Universidad Central "Marta Abreu" de las Villas
Facultad de Matemática Física y Computación

Trabajo de Diploma
Licenciatura en Ciencia de la Computación

NEngine v1.0

Una Herramienta Software para Redes Neuronales Recurrentes

Autor:

Sain Salazar Martínez

Tutores:

Lic. Isis Bonet Cruz
Dra. Maria Matilde García Lorenzo
Dr. Ricardo Grau Ábalo

Santa Clara
2005

A mi esposa, a mis padres, mis amigos.

Agradecimientos

A mis tutores y a todos los que contribuyeron en la realización de este trabajo.

Resumen

En el presente trabajo se mencionan algunos de los modelos unificadores existentes para redes neuronales, y se analizan las ideas esenciales para una posible plataforma de software; se propone una arquitectura para dicha plataforma basada en procesos concurrentes, y se desarrolla una estructura de clases que sirve de bloque constructivo para el desarrollo futuro de la plataforma. Se diseña una herramienta que incorpora algunas de las ideas manejadas en el enfoque unificador y se particulariza en la implementación del algoritmo Backpropagation estándar para redes feedforward y el algoritmo Backpropagation Through Time para redes recurrentes. Se prueba la herramienta con bases internacionales para modelos de clasificación feedforward y con un problema de análisis de secuencia para algoritmos recurrentes.

Abstract

In the present work we mention some of existent unifier models for neuronal networks; the essentials ideas are analyzed for a possible software platform. We propose an architecture for this platform based on concurrent processes that interact one with another; additionally a structure of classes is developed that serves as constructive block for the future development of the platform. A tool is designed that it incorporates some of the ideas managed in the unifier approach and it is particularized in the implementation of the standard backpropagation algorithm for feedfoward networks and backpropagation through time algorithms for recurrent networks. The tool is proven with international bases for classification with feedforward models and with a problem of sequence analysis for recurrent algorithms.

Tabla de Contenidos

Introducción 1

Capítulo I. Análisis de un Framework para Redes Neuronales Recurrentes. 6

I.1. Redes Neuronales Artificiales. 6

I.1.1. El Modelo de la Neurona. 7

I.1.2. El Esquema de Conexión de la Red Neuronal. 8

I.1.3. Los Algoritmos de Aprendizaje. 11

I.1.4. Modificaciones al Algoritmo Backpropagation. 14

I.2. Redes Neuronales Recurrentes. 16

I.2.1. Redes Recurrentes Discretas. Backpropagation ThroughTime (BPTT). 21

I.2.2. Aplicaciones de las Redes Neuronales Recurrentes. 26

I.3. Herramientas de Software Existentes para Redes Neuronales Artificiales. 27

I.4. Un Framework Unificador para Redes Neuronales 31

I.4.1. Dos Enfoques Unificadores 31

I.4.2. Una Máquina Virtual Neuronal basada en un Framework Unificador. 32

I.4.3. Funcionamiento General de la Máquina Virtual. 35

I.5. Conclusiones del Capítulo. 37

Capítulo II. Diseño e Implementación de una Herramienta para Redes Recurrentes. 39

II.1 Arquitectura de una Máquina Virtual para el Framework Unificador. 39

II.2. Diseño e Implementación de una Estructura de Clases para una Arquitectura Basada en Procesos Concurrentes 42

II.3. El *NEngine*, una Herramienta para el Trabajo con Redes Recurrentes. 51

II.3.1. El Proceso SysObjectTask. 53

II.3.2. El Proceso ExecTask. 53

II.3.3. El Proceso IOTask. 58

II.3.4. El Modelo de Sincronización. 60

Capítulo III. Manual de Usuario del NEngine v1.0. 62

III.1 Introducción al NEngine 62

III.1.1. Instalación. 63

III.2 Funcionalidades Básicas del NEngine. 64

III.2.1 Construcción de Topologías.....65
III.2.3 Entrenamiento de Modelos de Red Neuronal. 68
III.2.4 Salva y Recuperación de Proyectos. Explotación de Modelos. 70
III.3 Manual de Referencia de Comandos..... 71
III.4 Pruebas de Validación 78

Conclusiones 82

Recomendaciones 83

Bibliografía. 84

Anexo 1. Un Framework Basado en Diagramas de Bloques..... 89

1.1. Esquema del Proceso de Ejecución..... 89
1.2. El Proceso de Administración de Objetos. 93

Anexo 2. Extensión de la Estructura de Clases para Procesos Concurrentes. 96

Anexo 3. Sitios Web de Interés. 104

Anexo 4. Diagramas de la Estructura de Clases para la Gestión de Procesos Concurrentes..... 105

4.1 Diagrama de Clases del Paquete *Communication Infrastructure*..... 105
4.2 Diagrama de Clases del Paquete *Scheduling Infrastructure*..... 106

Anexo 5. Diagramas de la Herramienta NEngine 107

5.1 Diagrama de Casos de Uso. 107
5.2 Diagrama de Clases para Manipular Topologías 108

Introducción

Hoy día las redes neuronales constituyen uno de los campos de mayor desarrollo dentro de la Inteligencia Artificial; una muestra de ello es el amplio repertorio de aplicaciones existente en el ámbito académico, industrial, sector comercial, etc. En particular, ha cobrado sumo interés el desarrollo de las redes neuronales recurrentes. Las redes recurrentes poseen importantes características de cómputo que las hacen muy eficientes en el procesamiento de estructuras de información complejas donde estén presentes relaciones temporales y espaciales: el procesamiento de imágenes, series de tiempo, la generación de señales y la lingüística son contextos en los que las redes recurrentes cuentan con resultados notables.

Específicamente en el grupo de Bioinformática de la UCLV se han encontrado problemas que son susceptibles de ser tratados con las redes recurrentes: la predicción de estructura secundaria de proteínas y la extracción de descriptores para la predicción de resistencia a antivirales son dos ejemplos inmediatos. El tema de la predicción de estructuras secundarias posee avances significativos en los trabajos de Baldi [3], Qian y Sejnowski [47], etc. Este tipo de redes, tiene por demás, una importancia propia debido a sus posibilidades de aplicación en muchas otras esferas; y es precisamente esta característica la que llega a formular un reto a los especialistas en Ciencias de la Computación en el sentido de facilitar u optimizar su construcción.

Por otra parte, el surgimiento de máquinas computadores más veloces y de mejores prestaciones ha posibilitado un medio de investigación y experimentación ideal que ha contribuido aún más al desarrollo en este campo. Ésta es la razón por la que existe un amplio repertorio de herramientas y paquetes de software comerciales dedicados a facilitar el trabajo de los especialistas con los modelos de redes neuronales. Igualmente, cada vez es mayor el número de centros académicos y de investigación que procuran el desarrollo de herramientas propias que viabilicen la creación y validación de nuevas teorías en redes neuronales y sus aplicaciones. Los avances en el campo de la ingeniería de software permiten, a su vez, abordar la construcción de sistemas más complejos e integradores que ofrezcan entornos de trabajo mucho más productivos, y faciliten el desempeño de los usuarios no especialistas en programación. En el área de las redes neuronales se hace imprescindible disponer de herramientas

como éstas, donde la exploración y experimentación entre la variedad de modelos existentes, se pueda realizar de una manera simple y homogénea.

Esta necesidad puede quedar más clara si se tiene en cuenta que encontrar un modelo adecuado de red neuronal para la resolución de un problema real resulta una tarea trabajosa, debido a la complejidad y la gran variedad de modelos existentes. Cada modelo consiste en una arquitectura[1] específica de interconexión de las neuronas con un modelo particular de activación y un conjunto de algoritmos de entrenamiento. Generalmente, la selección del modelo particular a emplear no obedece a un formalismo lógico previamente establecido, siendo más bien un proceso subjetivo en el que vale, ante todo, la experiencia y la intuición del especialista. La situación puede complejizarse aún más si consideramos otros factores como la introducción de heurísticas de entrenamiento (cálculo dinámico de *velocidad de entrenamiento,* planificación de los conjuntos de entrenamiento, etc.) y conocimiento a priori sobre el problema determinado.

El proceso de búsqueda de la mejor solución puede verse como un conjunto de cuestionamientos que el especialista debe realizarse. El empleo de softwares para la simulación de redes neuronales puede brindar un apoyo inestimable en la búsqueda de las mejores respuestas para la solución final; sin embargo, la mayoría de ellos abarca una limitada gama de modelos y algoritmos de entrenamiento, a la vez que carecen, por una vía sencilla, de posibilidades de extensión a nuevos modelos y heurísticas. Generalmente el especialista debe dominar un gran repertorio de estos softwares en aras de disponer de un conjunto de herramientas suficientemente amplio; lo que limita el desarrollo flexible y homogéneo de la investigación. Esto sin contar la limitación que pueda existir en la capacidad de integración con otros enfoques computacionales (métodos Bayesianos, Sistemas Basados en Reglas, Sistemas Difusos, Computación

[1] El término *arquitectura* se usa aquí para referirse a un esquema o patrón determinado de conexión de las neuronas en una red. Pueden existir infinidad de redes que muestren el mismo esquema de conexión; aunque cada una presente diferencias en cuanto a cantidad de neuronas, conexiones, capas (para los patrones que puedan organizarse en capas, por ejemplo *MLP*). El término *topología* se emplea entonces para referir el aspecto concreto de una de estas redes. Varias topologías diferentes pueden obedecer a una misma arquitectura.

evolutiva, etc.) que puedan ser aplicados conjuntamente para la solución del problema[2].

En efecto, se hace evidente la necesidad de disponer de un ambiente de desarrollo que brinde la posibilidad de especificación y ejecución de los más diversos modelos de redes neuronales existentes, así como la posibilidad de extensión, combinación y creación de nuevos modelos, si es requerido. En realidad, hasta ahora han sido creados varios *simuladores* (SNNS [74], NeuroSolution [30], etc.) y *lenguajes de especificación* (Aspirin [36], CONNECT [43], etc.) con este propósito; pero todos presentan limitaciones de flexibilidad y alcance que dificultan el desarrollo sistemático de soluciones a problemas reales.

Un software con las características enunciadas puede desarrollarse a partir de la construcción de una máquina virtual [57] con la capacidad de abstraer cada modelo de red neuronal sobre una plataforma que represente los conceptos esenciales presentes en la computación neuronal.

La UCLV es uno de los centros puntera en nuestro país en el tema del desarrollo y aplicación de las redes neuronales. Se destacan los trabajos en la plataforma para redes *Backpropagation* SmartMlp [10], y la herramienta NeuroDeveloper [17] para la experimentación de redes asociativas. La experiencia alcanzada en este sentido en nuestro centro y el conocimiento que se tiene sobre las técnicas de ingeniería de software constituyen antecedentes idóneos para enfrentar un proyecto de integración que agrupe la totalidad de los desarrollos disponibles en forma de un producto único y homogéneo.

El presente trabajo pretende abordar esta problemática a partir del estudio de uno de los modelos de integración para redes neuronales existentes en la literatura, y la elaboración de una herramienta para redes recurrentes sobre una arquitectura que, basada en estas ideas, permita la evolución progresiva hacia una plataforma homogénea y flexible para redes neuronales. En este sentido se plantea el siguiente:

[2] Aunque este tipo de integración ha emergido recientemente y no hay bases teóricas rigurosas para su aplicación generalizada, es recomendable que las herramientas concebidas consideren la incorporación futura de esta posibilidad, una vez que existan trabajos desarrollados en este sentido.

Objetivo General

Contribuir al desarrollo de las herramientas de software para la creación, entrenamiento y explotación de modelos de redes neuronales recurrentes discretas a partir de una arquitectura de software extensible, con el fin de facilitar la aplicación de dichos modelos a problemas de análisis de secuencias en el área de la Bioinformática o análisis de señales en general.

Objetivos Específicos

1) *Resumir algunos de los diversos enfoques de modelos unificadores existentes en la literatura, que sistematicen los conceptos esenciales de las redes neuronales artificiales y seleccionar uno de ellos.*
2) *Diseñar e implementar una herramienta que incluya posibilidades para la construcción y entrenamiento de modelos de redes MLP y redes recurrentes; que permita una extensión futura hacia una plataforma unificadora para redes neuronales.*
3) *Validar la herramienta obtenida partiendo de bases de casos experimentales.*

El trabajo queda estructurado por capítulos de la siguiente manera:

En el Capítulo I se estudian los conceptos esenciales del fundamento de las redes Neuronales Artificiales, profundizando en los aspectos particulares de los modelos de redes recurrentes, su topología, algoritmos de entrenamiento y aplicaciones; se examina un modelo unificado para redes neuronales y se detalla el esquema general de una posible plataforma homogénea basada en este modelo.

En el Capítulo II se analiza una variante de arquitectura para la plataforma discutida en el capítulo anterior y se realiza el diseño y la implementación de una estructura de clases de soporte para el desarrollo futuro de la plataforma. Se hace una extensión de esta estructura de clases para obtener una herramienta que posibilite el diseño de topologías de redes neuronales recurrentes y feedforward, y su entrenamiento a partir de los algoritmos *Backpropagation Through Time* (BPTT) y *Backpropagation* respectivamente.

En el Capítulo III presentamos el manual de usuario de la herramienta NEngine v1.0 donde se muestran los detalles de la explotación (manual de instrucciones, ejemplos de uso, requerimiento de hardware) y algunas pruebas de validación realizadas.

Capítulo I. Análisis de un Framework para Redes Neuronales Recurrentes.

En el presente capítulo presentamos una descripción de los conceptos esenciales involucrados en el trabajo con las redes neuronales artificiales: modelos de neurona, topologías, algoritmos de entrenamiento, etc. Señalamos en particular los modelos de redes recurrentes y el algoritmo de entrenamiento *Backpropagation ThroughTime (BPTT)* para redes recurrentes discretas. Se presentan algunas aplicaciones de las redes recurrentes y herramientas para su explotación. Se examina un modelo unificado para redes neuronales y se detalla el esquema general de una posible plataforma homogénea para el trabajo con las redes neuronales.

I.1. Redes Neuronales Artificiales.

Muchos de los problemas del mundo real, debido a su carácter ambiguo e impreciso, no son susceptibles de ser resueltos con el empleo de enfoques simbólicos formales, o por medio de procedimientos algorítmicos bien establecidos. Por esta razón se han desarrollado métodos sub-simbólicos, entre los cuales se encuentran las redes neuronales. Las redes neuronales se agrupan dentro de las técnicas conexionistas de la Inteligencia Artificial y constituyen una de las especialidades más ampliamente difundidas. El objeto principal de este enfoque es la *red neuronal*. Las redes neuronales son herramientas matemáticas para la modelación de problemas, que permiten obtener las relaciones funcionales subyacentes entre los datos involucrados en problemas de clasificación, reconocimiento de patrones, regresiones, etc. Son consideradas excelentes aproximadores [24] de funciones, siendo capaces de aprender las características relevantes de un conjunto de datos, para luego reproducirlas en entornos ruidosos o incompletos.

Una red neuronal puede definirse como un conjunto de unidades computacionales (neuronas) interconectadas por medio de arcos pesados a manera de grafo dirigido. El objetivo de tal red (que puede verse como una caja negra) es brindar (calcular) una salida Y a partir de una información X recibida con anterioridad. Usualmente las redes destinan un conjunto de neuronas para la entrada (*neuronas de entrada*) de la información proveniente del exterior y un conjunto distinto para ofrecer las salidas

(*neuronas de salida*); el resto se consideran *neuronas ocultas*. Se concibe el cálculo general de la red a partir de la información que es procesada por cada una de sus neuronas en forma independiente. Cada una de ellas puede recibir información de las restantes y calcular su propia salida a partir de dicha entrada y de su *estado* actual, transitando eventualmente hacia un nuevo *estado*. Por lo general, el flujo de cálculo de la red avanza progresivamente desde las neuronas de entrada hacia las neuronas de salida, en un proceso en el que cada una de las neuronas ocultas va activándose progresivamente según el esquema de conexión particular de cada red [25].

Una red neuronal puede ser caracterizada por el *modelo de la neurona*, el esquema de conexión que presentan sus neuronas, o sea su *topología*, y el *algoritmo de aprendizaje* empleado para adaptar su función de cómputo a las necesidades del problema particular.

I.1.1. El Modelo de la Neurona.

Siguiendo el modelo natural (células neuronales en los animales), la neurona artificial posee un esquema simple de cálculo en una sola dirección. Recoge los niveles de señal en sus entradas y las procesa según una función de activación (que puede ser lineal o no lineal y es dependiente del estado actual en que se encuentre dicha neurona), brindando un nivel concreto de señal en la salida. El modelo más frecuente es el que se describe en la figura 1.1.

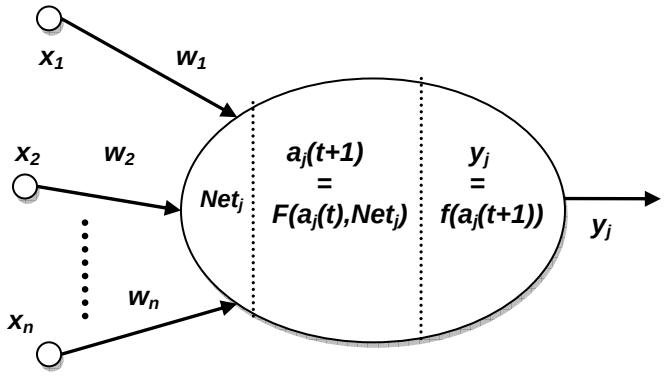


Figura 1.1. Modelo matemático de la neurona. (Tomado de [25])

El valor de entrada neta Net_j de la neurona es una combinación lineal de las señales provenientes de las neuronas de entrada más un valor constante w_0 llamado *umbral*.

$$Net_j = \sum_i x_i w_i + w_0 \qquad \text{(I.1)}$$

Los coeficientes de esta combinación lineal constituyen valores asociados a cada una de las conexiones de entrada a la neurona, denominados *pesos de las conexiones.* Estos constituyen los parámetros libres del modelo, que permiten el reajuste adecuado a la solución de los problemas concretos.

La función de activación F determina el nuevo estado de activación $a_j(t+1)$ teniendo en cuenta el estado actual $a_j(t)$ y la entrada neta Net_j. La función de salida f brinda la salida real de la neurona a partir de su estado de activación.

Existe una amplia variedad de modelos de neuronas, cada uno se corresponde con un tipo determinado de función de activación y de salida. Igualmente el cálculo de la entrada neta puede ser realizado de múltiples formas: en algunos modelos inspirados en la Biología puede conservarse una memoria de corto período del estado de la neurona o pueden concebirse fórmulas más complejas que ayuden a establecer tipos de procesamiento de mayor potencia. Las funciones de salida mayormente empleadas entre las discontinuas son las funciones mixtas (función a*x+b*, junto al establecimiento de umbrales hacia los extremos) y la función escalón. Entre las continuas son la función *sigmoidal* (I.2) (puede presentarse en varias variantes) y *softmax* (I.3) las de mayor empleo [25] en tareas de clasificación; y la función lineal $y=x$ en tareas de regresión. Para las neuronas de las capas ocultas suele emplearse con mucha frecuencia la función tangencial (I.4) [55].

$$y(net) = \frac{1}{1+e^{-net}} \quad \text{(I.2)} \qquad y_k = \frac{e^{net_k}}{\sum_{i \in O} e^{net_i}} \quad \text{(I.3)} \qquad y(net) = \frac{e^{net} - e^{-net}}{e^{net} + e^{-net}} \quad \text{(I.4)}$$

I.1.2. El Esquema de Conexión de la Red Neuronal.

La forma específica de conexión (arquitectura) y la cantidad de neuronas conectadas (el número de parámetros libres) que intervienen en una red determina directamente el poder computacional de la misma. En los últimos años se han producido una amplia variedad de arquitecturas de redes neuronales, sin embargo, la mayoría de ellas se encuentran ubicadas en dos grandes grupos: las redes muilticapa de alimentación

hacia adelante (*Feed-Forward Neuronal Networks, FFN)* y las redes recurrentes (*Recurrent Neuronal Networks, RNN*) [57], [25], [64], [24].

La característica fundamental de las redes *FFN* reside en que sus neuronas están conectadas a manera de grafo acíclico dirigido (con todos sus arcos en una sola dirección). Este tipo de red define una relación de orden parcial entre sus neuronas y con frecuencia éstas pueden agruparse en forma de capas siguiendo dicha relación.

Las redes *Multi Layer Perceptron (MLP)* constituyen un ejemplo genérico de las redes *FFN*. En general se encuentran formadas por un conjunto de capas de neuronas ordenadas secuencialmente de la forma siguiente: una *capa de entrada*, un conjunto de capas intermedias denominadas *capas ocultas* y una *capa de salida.* En la figura 1.2 se muestra un ejemplo típico de este tipo de red.

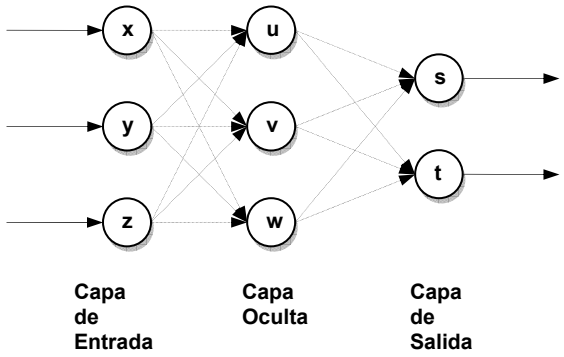


Figura 1.2. Una red MLP de 3 capas.

Cuando la red no tiene capas ocultas (*single-layer network*) ésta corresponde al *Perceptron* de Rosemblatt [25], que constituyó el modelo precursor de las redes *MLP.* La principal diferencia entre el *perceptron* y el *MLP* reside en la potencia de cómputo: el *perceptron* solo es capaz de separar linealmente (por un *hiperplano*) las entradas; mientras que el *MLP*, usando neuronas ocultas con funciones no lineales, es capaz de aproximar cualquier tipo de función continua y brindar excelentes resultados en las tareas de clasificación [78], [24]. El poder de representación de las redes *MLP* está relacionado con la existencia de al menos una capa de neuronas ocultas no lineales, las cuales transforman la entrada en una representación interna. Esta representación interna puede servir luego a las capas subsecuentes para resolver los problemas

tratados. En este sentido Lippman [35] señala el poder de representación arbitrariamente complejo que puede llegar a presentar un *MLP* de 4 capa.

Las *RNN* poseen una diferencia notable con la clase anterior: las redes recurrentes admiten conexiones hacia atrás, o sea, pueden formar ciclos en el grafo que describe sus conexiones. Estas conexiones hacia atrás, llamadas de retroalimentación (*feedback loops*) son las que permiten que la red sea capaz de guardar una memoria de los estados anteriores para su uso en el cálculo del estado y las salidas corrientes, o sea, mantener una especie de *memoria* de los procesamientos pasados; ésta es la característica esencial que convierte a este tipo de redes en una herramienta de amplio uso en tareas de reproducción de señales y análisis de secuencias, donde se reflejan relaciones causales en el tiempo y el espacio. De manera general son aplicables a problemas reales que reflejan relaciones dinámicas y estructurales de orden complejo [57], [46], [3].

En la figura 1.3 se muestra un ejemplo simple de una red recurrente de dos capas con conexiones hacia atrás. La misma puede emplearse para el análisis por etapas de una secuencia de valores de entrada donde cada salida está influenciada no solo por la entrada actual, sino por el estado de la red en la etapa anterior. O sea, el estado computado con la entrada provista a la red en el tiempo *t*, es usado junto a la entrada provista en el tiempo *t+1* para calcular el nuevo estado y la salida en *t+1.* Los *operadores de desplazamiento* (*shift operators*) d^{-1} representados en las conexiones hacia atrás señalan la sincronización de las dependencias entre etapas. Un exponente positivo advierte que la red requiere el estado computado en etapas anteriores y un exponente negativo, que se requiere el estado de etapas posteriores[3].

Existen otras arquitecturas, por ejemplo: las máquinas estocásticas (que usan las ideas de la mecánica estadística) y las redes Hopfield [25], en las que todas las neuronas están conectadas entre sí.

[3] El uso de exponentes negativos puede no tener sentido en redes donde se simula la reconstrucción de señales en el tiempo; este tipo de exponentes se emplea con mayor frecuencia en problemas de análisis de datos estructurales. A las redes que resuelven este tipo de problema se le suele llamar *recursivas,* dejando el término *recurrentes* para aquellas donde se manejen relaciones temporales. En todos los casos el uso de los *shift operators* es un recurso para la notación en las redes discretas.

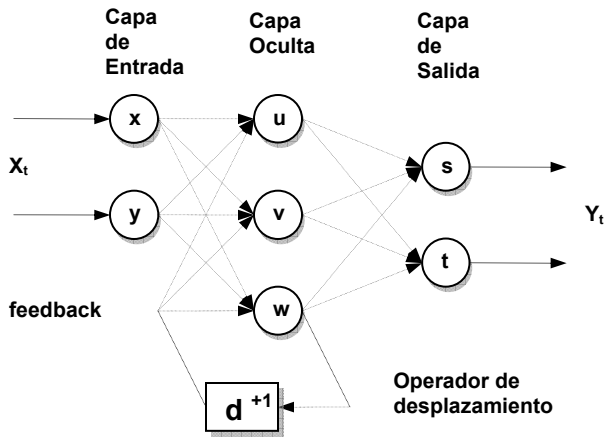


Figura 1.3. Red recurrente discreta con una conexión hacia una etapa anterior.

Una vez caracterizado un problema, la topología más adecuada para su solución debe ser seleccionada de manera empírica. Por un lado la potencia de cálculo para resolver los problemas complejos depende directamente de la cantidad de neuronas (cantidad de parámetros libres) de que disponga la red, y por otro, un número muy elevado de éstas puede afectar el desempeño de los algoritmos de entrenamiento. Tal situación conlleva a adoptar soluciones de compromiso entre el poder de representación de la red y su simplicidad. Por lo general la aplicación de heurísticas y técnicas especiales junto a la experiencia del especialista constituyen la forma más efectiva de abordar este problema.

I.1.3. Los Algoritmos de Aprendizaje.

Las redes neuronales poseen dos fases diferentes de trabajo: la *fase de entrenamiento* y la *fase operativa*. Ambas pueden estar mezcladas en ciertos modelos. La fase operativa es donde la red está disponible para el trabajo en modo óptimo de acuerdo al problema que trata de resolver. La fase de entrenamiento consiste en la aplicación de alguna técnica para lograr que la red ajuste sus parámetros, de manera que aprenda los aspectos esenciales del problema y logre la generalización de rasgos no previstos en el propio proceso de entrenamiento.

Los algoritmos de entrenamiento constituyen métodos que se aplican sobre los modelos de red para adaptar sus parámetros libres y obtener un comportamiento determinado. Con frecuencia los algoritmos de entrenamiento son caracterizados por la

clase de topologías sobre las que se aplica, los tipos de parámetros libres que afecta (pesos de las conexiones entre neuronas, parámetros del algoritmo de entrenamiento, la topología misma de la red, etc.) y la regla de modificación de los mismos (forma específica en que los parámetros libres son afectados). Existe una amplia variedad de algoritmos de entrenamiento disponibles, la mayoría de ellos desarrollados para arquitecturas muy específicas. Cada uno se clasifica usualmente en *supervisado* o no *supervisado* [25]. Los algoritmos supervisados son los que requieren, en su funcionamiento, información sobre las respuestas esperadas de la red, y los no supervisados los que no la requieren. Los algoritmos supervisados adaptan la red mediante la optimización de una función de *error* (que calcula las diferencias entre las respuestas deseadas y las obtenidas por el modelo, tomando como referencia un conjunto de datos fijos denominado conjunto de entrenamiento) sobre el espacio de los parámetros libres del modelo. Las técnicas supervisadas son caracterizadas por sus propiedades de convergencia global o local, rapidez con que logran dicha convergencia y los recursos en tiempo y espacio que requiere para su aplicación.

