DISEÑO E IMPLEMENTACIÓN DE UNA LIBRERÍA NEURONAL Y DE UNA SUITE DIDÁCTICA PARA LA ENSEÑANZA SOBRE REDES NEURONALES

William Caicedo Torres

DISEÑO E IMPLEMENTACIÓN DE UNA LIBRERÍA NEURONAL Y DE UNA SUITE DIDÁCTICA PARA LA ENSEÑANZA SOBRE REDES NEURONALES

WILLIAM ALEJANDRO CAICEDO TORRES

UNIVERSIDAD TECNOLÓGICA DE BOLÍVAR
PROGRAMA DE INGENIERÍA DE SISTEMAS
FACULTAD DE INGENIERÍA
CARTAGENA DE INDIAS
2010

DISEÑO E IMPLEMENTACIÓN DE UNA LIBRERÍA NEURONAL Y DE UNA SUITE DIDÁCTICA PARA LA ENSEÑANZA SOBRE REDES NEURONALES

WILLIAM ALEJANDRO CAICEDO TORRES

Proyecto para optar al título de Ingeniero de Sistemas

MOISÉS QUINTANA ÁLVAREZ
Magíster en Informática Aplicada

UNIVERSIDAD TECNOLÓGICA DE BOLÍVAR
PROGRAMA DE INGENIERÍA DE SISTEMAS
FACULTAD DE INGENIERÍA
CARTAGENA DE INDIAS
2010

TABLA DE CONTENIDO

1. RESUMEN 7

2. PLANTEAMIENTO DEL PROBLEMA 9

2.1. HIPÓTESIS DE INVESTIGACIÓN 9
2.2. DESCRIPCIÓN DEL PROBLEMA 10
2.3. FORMULACIÓN DEL PROBLEMA 11

3. JUSTIFICACIÓN 12

4. OBJETIVOS 13

4.1. OBJETIVO GENERAL 13
4.2. OBJETIVOS ESPECÍFICOS 13

5. ASPECTOS METODOLÓGICOS 14

5.1. TIPO DE INVESTIGACIÓN 14
5.2. METODOLOGÍA DE INVESTIGACIÓN 14

6. ESTADO DEL ARTE 16

6.1. INTRODUCCIÓN A LA NEUROFISIOLOGÍA HUMANA 16
6.2. REDES NEURONALES ARTIFICIALES 19
6.3. EVOLUCIÓN DE LOS CONCEPTOS Y PARADIGMAS DE LAS REDES NEURONALES 23
6.3.1. PERCEPTRON 23
6.3.2 ADALINE 27
6.3.3 APRENDIZAJE COMPETITIVO 30
6.3.4 MULTILAYER PERCEPTRON (PERCEPTRON MULTICAPA) 34
6.4 ACTUALIDAD DEL CAMPO E INVESTIGACIÓN RECIENTE 38
6.5 ANÁLISIS Y DISEÑO ORIENTADO A OBJETOS 39
6.5.2 PATRONES DE DISEÑO ORIENTADO A OBJETOS 40
6.6 OTRAS LIBRERÍAS Y PROYECTOS DE CÓDIGO LIBRE SOBRE REDES NEURONALES (JAVA). 42

7 DISEÑO E IMPLEMENTACIÓN DE JANE Y NEURALSTUDIO 44

7.1 JANE 45
7.1.1 MODELOS NEURONALES ESCOGIDOS. 45
7.1.2 ARQUITECTURA DE LA LIBRERÍA 47
7.1.2.1 NEURON 48
7.1.2.2 MODELOS NEURONALES 49
7.1.2.2.1 ABSTRACTNEURALNETWORK 49
7.1.2.2.2 ABSTRACTSUPERVISEDNEURALNETWORK 49
7.1.2.2.3 ABSTRACTUNSUPERVISEDNEURALNETWORK 49
7.1.2.2.4 SINGLELAYERFEEDFORWARDNETWORK 49

7.1.2.2.5 ADALINE 50
7.1.2.2.6 PERCEPTRON 50
7.1.2.2.7 MULTILAYERPERCEPTRON 50
7.1.2.2.8 COMPETITIVENETWORK 50
7.1.2.2.9 KOHONENSELFORGANIZINGMAP 50
7.1.2.3 REGLAS DE APRENDIZAJE 51
7.1.2.3.1 INETTRAINER 52
7.1.2.3.2 PERCEPTRONTRAINER 53
7.1.2.3.3 DELTARULETRAINER 53
7.1.2.3.4 BACKPROPAGATIONTRAINER 53
7.1.2.4 CAPAS DE NEURONAS 53
7.1.2.4.1 ABSTRACTNEURONLAYER 53
7.1.2.4.2 SIMPLELAYER 54
7.1.2.4.3 LATTICELAYER 54
7.1.2.5 FUNCIONES DE TRANSFERENCIA 55
7.1.2.5.1 ITRANSFERFUNCTION 55
7.1.2.5.2 PURELINFUNCTION 56
7.1.2.5.3 POSLINFUNCTION 56
7.1.2.5.4 HARDLIMITFUNCTION 56
7.1.2.5.5 LOGSIGMOIDFUNCTION 56
7.1.2.5.6 TANHFUNCTION 56
7.1.3 IMPLEMENTACIÓN DE LAS CLASES DE JANE 57
7.1.3.1 ORG.JANE.CORE 58
7.1.3.2 ORG.JANE.CORE.FUNCTIONS 58
7.1.3.3 ORG.JANE.CORE.LAYERS 59
7.1.3.4 ORG.JANE.CORE.NETWORKS 59
7.1.3.5 ORG.JANE.CORE.TRAINERS 60
7.1.3.6 ORG.JANE.UTILS 61
7.2 NEURALSTUDIO 61
7.2.1 APLICACIONES ESCOGIDAS 62
7.2.2 ASPECTOS CONCEPTUALES Y DE FUNCIONAMIENTO 62
7.2.2.1 CASOS DE USO DE NEURALSTUDIO 63
7.2.2.1.1 CREAR NUEVA RED (UC001) 64
7.2.2.1.2 ABRIR RED DESDE DISCO (UC002) 65
7.2.2.1.3 CREAR RED (UC003) 66
7.2.2.1.4 ENTRENAR RED (UC004) 67
7.2.2.1.5 GUARDAR RED (UC005) 68
7.2.2.1.6 PROCESAR DATOS (UC006) 69
7.2.2.1.7 CREAR DATASET (UC007) 70
7.2.2.1.8 GUARDAR GRÁFICAS (UC008) 71
7.2.2.2 TRAININGDATAMANAGER 71
7.2.2.3 INTERFAZ DE USUARIO 73

7.2.2.4 USO DE JANE 78
7.2.2.5 EXTENSIBLE MARKUP LANGUAGE - XML 79
7.2.2.5.1 ESTRUCTURA DE LOS DOCUMENTOS UTILIZADOS 80
7.2.2.5.2 IMPLEMENTACIÓN DE LA INTERFAZ XML 82
8 CONCLUSIONES Y RECOMENDACIONES FINALES DE LA INVESTIGACIÓN 84
9 REFERENCIAS 87

1. RESUMEN

Las redes neuronales constituyen un modelo de computación inspirado en la estructura del sistema nervioso de los animales superiores, y presentan grandes cualidades para resolver problemas difíciles de atacar usando metodologías convencionales. Por lo anterior las redes neuronales se utilizan para el desarrollo de sistemas inteligentes, con naturaleza adaptativa, y capaces de aprender de su entorno; sistemas resistentes al ruido y con un desempeño claramente superior a las técnicas usuales utilizadas en diversos campos.

La investigación sobre redes neuronales data de por lo menos 70 años atrás, pero en los últimos 20 años se ha observado una verdadera revolución en la aplicación de redes neuronales en problemas de control, reconocimiento biométrico, bioinformática, detección de fraude bancario, entre otros. Lo anterior se debe a la aparición de nuevos algoritmos de entrenamiento y de hardware cada vez más potente, que sumados han permitido este gran avance en el campo de la inteligencia artificial.

Por esto, para el profesional en informática y áreas afines, el entrenamiento en inteligencia artificial – específicamente en redes neuronales –, representa una herramienta cada vez más valiosa y fundamental, puesto que los problemas que enfrentamos en el campo se hacen cada vez más complejos. El reconocimiento de patrones arbitrariamente complejos en grandes sets de datos se ha vuelto una tarea de todos los días y la interacción hombre-máquina es cada vez más común.

Este trabajo propone y documenta el desarrollo de una librería de redes neuronales escrita en Java, de nombre JANE (Java Advanced Neural Engine); y de una herramienta de carácter didáctico para la presentación de redes

neuronales llamada NeuralStudio, también escrita en Java. JANE soporta las arquitecturas de red neuronal más conocidas (Perceptron, ADALINE, red competitiva, Multilayer Perceptron, Kohonen SOM) y además provee puntos de extensión para la incorporación de nuevas arquitecturas. NeuralStudio presenta al interesado en el aprendizaje y utilización de redes neuronales en el ámbito profesional, algunas de las aplicaciones más comunes de estas, tales como el ajuste de curvas (aproximación de funciones, control automático), clusterización y reconocimiento de patrones; de tal forma que se puedan asimilar ciertos conceptos fundamentales de la operación de las redes neuronales.

El aporte novedoso de este trabajo es que brinda la posibilidad a la universidad de tener un proyecto propio para la sensibilización y el trabajo e innovación continuados sobre las diferentes aplicaciones de las redes neuronales en nuestra comunidad científico – académica

.

2. PLANTEAMIENTO DEL PROBLEMA

2.1. *Hipótesis de Investigación*

- ✓ Las redes neuronales proveen una herramienta robusta para el análisis y solución de problemas que los modelos tradicionales no pueden resolver de manera eficiente.
- ✓ Java es una plataforma ideal para el desarrollo de software basado en redes neuronales, por su naturaleza orientada a objetos y su capacidad multiplataforma.
- ✓ Es necesario contar con recursos propios para la enseñanza y desarrollo sobre redes neuronales.
- ✓ Las redes neuronales se pueden representar eficazmente en XML.
- ✓ Los principios de la Programación Orientada a Objetos (OOP) y los patrones de diseño permiten el desarrollo de una arquitectura de clases fácilmente extensible.

2.2. Descripción del Problema

Los seres humanos son bastante buenos a la hora de reconocer patrones en conjuntos de información, y para nosotros diferenciar un objeto de otro es un asunto bastante trivial. Para los programas de computador tradicionales, esto no es cierto; por el contrario, es bastante complejo el poder establecer patrones, extraer características y agrupar información usando las técnicas usuales. Esta ventaja del ser humano se puede atribuir a la forma como procesa la información a través de una arquitectura masivamente paralela compuesta de unidades simples, que actuando juntas le dotan de estas grandes ventajas sobre los modelos tradicionales de computación desarrollados.

Sin embargo, el mundo de hoy se apoya cada vez más en las máquinas para realizar tareas que antes eran competencia específica de los hombres, por lo que se hace necesario contar con herramientas de inteligencia artificial que le permitan a las máquinas "aprender" de la experiencia, y desempeñarse de manera excelente en el procesamiento de la información, de manera similar o incluso superior a la de su contraparte humana. Es aquí donde entran las redes neuronales: Al ser un paradigma de computación bio-inspirado, nos permite aprovechar las características humanas anteriormente mencionadas, tales como la habilidad de aprender y la capacidad de extraer patrones de conjuntos de datos, aun aquellos de apariencia caótica.

La falta de conocimiento acerca de las redes neuronales y sus fortalezas nos priva de excelentes herramientas a la hora de enfrentar problemas relacionados al procesamiento de datos y extracción de información que por los métodos tradicionales no se pueden resolver de manera eficiente. Por lo anterior este trabajo propone el desarrollo de herramientas propias y libres para el desarrollo de

sistemas neuronales, con todos los beneficios que esto puede traer a la comunidad académico-científica de nuestra universidad.

2.3. *Formulación del Problema*

- ✓ ¿Es posible el desarrollo de una librería propia de redes neuronales basada en Java?
- ✓ ¿Cuáles son las arquitecturas de redes neuronales que se podrían implementar en esta librería?
- ✓ ¿Qué aplicaciones deberían estar disponibles en una suite didáctica sobre redes neuronales?
- ✓ ¿Qué jerarquía de clases podría resolver el problema de proporcionar una librería fácilmente extensible?
- ✓ ¿XML permitiría de manera adecuada la serialización de una red neuronal?

3. Justificación

Las redes neuronales se incorporan a diversos campos de investigación científica y de la ingeniería de manera acelerada. La demanda creciente de sistemas inteligentes, interfaces hombre-máquina, y procesamiento avanzado de datos requiere que los profesionales de la informática tengan herramientas adecuadas para poder aprender los conceptos fundamentales sobre redes neuronales y además poder identificar las posibles aplicaciones de estas en su vida profesional.

Por lo anterior, es imperativo desarrollar herramientas propias de bajo costo para la investigación sobre la aplicación de las redes neuronales en diferentes problemas relacionados a diversos ámbitos. Además, es necesario que exista la libertad suficiente para seguir incorporando nuevas características a estas herramientas de manera autónoma, para que la comunidad universitaria interesada en este campo pueda expandir el conocimiento disponible. Este trabajo pretende aliviar muchas de las limitaciones que actualmente tenemos en la universidad en términos de conocimiento y herramientas para el estudio, investigación y utilización de redes neuronales en los problemas relacionados al reconocimiento avanzado de patrones.

4. OBJETIVOS

4.1. *Objetivo General*

Diseñar e implementar una librería y una suite didáctica sobre redes neuronales en Java y utilizando software libre, con el objetivo de impulsar el interés, la investigación y desarrollo sobre redes neuronales en la Universidad Tecnológica de Bolívar.

4.2. *Objetivos Específicos*

- Conocer y comprender el funcionamiento de las arquitecturas más relevantes de redes neuronales y sus respectivos algoritmos de entrenamiento.
- Seleccionar las aplicaciones de las redes neuronales más relevantes para la ilustración de los conceptos asociados al funcionamiento de las redes neuronales.
- Diseñar e implementar de manera eficaz una estructura de clases para la librería de redes neuronales, de tal forma que sea fácilmente extensible.
- Establecer cuáles son los modelos más adecuados para ser incluidos en una librería multipropósito como JANE, e implementarlos.
- Evaluar la pertinencia de la utilización de XML para la representación y almacenamiento de una red neuronal y diseñar la estructura de un documento XML que permita alcanzar tal fin.

5. ASPECTOS METODOLÓGICOS

5.1. *Tipo de Investigación*

Investigación Aplicada

5.2. *Metodología de Investigación*

En este trabajo se aplica la metodología de investigación en el desarrollo de sistemas propuesta por Nunamaker, Chen y Purdin en [12]. Esta metodología involucra las siguientes etapas:

- **Construcción de un marco conceptual:** En esta etapa se recopiló la información necesaria y suficiente para comprender los conceptos relacionados al funcionamiento, desarrollo y aplicaciones de las redes neuronales. La consulta bibliográfica y el estudio de las teorías subyacentes son parte fundamental de esta fase de investigación.
- **Desarrollo de una arquitectura para el sistema:** Aquí se procedió a la selección de los modelos neuronales más relevantes, de tal forma que se pueda definir una arquitectura para la librería JANE (Java Advanced Neural Engine), basada en patrones arquitectónicos adecuados, para proveer al sistema de extensibilidad y modularidad suficientes. Una de las metas principales es que JANE pueda ser usada de forma *standalone.* Se comenzaron a considerar arquitecturas para NeuralStudio.
- **Análisis y diseño del sistema:** En esta fase se delinearon las estrategias a aplicar para el modelado de los modelos neurales anteriormente escogidos, se evaluaron las posibles implementaciones de los algoritmos de entrenamiento, de tal forma que al final de esta etapa el autor estuvo en capacidad de escoger las soluciones pertinentes en cada caso.

- **Construcción del prototipo:** En esta etapa se construyó un prototipo avanzado para JANE y uno rudimentario de NeuralStudio, de tal forma que se pudo evaluar la factibilidad y eficacia de las soluciones escogidas anteriormente. La idea en esta fase fue verificar el funcionamiento de JANE, de tal forma que una vez verificado se procedió a avanzar en la construcción de un prototipo avanzado de NeuralStudio. Se obtuvo conocimiento aun más profundo acerca del marco conceptual y la complejidad de los problemas planteados en la investigación.
- **Observación y evaluación del sistema:** En la etapa final se procedió a la prueba del sistema en su conjunto (NeuralStudio y JANE), observando su funcionamiento y verificando que se comporte dentro de los rangos esperados. Se contempló la retroalimentación de acuerdo a lo observado en aras de corregir y/o modificar aspectos del diseño arquitectural y la implementación cuando sea necesario. Se consolidaron las experiencias aprendidas para arribar a las conclusiones de la investigación.

6. ESTADO DEL ARTE

En esta sección se describen conceptos básicos relacionados con las redes neuronales, su inspiración biológica y la evolución del paradigma junto con las arquitecturas desarrolladas a lo largo de los últimos 60 años

Para poder comprender los fundamentos de la teoría de redes neuronales artificiales, debemos repasar ciertos conceptos relacionados a la estructura del sistema nervioso humano, de donde dicha teoría toma inspiración.

6.1. *Introducción a la neurofisiología humana*

El sistema nervioso humano, específicamente el cerebro, es una compleja obra de ingeniería que es capaz de llevar a cabo tareas extremadamente complejas, un volumen de cálculos gigantesco, con un muy bajo consumo de energía; además, es un sistema que posee una naturaleza inherente de tolerancia a la falla y de adaptabilidad a un entorno cambiante. Todas estas virtudes se deben a la arquitectura que, desde el nivel más sencillo, caracteriza dicho sistema.

La unidad fundamental que compone el tejido nervioso es la neurona. La neurona es un tipo de célula altamente especializada que presenta propiedades muy especiales que favorecen la conducción de impulsos eléctricos a través de su estructura, mediante el intercambio selectivo de iones a través de su membrana y la generación de pequeños potenciales eléctricos. A pesar de lo anterior, la neurona es una célula bastante simple y la potencia del cerebro y el sistema nervioso viene de la interacción asincrónica de millones de neuronas.

