capa*** esta compuesta por métodos y atributos para representar las capas Asociativa y de Reacción del Perceptron. Esta clase utiliza los métodos de la clase Neurona para la creación de las capas necesarias. En la figura 17, se muestra la representación gráfica de los atributos de esta clase.

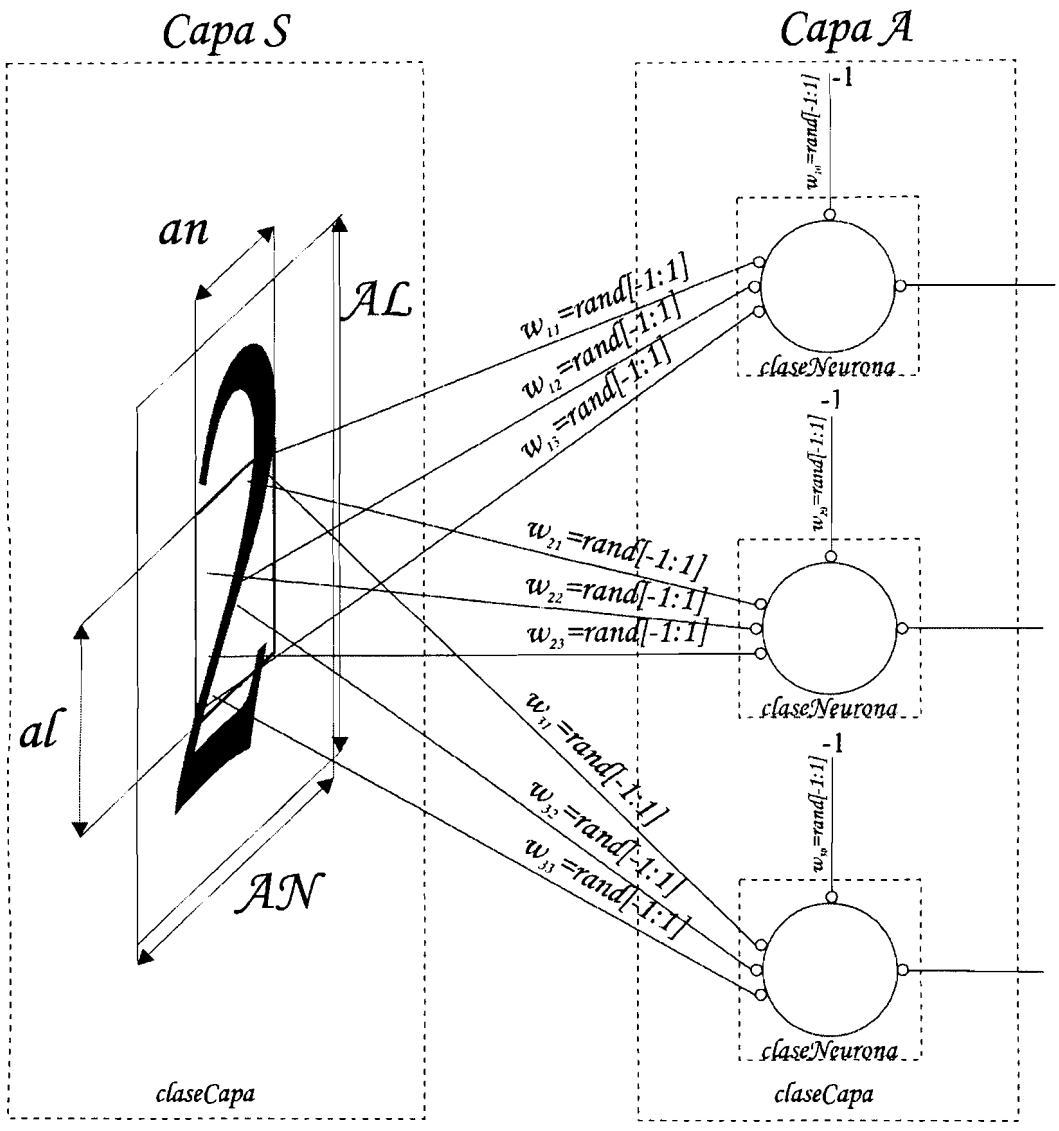


Figura 17. Representación gráfica de los atributos de la Clase Capa

Todos los atributos de esta clase se declaran como privados, a continuación se da una descripción de cada uno.

El atributo **numeroNeuronas** es un valor de tipo entero que representa el número de neuronas de la capa Asociativa o de la capa de Reacción. Para este trabajo de tesis el valor en la capa de Reacción es de 10 neuronas, una para cada

dígito. El número de neuronas de la capa Asociativa es variable dentro del rango [1000, 128] en incrementos de 2^n. $[\forall n \therefore n \in 0 \leq n \leq 7]$.

El atributo **numeroLineas** es un valor de tipo entero que representa el número de líneas de cada neurona de las capas Asociativa y de Reacción.

El atributo **tipoCapa** es un valor de tipo booleano que representa el tipo de la capa dentro de una red construida, este parámetro recibe el valor *true* para identificar la capa Asociativa y el valor *false* para identificar la capa de Reacción.

El atributo **neuronas** es un apuntador de tipo neurona que se utiliza para manejar un vector con los valores de las n neuronas de las capas Asociativa o de Reacción del Perceptron. El tamaño de este vector es el del valor almacenado en el atributo *numeroNeuronas*.

Para que clases externas puedan acceder a estos atributos se crearon métodos específicos para obtener su información. A continuación se da una descripción de cada método junto con sus parámetros correspondientes:

El método **Capa** es el constructor de esta clase. En este se asignan los valores a los atributos del objeto, se asigna un vector de memoria al apuntador de tipo Neurona y a cada objeto de este vector se le asigna memoria llamando al método AsignacionMemoriaNeuronas de la clase Neurona. El constructor recibe tres parámetros, el primero es un valor entero que representa el número de neuronas de la capa, el segundo es un valor entero que representa el número de líneas de cada neurona de la capa, y el último es un valor de tipo booleano que define el tipo de capa, *true* para la capa Asociativa y *false* para la capa de Reacción.

El método **AsignarValoresNeuronas** se utiliza para asignar las coordenadas, los pesos, los umbrales y las salidas a cada uno de los objetos de la clase Neurona del vector apuntado por el atributo neuronas de esta clase. Para esta asignación se utilizan los métodos *AsignarPeso*, *AsignarCoordenadaX*, *AsignarCoordenadaY*, *AsignarUmbral* y *AsignarSalida de la clase Neurona*. Estos métodos se invocan secuencialmente dentro de dos ciclos anidados, el primero recorre el número de neuronas de la capa que representa el objeto y el segundo recorre el número de líneas de cada neurona de la capa que representa el objeto. Este método no recibe ningún parámetro.

El método **CalcularCapaAsociativa** se utiliza para calcular la salida de cada neurona de la capa Asociativa. Para realizar este cálculo se llaman de forma secuencial métodos de la clase Neurona dentro de dos ciclos anidados que recorren las neuronas y las líneas de cada neurona respectivamente. En el ciclo que recorre las líneas se realiza una sumatoria del producto del valor del peso y el valor de entrada proveniente de la capa Sensorial, para esto se utilizan los métodos *ObtenerPeso*, *ObtenerCoordenadaX* y *ObtenerCoordenadaY* de la clase Neurona. En el ciclo que recorre las neuronas de la capa, se le suma el umbral a la suma generada por el ciclo anterior y posteriormente se verifica si esta suma es mayor a un valor de activación de la neurona, que para este trabajo de tesis es 0, si esta condición resulta verdadera se asigna 1 a la salida de la neurona de lo contrario se asigna -1, para este proceso se utilizan los métodos *ObtenerUmbral* y *AsignarSalida* de la clase Neurona. Este método no recibe parámetros.

El método **CalcularCapaReaccion** se utiliza para calcular la salida de cada neurona de la capa de Reacción. Para realizar este cálculo se llaman de forma secuencial métodos de la clase Neurona dentro de dos ciclos anidados que recorren las neuronas y las líneas de cada neurona. En el ciclo que recorre las líneas se realiza una sumatoria del producto del valor del peso y el valor de entrada proveniente de las salidas de la capa Asociativa, para esto se utilizan los

métodos *ObtenerPeso*, y *ObtenerSalida* de la clase Neurona. En el ciclo que recorre las neuronas de la capa se le suma el producto del valor del peso del umbral y -1 a la suma generada por el ciclo anterior, posteriormente se verifica si esta suma es mayor a un valor de activación de la neurona que para este trabajo de tesis es 0, si esta condición resulta verdadera se asigna 1 a la salida de la neurona en proceso, de lo contrario se asigna -1, en las dos opciones de la condición anterior se verifica si el proceso se encuentra en un estado de entrenamiento para determinar el error que se define por la resta del resultado obtenido y el resultado deseado, finalmente se vuelve a verificar si la red esta en estado de entrenamiento y que el error sea diferente de cero para realizar la modificación de todos los pesos de la neurona que marcó un error en el entrenamiento y sumarle el valor del error elevado al cuadrado al error de época. Para este proceso se utilizan los métodos *ObtenerUmbral*, *AsignarSalida* y *AsignarPeso* de la clase *Neurona*. Este método recibe cuatro parámetros, el primero es un objeto de tipo *Neurona* que nos ayuda a obtener y asignar valores de las neuronas de la capa Asociativa, el segundo es un apuntador de tipo entero a un vector que contiene los valores deseados, por ejemplo, para el dígito cero el vector tiene los valores {1,0,0,0,0,0,0,0,0,0}, para el dígito uno los valores del vector son {0,1,0,0,0,0,0,0,0,0} y así sucesivamente, el tercer parámetro es un valor de tipo flotante que indica el factor de aprendizaje (alfa) que se utiliza en la modificación de los pesos de alguna neurona errónea en el proceso de entrenamiento y el ultimo parámetro es un valor de tipo booleano que indica el proceso que realiza la red, *true* para indicar que la red se encuentra en estatus de entrenamiento y false para estatus de Identificación.

El método **GuardarParametrosCapa** se utiliza para guardar en un archivo con extensión out los valores de los atributos *numeroNeuronas*, *numeroLineas*, *tipoCapa* de cada objeto de la clase *Capa*. Este método recibe un parámetro que indica el nombre del archivo, para guardar los parámetros de las dos capas de una

red a este nombre se le agrega la etiqueta "*ParámetrosCapaAsociativa*" y "*ParámetrosCapaReaccion*" respectivamente.

El método **GuardarValoresNeuronas** se utiliza para guardar los valores de los pesos, las coordenadas, la salida y el umbral de las neuronas de cada capa. Este método recibe un parámetro que indica el nombre del archivo, para guardar los valores de las neuronas de las dos capas de una red a este nombre se le agrega la etiqueta "*ValoresCapaAsociativa*" y "*ValoresCapaReaccion*" respectivamente.

El método **CrearCapaDesdeArchivo** se utiliza para generar las capas de una red almacenada, para esto recupera los valores *númeroNeuronas*, *númeroLineas y tipoCapa* y se realiza todo el proceso de creación de una red con los métodos que se describieron anteriormente. Este método recibe un parámetro que indica el nombre del archivo con el que se creara la red, para generar las dos capas de una red a este nombre se le agrega la etiqueta "*ParámetrosCapaAsociativa*" y "*ParámetrosCapaReaccion*" respectivamente.

El método **AsignarValoresNeuronasDesdeArchivo** se utiliza para asignar los valores de las neuronas de una red que fue almacenada y creada utilizando un archivo, la forma de asignación es la misma que si realizáramos la creación de una red como se indicó en los métodos anteriores. Recibe un parámetro que indica el nombre del archivo con el que se asignaran los valores a las neuronas de cada capa de la red, para generar las dos capas de una red a este nombre se le agrega la etiqueta "*ValoresCapaAsociativa*" y "*ValoresCapaReaccion*" respectivamente.

Clase Red

La clase Red esta compuesta por métodos y atributos que representan las capas Asociativa y de Reacción del Perceptron de Rosenblatt. Estos métodos y atributos controlan el flujo de las operaciones de entrenamiento y de identificación de la red, en la figura 16, se ilustran las clases del perceptron así como la representación gráfica de los atributos de la clase red.