Entre las técnicas de entrenamiento supervisado más difundidas se encuentran aquellas que basan su funcionamiento en el método del gradiente descendente [64]. Entre éstas, el algoritmo *Backpropagation (BP,* aplicado a redes *feed-forward* de forma directa) es el de más amplio uso. Backpropagation aplica la técnica gradiente descendente para la minimización del error de funcionamiento de la red. El error de la red para un patrón p determinado puede calcularse como la semisuma de los errores cuadráticos de cada unidad de salida al presentarse dicho patrón a la red:

$$E_p(W) = \frac{1}{2}\sum_{i \in O}(y_i - d_i)^2 \qquad \text{(I.5)}$$

En la fórmula (I.5), W es el conjunto de pesos de las conexiones de la red (parámetros libres), y_i y d_i corresponden respectivamente a la salida actual y la salida esperada de la $i\text{-}ésima$ neurona de la red, dado un patrón de entrada determinado; O es el conjunto de neuronas de salida. El error total de funcionamiento de la red puede tomarse como la suma de los errores $E_p(W)$ para cada uno de los patrones presentes en el conjunto de entrenamiento. El error calculado por la fórmula (I.5) es empleado frecuentemente en

redes con funciones sigmoidales o lineales en la capa de salida. En tareas de clasificación múltiple donde se emplea *softmax* como función de activación, el error de un patrón puede calcularse mediante la fórmula conocida como *cross-entropy* para clasificación binomial o múltiple (I.6).[55]

$$E_p(W) = -\sum_{i \in O} d_i \ln(y_i) \qquad \text{(I.6)}$$

La función $E_p(W)$ se representa sobre un espacio multidimensional de dimensión igual al número de pesos de la red. La búsqueda de la solución (conjunto de pesos) que minimice el error se realiza de acuerdo a la fórmula (I.7).

$$\Delta w_{ij} = -\alpha \frac{\partial E(W)}{\partial w_{ij}} \qquad \text{(I.7)}$$

BP es capaz de calcular la magnitud de la influencia de cada peso sobre el error de respuesta de la red y realizar un reajuste de los mismos de manera proporcional (por medio de la constante de *velocidad de entrenamiento* α).

Las redes con aprendizaje no supervisado no requieren influencia externa para ajustar sus parámetros libres, o sea, no reciben información alguna sobre datos de entrenamiento. Su funcionamiento se basa en la construcción de clases o categorías a partir de las características comunes extraídas a los datos de entrada presentados. Dada una entrada, son capaces de construir la clase a la que pertenece, aún si la información brindada es parcial o ruidosa. Entre las técnicas más difundidas se encuentra el *aprendizaje hebbiano* y el a*prendizaje competitivo y cooperativo* [25]. En el aprendizaje hebbiano, la idea básica consiste en establecer la actualización de los pesos entre neuronas a partir de una magnitud proporcional a la activación conjunta de éstas, de manera que una conexión es reforzada si ocurre la activación simultánea de sus neuronas y es debilitada siempre que una de ellas (o ambas) se encuentre desactivada. Entre los modelos más conocidos se encuentran las *Memorias Asociativas Bidireccionales (Bidirectional Associative Memory BAM)* [33]. En el aprendizaje competitivo y cooperativo las neuronas compiten entre ellas a través de conexiones inhibitorias o participan cooperativamente a través de conexiones excitadoras en la realización de una tarea dada. En este tipo de redes solo una

neurona o un grupo de ellas es activada cada vez que es presentado un patrón en la entrada. La red evoluciona de acuerdo a las contribuciones de cada neurona hacia sus vecinas, logrando sobresalir (activarse) o minimizar su influencia (desactivarse) total en la respuesta final. Entre las técnicas más representativas se encuentra la denominada *Learning Vector Quantization (LVQ)* y los llamados *mapas auto-organizativos (self-organization maps)* [31], [56], [61].

Referido al problema mencionado en el epígrafe anterior sobre la selección adecuada de la topología, han sido propuestas varias técnicas (para redes *FFN y RNN*) que permiten su modificación dinámicamente como parte del proceso de entrenamiento. Entre las técnicas más usadas se encuentran los *algoritmos constructivos* y la *poda*. Los métodos constructivos, llamados también de crecimiento, permiten comenzar con una configuración mínima e ir añadiendo neuronas y conexiones paulatinamente hasta obtener una topología adaptada al problema en cuestión (*Cascade-correlation* [16], *Tiling*, *Upstart* [12]). La poda, al contrario, consiste en ir eliminando progresivamente las neuronas y conexiones menos útiles a partir de una red neuronal con una complejidad inicial arbitraria [60], [59].

I.1.4. Modificaciones al Algoritmo Backpropagation.

La correcta selección del valor de *velocidad de aprendizaje* α influye en el tiempo necesario para lograr la convergencia hacia el mínimo: la selección de un valor *pequeño* se traduce en un avance muy lento sobre la superficie de error, mientras que un valor demasiado *grande* posiblemente provoque una oscilación entre valores cercanos al mínimo sin llegar a éste. Una de las primeras propuestas para resolver este problema fue la introducción de un término *momento* que permite expresar una influencia del ajuste de los pesos en la etapa anterior sobre la variación de los pesos en la etapa actual [51].

$$\Delta w_{ij}^{\ t} = -\alpha \frac{\partial E(W)}{\partial w_{ij}} + \beta \Delta w_{ij}^{\ t-1} \qquad \text{(I.8)}$$

Esta modificación permite ajustar la longitud de actualización de los pesos de acuerdo a las características de la superficie de error en cada etapa, confiriendo mayor estabilidad al proceso de aprendizaje. De todas formas, la estimación de los valores

apropiados sigue siendo dependiente del problema particular que se quiera resolver, y su determinación requiere de la experiencia del especialista. Esta es la razón por la que se han desarrollado variantes de algoritmos que pretenden considerar las constantes de velocidad α y β como parámetros libres del entrenamiento.

De manera general, los algoritmos que son capaces de ajustarse automáticamente al problema que se pretende resolver se denominan *algoritmos adaptativos* [28], [29]. La mayoría de los algoritmos desarrollados en este sentido recogen información sobre el estado global del proceso, y con esta información actualizan los parámetros del algoritmo y los pesos de la red. Éste es el caso de los algoritmos SuperSab [62] y Delta-Bar-Delta [28], los que actualizan los parámetros que afectan la velocidad del aprendizaje (*alfa* y *beta*) a partir del comportamiento de la función de error.

Otra variante más eficiente de algoritmos adaptativos basados en el gradiente es el RPROP (*Resilient Backpropagation*) [49], cuyo funcionamiento se basa en considerar el cambio de los pesos proporcional a una magnitud Δ_{ij}, ajustable en el proceso de entrenamiento a partir de los cambios de signo del gradiente $\partial E / \partial w_{ij}$. Algunas variantes mejoradas de este algoritmo son discutidas en [26]. La principal ventaja de esta propuesta consiste en que evita la dependencia directa del cambio de los pesos de la red respecto a la magnitud del gradiente, evitando los problemas de convergencia que puede traer el comportamiento inestable de la misma.

BP no garantiza teóricamente la reducción del error a un mínimo global, sin embargo, los resultados obtenidos en su aplicación a diversos problemas prácticos han afirmado su versatilidad y aceptable nivel de eficiencia. Adicionalmente, Tsei y Gori [65] apuntan que cuando el espacio de los patrones de entrada es linealmente separable, se puede garantizar una convergencia global bajo ciertas condiciones de aplicación del algoritmo. Algunos autores han examinado la posibilidad de contrarrestar este problema a partir de la selección de valores de iniciación adecuados para los parámetros libres de la red. Diaz [14] propuso en este sentido, el uso de los algoritmos genéticos y técnicas con árboles de decisión. Yu en [73] adoptó un enfoque con redes de Petri que le permite seleccionar una topología y un conjunto de pesos iniciales más probables. El problema

de la convergencia local se ha abordado a partir del trazado de estrategias más eficientes para la presentación de los ejemplos de entrenamiento a la red. Cachin [11], Bonet [10] describen alternativas en este sentido.

Por otra parte, se ha considerado la aplicación de métodos de orden superior al lineal (BFGS [5], Gradiente Conjugado [41], [53]) para la obtención de mejores rendimientos en la convergencia.

I.2. Redes Neuronales Recurrentes.

Las redes recurrentes se consideran sistemas dinámicos cuyo estado evoluciona siguiendo un marcado comportamiento no lineal. Como se describió en el acápite anterior, constituyen redes con conexiones de retroalimentación (*feedback*) que exhiben dos tipos básicos de funcionamiento [57], [46]:

- *Sistema autónomo convergente*. Consiste en la evolución hacia un estado estable (punto fijo) siguiendo un proceso *autónomo* y partiendo de un conjunto de condiciones externas fijas (comportamiento convergente),
- *Sistema no autónomo no convergente.* Consiste en la variación del comportamiento en el tiempo de manera no *autónoma*, es decir exhibiendo cambios en el estado de las neuronas a partir de la influencia de estímulos variables recibidos en la entrada de la red a lo largo del tiempo (comportamiento no convergente).

El primer tipo de comportamiento permite realizar tareas de asociación (como las resueltas por los modelos de memorias asociativas [25]), la generación de señales con condiciones iniciales fijas, etc. El segundo permite el análisis y la reproducción de señales bajo condiciones variables en el tiempo, así como realizar tareas de clasificación y predicción en estructuras de información complejas. En la figura 1.4 se muestra una red recurrente sencilla que es capaz de resolver el problema del *XOR* con una evolución hacia un punto fijo [46].

La red de la figura 1.4 posee dos neuronas, una para cada una de las señales de entrada I_1 e I_2; una neurona de salida o y una neurona oculta h. Una conexión de retroalimentación (conexión recurrente) desde la neurona o hacia la neurona h permite

que la salida de la red dependa en cada momento del estado anterior. Suponiendo que la red trabaje actualizando continuamente los estados de activación de sus neuronas, la figura I.5 muestra el funcionamiento de la misma en modo convergente (estabilización hacia un punto fijo dependiente de las condiciones iniciales). En la figura se muestra el comportamiento asintótico de la red $y \xrightarrow{\infty} 0$ una vez que se fija en sus entradas la información correspondiente (I_1=1, I_2=0).

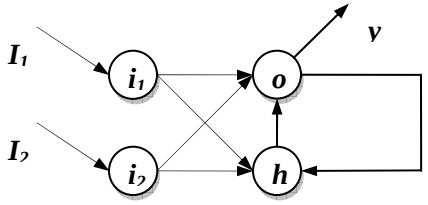


Figura 1.4. Red recurrente continua para el problema XOR.

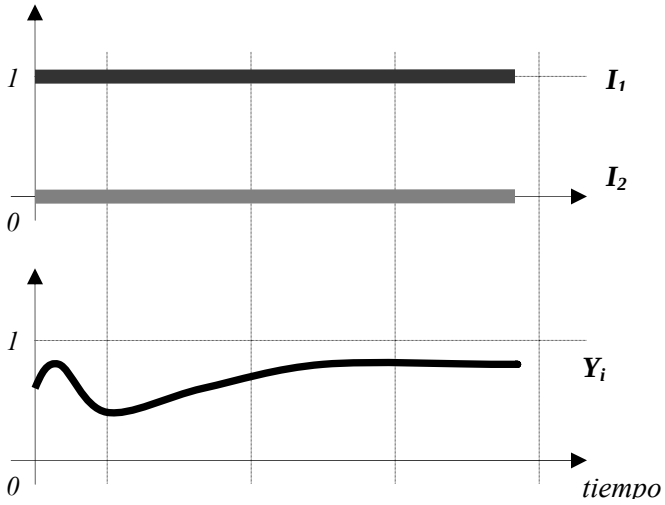


Figura 1.5. Funcionamiento convergente de la red recurrente para el problema XOR.

La figura I.6 muestra el comportamiento de la red en modo no convergente. La diferencia esencial es que la red no evoluciona esta vez hacia un estado estable, sino que varía su salida de manera dinámica, exhibiendo un comportamiento dependiente en cada momento de los valores de las señales de entrada.

De manera general, una red recurrente puede expresarse como un conjunto de ecuaciones de la forma (I.9) [77].

$$y_i(t) = \Phi_i\left(x_i(t), I_i(t)\right) \qquad \text{(I.9)}$$

Donde y_i denota el estado de activación de la neurona i en el tiempo t,

$$x_i = \sum_j y_j w_{ji} \qquad \text{(I.10)}$$

$I_i(t)$ denota la entrada exterior de la neurona i y Φ_i la función que gobierna el comportamiento del estado de activación de la neurona i en el tiempo t.

Φ_i constituye una función influida por la estructura[4] (topología) de la red (expresado en los valores de entrada netos x_i) y por la función de entrada $I_i(t)$ que aporta las señales provenientes del exterior.

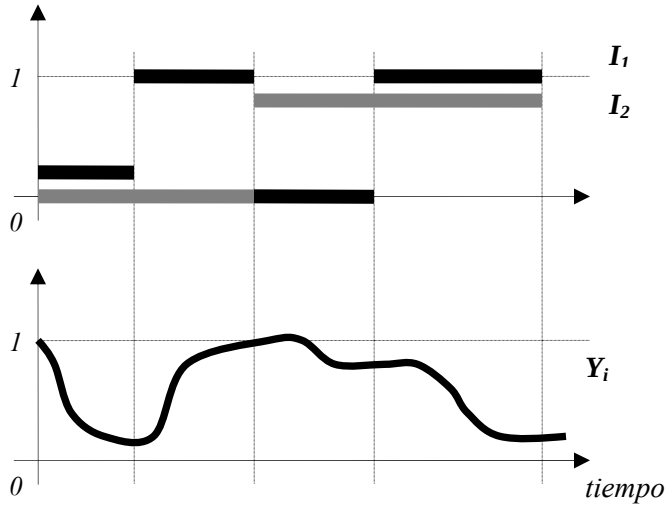


Figura 1.6. Funcionamiento no convergente de la red recurrente para el problema XOR.

Lo anteriormente dicho puede expresarse a través de las relaciones siguientes

$$N = (W, G(V,E), \{\Phi_i\}) \quad \left|\{\Phi_i\}\right| = |V|,\ \Phi_i : f \longrightarrow g,\ f, g \in F_{[\Re]}$$

$$L(W, G(V,E)) \longrightarrow \{\Phi_i\} \qquad W \in \Re^{|E|}, G \in \Im$$

[4] Nótese que una red recurrente es, como otras redes, un flujo de funciones interactuando entre sí (la salida de una es la entrada de otras incluyéndose a ella misma), por lo que una función de activación depende directamente de las demás, y la forma específica de esta dependencia está determinada por la estructura de las conexiones.

N representa una red neuronal con una topología que puede ser caracterizada por el grafo dirigido G, con un conjunto de nodos (neuronas) V y un conjunto de arcos (conexiones) E. Los pesos de las conexiones están expresados en el vector W. Estos elementos describen estructuralmente a la red. El conjunto de funciones de activación $\{\Phi_i\}$ brinda una caracterización dinámica del comportamiento de la red neuronal en el tiempo. La segunda expresión representa la aplicación L con dominio en $\Re^{|E|} \times \Im$ ($\Re$ es el conjunto de los número reales y $\Im$ el conjunto de los grafos dirigidos; cada elemento del conjunto producto representa una topología y un conjunto de pesos determinado para la red) e imagen en el conjunto potencia del conjunto de las funciones que transforman una función real en otra (en esta formalización solo se han tenido en cuenta las redes recurrentes con dominio de trabajo en los reales). La aplicación L expresa la relación existente entre estructura y comportamiento dinámico de las redes recurrentes [77], [1], [66], [23]. El objetivo principal de los investigadores en redes recurrentes estriba en el conocimiento de dicha relación y la producción de técnicas (algoritmos de entrenamiento, heurísticas) que permitan dominar su aplicación, es decir, encontrar las estructuras de redes adecuadas (topologías y pesos) para establecer un comportamiento dinámico correspondiente a los problemas que se pretendan resolver.

Resolver un problema determinado significa frecuentemente (como en el caso del problema XOR) *ajustar* las salidas de la red de manera que ésta exhiba el comportamiento requerido (en el caso del problema XOR se requiere que la salida Y se aproxime al valor 1, cuando las señales de entrada sean diferentes, y al valor 0 cuando sean iguales). Los valores de salida de la red a lo largo del tiempo son caracterizadas por las funciones de activación (que traducen una función de entrada $I(t)$ en una función de salida $y(t)$); y éstas a su vez dependen de la interacción de los parámetros libres (valores de los pesos en las conexiones w_{ij}) en un entorno (topología) determinado; por lo que, *ajustar* las salidas de la red, significa encontrar la topología G y el vector de pesos W que mejor aproximan cada función de activación de la red a la función real deseada. Esta situación es válida tanto para redes continuas como discretas. La diferencia sustancial entre estos dos tipos de redes reside en los

mecanismos que se emplean para deducir y aplicar las fórmulas relativas a las técnicas de entrenamiento.

Las redes con funciones reales que operan sobre una escala de tiempo continua se denominan *redes recurrentes continuas.* Igualmente se denomina *redes recurrentes discretas* a aquellas que expresan el comportamiento del estado de activación de las neuronas solo en puntos determinados t_0 , t_1, …, t_n en el eje del tiempo.

Muchos algoritmos exactos y aproximados han sido desarrollados para el entrenamiento de las redes recurrentes. La mayoría de ellos basan su trabajo en la técnica del gradiente descendente y pueden ser agrupados, según Atiya [1], en 5 clases generales: *Backpropagation ThroughTime (BPTT), Forward Propagation* o *Real Time Recurrent Learning (RTRL), Fast Forward Propagation, Funciones de Green y Actualización en Bloque (Block Update)*. Cada una de las formulaciones puede ser derivada para redes discretas y redes continuas de forma muy similar. En el caso continuo las técnicas se reducen, por lo general, a la aplicación de herramientas matemáticas propias a la solución de sistemas de ecuaciones diferenciales [46]; y en el caso discreto, la actualización de los estados de activación y de los pesos de las conexiones se realiza sincrónicamente por etapas siguiendo un orden determinado.

Entre las técnicas de gradiente descendente más empleadas para el entrenamiento de redes recurrentes se encuentran las variantes BPTT y RTRL [77], [9], [1], [67], [46], [64], [24], [70]. El algoritmo BPTT es una variante extendida del backpropagation original que posee requerimientos de tiempo y memoria crecientes, proporcional a la longitud de las secuencias que procesa. Solo es capaz de reajustar sus pesos una vez que toda la secuencia de entrada ha sido procesada, por lo que su aplicación a entornos de tiempo real es restringida. RTRL es un algoritmo derivado del BPTT por William y Zipser [77] para obtener una versión *on-line* del mismo. La principal ventaja de este algoritmo radica en limitar considerablemente la cantidad de memoria requerida por BPTT para su operación, a la vez que disfruta de la generalidad de aplicación de este último.

Existen variaciones del algoritmo BPTT (*Backpropagation ThroughStructure*, BPTS) para procesar estructuras de datos jerárquicas sin necesidad de transformarlos a un

formato de secuencia. Starita y Sperduti [58] han producido un enfoque sistemático para adaptar redes recurrentes y algoritmos de entrenamiento al análisis de estructuras de datos complejas. Hammer [22] presenta un estudio sobre las redes recurrentes aplicadas al procesamiento de estructuras de dato complejas; Siu-Yeung Cho [13] señala una heurística que mejora el desempeño de BPTS.

Bengio [6] realiza un análisis de la efectividad del BPTT aplicado a problemas donde se exige el aprendizaje de dependencias en secuencias muy grandes. Señala la imposibilidad de lograr, bajo ciertas circunstancias, el aprendizaje adecuado de cadenas de dependencias muy largas y a la vez lograr la capacidad de generalización en entornos ruidosos (problema del desvanecimiento del gradiente). A partir de un análisis teórico y experimental aporta sugerencias para mejorar los algoritmos y brindar conocimiento a priori en el proceso de entrenamiento.

Hammer brinda un informe abarcador en [23] sobre las perspectiva actuales de desarrollo de los algoritmos de aprendizaje en redes neuronales recurrentes. En el siguiente epígrafe centramos nuestra atención en el algoritmo *Backpropagation ThroughTime* para modelos de redes discretas.

I.2.1. Redes Recurrentes Discretas. Backpropagation ThroughTime (BPTT).

Una red discreta conserva el principio de funcionamiento general para las redes recurrentes, la diferencia radica en que la actualización del estado de activación de cada neurona se realiza en momentos puntuales de tiempo $t_1, t_2, \ldots t_n$, *etc.* La red para el problema XOR de la figura I.4 trabajando en modo convergente podría mostrar un comportamiento discretizado en el tiempo como se aprecia en la figura 1.7.

En cada momento de tiempo t_i la función de activación propia de cada neurona es calculada siguiendo el orden correspondiente. En la variante discreta de la red XOR resulta contradictorio la actualización simultánea de la neurona de salida *o* y la neurona oculta *h,* pues existe un ciclo de dependencia. En este caso, para el cálculo de la entrada neta de las neuronas *o* y *h* se emplean los valores de activación calculados en el momento de tiempo anterior (deben prefijarse los valores de los estados de activación para estas neuronas en el momento t_0, usualmente se escoge el valor 0). Si

aplicamos esta idea regresivamente desde el momento t_i hacia t_0 podemos establecer una relación de dependencia como la que se muestra en la figura 1.8.

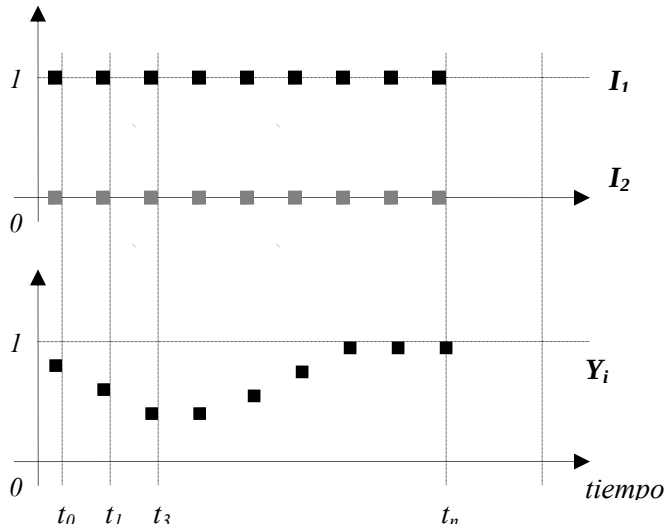


Figura 1.7. Funcionamiento convergente de una red recurrente discreta para el problema XOR.

La red neuronal así obtenida no es más que el resultado de la replicación de la red recurrente original (desprovista de las conexiones recurrentes) una vez por momento (etapa) de tiempo considerado en la secuencia. Las conexiones recurrentes conectan la neurona origen de la etapa anterior con la neurona destino en la etapa próxima. Cada conexión replicada es exactamente el mismo parámetro y comparte su valor w_{ij} en todas las etapas. El proceso mediante el cual se obtiene la red aumentada a partir de la red recurrente original se denomina *despliegue (Unfolding)*. Nótese que la red desplegada no es más que una red *feed-forward*, y es susceptible de la aplicación del algoritmo *Backpropagation* para su entrenamiento. Ésta es precisamente la idea subyacente en el algoritmo BPTT.

La red discreta puede representarse más concisamente con el empleo de los operadores de desplazamiento (*shift operators*) descritos en el acápite I.1.2. La figura 1.9 muestra las conexiones hacia la etapa posterior con el operador d^{+1}.

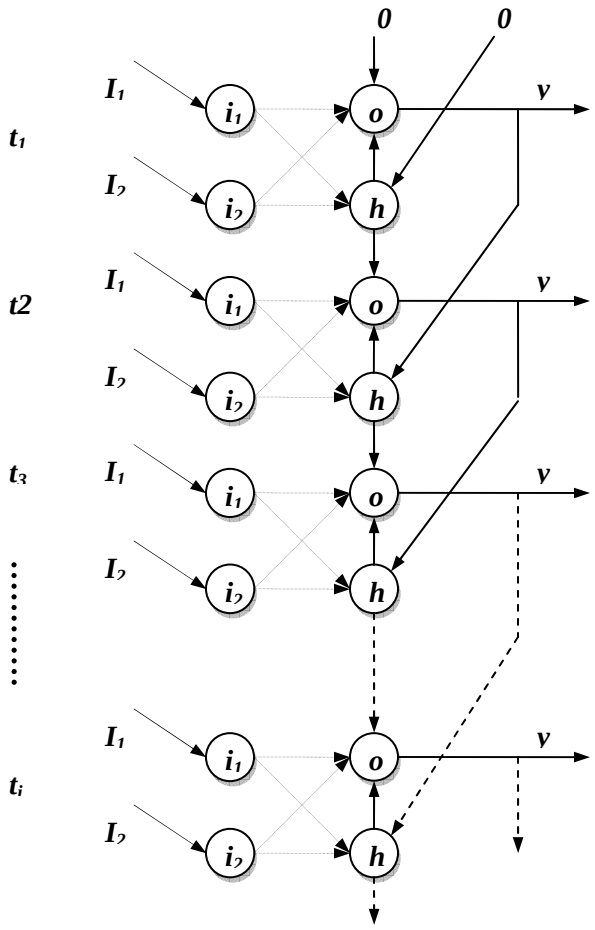


Figura 1.8. *Unfolding* de la red recurrente discreta para el problema XOR.

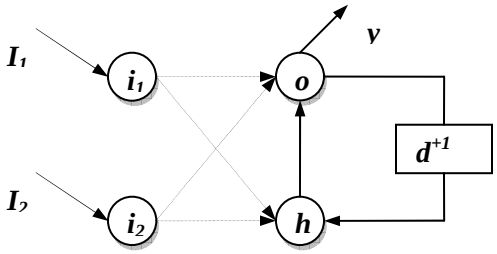


Figura 1.9. Red Recurrente Discreta para el problema XOR con operadores de desplazamiento.

En el ejemplo descrito hemos supuesto la conexión recurrente desde la etapa inmediata anterior. En sentido general, la conexión puede realizarse con un desplazamiento (*delay*) mayor que la unidad. Esta propiedad se emplea cuando se requiere que la red calcule los estados de las próximas etapas a partir de una información histórica más larga (la memoria de almacenamiento de los estados

anteriores puede hacerse tan larga como se necesite, siempre que los algoritmos de entrenamiento puedan hacer un uso efectivo de ella [6]).

Para las redes discretas la formulación (I.9) puede ser expresada más específicamente como:

$$y_i(t) = f(x_i(t)) \qquad \text{(I.10)}$$

$$x_i(t) = \sum_{j \in U} y_j(t) w_{ij} + \sum_{j \in I} x_j^{in} w_{ij} + \sum_{j \in S} y_j(t - \tau_{ij}) w_{ij} \qquad \text{(I.11)}$$

f se refiere a la función de activación del modelo de la neurona (frecuentemente una variante sigmoidal o tangencial); U denota los índices de las neuronas de la red (sin considerar las neuronas de entrada), I los índices de las neuronas de entrada, x_j^{in} es la *j*-ésima neurona de entrada. S denota los índices de las neuronas que almacenan la información de estados anteriores de la red (memoria de estado). $\tau_{ij} \geq 0$ es una magnitud entera que refiere el desplazamiento de las conexiones ***feedback*** en el tiempo (***shift operator*** $d^{-\tau}$, $d^{+\tau}$).

Más concisamente:

$$x_i(t) = \sum_{j \in U, j \in I} z_j(t - \tau_{ij}) w_{ij} \qquad \text{(I.12)}$$

$$z_j = \begin{cases} y_j, j \in U \\ x_j^{in}, j \in I \end{cases}$$

La notación ***shift operator*** puede usarse indistintamente siendo $d^{-\tau} z_j = z_j(t - \tau)$.

BPTT consiste en aplicar el proceso de *unfolding* a la red recurrente original, y ejecutar *Backpropagation* a la red *feed-forward* obtenida. *Backpropagation* (ver epígrafe I.1.2) posee dos fases: la primera, el cálculo de las salidas de las neuronas de la red (proceso *forward*), la segunda, la propagación del error hacia atrás (proceso *backward*). En el proceso de retropropagación del error cada neurona j es caracterizada por una magnitud de error δ_j. Para neuronas de la capa de salida (función de activación sigmoidal (I.2)) esta magnitud se calcula según:

$$\delta_j(t) = (d_j - y_j) y_j (1 - y_j) \qquad \text{(I.13)}$$

Para neuronas ocultas:

$$\delta_j(t) = y_j(1 - y_j) \sum_{i \in Suc(j)} wij\,\delta_i \qquad \text{(I.14)}$$

La actualización de los pesos de las conexiones se realiza según:

$$\Delta w_{ji}^{t+1} = \alpha \delta_j y_j \qquad \text{(I.15)}$$

Donde α corresponde a la tasa de convergencia.

