NEUROSCHEME: UN LENGUAJE PARA EL MODELAMIENTO DE REDES NEURONALES ARTIFICIALES

NEUROSCHEME: A MODELING LANGUAGE FOR ARTIFICIAL NEURAL NETWORKS

JUAN DAVID VELÁSQUEZ HENAO
Grupo de Inteligencia Artificial, Facultad de Minas, Universidad Nacional de Colombia
jdvelasq@unalmed.edu.co

Recibido para revisión 5 de Agosto de 2004, aceptado 28 de Marzo de 2005, versión final 10 de Junio de 2005

RESUMEN: La construcción de modelos basados en Redes Neuronales Artificiales, no solamente comprende la parte de creación de la red neuronal como tal y su entrenamiento, si no también, una cantidad de labores que se realizan antes y después de esta fase. En el mercado existen muchas herramientas para el modelamiento de Redes Neuronales Artificiales, sin embargo, ellas no brindan la versatilidad necesaria para cumplir con estas tareas adicionales. Para satisfacer esta necesidad, se implementó el Lenguaje Algorítmico NeuroScheme, el cual incorpora los modelos de Redes Neuronales Artificiales como un tipo de dato nativo dentro del lenguaje, y que es caracterizado por su capacidad expresiva, elegancia y simplicidad.

PALABRAS CLAVE: redes neuronales artificiales, lenguajes de programación, lenguajes declarativos.

ABSTRACT: Building of neural network-based models, is conformed not only by the creation of the artificial neural network and its training, but other number of activities realized before and after of this phase too. In the market exists many computational tools for artificial time series modelling, however, they are not enough versatile for doing these additional task. For satisfying this necessity, the Neuroscheme algorithmic language is developed, which incorporate the artificial neural networks models as a native type of data inside the language; it is characterized by its expression capacity, elegance and simplicity.

KEYWORDS: artificial neural networks, programming languages, declarative languages.

1 INTRODUCCIÓN

La construcción de sistemas de solución de problemas basados en Redes Neuronales Artificiales (RNA), comprende un grupo de tareas que van más allá de la especificación del modelo y su posterior entrenamiento, las cuales comprenden, el análisis de la información de entrada, la selección de los patrones para entrenamiento, preparación de la información para entrenar la RNA, la identificación del modelo, su entrenamiento, la validación de los resultados, y su aplicación propiamente dicha en la solución del problema particular en cuestión.

Estas tareas pueden llegar a involucrar procesos complejos de cálculo tales como pruebas estadísticas, procesos de descomposición de los datos de entrada, técnicas de preprocesamiento de la información de entrada, normalización de datos o estimación de intervalos de confianza. Desafortunadamente, no hay un derrotero a seguir en la forma que deben realizarse todos estos pasos, ya que ellos dependen en gran medida del problema particular que se esta resolviendo. Esta situación se agrava cuando una sola RNA no es suficiente para resolver el problema con la precisión adecuada, y es necesario combinarla con otras RNA, para formar un sistema de varias RNA, o con otras técnicas tales como modelos econométricos o estadísticos.

En la actualidad existen muchas herramientas comerciales para la construcción de modelos de RNA, de las cuales, la gran mayoría esta orientada a un usuario final que no domina la programación de computadores. Por este motivo, dichas herramientas se concentran en la parte de entrenamiento del modelo de RNA, construyendo un ambiente gráfico alrededor de este objetivo, que le permite al usuario la creación de modelos y su posterior entrenamiento. Para las demás tareas mencionadas, se dan, en algunas ocasiones, unas pocas opciones al usuario.

Visto de una manera integral, esta forma de ejecutar el proceso no es deseable, ya que se desperdician esfuerzos en el aprendizaje de las distintas herramientas y en el intercambio de información que en la mayoría de los casos es manual.

Si se analiza desde la necesidad de integrar diferentes paradigmas, este proceso es prácticamente inviable, ya que no se da ningún tipo de prestación para la construcción de este tipo de sistemas.

Puede concluirse entonces, que es necesaria una herramienta que permita realizar todos los pasos descritos para la preparación de modelos de RNA, así como la integración de estos con otros paradigmas, de tal forma que dicha integración sea natural, rápida y sencilla, manteniendo unos estándares de calidad, rapidez y eficiencia.