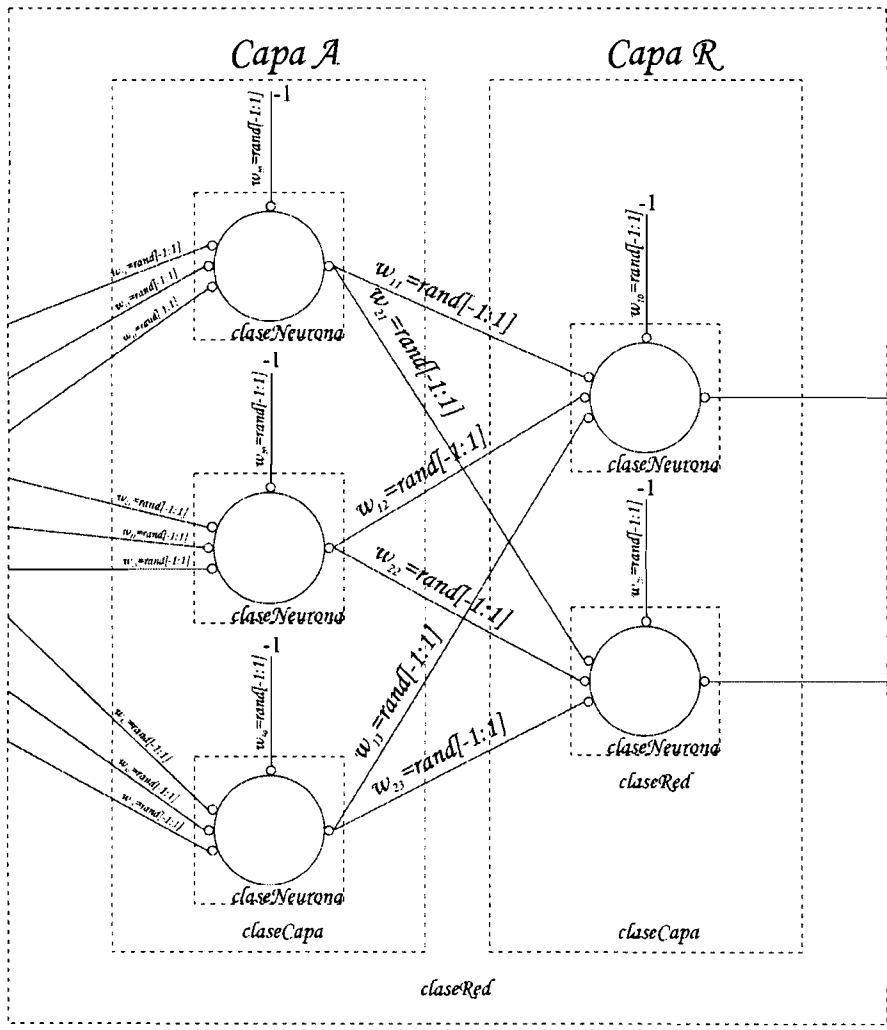


Figura 18. Representación gráfica de los atributos de la Clase Capa

Los atributos privados que se declaran en esta clase se enlistan a continuación y se da una descripción de cada uno:

El atributo **deseados** es un apuntador de tipo entero al que se asigna un vector con los valores deseados, este vector contiene los valores que se muestran el la tabla 1.

Tabla 1. Valores del Vector deseados

Dígito	Valores del vector deseados
0	[1, 0, 0, 0, 0, 0, 0, 0, 0, 0]
1	[0, 1, 0, 0, 0, 0, 0, 0, 0, 0]
2	[0, 0, 1, 0, 0, 0, 0, 0, 0, 0]
3	[0, 0, 0, 1, 0, 0, 0, 0, 0, 0]
4	[0, 0, 0, 0, 1, 0, 0, 0, 0, 0]
5	[0, 0, 0, 0, 0, 1, 0, 0, 0, 0]
6	[0, 0, 0, 0, 0, 0, 1, 0, 0, 0]
7	[0, 0, 0, 0, 0, 0, 0, 1, 0, 0]
8	[0, 0, 0, 0, 0, 0, 0, 0, 1, 0]
9	[0, 0, 0, 0, 0, 0, 0, 0, 0, 1]

El atributo **alfa** es un valor de tipo flotante que representa el índice de aprendizaje de la red en el proceso de entrenamiento. Con un valor de alfa muy grande la red converge rápidamente a su punto de aprendizaje óptimo sin embargo puede ser que nunca llegue al punto indicado, por el contrario con un valor de alfa muy pequeño la convergencia es tardada pero existe seguridad de que el punto a donde llegue sea el más óptimo. Este valor se asigna con un formato de un dígito y tres decimales.

El atributo **errorFinalizacion** es un valor de tipo flotante que indica el valor del error de finalización de un entrenamiento. Cada que termina una época se compara el error generado por la época contra el error de finalización, si el error

de época es menor que el error de finalización el entrenamiento termina de lo contrario continua.

El atributo **capaAsociativa** es un apuntador de tipo *Capa* al que se asigna un vector con los N objetos que representan las neuronas de la capa Asociativa de la red creada. Por medio de este objeto se invocan los métodos de la clase *Capa* que internamente invocan los métodos de la clase *Neurona*.

El atributo **capaReaccion** es un apuntador de tipo Capa al que se asigna un vector con los 10 objetos que representan las neuronas de la capa de Reacción de la red creada. Por medio de este objeto se invocan los métodos de la clase Capa que internamente invocan los métodos de la clase Neurona.

La función que realiza cada método de esta clase se describe a continuación así como los parámetros de cada uno de ellos.

El método **Red** es el constructor de la clase, en este se inicializan los atributos de la clase y se mandan llamar los métodos de la clase *Capa* para la asignación de memoria de toda la estructura de red. Recibe 5 parámetros, el primero indica el número de neuronas de la capa Asociativa, el segundo indica el número de líneas de cada neurona de la capa Asociativa, el tercero indica el número de neuronas de la capa de Reacción que para este trabajo de tesis es una constante de 10, el cuarto indica el índice de aprendizaje para la etapa de entrenamiento y el último indica el error de finalización para la etapa de entrenamiento.

El método **AsignarValoresRed** se utiliza para asignar los valores a los pesos, las coordenadas, salida de las neuronas de la capa Asociativa y de Reacción de la red. Para realizar esta asignación de invocan los métodos

AsignarValoresNeuronas de los objetos Capa declarados en esta clase. Este método no recibe ningún parámetro.

El método **GuardarRed** se utiliza para almacenar los valores y los parámetros de las capas de la red en archivos con extensión out invocando los métodos *GuardarValoresNeuronas* y *GuardarParametrosCapa* miembros de los objetos de tipo *Capa*. Recibe un parámetro que indica el nombre del archivo donde se almacena la información, a este nombre se le agregan etiquetas para indicar el contenido de cada uno, la tabla 2 muestra la relación de los archivos generados y su contenido.

Tabla 2. Relación de archivos y su contenido

Archivo	***Contenido***
NombreArchivoValoresCapaAsociativa.out	Valores de las neuronas de la capa Asociativa
NombreArchivoParametrosCapaAsociativa.out	Parámetros de la capa Asociativa
NombreArchivoValoresCapaReaccion.out	Valores de las neuronas de la capa de Reacción
NombreArchivoParametrosCapaReaccion.out	Parámetros de la capa de Reacción
NombreArchivoPERCEPTRON.txt	Almacena los nombres de los archivos anteriores

El método **CrearRedDesArchivo** se utiliza para crear una red desde un archivo almacenado, para realizar esta tarea se invocan los métodos miembros de la clase Capa AsignarValoresNeuronasDesdeArchivo. Este método recibe un parámetro que indica el nombre del archivo que se utilizará para generar la red, este archivo debe ser el que almacena los nombres de los archivos que contienen los parámetros y los valores de la estructura de la red.

El método **CalcularSalidasRed** se utiliza para calcular las salidas de las neuronas de la capa de Reacción de la red. Para realizar esto se invocan los métodos *CalcularCapaAsociativa* y *CalcularCapaReaccion* miembros de la clase Capa para calcular las salidas de todas las neuronas de la capa Asociativa y las salidas de todas las neuronas de la capa de Reacción respectivamente. Este método no recibe ningún parámetro.

El método **AsignarDeseados** realiza la asignación de los valores del vector deseados. Recibe un parámetro que indica la posición del vector que tendrá un 1 para indicar que neurona de la capa de Reacción debe estar activada.

El método **IdentificarPatron** realiza los mismos pasos que el método CalcularSalidasRed la única diferencia es que aquí se indica que el proceso debe realizarse en un estado de identificación y no de entrenamiento. No recibe ningún parámetro.

La interfaz de usuario de la aplicación del Perceptron se creó en la forma que se muestra en la figura 19. Esta interfaz esta dividida en 5 secciones, a continuación se describe la utilidad de cada sección así como los objetos que componen cada una.

La sección con la etiqueta "Perceptron" se utiliza para introducir los parámetros relacionados con la creación de la red neuronal. Contiene objetos de tipo TEdit para introducir el número de neuronas y el número de líneas de cada neurona de las capas Asociativa y de Reacción, el índice de aprendizaje de la red para la etapa de entrenamiento y por último el error de finalización del entrenamiento.

La sección con la etiqueta "*Red*" se utiliza para llamar los métodos necesarios para la creación, almacenamiento y carga de una red neuronal.

Contiene dos objetos de tipo TButton, el primero con la etiqueta "*Crear*" para crear una nueva red a partir de los parámetros de la sección Perceptron, en el evento OnClick de este botón se ejecuta el constructor de la clase Red y después el método *AsignarValoresRed* y el segundo con la etiqueta "*Cargar*" para crear una nueva red con parámetros y valores almacenados en un archivo de texto, en el evento OnClick de este objeto se ejecuta el método de la clase red CrearRedDesdeArchivo para la creación de la red.

La sección con la etiqueta "*Entrenamiento*" se utiliza para el control de los entrenamientos de la red. Contiene dos objetos de tipo TButton, el primero con la etiqueta "Inicializar" para iniciar el proceso de entrenamiento, en el evento OnClick de este botón se habilita un objeto oculto de tipo TTimer que ejecuta los métodos para realizar el entrenamiento en intervalos de tiempo de 1 milisegundo y el segundo con la etiqueta "Finalizar" para finalizar el proceso de entrenamiento, en el evento OnClick de este botón se deshabilita el objeto TTimer para detener la ejecución de los métodos para realizar el entrenamiento.

La sección con la etiqueta "*Imagen*" se utiliza para mostrar las imágenes que se le presentan a la red neuronal en la etapa de entrenamiento.

La sección con la etiqueta "*Errores de Época*" se utiliza para mostrar gráficamente los valores de los errores de época por cada época que se realiza en la etapa de entrenamiento.

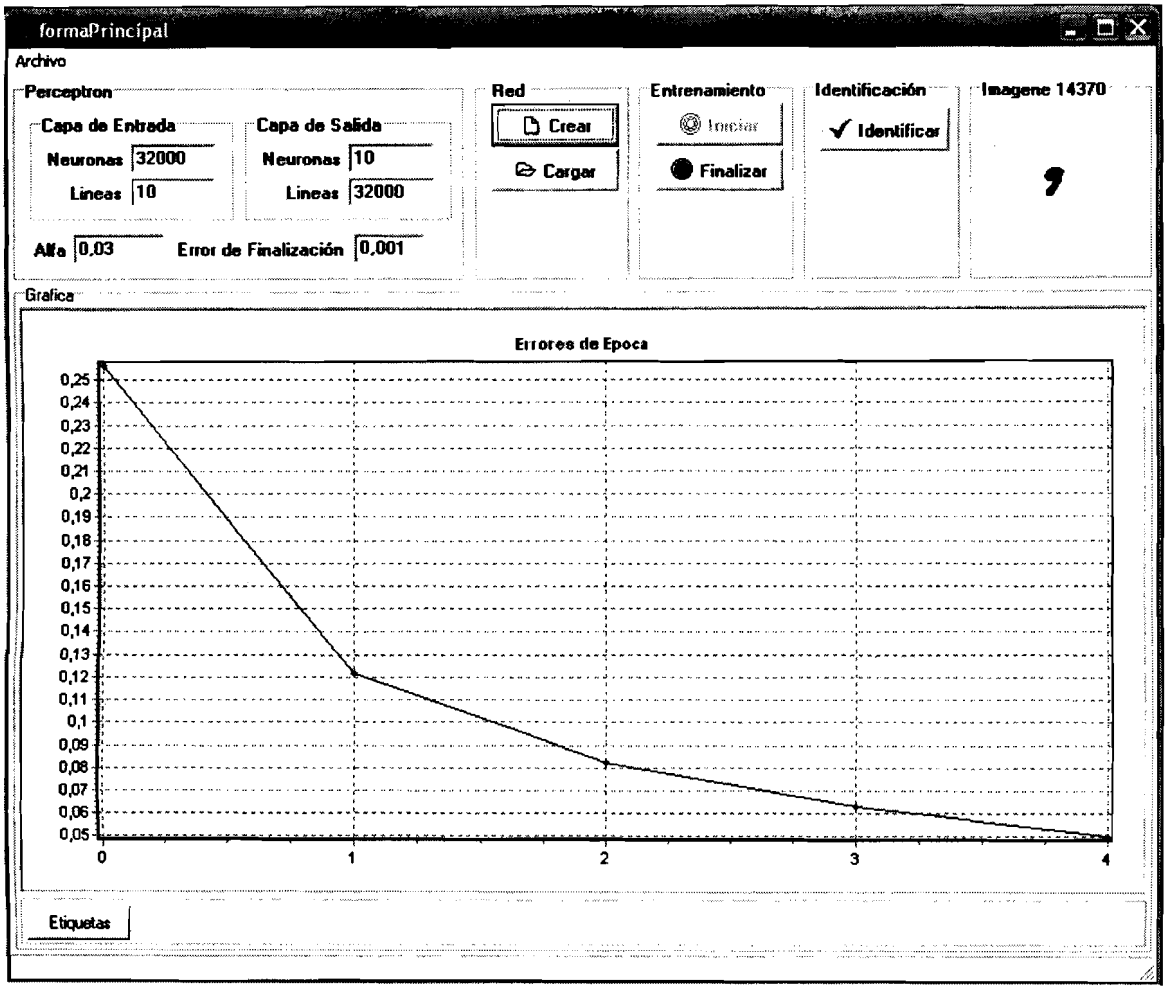


Figura 19. Forma Principal del Perceptron no Distribuido

CAPÍTULO 9: DESARROLLO DE LA RED NEURONAL DISTRIBUIDA

El presente capítulo muestra el desarrollo de la aplicación del Perceptron con características distribuidas en donde se adaptaron las clases utilizadas en la aplicación del Perceptron sin estas características. Se describirá detenidamente cada paso que se realizó para crear la aplicación distribuida utilizando un esquema cliente-servidor, en dicho esquema el servidor ejecutará las funciones de la capa de Reacción del Perceptron y los clientes ejecutan las funciones de la capa Asociativa del Perceptron, para realizar esto las neuronas de la capa Asociativa se dividen entre el número de clientes.