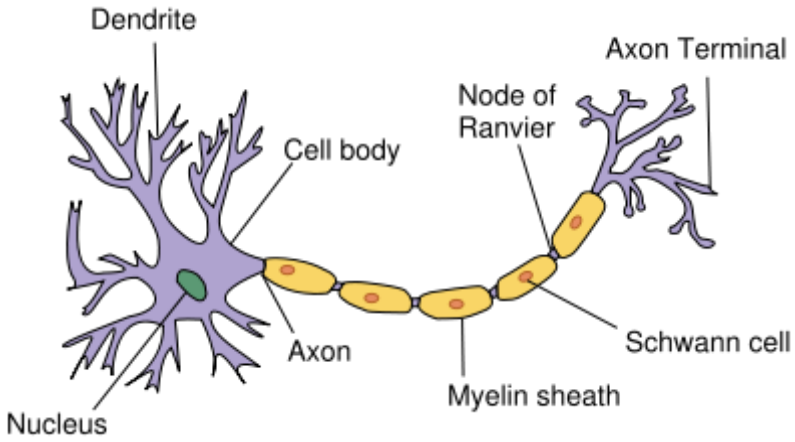

Figura 1. Estructura de una neurona. Fuente: "Anatomy and Physiology" por el programa de vigilancia, epidemiología y medición del instituto nacional del cáncer, Estados Unidos de América

Una neurona se conecta con otras a través de unas protuberancias conocidas como dendritas. La conexión entre neuronas se conoce como sinapsis, y permite la transmisión de impulsos eléctricos de una neurona a otra, ya sea de forma eléctrica o por medio del uso de sustancias secretadas en la sinapsis, conocidas como neurotransmisores.

El proceso de conducción eléctrica que ocurre al interior de una neurona, comprende cambios que se dan en la permeabilidad de la membrana plasmática; dichos cambios permiten el intercambio de iones sodio y potasio entre el interior y el exterior de dicha membrana. La activación de un mecanismo denominado bomba de sodio es lo que hace posible el desplazamiento de un potencial de acción a través de la extensión de la célula, hasta llegar al espacio sináptico y desencadenarse una reacción análoga en la siguiente neurona, produciéndose de esta manera el transporte de la información desde el sitio de origen (neurona aferente) hasta el sitio de destino.

Como se mencionó anteriormente, la potencia del sistema nervioso se puede apreciar en toda su extensión en el momento en que un gran número de neuronas

actúan formando circuitos. En el cerebro de un humano promedio se encuentran aproximadamente 10^{11} neuronas [1], y la evidencia demuestra que especialmente en el neocórtex existen zonas especializadas en el procesamiento de cierto tipo de información, compuestas por neuronas con estructura y funciones afines. Especialización y cantidad combinadas dotan al cerebro de su inmensa capacidad de procesamiento, y le confieren las características de tolerancia y adaptabilidad mencionadas.

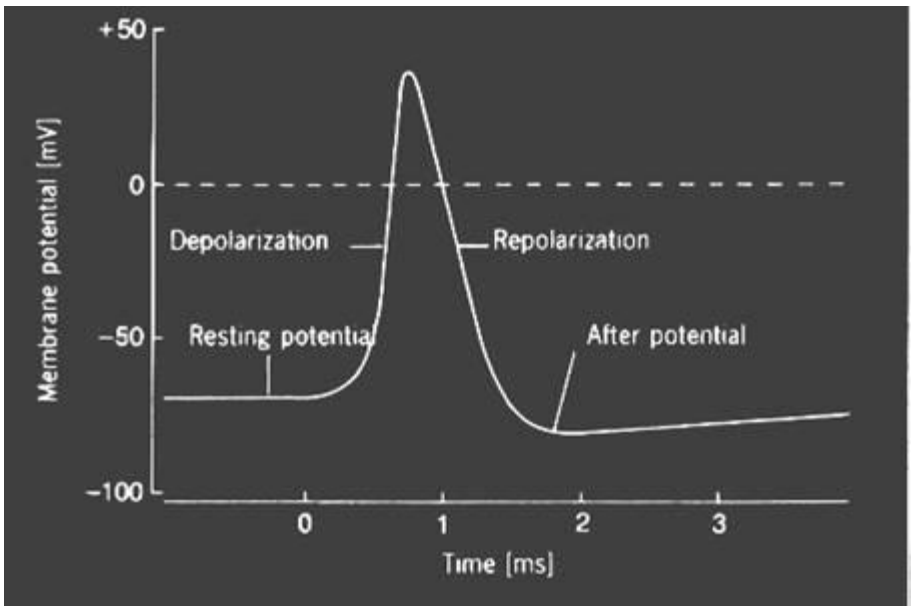

Figura 2. Potencial de acción. Tomado de http://electroneubio.secyt.gov.ar/Lamberti-Rodriguez_Hodgkin-Huxley.htm

Las sinapsis presentan una importancia muy grande en el estudio de las interacciones entre neuronas, y en la capacidad de almacenar recuerdos y aprender de la experiencia que presentamos los humanos. Hoy en día se cree que la memoria a largo plazo se genera por la modificación de las características físicas de las sinapsis; Es de destacar el hecho de que si el impulso eléctrico que una neurona recibe por una o más sinapsis no supera un umbral determinado, no se produce respuesta por parte de la neurona que recibe dicha excitación, lo que

permite que ciertas señales (posibles respuestas) sean desechadas si no cumplen con ciertos requisitos de potencia. Esta posición se fundamente en los estudios seminales de Donald Hebb, y el postulado conocido como el aprendizaje Hebbiano, que reza de la siguiente manera: “Cuando el axón de la célula A esta lo suficientemente cerca para excitar a la célula B y repetidamente toma parte en el proceso de activarla, algún proceso de crecimiento o cambio metabólico tiene lugar en una o ambas células de tal forma que la eficiencia de A, como una de las células activando B, se incrementa” [2]. Lo anterior puede entenderse como una expresión del comportamiento condicionado de Pavlov, en el que la respuesta a un estímulo se refuerza, por las sucesivas exposiciones a dicho estímulo. El principio de Hebb resultaría más adelante de gran importancia para el estudio de cierto tipo de redes neuronales.

El sistema nervioso y específicamente el cerebro son verdaderas maravillas que apenas comienza a ser comprendidas, y su estudio abre nuevos horizontes en el campo de investigación sobre sistemas inteligentes y adaptativos, como los basados en redes neuronales artificiales.

6.2. *Redes neuronales artificiales*

Las redes neuronales artificiales (en adelante redes neuronales) surgieron como un paradigma de computación, inspirado en la estructura del sistema nervioso de los animales superiores, donde un gran número de unidades altamente interconectadas entre sí conforman una estructura de procesamiento masivamente paralela. Estas conexiones le proveen a la red una propensión al almacenamiento de conocimiento basado en la experiencia, y una gran tolerancia a los fallos. Las redes neuronales representan un modelo de computación con poder equivalente a una máquina de Turing [3], y en los inicios del estudio del problema de la

computabilidad de una función, compitió con varios otros modelos de computación, incluyendo el modelo de Von Neumann, que a la postre resultó vencedor.

Algunas de las capacidades más espectaculares de este modelo de computación es la capacidad de “aprender” de la experiencia, almacenando conocimiento en los pesos asociados a las conexiones entre neuronas; el poder utilizar información ruidosa y/o incompleta para ser procesada con resultados satisfactorios, la capacidad de reconocer y organizar datos que no habían sido vistos con anterioridad, entre otras. Las redes neuronales presentan comportamientos (obviamente de forma muy rudimentaria) observados en animales superiores, por lo que se corrobora la validez del modelo biológico empleado.

Todo lo anterior hace de las redes neuronales una herramienta de gran aplicabilidad en problemas como el reconocimiento de patrones, clasificación y clusterización de datos, predicción, trabajo con información ruidosa, y en general en problemas donde hay la necesidad de modelar comportamientos de sistemas donde la relación entre las entradas y la salidas no es evidente y/o muestra una gran complejidad. Las redes neuronales hoy en día se usan en gran variedad de tareas, que van desde la estabilización y supresión de ruido en telefonía de larga distancia, el reconocimiento de imágenes, filtrado de correo no deseado (Spam), hasta la predicción del comportamiento de acciones en la bolsa, por mencionar unas cuantas [4].

Una red neuronal constituye una abstracción rudimentaria del modelo biológico, donde se consideran unidades fundamentales interconectadas entre sí llamadas neuronas. Una neurona en este contexto es una versión altamente simplificada de su contraparte biológica, que consta de entradas asociadas a un conjunto de pesos, un sumador, una función de transferencia, y una salida. De especial

importancia resulta la función de transferencia, puesto que esta es la que define el umbral que debe ser superado por la suma de las entradas para producir una salida. La representación matemática de una neurona artificial es la siguiente:

$$Y = f(w_1x_1 + w_2x_2 + w_3x_3 + \cdots + w_nx_n + b)$$

Donde w_i representa el peso asociado a la entrada x_1, b representa al sesgo o bias, un parámetro del que hablará más adelante; la salida Y viene dada por la función f o función de transferencia, la cual puede tener varias formas.

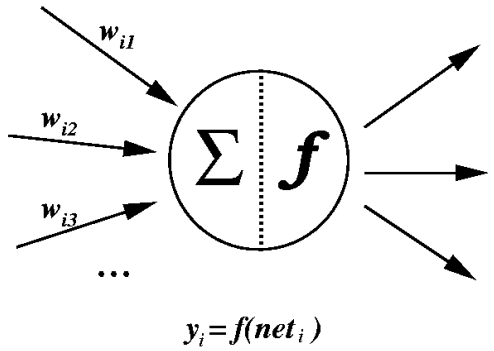

Figura 3. Modelo de neurona artificial. Tomado de http://www.idsia.ch/NNcourse/ann-overview.html

Una red neuronal puede estar compuesta de una o más neuronas, organizadas en una o más capas, y pueden existir conexiones cíclicas o no, dependiendo del modelo utilizado, y la arquitectura escogida le provee a la red de ciertas características que pueden ser útiles o no para resolver un problema determinado.

Al interactuar las neuronas, la red exhibe comportamientos que pueden llegar a ser muy complejos, a pesar de la relativa simplicidad de la neurona artificial. Debido a que una red neuronal se puede considerar como una agregación de funciones primitivas, se puede llegar a demostrar que una red neuronal con suficientes neuronas y capas es un aproximador universal de funciones, con precisión arbitraria. En esto, las redes neuronales exhiben similitud con las series

de Taylor y las series de Fourier; es más, las series de Fourier y Taylor pueden ser representadas como redes neuronales con funciones de transferencia específicas.

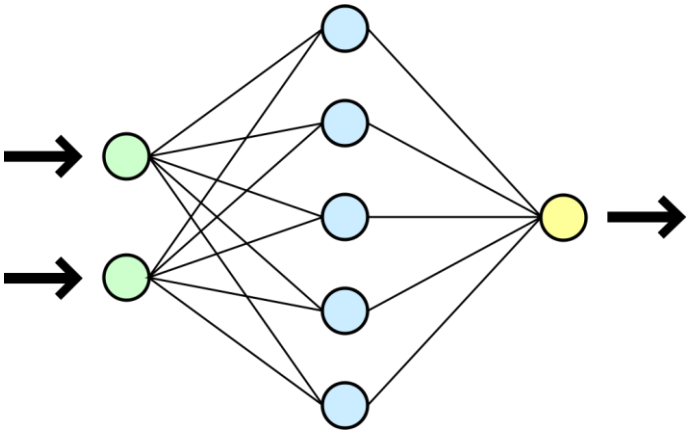

Figura 4. Representación de una red neuronal de 2 capas. Tomado de http://en.wikibooks.org/wiki/Artificial_Neural_Networks/Print_Version

Los algoritmos de aprendizaje utilizados para el entrenamiento de redes neuronales se pueden clasificar de la siguiente manera: algoritmos de entrenamiento supervisado y algoritmos de entrenamiento sin supervisión. Los algoritmos de entrenamiento supervisado requieren el uso de ejemplos del que debería ser el comportamiento adecuado de la red en presencia de entradas específicas. Ejemplos de entrenamiento supervisado son la regla de aprendizaje del perceptron, la regla delta (Widrow-Hoff) y el algoritmo de retro propagación (Backpropagation). De otro lado, los algoritmos de aprendizaje sin supervisión no requieren ejemplos de comportamiento adecuado, sino que a través de un proceso ya sea de refuerzo u auto-organización modifican la red de tal manera que aprenden a clasificar un espacio de entradas determinado.

A continuación se examinará de manera sucinta y general la evolución de los principales paradigmas relacionados a las redes neuronales y las diferentes

técnicas de entrenamiento que han aparecido a lo largo de la historia reciente del desarrollo en este campo.

6.3. *Evolución de los conceptos y paradigmas de las redes neuronales*

La era moderna de las redes neuronales comenzó en 1943 con la publicación del trabajo de McCulloch y Pitts, psiquiatra y matemático respectivamente. En esta publicación, describieron un modelo formal de neurona y mostraron que con un número suficiente de neuronas y sinapsis configuradas adecuadamente, una red de dichas neuronas podría computar cualquier función computable [5]. El trabajo de McCulloch y Pitts influenció en gran manera el trabajo de otros pioneros tales como John Von Neumann, y aun hoy sigue siendo de gran relevancia en el campo de las redes neuronales y la inteligencia artificial. En 1949, Donald Hebb publica su libro llamado The Organization of Behavior, donde se expresa por primera vez un postulado acerca de la modificación fisiológica de sinapsis dentro del proceso de aprendizaje. El trabajo de Hebb sirvió posteriormente de inspiración para el desarrollo de nuevas arquitecturas de redes neuronales y diversos trabajos no solo en el campo de la psicología, sino además en la ingeniería.

6.3.1. Perceptron

En 1958 Frank Rosenblatt presentó el perceptron, un modelo de red neuronal basado en el modelo de neurona artificial McCulloch-Pitts, e introdujo un nuevo método de entrenamiento supervisado llamado Perceptron Learning Rule (regla de aprendizaje del perceptron) [6]. El modelo de Rosenblatt se diferenciaba del modelo de McCulloch-Pitts en la introducción de pesos asociados a las entradas a la neurona, y su aparición causó toda clase de expectativas en la comunidad científica, debido a su aplicación en el reconocimiento de patrones de forma

efectiva. Uno de las características más importantes del perceptron y su regla de aprendizaje es que, dado un perceptron y un problema de clasificación de patrones, se puede demostrar que el perceptron podrá converger a una solución en un número de pasos finito, dado que exista un conjunto de pesos que resuelva dicho problema de clasificación. La estructura del perceptron se puede apreciar en la figura 5.

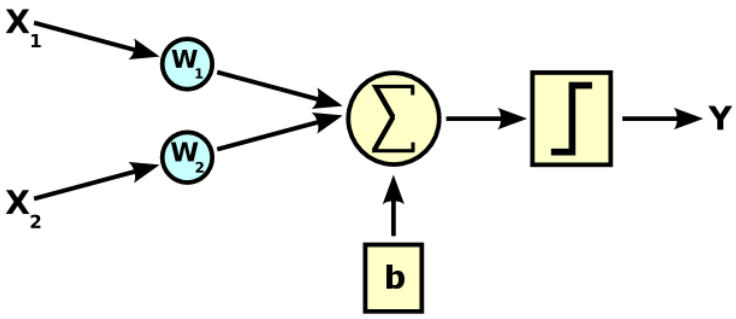

Figura 5. Estructura de un Perceptron. Derivado de http://es.wikipedia.org/wiki/Perceptrón

El perceptron se caracteriza por tener, además de entradas ponderadas, un término denominado bias que altera el comportamiento de la neurona de forma muy particular. Una neurona con un bias igual a cero está limitada en cuanto al número de problemas que puede resolver, en contraposición a una con un bias diferente de cero, como se verá más adelante. Además, de especial importancia es la función de activación del perceptron, la función Hardlim (limitador fuerte), que se define de la siguiente manera:

$$y = f(x) = \begin{cases} 0, & x < 0 \\ 1, & x \geq 0 \end{cases}$$

Teniendo en cuenta lo anterior, podemos definir formalmente un perceptron de una neurona como:

$$a = Hardlim(w_1x_1 + w_2x_2 + w_2x_2 + w_3x_3 + \cdots + w_nx_n + b)$$

De lo anterior podemos notar que el perceptron solo produce valores binarios, cero y uno, y que el elemento bias se comporta como un modificador del umbral de la neurona, elevándolo o disminuyéndolo de acuerdo a su valor. Si el bias no estuviese presente o fuese igual a cero, la suma ponderada de las entradas deberá ser mayor o igual a cero para que la neurona produjera un uno como salida.

Los elementos que componen la neurona de un perceptron pueden ser relacionados en una función lineal, que suministra información adicional acerca de su comportamiento. Dicha función tiene la siguiente forma:

$$y = w_1x_1 + w_2x_2 + w_2x_2 + w_3x_3 + \cdots + w_nx_n + b$$

Esta función constituye la *frontera de decisión* del perceptron, y define la separación que el perceptron hará del espacio de entradas, por ende su clasificación. Toda entrada que quede situada por debajo de dicha frontera provocara un cero como respuesta por parte de la red, mientras que las entradas situadas por encima ocasionarán una respuesta igual a uno.

El aprendizaje del perceptron es *supervisado*, es decir, necesita de ejemplos de comportamiento adecuado para cada entrada del set de entrenamiento, el cual debe tener la siguiente forma:

$\{p_1, t_1\}, \{p_2, t_2\}, \{p_3, t_3\} \ldots \{p_n, t_n\}$ donde p_i es cada entrada a la red, y t_i es la correspondiente salida deseada. Cada vez que se presenta una entrada p_i a la

red, se compara la salida producida por dicha entrada con el valor deseado t_i, y las neuronas de la red son ajustadas de acuerdo a esa diferencia, utilizando las siguientes ecuaciones:

$$w_{nuevo} = w_{ant} + \varepsilon p$$
$$b_{nuevo} = b_{ant} + \varepsilon$$

donde w_{nuevo} representa el nuevo valor del peso, w_{ant} el valor antiguo, p la entrada asociada al peso w y ε el error del perceptron; que está definido como

$$\varepsilon = t - a$$

donde t es la salida deseada y a es la salida real para la entrada p.

Mediante la aplicación de esta regla, el perceptron convergerá a una solución en un número finito de pasos dado que exista un conjunto de pesos que efectivamente resuelva el problema [1].

A pesar de ser un reconocedor de patrones inherentemente, el perceptron sufre de varias limitaciones, como fue descubierto por Minsky y Papert en su trabajo de 1969 titulado Perceptrons. Estos investigadores lograron demostrar que un perceptron no era capaz de implementar ciertas funciones elementales, como la función lógica XOR [7]. En general, se pudo establecer que el perceptron solo podía solucionar problemas linealmente separables, es decir, el grupo de problemas donde el espacio de entradas puede ser dividido en 2 por un hiperplano. El trabajo de Minsky y Papert efectivamente acabó con el interés en la investigación sobre redes neuronales al poner de manifiesto las grandes limitaciones del perceptron, y al afirmar erróneamente de que no había ninguna

razón para creer que un diseño multicapa pudiese superar dichas limitaciones. Solo más de una década después se pudo implementar un algoritmo eficiente de entrenamiento para el perceptron multicapa.