En la fórmula, t no se refiere a las etapas de la red recurrente sino a un ordenamiento sucesivo de las actualizaciones de los pesos en el proceso de entrenamiento.

Suponiendo una red recurrente N y un conjunto de secuencias de valores de entrada y salida (conjunto de entrenamiento); un orden general de los pasos del algoritmo podría ser:

$P \longleftarrow\!\!\!\!- LearningParameters$, $\xi \longleftarrow\!\!\!\!- MeanError$

$S \longleftarrow\!\!\!\!- \{(\bar{I}, \bar{d})\}$, *Learning Pattern Set*

1. $E \longleftarrow\!\!\!\!- 0$, $\forall s \in S$
   a. $N' \longleftarrow\!\!\!\!- Folding(N, |s|)$
   b. $O \longleftarrow\!\!\!\!- Fordware(N', \bar{I}_s)$
   c. $e_s \longleftarrow\!\!\!\!- \delta - errorBackware(N', \bar{d}_s, O, P)$
   d. $N \longleftarrow\!\!\!\!- ActualizeSharedWeights(N')$
   e. $E \longleftarrow\!\!\!\!- E + e_s$
2. $E \longleftarrow\!\!\!\!- \dfrac{E}{|S|}$.
3. *If* $E > \xi$ *goto* 1
4. *End.*

La entrada del algoritmo consiste en un conjunto de patrones de entrenamiento (secuencia de entrada $\bar{I}$ con su correspondiente salida $\bar{d}$), los parámetros de

entrenamiento P, y el error medio de entrenamiento ξ esperado sobre el conjunto S. El primer paso consiste en la aplicación del procedimiento backpropagation en la red *feed-forward* N' asociada a la red recurrente original, que es obtenida por el *unfolding sobre* cada secuencia de entrada particular (1.a – 1.c). Las conexiones replicadas en la red N' que corresponden a la misma conexión original en la red N se consideran un solo parámetro libre compartido (usualmente las actualizaciones diferentes de este mismo parámetros obtenidas por el proceso de retropropagación en las distintas etapas, son combinadas para obtener un único resultado de actualización sobre la red original *N*). El paso 1.d realiza la actualización de cada conexión w_{ij} de la red original N a partir de los valores compartidos por cada etapa w_{ijt}. Una posible forma de combinar estos valores podría ser la simple suma de las actualizaciones obtenidas en cada etapa [9], [57], [3], [2].

$$\Delta w_{ij} = \sum_{t} \Delta w_{ijt}$$

La siguiente etapa verifica si se ha superado el parámetro de error aceptable para el aprendizaje y concluye la ejecución.

I.2.2. Aplicaciones de las Redes Neuronales Recurrentes.

Las redes recurrentes (RR) poseen amplia aplicación en el área de los sistemas dinámicos y de control, y el procesamiento de señales: en este sentido Parlos presenta en [45] la construcción de filtros adaptativos para circuitos eléctricos; Benson y Carrasco muestran en [7] el entrenamiento de una red recurrente para el reconocimiento y separación de patrones de señales encaminado a la ecualización de canales de comunicación móviles. Ginnakakis en [19] muestra la manera de aplicar las redes recurrentes a la reducción del ruido en señales y a la identificación de complejos de señales. En el área de la robótica, Kolb [32] y Ruiz [50] muestran el uso de las RR en la generación automática de movimientos y el aprendizaje de patrones de movimiento periódico en robots; por otra parte Ziemke [76] muestra cómo controlar el comportamiento en robots de forma adaptativa. Giles y Lawrence [20], [21] presentan sistemas híbridos de redes recurrentes para el análisis de series de tiempo aplicadas al pronóstico de magnitudes financieras.

Igualmente, diversas arquitecturas recurrentes han sido propuestas para resolver problemas en el área del procesamiento de imágenes: Ziemke [75] presenta un esquema para el reconocimiento de movimiento a través del examen de secuencias de imágenes en el tiempo; Nattkemper [42] aplica variedades de redes recurrentes para la extracción de contornos, localización de objetos en imágenes, etc. Topi y Parisi [63] extienden la técnica de empleo de redes MLP para la compresión de video usando redes recurrentes que ayuden a incorporar elementos temporales en el proceso de segmentación del mismo.

En el campo lingüístico se han obtenido resultados satisfactorios en el entrenamiento de redes recurrentes para el análisis de cadenas de símbolos generados por gramáticas sensibles al contexto [54]. Carrasco y Forcada aplican las RR para la inferencia de autómatas de estado finitos a partir de secuencias de cadenas ruidosas. En el procesamiento de lenguaje natural el trabajo propuesto por Frank y Mathis [18] destaca el uso de las RR para el reconocimiento de estructuras sintácticas del idioma inglés; igualmente Giles y Lawrence [34] proponen el uso de redes recurrentes con técnicas especiales de convergencia para aprender a reconocer oraciones gramaticales o no gramaticales en el inglés. Adicionalmente han sido aplicadas con exitosos resultados en el reconocimiento de voz en [4], [69].

Las redes recursivas gozan de amplia aplicación en el análisis de estructuras de datos complejas, particularmente Baldi [3] ha aplicado las redes bidireccionales a la predicción de estructuras secundarias de proteínas y la generación de descriptores que apoyen la predicción de la estructura tridimensional de éstas.

I.3. Herramientas de Software Existentes para Redes Neuronales Artificiales.

En la actualidad las redes neuronales cuentan con un amplio repertorio de herramientas de software. La mayoría de ellas poseen aspectos comunes: brindan un sistema de interfaces basadas en ventanas gráficas o línea de comandos (en las versiones más antiguas) que posibilitan la introducción por parte del usuario de los datos y las instrucciones referidas al problema en cuestión; módulos de algoritmos que agrupan las más diversas técnicas de aprendizaje aplicable a los datos introducidos;

módulos de preprocesamiento de la información de entrada y postprocesamiento para el análisis de los resultados; etc.

NEURONAL NETWORKS MATLAB TOOLBOX [39] es una poderosa colección de funciones Matlab para el diseño, entrenamiento, y simulación de redes neuronales. Ésta soporta un amplio espectro de arquitecturas de red y técnicas de entrenamiento, entre las que se incluyen: *Backpropagation* (en varias variantes, gradiente conjugado, RPROP, BFGS, etc.), *Radial Basis Networks, Learning Vector Quantization, Recurrent Networks (Elman, Hopfield)*, etc. El toolbox es distribuido con un conjunto de ficheros escritos en el lenguaje Matlab, de manera que los usuarios pueden estudiar el código y modificarlo a conveniencia. Existen paquetes (*Toolbox*) adicionales creados por terceros como el NetLab [48] y el BNN [8] que implementan otras técnicas o mejoran las existentes.

NEUROSOLUTION (NS) [30] es el software comercial más popular dedicado específicamente a resolver problemas usando el paradigma neuronal. NS combina una interfaz de diseño de red basado en iconos con la implementación de las técnicas de entrenamiento, todo ello de manera modular. Es capaz de generar código fuente C++ que contiene los modelos entrenados. Posee tres niveles para la interacción con el usuario: el Educador (Educator), concebido para el adiestramiento de los no especialistas que desean aplicar las técnicas de redes neuronales a la solución de sus problemas; el nivel de usuario (User), el cual extiende el nivel Educador con una variedad de modelos neuronales para aplicaciones de reconocimiento de patrones estáticos; y el nivel Consultante (Consultant), que ofrece modelos mejorados para el reconocimiento de patrones dinámicos, predicción en series de tiempo y control de procesos. Existen dos productos adicionales relacionados con el NeuroSolution: NS-Excel, paquete diseñado para integrarse al programa Excel de Window y facilitar el proceso de obtención de los datos; el NS-Matlab Toolbox escrito para integrarse con Matlab y aprovechar sus características internas. NS implementa *Backpropagation Through Time* y *Real Time Recurrent Learning* como parte de los algoritmos de entrenamiento para redes dinámicas. NS está soportado sobre Windows y Mac OS 9/ Os X.

STUTTGART NEURAL NETWORK SIMULATOR (NSSN) [27] es un software elaborado por el Institute of Parallel and Distributed High-Performance Systems (IPVR) en la Universidad de Stuttgart para el procesamiento de redes neuronales con propósitos investigativos y de aplicación. El software está soportado solo en plataformas Unix (SunOs, Linux, HP-UX) y está formado básicamente por dos módulos: el kernel simulador y el sistema de interfaz gráfica basada en la tecnología X11R4 (R5). NSSN está escrito para correr como un solo proceso sobre las plataformas mencionadas, aunque existe una versión con propósitos académicos (escrita para la arquitectura de hardware SGI Indigo 2) que se ha recompilado para el procesamiento paralelo. Entre los algoritmos que se implementan se encuentran el *Backpropagation, Quickprop, Rprop, Kohonen Maps, Backpropagation Through Time, Generalized Radial Basis Functions, Cascade Correlations, etc*.

BASIC OBJECT ORIENTED NEURONAL ENGINE (BOONE) [40] es un ejemplo de biblioteca de clases creada sobre el ambiente de corrida de Java para brindar un ambiente de programación neuronal a los desarrolladores de aplicaciones que usan la tecnología de Sun. Consiste en una jerarquía de clases para manipular los modelos de red y los algoritmos de entrenamiento, entre los que se encuentran el *Backpropagation*, *Rprop* y las *redes Hopfield.* Boone es un proyecto OpenSource.

FAST ARTIFICIAL NEURONAL NETWORK (FANN) [15] es otro proyecto OpenSource que consiste en una biblioteca de funciones escritas para programadores Ansi C y con un nivel de eficiencia considerable en la implementación del Backpropagation.

WAIKATO ENVIROMENT FOR KNOWLEDGE ANALYSIS (WEKA) [71] es un entorno de software para la experimentación y prueba de diferentes algoritmos de *Machine Learning*. Es un proyecto libre (escrito hasta su versión 3-3-4) auspiciado por la Universidad de Waikato en Nueva Zelanda y constituye un conjunto de paquetes de clases escritos en Java y probados bajo Windows, Linux y Macintosh. Incorpora un amplio conjunto de algoritmos para clasificación entre los que se encuentra el *Multi Layer Perceptron* y *Radial Basis Function Networks*. Adicionalmente posee paquetes de filtrado para el preprocesamiento de los datos y el postprocesamiento de los resultados. Posee un sistema de interfaces gráficas que permiten la exploración de los

datos y la experimentación entre los diversos algoritmos implementados. La paquetería puede ser empleada por programadores java y posee una bien organizada estructura de manera que la extensión de los algoritmos existentes puede realizarse de manera natural redefiniendo una serie de clases establecidas al efecto.

PARALLEL DISTRIBUTED PROCESS FOR C++ (PDP++) [44] es un software libre actualmente auspiciado por la University Carnige Melloon (UCM) y la University Colorado Bould (UCB) con el propósito de poner en práctica las investigaciones de Rumelhart y McClelland publicadas [51]. Es altamente recomendado en entornos docentes y posee suficiente potencia para su uso en investigaciones. Existen versiones para Windows basado en CygWin desde el release 2.0. Incorpora interfaces gráficas para el diseño de las redes y la visualización del proceso de aprendizaje. Implementa algoritmos de entrenamiento *FeedForward* y *Recurrent error backpropagation, Self-Organizing Learning, Hebian Learning,* etc. El último release es el 3.1 donde se incorpora la posibilidad de procesamiento paralelo mediante memoria distribuida (usando MPI sobre un cluster de computadores) para la corrida de los algoritmos.

Aunque la mayoría de los softwares que se han examinado exportan una biblioteca de clases bien estructurada, en todos los casos la extensión de los modelos de red y las técnicas de aprendizaje existentes hacia nuevas arquitecturas y algoritmos se realiza a través de la recompilación del software original; por lo que la incorporación de los nuevos módulos, requiere dominio pleno de las tecnologías de programación que han sido empleadas para la construcción del software. Atendiendo a que ningún software desarrolla la totalidad de los métodos existentes, los especialistas en Redes Neuronales están obligados a poseer un conocimiento minucioso de una gran diversidad de tecnologías de programación en aras de poseer capacidad de extensión y experimentación con nuevos métodos no contemplados en las implementaciones. En este sentido algunas soluciones de tipo tecnológicas se han adoptado, sobre todo orientadas a la simplificación de los lenguajes de programación mediante técnicas de *scripting,* y la complejización de los diseños de software utilizando las técnicas de abstracción soportadas en la metodología orientada a objetos. A pesar de esto, en la actualidad no existe un sistema suficientemente homogéneo y flexible que presente a la vez un esquema simplificado de interacción con el usuario no especialista en técnicas

de programación. Otros enfoques más teóricos se desarrollan para lograr esquemas unificadores de las arquitecturas y técnicas de entrenamiento de redes neuronales existentes que sirvan de marco de trabajo para la concepción de ambientes con las características antes mencionadas [57]. En el epígrafe siguiente se analiza uno de estos enfoques y se describen sus conceptos principales.

I.4. Un Framework Unificador para Redes Neuronales

En la actualidad existen diversos modelos de redes neuronales, muchos de ellos pueden ser derivados unos respecto de otros. Anteriormente mencionamos la necesidad de lograr un sistema de conceptos generales (a manera de framework unificador) que recoja las características esenciales de la mayoría de los modelos: las arquitecturas y los algoritmos de entrenamiento. En la presente sección mencionamos algunos de los trabajos realizados en tal sentido y adoptamos un enfoque unificador particular.

I.4.1. Dos Enfoques Unificadores

En la literatura encontramos dos enfoques unificadores fundamentales: el *enfoque topológico* y el *enfoque de diagrama de bloques.* [57]. El primer enfoque centra su atención en considerar las redes neuronales como un todo, estableciendo modelos canónicos basados en las similitudes funcionales entre los más diversos esquemas. En este sentido Tsoi [66] presenta un enfoque unificador de las redes con arquitectura *feed-forward* y arquitectura *recurrente,* materializado en un sistema de notación matricial para deducir los algoritmos de entrenamiento basados en el método de gradiente descendiente. Atiya en [1] desarrolla también, usando nociones de optimización, un esquema de derivación de la mayoría de los algoritmos de entrenamiento en solo cinco formas diferentes de resolver una ecuación matricial.

El segundo enfoque parece ser el mayormente descrito en la literatura y a diferencia del enfoque anterior, considera la red neuronal como un entramado de bloques constructivos o subsistemas que conectados apropiadamente brindan la posibilidad de obtener las más diversas arquitecturas. La cuestión se centra entonces en la determinación de un conjunto finito de tipos distintos de bloques de interconexión que recojan la mayoría de las características presentes en los modelos. En este sentido se

destacan, entre otros, los trabajos de Lucas [37] y Wan y Beaufays [68], el primero aporta la concepción de bloques generales con dos sentidos de cómputo: *forward y backward*; particularmente útil para modelar el funcionamiento y el cálculo del error en el proceso de entrenamiento de una red tipo *backpropagation* y sus similares; el segundo en lugar de esto sugiere un enfoque basado en la *teoría de flujos en grafos* para manipular diagramas de bloque que representen redes neuronales.

I.4.2. Una Máquina Virtual Neuronal basada en un Framework Unificador.

El enfoque adoptado en el presente trabajo es el descrito por Sona [57]. Éste se basa en diagramas de bloque, definiendo cada bloque como una entidad generalizadora capaz de comportarse autónomamente como una máquina de transición de estados, que calcula una salida en dependencia del estado actual y las entradas brindadas. En este esquema una red neuronal no es más que la actuación colegiada de un conjunto adecuadamente conectado de dichos bloques o *unidades computacionales*. Una máquina virtual[5] que opere con dicho modelo solo debe controlar qué unidad computacional se ejecutará cada vez y cuáles son las funciones de transición adecuadas al modelo particular de red que se pretenda simular. Similarmente a los modelos antes mencionados, cada bloque define dos direcciones de operación: *forward y backward*.

Formalmente una *unidad computacional* puede expresarse en forma de una máquina de Mealy estándar como sigue:

$$x_f(t_f+1)=g_f(x_f(t_f),u(t_f)) \qquad \textit{sección forward de la unidad} \quad \text{(I.16)}$$

$$y_f(t_f+1)=f_f(x_f(t_f),u(t_f))$$

$$x_b(t_b+1)=g_b(x_b(t_b),u(t_b)) \qquad \textit{sección backward de la unidad} \quad \text{(I.17)}$$

$$y_b(t_b+1)=f_b(x_b(t_b),u(t_b))$$

[5] En su tesis Sona defiende la idea de construir una plataforma unificada para redes neuronales usando la arquitectura de una máquina virtual (MV), que sea capaz de operar con los conceptos de las redes neuronales a través de la interpretación de un conjunto de instrucciones especialmente concebidas para ello. La motivación fundamental reside en la posibilidad que brinda el enfoque de MV en el desarrollo de sistemas homogéneos y con altos niveles de abstracción. En el próximo capítulo, se propone básicamente una arquitectura de software que siga estas ideas.

Donde $u(t_f)$ representa la entrada *forward* en el momento t_f, $y_f(t_f+1)$ representa el valor de salida de la unidad en el momento t_f+1 y $x_f(t_f)$ el estado *forward* de la unidad en el momento t_f, g y f representan la función de transición de estados y de salida respectivamente. Lo mismo para la sección backward.

Las dos secciones de las unidades computacionales no están necesariamente sincronizadas. Las funciones g y f representan *operadores* y constituyen elementos programables para la expresión de las características de los diversos modelos de red.

Se adopta el término *operador* para designar los conceptos que sirven de elementos de abstracción a la mayoría de los modelos de red existentes y que constituyen las entidades que aportan la diversidad de posibilidades en la construcción de nuevos modelos. Se consideran operadores aquellas variables encaminadas a la especificación del modelo de las unidades computacionales (especificación de las funciones f y g vistas anteriormente), las empleadas en la descripción de aspectos particulares de los algoritmos de aprendizaje como la *planificación de la ejecución (scheduling)* y *condiciones de parada (Stop Conditions)*, entre otros.

La especificación (*redefinición*) en forma de un lenguaje (o la adopción de valores por defecto) de un conjunto de operadores determinado, para un conjunto determinado de unidades computacionales y/o un modelo de algoritmo de aprendizaje, constituye el *programa* de entrada a la Máquina Virtual Neuronal (MVN). Dicho programa es empleado por la MVN como argumento para refinar su *Mecanismo Básico de Procesamiento (*MBP*)*.

El MBP de la Máquina Virtual constituye la guía central de su trabajo, el elemento principal de abstracción sobre el cual se aplica la redefinición de operadores. Actúa como un esqueleto de todos los modelos de red expresables. Claramente, el MBP es un concepto abstracto con el que opera la MVN, su expresión concreta al que denominamos *entorno de ejecución*, solo se obtiene de la instanciación de los operadores abstractos establecidos en el MBP a partir de los elementos definidos en un *programa.*

Un Mecanismo Básico de Procesamiento debe recoger, de manera general, las etapas esenciales que involucra la manipulación de redes neuronales (en todos los casos nos

referimos a redes con aprendizaje supervisado), a saber: *creación, entrenamiento* y *explotación*.

La creación de una red neuronal, a la luz del enfoque tratado, puede considerarse como un típico proceso de construcción de un grafo orientado: cada nodo representa una unidad computacional y cada arco reconstruye la relación de precedencia entre unidades (esquema de conexión). En conjunto esto constituye la topología de la red. El entrenamiento y explotación de una red neuronal puede describirse como un único *proceso de ejecución* que garantiza un medio homogéneo para la especificación de los procedimientos de trabajo con la red.

Un mecanismo de procesamiento resulta entonces la conjunción de dos procesos generales: el *proceso de creación o manipulación (diseño)* de la topología de la red y el *proceso de ejecución.* En general estos dos procesos guardan una relación secuencial de acuerdo a las etapas mencionadas; sin embargo, para un tipo particular de modelos de red dinámicos, conocidos en la literatura como *modelos constructivos* [57], [22], [16], esta relación puede verse modificada en una suerte de combinación de ambos procesos, en el que el diseño de la topología de la red constituye una parte esencial del proceso de ejecución (el número de neuronas y su interconexión constituyen parámetros ajustables en el proceso de entrenamiento). De esto se deriva entonces la adopción de un esquema concurrente de cooperación de dos *procesos* como idea central de nuestro esquema.

En la figura 1.10 se muestra una posible forma en que puede ser organizada, siguiendo nuestra discusión, una máquina virtual para redes neuronales. En relación con lo descrito anteriormente, se aprecia que el núcleo de la MVN está compuesto de una tarea o proceso de *administración de objetos*, vinculado con las actividades relativas al diseño de topologías y la administración de *operadores; y* una tarea o proceso de *ejecución*, encargado de dirigir el proceso de entrenamiento y explotación de la red creada bajo el empleo de los *operadores* redefinidos y almacenados en el *Repositorio de Objetos y Modelos Neuronales (Object Repository).*

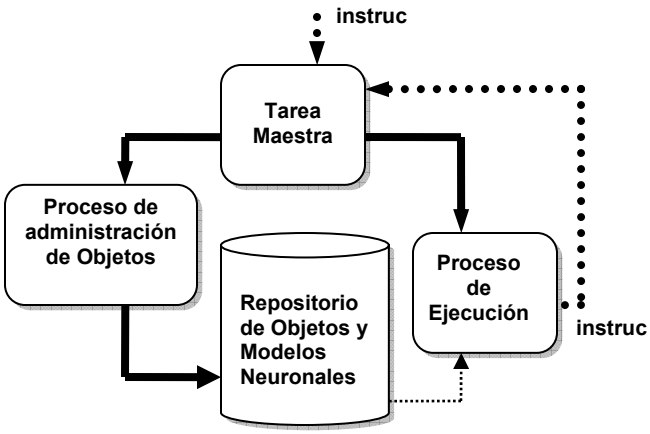


Figura 1.10. Organización de una Máquina Virtual para Redes Neuronales.

Una tarea maestra controla la actividad de los dos procesos concurrentes. Las instrucciones recibidas por la MV están entonces encaminadas a la especificación de nuevos modelos de red (creación de topologías y algoritmos de entrenamiento) y al control de su ejecución. La saeta de puntos que parte de la tarea de ejecución hacia la tarea maestra está especialmente concebida para permitir entornos de ejecución donde los conceptos propios de la topología y el algoritmo de aprendizaje puedan ser redefinidos dinámicamente. Ver más detalles sobre el *Procesos de Ejecución* y el *Proceso de Administración de Objetos* en el Anexo 1.

I.4.3. Funcionamiento General de la Máquina Virtual.

Las ideas presentadas anteriormente pueden materializarse, como hemos indicado, sobre una arquitectura de máquina virtual (MV) capaz de interpretar un repertorio de instrucciones representativas de la variedad de conceptos que pueden manejarse en torno a las redes neuronales. Dicho conjunto de instrucciones puede dividirse en dos grupos: un conjunto destinado a la definición de nuevos objetos, que reflejen los conceptos involucrados en la creación de nuevos modelos o la conjugación de los existentes; y otro conjunto de control que garantice la coordinación de las acciones dentro de la MV.

Esta división sugiere además un diseño que contemple por un lado un módulo de almacenamiento y de gestión de datos estructurados a partir de una jerarquía (*Object Administration Module,*) y por otro, un módulo que permita la coordinación y control de

la ejecución de los *programas* destinados a la MV (*Execution Module*). Esta idea puede estructurarse a partir de la concepción de dos tareas, una encargada de la interpretación o ejecución de las instrucciones de gestión de datos (definición de nuevas clases, creación de objetos, eliminación) y otra encargada de la administración de los programas (enlace de operadores abstractos, gestión de esquemas de ejecución abstractos, etc.) y su ejecución.

La comunicación del usuario con la MV puede establecerse mediante una interfaz gráfica con los elementos necesarios, o mediante el empleo de un *lenguaje de especificación* (lenguaje de alto nivel) que incorpore las estructuras adecuadas. Un lenguaje de especificación debe incorporar estructuras para la definición de nuevos tipos o clases a partir de un conjunto de tipos primitivos; la creación, modificación, eliminación de objetos asociados a clases, estructuras de control de flujo y almacenamiento de información primitiva (cadenas de caracteres, valores ordinales, de coma flotante, etc.) destinada a la definición de nuevos objetos operadores. La MV puede prepararse para interpretar directamente este lenguaje de especificación o una forma de código intermedio generado por un compilador creado al efecto. El objetivo esencial de dicho lenguaje consiste en permitir, de manera flexible, la especificación de nuevos modelos de redes neuronales de las más diversas características, y el empleo de los modelos existentes; ya sea para su explotación en forma simple o para su uso combinado.

En la figura 1.11 se muestra un posible flujo de trabajo de la máquina virtual neuronal. Básicamente, las instrucciones del *lenguaje de especificación* que conforman el programa brindado a la MV son interpretadas, y los objetos (unidades de cómputo, conexiones, operadores, etc.) que ellas definen son incorporados al repositorio de objetos; las instrucciones dedicadas al control de la ejecución (*exec train <Model1>* en la figura) activan un mecanismo (*linker*) encargado de refinar el *mecanismo abstracto de ejecución* usando la información definida previamente y contenida en el repositorio; con esto se obtiene un programa o instancia concreta (un modelo neuronal concreto) del mecanismo abstracto que es capaz de ser ejecutado definitivamente en la máquina virtual.

De esta manera, la MVN es capaz de ejecutar modelos de red (algoritmo de entrenamiento, topologías) diversos de manera flexible e integrada. Las bondades de aplicación de un framework como el descrito en esta sección se ven potenciadas por las amplias posibilidades que brinda en la creación de herramientas muy versátiles, capaces de generar y emplear en su trabajo grandes bibliotecas de modelos de redes neuronales. Esta oportunidad facilita ampliamente el proceso de producción y ensayo de mejores soluciones a problemas reales, y a la vez constituye una arquitectura potencialmente útil en la integración de la *computación neuronal* con otros paradigmas de la inteligencia artificial.

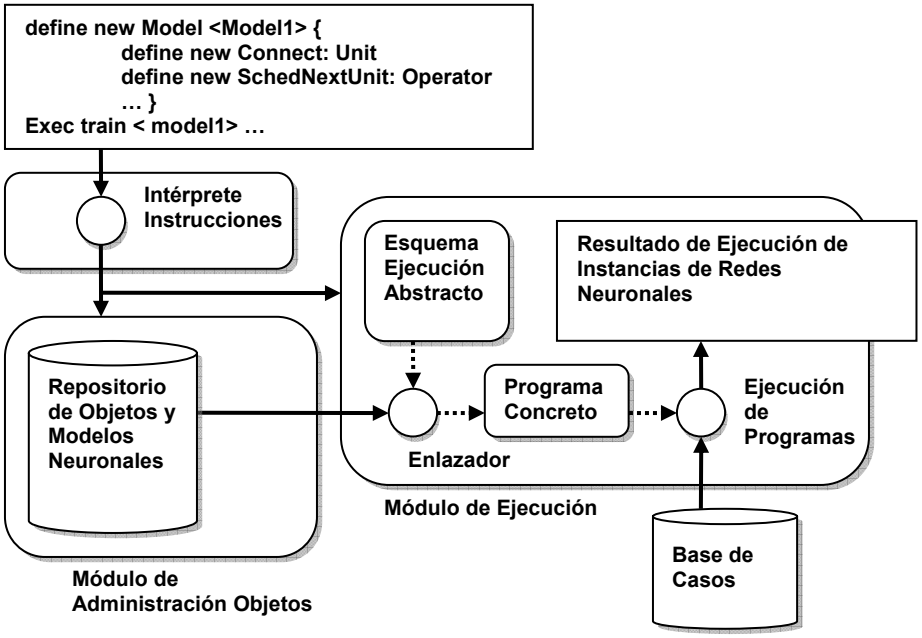


Figura 1.11. Esquema de Trabajo de la Máquina Virtual Neuronal.