La tecnología basada en el estándar CORBA integrada en el entorno de desarrollo visual C++Builder se llama Inprise VisiBroker para C++, este producto es desarrollado tanto para plataformas operativas UNIX y Windows. Inprise VisiBroker es un ORB que cuenta con un servicio de nombres y eventos, un agente inteligente para la localización de objetos, entre otros mecanismos que aplican el estándar CORBA de manera eficiente.

9.1 Desarrollo del Servidor

Para la realización del Servidor es necesario iniciar en Builder C++ un proyecto que permita crear un servidor CORBA. Para esto se debe seleccionar del menú File la opción New|other y elegir el elemento CORBA Server de la pestaña Multitier, como se puede apreciar en la figura 20.

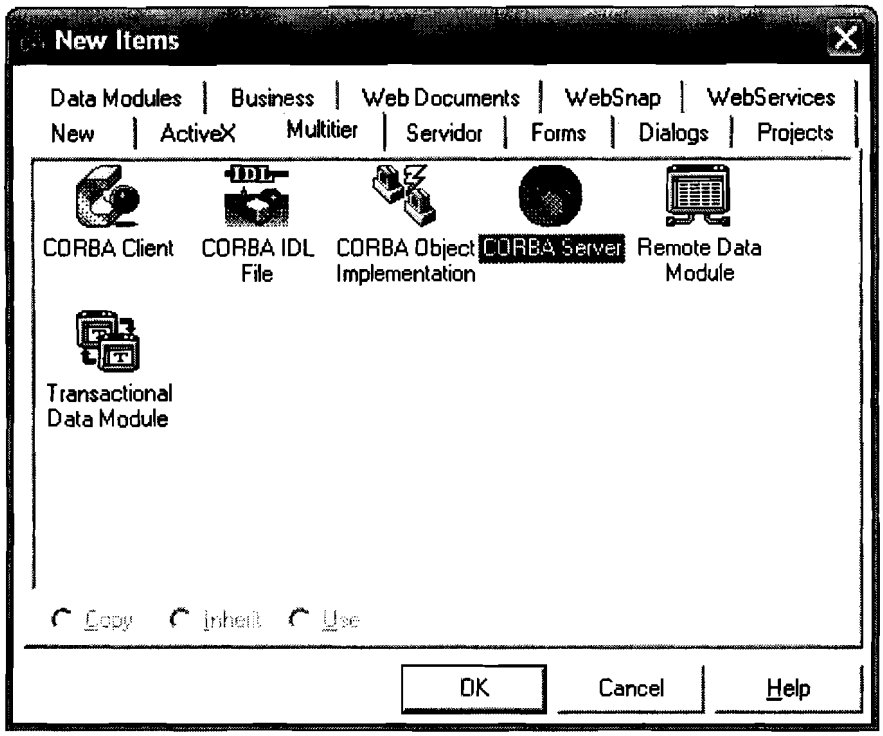


Figura 20. Inicio de un Servidor CORBA

A continuación aparece un asistente que permite especificar el tipo de servidor que desea crearse. Lo que puede ser una aplicación de consola o una aplicación estándar de Windows. En esta aplicación se optará por utilizar la segunda opción. En caso de que las interfaces IDL que van a utilizarse para implementar el servidor ya están definidas, generalmente almacenadas en archivos idl, es posible añadirlas en este mismo asistente. En el caso de que la interfaz o interfaces no se hayan definido, como ocurre en el presente trabajo, activaremos la opción Add New IDL File, tal como se muestra en la figura 21, al aceptar las características de este asistente aparecerá el editor de código mostrando dos módulos: el correspondiente al propio proyecto y el que deberá contener las definiciones IDL. El listado 1, corresponde a la función main() del servidor que se esta construyendo y el listado 2, corresponde a la definición de la interfaz de la Perceptron Distribuido.

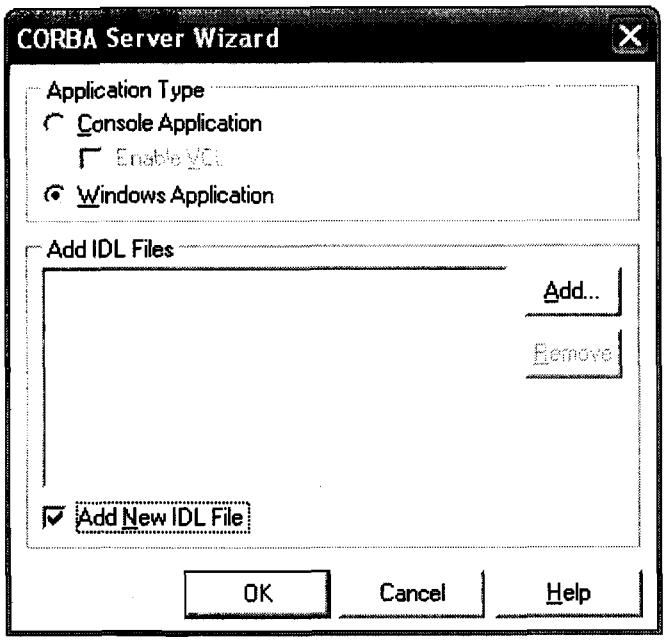


Figura 21. Aspecto del asistente para crear un nuevo servidor

```
000 #include <corbapch.h>
001 #pragma hdrstop
002 //---------------------------------------------------------------------
003 #include <corba.h>
004 #include <condefs.h>
005 #pragma argsused
006 main(int argc, char* argv[])
007 {
008   try
009   {
010     // Initialize the ORB and BOA
011     CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
012     CORBA::BOA_var boa = orb->BOA_init(argc, argv);
013     // Wait for incoming requests
014     boa->impl_is_ready();
015   }
016   catch(const CORBA::Exception& e)
017   {
018     Cerr << e << endl;
019     return(1);
020   }
021   return 0;
022 }
```

Listado 1. Función *main()* del Servidor CORBA

```
001 module RedNeuronal
002  {
003  interface Perceptron
004  {
005    typedef long Imagen[28][28];
006    typedef char NombreArchivo[50];
007    Imagen LeerImagen(in long EntrenamientoIdentificacion, in long Digito, in long NumeroMuestra);
008    void IncrementarNumeroClientes(in long IndicadorOperacion);
009    void EscribirEstatus(in long E);
010    long LeerEstatus(in long Cliente);
011    long LeerNumeroNeuronas();
012    long LeerNumeroLineas();
013    long LeerNombreCliente();
014    void AsignarSalidaNeuronasCapaAsociativa(in long Salida, in long Posicion);
015    NombreArchivo LeerNombreArchivo();
016    long LeerDigito EnCurso(in long Cliente);
017    long LeerNumeroClientes();
018  };
019 };
```

Listado 2. Modulo IDL con la definición de interfaz Perceptron

Un modulo IDL no es un elemento que pueda utilizarse directamente para implementar o utilizar un objeto CORBA si no que es necesario compilarlo para generar los correspondientes módulos de stub y skeleton. Al compilar la aplicación se generan dos nuevos módulos con el mismo nombre que la aplicación seguido del prefijo _c y _s, respectivamente para el stub y skeleton. En la figura 22, se muestra la generación de estos archivos.

El siguiente paso para la creación del servidor es generar una implementación de la interfaz que hemos creado para realizarlo se debe seleccionar la opción CORBA Object Implementation de la opcion New|Other del menú File de C++Builder, como se muestra en la figura 23.

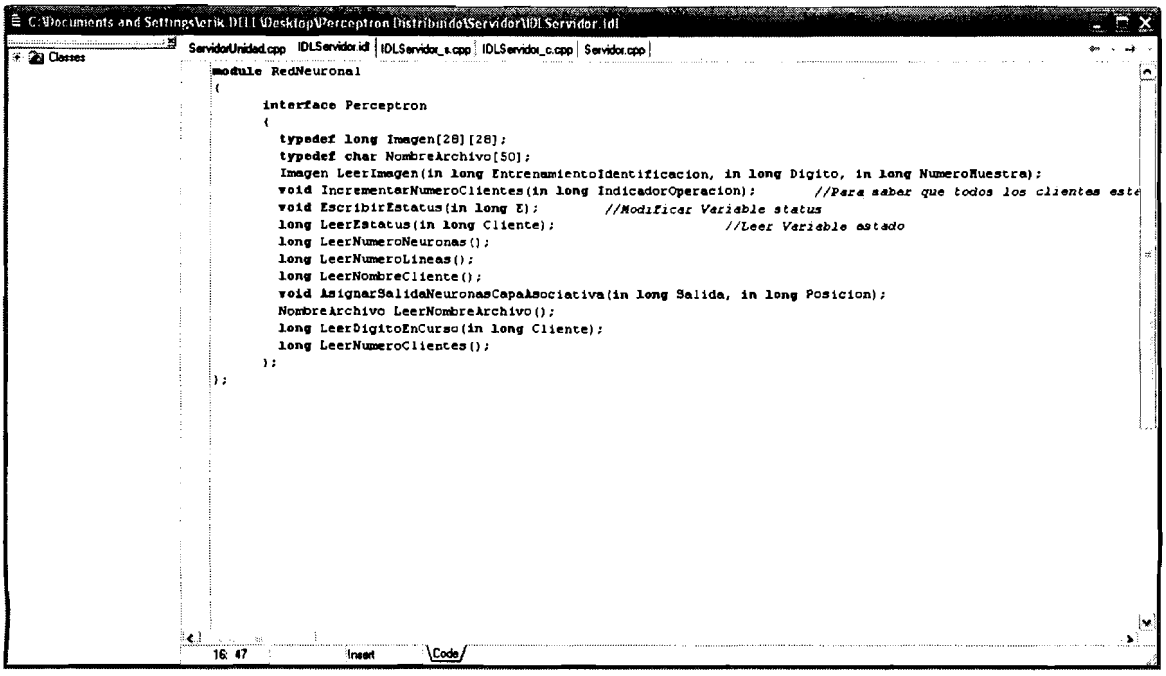


Figura 22. Archivos Stub y Skeleton del Servidor

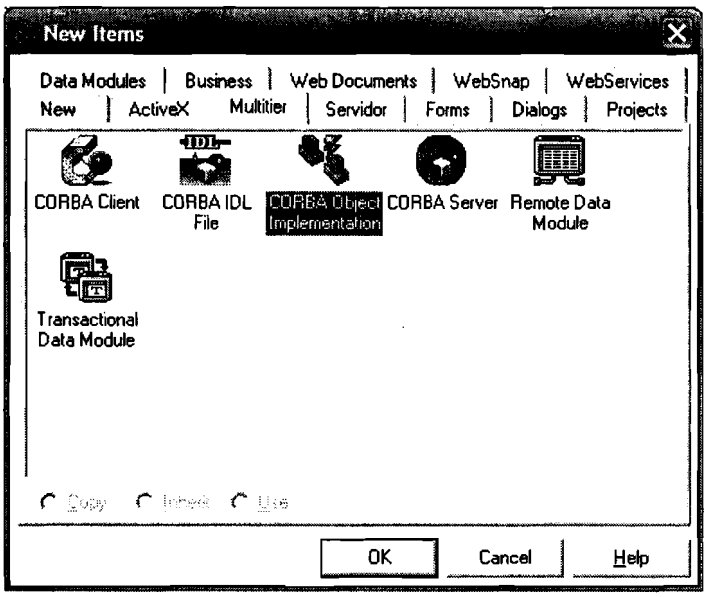


Figura 23. Selección de CORBA Implementation

La figura 24, muestra el Wizard que se utiliza para la implementación de la Interfaz, en este Wizard se indica la ubicación del archivo IDL, el nombre de la

interfaz. También es posible indicar el punto en el que se creará el objeto modificando el campo de texto Unit Name así como el nombre del objeto a través de la opción Class Name.