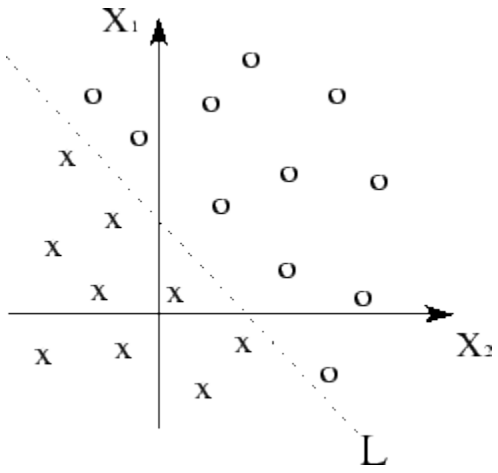

Figura 6. Problema linealmente separable. Tomado de http://www.ece.utep.edu/research/webfuzzy/docs/kk-thesis/kk-thesis-html/node19.html

A pesar de sus limitaciones, el perceptron constituye un clasificador de patrones eficiente, e inspiró el gran desarrollo que sobrevendría en la investigación sobre inteligencia artificial, neurodinámica, y redes neuronales.

6.3.2 Adaline

Adaline significa Adaptive Linear Element, y representó un gran adelanto en el campo de redes neuronales. En principio, la arquitectura adaline es muy similar al perceptron, siendo la única diferencia el hecho de que las neuronas de adaline usan una función de transferencia lineal (la cual llamaremos *Purelin*), en vez del limitador fuerte de sus contrapartes en el perceptron. La forma de la función lineal es:

$$y = x$$

La función identidad, de tal forma que la salida es igual a la entrada.

En 1960, Bernard Widrow junto a su estudiante Marcian Hoff desarrollaron el Adaline, un tipo de red neuronal de entrenamiento supervisado que aun hoy se sigue usando con gran éxito en varias aplicaciones. Tal vez el adelanto más importante de adaline fue el algoritmo de entrenamiento, llamado Delta Rule (Regla Delta), algoritmo LMS (Least Mean Square, mínimo error cuadrado en español) o algoritmo Widrow-Hoff, en honor a sus desarrolladores. Este algoritmo resulta ser mucho más poderoso que la regla de entrenamiento del perceptron, por lo que adaline suele ser una red más robusta y con menor sensibilidad al ruido que el perceptron.

La salida de una red adaline de una sola neurona es de la forma:

$$a = Purelin(w_1x_1 + w_2x_2 + w_2x_2 + w_3x_3 + \cdots + w_nx_n + b)$$

De lo anterior podemos extraer el hecho de que la salida de la red se puede encontrar en el intervalo $(-\infty, +\infty)$.

De igual forma que el perceptron, un adaline forma una frontera de decisión con base a sus parámetros internos. La forma de la frontera de decisión es similar a la del perceptron y está dada por la expresión:

$$y = w_1x_1 + w_2x_2 + w_2x_2 + w_3x_3 + \cdots + w_nx_n + b$$

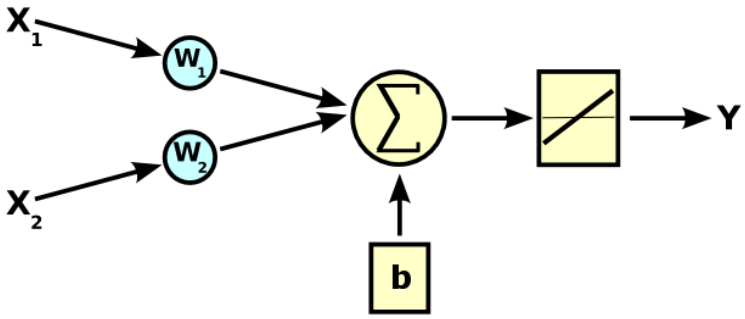

Figura 7. Estructura de ADALINE. Derivado de http://es.wikipedia.org/wiki/Perceptrón

El algoritmo LMS como su nombre lo indica, busca minimizar el cuadrado del error de la red, a través de una búsqueda secuencial sobre una superficie conformada por los parámetros internos de la red llamada *superficie de desempeño* [1]; por lo que se trata de una forma de entrenamiento supervisado. Este algoritmo pertenece a la familia de los métodos tradicionales de optimización, en los cuales se persigue encontrar un mínimo global de una función de costo determinado. Más adelante se verá que el algoritmo Backpropagation (perceptron multicapa) es solo la generalización del algoritmo Widrow-Hoff.

La regla de aprendizaje para los pesos y sesgos (biases) de cada neurona del adaline en la iteración k es:

$$w_{k+1} = w_k + 2\alpha\varepsilon_k p_k$$
$$b_{k+1} = b_k + 2\alpha\varepsilon_k$$

donde w_{k+1} es el nuevo peso producto del aprendizaje, w_k el peso en la iteración k, p_k la entrada a la red en la iteración k, ε_k el error de la iteración k definido como:

$$\varepsilon = t - a$$

donde t es la salida deseada y a es la salida real de la red en la iteración k; y el término α representa un concepto no visto hasta el momento, llamado tasa de aprendizaje. La tasa de aprendizaje es un parámetro que afecta la capacidad y la velocidad de la red para aprender de la experiencia, puesto que dicta el tamaño de los incrementos de búsqueda que el algoritmo utilizara en su búsqueda del menor error. Este parámetro es de suma importancia para este tipo de algoritmos de entrenamiento, puesto que si es muy pequeño la red demorara mucho tiempo en alcanzar errores pequeños, mientras que si es muy grande conducirá a grandes oscilaciones en la trayectoria de búsqueda, y eventualmente a la imposibilidad de aprender.

De igual forma que el perceptron, el uso efectivo de adaline está limitado a los problemas de clasificación linealmente separables. A pesar de lo anterior, adaline ha sido un tipo de red neuronal tremendamente exitoso en diversos campos de la ingeniería, en parte debido a la gran robustez del algoritmo de entrenamiento que utiliza, puesto que por su estructura de minimización permite alejar la frontera de de decisión lo más posible de los datos, mejorando enormemente su tolerancia al ruido en comparación al perceptron. Por otro lado, adaline sentó las bases para el entrenamiento de perceptrons multicapa, a partir de su revolucionario algoritmo de aprendizaje.

6.3.3 Aprendizaje competitivo

Hasta ahora se han considerado ejemplos de *entrenamiento supervisado*, donde además de las entradas se le suministra a la red ejemplos de comportamiento adecuado para dichas entradas. Procederemos a revisar uno de los paradigmas más famosos y utilizados dentro del grupo de algoritmos de entrenamiento no supervisado: el aprendizaje competitivo.

Luego del trabajo de Minsky y Papert en 1969 el desarrollo sobre redes neuronales similares o basadas en el perceptron quedó relegado a un segundo plano, y excepción de investigadores de las áreas de la psicología y las neurociencias el interés general de la comunidad científica y de ingeniería decayó grandemente en la década subsiguiente. Podemos decir a partir de lo anterior que desde el punto de vista de la ingeniería, la década de los setenta fue de poquísima actividad [1]. De la década de los setenta podemos rescatar el desarrollo de los sistemas auto-organizativos, que reflejaban el comportamiento de ciertas áreas del sistema nervioso central, donde neuronas con tareas similares (procesamiento de la audición, visión, habla por ejemplo) se encuentran agrupadas en lugares específicos. La inhibición lateral fue un fenómeno propuesto para explicar por qué ciertas áreas del cerebro se activaban solo en presencia de un estímulo particular, mientras que áreas subyacentes no; y este principio terminó convertido en parte esencial del *aprendizaje competitivo.* Un grupo de neuronas compiten entre sí para determinar cuál de ellas produce una respuesta más intensa al estímulo (entrada), luego los parámetros internos de la neurona ganadora son modificados de tal manera que para posteriores exposiciones al mismo estímulo, su respuesta sería aun más intensa.

El principio del aprendizaje competitivo ha sido utilizado por investigadores como Stephen Grossberg, Christoph von der Malsburg y Teuvo Kohonen en diversas arquitecturas de redes neuronales de entrenamiento sin supervisión con gran éxito. De especial importancia resultan los modelos de Grossberg (Adaptive Resonance Theory - ART) y Kohonen (Self Organizing Maps SOMs), el primero diseñado para proveer de un modelo biológicamente plausible, mientras que el segundo orientado más específicamente para aplicaciones de ingeniería.

El modelo más básico de red competitiva está compuesto por 2 capas. La primera se encarga de correlacionar la entrada con un conjunto de categorías (las

neuronas de la capa), y obtener las respuestas de cada neurona; y la segunda capa se encarga de establecer una “competencia” entre las neuronas de la primera capa, de tal forma de que la ganadora de la competencia sea la neurona con la mayor respuesta a la entrada.

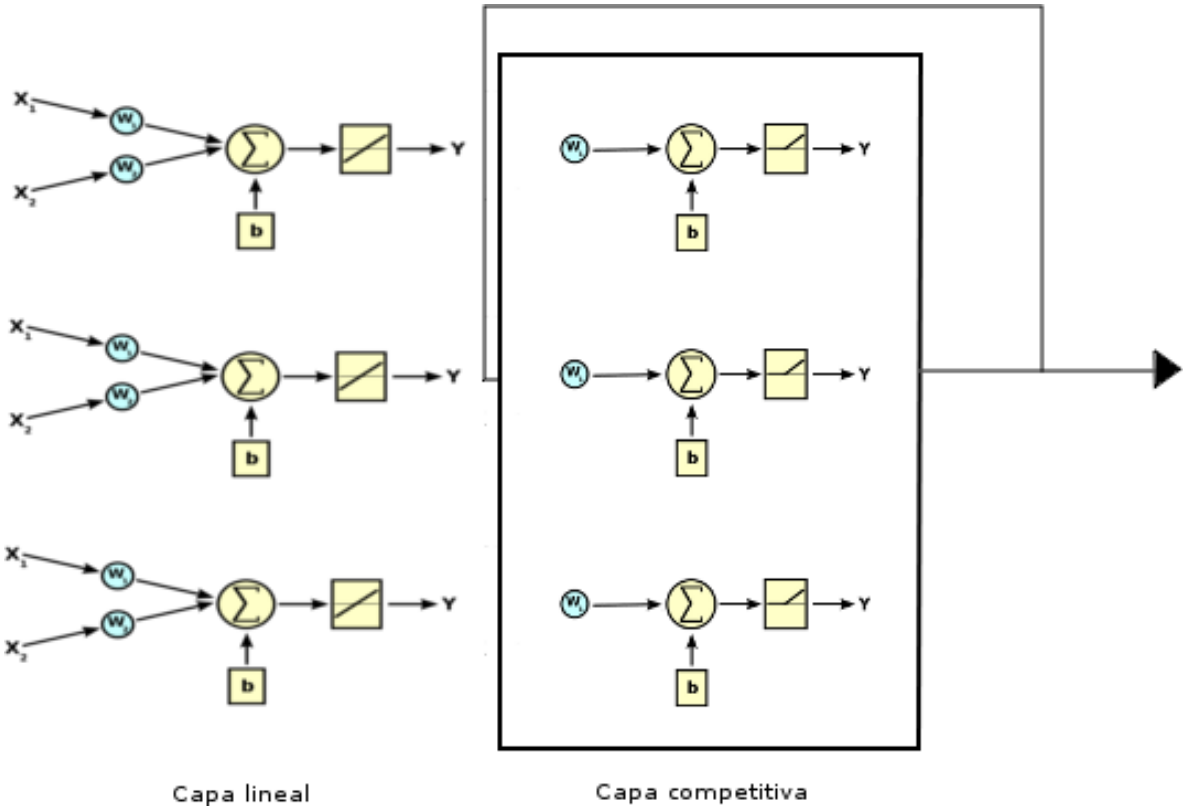

Figura 8. Red competitiva. Derivado de http://es.wikipedia.org/wiki/Perceptrón

Matemáticamente, el estudio de las redes competitivas se realiza en términos de vectores y matrices. La correlación efectuada entre la entrada y las neuronas de la primera capa es equivalente al producto punto entre el vector correspondiente a las entradas y el vector correspondiente a los pesos de cada neurona. De acuerdo a la definición del producto punto, el resultado será máximo si el ángulo entre los vectores es cero, es decir apuntan en la misma dirección (siempre que los

vectores tengan la misma magnitud); de esta manera podemos encontrar la neurona que se encuentra más “cerca” de la entrada.

En la segunda capa se produce la competición entre los resultados de la primera capa. Los pesos de las neuronas pertenecientes a esta capa se inicializan acorde a los elementos de la siguiente matriz de tamaño $m\ x\ m$, donde m es el número de neuronas de la primera capa:

$$A = \begin{pmatrix} 1 & \cdots & -\varepsilon \\ \vdots & \ddots & \vdots \\ -\varepsilon & \cdots & 1 \end{pmatrix}$$

donde los elementos de cada fila representan los pesos de cada neurona de la segunda capa. Los elementos pertenecientes a la diagonal principal son iguales a uno, mientras que los elementos restantes son iguales a un valor $-\varepsilon$, donde ε se encuentra en el siguiente intervalo:

$$0 < \varepsilon < \frac{1}{S-1}$$

siendo S el número de elementos del vector que contiene las entradas. La segunda capa de la red competitiva es recurrente, y se continuará con la recurrencia hasta que solo haya una neurona con salida diferente a cero. La matriz A produce el fenómeno de inhibición lateral, donde cada neurona se excita a si misma al tiempo que inhibe la respuesta de las demás.

Los pesos de neurona de la primera capa que resulta ser la ganadora de la competición (llevada a cabo en la segunda capa) se modifican de acuerdo a la regla de Kohonen (nombrada así en honor al investigador finés Teuvo Kohonen, mencionado anteriormente), la cual tiene la siguiente forma:

$$w_{nuevo} = (1 - \alpha)w_{ant} + \alpha p$$

donde w_{nuevo} es el peso i de la neurona luego de ser actualizado, w_{ant} el peso antes de ser actualizado, p el elemento i del vector de entradas, y α la tasa de aprendizaje. La regla de Kohonen es uno de los algoritmos de entrenamiento no supervisado más famosos y más utilizados desde su invención.

A partir del modelo básico de red competitiva se han propuesto varios tipos de red neuronal de entrenamiento sin supervisión, siendo tal vez el más famoso el SOM (Self Organizing Map) de Kohonen. Todos comparten los mismos elementos básicos de la red competitiva y se han usado con gran éxito en diferentes campos de la ingeniería y afines.

6.3.4 Multilayer Perceptron (Perceptron Multicapa)

Los años ochenta significaron un resurgimiento de la investigación sobre redes neuronales, alcanzándose grandes logros en el campo. Además, durante este periodo se logró desarrollar exitosamente los primeros modelos de red neuronal multicapa, junto a sus respectivos algoritmos de entrenamiento. En el año de 1985, David Rumelhart, Geoffrey Hinton, Ronald Williams, David Parker y Yann Le Cun re-descubrieron independientemente el algoritmo Backpropagation (retro-propagación), el cual se convirtió a la postre en el algoritmo más utilizado en el entrenamiento de redes multicapa y dio origen al modelo de red neuronal más famoso, el perceptron multicapa (Feedforward Multilayer Perceptron).

El trabajo de Minsky y Papert demostró rigurosamente las fortalezas y debilidades de los perceptrons de una sola capa, sin embargo los autores supusieron incorrectamente que la adición de nuevas capas no iba ayudar a resolver dichas

limitaciones. Esta suposición desmotivó la investigación sobre redes neuronales y retrasó enormemente el desarrollo en el campo específico de las redes de entrenamiento supervisado. Muchos de los conocimientos necesarios para el desarrollo de algoritmos de entrenamiento para redes neuronales multicapa, específicamente del algoritmo de retro-propagación, habían sido desarrollados tiempo atrás, y la primera descripción del algoritmo Backpropagation se estableció en el trabajo de tesis doctoral de Paul Werbos en 1974. Sin embargo Rumelhart et al. fueron los primeros en proponer dicho algoritmo en el contexto del aprendizaje de redes neuronales multicapa.

El problema inicial sobre el entrenamiento de redes multicapa planteado por Minsky y Papert fue denominado el “problema de la asignación de créditos”, y se refería fundamentalmente a la imposibilidad de establecer la responsabilidad de las neuronas de las capas intermedias de la red, sobre el error total. Ese es el punto específico que el algoritmo Backpropagation resuelve: usando cálculo diferencial, es posible cuantificar la contribución sobre el error de cada neurona en la red, y alterar sus parámetros con miras a minimizar el error. Este algoritmo puede ser visto como una generalización del algoritmo LMS (Adaline), y al ser aplicado en una red de una sola capa, el algoritmo Backpropagation toma su forma.

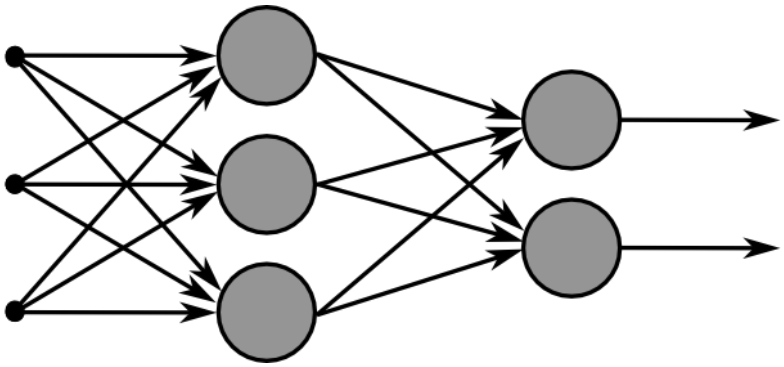

Figura 9. Red Neuronal Feedforward Multicapa. Tomado de http://en.wikibooks.org/wiki/Artificial_Neural_Networks/Print_Version.

Los perceptron multicapa son redes neuronales de entrenamiento supervisado, y se usan ampliamente en problemas de reconocimiento de patrones (identificación biométrica, visión artificial, reconocimiento del lenguaje hablado y escrito), aproximación de funciones (control automático), predicción de series temporales (mercado de valores), entre otros. Debido a la naturaleza del algoritmo Backpropagation, la función de transferencia de las neuronas que la componen debe ser diferenciable y suele ser la función logística sigmoidea, la función tangente hiperbólica o la función identidad (usualmente en la capa final). La tangente hiperbólica y la función logística sigmoidea sustituyen al limitador fuerte del perceptron de una sola capa, puesto que restringen la salida al intervalo (-1,1). La forma de dichas funciones se detalla a continuación:

Función logística sigmoidea:

$$y = \frac{1}{1 + e^{-x}}$$

Función tangente hiperbólica:

$$\frac{e^{2x} - 1}{e^{2x} + 1}$$

Función identidad:

$$y = x$$

Al ser una red de entrenamiento supervisado, el perceptron multicapa requiere de un conjunto de entradas con sus respectivas salidas para la fase de entrenamiento, y usualmente un número importante de repeticiones del proceso que conllevan la presentación de todos los datos de entrenamiento hasta alcanzar un error tolerable.