I.5. Conclusiones del Capítulo.

En el presente capítulo hemos abordado los conceptos generales del paradigma neuronal, en particular se aportan elementos de las redes recurrentes. Se presentan los algoritmos de entrenamiento *backpropagation* (BP) para redes feed-forward y su extensión para las redes recurrentes, el *Backpropagation Through Time* (BPTT). Se enuncian algunas heurísticas que mejoran la clasificación en redes BP, así como técnicas alternativas para lograr mejoras en la velocidad de convergencia. Se enuncia un grupo de las aplicaciones más recientes de las RR, entre las que se encuentran el

procesamiento de señales, análisis de lenguaje natural, modelación de sistemas dinámicos y de control, procesamiento de imágenes, robótica, etc. Se considera oportuno seguir profundizando en el estudio de tales modelos con el objetivo de encontrar posibles aplicaciones a los problemas que enfrentan hoy los seminarios de Inteligencia Artificial y Bioinformática. Se resumen un conjunto de herramientas aportando información de solo algunas y se observa en todos los casos la carencia de un medio suficientemente homogéneo y flexible que presente a la vez un esquema simplificado de interacción con los usuarios no especialistas en técnicas de programación. Se exploran algunas alternativas de framework unificador para redes neuronales y se adopta uno basado en diagrama de bloques. Se discute la organización general de una máquina virtual basada en el framework adoptado y se aportan elementos para la selección de una posible arquitectura de software. En el capítulo siguiente se selecciona la arquitectura más apropiada para una plataforma que incorpore las ideas de framework unificador discutidas hasta aquí; y se realiza el análisis, diseño e implementación de una herramienta basada en tal arquitectura que implemente los algoritmos de entrenamiento BP y BPTT.

Capítulo II. Diseño e Implementación de una Herramienta para Redes Recurrentes.

En el presente capítulo se muestra una arquitectura para la plataforma examinada en el capítulo anterior, y se implementa una herramienta para redes recurrentes que sigue esta arquitectura. Se discute una estructura de clases sobre la que se implementa la herramienta y que obedece al enfoque basado en procesos concurrentes.

II.1 Arquitectura de una Máquina Virtual para el Framework Unificador.

Desde el capítulo anterior se viene manejando la idea de emplear una máquina virtual para implementar el Framework Unificador descrito. En el presente capítulo pretendemos mostrar una arquitectura para dicha máquina virtual. La arquitectura en sentido general puede estructurarse mediante un núcleo (*back-end*) de procesos concurrentes capaces de responder a tareas específicas dentro de la máquina virtual y una interfaz de usuario que sea capaz de adaptar la funcionalidad del núcleo a los diferentes contextos de aplicación (interfaz gráfica para usuarios, protocolo de comunicación en ambientes distribuidos, etc.).

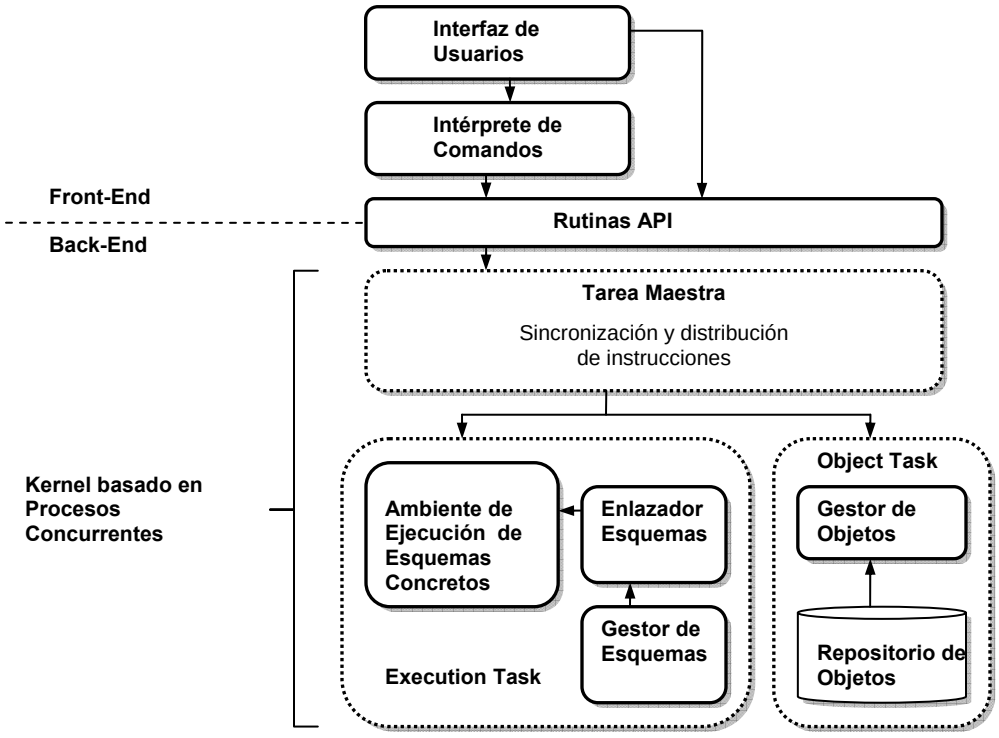


Figura 2.1 Arquitectura de software de una Máquina Virtual para Redes Neuronales.

La arquitectura específica planteada en la figura anterior obedece a un modelo más general, a saber, una arquitectura de procesos concurrentes interactuando entre sí, controlados por un proceso principal (proceso coordinador), y actuando interactivamente en pos de la solución de un problema determinado; o sea, desde un punto de vista conceptual en lo que se refiere a modelos de desarrollo de aplicaciones, podemos identificar un nivel superior de abstracción[6].

En la figura 2.2 se aprecia este nivel, que por poseer su propio valor semántico merece ser tomado en cuenta como un objeto de desarrollo independiente y separable. El hecho de que exista este nivel abstracto significa que podemos hacer una separación entre los conceptos propios de la máquina virtual (su organización en *Estructuras para la Administración de Objetos, Gestión de Esquemas Abstractos de Algoritmos, Interpretación de Instrucciones, etc.*) y la forma exacta en que se gestionan las funciones de base de una arquitectura de procesos (tareas) concurrentes (envío y recepción de mensajes entre procesos, sincronización y planificación de los procesos, etc.). Esto deviene fuente de análisis de varias posibles soluciones que pueden ir desde desarrollos propios hasta la asimilación de tecnologías concebidas al efecto.

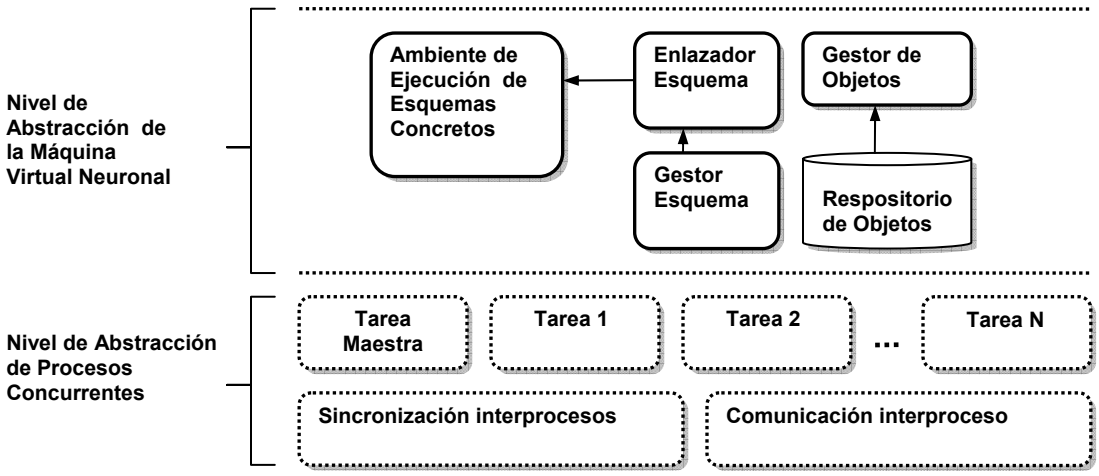


Figura 2.2 Arquitectura de software de una Máquina Virtual para Redes Neuronales.

[6] *Abstracción* se refiere aquí al concepto de capas del software abstractas manejado en el ámbito de la modelación conceptual del software. Ver [38], [72], [52] para más detalle sobre la modelación multicapa de software.

Como se aprecia, diseñar la máquina virtual siguiendo esta arquitectura es fácilmente adaptable a entornos distribuidos y muestra ventajas como las siguientes:

- Extensible mediante la agregación de nuevas tareas a la aplicación o la incorporación de nuevos controladores de mensajes (*messages handlers*) a las tareas existentes.
- Cada tarea puede considerarse un módulo independiente, lo cual aporta un muy bajo nivel de acoplamiento, característica que aporta robustez y flexibilidad al producto. Este aspecto posee vital importancia no solo en la adaptación que muestra a sistemas abiertos, sino en la posibilidad que brinda en la paralelización misma del proceso de ejecución interno de la máquina virtual neuronal; pues un módulo de la aplicación podría comunicarse con módulos de otra máquina virtual exactamente de la misma forma en que lo hace con sus módulos vecinos.
- Aporta rapidez en el desarrollo de aplicaciones extensibles, pues el programador no tiene que preocuparse de los aspectos globales de interrelación y soporte, sino que se concentra más en los detalles de los requerimientos que quiere satisfacer con la aplicación.
- Pueden desarrollarse tareas en contextos paralelos, lo que aumenta la eficiencia del sistema, sobre todo en computadoras con soporte de procesamiento paralelo.

Hasta aquí se ha propuesto una variante de arquitectura general para la máquina virtual. Dicha arquitectura constituye la idea central para el desarrollo de una herramienta con características extensibles, y pensamos pueda ser usada como guía del desarrollo de un proyecto a más largo plazo para la construcción de una plataforma unificadora. En el siguiente epígrafe se discute el diseño y la implementación de una estructura de clases para el nivel de gestión de procesos concurrentes. El objetivo es que esta estructura sea el punto de partida en la construcción de la herramienta para redes recurrentes; y posteriormente pueda ser empleada para la construcción de una plataforma más amplia.

II.2. Diseño e Implementación de una Estructura de Clases para una Arquitectura Basada en Procesos Concurrentes.

Las clases diseñadas para este nivel de gestión de procesos concurrentes han sido agrupadas en dos paquetes fundamentales:

Comunication-Infrastructure: comprende las clases involucradas en la gestión de mensajes, búferes para el almacenamiento de la información, canales de comunicación entre procesos, etc. (Ver Anexo 4.1)

Schedule-Infrastructure: agrupa las clases para la sincronización y la planificación de los procesos, así como la estructura general de una tarea y su información asociada. (Ver Anexo 4.2).

A continuación se describen las clases fundamentales del paquete *Comunication-Infrastructure*.

`tStream` es una clase que abstrae el concepto de memoria lineal (secuencia de bytes); está concebida para servir de estructura de almacenamiento para la información en formato binario que sea manejada por los procesos. Entre los métodos principales se encuentran: `SetValueAddress()` para establecer un valor en una dirección determinada de la secuencia de bytes, `GetValueAddress()` para recuperar valores, `Add()` para adicionar valores al final de la secuencia, `Copy()` para copiar subsecuencias binarias de otra secuencia. Ésta es una clase generalizada que posee dos especializaciones `cBuffer` y `lBuffer`.

`tCBuffer` implementa el concepto de búfer circular y es empleada como estructura para depositar los mensajes recibidos por un proceso. Posee dos métodos adicionales: `isEmpty()` que devuelve si el búfer está vacío; y `nextValue()` que permite recuperar el próximo valor en el búfer. `tLBuffer` implementa la versión lineal del trabajo con la secuencias de bytes.

`tMessage` es una clase de utilidad que representa la estructura de un mensaje. Está formada por dos atributos principales: `header`, que representa la información de encabezado del mensaje y `body` que representa el contenido. Cada mensaje está asociado a una *clase de mensaje* que determina la información y la organización de

esta información dentro del cuerpo del mensaje (cada mensaje puede ser visto como un objeto que pertenece a una clase determinada). El encabezado posee 4 atributos: `msgID` y `classID` brindan el identificador del mensaje y la clase a la que pertenece respectivamente; `srcTask` y `trgTask` ofrecen los identificadores del proceso que originó el mensaje y el proceso que debe recepcionarlo. `tMessage` es un *wrapper*[7] de la funcionalidad que involucra la manipulación de mensajes. Posee dos métodos básicos: GetPropertyValue() y SetPropertyValue() que se encargan de obtener el valor de un campo o propiedad contenido en el cuerpo del mensaje, y el establecimiento de su valor.

En su forma más común un mensaje está contenido en un búfer como una secuencia de bytes, siguiendo un formato binario determinado. Esta manera de presentación de un mensaje es la empleada por los procesos para intercambiar mensajes entre ellos.

En la siguiente figura se muestra el formato binario propuesto para los mensajes. En este formato se distinguen claramente el encabezado formado por los cuatro atributos antes mencionados y el cuerpo del mensaje que es una secuencia de campos (*Fields*) pertenecientes cada uno a una clase determinada.

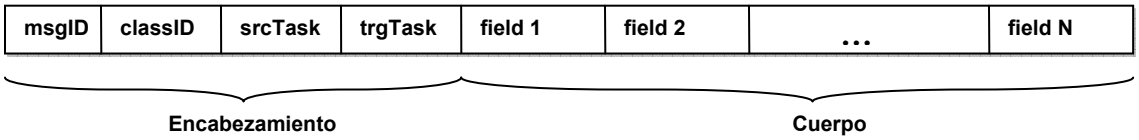


Figura 2.3 Formato binario de un mensaje.

Visto así, un mensaje es un objeto[8] formado por la agregación de otros objetos de distintos tipos o clases. `tClass` es una *metaclase*[9] concebida para contener la información de cada *clase de mensaje* que incorpore el sistema. Posee cuatro atributos fundamentales: `name`, que identifica el nombre de la clase como una cadena de caracteres; `classID`, un entero identificador de la clase en el conjunto; `size`, un

[7] Una clase cuya función es envolver el comportamiento de una estructura compleja de información.
[8] Aquí no nos referimos a objetos primarios (objetos manipulados por el lenguaje de programación) sino como un concepto empleado para denominar la agrupación de los datos que caracteriza a un mensaje.

[9] Una clase primaria (de programación) que establece la estructura de información para manipular las *clases de mensajes* (creadas dinámicamente) posibles en la aplicación.

campo calculado para determinar el tamaño total en bytes que ocupan de memoria los objetos creados a partir de dicha clase; y `collection`, un campo para caracterizar una clase como una colección de datos de otra clase. La información de los campos de una clase es registrada como una lista de elementos de tipo `tField`. Ésta es una clase pasiva que almacena el nombre del campo y el tipo de clase de mensaje al cual pertenece. Cada elemento `tField` controla además el *desplazamiento* o la dirección exacta donde comienza el campo dentro del formato binario en el que se expresan los objetos mensaje. Esta clase juega un papel esencial en la manipulación de los objetos mensaje, pues es la que controla los detalles de la estructura de cada uno de éstos. Se conciben 4 tipos predefinidos de clases de mensajes, éstas corresponden a algunos de los tipos de datos primarios de cualquier lenguaje de programación: byte, int32, int64, real y son usadas como los constructores básicos para la definición de nuevas clases de mensajes. Las clases caracterizadas como colecciones poseen un solo campo, correspondiente a la clase de los elementos que almacenará la colección. El campo `size` esta vez se interpreta como el tamaño en bytes de un elemento en la colección. Igualmente los mensajes de tipo colección poseen un formato binario especial, donde los primero 4 bytes del cuerpo indican la cantidad de elementos contenidos en la colección y los restantes la secuencia de los objetos contenidos. Los objetos (mensajes) de tipo colección no pueden ser agregados junto a otros campos en una clase.

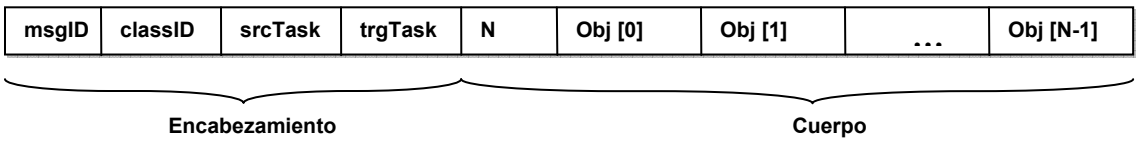


Figura 2.4 Formato binario de un mensaje tipo colección.

`tMsgFactory` obedece al patrón *Factory*[10] y constituye la clase principal encargada de manipular los mensajes y las clases (agregar nuevas clases, construir mensajes de una clase determinada, obtener y establecer los valores de los campos de un mensaje, etc.). Esta clase constituye un repositorio de clases y una fábrica para construir nuevos mensajes a partir de las clases que contiene; se auxilia de `tClass` y `tMessage` para

[10] Se refiere al patrón de diseño concebido como una fábrica de objetos de un tipo determinado.

realizar su trabajo. El método `AddClass()` es el dispuesto para adicionar nuevas clases, junto a `AddFieldToClass()` para fijar la información de los campos. `BuildMessage()` crea un nuevo mensaje a partir de la información suministrada para el encabezado. Cada mensaje es generado con un identificador único. `tMsgFactory` es capaz de generar el formato de un mensaje sobre una secuencia de bytes (`tStream`) usando el formato binario adecuado; y manipular la información de los mensajes sobre dicha secuencia. `GetPropertyValue()` y `SetPropertyValue()` permiten recuperar y actualizar la información de los mensajes. Esta clase es un controlador del trabajo con mensajes y es la abstracción fundamentalmente empleada para la comunicación interproceso.

`tChannel` y `tChannelLinker` son un par de clases que trabajan relacionadas para abstraer el concepto de *canal de información*. Un canal de información es un *pipeline* empleado para la comunicación entre los procesos. La información transmitida por el canal es almacenada en un búfer circular que posee protección para el acceso concurrente y una señal para indicar la existencia de información en el búfer. Los métodos `put()` y `get()` son brindados para la incorporación y extracción de datos del canal. La información es intercambiada hacia o desde secuencias de bytes (`tStream`). El método `getMsg()` está especialmente concebido para la extracción de mensajes. La forma de llamar estos métodos puede ser modificada según el contexto para bloquear o no, el proceso solicitante si el búfer está vacío o para desestimar mensajes, si éstos son demasiado grandes o se encuentran corruptos.

`tChannelLinker` es la encargada de controlar (concentrar) los canales de información disponibles. Posee un conjunto de elementos de tipo `tChannel` y es la abstracción empleada como interfaz para la publicación y recepción de mensajes entre los procesos. Puede adicionarse un nuevo canal usando el método `AddChannel();` nuevamente aquí existen métodos `get` y `put` para el trasiego de los mensajes.

En general el paquete *Infrastructure-Comunication* brinda las estructuras necesarias para garantizar la comunicación entre procesos y posee un mecanismo flexible para la manipulación de las clases de los mensajes de forma dinámica; esto garantiza que la información intercambiada posea una estructura compleja haciendo homogéneo el

proceso de comunicación. Lo planteado puede mejorarse incorporando relaciones de herencia entre las clases de los mensajes y la agregación de mensajes de tipo colección.

El paquete *Schedule-Infrastructure* está organizado en dos grupos de clases fundamentales: el grupo de las clases que abstraen el concepto de *proceso*, y el grupo de las que soportan la lógica para la sincronización de la ejecución de éstos.

La clase fundamental del primer grupo es `tTask`. Esta clase debe encapsular todos los aspectos concernientes a la ejecución de una tarea (o proceso) en un contexto independiente (un hilo de ejecución).

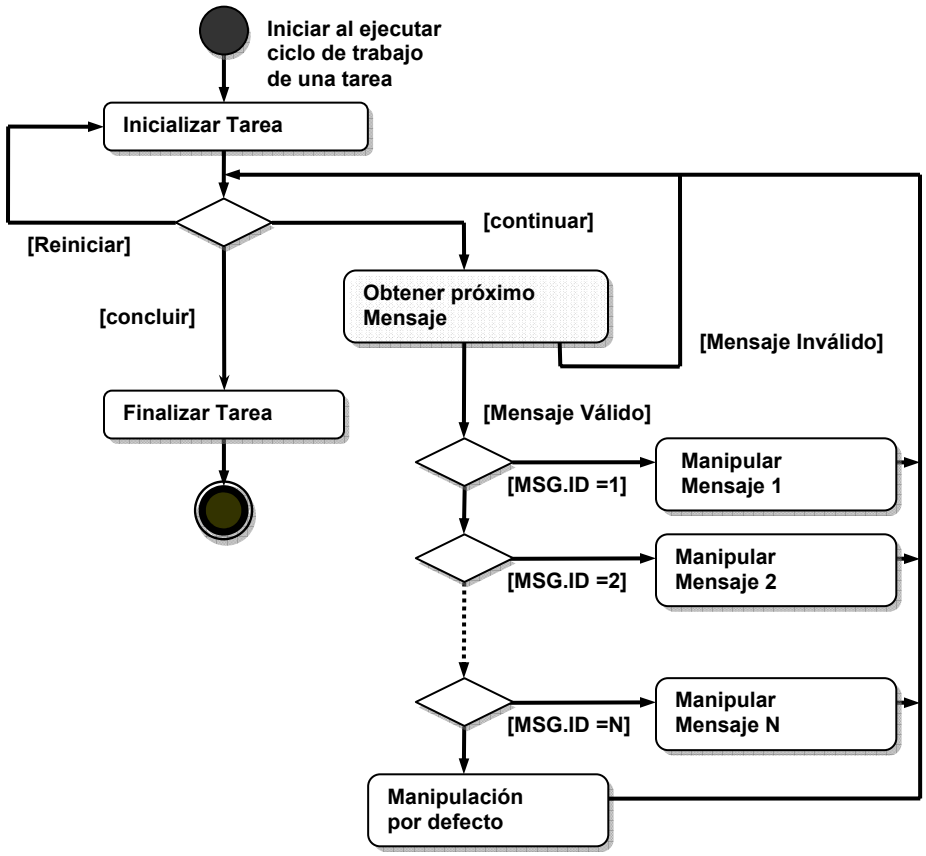


Figura 2.5 Diagrama del flujo de actividades de una *tarea* genérica.

`tTask` posee un canal de comunicación dedicado `msgChannelID` (índice de un canal en `tChannelLinker`) a la recepción de los mensajes que le son enviados; y una secuencia de bytes `memory` para la manipulación de los mensajes en formato binario. El método `execute` encapsula el modelo de ejecución, éste sigue un flujo como el

representado en el diagrama de actividad de la figura 2.5. La primera actividad es la inicialización de la tarea (se realiza por parte del método `task_initialization`); y a continuación se ejecuta un ciclo de lectura y manipulación de cada mensaje que llega al búfer mediante el método. Una vez obtenido el mensaje con el método `getNextMessage`, éste es identificado por su clase y se ejecuta la rutina manipuladora (*message handler*) pertinente. Una vez que se detecta la condición de salida, se realiza la actividad de finalización contenida en el método `task_finalization()` y se detiene la ejecución del proceso.

La unidad básica para la gestión de la concurrencia es el mensaje, o sea podemos planificar cuándo se atenderá un mensaje en aras de prevenir la ejecución simultánea de manipuladores excluyentes. El modelo de sincronismo está basado en un sistema de *permisos de ejecución* (*execution tokens*). Un manipulador de mensaje, antes de su ejecución debe solicitar el permiso correspondiente y éste le será otorgado solo cuando se garanticen las condiciones de no concurrencia con mensajes excluyentes. La relación de exclusión se expresa entre pares de mensajes cuyos manipuladores no pueden ejecutarse simultáneamente. Pueden establecerse *clases de permisos* atendiendo a las relaciones de exclusión que se tengan en cada caso. El modelo puede expresarse más formalmente como sigue:

Siendo M el conjunto de mensajes y T el conjunto de procesos, TxM expresa el conjunto de sucesos asociados a la manipulación de un mensaje m_i en un proceso t_j. En el conjunto producto anterior puede definirse una relación R reflexiva y simétrica que expresa las relaciones de exclusión entre los sucesos (m_i, t_j). Si consideramos la función $F(m_i, t_j) = \{ (m_t, t_r) \mid (m_i, t_j) R (m_t, t_r) \}$, la relación $(m_i, t_j) Tk (m_t, t_r)$ definida como $(F(m_i, t_j) = F(m_t, t_r)) \wedge (t_i = t_r)$ es la relación de equivalencia correspondiente a las clases de *permisos de ejecución*; $(m_i, t_j) Tk (m_t, t_r)$ significa que los mensajes m_i y m_t son manipulados por el mismo proceso y poseen las mismas relaciones de exclusión, por lo que deben solicitar el mismo tipo de permiso. Aquellos mensajes m_i cuyo valor $F(m_i, t_j) = \emptyset$ no poseen relaciones de exclusión por lo que no necesitan solicitar un permiso para ejecutar sus manipuladores. La condición $t_i = t_r$ es necesaria

para conservar el orden de ejecución de los mensajes según son asimilados por el proceso maestro.

`teToken` es la clase encargada de abstraer el concepto de permiso. Posee un atributo `state` para definir el estado actual del permiso: activo cuando es solicitado y se ha otorgado e inactivo en el caso de que no se haya otorgado aún. Este atributo es la información primordial para concluir la activación (otorgamiento) de un permiso atendiendo a la inactividad de los permisos excluyentes con él. El método `addeTokenExclu()` posibilita definir las relaciones de exclusión entre permisos.

La clase de programación `teTokenGroup` se encarga de almacenar un conjunto de *clases de permisos de ejecución (execution tokens, eToken)* y controlar su funcionamiento. El método `tAddeToken()` posibilita la introducción de nuevas clases de permisos. `mountEToken()` es el método usado para realizar la solicitud de *permiso*; `getEToken()` se emplea para tomar el *permiso; y* `releaseEToken()` se encarga de liberar el *permiso* una vez que el manipulador de mensajes ha terminado su trabajo. El funcionamiento general para lograr el sincronismo de los procesos concurrentes está basado en la concentración de todos lo mensajes por parte del proceso maestro, quien se encarga de distribuirlos hacia todos los demás procesos, realizando la solicitud (*mount*) de permiso para la manipulación de los mensajes que así lo requieran. Una vez que los mensajes llegan a los procesos correspondientes, sus manipuladores deben tomar (*get*) el permiso antes de la ejecución; esto es lo que garantiza entrar a la rutina con las condiciones de exclusión exigidas. El método `getEToken()` bloquea la ejecución del proceso solicitante y solo le devuelve el control cuando se haya cerciorado de haber resuelto los conflictos de concurrencia.

La regla que establece qué proceso se ejecuta en cada momento obedece al principio de que *de todos los permisos que puedan ser otorgados en cada momento se escoja el que haya sido solicitado primero.* Para controlar esto `teTokenGroup` posee una lista de solicitudes (`tokenRequestList`) no satisfechas. Cada vez que se solicita un permiso y no puede ser satisfecho en ese mismo momento, se genera una entrada en la lista de solicitudes, que será inspeccionada la próxima vez que un manipulador de mensajes libere un permiso otorgado (*release*), y las condiciones de exclusión puedan

presumiblemente ser favorables. En el Anexo 2 se muestra un ejemplo de cómo usar esta clase.

En la figura 2.6 se muestra un diagrama de secuencia que ilustra las relaciones que se establecen entre los procesos y la clase `teTokenGroup` en la sincronización de las actividades.

El siguiente diagrama ilustra la sincronización entre el proceso task1 y task2 cuando se manipulan dos mensajes excluyentes. Nótese que cuando la tarea task1 solicita el permiso, el control no le es devuelto hasta que la tarea 2 no libere el permiso que le ha sido otorgado con anterioridad.