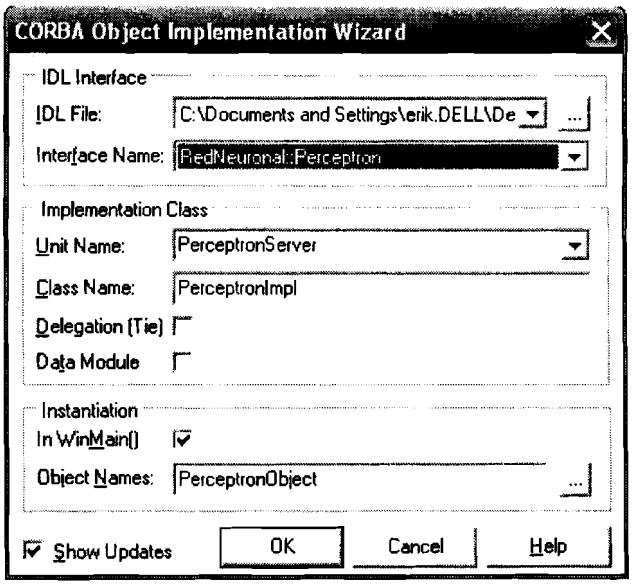


Figura 24. Selección del Objeto de Implementación CORBA

El último paso para la creación del servidor es codificar el funcionamiento de cada método de la interfaz de la aplicación, el listado 3, muestra el código de la implementación del perceptron.

```
001 //---------------------------------------------------------------
002 #pragma hdrstop
003 #include <corba.h>
004 #include "PerceptronServer.h"
005 #include "UnidadServidor.h"
006 //---------------------------------------------------------------
007 #pragma package(smart_init)
008 PerceptronImpl::PerceptronImpl(const char *object_name):
009 _sk_RedNeuronal::_sk_Perceptron(object_name)
010 {
011 }
012 CORBA::Long PerceptronImpl::LeerDigitoEnCurso(CORBA::Long Cliente)
013 {
014   return Principal->digitoClientes[Cliente];
015 }
016 CORBA::Long PerceptronImpl::LeerEstatus(CORBA::Long Cliente)
017 {
018   return Principal->estatusServidor[Cliente];
019 }
```

```
020 CORBA::Long PerceptronImpl::LeerNombreCliente()
021 {
022   return Principal->clientesConectados;
023 }
024 CORBA::Long PerceptronImpl::LeerNumeroClientes()
025 {
026   return StrToInt(Principal->oNumeroClientes->Text);
027 }
028 CORBA::Long PerceptronImpl::LeerNumeroLineas()
029 {
030   return StrToInt(Principal->oNumeroLineasCapaEntrada->Text);
031 }
032 CORBA::Long PerceptronImpl::LeerNumeroNeuronas()
033 {
034   return StrToInt(Principal->oNumeroNeuronasCapaEntrada->Text);
035 }
036 RedNeuronal::Perceptron::_VISanon_arr_28_28_long_slice                    *PerceptronImpl::LeerImagen(CORBA::Long
EntrenamientoIdentificacion, CORBA::Long Digito, CORBA::Long NumeroMuestra)
037 {
038   int i,j;
039   if(StrToInt(FloatToStr(EntrenamientoIdentificacion)) == 0)     //0 -> Entrenamiento
040   {
041     switch (StrToInt(FloatToStr(Digito)))
042     {
043       case 0:
044         for (i = 0;i<pixelesX;i++)
045           for(j=0;j<pixelesY;j++)
046             Principal->Vector[i][j] = Principal->ImagenesCero[StrToInt(FloatToStr(NumeroMuestra))].Pixeles[i][j];
047       break;
048       case 1:
049         for (i = 0;i<pixelesX;i++)
050           for(j=0;j<pixelesY;j++)
051             Principal->Vector[i][j] = Principal->ImagenesUno[StrToInt(FloatToStr(NumeroMuestra))].Pixeles[i][j];
052         break;
053         case 2:
054           for (i = 0;i<pixelesX;i++)
055             for(j=0;j<pixelesY;j++)
056               Principal->Vector[i][j] = Principal->ImagenesDos[StrToInt(FloatToStr(NumeroMuestra))].Pixeles[i][j];
057         break;
058         case 3:
059           for (i = 0;i<pixelesX;i++)
060             for(j=0;j<pixelesY;j++)
061               Principal->Vector[i][j] = Principal->ImagenesTres[StrToInt(FloatToStr(NumeroMuestra))].Pixeles[i][j];
062         break;
063         case 4:
064           for (i = 0;i<pixelesX;i++)
065             for(j=0;j<pixelesY;j++)
066               Principal->Vector[i][j] = Principal->ImagenesCuatro[StrToInt(FloatToStr(NumeroMuestra))].Pixeles[i][j];
067         break;
068         case 5:
069           for (i = 0;i<pixelesX;i++)
070             for(j=0;j<pixelesY;j++)
071               Principal->Vector[i][j] = Principal->ImagenesCinco[StrToInt(FloatToStr(NumeroMuestra))].Pixeles[i][j];
072         break;
073         case 6:
074           for (i = 0;i<pixelesX;i++)
075             for(j=0;j<pixelesY;j++)
076               Principal->Vector[i][j] = Principal->ImagenesSeis[StrToInt(FloatToStr(NumeroMuestra))].Pixeles[i][j];
077         break;
078         case 7:
```

```
079           for (i = 0;i<pixelesX;i++)
080             for(j=0;j<pixelesY;j++)
081               Principal->Vector[i][j] = Principal->ImagenesSiete[StrToInt(FloatToStr(NumeroMuestra))].Pixeles[i][j];
082         break;
083         case 8:
084           for (i = 0;i<pixelesX;i++)
085             for(j=0;j<pixelesY;j++)
086               Principal->Vector[i][j] = Principal->ImagenesOcho[StrToInt(FloatToStr(NumeroMuestra))].Pixeles[i][j];
087         break;
088         case 9:
089           for (i = 0;i<pixelesX;i++)
090             for(j=0;j<pixelesY;j++)
091               Principal->Vector[i][j] = Principal->ImagenesNueve[StrToInt(FloatToStr(NumeroMuestra))].Pixeles[i][j];
092         break;
093       }
094     }
095     else
096     {
097         switch (StrToInt(FloatToStr(Digito)))
098          {
099             case 0:
100               for (i = 0;i<pixelesX;i++)
101                 for(j=0;j<pixelesY;j++)
102                   Principal->Vector[i][j]=Principal->ImagenesCeroIdentificacion[StrToInt(FloatToStr(NumeroMuestra))].Pixeles[i][j];
103             break;
104             case 1:
105               for (i = 0;i<pixelesX;i++)
106                  for(j=0;j<pixelesY;j++)
107                     Principal->Vector[i][j]=Principal-
>ImagenesUnoIdentificacion[StrToInt(FloatToStr(NumeroMuestra))].Pixeles[i][j];
108             break;
109             case 2:
110               for (i = 0;i<pixelesX;i++)
111                 for(j=0;j<pixelesY;j++)
112                    Principal->Vector[i][j]=Principal->ImagenesDosIdentificacion[StrToInt(FloatToStr(NumeroMuestra))].Pixeles[i][j];
113              break;
114             case 3:
115               for (i = 0;i<pixelesX;i++)
116                 for(j=0;j<pixelesY;j++)
117                    Principal->Vector[i][j]=Principal->ImagenesTresI[StrToInt(FloatToStr(NumeroMuestra))].Pixeles[i][j];
118             break;
119             case 4:
120               for (i = 0;i<pixelesX;i++)
121                 for(j=0;j<pixelesY;j++)
122                    Principal->Vector[i][j]=Principal->ImagenesCuatroI[StrToInt(FloatToStr(NumeroMuestra))].Pixeles[i][j];
123             break;
124             case 5:
125               for (i = 0;i<pixelesX;i++)
126                 for(j=0;j<pixelesY;j++)
127                    Principal->Vector[i][j]=Principal->ImagenesCincoI[StrToInt(FloatToStr(NumeroMuestra))].Pixeles[i][j];
128             break;
129             case 6:
130               for (i = 0;i<pixelesX;i++)
131                 for(j=0;j<pixelesY;j++)
132                    Principal->Vector[i][j]=Principal->ImagenesSeisI[StrToInt(FloatToStr(NumeroMuestra))].Pixeles[i][j];
133              break;
134             case 7:
135               for (i = 0;i<pixelesX;i++)
136                 for(j=0;j<pixelesY;j++)
137                    Principal->Vector[i][j]=Principal->ImagenesSieteI[StrToInt(FloatToStr(NumeroMuestra))].Pixeles[i][j];
```

```
138                 break;
139                 case 8:
140                   for (i = 0;i<pixelesX;i++)
141                     for(j=0;j<pixelesY;j++)
142                         Principal->Vector[i][j]=Principal->ImagenesOchoI[StrToInt(FloatToStr(NumeroMuestra))].Pixeles[i][j];
143                 break;
144                 case 9:
145                   for (i = 0;i<pixelesX;i++)
146                     for(j=0;j<pixelesY;j++)
147                         Principal->Vector[i][j]=Principal->ImagenesNuevel[StrToInt(FloatToStr(NumeroMuestra))].Pixeles[i][j];
148                 break;
149               }
 50       }
151           return Principal->Vector;
152     }
153  RedNeuronal::Perceptron::_VISanon_arr_50_char_slice * PerceptronImpl::LeerNombreArchivo()
154  {
155    return Principal->horaFechaArchivo.c_str();
156  }
157  void PerceptronImpl::AsignarSalidaNeuronasCapaAsociativa(CORBA::Long Salida,
 CORBA::Long Posicion)
158  {
159      Principal->salidaNeuronas[StrToInt(FloatToStr(Posicion))] = StrToInt(FloatToStr(Salida));
160  }
161  void PerceptronImpl::EscribirEstatus(CORBA::Long E)
162  {
163   for(int i=0;i<=Principal->clientesAConectar;i++)
164      Principal->estatusServidor[i] = E;
165  }
166  void PerceptronImpl::IncrementarNumeroClientes(CORBA::Long IndicadorOperacion)
167  {
168    Principal->clientesConectados++;
169      if(StrToInt(FloatToStr(IndicadorOperacion)) == 0)
170        {
171          if(Principal->clientesConectados == Principal->clientesAConectar)
172            {
173              Principal->clientesConectados = -1;
174              Principal->Timer1->Enabled = true;
175                 for(int i=0;i<=Principal->clientesAConectar;i++)
176                   Principal->estatusServidor[i] = 200;
177            }
178        }
179        else if(StrToInt(FloatToStr(IndicadorOperacion)) == 1)
180        {
181          if(Principal->clientesConectados == Principal->clientesAConectar)
182            {
183              Principal->clientesConectados = -1;
184                for(int i=0;i<=Principal->clientesAConectar;i++)
185                  Principal->estatusServidor[i] = 251;
186        }
187  }
188  else if(StrToInt(FloatToStr(IndicadorOperacion)) == 2)
189  {
190  if(Principal->clientesConectados == Principal->clientesAConectar)
191    {
192      Principal->clientesConectados = -1;
193      Principal->oReloj->Enabled = true;
194    }
195  }
196  else if(StrToInt(FloatToStr(IndicadorOperacion)) == 3)
```

```
197   {
198     if(Principal->clientesConectados == Principal->clientesAConectar)
198       {
199         Principal->clientesConectados = -1;
200         Principal->Timer1->Enabled = true;
201           for(int i=0;i<=Principal->clientesAConectar;i++)
202             Principal->estatusServidor[i] = 200;
203       }
204   }
205  else if(StrToInt(FloatToStr(IndicadorOperacion)) == 4)
206   {
207     if(Principal->clientesConectados == Principal->clientesAConectar)
208       {
209         Principal->clientesConectados = -1;
210         Principal->Identificar1->Enabled = true;
211       }
212   }
213 }
```

La figura 25, muestra el aspecto final de la interfaz de usuario del servidor del Perceptron Distribuido. La única diferencia de esta interfaz y la de la figura 19 es la sección Distribuido en donde se ponen objetos para controlar el número de clientes que se conectaran al servidor y para mostrar el número de clientes conectados al mismo.