En cada iteración, el ajuste de los pesos y los bias de las neuronas de la red, en la iteración k, deberá ser de la siguiente forma:

$$w_{k+1} = w_k - \alpha S_m a$$
$$b_{k+1} = b_k - \alpha S_m$$

donde α es la tasa de aprendizaje de la red, a es la entrada a la neurona, y S_m es la contribución al error de las neuronas de la capa m, la cual se define de la siguiente manera:

Para la última capa de la red:

$$S_M = -2\frac{dF}{dx}(t - a)$$

Para cualquier otra capa:

$$S_m = \frac{dF}{dx}c$$

donde t es la salida esperada, a la salida real, c es el producto de la contribución S_{m+1} al error de la capa anterior y los pesos de la capa siguiente asociados a la salida de esta neurona y $\frac{dF}{dx}$ es la derivada de la función de transferencia de la neurona en cuestión.

6.4 Actualidad del campo e investigación reciente

Las características de las redes neuronales han permitido su utilización en diversas aplicaciones de inteligencia artificial tales como, reconocimiento del lenguaje hablado y escrito, reconocimiento de imágenes, reducción del ruido y estabilización de líneas telefónicas de larga distancia, control automático, predicción de sistemas caóticos, reconocimiento de patrones, filtrado de Spam, sistemas médicos de diagnóstico, detección de fraudes bancarios; entre otros, probando ser muchas veces superiores a las técnicas conocidas.

En los últimos tiempos las redes neuronales se han venido utilizando junto a herramientas de análisis espectral en la interpretación de señales cerebrales para el control mental de equipo electrónico [8]. También se ha registrado el uso de redes neuronales en el campo de la bioinformática, específicamente en el análisis de proteínas y su dinámica en diferentes ambientes [9]. Las redes neuronales han sido aplicadas en años recientes al campo del control, operación, diseño y estudio de condiciones de seguridad de sistemas de electricidad [10] con gran éxito.

Por otro lado el uso de nuevas técnicas de optimización para el entrenamiento de redes como Particle Swarm Optimization [11], algoritmos evolucionarios, algoritmos genéticos, entre otros; han permitido avances importantes en la capacidad de las redes neuronales de adaptarse a nuevas tareas, incrementando su eficacia y eficiencia. En general, el campo de la investigación en redes neuronales se mantiene bastante activo, y a medida que hardware más poderoso

aparece, más posibilidades se abren para los investigadores en redes neuronales; y cada vez más estas se incorporan en aparatos “inteligentes” de uso diario.

6.5 Análisis y diseño orientado a objetos

El análisis y diseño orientado a objetos (OOA/D, por sus siglas en inglés) nace en los años 80, de la mano del estudio de lenguajes de programación tales como ADA y SmallTalk, y el término fue acuñado probablemente por Grady Booch en 1982, en su trabajo seminal titulado Object-Oriented Design[13]. En dicho trabajo Booch muestra una metodología de diseño para ADA, sin embargo los conceptos tratados probaron ser de aplicabilidad a un sinnúmero de lenguajes y sentaron las bases para el OOA/D moderno.

De esta forma, OOA/D se fue erigiendo como una disciplina en sí misma, abstraída de los lenguajes en los que los sistemas se implementan. Luego de Booch , muchos autores empezaron a escribir sus ideas y aproximaciones al tema, entre ellos Bertrand Meyer, quien en 1988 publicó su libro *Object Oriented Software Construction*[14], Peter Coad con *Object-Oriented Analysis*[15] y *Object-Oriented Design*[16], publicados en 1990 y 1991 respectivamente. De igual forma, el paradigma del diseño motivado por responsabilidades fue puesto de relieve en el trabajo de Wirfs-Brock et al titulado *Designing Object-Oriented Software*[17]. En 1991 se publicaron los trabajos de Jim Rumbaugh et al y Booch titulados *Object-Oriented Modeling and Design*[18] y *Object-Oriented Design with Applications*[19] respectivamente.

Todos los trabajos tenían como intención el proveer a los desarrolladores de software orientado a objetos, de técnicas y principios que les permitiesen construir sistemas robustos, extensibles y que cumplieran con los requerimientos del

cliente. Sin embargo, los métodos propuestos tenían obvias diferencias en su concepción y aplicación, y surgieron intentos para combinar las fortalezas de los diferentes métodos. En 1994 Booch y Rumbaugh intentaron unir sus 2 métodos de OOA/D (OMT y Booch) y a partir de este esfuerzo nació el primer borrador de lo que hoy conocemos como UML (Unified Modeling Language). Posteriormente se unió al grupo Ivar Jacobson, el creador del método de OOA/D Objectory y autor del libro Object-Oriented Software Engineering, y UML tomó la forma de notación para la descripción de software orientado a objetos. A estos 3 personajes se les conoce como “The Three Amigos”. Posteriormente la OMG (Object Managment Group), un ente de estándares industriales en el mundo de software orientado a objetos, decide adoptar UML como el estándar de facto para la descripción y diagramación de sistemas orientados a objetos.

6.5.2 Patrones de diseño orientado a objetos

La naturaleza del desarrollo de software orientado a objetos se basa en el reúso – la re-utilización del código escrito, mediante la utilización de principios tales como la herencia, el polimorfismo y la encapsulación. Siguiendo estos principios y aplicándolos al diseño orientado a objetos se llega a los patrones de diseño, que no son más que soluciones probadas para problemas comunes de diseño, que pueden ser aplicadas en diferentes contextos, y la utilización de patrones de diseño tiene como meta garantizar metas tales como la robustez, extensibilidad, y desacoplamiento del software. De manera más formal un patrón de diseño es “un patrón que cuya forma esta descrita a través de elementos de construcción del diseño de software, por ejemplo objetos, clases, herencia, agregación y relaciones de uso”[20].

Los patrones de diseño se originaron a partir de conceptos de la arquitectura, expresados por primera vez por Christopher Alexander en 1977. A partir de estas definiciones, en 1987 Kent Beck y Ward Cunningham comenzaron a experimentar

con sus contrapartes conceptuales en el diseño de software orientado a objetos, dando origen a los patrones de diseño, y muchos autores se sumaron al estudio y desarrollo de patrones de diseño. Los patrones de diseño ganaron popularidad en el medio tras el trabajo de Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides (conocidos popularmente como the Gang Of Four - *GOF*) titulado *Design Patterns: Elements of Reusable Object-Oriented Software*[21] en 1995.

Según [21], los patrones de diseño se pueden describir a través de los siguientes elementos:

- Nombre y clasificación del patrón: El nombre del patrón transmite la esencia del patrón de manera sucinta. La clasificación del patrón se hace de acuerdo a dos criterios: Propósito del patrón (creacional, estructural y comportamiento) y ámbito (clase y objeto).
- Intención: Una afirmación corta que responde las siguientes preguntas: ¿Qué hace el patrón? ¿Cuál es la lógica detrás del patrón y su intención? ¿Qué problema o situación de diseño en particular pretende atacar el patrón?
- Otros nombres: Nombres alternativos para el patrón de diseño.
- Motivación: Un escenario que ilustra un problema de diseño y como las estructuras de clases y objetos del patrón resuelven el problema. El escenario ayuda a comprender la descripción más abstracta que se incluye en los siguientes elementos.
- Aplicabilidad: ¿Cuales son las situaciones en las que el patrón puede ser aplicado? ¿Qué ejemplos hay de malos diseños que el patrón pueda atacar? ¿Cómo se pueden reconocer estas situaciones?
- Estructura: Una representación gráfica de las clases pertenecientes al patrón y de sus interacciones en UML.

- Participantes: Las clases y objetos que participan en el patrón de diseño y sus responsabilidades.
- Colaboración: Cómo los participantes colaboran para llevar a cabo sus responsabilidades.
- Consecuencias: ¿Cómo el patrón soporta sus objetivos? ¿Cuáles son las concesiones, elecciones y resultados implicados en la utilización del patrón? ¿Qué aspecto de la estructura del sistema permite el patrón cambiar de manera independiente?
- Implementación: ¿Qué peligros, consejos o técnicas se deberían tener presentes al momento de implementar el patrón? ¿Existen problemas relacionados al patrón, inherentes a los lenguajes de programación?
- Código de ejemplo: Fragmentos de código que ilustren cómo se debería implementar el patrón (Gamma et al en el original se refieren a C++ o SmallTalk, pero el lenguaje puede ser cualquiera).
- Usos conocidos: Ejemplos del patrón encontrados en sistemas reales.
- Patrones relacionados: ¿Qué patrones de diseño se relacionan al actual? ¿Cuáles son las diferencias más importantes? ¿Con que otros patrones debería ser usado?

Una lista completa de los patrones originales de *GOF* se proporciona en [21].

6.6 Otras librerías y proyectos de código libre sobre redes neuronales (Java).

El aumento de popularidad del software de código abierto en los últimos años ha llevado al desarrollo de muchos proyectos de corte científico bajo esta modalidad de distribución. A través de internet, se distribuyen aplicaciones y librerías de software con diversos propósitos, entre ellos el uso de redes neuronales. Se realizó una búsqueda en la web para hacer un compendio de los proyectos más relevantes al respecto y los resultados se resumen en la tabla 1.

Librería	Lenguaje	¿Incluye editor visual?	Modelos soportados	Website
JOONE	Java	Si	Arquitectura flexible, provee de bloques de construcción para que el usuario cree el modelo deseado.	http://sourceforge.net/projects/joone/
Neuroph	Java	Si	• Adaline. • Perceptron. • Multilayer Perceptron. • Hopfield network. • Bidirectional associative memory. • SOM. • Hebbian network. • Maxnet. • Competitive network. • Instar. • Outstar. • RBF network.	http://neuroph.sourceforge.net/
Encog	Java	No	• Adaline • Adaptive Resonance Theory 1 (ART1) • Bidirectional Associative Memory (BAM) • Boltzmann Machine • Counterpropagation Neural Network (CPN) • Elman Recurrent Neural Network • Perceptron • Hopfield Neural Network • Jordan Recurrent Neural Network • Radial Basis Function Network • Recurrent Self Organizing Map (RSOM) • SOM (Kohonen)	http://code.google.com/p/encog-java/

Tabla 1. Librerías open source para redes neuronales escritas en Java.

7 DISEÑO E IMPLEMENTACIÓN DE JANE Y NEURALSTUDIO

La metodología de investigación propuesta, se basa en la construcción de sistemas para la validación de hipótesis planteadas, como parte de un proceso iterativo, que involucra la puesta en práctica de modelos conceptuales definidos con anterioridad. El primer paso fue la construcción de un marco teórico, y luego el desarrollo de un sistema de prueba para la validación de dicho marco, de acuerdo a [12]. Durante el desarrollo de esta investigación, se construyó una pequeña aplicación de prueba para validar los conceptos que le dieron forma a JANE. Luego de validados, se procedió a la estructuración definitiva de tanto la librería (JANE) como la suite didáctica (NeuralStudio).

Se mencionarán los aspectos relevantes de dicho proceso, luego de que el marco teórico fue validado a través del sistema de prueba, y las experiencias obtenidas en el proceso.

Se puede comenzar afirmando que en el momento, existen varias alternativas bastante buenas para el desarrollo de software (léase lenguajes de programación/plataformas), entre las que podemos mencionar .NET, Java, C++, Python, etc. Sin embargo, la única alternativa realmente multiplataforma en estos momentos, con un excelente soporte OOP (Object Oriented Programming) es Java, por lo que es la elección más o menos obvia para adelantar el desarrollo de los sistemas planteados. La naturaleza de Java permitirá construir una aplicación que pueda ser ejecutada en un gran número de sistemas operativos tales como Unix, Linux, Mac OSX, Windows, entre otros, sin que esto signifique la necesidad de realizar algún cambio en el código ya escrito.

Ya hablando específicamente de Java, los requerimientos de la librería y la aplicación implicaron la utilización extensiva del Collections Framework,

especialmente de la clase ArrayList, Generics, Threads, Swing, entre otros. Más adelante se visitarán estos aspectos en detalle.

De acuerdo a los objetivos planteados, se revisarán los aspectos relevantes de la construcción del sistema, empezando por JANE y posteriormente se seguirá con NeuralStudio.

7.1 JANE

JANE significa Java Advanced Neural Engine (Máquina Neuronal Avanzada de Java), y apunta a ser una librería multipropósito para el desarrollo de sistemas que requieran de redes neuronales. JANE contiene implementaciones de los modelos neuronales más relevantes e importantes dentro del panorama de las redes neuronales. Además, se diseñó teniendo en cuenta la posibilidad de incorporar nuevos modelos como parte de la evolución de la librería.

Las metas de diseño estaban orientadas hacia conseguir una librería extensible y eficaz. Además, se persiguió encapsular todas las dependencias en componentes altamente cohesionados.

7.1.1 Modelos neuronales escogidos.

Para la escogencia de los modelos que iban a integrar la primera versión de la librería, se analizó de acuerdo al estado del arte la importancia relativa de los modelos neuronales existentes, utilizando criterios tales como grado de aplicación práctica, importancia histórica, complejidad de implementación (en términos de la complejidad de la arquitectura, si era o no recurrente y la dificultad de los cálculos relacionados al entrenamiento) y posibilidad de reutilización de la arquitectura para la posterior inclusión de nuevos modelos que constituyan refinamientos y/o evoluciones de dicha arquitectura (Tabla 1).De acuerdo a lo anterior, se escogieron los siguientes modelos neuronales: Perceptron, Adaline, Multilayer Perceptron, la red de aprendizaje competitivo y los mapas auto-organizativos de Kohonen (SOM).

MODELO/CRITERIO	APLICACIÓN PRÁCTICA	IMPORTANCIA HISTÓRICA	COMPLEJIDAD DE IMPLEMENTACIÓN	REUTILIZACIÓN
PERCEPTRON	Baja	Muy Alta	Baja	Media
ADALINE	Alta	Alta	Baja	Media
MULTILAYER PERCEPTRON	Muy Alta	Muy Alta	Alta	Alta
RED COMPETITIVA	Alta	Media	Media	Alta
NEOCOGNITRON	Alta	Baja	Alta	Media
ADAPTIVE RESONANCE THEORY (ART)	Alta	Alta	Alta	Media
KOHONEN SELF ORGANIZED FEATURE MAPS (SOM)	Alta	Alta	Alta	Media
HOPFIELD	Media	Media	Alta	Baja

Tabla 2. Criterios de escogencia para JANE

De acuerdo a lo anterior, se incluyeron implementaciones de los siguientes algoritmos de entrenamiento:

- Regla del Perceptron (Perceptron)
- LMS (Least Mean Square) o Algoritmo Widrow-Hoff (ADALINE)
- Backpropagation (Multilayer Perceptron)
- Regla de Kohonen (Red de aprendizaje competitivo y SOM)

Y en consecuencia a esto, implementaciones de las siguientes funciones de transferencia fueron incluidas:

- HardLimit (Perceptron)
- Logística sigmoide (Multilayer Perceptron)
- Tangente hiperbólica (Multilayer Perceptron)

- Lineal (MultilayerPerceptron, ADALINE, Red Competitiva)
- Lineal Positiva (Red Competitiva)

7.1.2 Arquitectura de la librería

Para cumplir con los objetivos anteriormente mencionados, se emplearon los principios del diseño orientado a objetos, de tal forma que se construyó una jerarquía de clases que ayudara a alcanzar las metas de diseño propuestas. Además, las clases se agruparon en paquetes, para tener una mayor claridad a la hora de la implementación.

Los diferentes modelos de red neuronal tienen en común el hecho de que son arreglos de neuronas, independientemente de su disposición o método de entrenamiento. Lo anterior sugiere que se puede establecer una jerarquía de clases, partiendo de una clase padre general, de donde se desprenderían una serie de hijos, cuya especialización aumente en la medida que se desplazan desde el padre hasta implementaciones concretas.

Recorriendo el árbol formado por dicha jerarquía de clases, se puede notar fácilmente que sería de gran utilidad definir dos ramas, las redes neuronales de entrenamiento supervisado y no supervisado. Estas ramas a su vez se pueden representar por abstracciones a partir de las cuales hereden el resto de modelos neuronales a incluir dentro de la librería.

A partir de la clase que representa el padre de las redes neuronales con entrenamiento supervisado, podemos establecer ciertas similitudes entre los modelos que corresponden a dicha categoría. Adaline y Perceptron comparten muchos detalles de funcionamiento y además son redes de una sola capa, por lo que tiene sentido agruparlos y hacer que hereden de una clase prototipo que encapsule los detalles comunes, simplificando el diseño de estos modelos neuronales. Por otro lado, el otro modelo neuronal a incluir dentro de JANE es Multilayer Perceptron, el cual a pesar de tener relación histórica y de fundamentos con los modelos anteriormente mencionados, es una red neuronal de múltiples

capas y de entrenamiento diferente en su implementación. Por lo anterior, es lógico hacer que herede de la clase padre de los modelos de entrenamiento supervisado.

Partiendo de la clase que representa los modelos neuronales de entrenamiento no supervisado, se encontraría el único modelo de estas características que se pretende incluir en JANE, la red de aprendizaje competitivo. Es importante destacar la importancia de esta clase, puesto que es la base de todos los modelos neuronales de aprendizaje no supervisado, y su inclusión dentro de la jerarquía se convierte en un punto de extensibilidad bastante importante, teniendo en cuenta que a futuro sería de gran interés para el autor implementar ciertos modelos de aprendizaje no supervisado. Cabe resaltar el hecho de que a través de toda la jerarquía de modelos neuronales, se prefiere el uso de clases abstractas debido a que estas permiten incorporar código en los cuerpos de los métodos que incluyen. Lo anterior ha sido de gran utilidad, puesto que ha hecho más fáciles las refactorizaciones que se han realizado sobre la arquitectura.

7.1.2.1 Neuron

La clase *Neuron* modela una neurona McCulloch – Pitts [5], y es el núcleo de la librería y todos los modelos neuronales incluidos dependen de esta clase para llevar a cabo el procesamiento de información. De acuerdo a lo anterior, la clase *Neuron* encapsula la lógica necesaria para el funcionamiento de una neurona artificial, incluyendo la suma ponderada de las entradas y el cómputo de la salida utilizando una función de transferencia, representada por una implementación de la interfaz *ITransferFunction*.

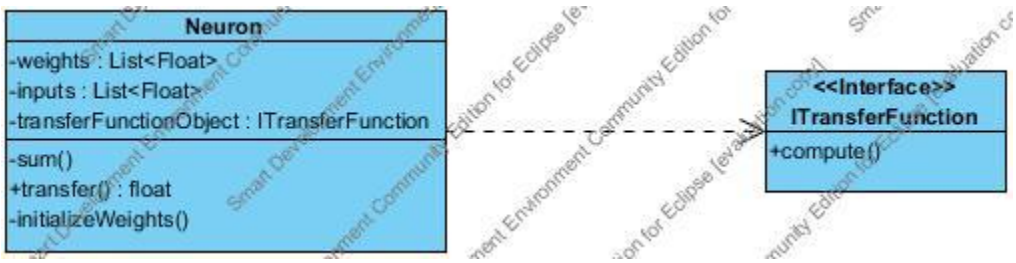

Figura 10. Clase Neuron.