2 DESCRIPCIÓN DE LA HERRAMIENTA

Para el desarrollo de la herramienta se ha seleccionado como lenguaje base al dialecto Scheme (Sussman et al, 1975; Steele et al, 1978; Dybig, 1987) que proviene de Lisp (McCarthy, 1960), el cual se encuentra fundamentado en el cálculo lambda (Church, 1941; Curry and Feys, 1958); este lenguaje es caracterizado por su elegancia conceptual y simplicidad, por lo que es ampliamente usado en la enseñanza de principios de programación y la investigación en lenguajes de programación. A diferencia de otros dialectos de Lisp, Scheme define el ámbito de sus definiciones léxicamente, es estructurado en bloques, soportando funciones y continuaciones como objetos de datos de primera clase, lo que quiere decir, que ellos pueden ser pasados como parámetros a funciones, retornados como el valor de una función, y permanecer indefinidamente en memoria.

Los programas escritos en Scheme están basados en funciones y recursividad, más que en comandos y ciclos, usando con poca frecuencia asignación de variables.

En general, y no solamente sobre el tópico de la herramienta que se está bosquejando, la programación funcional presenta varias ventajas sobre la programación imperativa, que es bien ejemplificada por lenguajes de programación como C o Pascal:

Primero, la programación funcional es mucho más simple, ya que las expresiones son construidas en forma natural inherentemente recursiva, mientras que los programas imperativos son construidos con base en declaraciones o comandos complejos que se combinan con expresiones.

Segundo, la programación funcional es mucho más fácil de entender, ya que cada pieza de código ejecuta una tarea específica, facilitando el seguimiento del código.

Tercero, es posible ejecutar pruebas para validar el programa de una manera mucho más simple en un lenguaje declarativo que en uno imperativo, debido a la misma regularidad del código.

Cuarto, las variables locales se declaran directamente como parámetros de las funciones, inicializándose siempre al ser invocada una función, y representa más el nombre para un valor que una localización en memoria, tal como ocurre en C o Pascal.
Quinto, es posible alternar el orden de evaluación de las expresiones en los lenguajes funcionales, un aspecto que es imposible modificar en un lenguaje imperativo,
Y sexto, es posible modificar el orden de ejecución de las funciones, de tal forma que pueden construirse complejas instrucciones con retornos no locales, lo que facilita complejas estructuras de control, capacidad que no está presente en ningún lenguaje imperativo.
Una prueba de la simplicidad de la herramienta, se puede evidenciar en la especificación de la misma, la cual consta de unas cincuenta páginas aproximadamente. Sin embargo, y a través del uso de características tales como la manipulación de la ejecución, es posible crear estructuras complejas de control, tales como los bloques try...catch de C++.
Scheme es un lenguaje declarativo débilmente tipado, es decir, el tipo de la variable se determina durante la fase de ejecución y no de compilación como ocurre en C o Pascal. Scheme permite usar funciones como argumentos en la llamada a otros funciones, e inclusive permite devolver funciones como resultado de la aplicación de funciones.
Estas características le dan una capacidad de expresión aún más potente que la del lenguaje C, pero dejando a un lado temas como el manejo de punteros, que hacen de C un lenguaje difícil de aprender. Es por esta razón que los programas escritos en Scheme son más cortos que sus equivalentes en C o Pascal.
Así mismo, y debido a su simplicidad y elegancia conceptual, el lenguaje promueve la construcción de abstracciones complejas a través del uso de barreras de abstracción, facilitando la construcción de sistemas complejos.
Por otro lado, la comunidad científica ha utilizado ampliamente Lisp y sus dialectos en aplicaciones de Inteligencia Artificial, y es normal encontrar dentro de sus ejemplos típicos de programación, la implementación de algoritmos de búsqueda, sistemas de razonamiento similares a Prolog, o Sistemas de Producción. Y aunque se argumenta que Scheme es un lenguaje orientado a la enseñanza de principios de programación, y que para aplicaciones reales debería utilizarse Common Lisp, Scheme contiene todas las características deseables para realizar la construcción de sistemas para solución de problemas. Esto se ejemplifica en el proyecto Guile de la GNU [véase por ejemplo los siguientes sitios WEB www.schemers.org, www.swiss.ai.mit.edu/projects/scheme/], el cual es una librería escrita en Lenguaje C que implementa Scheme que se usa embebido dentro de la aplicación principal.
Estos hechos son la base para afirmar que la herramienta que se plantea para la integración de paradigmas es precisamente Scheme.