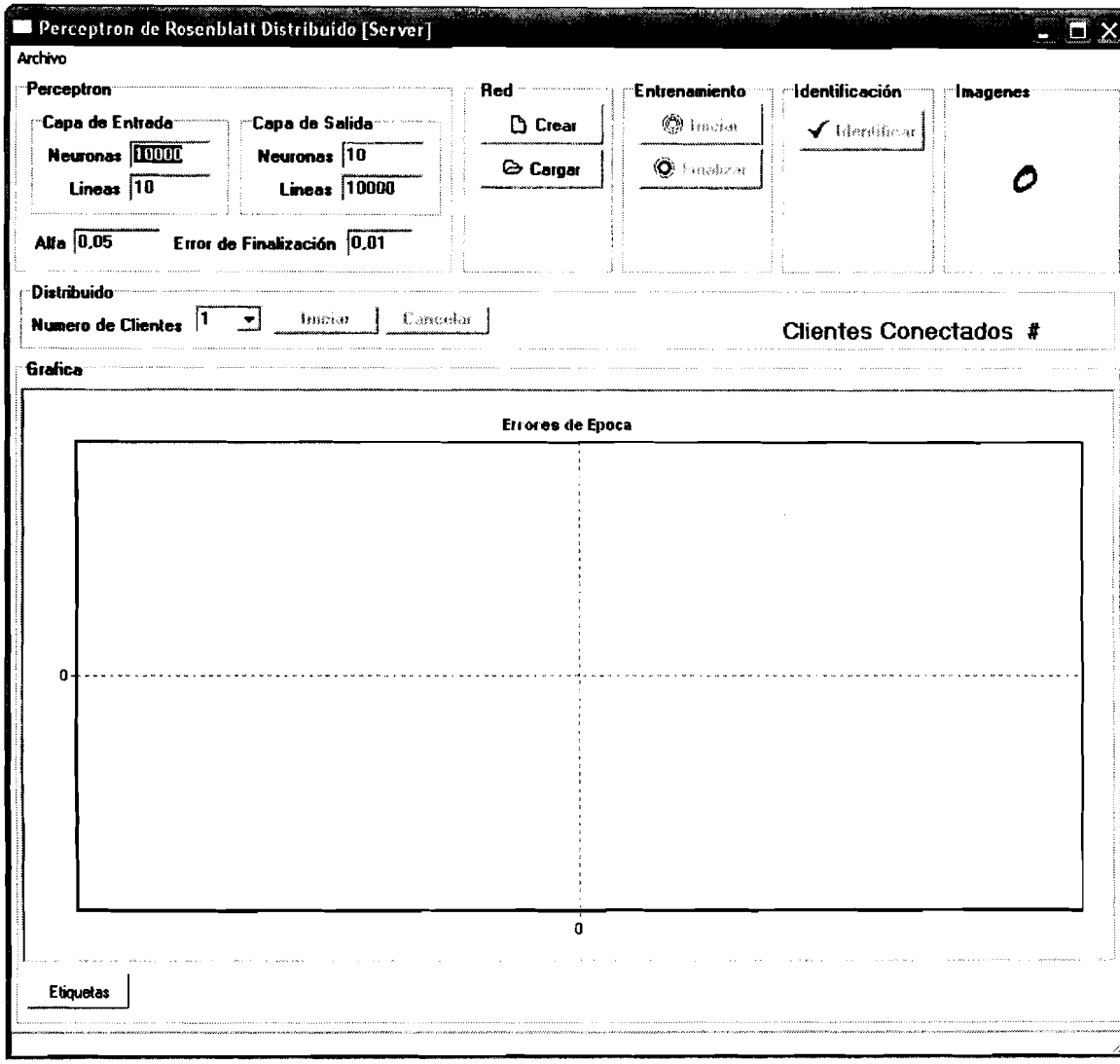


Figura 25. Interfaz del Servidor Distribuido

9.2 Desarrollo del Cliente

Para el desarrollo del cliente que utilizará los servicios del servidor debemos crear un nuevo proyecto, seleccionando de la opción *New|Other* del menú *File* y de la pestaña *Multitier* la opción *CORBA Client*, como se muestra en la figura 26, esto abrirá el asistente de la figura 27, donde debemos agregar el archivo idl creado en el servidor.

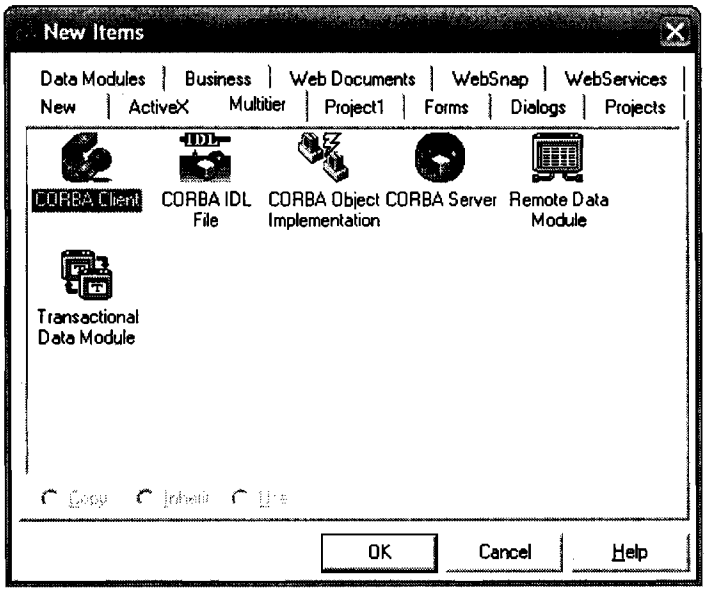


Figura 26. Selección de la opción Cliente CORBA

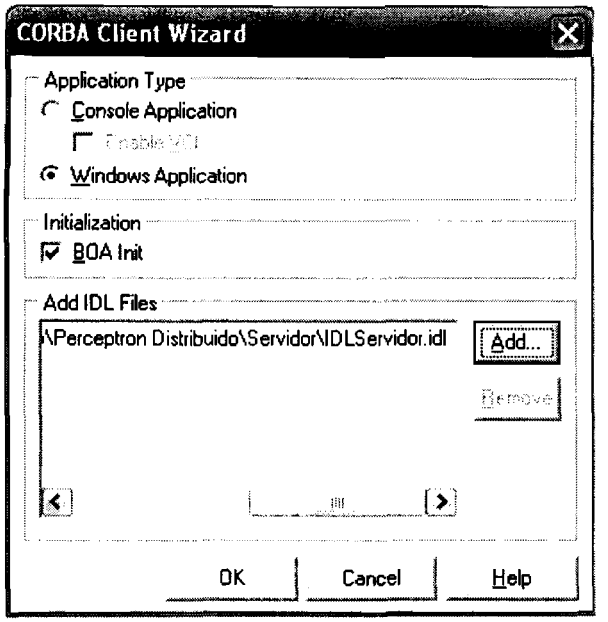


Figura 27. Selección del modulo IDL

Al compilar la aplicación se generan dos nuevos módulos con el mismo nombre que la aplicación seguido del prefijo _c y _s, respectivamente para el stub y skeleton, tal como se muestra en la figura 28, la cual muestra los archivos del stub y skeleton.

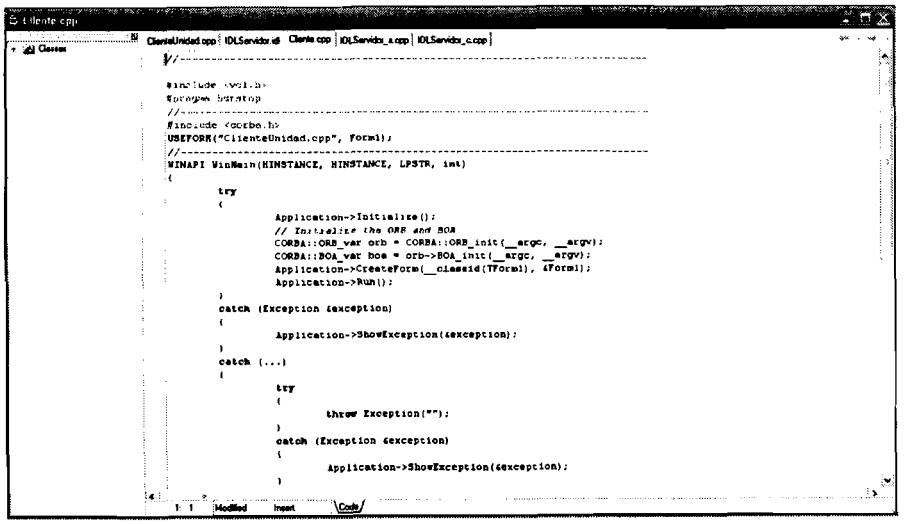


Figura 28. Archivos Stub y Skeleton del Cliente

Para la creación del objeto en la opción *Use CORBA Object* del menú *Edit*, se selecciona la interfaz necesaria y el punto de la aplicación donde debe crearse el objeto, tal como se muestra en la figura 29.

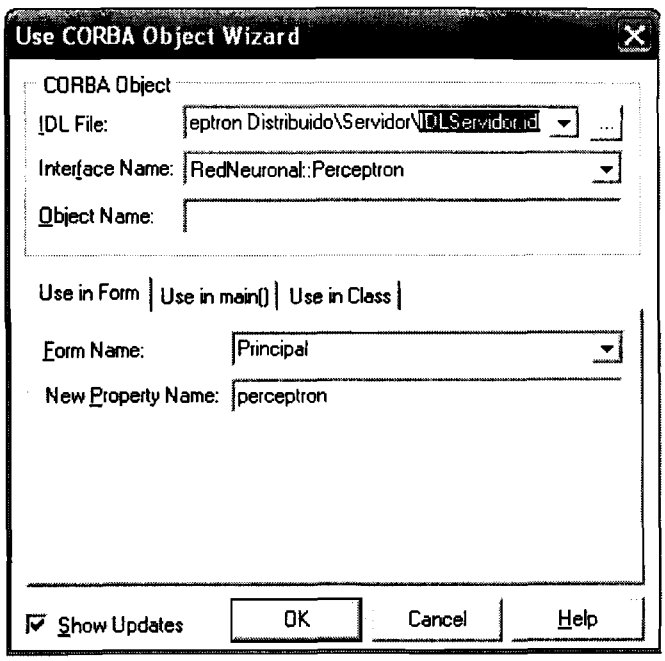


Figura 29. Selección del nombre de la interfase

La figura 30, muestra el aspecto final de la interfaz de usuario del cliente del Perceptron Distribuido. Esta interfaz contiene componentes que se utilizan para mostrar información de la actividad que tienen los clientes con le servidor. Además de información relacionada con los dígitos que el cliente a copiado del servidor.

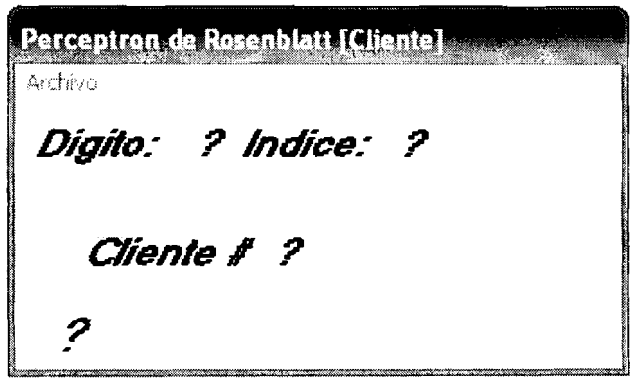


Figura 30. Interfaz del Cliente Distribuido

9.3 Diseño y Configuración de la Red

Para comunicar los módulos de la Red Neuronal Distribuida, se configuró una red de área local utilizando un switch con ocho puertos ethernet 10/100 Mbps. A cada una de las máquinas que conforman la Red se les asignó una dirección IP fija de clase B. La aplicación distribuida tuvo un identificador de red 192.168.0.0. La configuración completa de las direcciones se muestra en la tabla 3.

Tabla 3. Configuración completa de la red

IP	Máscara	Puerta de Enlace	Cliente/Servidor
192.168.1.1	255.255.0.0	192.168.1.254	Servidor
192.168.1.2	255.255.0.0	192.168.1.254	Cliente
192.168.1.3	255.255.0.0	192.168.1.254	Cliente
192.168.1.4	255.255.0.0	192.168.1.254	Cliente

La figura 31, muestra la configuración de la red y como es la interacción entre los clientes y el servidor. Para iniciar esta interacción el servidor debe estar corriendo y estacionarse en un estado de aceptación de clientes. Cuando el servidor se encuentra en este estado los clientes se conectan y comienzan a pedirle las 54,200 imágenes de la base de datos NMIST, cuando terminan los clientes de copiar las imágenes se lo indican al servidor para que éste ponga al sistema en la fase de entrenamiento, en esta fase cada cliente calcula sus salidas con la imagen en curso y coloca los resultados de cada neurona de la capa Asociativa en un vector de salidas localizado en la parte del servidor para que éste pueda calcular la capa de Reacción. Cuando se termina la fase de entrenamiento el servidor les indica a los clientes la ejecución de la última fase del proceso que es el proceso de identificación. En el proceso de identificación los clientes realizan el mismo cálculo de las salidas de cada neurona pero con las imágenes de identificación de la base de datos NMIST y se lo indican la servidor.