7.1.2.2 Modelos neuronales

De manera formal, las clases que representan los modelos neuronales incluidos en JANE y sus respectivas responsabilidades son las siguientes:

7.1.2.2.1 AbstractNeuralNetwork

Esta clase abstracta representa el inicio de la jerarquía de clases que representan los diferentes modelos neuronales, y se incluye por razones de polimorfismo*.*

7.1.2.2.2 AbstractSupervisedNeuralNetwork

Esta clase abstracta representa el padre de las redes con entrenamiento supervisado y extiende *AbstractNeuralNetwork.* Esta clase declara una dependencia sobre una implementación de la interfaz *INetTrainer*, y proporciona la definición de algunos métodos de utilidad para las redes con entrenamiento supervisado.

7.1.2.2.3 AbstractUnsupervisedNeuralNetwork

Esta clase representa dentro de la jerarquía, el padre de las redes de entrenamiento supervisado. Se incluye por razones de polimorfismo.

7.1.2.2.4 SingleLayerFeedForwardNetwork

Debido a que tanto Adaline como Perceptron comparten la mayoría de detalles de funcionamiento, se puede pensar en refactorizar la lógica de estas redes e incluir

en el diseño una clase que contuviera dicha lógica. Esta clase abstracta la implementación del procesamiento de los datos (método *processData()*).

7.1.2.2.5 Adaline

Esta clase es una implementación concreta de la red ADALINE. Extiende la clase *SingleLayerFeedForwardNetwork* e incluye el manejo de la tasa de aprendizaje y de la regla de entrenamiento DeltaRule.

7.1.2.2.6 Perceptron

Esta clase representa la implementación concreta del Perceptron original, e incluye el uso de la regla del Perceptron para el entrenamiento.

7.1.2.2.7 MultilayerPerceptron

Esta es la implementación del Perceptron multicapa y extiende la clase *AbstractSupervisedNeuralNetwork.* Incluye la lógica necesaria para el procesamiento de la información a través de múltiples capas y soporte para el algoritmo Backpropagation.

7.1.2.2.8 CompetitiveNetwork

La implementación de la red competitiva básica. Esta clase representa un punto de extensión, puesto que muchos modelos neuronales se basan en el mismo principio de aprendizaje sin supervisión.

7.1.2.2.9 KohonenSelfOrganizingMap

Esta clase extiende de *CompetitiveNetwork* y representa los mapas auto-organizativos de Kohonen (SOM). Incluye el concepto de vecindario y redefine los métodos *init*() y *learn*(), de acuerdo al funcionamiento interno de este modelo neuronal.

Por otro lado, esta clase utiliza un arreglo bidimensional de neuronas en cada capa, por lo que declara una dependencia sobre la clase *LatticeLayer,* que se discutirá más adelante.

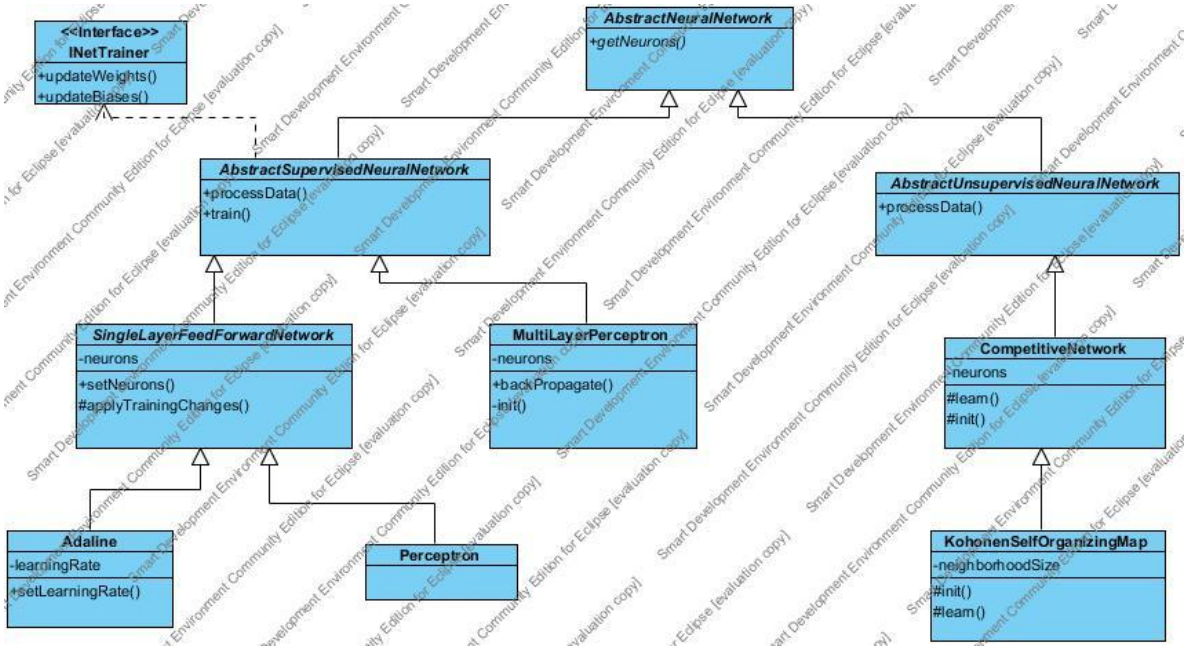

Figura 11. Jerarquía de clases – Modelos neuronales.

7.1.2.3 Reglas de aprendizaje

Un aspecto bien importante del diseño de una librería neuronal es el entrenamiento de las redes, especialmente en el subconjunto de las redes de aprendizaje supervisado; puesto que existen diversos algoritmos o reglas de entrenamiento que se aplican a uno o más modelos neuronales. Es lógico pensar que dichas reglas de entrenamiento sean representadas por abstracciones independientes de los modelos neuronales en sí, para garantizar un bajo acoplamiento y en el caso de incluirse nuevas reglas de aprendizaje, una rápida implementación con poco impacto sobre la arquitectura. De acuerdo a esto, se propone una jerarquía paralela de reglas de aprendizaje que nace a partir de un contrato general que especifica las responsabilidades de la regla de aprendizaje que se quiera incluir en la librería.

Debido a la importancia del concepto de regla de aprendizaje para las redes de aprendizaje supervisado, se propone que a partir de la clase que represente el padre de dichos modelos neuronales, las clases reciban en su constructor la instancia de la regla de aprendizaje apropiada. Para garantizar esto, en el constructor de la clase que representa el padre de las redes supervisadas se incluye un parámetro del tipo de la clase padre de las reglas de aprendizaje.

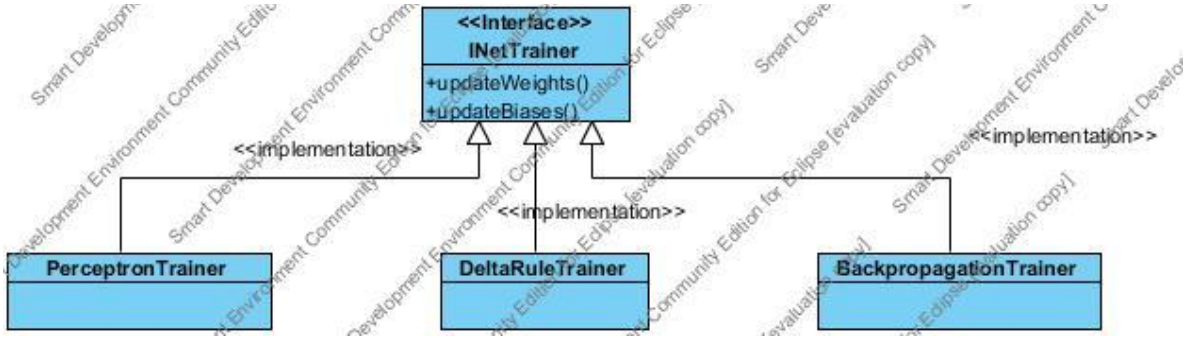

Figura 12. Jerarquía de clases – Reglas de aprendizaje.

Las clases que representan en JANE las diferentes reglas de aprendizaje para el entrenamiento supervisado son las siguientes:

7.1.2.3.1 INetTrainer

Esta es una interfaz que especifica el contrato de las reglas de aprendizaje con las clases que hacen uso de ellas. Toda regla de aprendizaje utilizada en JANE debe implementar esta interfaz. *INetTrainer* especifica 2 métodos a ser implementados: *updateWeights()* y *updateBiases()*, debido a que las reglas de aprendizaje tratan con estos 2 elementos de las neuronas, los pesos (weights) y bias (sesgo o umbral).

7.1.2.3.2 PerceptronTrainer

Esta clase implementa *INetTrainer* para la regla de aprendizaje del perceptron. Es usada por la clase *Perceptron*.

7.1.2.3.3 DeltaRuleTrainer

Esta clase representa la regla de aprendizaje Widrow-Hoff o Delta Rule. Es utilizada por la clase *Adaline* para su entrenamiento.

7.1.2.3.4 BackpropagationTrainer

Esta implementación de *INetTrainer* representa la regla de aprendizaje del algoritmo Backpropagation, utilizado por la clase *MultiLayerPerceptron* para su entrenamiento.

7.1.2.4 Capas de neuronas

Retomando el concepto primario de red, al ser un arreglo de neuronas conectadas entre sí, observamos una relación de composición. Ahora bien, debido a que la forma como se disponen las neuronas en cada modelo particular de red varía (número de capas, representación interna de la capa), la idea de introducir una abstracción que represente una capa de neuronas es atractiva, puesto que permitiría distribuir las responsabilidades relacionadas al procesamiento de la información y el ajuste de parámetros neuronales, y encapsular la complejidad, lejos de la implementación propia de los modelos neuronales. A su vez, las capas tendrán relación de composición con las clases que representan los modelos neuronales.

En síntesis, las clases que representan las capas de neuronas en JANE son las siguientes:

7.1.2.4.1 AbstractNeuronLayer

Esta clase abstracta es el padre de las clases que representan capas de neuronas de JANE. Establece una serie de operaciones sobre la capa, que incluyen el procesamiento de datos (método *process*()), agregar o reemplazar neuronas en la

capa y cambiar los pesos y biases de las neuronas contenidas a la vez. En el constructor de esta clase, se encapsula la lógica necesaria para instanciar el número de neuronas del tipo deseado (utilizando la enumeración *TransferFunctions*) e incluirlas en la capa. La estructura interna de la capa se deja bajo la responsabilidad de las clases hijas.

7.1.2.4.2 SimpleLayer

Representa una capa de neuronas sencilla, dispuestas en un arreglo unidimensional. Para efectos de flexibilidad se utiliza una instancia de List para el almacenamiento de dichas neuronas. Esta clase es utilizada por la mayoría de modelos neuronales de JANE.

7.1.2.4.3 LatticeLayer

Esta subclase representa un arreglo bidimensional de neuronas y su inclusión en la librería obedece a la clase *SelfOrganizingMap*, la cual utiliza dichos arreglos bidimensionales para la representación de la topología de los conjuntos de datos procesados.

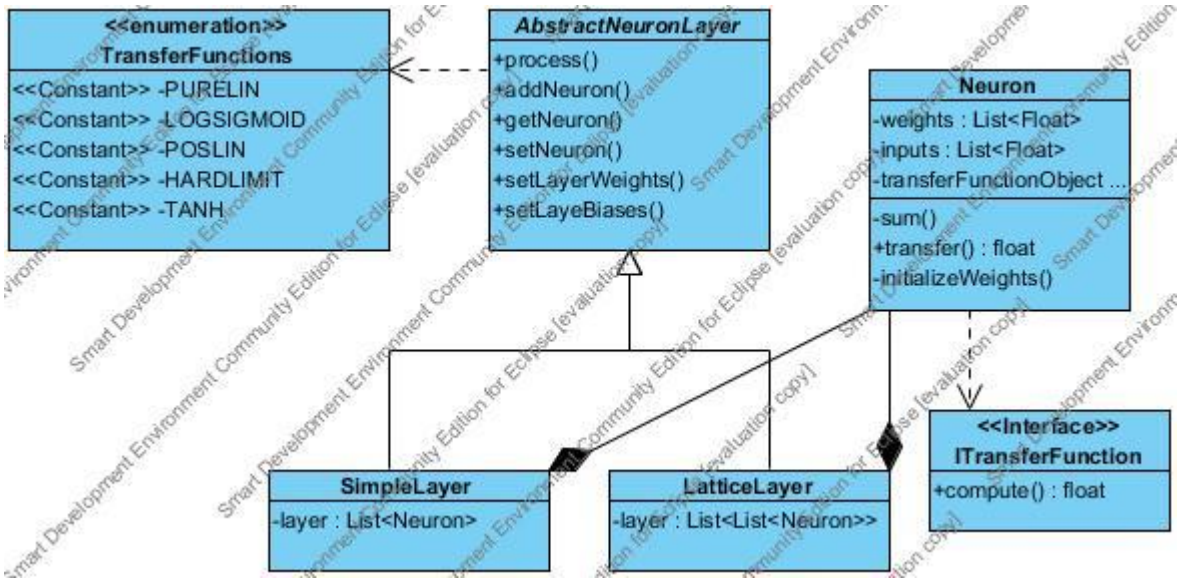

Figura 13. Jerarquía de clases – Capas de neuronas.

7.1.2.5 Funciones de transferencia

Por otro lado, cada modelo neuronal trabaja con ciertas funciones de transferencia, y dentro de una red neuronal se pueden encontrar capas de neuronas con distintas funciones de transferencia. En el marco de esta situación, el polimorfismo se convierte en una gran herramienta para lograr que la implementación de la clase *Neuron* trabaje con varias funciones de transferencia, sin requerir cambios en su código. Para lo anterior se definió una estructura de clases a partir de una interfaz, y se declaró una dependencia sobre esta interfaz en la clase que representa a las neuronas, por lo cual se garantiza un punto importante de extensibilidad, y la posibilidad de incluir nuevas funciones de transferencia en cualquier momento. Los artefactos involucrados se describen a continuación:

7.1.2.5.1 ITransferFunction

Esta interfaz declara el contrato de las funciones de transferencia en JANE. De acuerdo al marco teórico, la responsabilidad de la función de transferencia de la neurona es tomar la suma ponderada de las entradas y el bias y computar la salida a partir de ella. Esta interfaz se incluye para garantizar un bajo acoplamiento de la clase *Neuron* con la implementación concreta de las funciones de transferencia, y todas las funciones de transferencia en JANE deben implementar esta Interfaz.

ITransferFunction declara un solo método: *compute*(), el cual recibe como parámetro la suma ponderada de las entradas a la neurona más el bias, y devuelve el resultado de acuerdo a la implementación de la respectiva función de transferencia. Dicha implementación, como es obvio, se deja a cargo de las clases que implementan esta interfaz.

7.1.2.5.2 PureLinFunction

Esta clase es una implementación de *ITransferFunction* para representar una función lineal de transferencia (y = x), y es utilizada por varios modelos neuronales en JANE.

7.1.2.5.3 PosLinFunction

Esta clase implementa *ITransferFunction* y representa una función de transferencia con comportamiento lineal para entradas mayores a cero, y que devuelve cero para entradas menores o iguales a cero. Es utilizada en las redes competitivas (*CompetitiveNetwork* y *SOM*).

7.1.2.5.4 HardLimitFunction

Esta implementación de *ITransferFunction* representa el limitador fuerte, la función de transferencia utilizada en la clase *Perceptron*.

7.1.2.5.5 LogSigmoidFunction

Esta implementación de *ITransferFunction* representa la función logística sigmoide, utilizada en *MultiLayerPerceptron*.

7.1.2.5.6 TanhFunction

Esta implementación de *ITransferFunction* representa la función sigmoide tangente hiperbólica, de uso común en *MultiLayerPerceptron*.

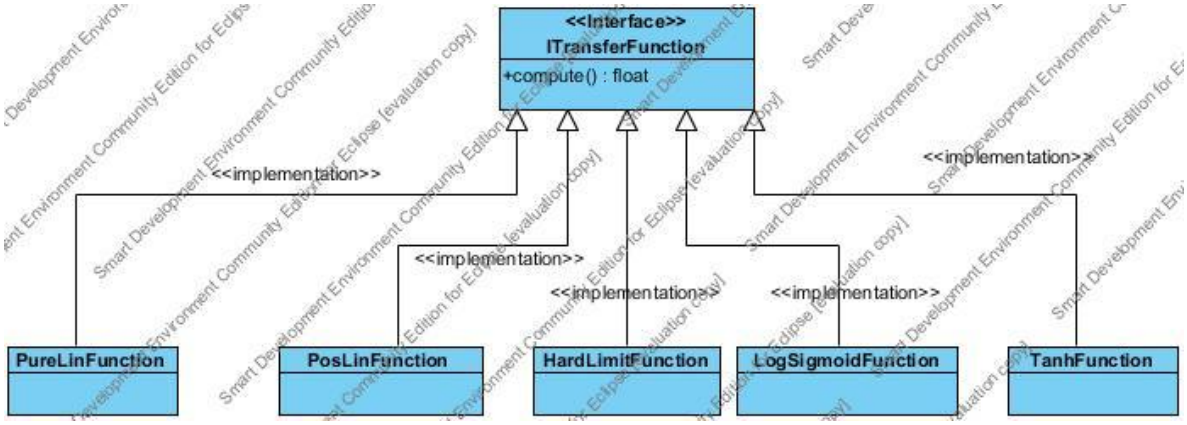

Figura 14. Jerarquía de clases. Funciones de transferencia.

De esta forma culminamos un recorrido por el diseño de JANE y a continuación se exploran los diferentes paquetes que componen la librería y lo relacionado a la implementación de las clases que los integran, y se revisan más a fondo las responsabilidades asignadas.

7.1.3 Implementación de las clases de JANE

Las clases propuestas en el diseño de JANE se implementaron de acuerdo a la estructura de paquetes descrita a continuación.

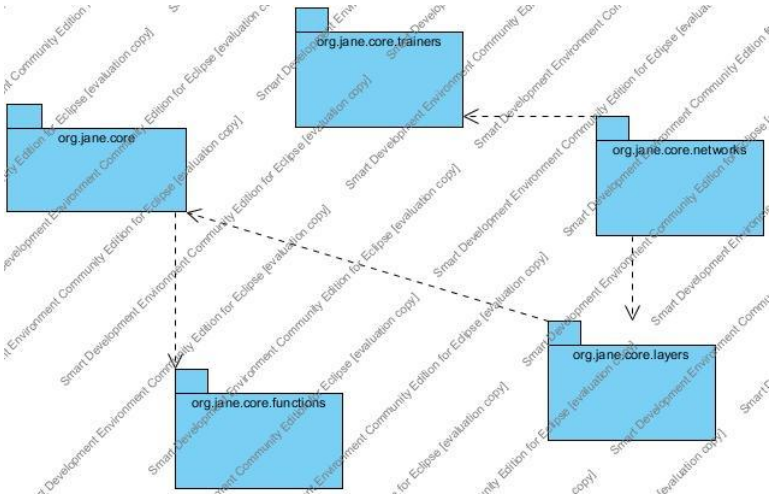

Figura 15. Paquetes de JANE

7.1.3.1 org.jane.core

En este paquete encontramos la clase *Neuron* y la enumeración *TransferFunctions.* La clase *Neuron* representa a una neurona artificial. Esta es la clase fundamental dentro de la librería, y tiene relaciones de composición con muchas clases de la librería. Su responsabilidad es el modelado del comportamiento de una neurona artificial. Está relacionada íntimamente con la interfaz *org.jane.core.functions.ITransferFunction*, puesto que los objetos que implementan dicha interfaz proveen la lógica relacionada a la función de transferencia.