3 ARQUITECTURA DE LA HERRAMIENTA IMPLEMENTADA

3.1 NÚCLEO DE LA HERRAMIENTA

El núcleo de la herramienta es un interprete de Scheme basado en un montículo de memoria, similar a las primeras implementaciones de la herramienta, descrita por Sussman (1975) y que ha sido usada por otros autores (Dybig, 1983).
La herramienta se divide en cuatro grandes partes: un módulo de interfaz con el usuario, encargado de administrar las entradas del usuario, los mensajes del sistema, y utilitarios como la historia de comandos; un preprocesador de código fuente, capaz de manipular el código ingresado por el usuario para expandir las macros de forma higiénica, que es un mecanismo mucho más potente que los preprocesadores normalmente encontrados en lenguajes de alto nivel; un compilador que convierte el código fuente expandido a bytecodes o instrucciones de la máquina virtual del interprete; este compilador esta formado por un analizador léxico y un analizador sintáctico que

interactúan con el administrador del montículo de memoria, para generar el bytecode; y finalmente, una máquina virtual que ejecuta las instrucciones de bajo nivel representadas por los bytecodes, tal como puede observarse en la Figura 1.
La herramienta está completamente escrita en Lenguaje C (Kernigan & Richie, 1978), y hace uso intensivo de manejo de punteros, estructuras de datos, y administración de memoria dinámica. Las funciones primitivas proporcionadas por el intérprete, son también codificadas en Lenguaje C, permitiendo optimizar funciones críticas tales como algoritmos para el entrenamiento de RNAs.

3.2 FUNCIONES INCORPORADAS

El interprete soporta todas las funciones descritas en el Reporte del Lenguaje Algorítmico Scheme, así como otras funciones definidas en Common Lisp y Emacs Lisp, y otras extraídas de las bibliotecas de funciones del Lenguaje C. Dichas funciones son soportadas dentro del interprete a través de su codificación directa en Lenguaje C, o como macros que se expanden en secuencias de funciones codificadas como primitivas en C. Así mismo, se han implementado otras primitivas presentes en MIT Scheme y Dr Scheme, que no aparecen en los reportes, pero que son consideradas como necesarias para que la herramienta funcione de una manera adecuada.

3.3 INTERFAZ CON EL SISTEMA

El intérprete implementado puede cargar, compilar y ejecutar programas en código fuente almacenados en el disco duro del computador, posibilitando que otras aplicaciones puedan ejecutar programas dentro del interprete. Igualmente puede dejar registro escrito de la interacción con el usuario.

3.4 INTERFAZ CON EL USUARIO

El usuario interactúa con el sistema a través de una ventana de comandos donde digita las ordenes que el sistema debe ejecutar, o a través de la interfaz gráfica que presenta cajas de diálogo, que pueden ser accesadas a través de la barra de menús, y en las cuales el usuario proporciona la información necesaria para ejecutar los distintos comandos, tal como aparece en la Figura 2.
De igual forma, puede interactuar con el interprete a través de interfases de usuario mucho más elaboradas, tales como editores de texto o simuladores gráficos; sin embargo, cualquier sistema anexo, tales como los descritos, no manipulará ninguno de los componentes claves del interprete, debiendo interactuar con el a través del envío de comandos, usando cadenas de texto con comandos Scheme, las cuales serán interpretadas usando un minicompilador que posee el interprete dentro de su máquina virtual. De esta forma se garantiza un adecuado intercambio de información entre las distintas partes del sistema.
Como una prestación adicional, se han agregado diálogos como una alternativa para invocar funciones, en los cuales el usuario llena los parámetros requeridos, y tiene la opción de ejecutar directamente la función, o pegarla en la línea de órdenes como parte de un comando más complejo.