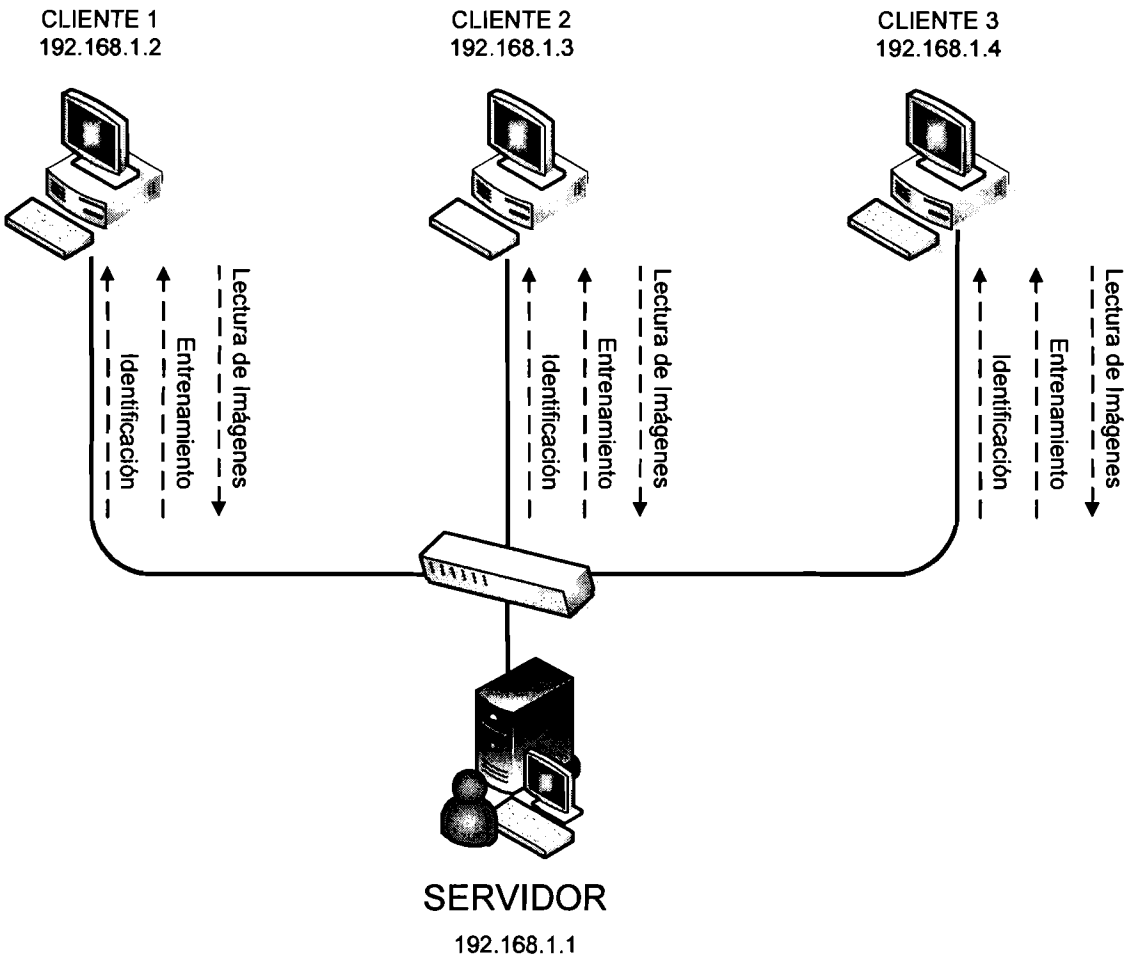


Figura 31. Red Neuronal Distribuida


PARTE III:
RESULTADOS

CAPÍTULO 10: PRUEBAS DEL PERCEPTRON

Las pruebas del perceptron se realizaron utilizando la base de datos NMIST la cual contiene 60,000 muestras para la fase de entrenamiento y 10,000 para la fase de reconocimiento. En este trabajo de tesis se utilizaron únicamente 54,200 muestras en la fase de entrenamiento y el total de muestras en la fase de reconocimiento. La razón de utilizar solo 54,200 muestras de las 60,000 en la fase de entrenamiento se debe a la forma en la que se presentaron los patrones a la red que fue consecutivamente, es decir, se le presentaba la secuencia 0123456789 de la primera muestra de cada dígito, después se le presentaba nuevamente la misma secuencia 0123456789 de la segunda muestra de cada dígito y así consecutivamente hasta terminar las 54,200 muestras. Sin embargo en la base de datos NMIST el número de muestras para cada dígito no es la misma, por ejemplo para el dígito 0 tiene 5,923 muestras y para el dígito 1 tiene 6,742 muestras, por esta razón se selecciono el dígito con el menor número de muestras que es el 5 con 5,420 y se descartaron de los demás dígitos las muestras con índice mayor a éste.

Para realizar una comparación de tiempos en el proceso de aprendizaje se realizaron pruebas con el desarrollo de software del Perceptron sin características distribuidas y con características distribuidas en varios equipos con especificaciones diferentes, la tabla 4 muestra las especificaciones de cada equipo que se utilizó para llevar acabo la fase de pruebas del perceptron.

Tabla 4. Características de los equipos

Equipo	Procesador	Disco Duro	Memoria RAM
1	Intel Centrino Duo 1.73GHz	120Gb	2GB
2	Intel Pentium 4 2.8Ghz	120Gb	512MB
3	Intel Pentium 4 2.8Ghz	80Gb	1GB

En las pruebas se utilizaron diferentes combinaciones cambiando los parámetros Número de Neuronas de la capa Asociativa, Factor de Aprendizaje y Error de Finalización. El uso de los valores para el número de neuronas de la capa Asociativa del Perceptron se retoma de [KUSSUL, 2002, KUSSUL, 2003] y los valores del Factor de Aprendizaje y del Error de Finalización se tomaron de pruebas preliminares que no se presentan en esta sección y que se realizaron para delimitar a un número menor los casos de prueba.

Los resultados obtenidos en los equipos se muestran en las siguientes tablas donde se detalla la información de los parámetros de la red y los resultados obtenidos en cada equipo listando el número de aciertos obtenidos con los diferentes parámetros, el tiempo en el que se realizó el entrenamiento y el número de épocas que se ejecutaron.

La tabla 5, muestran los resultados de la máquina 1. En donde el resultado de identificación más alto fue en el entrenamiento 7 obteniendo 9924 aciertos lo que representa un 99.24% de aprendizaje. La figura 32, muestra gráficamente el comportamiento del aprendizaje durante los 8 ciclos realizados en la maquina y la figura 33, muestra los tiempos de cada uno de estos ciclos.

MÁQUINA 1						
Parámetros de la Red				Resultados		
Número Entrenamiento	Neuronas	Factor de Aprendizaje	Error de Finalización	Número de Aciertos	Tiempo de Entrenamiento	Número de Épocas
1	1000	03	001	8719	2:2:42:26:168	11
2	2000	03	001	9210	2:2:35:15:625	11
3	4000	03	001	9440	2:2:35:15:625	11
4	8000	03	001	9781	2:2:42:25:356	11
5	16000	03	001	9820	6:6:7:53:808	11
6	32000	03	001	9798	7:7:46:20:860	11
7	64000	03	001	9924	15:21:5:31:949	11
8	128000	03	001	9901	22:45:33:12:567	11

Tabla 5. Tabla de Resultados de la Máquina 1

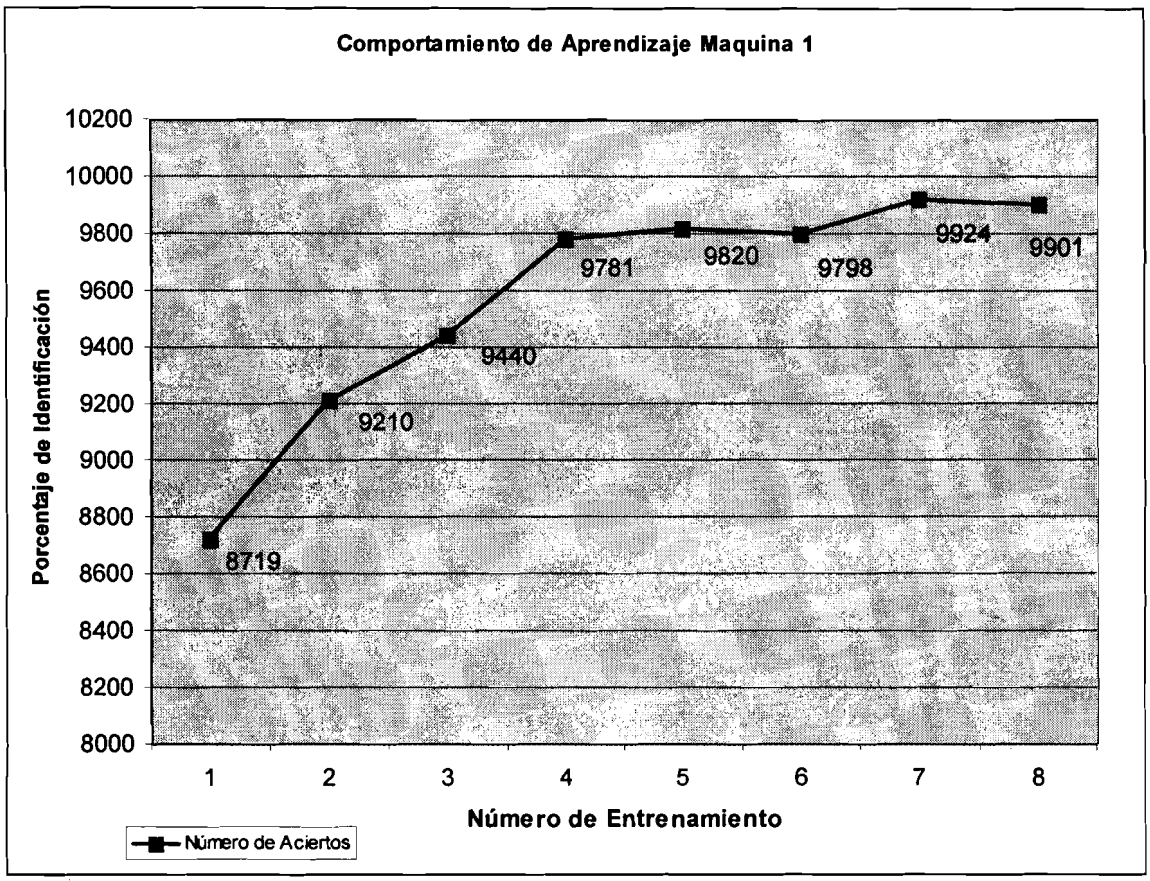


Figura 32. Gráfica de Aprendizaje Maquina 1

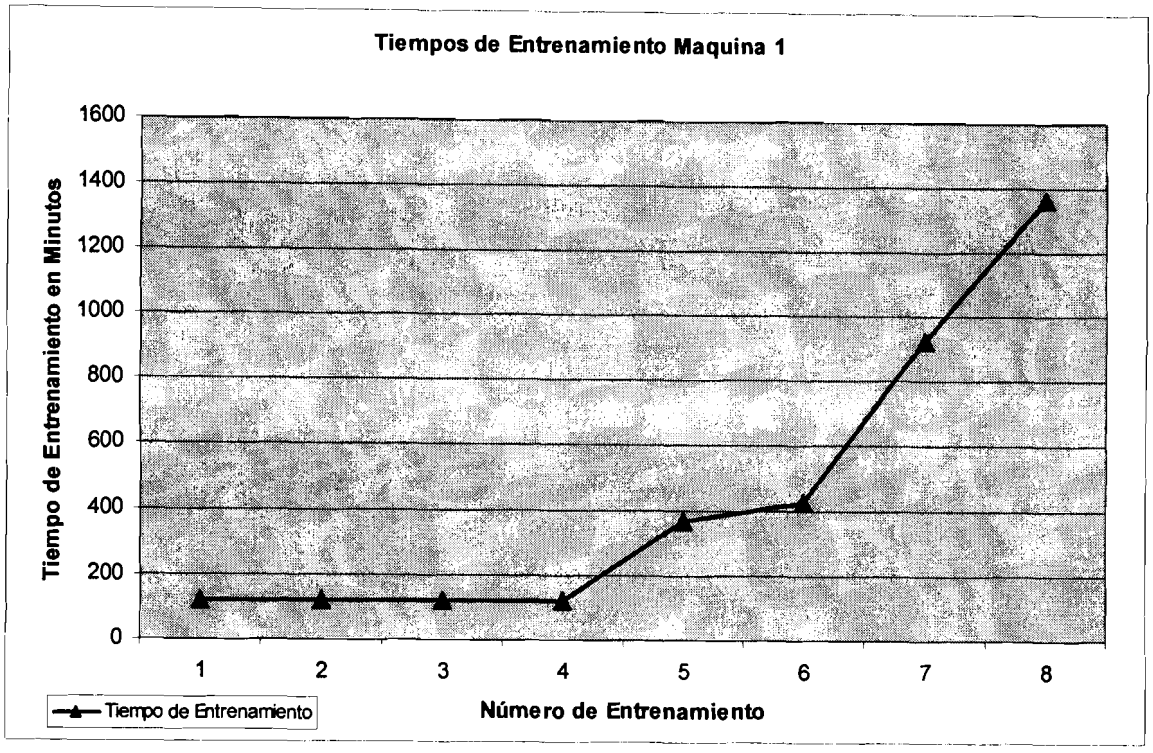


Figura 33. Gráfica de Tiempos Maquina 1

La tabla 6, muestra los resultados obtenidos en la máquina 2. En donde el resultado de identificación más alto fue en el entrenamiento 7 obteniendo 9943 aciertos lo que representa un 99.43% de aprendizaje. La figura 34, muestra gráficamente el comportamiento respecto al aprendizaje durante los 8 ciclos realizados en la maquina y la figura 35, muestra los tiempos de cada uno de estos ciclos.