La enumeración *TransferFunctions* provee soporte para el uso de diferentes tipos de neuronas (neuronas con diferente función de transferencia) dentro de la misma red.

7.1.3.2 org.jane.core.functions

En este paquete se encuentran las implementaciones de las funciones de transferencia, concepto de importancia fundamental en la teoría de redes neuronales. Se encuentra la interfaz *ITransferFunction,* y cualquier clase que se quiera utilizar como función de transferencia debe implementar esta interfaz. Utilizando el polimorfismo de esta forma podemos incorporar nuevas funciones de transferencia de forma fácil y sin necesidad de cambiar el código existente. La interfaz *ITransferFunction* solo declara un método: *public float compute(float input).* Este método recibe la suma de las entradas a la neurona, y devuelve el valor que representará la respuesta de la neurona.

El resto de clases de este paquete son *HardLimitFunction, LogSigmoidFunction, PosLinFunction* y *PureLinFunction.* Dichas clases en su orden implementan las funciones HardLimit(Limitador Fuerte), Logística Sigmoidea, Lineal Positiva y Lineal.

7.1.3.3 org.jane.core.layers

En JANE, las neuronas se relacionan con las redes a través de una abstracción que representa las capas que componen a una red neuronal. Esta abstracción comprende las clases que extienden a *AbstractNeuronLayer*, y esta clase y sus hijas están ubicadas en el paquete *org.jane.core.layers*.

La clase *AbstractNeuronLayer* define métodos para el procesamiento de entradas, y para acceder y modificar los parámetros internos de cada neurona (para efectos del entrenamiento); además define una dependencia sobre la enumeración *org.jane.core.TransferFunctions*, necesaria para determinar el tipo de neuronas que compondrán la capa. Las hijas de *AbstractNeuronLayer* son *LatticeLayer* y *SimpleLayer,* la primera contiene una matriz (Lista de Listas) de *Neuron*, mientras que la segunda contiene una Lista sencilla de Neuron. *SimpleLayer* es el tipo de capa que utilizan la mayoría de arquitecturas neurales implementadas en JANE.

7.1.3.4 org.jane.core.networks

En este paquete se encuentran los diferentes modelos de red neuronal implementados en JANE. Dentro de la jerarquía de clases, encontramos la clase abstracta *AbstractNeuralNetwork*, de donde heredan todos los modelos implementados. Esta clase existe para efectos de aprovechar las ventajas del polimorfismo. De esta clase heredan directamente las clases *AbstractSupervisedNeuralNetwork* y *AbstractUnsupervisedNeuralNetwork*, que representan las redes neuronales de aprendizaje supervisado y no supervisado respectivamente. *AbstractSupervisedNeuralNetwork* declara además de métodos para el proceso de información, métodos para el entrenamiento de la red, mientras que *AbstractUnsupervisedNeuralNetwork* solo declara métodos para procesamiento de la información y métodos de utilidad específicos de redes no supervisadas.

Más abajo en la jerarquía de clases, encontramos las clases *SingleLayerFeedForwardNetwork* y *MultilayerPerceptron*, que heredan de

AbstractSupervisedNeuralNetwork. *SingleLayerFeedForwardNetwork* es una clase abstracta que sirve de base para la implementación de las clases *Perceptron* y *Adaline,* que representan los modelos de red neuronal del mismo nombre; mientras que *MultilayerPerceptron* representa la red del mismo nombre. Estas clases (*MultilayerPerceptron, Perceptron* y *Adaline*) ya son implementaciones concretas, listas para ser usadas.

Por el lado de *AbstractUnsupervisedNeuralNetwork* encontramos su descendiente *CompetitiveNetwork,* la cual es la clase que implementa la red neuronal de aprendizaje competitivo y esta lista para ser usada. Adicionalmente encontramos una implementación experimental de la red neuronal KohonenSOM, en la clase *KohonenSelfOrganizingMap,* la cual extiende de *CompetitiveNetwork. KohonenSelfOrganizingMap* es la única clase que hasta el momento incorpora una capa tipo *org.jane.core.layers.LatticeLayer.*

7.1.3.5 org.jane.core.trainers

Las redes neuronales necesitan de entrenamiento para poder funcionar adecuadamente. Cada tipo de red neuronal tiene un algoritmo de entrenamiento específico, y en este paquete podemos encontrar los respectivos algoritmos para cada modelo de red que JANE implementa. Para efectos de polimorfismo tenemos la clase abstracta *AbstractNetTrainer* que es la clase padre de todas las reglas de entrenamiento implementadas en JANE. La clase *org.jane.core.networks. AbstractSupervisedNeuralNetwork* recibe en su constructor una instancia de *AbstractNetTrainer,* de tal forma que todas las redes que implementan dicha clase abstracta necesitan una referencia a un descendiente de *AbstractNetTrainer.*

Los descendientes de *AbstractNetTrainer son PerceptronTrainer, DeltaRuleTrainer* y *BackpropagationTrainer*; y representan las reglas de entrenamiento del perceptron, adaline y perceptron multicapa, respectivamente.

7.1.3.6 org.jane.utils

En este paquete encontramos clases de utilidad general como *VectorUtils*, que contiene métodos para el tratamiento de vectores (normalización). Además encontramos abstracciones para un concepto de vital importancia en JANE: Los conjuntos de datos (Dataset), y están representadas por las clases *AbstractDataset, SupervisedDataset* y *UnsupervisedDataset*. Las redes neuronales toman datasets como entrada y a partir de ellos generan una salida.

AbstractDataset es la clase abstracta de donde extienden *SupervisedDataset* y *UnsupervisedDataset,* y existe por consideraciones de polimorfismo. *SupervisedDataset* es la implementación concreta para representar los datos de entrada de las redes con entrenamiento supervisado y *UnsupervisedDataset* es su contraparte para redes con aprendizaje sin supervisión.

7.2 NEURALSTUDIO

NeuralStudio es una aplicación construida para la enseñanza sobre redes neuronales, que utiliza JANE para su funcionamiento. La idea detrás de la aplicación era el convertirse en una herramienta que pudiese proveer al interesado en el campo de las redes neuronales, un primer contacto con los conceptos detrás de ellas y sus aplicaciones en el mundo real. NeuralStudio al igual que JANE, es una aplicación construida en Java, de tal forma que para su construcción se pudieron aplicar muchas de las herramientas que nos brinda la programación orientada a objetos.

La aplicación se planteó desde un principio como una aplicación de escritorio MDI (Multi Document Interface), para ser desarrollada utilizando Swing. Debido a la naturaleza intensiva en recursos computacionales del entrenamiento de redes neuronales, se identificó la necesidad de desarrollar una interfaz de usuario multihilos, donde los procesos de entrenamiento fuesen delegados a hilos diferentes al principal (donde se ejecuta el procesamiento de eventos), de tal forma que la interfaz de usuario no se “trabe”. Cabe resaltar que durante la etapa

de diseño de la interfaz gráfica se puso especial cuidado en proveer la mejor experiencia posible para el usuario.

7.2.1 Aplicaciones escogidas

En el proceso de diseño de NeuralStudio se presentó la necesidad de escoger que aplicaciones estarían presentes en NeuralStudio. Se partió del hecho de que la principal aplicación de las redes neuronales es el reconocimiento de patrones en conjuntos de datos, por lo que inmediatamente se consideró que NeuralStudio debía contener una aplicación para el reconocimiento de patrones.

Por otro lado, se identificó una aplicación bastante interesante y con gran potencial didáctico (e incluso ingenieril), la aproximación de funciones. Las redes neuronales son aproximadores universales (con precisión arbitraria) del conjunto de funciones integrables, pudiendo “imitar” el comportamiento de una función en un intervalo determinado. Esta aplicación permitiría observar de manera gráfica el comportamiento de la respuesta de la red durante el entrenamiento, por lo que se tomó la decisión de incluirla dentro de NeuralStudio.

Otro factor que se tomó en cuenta fue el hecho de que cada modelo de red neuronal tiene ciertas aplicaciones donde se desempeña mejor. Los modelos soportados por JANE tienen en común su fortaleza en el reconocimiento de patrones, y esto le dio más solidez a la decisión de incluir esta aplicación en NeuralStudio; además el reconocimiento de patrones puede ser llevado a cabo por redes neuronales tanto de entrenamiento supervisado como sin supervisión (JANE soporta modelos de ambos tipos de red).

7.2.2 Aspectos conceptuales y de funcionamiento

Como ya se comentó, NeuralStudio se construyó en Java, como una aplicación de escritorio con interfaz Swing, además se escogió una estructura MDI para permitir trabajar en varios proyectos al tiempo y se utilizaron hilos para el manejo de las tareas costosas en CPU.

NeuralStudio incorpora varias librerías de terceros, específicamente JMathPlot[1] para la visualización de datos, JEP[2] para el reconocimiento e interpretación de expresiones matemáticas y NetBeans Visual API[3] para la representación gráfica de las redes neuronales.

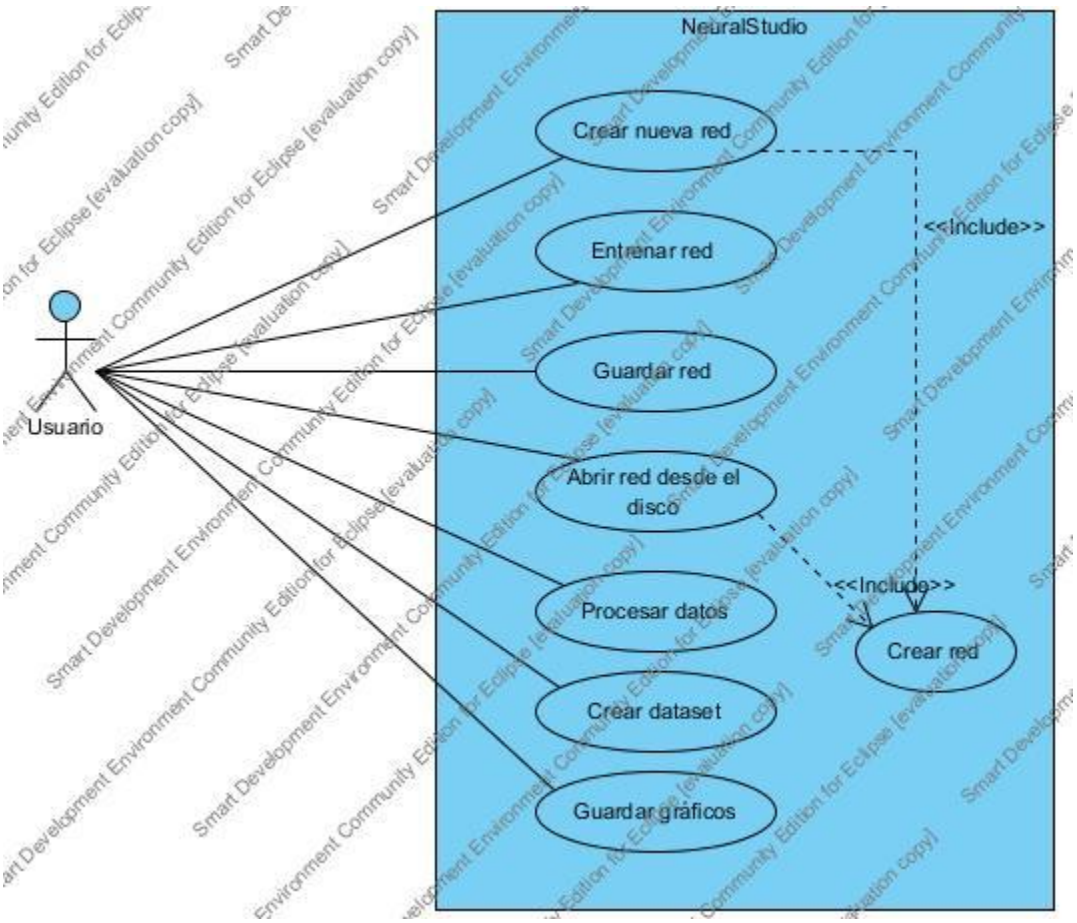

Figura 16. Casos de uso de NeuralStudio.

Los casos de uso identificados en el análisis hecho para NeuralStudio se detallan a continuación:

7.2.2.1 Casos de uso de NeuralStudio

[1] http://code.google.com/p/jmathplot/
[2] http://sourceforge.net/projects/jep/files/
[3] http://platform.netbeans.org

7.2.2.1.1 Crear nueva red (UC001)

Nombre: Crear nueva red (UC001)

Descripción: En este caso de uso el sistema permite a un usuario crear una nueva red neuronal con una arquitectura determinada por él.

Actor Principal:

- Usuario

Actores Secundarios:

Precondiciones:

- El usuario debe haber iniciado el sistema

Flujo De Eventos

Pasos	Usuario	Sistema
1		Muestra un menú que contiene la opción para crear una nueva red neuronal.
2	Selecciona la opción crear nueva red.	
3		Muestra una nueva ventana con controles para configurar la arquitectura de la nueva red neuronal.
4	Configura la arquitectura y presiona el botón correspondiente.	
5		El sistema crea una nueva instancia de una red neuronal con las características seleccionadas por el usuario, y activa las aplicaciones adecuadas. En caso de existir una red creada con anterioridad, el sistema muestra un mensaje de advertencia.

Flujo Alternativo:

*El sistema en cualquier momento falla: Se debe reiniciar manualmente el sistema.

5a. Si el usuario no desea reemplazar la información de la red neuronal existente, el sistema cancela la creación y regresa al paso 3.

Post Condición:

- El sistema crea una nueva red neuronal lista para ser usada.

7.2.2.1.2 Abrir red desde disco (UC002)

Nombre: Abrir red desde disco (UC002)

Descripción: En este caso de uso el sistema permite a un usuario crear una nueva red neuronal a partir de un archivo XML guardado en disco.

Actor Principal:

- Usuario

Actores Secundarios:

Precondiciones:

- El usuario debe haber iniciado el sistema

Flujo De Eventos

Pasos	Usuario	Sistema
1		Muestra un menú que contiene la opción para crear una nueva red neuronal.
2	Selecciona la opción abrir red desde disco.	
3		Muestra un cuadro de diálogo para escoger el archivo que contiene la red que el usuario desea abrir.
4	Escoge el archivo correspondiente.	
5		El sistema crea una nueva instancia de una red neuronal con las características almacenadas en el archivo XML.

Flujo Alternativo:

*El sistema en cualquier momento falla: Se debe reiniciar manualmente el sistema.

5a. Si el usuario no desea reemplazar la información de la red neuronal existente, el sistema cancela la creación y regresa al paso 3.

5b. Si ocurre un error al abrir e interpretar el archivo el sistema alerta al usuario y regresa al paso 1.

Post Condición:

- El sistema crea una nueva red neuronal lista para ser usada.

7.2.2.1.3 Crear red (UC003)

Nombre: Crear red (UC003)

Descripción: En este caso de uso el sistema ofrece la funcionalidad de crear una nueva red neuronal.

Actor Principal:

- Sistema

Actores Secundarios:

Precondiciones:

- El usuario debe haber iniciado el sistema

Flujo De Eventos

Pasos	Usuario	Sistema
1	Presenta los parámetros de la red a crear y solicita su instanciación.	
2		Crea una nueva instancia de la red neuronal deseada.

Flujo Alternativo:

*El sistema en cualquier momento falla: Se debe reiniciar manualmente el sistema.

Post Condición:

- El sistema crea una nueva red neuronal.

7.2.2.1.4 Entrenar red (UC004)

Nombre: Entrenar red (UC004)

Descripción: En este caso de uso el sistema permite a un usuario entrenar una red neuronal previamente creada.

Actor Principal:

- Usuario

Actores Secundarios:

Precondiciones:

- El usuario debe haber iniciado el sistema.
- El usuario debe haber creado o abierto una red neuronal.

Flujo De Eventos

Pasos	Usuario	Sistema
1		Muestra las aplicaciones de aproximación de funciones y clasificación de datos.
2	Selecciona la aplicación, el set de datos correspondiente, y hace click en el botón de entrenamiento.	
3		Realiza el entrenamiento de la red seleccionada de acuerdo al modelo escogido.
4		Muestra gráficamente el conjunto de datos de entrenamiento, la respuesta de la red y la evolución del error.

Flujo Alternativo:

*El sistema en cualquier momento falla: Se debe reiniciar manualmente el sistema.

Post Condición:

- El sistema crea una nueva red neuronal lista para ser usada.

7.2.2.1.5 Guardar red (UC005)

Nombre: Guardar red (UC005)

Descripción: En este caso de uso el sistema permite a un usuario guardar en un archivo XML una red creada.

Actor Principal:

- Usuario

Actores Secundarios:

Precondiciones:

- El usuario debe haber iniciado el sistema
- El usuario debe haber creado e inicializado una red neuronal.

Flujo De Eventos

Pasos	Usuario	Sistema
1		Muestra un menú que contiene la opción para guardar una nueva red neuronal.
2	Selecciona la opción guardar red.	
3		Muestra un cuadro de diálogo para escoger el nombre de archivo y la ubicación para el documento XML.
4	Selecciona el nombre de archivo y la ubicación correspondiente.	
5		El sistema serializa la red neuronal al archivo XML escogido por el usuario.

Flujo Alternativo:

*El sistema en cualquier momento falla: Se debe reiniciar manualmente el sistema.

5a. Si ocurre un error el sistema alerta al usuario y regresa al paso 1.

Post Condición:

- El sistema crea un archivo XML con la representación de la red guardada

7.2.2.1.6 Procesar datos (UC006)

Nombre: Procesar datos (UC006)

Descripción: En este caso de uso el sistema permite a un usuario procesar la información a través de una red creada.

Actor Principal:

- Usuario

Actores Secundarios:

Precondiciones:

- El usuario debe haber iniciado el sistema
- El usuario debe haber creado una red neuronal.

Flujo De Eventos

Pasos	Usuario	Sistema
1		Muestra controles para introducir una entrada a la red.
2	Ingresa los datos a procesar y hace click en el botón correspondiente.	
3		Muestra la salida de la red en un cuadro de texto.

Flujo Alternativo:

*El sistema en cualquier momento falla: Se debe reiniciar manualmente el sistema.