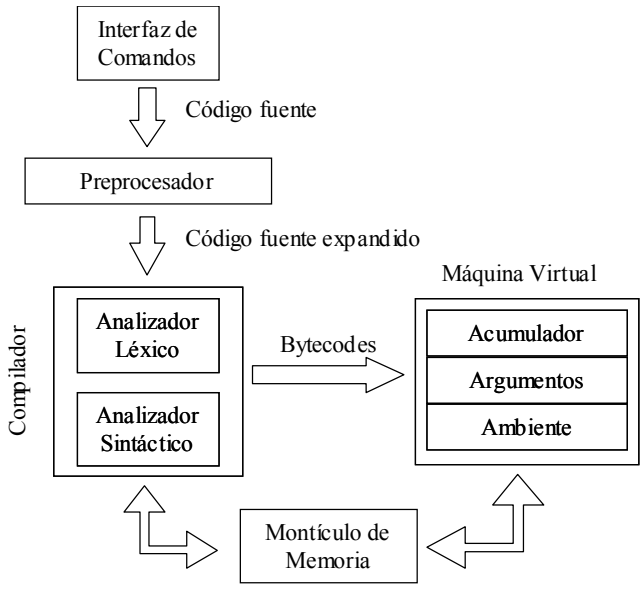

Figura 1. Esquema del Intérprete
Figure 1. Interpreter scheme

3.5 MODELOS DE REDES NEURONALES ARTIFICIALES

Los modelos de RNAs son manejados por el interprete como tipos primitivos de datos, de igual forma a como se manejan enteros, símbolos o cadenas de caracteres; y no como abstracciones codificadas directamente en Scheme. Esto hace de los modelos de RNA objetos de primer nivel que pueden ser almacenados en variables, pasados dentro de los argumentos de las funciones, e inclusive ser devueltos como resultado de ellas.

Las funciones para la creación y manipulación de los modelos de RNAs, son también codificados como primitivas, directamente en Lenguaje C, lo que permite tener código ejecutable altamente optimizado, debido a la criticidad que la velocidad de ejecución de estas funciones tiene dentro del desempeño de la aplicación.

La mezcla de código interpretado con código ejecutable, tiene un efecto muy importante en el desempeño de la herramienta, ya que todos los procesos críticos se ejecutan directamente en binario, mientras que el cascaron que esta encima de ellos, el cual permite codificar algoritmos complejos, y dar la funcionalidad requerida, se ejecuta más lentamente al ser interpretado, pero esto requiere un porcentaje muy pequeño del tiempo total de la aplicación.

El modelo interno para las RNAs permite construir cualquier tipo de arquitectura, y adicionalmente, el usuario puede crear nuevas arquitecturas eliminando conexiones o adicionando capas. En la actualidad, ya que el uso primordial dado hasta ahora para el entorno, ha sido la construcción de modelos de series de tiempo basados en RNAs, se han implementado tres modelos de propagación hacia delante: feedforward, cascada-correlación y cascada hacia delante; los algoritmos de entrenamiento implementados incluyen: montecarlo, regla delta generalizada, regla delta con momento, gradiente descendente, estrategias de evolución, temple simulado, así como versiones de algunas de ellas combinadas con mínimos cuadrados.

4 FUNCIONES SOPORTADAS

NeuroScheme soporta las siguientes funciones, las cuales aparecen referenciadas dentro de la especificación del Lenguaje.

- Condicionales y lógicas: If, cond, case, and, or, when, unless.
- Enlazado de variables: Let, let*, letrec.
- Secuenciamiento: begin
- Iteración: do, let, repeat, while, until, dotimes, dolist, loop
- Quasiquotation: quasiquote
- Asignación: set!, incr, decr, pset!, rotate!
- Equivalencia: eqv?, eq?, equal?
- Números: number?, real?, Rational? Integer?, exact?, inexact?, >, >=, <, <=, =, <>, even?, odd?, negative?, positive?, zero?, max, min, +, -, *, /, abs, quotient, remainder, modulo, gcd, lcm, numerator, denominator, floor, ceiling, truncate, round, exp, log, sin, cos, tan, cot, sec, csc, asin, acos, atan, acot, asec, acsc, sqrt, expt, exact->inexact, inexact->exact
- Números aleatorios: get-seed, rand-max, rand-uniform, rand-normal, inversa-normal
- Entrada y salida numérica: number->string, string->number
- Boléanos: not, bolean?
- Pares y listas: pair?, cons, car, cdr, set-car!, set-cdr!, caar, cadr, … cddddr, first, second, third, fourth, fifth, sixth, seventh, eighth, ninth, tenth, null?, list?, list, append, reverse, list-tail, list-ref, last-pair, list-copy, push, pop, push-new, memq, memv, member, assq, assv, assoc
- Símbolos: symbol?, string->symbol, symbol->string.