MÁQUINA 2						
Parámetros de la Red				Resultados		
Numero Entrenamiento	Neuronas	Factor de Aprendizaje	Error de Finalización	Número de Aciertos	Tiempo de Entrenamiento	Número de Épocas
1	1000	03	001	8868	2:2:35:15:718	11
2	2000	03	001	9532	2:2:42:25:137	11
3	4000	03	001	9759	2:2:42:26:89	11
4	8000	03	001	9703	2:2:39:26:109	11
5	16000	03	001	9633	4:4:2:0:496	11
6	32000	03	001	9888	5:5:15:27:750	11
7	64000	03	001	9943	8:8:30:38:215	11
8	128000	03	001	9838	15:15:9:12:813	11

Tabla 6. Tabla de Resultados de la Máquina 2

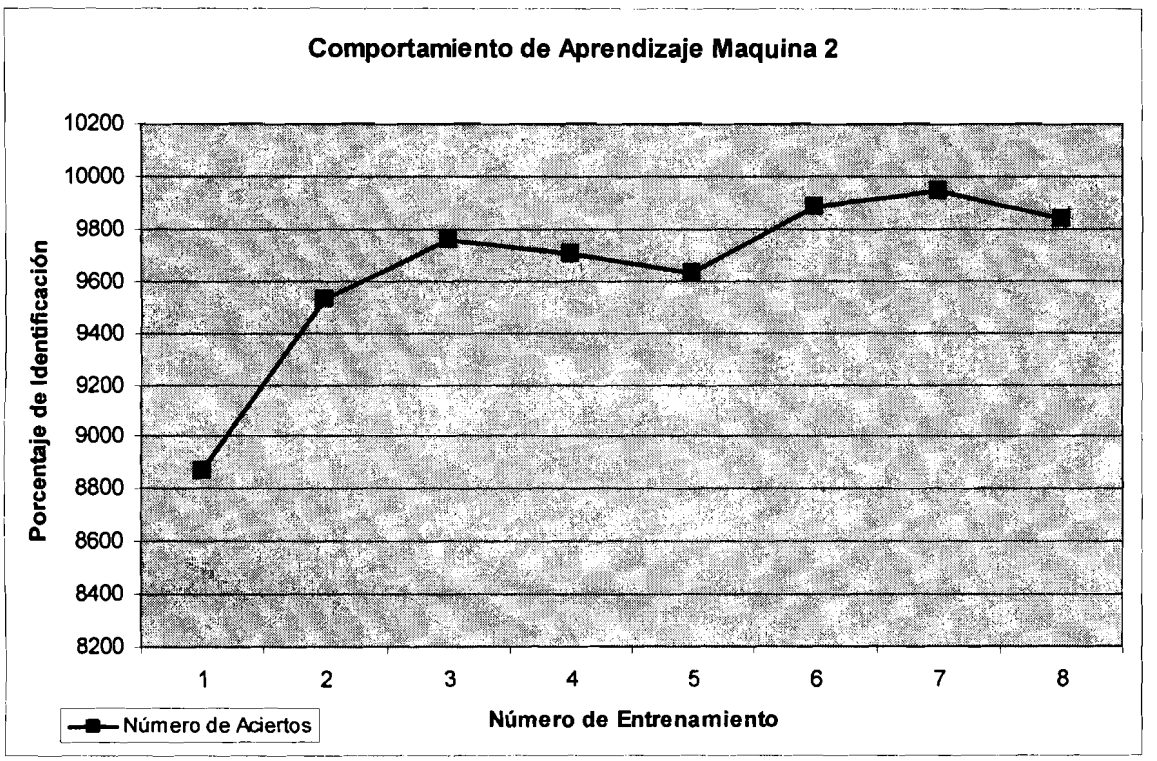


Figura 34. Gráfica de Aprendizaje Maquina 2

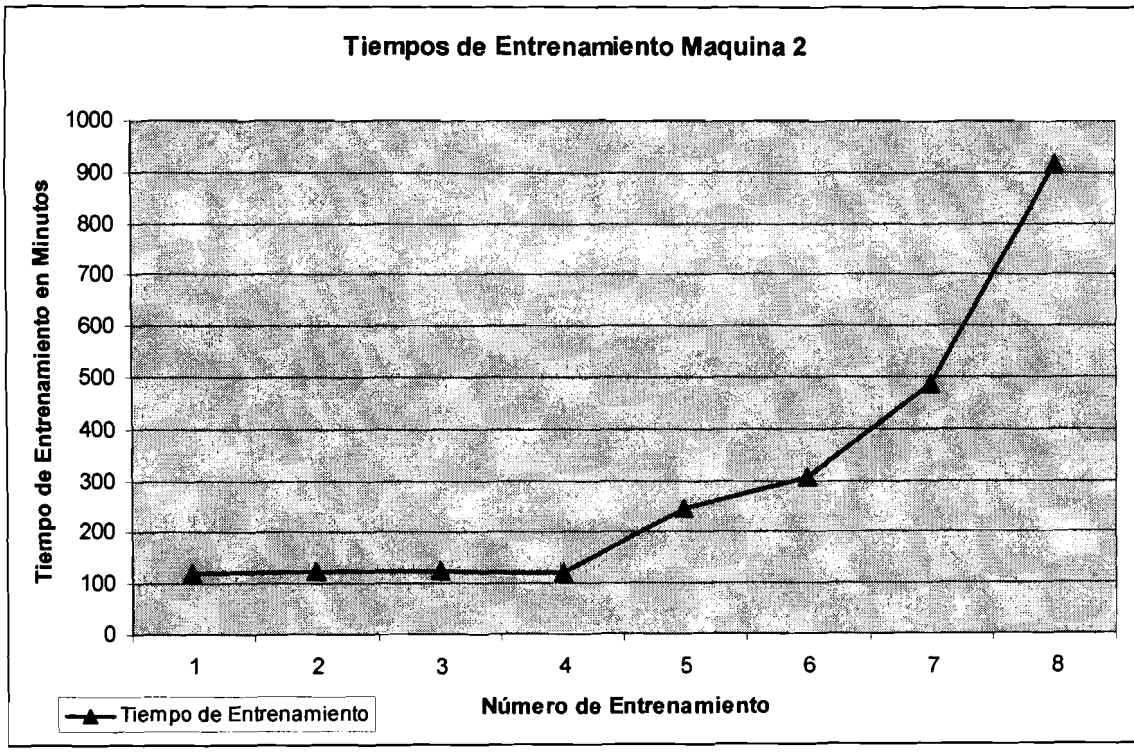


Figura 35. Gráfica de Tiempos Maquina 2

La tabla 7, muestra los resultados obtenidos en la máquina 3. En donde el resultado de identificación más alto fue en el entrenamiento 7 obteniendo 9954 aciertos lo que representa un 99.54% de aprendizaje. La figura 36, muestra gráficamente el comportamiento respecto al aprendizaje durante los 8 ciclos realizados en la maquina y la figura 37, muestra los tiempos de cada uno de estos ciclos

Tabla 7. Tabla de Resultados de la Máquina 3

MÁQUINA 3						
Parámetros de la Red				Resultados		
Numero Entrenamiento	Neuronas	Factor de Aprendizaje	Error de Finalización	Número de Aciertos	Tiempo de Entrenamiento	Número de Épocas
1	1000	03	001	9055	2:2:49:0:871	11
2	2000	03	001	9040	2:2:40:0:328	11
3	4000	03	001	9551	2:2:43:10:683	11
4	8000	03	001	9795	2:2:43:16:43	11
5	16000	03	001	9865	2:2:48:18:266	11
6	32000	03	001	9925	5:5:34:5:718	11
7	64000	03	001	9954	11:11:23:56:123	11
8	128000	03	001	9923	19:19:45:0:345	11

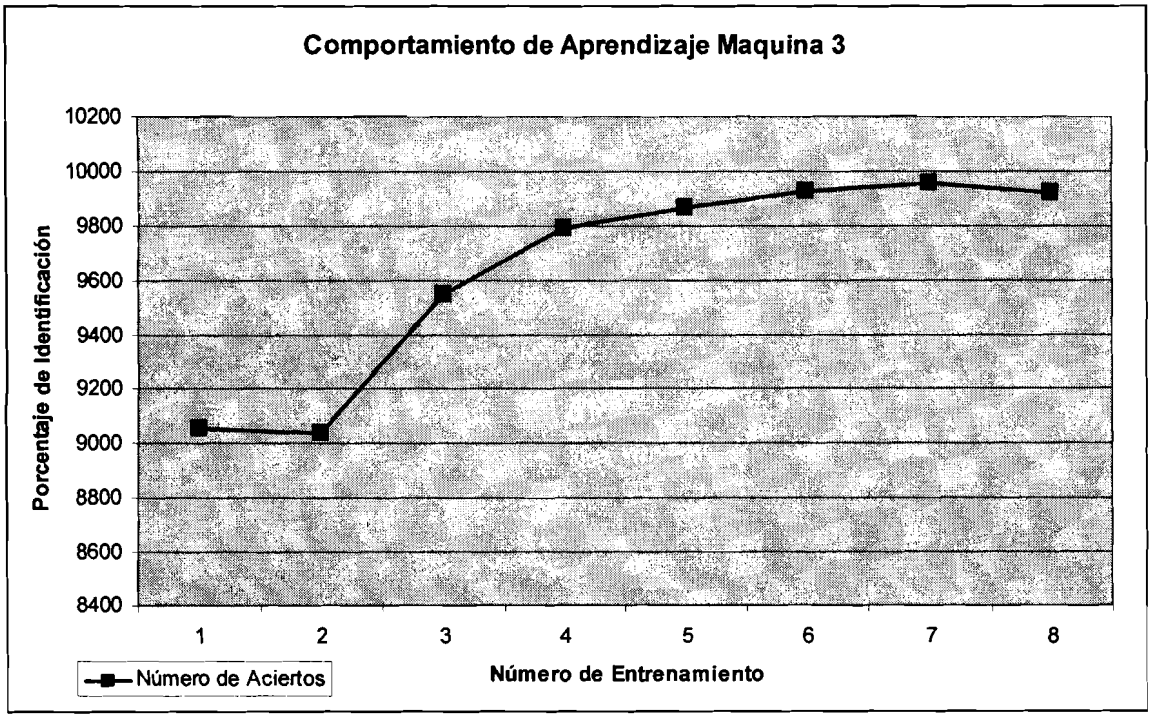


Figura 36. Gráfica de Aprendizaje Maquina 3

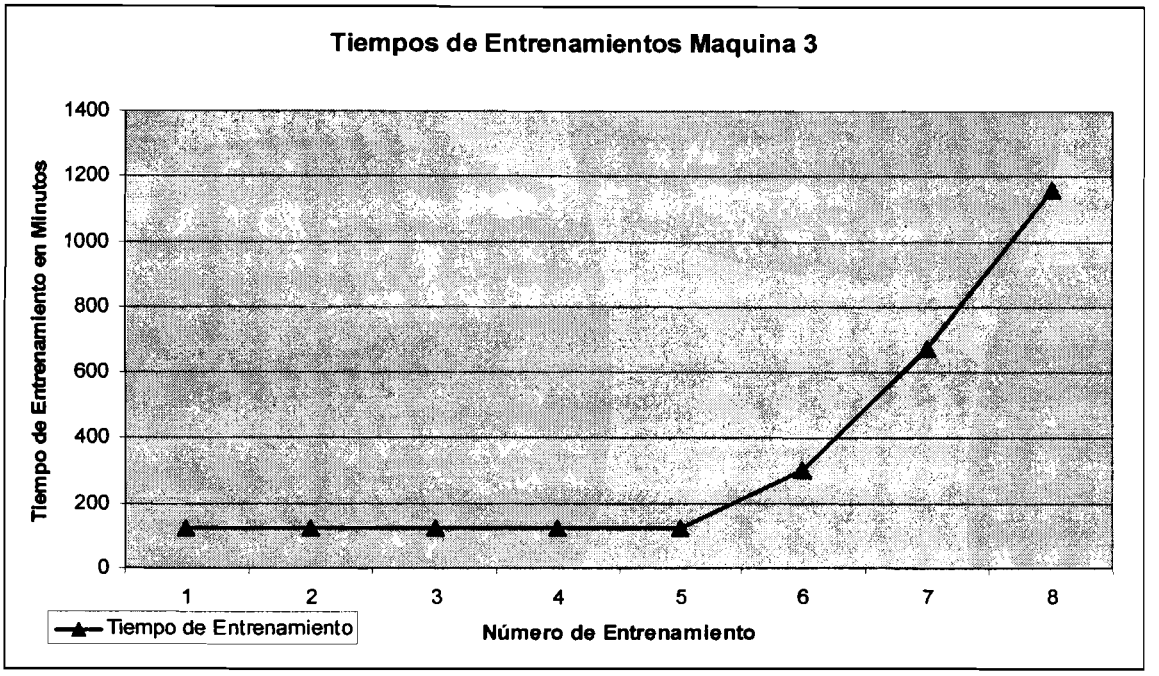


Figura 37. Gráfica de Tiempos Maquina 3

El entrenamiento de la red neuronal distribuida generó los resultados que se muestran en la tabla 8. En donde el resultado de identificación mas alto fue en el entrenamiento 7 obteniendo 9940 aciertos lo que representa un 99.40% de aprendizaje.