2a. Si ocurre un error al ingresar los datos el sistema alerta al usuario y regresa al paso 1.

Post Condición:

- El sistema muestra el resultado del procesamiento de la entrada por parte de la red neuronal.

7.2.2.1.7 Crear dataset (UC007)

Nombre: Crear dataset (UC007)

Descripción: En este caso de uso el sistema permite a un usuario crear un nuevo dataset.

Actor Principal:

- Usuario

Actores Secundarios:

Precondiciones:

- El usuario debe haber iniciado el sistema.

Flujo De Eventos

Pasos	Usuario	Sistema
1		Muestra un menú que contiene la opción crear un nuevo dataset.
2	Selecciona la opción crear nuevo dataset.	
3		Muestra una ventana con los controles necesarios para crear un nuevo dataset.
4	El usuario introduce el nombre y los datos del nuevo dataset.	
5		El sistema instancia un objeto dataset a partir de los datos introducidos.

Flujo Alternativo:

*El sistema en cualquier momento falla: Se debe reiniciar manualmente el sistema.

4a. Si ocurre un error el sistema alerta al usuario y regresa al paso 1.

Post Condición:

- El sistema crea un objeto que contiene los datos de entrenamiento.

7.2.2.1.8 Guardar gráficas (UC008)

Nombre: Guardar gráficas (UC008)

Descripción: En este caso de uso el sistema permite a un usuario guardar en disco las gráficas de respuesta de la red actual.

Actor Principal:

- Usuario

Actores Secundarios:

Precondiciones:

- El usuario debe haber iniciado el sistema
- El usuario debe haber creado e inicializado una red neuronal.

Flujo De Eventos

Pasos	Usuario	Sistema
1		Muestra la opción para guardar las gráficas de respuesta.
2	Selecciona la opción guardar gráfica.	
3		Muestra un cuadro de diálogo para escoger el nombre de archivo y la ubicación para la gráfica a guardar.
4	Selecciona el nombre de archivo y la ubicación correspondiente.	
5		El sistema graba las correspondientes gráficas.

Flujo Alternativo:

*El sistema en cualquier momento falla: Se debe reiniciar manualmente el sistema.

Post Condición:

- El sistema crea un archivo gráfico con la respuesta de la red.

7.2.2.2 TrainingDataManager

Como se mencionó anteriormente en el análisis de la estructura de JANE, el concepto de dataset (conjunto de datos) es vital dentro del ámbito de este trabajo. Al ser una una herramienta MDI, NeuralStudio permite abrir varias ventanas de trabajo (instancias de *JInternalFrame*), por lo que se consideró de utilidad contar

con un punto central de almacenamiento para los conjuntos de datos de entrenamiento (Instancias de alguno de los descendientes de *org.jane.utils.AbstractDataset*), y poder así tener disponibles los datasets en todas las ventanas internas de trabajo.

De acuerdo a esto, NeuralStudio incorpora una clase para el manejo de los datasets, llamada *TrainingDataManager*. *TrainingDataManager* es un *Singleton[21]* que contiene un *HashMap* que a su vez contiene los datasets disponibles para las diferentes aplicaciones. La decisión de diseño de utilizar este patrón se debió a la necesidad de contar con un repositorio de datasets único y de mantener la integridad del mismo, garantizando un acceso concurrente adecuado.

Además, se destinó un *JInternalFrame* específicamente para agregar datasets. *TrainingDataManager* expone métodos para obtener modelos de lista (*DefaultListModel*), de combobox (*DefaultComboBoxModel*) y de tablas (Subclase de *DefaultTableModel*), de forma que los componentes dentro de los diferentes *JInternalFrame* puedan tener acceso a la información almacenada en el *Singleton*.

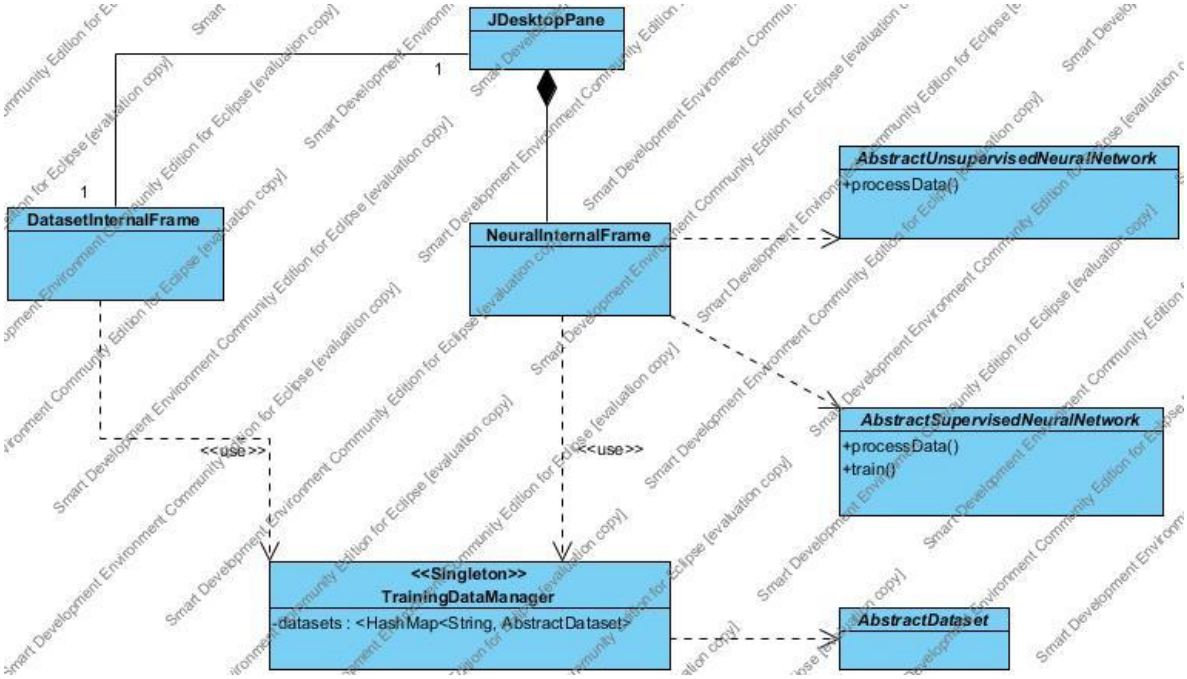

Figura 17. Diagrama de clases de NeuralStudio.

7.2.2.3 Interfaz de usuario

Como se ha mencionado en varias oportunidades, NeuralStudio cuenta con una interfaz gráfica MDI, para permitirle al usuario tener varias redes abiertas al tiempo dándole la posibilidad de comparar diversas arquitecturas. Por esto, el trabajo se realiza en ventanas internas representadas por la clase *NeuralInternalFrame*, descendientes de *JInternalFrame* que contienen una disposición de controles Swing que permiten interactuar con las aplicaciones escogidas (Reconocimiento de Patrones y Aproximación de funciones. Cada *NeuralInternalFrame* contiene un Tabbed Pane con tres pestañas: La primera permite crear una nueva red neuronal, especificando el modelo y la estructura interna; la segunda contiene la aplicación de aproximación de funciones, y la tercera la aplicación de reconocimiento de patrones

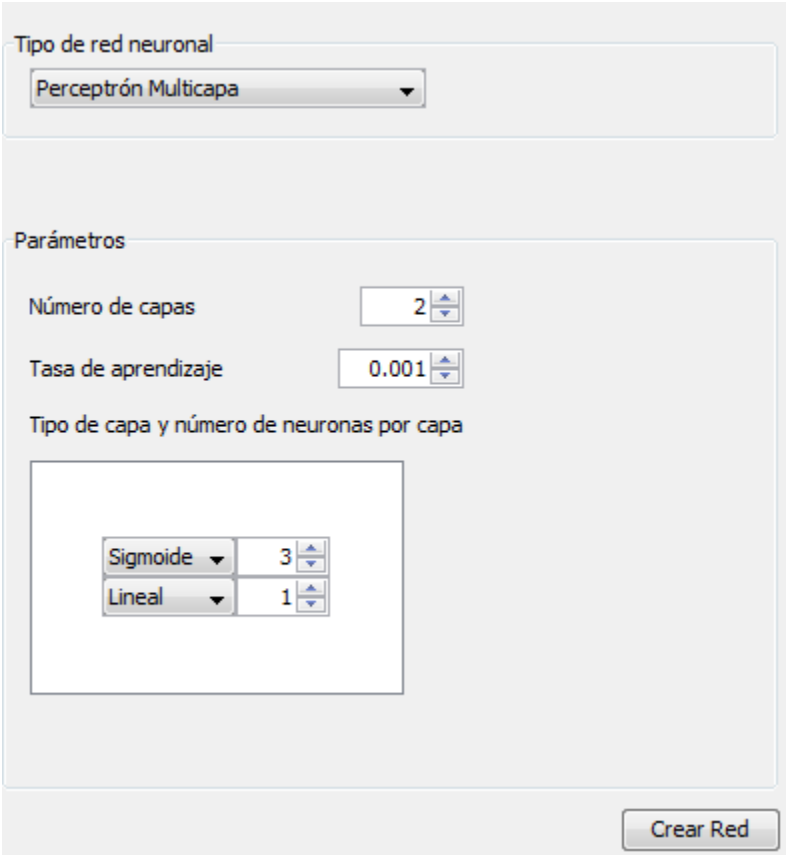

Figura 18. Creación de una red neuronal en NeuralStudio

Para facilitar la comprensión de la estructura interna de las redes, se agregó un panel que muestra la representación gráfica de cada modelo de red, y dicho panel muestra la estructura interna del modelo de red creado, de acuerdo al número de capas y número de neuronas por capa seleccionados. Para esto se utilizó el Visual API de Netbeans, y se desarrolló por inducción matemática un pequeño modelo para que las neuronas aparecieran centradas en el lienzo. Dicho modelo tiene la siguiente forma:

$$C_x = a - ((N-1)R + \frac{(N-1)n}{2})$$

Donde C_x es la coordenada x de la primera neurona de la capa, a la coordenada x de una línea imaginaria trazada en el medio del lienzo, N el número de neuronas

de la capa, R el radio del circulo que representa a cada neurona y n la distancia entre cada neurona.

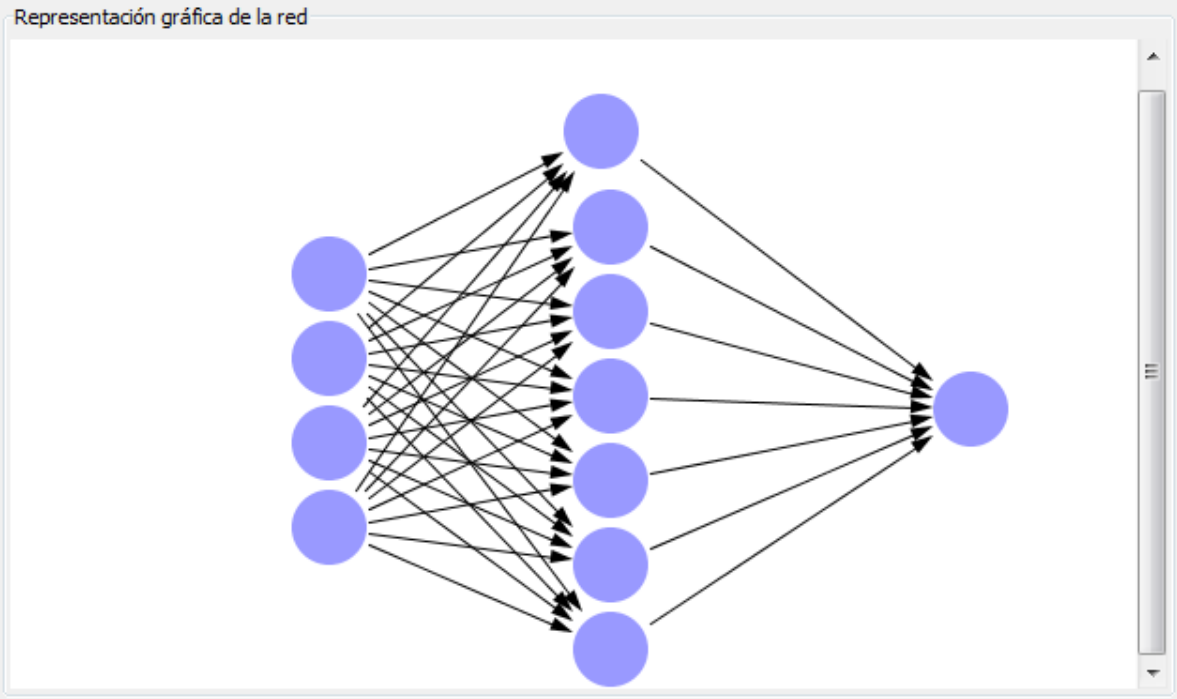

Figura 19. Representación gráfica de un perceptron multicapa en NeuralStudio.

La pestaña perteneciente a la aplicación de aproximación de funciones contiene controles para introducir una función matemática de una variable (x), y los límites de un intervalo en que dicha función será evaluada. Adicionalmente permite seleccionar un conjunto de datos que represente una función para la que no se tenga expresión matemática. Adicionalmente permite establecer el número de iteraciones de entrenamiento, junto con paneles de información y un panel para graficar la función y el error de la red.

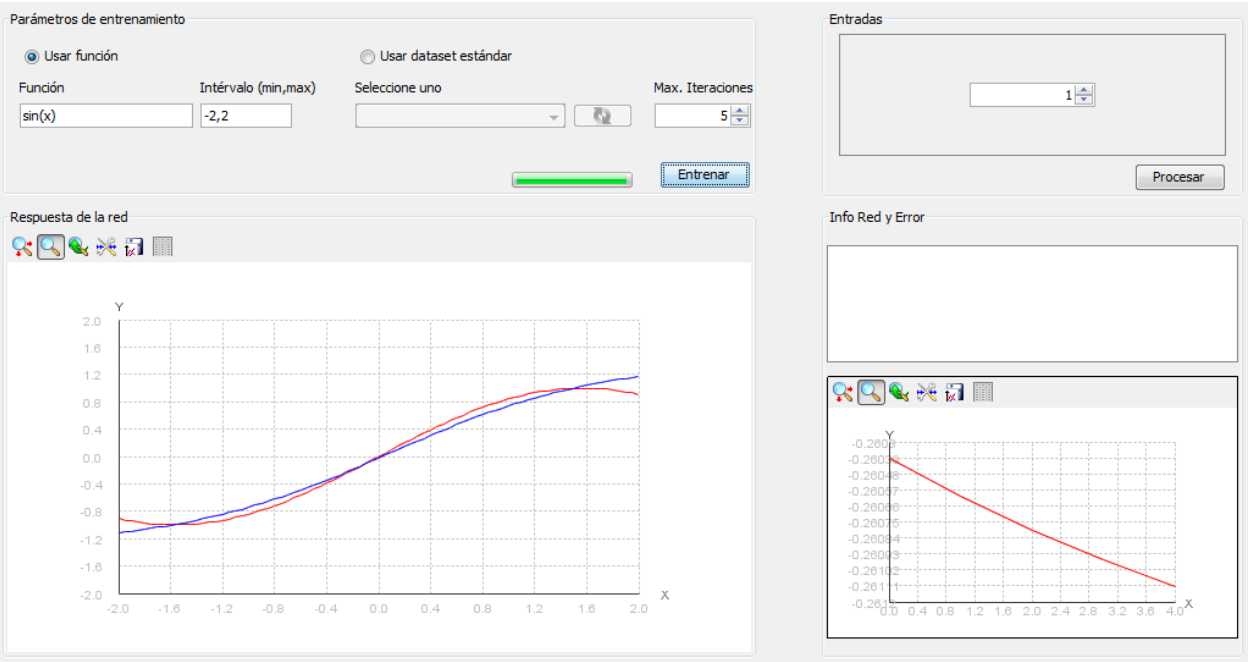

Figura 20. Aplicación de aproximación de funciones

Por último, la pestaña perteneciente a la aplicación de reconocimiento de patrones presenta una disposición similar a la anterior, con todo lo necesario para observar el comportamiento de la salida de la red. Cabe destacar, que el número de iteraciones de entrenamiento es ajustable, de tal forma que se puede llegar a niveles de error bajos rápidamente o ver el comportamiento de la red paso a paso.

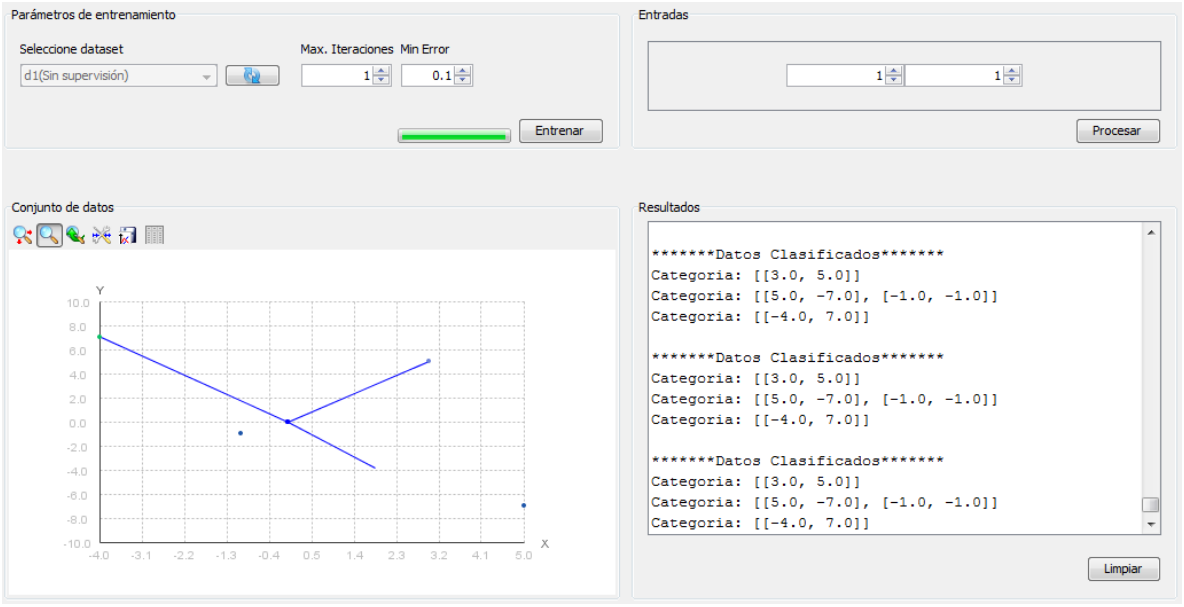

Figura 21. Aplicación de reconocimiento de patrones

Como se mencionó anteriormente, el manejo de los datasets se realiza de manera centralizada, a través de un *JInternalFrame* específico. Se trata de *DatasetInternalFrame*, el cual permite agregar nuevos datasets y modificar los existentes a través de su interfaz gráfica. Adicionalmente, *DatasetInternalFrame* nos permite exportar datasets a XML y posteriormente volverlos a cargar.