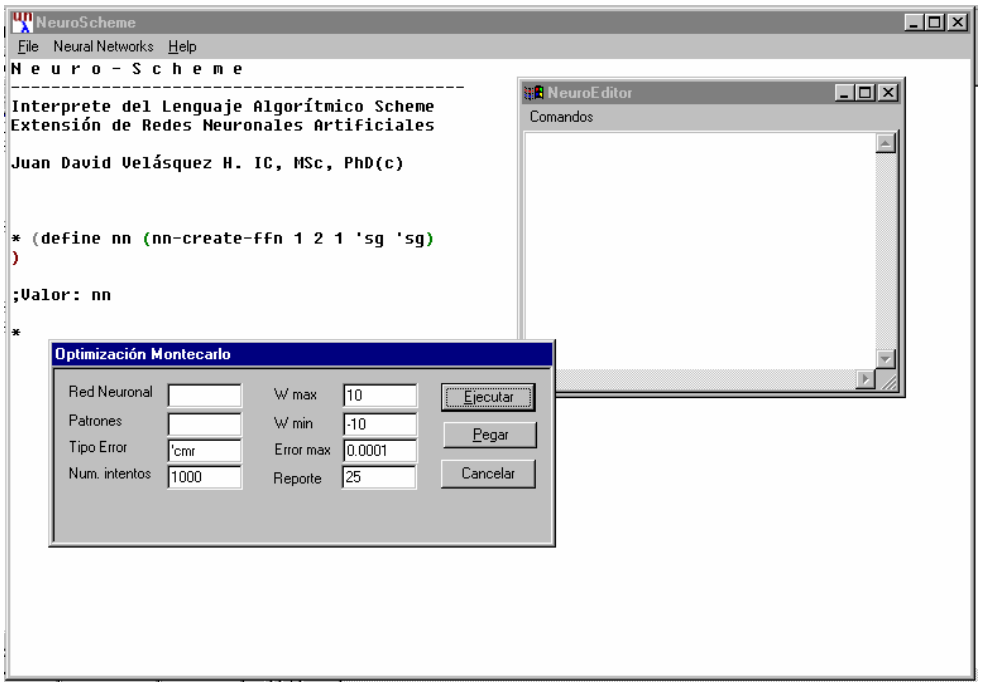

Figura 2. Interfaz de NeuroScheme
Figure 2. NeuroScheme user interface

- Caracteres: char?, char=?, char >?, char=>?, char<?, char<=?, Char-ci=?, char-ci>?, char-ci>=?, char-ci<?, char-ci=<?, char-alphabetic?, char-numeric?, char-whitespace?, char-upper-case?, char-lower-case?, char->integer char, integer->char, char-upcase, char-downcase
- Cadenas de Caracteres: string?, make-string, string-char, string-length, string-append, string-ref, string-set!, string=?, string-ci=?, string<?, string>?, string<=?, string>=?, string-ci<?, string-ci>?, string-ci<=?, string-ci>=?, substring, string->list, list->string, string-copy, string-fill!, string-downcase, string-upcase, string-downcase!, string-upcase!, substring-downcase!, substring-upcase!, reverse-string, reverse-string!, reverse-substring, reverse-substring!, string-head, string-tail, string-capitalize, string-capitalize!, substring-capitalize, substring-capitalize!, string-capitalized?, substring-capitalized,? string-upper-case?, substring-upper-case?, string-lower-case?, substring-lower-case?, string-replace, string-replace!
- Vectores: vector?, make-vector, vector, vector-length, vector-ref, vector-set!, vector->list, list->vector, vector-fill!, vector-first, vector-second, vector-third, vector-fourth, vector-fifth, vector-sixth, vector-seventh, vector-eighth, vector-ninth, vector-tenth, subvector, vector-head, vector-tail
- Control: procedure?, map, for-each, call/cc
- Evaluación: eval
- Puertos: input-port?, output-port?, open-input-filename, open-output-filename, open-append-filename, close-input-port, close-output-port.
- Entrada: read, read-char, eof-object?, char-ready?.
- Salida: write, display, newline, write-char.
- Interface: load, transcript-on, transcript-off, ed.