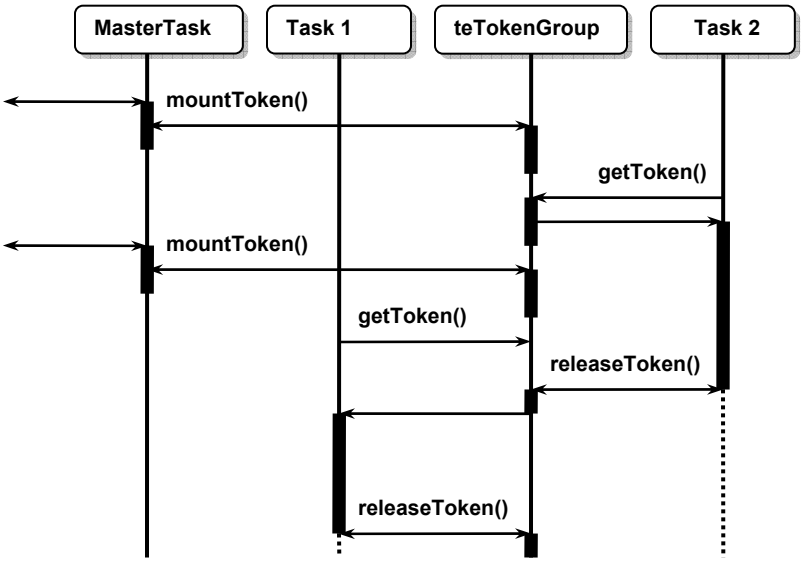


Figura 2.6 Diagrama de secuencia que ilustra la sincronización de procesos.

En la figura 2.7 se muestra un diagrama de estado de la clase `tTask`. Cada proceso tiene asociado un solo canal de recepción de mensajes, y esto conduce a la necesidad de implementar una lógica especial para atender a los mensajes más prioritarios, como son los mensajes de *reinicialización* y *terminación* del proceso. Nótese que todos los estados donde se realiza actividad, poseen una transición hacia el estado de inicialización. Los procesos comienzan por el estado de *inicialización* y pasan (si la condición de bloqueo está desactivada) al estado de *obteniendo mensaje,* si existe un mensaje en cola esperando a ser atendido se pasa directamente al superestado de *ejecución,* si no se espera a la ocurrencia del evento *arriba mensaje*.

El superestado de *ejecución* posee dos puntos de entrada: el primero por el estado *obteniendo permiso* si se requieren controles de sincronización para la ejecución del manipulador, y el segundo directamente por el estado *ejecutando handler,* en caso de no ser necesaria la sincronización.

Una vez terminado de ejecutar el manipulador, si no se encuentra la condición de concluir se pasa al estado de *obtención de mensaje* para procesar el próximo mensaje. Una vez detectada la condición *concluir* se pasa al estado de inicialización y se finaliza el proceso. El estado *esperando* está concebido para tener un control global sobre los procesos para promover la resincronización de las tareas una vez que sean reiniciadas.

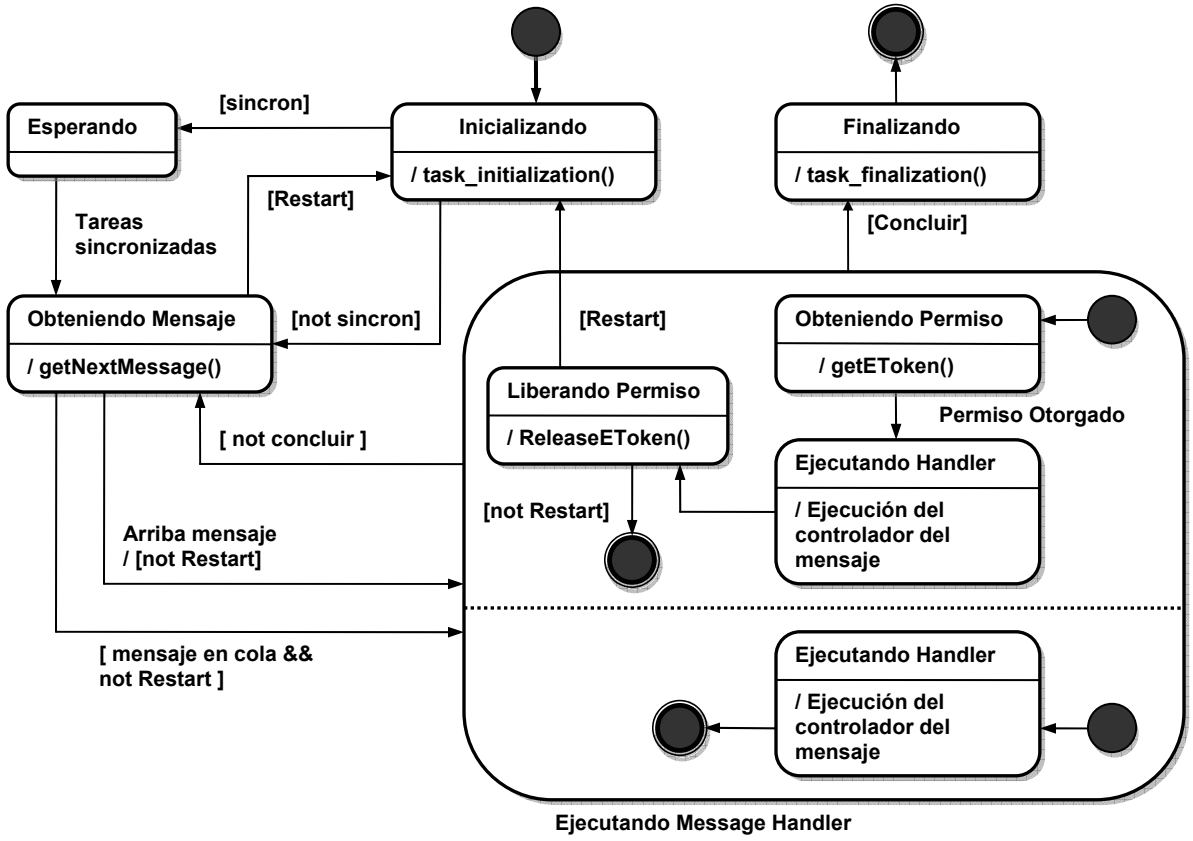


Figura 2.7 Diagrama de estado de la clase `tTask`.

La estructura de clases examinada hasta aquí puede ser reusada directamente para obtener la arquitectura de la figura 2.1. El mecanismo básico para este reuso se basa en la especialización de la clase abstracta `tTask` redefiniendo los métodos `task_initialization`, `task_finalization` y `execute`. La incorporación de nuevas tareas se realiza de acuerdo a la organización que va a tener el sistema

integralmente y obedece a necesidades de robustez, independencia y eficiencia en el intercambio entre procesos. `tTask` puede concretarse en las dos tareas vistas en el epígrafe anterior: *ObjectTask* y *ExecuteTask.* La actividad general del sistema se realiza definiendo los mensajes con los que va a operar cada una de estas tareas, los permisos de ejecución pertinentes y las estructuras de clases para el almacenamiento y gestión de la información discutida en el capítulo anterior.

II.3. El *NEngine*, una Herramienta para el Trabajo con Redes Recurrentes.

NEngine es una herramienta basada en la arquitectura de procesos concurrentes mencionada en el epígrafe anterior; ha sido creada para la construcción, entrenamiento y explotación de redes neuronales. En su primera versión soporta el trabajo con redes recurrentes y feedforware (usando BP y BPTT), aunque puede ser extendida a otros algoritmos si se incorporan las estructuras necesarias para ellos siguiendo el modelo de mensajes. NEngine todavía no constituye plenamente una plataforma para redes neuronales, pues no cuenta con las estructuras para la gestión de objetos y esquemas abstractos de ejecución, sin embargo la arquitectura sobre la que está soportada permite que pueda ser reusada fácilmente con tales propósitos.

Los casos de uso en la herramienta están agrupados por paquetes: el paquete de *Objetos de Topología* agrupa las funcionalidades propias de la creación de topologías de red; el paquete de *Entrenamiento* agrupa las funciones de entrenamiento en dos variantes de estrategia de presentación de los casos de la base de aprendizaje (la secuencial y la uniforme); y el paquete de *Entrada/Salida* contiene las funcionalidades para la entrada y salida de la información hacia y desde ficheros usando un formato de almacenamiento determinado.(Ver Anexo 5.1)

La figura 2.8 muestra la organización de la herramienta. La arquitectura es muy similar a la de la máquina virtual neuronal. Posee dos estratos bien definidos: *Back-End* o substrato de soporte, y el *Front-End* o interfaz de usuario (IGU). El *Back-End* está formado por un conjunto de rutinas API, y el núcleo o kernel que es el que ejecuta los algoritmos de entrenamiento. Las rutinas API exportan la funcionalidad del kernel hacia el exterior y propician la comunicación con la interfaz de usuario. El kernel de la herramienta está formado por tres procesos (especialización de la clase abstracta

tTask) corriendo concurrentemente: t*SysObjectTask, tExecTask* y t*IOTask.* El proceso maestro t*ControlTask* distribuye los mensajes arribados al kernel y controla el sincronismo. Adicionalmente posee el objeto *ChannelLinker* (tChannelLinker) para la administración de los canales de información que emplearán los procesos; y el objeto *Scheduler* (tETokenGroup) para la sincronización de los manipuladores de mensajes.

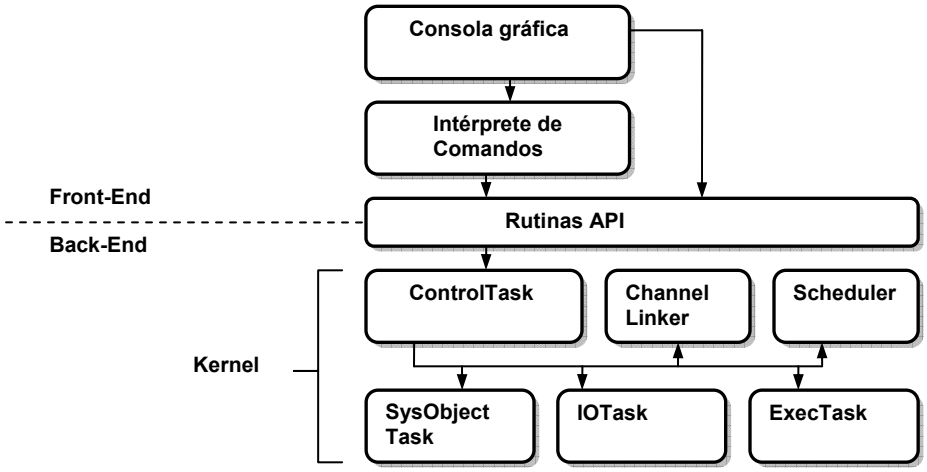


Figura 2.8 Arquitectura del *NEngine* basada en una estructura de procesos concurrentes.

El *Front-End* está formado por un conjunto de interfaces gráficas que posibilitan el intercambio de información bilateral entre el sistema y el usuario. El usuario comunica las órdenes al sistema a través de un lenguaje de comandos que es interpretado por un *parser* construido al efecto; y recibe de éste información periódica acerca del estado de los procesamientos.

Los comandos son traducidos por el *parser* para ejecutar las rutinas API pertinentes. Cada una de estas rutinas genera los mensajes correspondientes y los envía a la tarea de control; ésta los reconoce y distribuye hacia los tres restantes procesos para su manipulación, previamente realiza la solicitud de los permisos de ejecución necesarios para garantizar el sincronismo. El proceso *SysObjectTask* se encarga de la administración de los objetos propios de la topología de las redes neuronales: capas, neuronas, conexiones, etc. *ExecTask* manipula los mensajes concernientes a la ejecución de los algoritmos de entrenamiento y cálculo en las redes diseñadas

previamente. *IOTask* realiza las operaciones de entrada y salida (persistencia) de información de proyectos de diseño, entrenamiento, bases de casos, etc.

II.3.1. El Proceso SysObjectTask.

Este proceso implementa rutinas para el diseño de topologías de red. (El Anexo 5.2 muestra clases para el almacenamiento de topologías). En él se manipulan mensajes para la creación y eliminación de neuronas, capas de neuronas y conexiones. Internamente el proceso es capaz de manejar una lista de objetos de diseño a manera de grafo orientado que refleja las capas y las conexiones que conforman una red determinada. Cada objeto que conforma una topología posee un nombre único y datos particulares sobre el mismo. Las capas de redes son una forma de manipular topologías formadas por capas con grandes números de neuronas. Cada capa es descrita por su nombre, su tipo (atendiendo a si es una capa de entrada, de salida o una capa oculta), cantidad de neuronas y las conexiones que posee. Una neurona se modela como una capa de una sola neurona. Las conexiones poseen igualmente un nombre y la capa de origen y destino que conecta. Cada conexión posee un atributo adicional un número entero que designa el desplazamiento (*shift operator*) del que se habló en el capítulo anterior. La estructura de dato que se emplea para almacenar estos objetos es una lista *hash* indexada por el nombre de cada objeto.

El proceso implementa un manipulador para el mensaje `OMSG_SYSOBJECT`. Cuando el usuario realiza un cambio en el diseño de la topología, el proceso SysObjectTask recibe un mensaje de este tipo comunicando los detalles del cambio.

II.3.2. El Proceso ExecTask.

Este proceso se encarga de recibir la orden para la ejecución de los algoritmos de entrenamiento o el cálculo de las salidas de la red dado un conjunto de patrones de entrada. *ExecTask* posee dos mensajes principales: `OMSG_BPTT_TRAIN` y `OMSG_BPTT_OUTS` para ordenar la ejecución del algoritmo de entrenamiento y el cálculo de las salidas respectivamente. El algoritmo implementado para el entrenamiento de la red está basado en las ideas discutidas en el capítulo anterior sobre el modelo unificador basado en diagramas de bloques, por lo que considera cada neurona como una unidad computacional con dos direcciones de cálculo: una forward,

para el cálculo de las salidas y la otra backward, para la retropropagación del error. (Ver epígrafe I.4). Presentar un patrón a la red y reajustar el valor de las conexiones según el gradiente de error, involucra la ejecución de cada unidad computacional en las dos direcciones. La planificación de qué unidad computacional se ejecutará cada vez se realiza comenzando por las unidades de entrada según la dirección que sea (unidades de la capa de entrada para la dirección forward, o de la capa de salida si es la backward) y activando sucesivamente las unidades cuyos antecesores (según la dirección) ya hayan sido calculados (Ver figura A1.5). Para calcular la salida de cada unidad se emplea una rutina que aplica la función de activación cuando se ejecuta la dirección forward, y aplica la regla de retropropagación en el caso backward. La implementación puede realizar de la misma forma el proceso forward y el backward, solo varían las funciones empleadas para calcular los valores de salida de cada unidad según el caso. Como se explicó en el epígrafe (I.2.1), el algoritmo BPTT replica la red neuronal por cada etapa del ejemplo de entrenamiento y emplea las conexiones recurrentes como conectores entre las redes de una etapa y otra. Esto hace que las conexiones y los umbrales de las neuronas (*bias*) también sean replicadas. Cuando se va a ejecutar la rutina de entrenamiento la información contenida en las estructuras manipuladas por el proceso *SysObject* son traducidas a una estructura de datos adicional en términos de neuronas y conexiones entre neuronas solamente, o sea, las capas son desplegadas en neuronas y sobre éstas se ejecuta el algoritmo. Los valores de salida de cada unidad computacional y los valores de actualización de las conexiones son almacenados en el NEngine sobre estructuras matriciales, donde cada columna representa una conexión o neurona y cada fila las distintas etapas (o elementos) de las que se compone la secuencia. Se emplea una matriz para calcular las actualizaciones de los pesos y otra matriz para calcular las actualizaciones de los umbrales. En la figura 2.9 se muestran las matrices y la estructura de los elementos de cada matriz.

Los valores de las actualizaciones de los parámetros por cada etapa son sumados para obtener el valor final de la actualización del peso correspondiente en la red original. Cuando se intenta usar la red como clasificador el algoritmo usa el principio Winner-Take-All, donde la salida de mayor valor representa la clase obtenida.

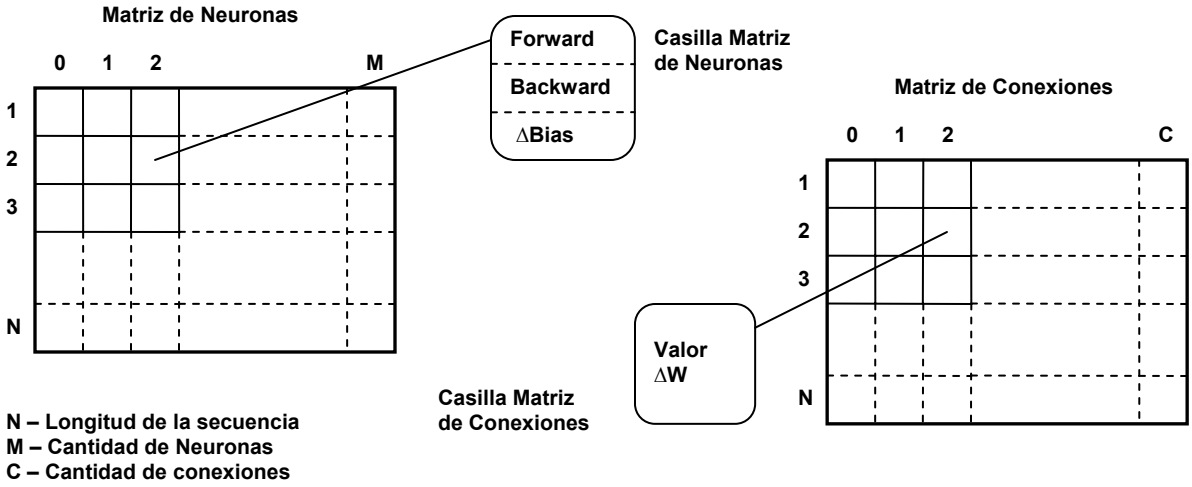


Figura 2.9 Estructura de datos para el entrenamiento.

A continuación mostramos en seudolenguaje el algoritmo general implementado:

```
PROC Train()
NextCase ⟵ SelectNextCase()
WHILE NOT (NextCase = ϕ) DO
 Exec(FORWARD, NextCase)
 Exec(BACKWARD, NextCase)
 RecoverCaseInfo(NextCase)
 NextCase ⟵ SelectNextCase()
ENDW
ENDP Train
```

La rutina anterior constituye un esquema general usado para todas las estrategias de entrenamiento, y consiste básicamente en un ciclo repetitivo de presentación de ejemplos (`NextCase`) a la red mediante la rutina `Exec(FORWARD, NextCase)` y la actualización de los pesos mediante `Exec(BACKWARD, NextCase)`. La rutina `SelectNextCase` selecciona el nuevo caso a presentar a la red siguiendo una estrategia determinada. La rutina `RecoverCaseInfo` permite la realización de alguna operación adicional cada vez que se ejecute cada caso (por ejemplo, calcular el error cuadrático producido por el caso y su actualización en una variable global, o en entrenamientos dirigidos por casos, actualizar los valores de los pesos a partir de las variaciones calculadas por la ejecución backward). A continuación se muestra el esquema general de esta rutina:

```
PROC SelectNextCase()
```

```
IF ScheduledCases = ϕ THEN
 IF StopCondition() = TRUE THEN
  Result ⟵ ϕ
 ELSE
  ScheduledCases ⟵ NextScheduledCases()
  Result ⟵ First(ScheduledCases)
 ENDIF
ELSE
 Result ⟵ First(ScheduledCases)
ENDIF
ENDW
ENDP SelectNextCase
```

Esta rutina basa su funcionamiento en la planificación de la presentación de los casos por iteraciones, hasta que se detecta la condición de parada `StopCondition() = TRUE`. `ScheduledCases` actúa como una cola donde cada caso posee un orden determinado; cada vez que un caso es presentado a la red (`First(ScheduledCases)` selecciona el primer caso de la cola), éste es eliminado de la cola. La rutina `NextScheduledCases` determina la forma en que los casos son ordenados en cada iteración. Diferentes estrategias de presentación de los ejemplos de entrenamiento conllevan a diferentes implementaciones concretas de esta rutina.

La rutina `Exec()` se presenta como sigue:

```
PROC Exec(direction, case)
ComputedUnits ⟵ ϕ
ScheduledUnits ⟵ InitialUnits(direction,case)
WHILE ScheduledUnits ≠ ϕ DO
 FOR i ⟵ 1 TO LENGTH(ScheduledUnits) DO
  ComputeUnit(ScheduledUnits[i], direction, case)
  ComputedUnits ⟵ ComputedUnits ∪ ScheduledUnits[i]
 ENDF
 ScheduledUnits ⟵ NextScheduledUnits(ComputedUnits)
ENDW
ENDP Exec
```

La rutina `InitialUnits` inicializa las estructuras de almacenamiento necesarias y planifica las primeras unidades a computar:

```
PROC InitialUnits(direction, case)
InitNeuronAndConnecMatrix(case)
External ⟵ ExternalNeurons(direction)
FOR i ⟵ 1 TO LENGTH(case) DO
```

```
 FOR j ⟵ 1 TO LENGTH(External) DO
  Unit.Step ⟵ i
  Unit.Neuron ⟵ j
  ScheduledUnits ⟵ ScheduledUnits ∪ Unit
 ENDF
ENDF
ENDP InitialUnits
```

La rutina actualiza las estructuras matriciales descritas en la figura 2.9 y planifica, para su ejecución, las unidades externas correspondientes. Cada unidad es caracterizada por la neurona y la etapa de la secuencia que representa. Finalmente la rutina ComputeUnit se encarga de calcular el valor de salida de las unidades computacionales (ver figura A1.4). Su definición es como sigue:

```
PROC ComputeUnit(Unit, direction, case)
IF direction = FORWARD THEN
 IF Unit.Neuron ∈ ExternalNeurons(FORWARD) THEN
  Value ⟵ CalcExtForwardValue(Unit, case)
 ELSE
  Value ⟵ CalcForwardValue(Unit)
 ENDIF
NeuronMatrix[Unit.Step,Unit.Neuron].ForwardValue ⟵ Value
ELSE
 IF Unit.Neuron ∈ ExternalNeurons(BACKWARD) THEN
  Value ⟵ CalcExtBackwardValue(Unit, case)
 ELSE
  Value ⟵ CalcBackwardValue(Unit)
 ENDIF
NeuronMatrix[Unit.Step,Unit.Neuron].BackwardValue ⟵ Value
ENDIF
ENDP ComputeUnit
```

Los valores de actualización contenidos en la matriz de conexiones `ConnecMatrix` son calculados como parte de la rutina `CalcBackwardValue(Unit)`, al determinar el valor de error correspondiente a la unidad `Unit` siguiendo las fórmulas (I.13), (I.14).

Un entrenamiento secuencial (estrategia secuencial para la presentación de los casos) dirigido por casos (las actualizaciones de los pesos se realizan después de presentar cada caso a la red) y con un umbral de error como condición de parada puede obtenerse con las siguientes redefiniciones:

```
PROC RecoverCaseInfo(case)
Error ⟵ CalculateSquareError(case)
SumError ⟵ SumError + Error
ActualizeConnectionAndBias()
ENDP RecoverCaseInfo
```

La rutina `CalculateSquareError` calcula el error cuadrático medio producido por la presentación del caso `case` a la red. `ActualizeConnectionAndBias` calcula la actualización de los pesos a partir de los valores contenidos en la matriz de conexiones. `SumError` almacena el error total producido por la presentación a la red de todos los casos de una iteración.

```
PROC StopCondition()
IF SumError / LENGTH(CaseBase) < ErrorThreshold THEN
 Result ⟵ TRUE
ELSE
 Result ⟵ FALSE
ENDIF
ENDP StopCondition

PROC NextScheduledCases()
ScheduledCases ⟵ ϕ
FOR i ⟵ 1 TO LENGTH(CaseBase) DO
 ScheduledCases ⟵ ScheduledCases ∪ i
ENDF
ENDP NextScheduledCases
```

`CaseBase` representa la base de casos, `ErrorThreshold` el umbral de error usado para terminar el entrenamiento.

NEngine contempla una variedad de funciones de activación para las neuronas de la capa de salida (función lineal, sigmoidal (I.2), softmax(I.3)), que pueden ser combinadas con distintas funciones de error (error cuadrático medio (I.5), *cross-entropy* para clasificación binomial y multinomial (I.6)) para lograr el modelo de red adecuado a cada tarea particular. Adicionalmente implementa *random cross-validation* como método para la estimación de los parámetros de generalización de la red.

II.3.3. El Proceso IOTask.

Este proceso se encarga de cargar y salvar proyectos de diseño en fase de construcción, entrenamiento, o explotación. Se ocupa además de la carga de las bases de casos. Un proyecto de red neuronal es considerado como el conjunto de capas y

conexiones que conforman la topología de la red, los valores de los parámetros libres del modelo (pesos y umbrales) y los parámetros de entrenamiento (valores de las constantes alfa y beta, etc.). En el disco, un proyecto se conforma de un archivo .netprj que contiene información sobre el nombre y la versión (un número entero de 4 bytes) del proyecto, algunos parámetros de entrenamiento y la referencia hacia el fichero que contiene la información de la estructura topológica de la red (ficheros con extensión .netdsg), y el fichero que contienen los valores de los pesos y umbrales (extensión .netbin), si es que el proyecto ha ejecutado alguna vez el algoritmo de entrenamiento sobre la red del proyecto. Este proceso dispone de un mensaje para cargar y salvar ficheros con extensión .netbin sin necesidad de que esté asociado a ningún proyecto; esto le da la oportunidad a los usuarios una vez que sus redes neuronales han sido entrenadas, de salvarlas sin información adicional acerca del diseño de la topología.

OMSG_LOAD_PRJDSG, OMSG_SAVE_PRJSG son los mensajes para la carga y salva de un proyecto, OMSG_LOAD_CASES es para la carga de una base de casos.

El formato de los archivos de proyecto es como sigue:

Version	Bin	Norm	RMin	RMax	alfa	beta
Nombre Proyecto		Archivo Diseño		Archivo Parámetros		

Figura 2.10 Formato de los fichero de proyecto (.netprj).

Bin es una byte que actúa como bandera y expresa si el proyecto tiene un fichero de parámetros; Norm almacena información sobre la normalización de las entradas, el rango [RMin, Rmax] es usado para la normalización; alfa y beta son los parámetros del algoritmo de entrenamiento.

Cant. N	Objeto 1	Objeto 2	.....	Objeto N

Si el objeto i es:					
	conexión	delay	Source	target	name
	capa	type	length	name	

Figura 2.11 Formato de los fichero de diseño (.netdsg).

En los objetos conexión, `source` y `target` se refieren a los nombres de las capas de origen y destino; en los objetos capa, `type` se refiere al tipo de capa (entrada, salida, oculta) y `length` se refiere a la cantidad de neuronas de la capa.

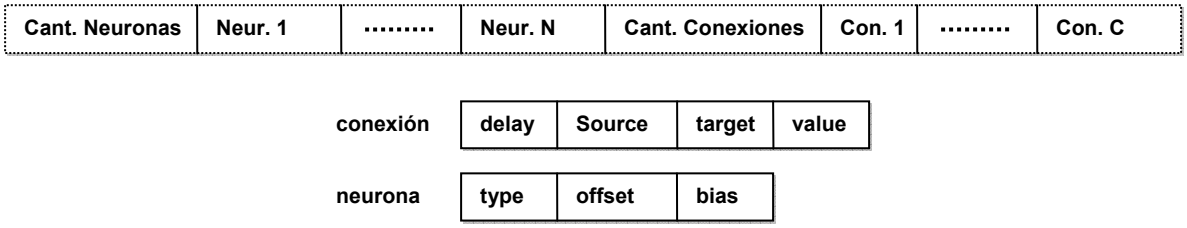


Figura 2.12 Formato de los fichero de parámetros (`.netbin`).

II.3.4. El Modelo de Sincronización.

Entre los mensajes que son atendidos por los procesos en NEngine, solo los que se han mencionado aquí poseen alguna relación de exclusión. A continuación mostramos un grafo cuyos arcos muestran las relaciones de exclusión entre ellos.