Tabla 8. Tabla de Resultados de la Red Neuronal Distribuida

SISTEMA DISTRIBUIDO						
Parámetros de la Red				Resultados		
Numero Entrenamiento	Neuronas	Factor de Aprendizaje	Error de Finalización	Número de Aciertos	Tiempo de Entrenamiento	Número de Épocas
1	1000	03	001	8881	0:45:06:16:673	11
2	2000	03	001	9261	0:44:34:08:238	11
3	4000	03	001	9583	0:43:23: 10:383	11
4	8000	03	001	9760	0:42:35:23:83	11
5	16000	03	001	9773	0:46:54:84:86	11
6	32000	03	001	9870	1:40:05:65:18	11
7	64000	03	001	9940	3:59:56:6:245	11
8	128000	03	001	9887	5:58:33:12:456	11

En la figura 38, se muestra una gráfica con la comparación de tiempos que se obtuvieron de los entrenamientos en cada equipo y en el sistema distribuido, en esta gráfica se puede observar que el sistema distribuido realiza las fases de entrenamiento en intervalos de tiempo más cortos.

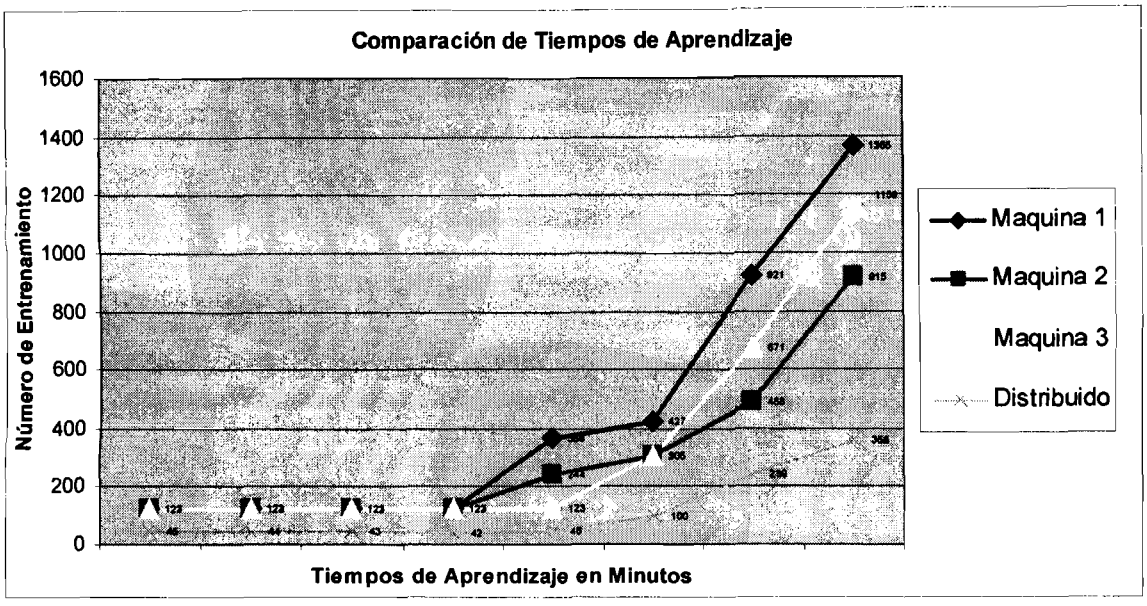


Figura 38. Gráfica Comparativa – Tiempo de Aprendizaje

En la figura 39, se muestra una gráfica con la comparación de aciertos que se obtuvieron en los entrenamientos en cada equipo y en el sistema distribuido, en esta gráfica se puede observar que en el entrenamiento 7 se obtuvo el mayor número de aciertos para las 3 maquinas y para el sistema distribuido.

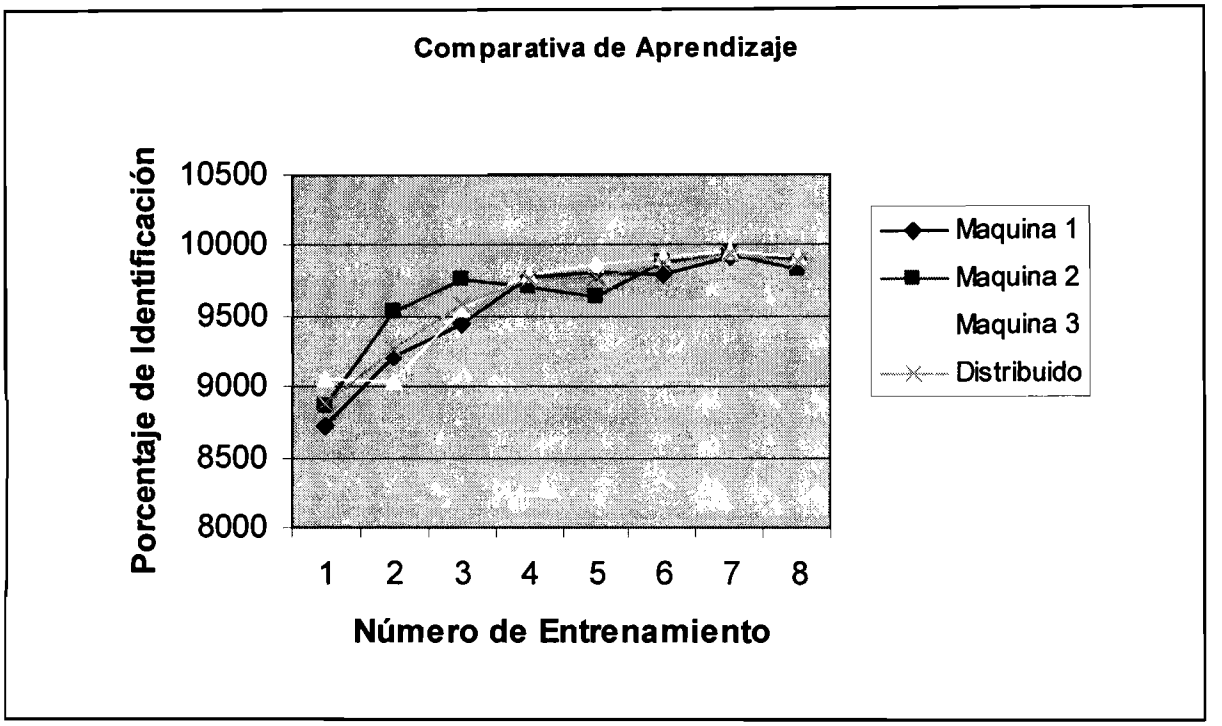


Figura 39. Gráfica Comparativa – Porcentaje de Aprendizaje

CAPÍTULO 11: CONCLUSIONES

Se ha logrado tener un sistema distribuido funcional utilizando la arquitectura Cliente-Servidor donde el sistema operativo Windows XP y el lenguaje de programación C++ interactúan mediante la arquitectura CORBA utilizando el Middleware Borland Inprise VisiBroker para C++. Se han realizado pruebas de entrenamientos con diferentes valores en los parámetros de la red neuronal combinando los valores del número de neuronas, el factor de aprendizaje y el error de finalización.

El uso de sistemas distribuidos hace posible el intercambio de una cantidad importante de información a través de la red, esto permite que la transferencia de tipos de datos como matrices de imágenes se pueda realizar eficientemente; además, todas las características que ofrece un sistema distribuido como son: la reutilización de software existente, la interoperabilidad, la incorporación de facilidad para escalar los sistemas, etc. son adquiridas de forma automática por el presente trabajo.

En teoría los tiempos del sistema distribuido deberían estar representados por un factor, resultado de la división del tiempo que tarda un cliente A, entre el número de clientes, es decir, si un cliente tarda 10 horas en la fase de entrenamiento e identificación y el sistema distribuido se forma por dos clientes el tiempo que debería tardar para el entrenamiento e identificación debería ser de 5 horas, sin embargo, en estas fases el sistema distribuido tiene que sincronizar la ejecución del entrenamiento e identificación, es decir, si un cliente A tarda 10 segundos en calcular las salidas de sus neuronas de la capa Asociativa y un cliente B tarda 15 segundos, el cliente A debe esperar a que el cliente B termine de calcular sus salidas para continuar con el entrenamiento o la identificación. Este efecto de sincronización se presenta en intervalos de tiempo en clientes que

cuenta con características de hardware superiores a los demás equipos que conforman el sistema, en donde se tienen que esperar a que los demás componentes del sistema distribuido terminen su proceso para que se continué con el ciclo siendo esta una desventaja importante que limita un desempeño más eficiente de los ciclos de entrenamiento.

A pesar de esta desventaja el sistema distribuido adquiere cierto grado de paralelismo debido a que el problema lo permite al ejecutar los cálculos de las fases de entrenamiento e identificación en intervalos de tiempo de manera simultanea, siendo esta una de las principales ventajas del Perceptron distribuido.

De acuerdo a los resultados obtenidos en las fases de entrenamiento e identificación se observa una diferencia de tiempos entre los sistemas no distribuidos y el sistema distribuido, siendo más corto el tiempo del último, comprobando la hipótesis planteada en este trabajo de tesis. Aunque el porcentaje en que el sistema distribuido disminuye el tiempo de procesamiento no es proporcional al numero de clientes debido a explicaciones previas, si se realiza una disminución considerable simplemente con el uso de 3 clientes, si se tomo en cuenta que un sistema distribuido se reconfigura fácilmente al ingresar nuevos clientes al sistema esto significa que la disminución de tiempo puede incrementarse.

La ausencia de fallas en la sesión de pruebas en la que se realizaron los 8 ciclos de entrenamientos con el Perceptron Distribuido, demuestra la estabilidad y robustez que adquieren los sistemas distribuidos que utilizan *middlewares* que son construidos bajo los estándares de CORBA.

CAPÍTULO 12: TRABAJO A FUTURO

En el presente trabajo de tesis se realizó el análisis y desarrollo de una red neuronal distribuida obteniendo resultados satisfactorios en los ciclos de entrenamiento con porcentajes altos de identificación de imágenes de dígitos manuscritos de la base de datos NMIST, sin ser este el principal objetivo del trabajo. Sin embargo, se pretende utilizar otras adaptaciones al Perceptron de Rosenblatt de tres capas con alimentación hacia adelante que permitan que los niveles de identificación sean mejores.

El producto final del desarrollo del Perceptron distribuido de este trabajo de tesis no contempla el uso de mecanismos para realizar reconfiguraciones en tiempo real cuando el sistema presenta fallos en la capa Asociativa, es decir, si en un tiempo X el número de clientes cambia porque alguno de ellos falla, el sistema no es capaz de cambiar los parámetros de los clientes funcionales en ese momento. Por esta razón se realizaran cambios en el sistema distribuido para que cuando ocurran eventos como el antes descrito se asigne nuevamente a los clientes el nuevo número de neuronas y se inicie nuevamente el proceso de entrenamiento. Esta reconfiguración también se utilizará cuando el número de clientes se incremente en cualquier momento con el ingreso de un nuevo cliente o cuando un cliente se separe del sistema por razones diferentes a fallos.

Debido a que los resultados de los tiempos en que realizó los ciclos de entrenamiento el sistema distribuido fueron buenos se utilizará este sistema integrando un número de clientes mayor, lo cual aumentará el desempeño en tiempo en diversas aplicaciones que demanden mayor procesamiento de información como pudieran ser, identificación de huellas digitales, rostros utilizando imágenes de dimensiones más grandes e incluso reconocimiento de

video en tiempo real, lectura automática de los cheques de bancos, las declaraciones de impuestos, las direcciones postales y otros documentos.

BIBLIOGRAFÍA

LIBROS

[RICH, 2002] RICH Elaine, Knight evin, "Inteligencia Artificial", Segunda Edición, Editorial Mc-Graw-Hill.

[BONIFACIO, 2002] BONIFACIO Martin del campo,Alfredo Sanz Molina, "Redes Neuronales y Sistemas Difusos", Segunda Edición Ampliada y Revisada, Alfaomega, Año 2002.

[STUART, 1996] STUART J. Russell, Peter Norving, "Inteligencia Artificial, Un enfoque Moderno", Primera Edición, Prentice-Hall, Año 1996.

[GEORGE, 2001] George Coulouris, Jean Dollimore, Tim Kindberg, "Sistemas Distribuidos Conceptos y Diseños", Addison Wesley, Año 2001.

[CHARTE, 2000] CHARTE Francisco, "Builder C++ 4, Adquiera los Fundamentos Avanzados de Programación", Anaya Multimedia, Año.

[COULOURIS, 2001] Coulouris, G., J. Dollymore y T. Kindberg. Sistemas Distribuidos: Conceptos y Diseño. Addison-Wesley, 3 Edición, 2001.

ARTICULOS

[WU, 1999] L.Wu, S.L. Oviatt, P.R. Cohen, 1999, "Multimodal Integration – A Statical View", IEEE Transactions on Multimedia, vol 1, Num 4, pp334-341.

[OMG, 2000] Object Management Group, The Common Object Request Broker: Architecture and Specification, Revisión 2.4, October 2000.

[ALDABAS, 2002] Emiliano Aldabas Rubira, JCEE'02, "Reconocimiento de patrones mediante redes neuronales".

[KUSSUL, 2002] Ernest Kussul, Tatiana Baidyk, Center of Applied Science and Technological Development, UNAM, "Improved Method of Handwritten Digit Recognition".

[KUSSUL, 2003] Ernest Kussul, Tatiana Baidyk, Center of Applied Science and Technological Development, UNAM, "Permutative coding technique for handwritten digit recognition system".