- Redes Neuronales Artificiales: nn-create-ffn, nn-create-ccn, nn-create-cfn, nn-propagate-signal, nn-display-input-weights, nn-display-neurons-by-layer, nn-display-bias-weights, nn-display-layer-weights, nn-display-output-neurons, nn-generate-random-weights, nn-calculate-error, nn-train-montecarlo, nn-train-backpropagation, nn-train-backpropagation-mom, nn-train-graddesc, nn-train-simanneal, nn-train-sa, nn-train-es, nn-calculate-mc, nn-get-input-weight, nn-get-bias-weight, nn-get-layer-weight, nn-set-input-weight!, nn-set-bias-weight!, nn-set-layer-weight!, nn-load-network, nn-save-network

5 RESULTADOS OBTENIDOS

En la actualidad, el primer prototipo es operable e incorpora el paradigma de las redes neuronales artificiales. Como ya se indicó, la red neuronal se modela como un tipo de dato propio del lenguaje, lo que implica que se encuentra codificada en Lenguaje C, haciendo que la velocidad de los algoritmos de entrenamiento sea tan alta como en las aplicaciones de usuario final disponibles hoy en día.

Como un ejemplo de la funcionalidad del entorno, se presenta en la Figura 3, un programa en NeuroScheme cuyo objetivo es entrenar una RNA para que aprenda la función gaussiana, a partir de valores (x, y) simétricos alrededor de cero. En el se ilustra un algoritmo complejo en el cual se ejecuta un proceso cíclico de generación de un nuevo punto de arranque y posterior entrenamiento de la RNA usando la Regla Delta Generalizada, así como la selección del mejor modelo encontrado.

La utilización de diálogos como mecanismo primario de comunicación con el usuario de la aplicación, permite que este con unas pocas horas de entrenamiento, y aún sin tener un conocimiento previo del lenguaje Scheme, pueda estar entrenando modelos de redes neuronales artificiales. Los diálogos, en nuestra aplicación, son una forma alterna de proporcionar comandos al interprete, ya que cuando el usuario indica su aceptación con respecto al contenido de la caja de diálogo, esta no interactúa directamente con el núcleo de la herramienta, si no que convierte la entrada del usuario en un comando de Scheme equivalente el cual es enviado al interprete para que sea procesado, haciendo eco en la pantalla, de tal forma que el comando es visualizado para ser ejecutado posteriormente.

La existencia de una línea de comandos, donde el usuario de la herramienta puede digitar directamente las instrucciones de forma interactiva, permite no solamente la creación o el entrenamiento de la red neuronal, sino también el uso de comandos más complejos como instrucciones para ejecutar instrucciones cíclicamente, automatizando diferentes procesos. En consecuencia, es posible por ejemplo, crear algoritmos complejos de entrenamiento donde, de acuerdo con los resultados obtenidos, se pueden alternar entre distintas técnicas de aprendizaje.

Adicionalmente, al contar con dicha línea de comandos, es posible construir modelos complejos de redes neuronales, donde las entradas de una red neuronal pueden estar formadas por las salidas de otros modelos, tales como otras redes neuronales, modelos de programación lineal, o modelos econométricos. Dentro de las primitivas implementadas en nuestra herramienta, se encuentra la función load, que recibe como parámetro el nombre de un archivo en disco, el cual contiene en formato de texto un grupo de comandos que se ejecutarán con dicha función.

6 CONCLUSIONES Y TRABAJO FUTURO

Las distintas fases que deben realizarse para la correcta construcción de modelos basados en RNAs, hace que se requiera seguir un proceso laborioso para llegar a un resultado adecuado. Es por ello, que en el desarrollo de estos modelos, es necesario contar con ambientes que vayan mucho más allá de las prestaciones normalmente encontradas para su entrenamiento.

```
(define normal-pattern         ; almacena en la variable normal-pattern
  '(((  0.00000 ) ( 1.0000 ))  ; 11 parejas (x, y)
    ((  0.54546 ) ( 0.7427 ))
    ((  1.09090 ) ( 0.3042 ))
    ((  1.63640 ) ( 0.0687 ))
    ((  2.18180 ) ( 0.0086 ))
    ((  2.72730 ) ( 0.0006 ))
    (( -0.54546 ) ( 0.7427 ))
    (( -1.09090 ) ( 0.3042 ))
    (( -1.63640 ) ( 0.0687 ))
    (( -2.18180 ) ( 0.0086 ))
    (( -2.72730 ) ( 0.0006 ))))

(seed 101) ; cambia la semilla del generador de números aleatorios

; crea la variable nn y almacena en ella una RNA feedforward con 1
; neurona de entrada, 2 ocultas y una de salida. La capa oculta tiene
; como función de activación la Sigmoidea y la de salida, transferencia
; lineal
(define nn (nn-create-ffn 1 2 1 'sg 'tl))

; salva la RNA en un archivo en disco
(nn-save-network nn "normal-best-bp.nn")

; define la variable best-error y almacena en ella el error cuadrático
; medio de la aproximación producida por la RNA
(define best-error (nn-calculate-error nn normal-pattern 'cm))


(repeat 20                       ; repite el siguiente ciclo 20 veces
  ; genera pesos aleatorios para la RNA entre -0.5 y 0.5
  (nn-generate-random-weights nn 0.5)

  ; entrena la RNA usando Regla Delta Generalizada
  (nn-train-backpropagation nn normal-pattern
     'cm 0.05 10000 0.0000001 5 )

  ; verifica si el error obtenido es menor que el mejor
  ; encontrado hasta ahora
  (if (> best-error (nn-calculate-error nn normal-pattern 'cm ) )
      ; si es menor, salva la nueva RNA y actualiza el valor
      ;de best-error
      (begin
        (nn-save-network nn "normal-best-bp.nn")
        (set! best-error
              (nn-calculate-error nn normal-pattern 'cm ) ))))
```