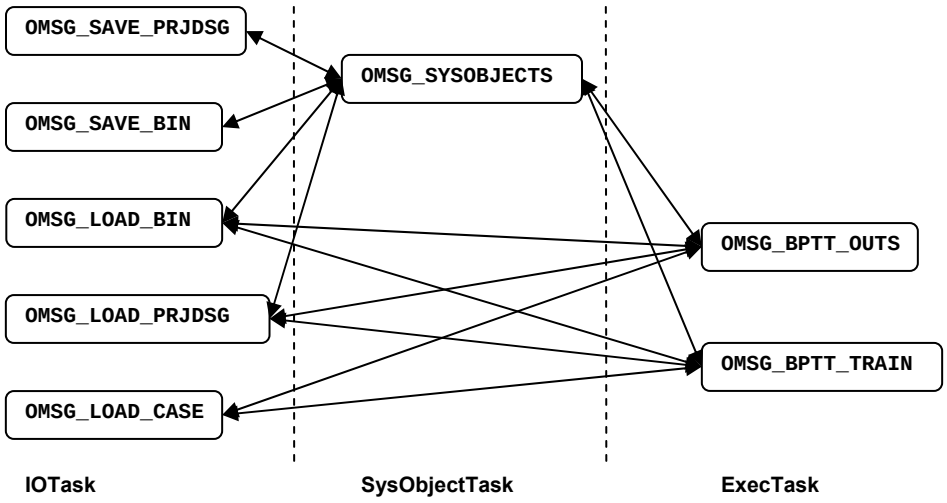


Figura 2.13 Relaciones de exclusión entre mensajes

Atendiendo al modelo de sincronización descrito en el epígrafe II.2 se pueden establecer las siguientes clases de permisos de ejecución:

```
{OMSG_BPTT_OUTS, OMSG_BPTT_TRAIN}   = TK_BPTT
{OMSG_SAVE_PRJDSG, OMSG_SAVE_BIN}   = TK_SAVE_STRUCT
{OMSG_LOAD_PRJDSG, OMSG_LOAD_BIN}   = TK_LOAD_STRUCT
(OMSG_LOAD_CASE}                    = TK_LOAD_CASE
(OMSG_SYSOBJECTS}                   = TK_SYSOBJECTS
```

Con las siguientes relaciones de exclusión:

```
TK_BPTT           {TK_BPTT,TK_SYSOBJECTS, TK_LOAD_CASE, TK_LOAD_STRUCT}
TK_SAVE_STRUCT    {TK_SAVE_STRUCT,TK_SYSOBJECTS}
TK_LOAD_STRUCT    {TK_LOAD_STRUCT, TK_SYSOBJECTS, TK_BPTT}
TK_LOAD_CASE      {TK_LOAD_CASE, TK_BPTT}
TK_SYSOBJECTS     {TK_SYSOBJECTS, TK_LOAD_STRUCT, TK_SAVE_STRUCT}
```

Capítulo III. Manual de Usuario del NEngine v1.0.

El presente capítulo muestra información orientada a la explotación de la herramienta NEngine por parte del usuario. En el primer epígrafe se brinda un tutorial sobre el trabajo básico con el NEngine: instalación, requerimientos técnicos, principales funcionalidades. En el segundo epígrafe se presenta una referencia del modo de uso de los comandos fundametales disponibles en esta versión. El tercer epígrafe muestra algunas pruebas de validación realizadas al NEngine.

III.1 Introducción al NEngine

NEngine version 1.0 es un software previsto para asistir la construcción de modelos de redes neuronales y su aplicación a problemas del mundo real. Específicamente problemas de clasificación y análisis de secuencias de datos. (secuencias espaciales o temporales). NEngine provee soporte para la construcción, entrenamiento y explotación de redes recurrentes discretas (*Backpropagation ThroughTime*) y redes *feedforward* (*Backpropagation*).

Esta herramienta incorpora resultados teóricos importantes en el tema de la unificación de modelos de redes neuronales (Ver Capítulo I y II); esto junto a que posee una estructura basada en procesos concurrentes le confiere a la aplicación un carácter robusto y flexible para su extensión a una plataforma de redes neuronales; además de su fácil adaptación a entornos paralelos.

La aplicación se organiza en forma de dos capas (ver figura III.1): una capa núcleo (*back-end*) que implementa todas las funcionalidades: gestión de topologías, ejecución de los algoritmos de entrenamiento, etc. y una capa exterior (*front-end)* que encapsula al núcleo y traduce sus funcionalidades en forma accesible por los usuarios. La forma primaria de comunicación del usuario con el NEngine es a través de un lenguaje de comandos con instrucciones especiales para la creación de topologías, entrenamiento de las redes y la configuración del trabajo del kernel. El *front-end* interactúa con el núcleo a partir de la interpretación de las instrucciones ordenadas por el usuario y la llamada a las funciones correspondientes del núcleo.

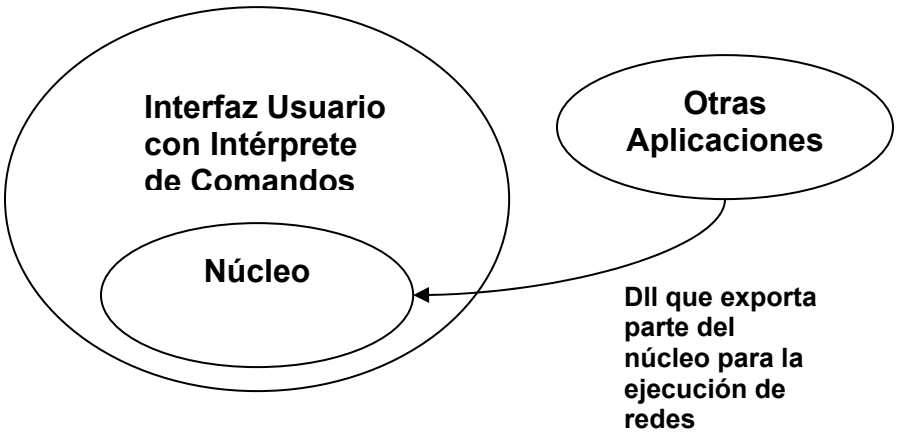


Figura 3.1 Organización del NEngine

El NEngine posee una interfaz de usuario basada en ventanas que permiten la edición y ejecución de comandos en forma independiente o de lote (ficheros con extensión `.fb`); además de brindar información gráfica sobre el proceso de construcción de las topologías (capas y conexiones) de red y el entrenamiento (curva de error en la convergencia, etc.).

III.1.1. Instalación.

El NEngine se distribuye como un programa de instalación con nombre `setupNEngine10`. Este programa despliega una estructura de directorios donde se encuentran los archivos ejecutables, ejemplos y documentación sobre el trabajo con NEngine. La estructura se crea en el directorio que sea especificado por el usuario y su organización es la siguiente:

`..\[Directorio Instalación]\`

`Bin\`	Directorio de archivos ejecutables	
	`NEngine.exe`	Ejecutable del NEngine
	`NEngine.dll`	Biblioteca para la exportación de modelos
	`ImportNEngine.pas`	Unit de prototipos para la importación
	`NEngine.ini`	Archivo de inicialización del NEngine
`Cases\`	Directorio de bases de Casos	
	`xor.csb`	Archivo texto base de casos Xor
	`Xor.csb..bin`	Archivo binario base de caos Xor
	`Iris.csb`	Archivo texto base casos Iris

	Iris.csb..bin	Archivo binario base casos Iris
	Bcancer.csb	Archivo texto base casos BCancer
	Bacancer.csb..bin	Archivo binario base casos BCancer
Doc\	Directorio de Documentación	
	NEngine.chm	Archivo de ayuda del NEngine
Projects\	Directorio de proyectos	
	Iris-1\	Proyecto Iris
	BCancer\	Proyecto BCancer
	Xor\	Proyecto Xor
Samples\	Ejemplos de archivos de comandos (.bf)	
	Xor.bf	Comandos para el proyecto Xor
	Iris.bf	Comandos para el proyecto Iris
	bCancer.bf	Comandos para el proyecto BCancer

El directorio de instalación debe poseer atributos de escritura para todos los usuarios del NEngine. Para poseer un icono en el menú de inicio de todos los usuarios debe incorporar el correspondiente acceso directo en la carpeta [All Users].

Esta distribución del NEngine está compilada para plataforma Windows NT/2000 con un requerimiento de memoria inicial de 6 Mb aproximadamente.

III.2 Funcionalidades Básicas del NEngine.

Típicamente, el uso de NEngine para resolver un problema de redes neuronales involucra las siguientes etapas:

- Construcción de la topología de red.
- Edición y carga de la Base de Casos.
- Entrenamiento del modelo de red. Experimentación con diferentes parámetros del algoritmo de entrenamiento.
- Salva y exportación de los resultados para su utilización en otras aplicaciones.

El usuario puede seguir el siguiente flujo:

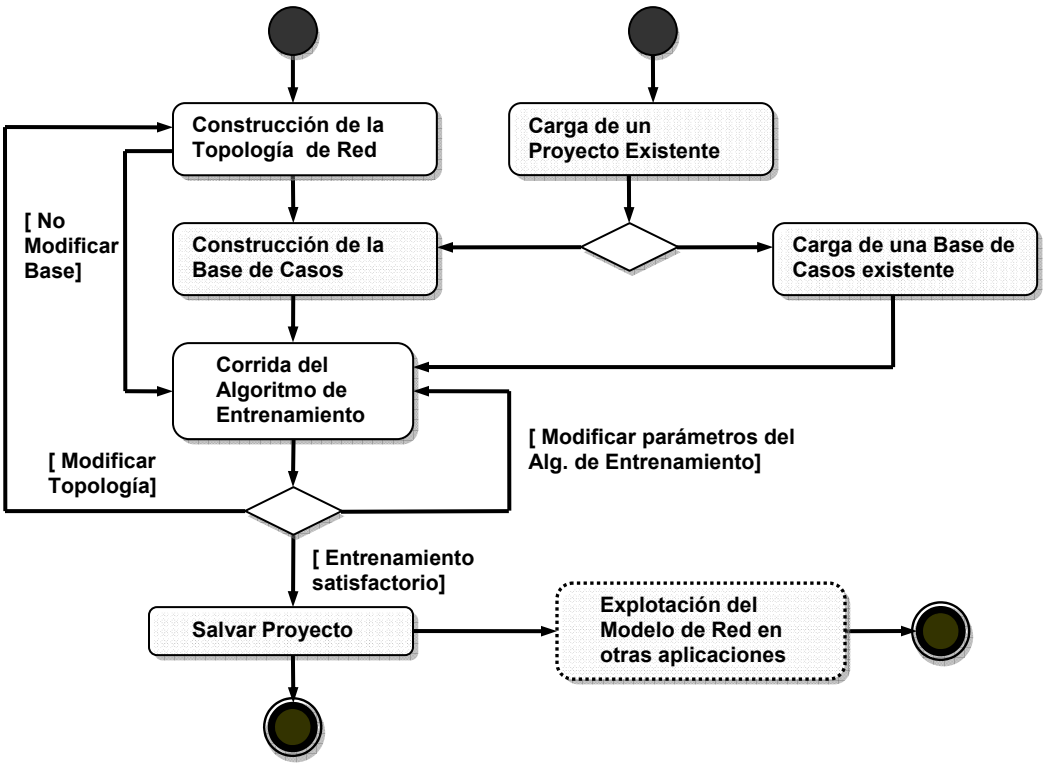


Figura 3.2 Flujo de Trabajo en el NEngine

III.2.1 Construcción de Topologías.

La construcción de los modelos de red se realiza agregando las capas y las conexiones pertinentes con el empleo de los comandos `CREATE LAYER` y `CREATE CONECCTION`. `CREATE LAYER` permite crear una capa con una cantidad arbitraria de neuronas, especificando si es una capa de salida, entrada u oculta. `CREATE CONNECTION` permite especificar el desplazamiento de la conexión (*shift operator*). Cada objeto en la topología posee un nombre único. Las instrucciones `DROP LAYER` y `DROP CONNECTION` permiten eliminar capas y las conexiones respectivamente. `DROP ALL` reinicia el modelo, eliminando todos los objetos creados.

NEngine posee una ventana especial que refleja gráficamente la estructura de la topología (ver figura 3.3). Cada capa es representada por un bloque de color rojo si es una capa de entrada, azul si es una capa oculta y verde en caso de que sea de salida. Un número dentro de cada bloque indica la cantidad de neuronas de la capa. Debajo de cada bloque se indica el nombre de la capa y el tipo de neuronas que agrupa atendiendo a si son neuronas reales o binarias. Las neuronas binarias poseen un

modelo de neurona exactamente igual que las neuronas reales, solo que su salida es binarizada al ocupar la capa de salida. Las conexiones entre capas se reflejan con saetas que parten desde la capa origen hacia la capa destino; y son caracterizadas por un nombre y el operador de desplazamiento. Las conexiones no recurrentes (operador de desplazamiento igual a 0) no muestran su valor.

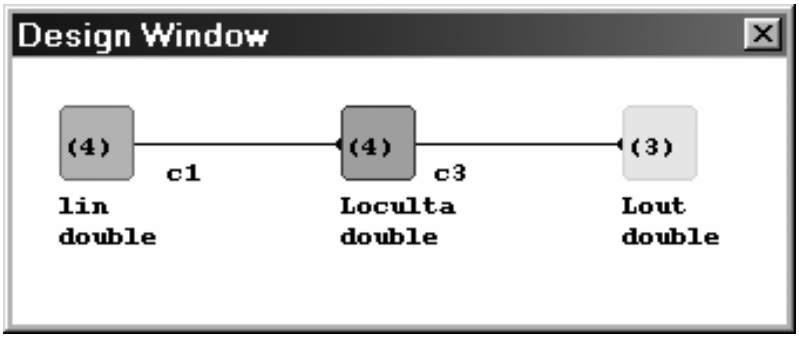


Figura 3.3 Ventana de Diseño de Topología.

La figura muestra la topología de una red recurrente con una capa de entrada de 4 neuronas (Lin), una capa de salida de 3 neuronas (Lout) y una capa oculta (Loculta) con una conexión recurrente (c2) ; y a continuación se presenta la secuencia de comandos usados para generar tal topología:

```
CREATE LAYER lin AS LENGTH  4, DOUBLE, INPUT
CREATE LAYER loculta AS LENGTH  2, DOUBLE, MIDDLE
CREATE LAYER lout AS LENGTH  3, DOUBLE, OUTPUT
CREATE CONNECTION c1 AS DELAY 0, SOURCE lin, TRAGET loculta
CREATE CONNECTION c2 AS DELAY +1, SOURCE loculta, TRAGET loculta
CREATE CONNECTION c3 AS DELAY 0, SOURCE loculta, TRAGET lout
```

Si queremos eliminar la conexión recurrente usamos la instrucción:

```
DROP CONNECTION c2
```

Si deseáramos eliminar la capa `loculta`, por ejemplo, usamos la instrucción:

```
DROP LAYER loculta
```

Al eliminar la capa `loculta` también serían eliminadas las conexiones `c1` y `c2`.

III.2.2 Edición y Carga de Bases de Casos.

NEngine posee una sintaxis particular para editar una base de casos en forma de texto (ficheros con extensión .csb) (ver figura 3.4).

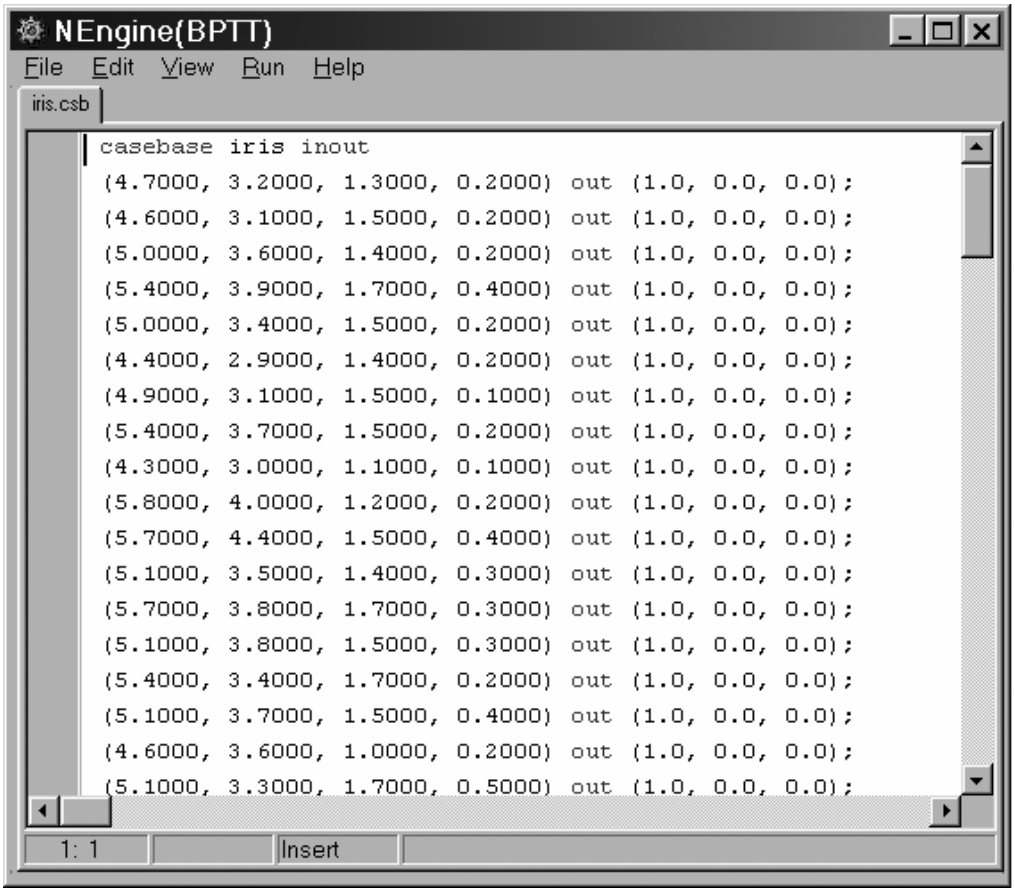


Figura 3.4 Fichero texto para la edición de una Base de Casos.

La sintaxis de una base de casos con entradas y salidas es como sigue:

```
<casebase_INOUT> ::= casebase <name> inout <cases_list> .
<case_list>  ::= <case> | <case> ; <case_list>
<case> ::=  <step_list> out <step_list>
<step_list> ::= <step> | <step> , <step_list>
<step> ::= ( <number_list> )
<number_list> ::= <number> | <number> , <number_list>
<step_list> ::= <step> | <step> , <step_list>
```

Para una base de casos de entradas solamente:

```
<casebase_IN> ::= casebase <name> in <cases_list> .
<case_list>  ::= <case> | <case> ; <case_list>
<case> ::=  <step_list>
<step_list> ::= <step> | <step> , <step_list>
<step> ::= ( <number_list> )
<number_list> ::= <number> | <number> , <number_list>
```

<step_list> ::= <step> | <step> , <step_list>

El análisis semántico de una base de casos incluye verificar que el número de etapas en la entrada y la salida de cada ejemplo coincidan. NEngine hasta el momento no admite casos con atributos reales y binarios mezclados, por lo que todas las entradas y salidas deben ser formadas completamente por atributos reales o por atributos binarios. El kernel de NEngine carga las bases de casos usando un formato binario; por lo que cada base de caso en texto (.csb) debe ser compilada (.csb..bin) a dicho formato antes de su presentación. El editor de texto del sistema NEngine está preparado para generar el fichero binario (.csb..bin) cada vez que se salve una modificación del fichero texto (.csb). Adicionalmente se encuentra disponible la instrucción GENERATE CASEBASE FROM <filename.csb>, cuya misión es generar el formato binario de una base de casos contenida en el fichero <filename.csb >. El fichero generado tiene el nombre <filename.csb..bin >.

Para cargar un fichero binario en el kernel de NEngine se emplea el comando LOAD CASES IN <filename.csb>. Este comando en realidad espera que exista la versión binaria <filename.csb..bin>.

III.2.3 Entrenamiento de Modelos de Red Neuronal.

Una vez que ha sido construida la topología de la red, y una base de casos apropiada ha sido cargada en el kernel del NEngine, el usuario puede disponer la ejecución de los algoritmos de entrenamiento. NEngine tiene implementado básicamente dos algoritmos: el *backpropagation estándar* y una extensión de éste para redes recurrentes, el *backpropagation throughtime* (BPTT). El kernel es capaz de detectar, automáticamente a partir de la topología creada, cuál es el algoritmo que debe aplicarse. Esto se debe a que NEngine incorpora una serie de ideas teóricas que facilitan el trabajo en este sentido. Por lo tanto existe una sola instrucción para lanzar el entrenamiento, EXEC BPTT TRAIN. Esta instrucción posee dos variantes según la condición de parada que desee establecerse:

EXEC BPTT TRAIN UNTIL ITERATION <number> para ejecutar hasta que un número de iteraciones <number> sea alcanzado, y

EXEC BPTT TRAIN UNTIL ERROR <error_expected> para ejecutar hasta que sea alcanzado un error cuadrático medio inferior a <error_expected>.

Antes de ejecutar los algoritmos de entrenamiento se aplica una rutina de chequeo de validez de la topología creada. En el caso del BPTT, el modelo de red creado previamente no puede contener ciclos de longitud cero (suma de los valores de desplazamiento, *delays* de las conexiones involucradas). Adicionalmente existe un conjunto de instrucciones conmutadoras (*switch instructions*) que permiten modificar los parámetros del algoritmo, la estrategia de presentación de los ejemplos, el procesamiento de las salidas, las técnicas de estimación de resultados, etc. Podemos encontrar en este caso la instrucciones SET ALFA, SET BETA, que modifican parámetros que afectan la *velocidad de convergencia* del algoritmo de entrenamiento; SET STRATEGY, encargada de seleccionar la estrategia de presentación de los ejemplos a la red; SET NORMALIZATION, para el pre-procesamiento de las entradas, SET CROSSVALIDATION para establecer la técnica de estimación de la generalización; entre otras. (Para una descripción más detallada sobre el funcionamiento de estos comandos ver el manual de Referencia de Comandos en el próximo epígrafe).

Existe el comando EXEC BPTT OUTPUT para calcular las salidas que brinda la red para toda la base de casos cargada; esta instrucción permite obtener detalles sobre el error cuadrático promedio y los porcientos de clasificación correcta usando la base de casos total. Cuando el usuario lo estime conveniente puede detener el proceso de entrenamiento usando el comando ABORT BPTT. RESET se usa para reiniciar los parámetros libres de los modelos (pesos y umbrales de las neuronas).

En la siguiente figura se muestra una pantalla del NEngine mientras entrena una red MLP de tres capas para analizar la base de casos Iris.

La ventana inferior ScaleWindow muestra una gráfica del comportamiento del error a través de las etapas de entrenamiento. Ésta, junto a la ventana MessagesWindow, constituyen las formas básicas de recepción por parte del usuario de la información acerca del funcionamiento de los procesos en el kernel. El comando PRINT EXEC imprime información a través de la ventana de mensajes (MessagesWindow) sobre el

estado actual del proceso de entrenamiento: error cuadrático promedio, estrategias usadas, parámetros de velocidad de convergencia empleados, iteración actual, etc.

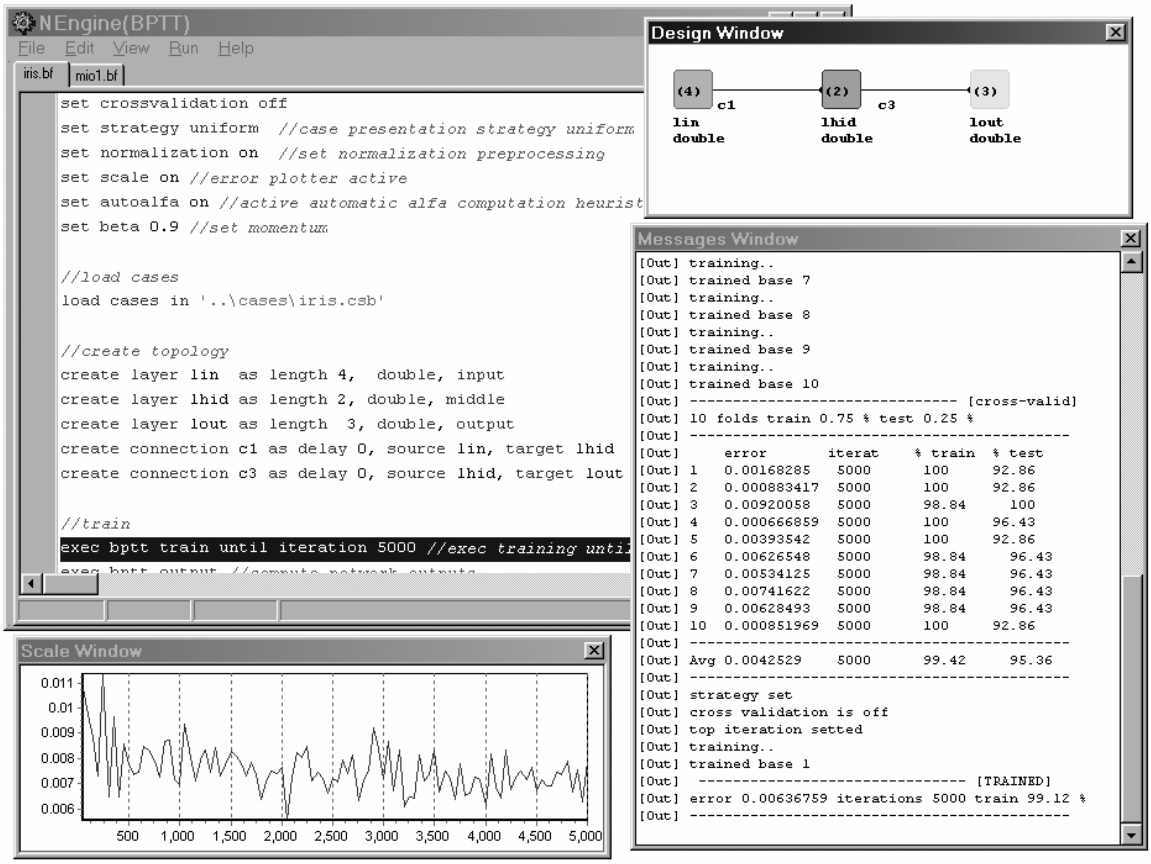


Figura 3.5 Entrenamiento de una red MLP para la base de casos Iris.

III.2.4 Salva y Recuperación de Proyectos. Explotación de Modelos.

Después que ha obtenido un buen resultado en el entrenamiento de la red, o incluso un resultado parcial de consideración, el usuario puede salvar los parámetros de la red, su topología y algunos detalles sobre el contexto de entrenamiento. Esto permite una mejor planificación del diseño y entrenamiento de los modelos, además de la generación de múltiples versiones, útiles en la selección de la mejor alternativa. El usuario puede entrenar una red en diversos momentos de tiempo sin necesidad de recomenzar todo el trabajo cada vez. NEngine dispone de una estructura de ficheros donde almacena la información de la topología, parámetros libres e información general del proyecto:

- Fichero de Proyecto (`.netprj`). Contiene información general sobre el proyecto y referencias a los demás archivos.
- Fichero de Parámetros Libres (`.netbin`). Este fichero se genera con el proyecto si el modelo ha sido entrenado y dispone de la información de los parámetros libres (pesos y umbrales) en el momento de salvar el proyecto.
- Fichero de Topología (`.netdsg`). Este fichero se genera cada vez que se salva un proyecto y contiene la información relativa a la topología de la red.

Para salvar un proyecto se usa el comando `SAVE PROJECT <project_name> AS VERSION <number> TO < file_name >`. Para cargar un proyecto en el kernel se emplea el comando `LOAD PROJECT IN <file_name>`. Cuando el usuario ha logrado un conjunto de parámetros definitivo para el modelo de red que investiga, tiene la oportunidad de salvar los parámetros en un fichero de parámetros libres simplemente, sin necesidad de otro fichero adicional. EL NEngine permite cargar también la red contenida de forma independiente en este tipo de ficheros (no se pueden realizar cambios en la topología de una red almacenada en un fichero `.netbin`). Estas redes pueden reentrenarse en el NEngine o usarse en aplicaciones externas por medio de la librería de enlace dinámico disponible con la instalación del NEngine. (Ver la ayuda del NEngine para más detalles sobre cómo usar esta librería).