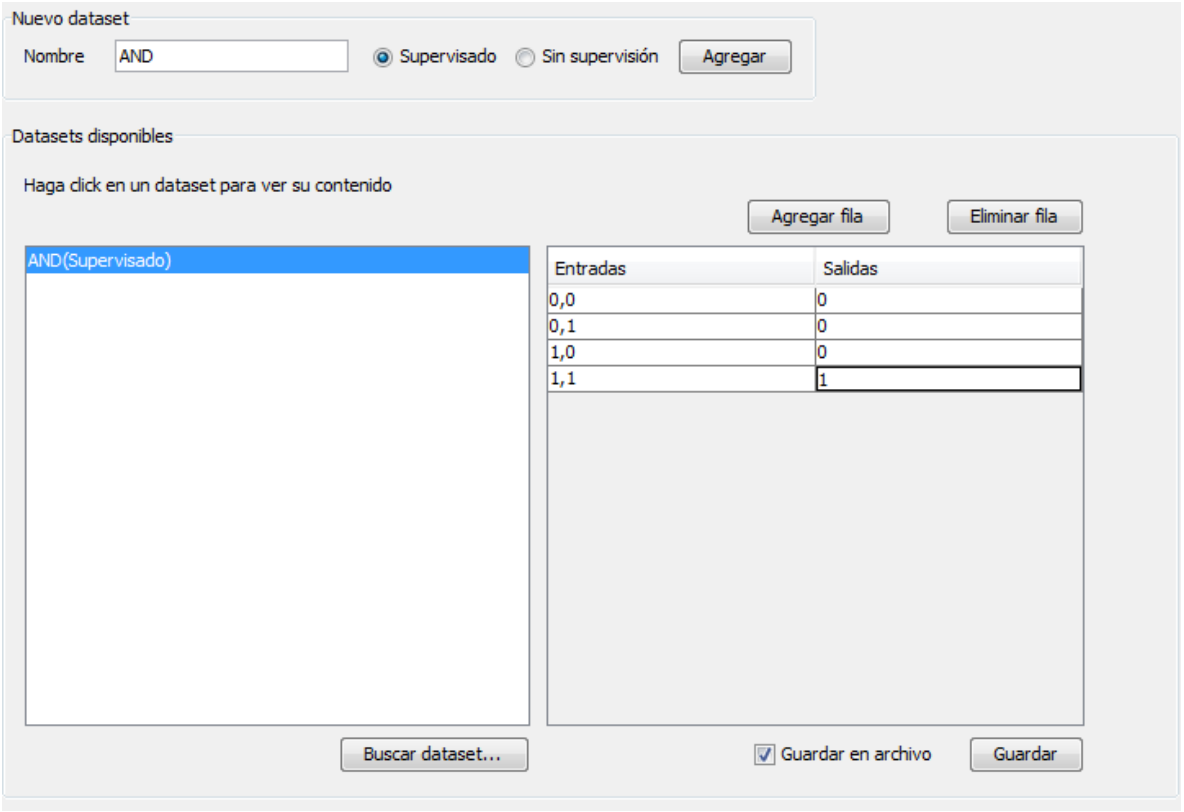

Figura 22. DataSetInternalFrame

7.2.2.4 Uso de JANE

NeuralStudio es una aplicación de 2 capas: La capa de presentación y la capa lógica, esta última representada por JANE, la cual proporciona el acceso a los diferentes modelos de redes neuronales. Como se manifestó anteriormente, la aplicación interactúa con JANE fuera del hilo de procesamiento de eventos (Event Dispatch Thread), para garantizar que la aplicación no quede colgada esperando la respuesta de los objetos del modelo. De acuerdo a lo anterior, las clases *AproximationTrainerThread* y *ClassificationTrainerThread* son las que invocan las rutinas de entrenamiento de las redes neuronales específicamente.

Los responsables de la instanciación desde *NeuralInternalFrame* de la serialización y deserialización hacia y desde XML también interactúan con JANE,

utilizando los constructores adecuados en cada caso. Es particularmente interesante el caso de XML, por lo que lo trataremos con más detalle a continuación.

7.2.2.5 eXtensible Markup Language - XML

De acuerdo a las hipótesis de investigación, se creía que XML se prestaba para la descripción de la estructura interna de una red neuronal, y dada su amplia prevalencia como lenguaje de descripción y compartición de datos, sería una elección provechosa para darle capacidades de serialización a NeuralStudio.

Se procedió entonces a analizar cómo se podría “traducir” una red neuronal a un documento XML, y se identificó un hecho muy interesante: Una red neuronal se puede representar como grafo dirigido con aristas ponderadas, además es bien sabido que los arboles son estructuras de naturaleza jerárquica y recursiva, donde hay nodos padres y nodos hijos. En este punto se detecta la semejanza con la naturaleza jerárquica de un documento XML, puesto que si se establecen elementos para representar la red, capas y neuronas, respectivamente, sería posible expresar de manera muy natural e intuitiva la estructura de la red neuronal en XML, aprovechando de esa manera todo el potencial de las herramientas que rodean a este popular lenguaje de intercambio de datos. Se definió un esquema para el documento XML, que reflejara de manera intuitiva la estructura interna de las redes neuronales sujetas a serialización, teniendo en cuenta la jerarquía de clases definida para JANE.

Por otro lado, los datasets en JANE y NeuralStudio contienen estructuras de datos basadas en listas, por lo que su representación en XML también probó ser bastante sencilla. Se creó un SAX handler para esta tarea, llamado DatasetSaxHandler, el cual funciona de manera análoga a su contraparte *NeuralSaxHandler*, dándole capacidades a NeuralStudio para guardar datasets en disco.

7.2.2.5.1 Estructura de los documentos utilizados

De acuerdo a lo identificado anteriormente, para la representación de las redes se concibió una estructura integrada por los siguientes elementos:

- Elemento ***<network>*:** Este elemento es la raíz de la jerarquía, y su justificación es bastante obvia, puesto que es la representación de lo que queremos serializar. Contiene los atributos *type* y *learningrate*. El elemento *type* permite identificar el modelo neuronal representado en el documento, mientras que *learningrate* representa la tasa de entrenamiento asociada a las neuronas de la red.
- Elemento ***<layer>*:** De acuerdo a la estructura de las clases que representan la estructura interna de las redes, se incluyó un elemento que representa la clase *SimpleLayer.* Este elemento es hijo de ***<network>*** y a su vez contiene los elementos que representan las neuronas dentro de la red. Este elemento contiene el atributo *type,* cuyo valor determina la función de transferencia de las neuronas pertenecientes a dicha capa.
- Elemento ***<neuron>*:** Este elemento representa la unidad fundamental de la red, la neurona y además es hijo de ***<layer>***. Contiene los atributos *sensitivity*, que representa la contribución al error total de la red (en el perceptron multicapa), y *bias* que representa el bias (sesgo) de cada neurona.
- Elemento ***<weight>*:** Este elemento es hijo de ***<neuron>*** y representa los pesos (weights) asociados a las conexiones externas de la neurona.

En el caso de los datasets, la estructura del respectivo documento XML refleja la disposición de los datos organizados en filas. Al momento del diseño del documento se tuvo en cuenta el hecho de que existen 2 tipos de datasets: De entrenamiento supervisado y sin supervisión, y se diseño una jerarquía que pudiese reflejar dichas características. Los elementos creados son los siguientes:

- Elemento **<*dataset*>:** Es el elemento padre de la jerarquía y representa el conjunto de datos de entrenamiento serializado. Sus atributos son *nombre*, el cual representa el nombre del dataset que NeuralStudio utilizará para identificarlo dentro del sistema; y *type*, que dice si el dataset es para entrenamiento supervisado o sin supervisión.
- Elemento **<*row*>:** Este elemento es hijo de **<*dataset*>** y representa un caso de entrenamiento (una fila de datos). Si el dataset es para entrenamiento supervisado, este elemento contendrá los elementos **<*input*>** y **<*output*>**, respectivamente; mientras que si se trata de un dataset para entrenamiento sin supervisión, solo contendrá elementos **<*input*>**.
- Elemento **<*input*>:** Este elemento representa las entradas a la red neuronal y es hijo del elemento **<*row*>**.
- Elemento **<*output*>:** Representa la salida deseada para las entradas correspondientes. Solo se encuentra en datasets para entrenamiento sin supervisión.
- Elemento **<value>:** Debido a que una red neuronal puede tener más de una entrada y/o más de una salida al tiempo, se incluyó el elemento **<value>** como hijo de **<*input*>** y **<*output*>**. Hay un elemento **<value>** por cada entrada o salida de la red neuronal.

```
<?xml version="1.0" encoding="UTF-8"?>
<network type='mperceptron' learningrate='0.011'>
<layer type='logsigmoid'>
<neuron sensitivity='-5.677291E-6' bias='-2.5245104'>
<weight>1.4596794</weight>
</neuron>
<neuron sensitivity='7.2095566E-4' bias='-1.2754928'>
<weight>0.24235605</weight>
</neuron>
<neuron sensitivity='0.0015643599' bias='-1.3767318'>
<weight>0.09276301</weight>
</neuron>
<neuron sensitivity='0.0012955125' bias='-5.9766393'>
<weight>1.1922125</weight>
</neuron>
<neuron sensitivity='-0.0032762368' bias='-7.9092107'>
<weight>1.1370659</weight>
</neuron>
</layer>
<layer type='purelin'>
<neuron sensitivity='0.0017806292' bias='1.0260576'>
<weight>3.2539034</weight>
<weight>-0.8441707</weight>
<weight>-2.002596</weight>
<weight>-4.4283586</weight>
<weight>3.679919</weight>
</neuron>
</layer>
</network>
```

Figura 23. Representación en XML de un perceptron multicapa

7.2.2.5.2 Implementación de la interfaz XML

La serialización se implementó con ayuda de la clase *StringBuffer* que nos permitió tomar las características de la red y ensamblar eficientemente el documento XML. Una vez terminado, el documento se guarda en disco, en un archivo XML.

En el caso de Java, contamos con varias herramientas para la interpretación de documentos XML, específicamente JAXP (Java API for XML Processing), compuesta por SAX y DOM. Se evaluaron todas las alternativas y la discusión se redujo al hecho que de utilizando SAX es posible interpretar un documento XML sin mantenerlo en memoria completamente, por lo que se comenzó a explorar la posibilidad de construir un manejador para la deserialización basado en SAX, y nació la clase *NeuralSaxHandler*.

NeuralSaxHandler extiende la clase *DefaultHandle*r, que es una de las clases básicas dentro de SAX. Esta clase declara una serie de listeners o handlers, que son métodos que se van a ejecutar en el evento en que un elemento XML se abra o se cierre. De esta manera se escribió código para recoger las características de la red representada en XML, y una vez terminado el análisis del archivo proceder a la instanciación del objeto correspondiente. El enfoque probó ser bastante adecuado para el almacenamiento de redes neuronales en disco.

```
<?xml version="1.0" encoding="UTF-8"?>
<dataset nombre='AND' type = 'supervised'>
<row>
<input>
<value>0.0</value>
<value>0.0</value>
</input>
<output>
<value>0.0</value>
</output>
</row>
<row>
<input>
<value>0.0</value>
<value>1.0</value>
</input>
<output>
<value>0.0</value>
</output>
</row>
<row>
<input>
<value>1.0</value>
<value>0.0</value>
</input>
<output>
<value>0.0</value>
</output>
</row>
<row>
<input>
<value>1.0</value>
<value>1.0</value>
```

Figura 24. Estructura XML de un dataset

8 CONCLUSIONES Y RECOMENDACIONES FINALES DE LA INVESTIGACIÓN

A partir del desarrollo de esta investigación se puede concluir que efectivamente Java es una herramienta idónea para el desarrollo de un proyecto de características similares al considerado en este trabajo. Durante la implementación de los sistemas presentados, se pudieron resolver los retos que aparecieron, tales como los relacionados a la complejidad de los algoritmos de entrenamiento, la representación de las redes a través de colecciones, el entrenamiento multi-hilos, etc., de manera elegante, tal como se puede comprobar en la implementación realizada. El vasto número de librerías, frameworks y recursos en general, que están disponibles en el ecosistema Java es realmente superior al de otras alternativas consideradas.

El problema de modelar las diferentes arquitecturas de redes neuronales, junto con las relaciones que existen entre ellas fue resuelto elegantemente mediante la aplicación de los principios de la programación orientada a objetos (OOP). La reutilización del código, encapsulación y polimorfismo permitieron el cumplimiento del objetivo de construir una librería con implementaciones eficaces de los modelos neuronales escogidos. Una jerarquía de clases conformada por clases abstractas e implementaciones concretas en sus diferentes niveles, se constituye (debido a lo cual fue efectivamente implementada) en la mejor alternativa para garantizar la extensibilidad a futuro de la Liberia JANE.

Por otro lado, la representación en XML de redes neuronales se implementó de manera satisfactoria, dejando una buena impresión en el sentido de que la estructura de una red neuronal se pudo expresar de manera muy intuitiva y sencilla a través de XML. Esto posibilita la interoperabilidad de la herramienta NeuralStudio con el gran mundo de recursos y el inmenso ecosistema generado en torno a XML.

De acuerdo a consideraciones históricas, de aplicabilidad práctica y complejidad de implementación, se escogió un grupo de modelos neuronales para conformar la librería. Dicho grupo está compuesto por el Perceptron, Adaline, Perceptron Multicapa, la Red de Aprendizaje Competitivo y el Kohonen SOM. Estas redes son una muestra bastante representativa de los modelos que han sido considerados hitos dentro del campo y que han alcanzado aplicabilidad practica en muchos ámbitos del mundo real.

Además fue posible concluir que las aplicaciones con mayor potencial dentro del ámbito de NeuralStudio son la aproximación de funciones y la clasificación de datos/reconocimiento de patrones. Ambas permiten ver claramente las fortalezas del uso de redes neuronales en problemas bien específicos y con gran relevancia en el mundo ingenieril.

Este trabajo representa un punto de partida para la investigación propia sobre redes neuronales. Los puntos de extensión están bien establecidos dentro de los sistemas desarrollados, y quedan ciertas tareas que a juicio del autor sería importante desarrollar en el futuro cercano, las cuales serán formuladas como recomendaciones:

- Inclusión en JANE de nuevos algoritmos de entrenamiento, en especial el entrenamiento basado en enjambres de partículas (Particle Swarm Optimization), para lo cual sería deseable una refactorización de las clases encargadas del entrenamiento y terminar de desplazar las responsabilidades de entrenamiento hacia los descendientes de *AbstractNetTrainer*.
- Inclusión en JANE de nuevas modelos neuronales, especialmente ART (Adaptive Resonance Theory), debido a las interesantes oportunidades de investigación que su implementación y aplicación ofrece.

- Inclusión de la aplicación de predicción temporal en NeuralStudio, puesto que esta es una aplicación realmente importante y prometedora de las redes neuronales, donde superan inclusive a las mejores técnicas estadísticas disponibles.
- Desarrollo de una interfaz web para NeuralStudio, de manera que se pueda ir de la mano con las tendencias actuales de desarrollo de software y convertirlo en un sistema multiusuario.

El trabajo continuado sobre los sistemas desarrollados en este trabajo representará grandes oportunidades en el mediano y largo plazo en el campo pedagógico, investigativo e ingenieril, debido a la relevancia que los sistemas de reconocimiento avanzado de patrones y predicción tienen dentro del acerbo conceptual de los nuevos profesionales de las ciencias de la computación y la ingeniería.

9 REFERENCIAS

[1] Hagan, Martin; Demuth, Howard. *Neural Network Design*, 1996, PWS Publishing Company. Num. páginas: 730.

[2] Hebb, Donald. *The organization of behavior*, 1949, Wiley: New York. Num. páginas: 335.

[3] Arbib, Michael. *Turing Machines*, Finite Automata and Neural Nets, Journal of the ACM volume 8, 1961, pp. 467-475.

[4] *MIT, DARPA. DARPA Neural Network Study, 1988. Num páginas: 608.*

[5] McCulloch, W. and Pitts, W. (1943). *A logical calculus of the ideas immanent in nervous activity.* Bulletin of Mathematical Biophysics, 7:115 – 133.

[6] *Rosenblatt, F. The perceptron: A probabilistic model for information storage and organization in the brain. Psychological review, vol 65, 1958. pp 386-408*

[7] Minsky, Marvin; Papert, Seymour. *Perceptrons*, 1969, MIT Press.

[8] Lin, Cheng-Jian; Hsieh, Ming-Hua. *Classification of mental task from EEG data using neural networks based on particle swarm optimization*. Journal of Neurocomputing volume 72, 2009, pp 1121-1130. Elsevier Science Publishers B. V.

[9] Apolloni, Bruno; Marinaro Maria; Tagliaferri, Roberto. *Biological and Artificial Intelligence Environments*. 15th Italian Workshop on Neural Nets, WIRN WIETRI 2004.

[10] Chuangxin, Guo; Quanyuan, Jiang; Xiu,Cao; Yijia, Cao. *Recent Developments on Applications of Neural Networks to Power Systems Operation and Control: An Overview*, 2004, pp 216-218. , Springer Berlin/Heidelberg.

[11] Meissner, Michael; Schmuker, Michael; Schneider, Gisbert. *Optimized Particle Swarm Optimization (OPSO) and its application to artificial neural network training*. BMC Bioinformatics, 2006, pp 125-136.

[12] Nunamaker JR, Jay F.; Chen, Minder; Purdin, Titus. *Systems Development in Information Systems Research*. Journal of Management Information Systems, Vol. 7, No. 3, pp 89-106. M. E. Sharpe Inc.

[13] Booch, G. *Object-oriented design*. Ada Lett. I, 3 (Mar. 1982), 64-76.

[14] Meyer, Bertrand. *Object-Oriented Software Construction, 2nd Edition. Prentice Hall Professional Technical Reference.*

[15] Coad, P.; Yourdon, E. *Object-Oriented Analysis*. Yourdon Press 1991.

[16] Coad, P.; Yourdon, E. *Object-Oriented Design*. Yourdon Press 1991.

[17] Wirfs-Brock, R.; Wilkerson, B.; Wiener, L. *Designing Object-Oriented Software*. Prentice-Hall, Inc. 1990.

[18] Rumbaugh, J.; Blaha, M.; Premerlani, W.; Eddy, F.; Lorensen, W. *Object-Oriented Modeling and Design*. Prentice-Hall, Inc. 1991.

[19] Booch, G. *Object Oriented Design with Applications*. Benjamin-Cummings Publishing Co., Inc. 1991.

[20] Riehle, D.; Züllighoven, H. *Understanding and using patterns in software development. Theor. Pract*. Object Syst. 2, 1 (Nov. 1996), 3-13.

[21] Gamma, Erich; Helm, R.; Jonhson, R.; Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading MA. 1994