Figura 3. Programa en NeuroScheme que crea una red feedforward y la entrena para que aprenda la campana de Gauss
Figure 3. Neuroscheme program for creating a feerforward network, and training it for learning the Gauss bell.

Por esto, es necesario entonces, contar con una herramienta mucho más versátil que permita realizar este proceso. Se escogió como base del desarrollo, el dialecto Scheme, debido a sus características de elegancia, simplicidad conceptual, facilidad de uso y capacidad de abstracción, incorporando los modelos de RNAs como tipos de datos propios del lenguaje.

El interprete implementado es altamente versátil, permitiendo construir algoritmos para la manipulación y entrenamiento de RNAs, de tal forma, que es posible automatizar procesos, y construir modelos complejos de RNAs. Estas facilidades no están presentes en las herramientas comerciales para el desarrollo de modelos de RNAs.

La incorporación de cajas de diálogo a la interfaz, las cuales envían comandos directamente al intérprete, ha permitido que el usuario final pueda utilizar la herramienta con pocas horas de entrenamiento, aún sin necesidad de aprender el lenguaje de programación.

Los resultados obtenidos hasta ahora, permiten concluir que debe continuarse con el desarrollo de la herramienta, e incorporarse otros paradigmas como Redes Neurodifusas, y Sistemas Borrosos, para habilitar la herramienta para construir sistemas híbridos para solución de problemas.

7 REFERENCIAS

[1] ABELSON H and SUSSMAN G. 1984. Structure and interpretatión of Computer Programs. MIT Press.

[2] AHO A.V. and ULLMAN J.D. 1979. Principles of Compiler Design. Addison-Wesley Publishing Company.

[3] BACKER H.G. and HEWITT C. 1977. The Incremental Garbage Collection of Processes. Procedings of the Symposium on Artificial Intelligence. Published as SIGPLAN Notices 12, 8 (August 1977), pp 55-59

[4] BARTLEY D.H. and JENSEN J.C. 1986. The implementation of PC Scheme. Proceedings of the 1986 ACM Conference on Lisp and Functional Programming. Pp 86-93.

[5] CHURCH A. 1941. The Calculi of Lambda Convension. Annals of Mathematics Studies 6, Princenton University Press.

[6] CURRY H.B. and FEYS T. 1958. Combinatory Logic. North-Holland Publishing Company. Amsterdam.

[7] DYBVIG R. K.1987a. The Scheme Programming Language. Prentice Hall

[8] DYBVIG R. K.1987b. Three Implementation Models for Scheme. PhD Disertation.

[9] KERNIGHAN B.W. and RITCHIE D.M. 1978. The C Programming Language. Prentice-Hall, 1978.

[10] MCCARTHY J. 1960. Recursive Functions of Symbolic Expressions and Their Computation by Machine. CACM 3 (april 1960). Pp 184-195.

[11] PETZOLD C. 1996. Programming Windows 95. Microsoft Press. McGraw Hill

[12] STEELE G.L and SUSSMAN G.J. 1978. The Revised Report on Scheme. Massachusetts Institute of Technology Artificial Intelligencel. Memo 452.

[13] SUSSMAN GJ and STEELE GL. 1975. Scheme: an Interpreter for Extended Lambda Calculus. Massachusetts Institute of Technology Artificial Intelligence Memo 349.

[14] STEELE G.L. Common Lisp The Language. Second Edition.