En el siguiente manual se especifica el uso de algunos de los comando incluidos en la versión 1.0 del NEngine. Para una lista completa ver la ayuda del sistema.

III.3 Manual de Referencia de Comandos.

ABORT BPTT.

Abortar el proceso de entrenamiento.

Sintaxis.

```
ABORT BPTT
```

CREATE CONNECTION.

Permite conectar dos capas neurona a neurona (conexión completa entre capas).

Sintaxis.

```
CREATE CONNECTION <connection_name> AS DELAY <number>, SOURCE <src_layer_name>, TARGET <trg_layer_name>
```

Descripción.

Crea una conexión con el nombre `<connection_name>`, que conecta completamente las capas `<src_layer_name>` y `<trg_layer_name>`. EL argumento `DELAY` especifica el operador de desplazamiento (*shift operator*) de la conexión. Para conexiones no recurrentes se emplea el 0, para conexiones desde etapas anteriores se emplean números positivos y para conexiones desde etapas posteriores, números negativos. El valor absoluto indica la cantidad de etapas hacia atrás o hacia delante en la conexión.

Ejemplos.

El ejemplo muestra como crear una conexión no recurrente entre las capa Lin y Lmid.

```
CREATE CONNECTION c1 AS DELAY 0, SOURCE Lin, TARGET Lmid
```

CREATE LAYER.

Incorporar una capa de neuronas en la topología de la red que se construye.

Sintaxis.

```
CREATE LAYER <layer_name> AS LENGTH <number>, [DOUBLE| BINARY], [INPUT| MIDDLE| OUTPUT]
```

Descripción.

Crea una capa con el nombre `<layer_name>`, que posee un número de neuronas especificado en el primer argumento (`LENGTH`). El segundo modificador permite indicar si la capa es de neuronas binarias o reales. El tercer argumento especifica si la capa es de entrada, salida u oculta. En las capas de salida el valor de activación real es convertido a un valor binario de acuerdo a un umbral especificado (ver `SET BINARY RANGE`). En cualquier diseño el nombre de las capas creadas es único y debe comenzarse especificando la capa de entrada. Una capa con una sola neurona puede considerarse como una neurona independiente.

Ejemplos.

El ejemplo muestra como crear una capa de entrada de 3 neuronas reales llamada Lin.

```
CREATE LAYER Lin AS LENGTH 3, INPUT, DOUBLE
```

DROP ALL.

Eliminar todos los objetos de topología (capas y conexiones) creados.

Sintaxis.

```
DROP ALL
```

DROP CONNECTION.

Eliminar una conexión entre capas.

Sintaxis.

```
DROP CONNECTION <connection_name>
```

DROP LAYER.

Eliminar una capa de neuronas.

Sintaxis.

```
DROP LAYER <layer_name>
```

Descripción.

Elimina automáticamente las conexiones de entrada y salida de la capa especificada. Si no existe la capa, retorna un error.

EXEC BPTT OUTPUT.

Calcular la salida de la red sobre los casos cargados en el núcleo.

Sintaxis.

```
EXEC BPTT OUTPUT
```

Descripción.

Este comando brinda la salida de cada neurona en cada secuencia de entrada. Si WTALL está activado, este comando brinda además el porciento de clasificación correcta.

EXEC BPTT TRAIN.

Ejecutar el algoritmo de entrenamiento.

Sintaxis.

```
EXEC BPTT TRAIN {UNTIL ERROR <error> | UNTIL ITEATION <number>}
```

Descripción.

Este comando ordena al kernel que ejecute el algoritmo de entrenamiento. Antes se comprueban una serie de condiciones: exista una base de casos, la topología de la red concuerde con la estructura de la base de casos, no exista una topología inválida (ciclos), etc. `UNTIL ERROR` establece como condición de parada alcanzar el umbral <error>, y `UNTIL ITERATION` la ejecución de <number> iteraciones.

LOAD CASES.

Cargar una base de casos.

Sintaxis.

```
LOAD CASES <casebase_name>
```

Descripción.

Este comando carga una base de casos en el kernel desde un archivo en el disco. Este comando espera encontrar el archivo binario correspondiente a la base de casos <casebase_name>, si no lo encuentra, abre el archivo de texto y genera el binario.

LOAD PROJECT.

Cargar un proyecto de modelo de red neuronal.

Sintaxis.

```
LOAD PROJECT <project_name>
```

PRINT EXEC .

Imprimir hacia la salida la configuración actual del proceso de entrenamiento.

Sintaxis.

```
PRINT EXEC
```

Descripción.

Este comando imprime hacia la salida (Mssage Window) los valores de los parámetros del algoritmo de entrenamiento y la configuración general del sistema en su ejecución. Imprime la iteración actual, el error cuadrático total alcanzado por el proceso de convergencia hasta el momento, etc. Este comando es útil para dar seguimiento al entrenamiento de la red.

RESET WEIGHT.

Reiniciar los valores de los parámetros libres de la red (pesos y umbrales).

Sintaxis.

RESET WEIGHT

Descripción.

La reinicialización puede efectuarse aún en el proceso de entrenamiento. Los parámetros toman valores aleatorios generados con una distribución uniforme en el rango [-0.5, 0.5].

SAVE PROJECT.

Salvar un proyecto de diseño de red.

Sintaxis.

SAVE PROJECT <name> AS VERSION <number> TO <directory>

Descripción.

Este comando crea un directorio <directory>\<name>-<vesion> y en él salva los fichero del proyecto.

SAVE WEIGHT.

Guardar los pesos y umbrales de la red en un fichero de parámetros (.netbin).

Sintaxis.

SAVE WEIGHT TO <file_name>

SET ALFA.

Establecer el parámetro alfa (velocidad de convergencia, ver (I.7)) del algoritmo *backpropagation*

Sintaxis.

SET ALFA <real_number>

SET AUTOSAVE TO.

Activar la opción de salvado automático de los parámetros libres en el proceso de entrenamiento.

Sintaxis.

SET AUTOSAVE TO <filename> EACH <number>

Descripción.

Este comando activa el salvado automático de los parámetros libres de la red cada <number> iteraciones en el proceso de entrenamiento. La salva se realiza en el formato de fichero de parámetros (.netbin) hacia el archivo <filename>.

SET AUTOSAVE OFF.

Desactivar la opción de salvado automático de los parámetros libres en el proceso de entrenamiento.

Sintaxis.

```
SET AUTOSAVE OFF
```

SET BETA.

Establecer el parámetro beta (magnitud de la influencia de los pesos, ver (I.8)) del algoritmo *backpropagation.*

Sintaxis.

```
SET BETA <real_number>
```

SET BINARY RANGE.

Establecer el umbral para la binarización de las salidas.

Sintaxis.

```
SET BINARY RANGE <real_number>
```

Descripción.

Este parámetro es usado por las técnicas de post-procesamiento para convertir un valor de activación real a uno binario.

SET CROSSVALIDATION .

Activar la técnica *crossvalidation* aleatoria para la estimación del error total de entrenamiento y los porcientos de acierto en la clasificación.

Sintaxis.

```
SET CROSSVALIDATION TIMES <times_number> TRAIN <train_prctg>
TEST <test_prctg>
```

Decripción.

Este comando activa el *cross-validation* de manera que cada vez que se ejecute el entrenamiento, éste se realice <times_number> veces, con una organización de la base de casos distinta cada vez (siguiendo una organización aleatoria). La base de casos original es dividida en una base de entrenamiento y otra de prueba según los porcientos <train_prctg> y <test_prctg> respectivamente. Estos parámetros representan números positivos que suman 1 entre ambos.

Ejemplo.

```
SET CROSSVALIDATION TIMES 10 TRAIN 0.75 TEST 0.25
```

Este comando activa el *cross-validation* 10-fold aleatorio sobre una división de la base de casos de 75 porciento para el entrenamiento y 25 porciento para prueba.

SET CROSSVALIDATION OFF.

Desactivar el *cross-validation*.

Sintaxis.

```
SET CROSSVALIDATION OFF
```

SET ERROR.

Establecer el umbral de error utilizado como condición de parada en el algoritmo de entrenamiento.

Sintaxis.

```
DROP ERROR <real_number>
```

SET NORMALIZATION.

Activar/desactivar la normalización de las entradas a la red.

Sintaxis.

```
SET NORMALIZATION <ON|OFF>
```

Descripción.

Esta heurística se basa en la transformación del valor de los atributos de entrada hacia un nuevo rango de valores, [0,1] por defecto.

SET NORMALIZATION RANGE.

Establecer el nuevo rango de valores hacia el cual se transformarán los atributos de entrada al aplicar la normalización.

Sintaxis.

```
SET NORMALIZATION RANGE (<min_real>,<max_real>)
```

Descripción.

<min_real> y <max_real> son número reales que especifican el nuevo rango de transformación.

SET STRATEGY.

Establecer la estrategia de presentación de los ejemplos de entrenamiento a la red.

Sintaxis.

```
SET STRATEGY <sequential|uniform>
```

Descripción.

`sequential` presenta los ejemplos a la red siguiendo una estrategia secuencial, y `uniform` presenta los ejemplos siguiendo una estrategia aleatoria con distribución uniforme.

SET WTALL.

Establece winner-takes-all (WTALL) como estrategia de post-procesamiento. WTALL establece como salida de la red la clase correspondiente a la neurona de mayor activación.

Sintaxis.

```
SET WTALL <ON|OFF>
```

III.4 Pruebas de Validación

En el caso del algoritmo *backpropagation,* NEngine fue probado con bases de casos internacionales. En todos los casos se usó la función de activación sigmoide y el error cuadrático medio, así como una estrategia de entrenamiento secuencial. En el siguiente recuadro se dan los resultados.

Base de Casos	% Gener. Internac.	% Gener. SmartMlp	Capa Oculta	Alfa $x10^{-4}$	Cross-Val.	% Gener. NEngine	Iterac.
Pima	75.8	76.04	9	26	RCV 75	75.10	10000
Iris	96.0	97.37	2		RCV 75	97.14	5000
BCancer	96.3	97.66	2	29	RCV 75	97.27	10000
Heart	82.6	83.82	2	74	RCV 75	83.50	200

Todos los porcientos fueron estimados usando *random cross-validation (RCV)* con 5 ejemplos de partición (75% entrenamiento, 25% prueba) de la base de casos.

El módulo de redes recurrentes fue probado con un problema artificial[11] descrito por Schuster [92]. Este ejemplo fue especialmente concebido para demostrar la capacidad

[11] Los problemas artificiales no obedecen a una situación real, sino que involucran datos generados de manera intencional siguiendo una lógica previamente establecida)

de análisis secuencial de una red recurrente. Los datos del problema fueron obtenidos de la manera siguiente.

La entrada constituye una secuencia de números x_t obtenidos aleatoriamente siguiendo una distribución uniforme en el intervalo $[0,1]$. La secuencia de salida se obtiene siguiendo la fórmula:

$$y(t) = \frac{1}{10} \sum_{\Delta t=-10}^{-1} x(t+\Delta t) \cdot (1 - \frac{|\Delta t|}{10}) + \frac{1}{20} \sum_{\Delta t=0}^{19} x(t+\Delta t) \cdot (1 - \frac{|\Delta t|}{20})$$

Para cada elemento y_t de la secuencia de salida se tiene en cuenta su contexto (10 vecinos a la izquierda y 20 vecinos a la derecha) con una importancia que decrece con los elementos más alejados (hacia la izquierda o hacia la derecha). A partir de esta información puede definirse un problema de clasificación si binarizamos las salidas y_t asignándole 0 si $y_t <= 0.5$ y 1 en caso contrario.

Schuster en sus experimentos construye una base de un solo caso que posee 10000 elementos, y la emplea para entrenar varias arquitecturas diferentes de redes recurrentes. En nuestro experimento usaremos solo dos redes recurrentes simples (unidireccionales, ver figura): una con la información contextual viajando en sentido directo (forward) y otra en sentido inverso (backward). Construiremos dos bases de casos: la primera (Base1), siguiendo la idea de Schuster (una sola secuencia de 10000 elementos), y la segunda (Base2) formada por 100 secuencias de 500 elementos cada una.

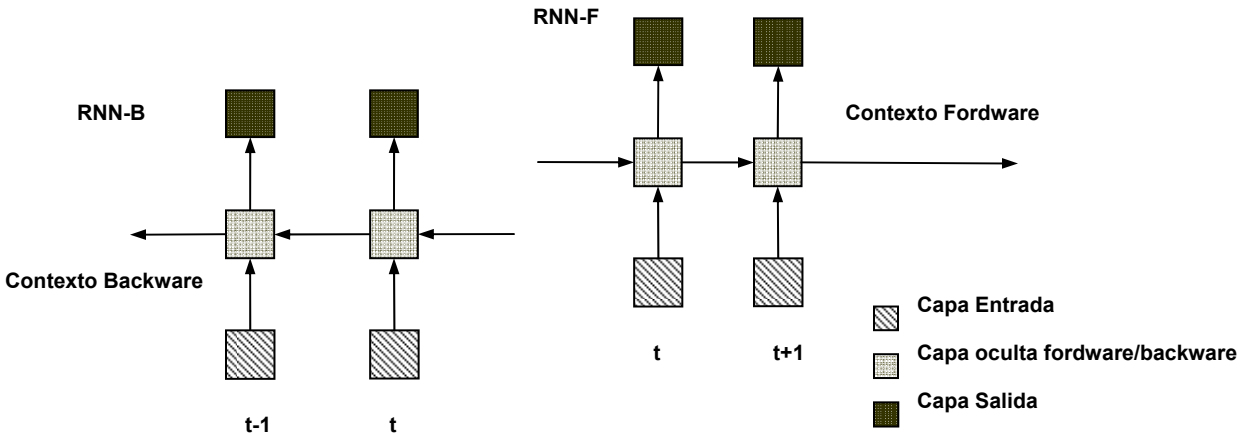


Figura 3.6 Redes Recurrentes Simples *forward y backward*.

Tanto la red recurrente directa (RNN-F) como la inversa (RNN-B) poseen una sola capa oculta, que a la vez sirve de contexto para la etapa siguiente; esta capa posee 2 neuronas. La capa de salida representa cada clase a través de una neurona. La función de activación para la capa de salida es *softmax;* como función objetivo se emplea *cross-entropy*; la función de activación de la capa oculta es la sigmoide. Los porcientos de generalización fueron estimados en todos los casos usando *random cross-validation* (RCV 75) con 5 particiones diferentes de la base de casos (75% entrenamiento, 25% prueba). En la siguiente tabla se muestran los resultados de los experimentos realizados:

Red	Base	% Train	% Test	Iterac.	% Max	Alfa	% Schuster
RNN-F	Base 1	72.67	-	200	72.67	0.0002	77.0
		77.63	-	1200	80.99	0.0002	-
	Base 2	73.82	74.45	200	81.31	0.005	-
		76.96	76.57	500	80.36	0.005	-
RNN-B	Base 1	72.67	-	200	72.67	0.0002	72.0
		72.23	-	1200	72.76	0.0002	-
	Base 2	73.92	73.47	200	74.68	0.003	-
		73.66	74.33	500	77.82	0.003	-

Los resultados de comparación disponibles fueron obtenidos por Schuster [92] (77% RNN-F, 72% RNN-B) a partir del entrenamiento de las arquitecturas mostradas usando una base de casos obtenida de la misma forma que Base1; pero con el empleo del algoritmo RPROP[12] (200 iteraciones) y siguiendo un modelo de neurona distinto para la capa oculta (usa la función tangencial (I.4)). Presumiblemente ésta es la razón por la que el experimento [RNN-F/Base1/200 iter.] arrojó un resultado inferior al esperado. Al aumentar la cantidad de iteraciones (a 1200) se puede observar un resultado comparable al obtenido por Schuster para [RNN-F/Base1]. En el caso de RNN-B no se

[12] RPROP es un algoritmo con mejores cualidades que el *backpropagation* estándar para el entrenamiento recurrente. En la literatura se reportan mejores velocidades de convergencia y porcientos de generalización si se emplea este algoritmo como base para el BPTT.

observan diferencias al elevar la cantidad de iteraciones; en ambos casos los resultados son comparables a los obtenidos por Schuster para RNN-B. Los experimentos realizados con la Base2 fueron pensados para atenuar el efecto del desvanecimiento del gradiente, pues las secuencias de entrenamiento se hacen mucho más cortas (500 elementos). Los resultados corresponden con lo esperado: para el caso de RNN-F se obtiene una mejor generalización para igual número de iteraciones (200), además de reducir (a 500) el número de iteraciones necesarias para obtener el resultado reportado por Schuster. Para el caso de RNN-B la generalización es incluso superior.

Conclusiones

- Se estudiaron diversos enfoques de modelos unificadores para redes neuronales existentes en la literatura, y se seleccionó uno cuyo fundamento se basa en diagrama de bloques. A partir de dicho modelo se ha descrito una arquitectura de software para el desarrollo de una plataforma que facilita la investigación de los especialistas en redes neuronales. La arquitectura consiste en la acción colectiva de dos procesos concurrentes: uno dedicado a la administración de los objetos de la plataforma y el otro a la ejecución de los algoritmos de entrenamiento.

- Se diseñó e implementó la herramienta NEngine para la creación, entrenamiento, validación y explotación de redes feedforward y redes recurrentes con los algoritmos de aprendizaje: Backpropagation y *Backpropagation Through Time*. Esta herramienta brinda un lenguaje que le permite al usuario crear cualquier topología, usar varias funciones de error, varias funciones de activación de la capa de salida, así como la realización de crossvalidation para la validación de la red. Se basó en una estructura de clases para la gestión de procesos concurrentes, con facilidades para su extensión a una plataforma unificadora de redes neuronales.

- Se usaron bases de casos internacionales como Iris, Pima, Heart, etc, para validar el algoritmo *backpropagation*. El algoritmo *BPTT* ha sido validado usando el problema artificial para redes recurrentes descrito por Schuster en su tesis doctoral. En el caso de *backpropagation* los resultados fueron comparables a los obtenidos por el SmartMlp para los mismos ejemplos. En todos los casos fueron superiores a los reportados en la literatura. Los resultados de los experimentos con BPTT pueden considerarse satisfactorios (atendiendo a los resultados reportados por Schuster) tomando en consideración que en nuestro caso se ha usado un *backpropagation* estándar como algoritmo base para el entrenamiento recurrente (Schuster usa RPROP como algoritmo base, lo cual significa una mejora en la velocidad de convergencia y poder de aprendizaje).

Recomendaciones

1. Ampliar el lenguaje de especificación neuronal e incorporar estructuras de información que incrementen las posibilidades del usuario, llevando el NEngine hacia una plataforma unificadora de redes neuronales basada en el modelo de framework unificador descrito en el capítulo 1.
2. Incorporar nuevas técnicas y heurísticas que ayuden a tratar el problema del desvanecimiento del gradiente y el mejoramiento de la velocidad de convergencia de los algoritmos *Backpropagation* y *Backpropagation ThroughTime*. En este sentido se recomienda el estudio del algoritmo de entrenamiento RPROP y sus variantes.
3. Implementar alguna técnica de tratamiento del *overfitting* para redes neuronales como *weight decay* o *cross-validation a través de un conjunto de validación adicional*. Se recomienda incorporar además otras técnicas como el *K-fold* y el *Leave-one-out* para la estimación de los porcientos de generalización de la red.

Bibliografía.

[1]. Atiya, Amir; Parlos Alexander, *New Results on Recurrent Networks Training: Unifying the Algorithms and Accelerating Convergence*, *IEEE Transactions on Neural Network*, 11(3):697-709, 2000.

[2]. Baldi P., Soren B. *Bioinformatics: The machine learning Approach*. 2da.Edición. MIT Press. 2001.

[3]. Baldi P. *New Machine Learning Methods for the Prediction of Protein Topologies,* Universidad de Firenze, Italia, 2002.

[4]. Baltersee J., Chambers, J. *Non-linear adaptive prediction of speech signals using a pipelined recurrent network*. IEEE Trans. on Signal Proc. 1998.

[5]. Battiti, R., F. Masulli. *BFGS Optimization for Faster and Automated Supervised Learning*, INCC Paris, International Neural Network Conference, pp. 757–760, 1990.

[6]. Bengio, Y. *Learning Long-Term dependencies with Gradient descent is difficult*, *IEEE Transactions on Neural Network,* 5(2):157-166, 1994.

[7]. Benson, M., Carrasco R. A. *Application of a RNN to space diversity in SDMA and CDMA mobile communication systems*. Neur. Comp. and Appl., 10:136–147, 2001.

[8]. Bishop, C.M. *Neural Networks for pattern recognition.* Oxford University Press, 1995.

[9]. Boden, Mikael. *A Guide to Recurrent Neural Networks and Backpropagation.* Halmstad University, 2001.

[10]. Bonet, I. etal. *Learning optimization in a MLP Neural Network Applied to OCR*. MICAI 2002: Advances in Artificial Intelligence. LNAI 2313. pp 292-300.

[11]. Cachin, C. *Pedagogical Pattern Selection Strategies*. Neural Networks, Vol. 7, No. 1, pp. 175-18. 1994.

[12]. Campbell, C. *Constructives Learning Techniques for Designing Neural Netwoks Systems.* Bristol University, 1997.

[13]. Cho Siu-Yeung. *Efficient Learning in Adaptive Processing of Data Struictures.* Neural Processing Letters, 17, pp. 175-190. 2003.

[14]. Díaz, A. *Heurísticas para el diseño de redes neuronales tipo Multilayer Perceptron (MLP) que utilizan Backpropagation (BP) como algoritmo de aprendizaje*. Trabajo por el grado de master. 2000.

[15]. Electronic Project Manager. http://SourceForge.net/

[16]. Fahlman S.E., Lebiere C. *The cascade-correlation learning architecture.* Advances in Neural Processing Systems. Vol. 2 pp.524-532, 1990.

[17]. Falcon, R. *NeuroDeveloper, una herramienta para el desarrollo de sistemas conexionistas en presencia de rasgos difusos.* Tesis de Grado, 2003.

[18]. Frank, R., Mathis D. . The Acquisition of Anaphora by Simple Recurrent Networks. John Hopskin University. 2004.

[19]. Giannakakis, G. *Highlights of signal processing for communication*. Signal Proc. Magazine, 16:pp.14–50, 1999.

[20]. Giles C., Lawrence S. *Rule inference for financial prediction using recurrent neural networks*. Proceedings of IEEE/IAFE Conference on Computational Intelligence for Financial Engineering: pp.253-259. 1997.

[21]. Giles C., Lawrence S. *Noisy Time Series Prediction using Symbolic Representation and Recurrent Neural Network Grammatical Inference*. Proceedings of IEEE/IAFE Conference on Computational Intelligence for Financial Engineering: 1997.

[22]. Hammer, B. *Recurrent networks for structured data – a unifying approach and its properties*. Cognitive Systems Research 3(2), pp. 145-165, 2002.

[23]. Hammer, B., Steil, J. *Tutorial: Perspectivas on Learning with RNNs.* European Symposium on Artificial Neural Networks'2002, d-side publi, pp. 357-368. 2002.

[24]. Hammer B., Villmann T. *Mathematical Aspects of Neural Networks*. European Symposium on Artificial Neural Networks'2003, d-side public, pp.59-72. 2003.

[25]. Hilera J. Martínez V. *Redes Neuronales Artificiales: Fundamentos, modelos y aplicaciones*. Addison-Wesley, 1995.

[26]. Igel, C. Husken, M. *Empirical Evaluation of Improved RPROP. Learning Algorithms.* Neurocomputing, 50, pp.105–123, 2003.

[27]. Institute of Parallel and Distributed High-Performance Systems. http://www.informatik.uni-stuttgart.de/ipvr/

[28]. Jacobs R.A. *Increased rate of convergences throughlearning rate adaptation.* Neural Networks. Vol. 1 pp. 295-307, 1998.

[29]. Kim, H. B, et al. *Fast Learning Method for Backpropagation Neural Network by Evolutionary Adaptation of Learning Rates*. Neurocomputing, Vol. 11, No, 1, pp.101-106, 1996.

[30]. Kock, G. et al. Neurosolution: Integrated hardware and software for the development of neural applications. *System Analysis, Modeling, Simulation (SAMS),* 1998.

[31]. Kohonen, T. *Self-Organizing Maps of Symbol Strings.* Technical Report Lab. CISC, Nro A42, 1996.

[32]. Kolb, T. etal. *Using time-discrete RNNs in nonlinear control*. Proceeding. of the World Congress on Comp. Int. '98, pages 1367–1371, 1998.

[33]. Kosko, B. *Bidirectional Associatives Memories.* IEEE Transactions on Systems, Nro 18. pp.42-60, 1988.

[34]. Lawrence, S., Giles, L. *Natural Language Gramatical Inference with Recurrent Neural Networks.* IEEE Transaction on Knowledge and Data Engineering, Vol.12, Nr. 1, 2000.

[35]. Lipmann, R. *An Introduction to Computing with Neural Nets*. IEEE ASSP Magazine, Vol 4, Nro. 2, 1987, pp. 4-22.

[36]. Leighton,R.R, *The Aspirin / MIGRAINES Neural Network Software:User's Manual.* MITRE Corporation, 1992. *http://yoda.cis.temple.edu:8080/aspirin /doc/manual.ps*

[37]. Lucas, S. *Forward-Backward building blocks for evolving neural networks with intrinsic learning behaviours. LNCS, Vol* 1240:723-732, 1997

[38]. Martin, D. L., Cheyer A. J., Moran, D. B. *The open agent architecture: A framework for building distributed software systems.* Applied Artificial Intelligence, 13(1-2):pp. 91–128, 1999.

[39]. MathWork Inc. http://www.mathwork.com

[40]. Mayer, H. *Basic Object Oriented Neuronal Engine.* Tech Report, http://sf.gds.tuwien.ac.at, 2004.

[41]. Moller, M. *A Conjugated Gradient Algorithm for Fast Supervised Learning.* 1990

[42]. Nattkemper, W., etal. *A neural network architecture for automatic segmentation of fluorescence micrographs*. Proceeding. of the 8th Europ. Symp. on Art. Neur. Netw., pp. 177–182 2000.

[43]. Neural Dimension Inc. http://www.nd.com/

[44]. O'Reilly, R., Munakata, Y. *Computational Exploration in Cognitive Neuroscience: Understanding the Mind by Simulating the Brain.* MIT Press, 2000.

[45]. Parlos A. G., Menon S. K., Atiya A. F. *An algorithmic approach to adaptive state filtering using recurrent neural networks*. IEEE Trans. Neural Networks, 12(6):pp.1411–1432, 2001.

[46]. Pearlmutter B. *Dynamic Recurrent Neural Networks.* DARPA Research, 1990.

[47]. Qian, N, Sejnowski, T. *Predicting the secondary structure of globular proteins using neural network models* J. Mol. Biol, pp865-884, 1988.

[48]. Reza, A., Alamdari, A. *Biological Neural Networks: A toolbox for Matlab.* http://www.yner.org , 2004.

[49]. Riedmiller, M. Braun, H. *A direct adaptive method for faster backpropagation learning: The RPROP algorithm.* Proceedings of the IEEE International Conference on Neural Networks, pp. 586-591, 1993.

[50]. Ruiz, A., etal. *Existence, learning, and replication of periodic motions in recurrent neural networks.* IEEE Transactions on Neural Networks, 9:651–661, 1998.

[51]. Rumelhart, D. E., etal. *Learning internal representations by error backpropagation* . Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1 pp.318-362. MIT Press, 1986.

[52]. Sadosky, D. *Client/Server Software Architectures - An Overview.* C. Mellon University. 1997.

[53]. Saito, K., Nakano, R. *Second-Order Learning Algorithm with Squared Penalty Term.* Neural Computation, 12: pp. 709-729, 2000.

[54]. Schmidhuber, J., Gers, F., Eck, D. *Learning nonregular languages: A comparison of simple recurrent networks and LSTM*. Neural Computation, 14(9): pp.2039-2041, 2002.

[55]. Schuster, M. *On Supervised Learning from Sequential Data with application for Speech Recognition*. NIST, 1999.

[56]. Somervuo, P., Kohonen, T. *Self-Organizing Maps and Learning Vector Quantization for feature sequences.* Neural Processing Letters, 10(2), pp.151-159, 1999.

[57]. Sona, Diego. *PhD Tesis, A proposal for an Abstract Neural Machine*, Pisa University, 2002.

[58]. Starita, A., Sperduti, A. *Supervised neural networks for the classification of structures.* IEEE Transactions on Neural Networks, 8(3):714-735. 1997.

[59]. Tin-Yau Kwok, Dit-Yan Yeung. *Objective Functions for Training New Hidden Units in Constructive Neural Networks.* Transactions on Neural Networks, Vol. 20, 1999.

[60]. Tin-Yau Kwok, Dit-Yan Yeung. *Constructive Algorithms for Structure Learning in Feedforward Neural Networks for Regression Problems.* Transactions on Neural Networks. 1997.

[61]. Togneri, R. etal. *A Comparison of the LBG, LVQ, SOM and GMM Algorithms for Vector Quantization Clustering Analysis.* Proc. Fourth Australian International Conference on Speech Science and Technology*, pp. 173-177, 1992.*

[62]. Tollenaere, T. *Supersab: Fast adaptive backpropagation with good scaling properties.* Neural Networks, 3(5), 1990.

[63]. Topi, L., Parisi, R. *Spline Recurrent Neural Networks for Quad-Tree Video Coding*, Proceedings of the XIII Italian Workshop on Neural Nets, 2002.

[64]. Trhun S. *A General Feed-Forward Algorithm for Gradient Descent in Connectionist Networks,* Germany National Research Center, 1990.

[65]. Tsei, A Gori, M. *On the Problem of Local Mínima in Backpropagation*. IEEE Computer Society, 1992.

[66]. Tsoi, A. *Recurent neural network architectures: An overview*. *LNCS, Vol 1387:1-26,* 1998.

[67]. Tsoi,A., Back,A. *Discrete time recurrent neural network architectures: A unifying review,* Neurocomputing*,* 15,(3,4):183-223, 1997.

[68]. Wan, E. Beaufays. *Diagrammatic methods for deriving and relating temporal neural network algorithms. LNCS, Vol* 1387:63-98, 1998*.*

[69]. Weilan W., Stan C. *Identifying Language from Raw Speech — An Application of Recurrent Neural Networks*. 5th Midwest Artificial Intelligence and Cognitive Science Conference, 1993.

[70]. Werbos, P. J. *Backpropagation ThroughTime: What it does and How to do it.* Proceedings of IEEE, Vol. 78. Nro. 10. 1990.

[71]. Witten, I., Frank, E. *DataMining: Practical Machine Learning Tools and Techniques with Java Implementation*. Morgan Kauffman Publisher, 2000.

[72]. Wolfgang, E. *Engineering Distributed Objects.* Wiley Editions. 2000.

[73]. Yu, X., Chen, G., Cheng, S. *Dynamic Learning Rate Optimization of the Backpropagation Algorithm*. IEEE Transaction on Neural Networks, Vol. 6, Nr. 3, pp. 669-677, 1995.

[74]. Zell, A. Mamier G., *et al*, *SNSS. Stuttgart Neural Network Simulator. User Manual, Version 4.1,* 1995.

[75]. Ziemke, Boden, M. *Oil spill detection: A case study of recurrent artificial neural networks.* Neural Network Analysis, Architectures and Appl. IOP Press, 1997.

[76]. Ziemke,T. *Remembering how to behave: RNNs for adaptive robot behavior*. Recurrent Neural Networks, Design and Appl., pp. 355–389. CRC Press, 2000.

[77]. Zipser, David; William, Ronald, "*Gradient-Based Learning Algorithm for RNN and their Computation Complexity"*: *BackPropagation: Theory, Architectures and Applications,* Y.Chauvin & Rumelhart Editions, Hillsdale, 1995.

[78]. Zurada, Jacek M. *Introduction to artificial neural systems*. West Publishing Company. 1992.

Anexo 1. Un Framework Basado en Diagramas de Bloques.

En el presente anexo se explican más detalles sobre el framework unificador, particularmente el Esquema del Proceso de Ejecución y el Proceso de Administración de Objetos.

1.1. Esquema del Proceso de Ejecución.

El proceso de *ejecución* contempla las etapas de entrenamiento y explotación de la red neuronal; esto está, en esencia, soportado sobre el carácter dual del modelo de unidad computacional tratado en el acápite anterior (una dirección *forward* asociada a los procesos de evaluación de los casos en la red o procesos de explotación, y otra *backward* asociada a los procesos de entrenamiento), que es capaz de simular de manera homogénea los algoritmos de explotación y entrenamiento de la red. Este aspecto es más evidente en redes que basan su funcionamiento sobre el modelo de redes *backpropagation,* donde existe de manera clara un proceso directo (cálculo de las salidas dada una entrada particular) y un proceso inverso (propagación del error aportado por cada conexión de la red).

En correspondencia con esto, el flujo de procesamiento prevé un tratamiento homogéneo para la ejecución directa e inversa basado en la redefinición de *operadores abstractos* aportados en el *programa* de la MVN. Un diagrama abstracto del proceso general puede observarse en la figura A1.1

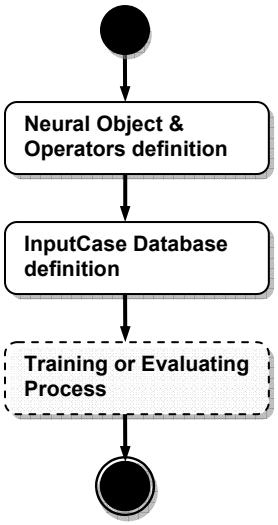


Figura A1.1. Esquema abstracto del proceso de ejecución.

El primer paso (*neural object & operador definition*) junto al segundo (*Inputcase database definition*) establecen las fases iniciales del esquema y el marco de trabajo necesario para la (re)definición de los objetos neuronales y los *operadores* involucrados en el establecimiento del modelo de red a ejecutar. El último paso determina el esquema de ejecución específico para el modelo que se construye y el tipo de procesamiento a realizar (entrenamiento o explotación). Los bloques con contorno punteado representan fases abstractas (*operadores*) susceptibles de ser redefinidas por el usuario en las etapas previas de (re)definición. Un proceso típico de evaluación o entrenamiento para redes tipo *backpropagation* (o basadas en el mismo principio) puede redefinirse como se muestra en la figura A1.2.

Este esquema consiste sencillamente en la presentación a la red de los casos para su asimilación en un orden determinado. El operador *StopCondition* establece el estado bajo el cual puede considerarse la detención del proceso. *ExecuteInputCase* encierra la estructura esencial de la ejecución de las unidades de cómputo que conforman la red y puede refinarse aún más bajo el esquema de la figura A1.3. Tanto *StopCondition*, *BuildFirstInputCase* como *BuildNextInputCase* constituyen operadores redefinibles según el tipo de proceso particular (entrenamiento, explotación) y la forma de presentación de los casos (secuencial, aleatoria, etc.).

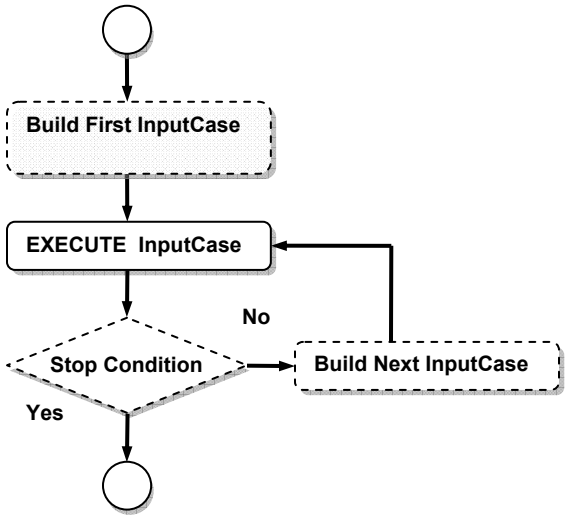


Figura A1.2. Esquema abstracto del proceso de entrenamiento para redes backpropagation.

La redefinición de estos operadores depende del proceso que se esté llevando a cabo: entrenamiento o explotación y de las diferentes heurísticas concebibles para aumentar la eficiencia en los algoritmos de entrenamiento. Por ejemplo, en el proceso de explotación la condición de parada podría estar determinada a priori por la cantidad de casos de entrada que se van a calcular en la red; contrariamente, en el proceso de entrenamiento debe comportarse como una función de los parámetros óptimos establecidos para la red que se está creando (digamos el error medio de evaluación de todos los casos examinados en la base de casos de entrenamiento). Adicionalmente, en el proceso de entrenamiento debe tenerse en cuenta la ejecución de un caso en las dos direcciones: la dirección directa (*forward)* para calcular las salidas de la red dada una entrada particular, y la inversa (*backward*), para la propagación del error. En este caso una definición adecuada del operador de construcción de la próxima entrada (*BuildNextInputCase*), es la que garantiza que no se presenta un nuevo caso si no se ha preparado la ejecución del anterior en ambas direcciones. En realidad pueden establecerse diversos esquemas de ejecución en dependencia del patrón de conexión de cada unidad computacional y de los algoritmos de entrenamiento específicos existentes. En [4] pueden encontrase refinamientos específicos para modelos de redes recurrentes, modelos *constructivos* y redes estocásticas (*Boltzman Machine, etc.*) sobre un esquema similar al presentado aquí.

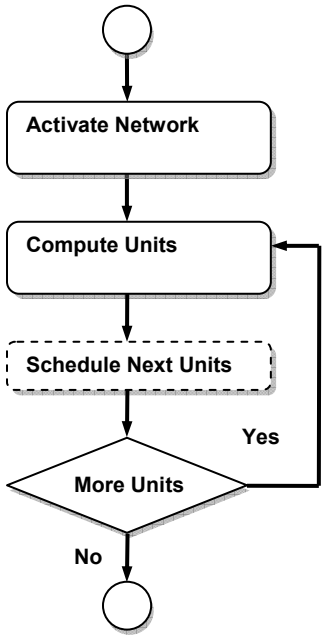


Figura A1.3. Refinamiento del paso *Execute InputCase*.

La figura A1.3 muestra el núcleo del esquema de procesamiento. La primera etapa *ActivateNetwork* inicia la red para un nuevo procesamiento, estableciendo las unidades computacionales que serán ejecutadas primero y los parámetros de ajuste pertinentes. *ComputeUnits* es la etapa prevista para la ejecución de las unidades computacionales que conforman la red en un orden establecido por el operador *ScheduleNextUnits;* toma las unidades computacionales previamente planificadas y ejecuta su función de transición de estados y función de salida (*operadores* que refinan el funcionamiento de cada unidad computacional: el cálculo de las salidas a partir de las entradas y el estado de activación actual de la unidad), garantizando la ejecución de la red entera y la propagación de los valores de salida según la dirección de entrenamiento especificada en la entrada (*InputCase*) particular. La figura A1.4 muestra un posible refinamiento de la etapa *ComputeUnits.*

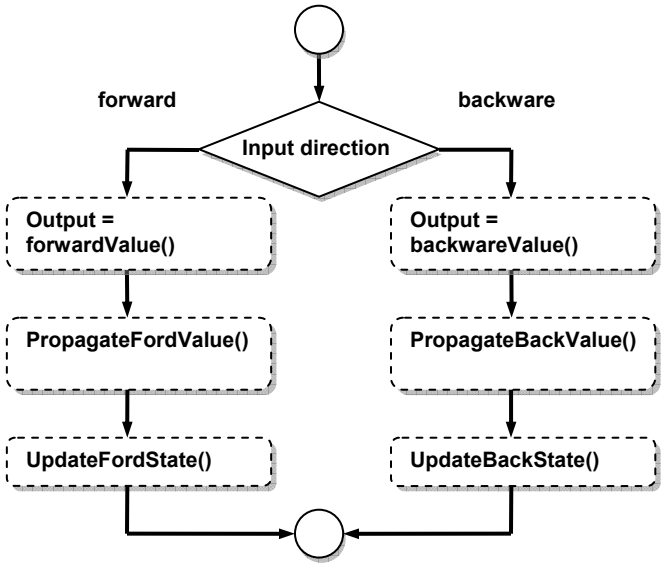


Figura A1.4. Refinamiento del paso *ComputeUnit* con dos direcciones de cómputo

Los operadores *forwardValue y backwardValue* corresponden a las funciones de salida f_f y f_b en definidas en (I.16) y (I.17) respectivamente, y constituyen las funciones que calculan la salida de la unidad computacional de acuerdo a las entradas a dicha unidad y su estado actual. Igualmente *UpdateFordState y UpdateBackState* corresponden a las funciones de transición de estados g_f y g_b que garantizan el cambio de estado de las unidades computacionales cada vez que estas son ejecutadas.

PropagateFordValue y *PropagateBackValue* son operadores establecidos para lograr la conexión entre las unidades computacionales de la red y la propagación de los valores de salida de cada una de ellas a las demás. Pueden establecerse versiones especiales de estos dos últimos operadores para dar un tratamiento adecuado a las neuronas de las capas de salida y entrada.

Finalmente, el operador *ScheduleNextUnits* determina cuáles son las próximas unidades computacionales en ejecución y constituye otro elemento esencial en el modelo unificador. Básicamente especifica una relación de orden entre conjuntos de unidades computacionales donde cada conjunto representa unidades que pueden ser ejecutadas simultáneamente. La figura A1.5 muestra el orden de ejecución de las unidades computacionales en una red *backpropagation* sencilla. El operador actúa como una función capaz de brindar como resultado el próximo conjunto de unidades computacionales a partir del conjunto actual.

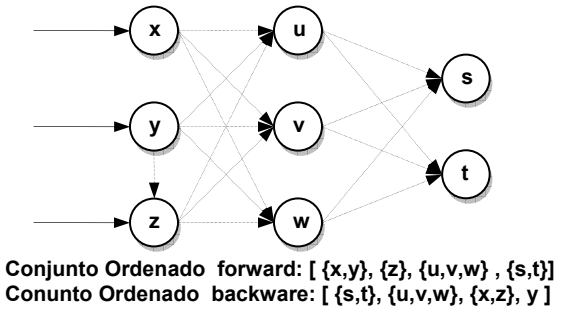


Figura A1.5. Planificación de una red *backpropagation*. (Tomada de [4])

1.2. El Proceso de Administración de Objetos.

El proceso de administración de objetos constituye un gestor de información asociada a los modelos de red que se encuentran en ejecución en la MVN. Esta información debe estar organizada a partir de un sistema de clases de objetos (estructuras encaminadas al almacenamiento de datos) referidos a los *operadores* del esquema de ejecución de los algoritmos de entrenamiento y explotación, la información de estado de cada unidad computacional (el modelo de neurona de la red), las unidades computacionales (el tipo y la cantidad) y el esquema de conexión (la topología) de la red, así como la información adicional pertinente. La Base de Datos de Objetos (*Object Repository*) se

podría estructurar a partir de tablas capaces de almacenar *clases* de objetos *predefinidos* o definidos por el modelador, los objetos propiamente y la agrupación de dichos objetos en modelos.

La unidad de trabajo fundamental del proceso de administración es el *modelo*. Entendiendo éste como la colección o agrupación de objetos relativos a la definición de la información de un modelo de red neuronal totalmente nuevo o la variación de un modelo existente. Cada objeto debe pertenecer a una *clase* determinada (predefinida o definida por el usuario). Entre las *clases* predefinidas podrían aparecer: *ComputationalUnitModel, Operator, Connection, Unit, UnitCollection, Model, ConnectionCollection,* etc.

La clase *ComputationalUnitModel* se refiere al tipo de objeto que recoge la información sobre la definición de nuevos tipos de unidades computacionales y sus operadores (de manera que es una agregación de objetos de tipo *Operator* y otras clases primitivas). *Connection* agruparía a los objetos destinados a almacenar información sobre las conexiones del la red particular (topología). *Unit* recogería igualmente información sobre la topología de la red; mientras *Model* podría actuar como clase maestra para recoger toda la información sobre cada modelo. La clase *Operator* agrupa los objetos concebidos para almacenar el código o conjunto de instrucciones capaces de *enlazarse* al esquema de ejecución discutido en el epígrafe anterior y formar los algoritmos de entrenamiento y evaluación de los nuevos tipos de red que puedan crearse. La clase *Operator* podría especializarse a partir de la definición de clases de operadores más específicos como *ScheduleNextUnits, BuildNextInputCase, forwardValue,* etc. En la figura A1.6 se muestra una organización de la información manejada por el proceso de administración de objetos.

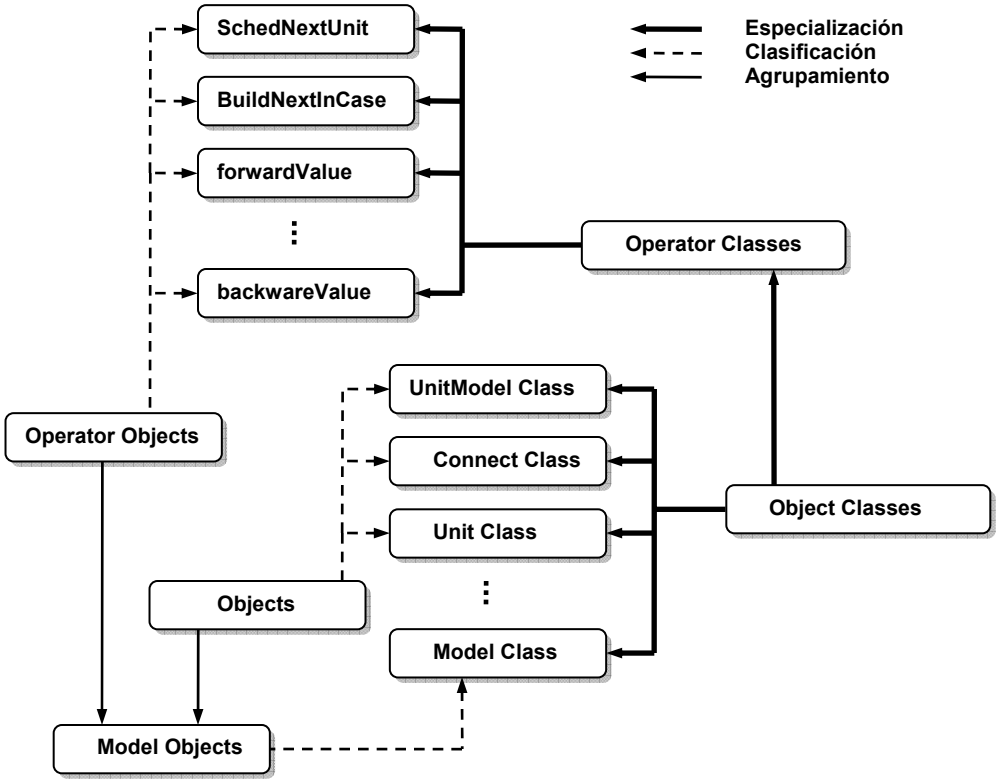


Figura A1.6. Organización general del Repositorio de Objetos.

Anexo 2. Extensión de la Estructura de Clases para Procesos Concurrentes.

La estructura de clases descrita en el Capítulo II está implementada en tres módulos (*Units*) en Object Pascal:

`stream.pas`, incorpora las clases para el trabajo con búferes y secuencias de bytes `tcBuffer`, `tlBuffer`.

ComInfrastructure.pas, incorpora la clase para gestión de mensajes y clases de mensajes `tMsgFactory`, y `tchannelLinker` para crear canales de comunicación.

`SchedInfrastructure.pas` agrupa la clase para el sincronismo de las tareas `tOperGroup` (`tETokenGroup`).

El reuso de estas tareas puede realizarse básicamente de dos formas distintas: la primera, mediante la incorporación de una nueva tarea; la segunda mediante la adición de nuevos mensajes y sus controladores (*handlers*) a una tarea existentes. En este anexo se describe cómo codificar estas dos formas de extensión tomando un pequeño ejemplo de tratamiento de cadenas.

El problema que queremos resolver es como sigue: dos procesos A y B reciben continuamente mensajes consistentes de cadenas S de caracteres, de longitud variable; los dos deben depositar la cadena recibida en un mismo búfer circular, después de realizar cierta operación sobre la misma: el proceso A debe colocar la cadena con todos sus caracteres en mayúscula y B con todos sus caracteres en minúscula. El búfer circular acepta la adición. Una vez que se llena el búfer circular, éste debe vaciarse y para ello la tarea A recibe un mensaje determinado.

Suponiendo que la tarea C hace las veces de tarea controladora, el problema puede resolverse si concebimos un par de mensajes (`MSG_A_REC_STR` y `MSG_B_REC_STR`) para la recepción de las cadenas por parte de cada una de los procesos A y B. Como el búfer circular es el mismo para los dos procesos, es necesario establecer una relación de exclusión entre estas dos clases de mensajes. De esta forma las dos tareas no podrán tributar a la misma vez en el búfer circular, evitándose la confusión de los caracteres entre una cadena y otra. La tarea A implementa un mensaje (`MSG_DUMP`)

para vaciar el búfer. Este mensaje también deberá compartir una relación de exclusión con el mensaje `MSG_B_REC_STR`, garantizando que no comience el vaciado hasta se haya completado la incorporación de una cadena por parte del proceso B, igualmente el proceso B no podrá incorporar una cadena cuando un vaciado esté ocurriendo, evitando así inconsistencias. Este ejemplo es muy sencillo, pero ilustra los conceptos esenciales implementados en la estructura de clases.

Para crear la clase de estos mensajes y la clase de los permisos (execution token) de ejecución se usa el módulo `ComInfrastructure` y `SchedInfrastructure` de la siguiente forma:

```
Unit Messages;

Interface
Uses ComInfrastructure, SchedInfrastructure;

// Message class constants
Const MSG_DUMP =0;
Const MSG_A_RECEIVE_STRING =5;
Const MSG_B_RECEIVE_STRING =6;

// Message execution token class
Const TK_A_ACCESS =0;
Const TK_B_ACCESS =1;

//Message channels
Const CHANNEL_ATASK = 0;
Const CHANNEL_BTASK = 1;
Const CHANNEL_CTASK = 2;

Var msgFactory  :tMsgFactory;
    Syncro     :tOprGroup;
    channels    :tChannelLinkers
Implementation
Var classID,operID:longword;

Initialization
msgFactory:= tMsgFactory.Create;
Syncro:=TOprGroup.Create;
//defining message class
msgFactory.addclass('MSG_DUMP',classID)
msgFactory.addclass('MSG_A_REC_STR',true,CLS_BYTE,256,classID);
msgFactory.addclass('MSG_B_REC_STR',true,CLS_BYTE,256,classID);
//defining execution token class
Syncro.addoperator(1, operID);
Syncro.addoprExclu(operID,TK_B_ACCES);
Syncro.addoperator(1, operID);
```

```
Syncro.addoprExclu(operID,TK_A_ACCES);
//defining channels
Channels:=tChannelLinkers.create;
Channels.addChannel(3*1024,1024,chnID);     //ATASK
Channels.addChannel(3*1024,1024,chnID);     //BTASK
Channels.addChannel(3*1024,1024,chnID);     //CTASK
End.

Unit tasks;
Interface
Uses Messages, SchedInfrastructure;

type tATask = class(tTask)
                protected
                     procedure task_initialization; override;
                     procedure task_finalization; override;
                     procedure execute; override;
                     end;

          tBTask = class(tTask)
                     protected
                     procedure task_initialization; override;
                     procedure task_finalization; override;
                     procedure execute; override;
                     end;

          tBTask = class(tTask)
                     protected
                     procedure task_initialization; override;
                     procedure task_finalization; override;
                     procedure execute; override;
                     end;

var     buffer : tcBuffer;
        aTask  : tATask;
        bTask  : tBTask;
        cTask  : tCTask;

implememtation
procedure tATask.task_initialization
var
begin
end;

procedure tATask.task_finalization
begin
end;

procedure tBTask.task_initialization
begin
end;
```

```
procedure tBTask.task_finalization
begin
end;

procedure tCTask.task_initialization
begin
end;

procedure tCTask.task_finalization
begin
end;

procedure tATask.execute
var     head    :tmsgHead;
        nextMsg :tmsgRef;

        i       :longword;
        str     :string;
label   start,
        next;

begin
        nextMsg.memory:=memory;
        nextMsg.address:=0;
start:
        task_initialization;
        scheduler.blockInitSignal.WaitFor(OPER_PARAM_WAITFOR);

while not sycro.shutdown do
 begin

 if GetNextMessage(nextMsg,head)=RESTART then
   goto start;

 case  head.classID of
    MSG_AREC_STR:
       begin
       if syncro.getOper(TK_A_ACCESS)= RESTART then
         goto start;
      msgFactory.getproperty(nextMsg.memory, 'VALUE', str,
      nextMsg.address);
      if str = 'ERROR' then
       begin
       if syncro.releaseOper(TK_A_ACCESS)=RESTART then
         goto start;
       end;

       for i:=1 to length(str) do
        buffer.addByte(str[i])

      if syncro.releaseOper(TK_A_ACCESS)=RESTART then
```

```
        goto start;
       end;

    MSG_DUMP:
       begin
        if syncro.getOper(OPER_BPTT_EXEC)= RESTART then
          goto start;
        dump(buffer);
       buffer.flush;
       if syncro.releaseOper(OPER_BPTT_EXEC)=RESTART then
          goto start;
        end;

  end; //endcase

 next:
 end;

task_finalization;
end;

procedure tBTask.execute
var     head    :tmsgHead;
        nextMsg :tmsgRef;
        i       :longword;
        str     :string;
label   start,
        next;

begin
        nextMsg.memory:=memory;
        nextMsg.address:=0;
start:
        task_initialization;
        syncro.blockInitSignal.WaitFor(OPER_PARAM_WAITFOR);

while not syncro.shutdown do
 begin

 if GetNextMessage(nextMsg,head)=RESTART then
   goto start;

 case  head.classID of
    MSG_B_REC_STR:
       begin
       if syncro.getOper(TK_B_ACCESS)= RESTART then
         goto start;
      msgFactory.getproperty(nextMsg.memory, 'VALUE', str,
      nextMsg.address);
      if str = 'ERROR' then
       begin
       if syncro.releaseOper(TK_B_ACCESS)=RESTART then
```

```
          goto start;
         end;

        for i:=1 to length(str) do
          tcbuffer.addByte(str[i])

       if syncro.releaseOper(TK_B_ACCESS)=RESTART then
        goto start;
       end;


  end; //endcase

 next:
 end;

task_finalization;
end;

procedure tCTask.execute
var     head    :tmsgHead;
        nextMsg :tmsgRef;
label   start,
        next;

begin
        nextMsg.memory:=memory;
        nextMsg.address:=0;
start:
        task_initialization;
        syncro.blockInitSignal.WaitFor(OPER_PARAM_WAITFOR);

while not syncro.shutdown do
 begin

 if GetNextMessage(nextMsg,head)=RESTART then
   goto start;

 case  head.classID of
    MSG_A_REC_STR:
       begin
       if chnLinker.putMsg(CHANNEL_ATASK, nextMsg.memory,
       nextMsg.address, DELAY) = LNK_OK then
       syncro.mountOper(TK_A_ACCESS)
       end;

    MSG_B_REC_STR:
       begin
       if chnLinker.putMsg(CHANNEL_BTASK, nextMsg.memory,
       nextMsg.address, DELAY) = LNK_OK then
       syncro.mountOper(TK_B_ACCESS)
       end;
```

```
    MSG_DUMP:
       begin
       if chnLinker.putMsg(CHANNEL_ATASK, nextMsg.memory,
       nextMsg.address, DELAY) = LNK_OK then
       syncro.mountOper(TK_A_ACCESS)
       end;

  end; //endcase

 next:
 end;

task_finalization;
end;

initialization
buffer := tcBuffer.create(3*1024,1024);
aTask  := tATask.create(false);
bTask  := tBTask.create(false);
cTask  := tCTask.create(fasle);
end;
```

El ejemplo está dividido en dos módulos `messages` y `tasks`. El primero contiene las definiciones de clases de mensajes, tipos de permiso de acceso y los canales para la comunicación interproceso. El segundo posee las definiciones de los ciclos de trabajo para cada tarea. La instrucción `msgFactory.addclass('MSG_DUMP',classID)` registra la clase de mensajes que usará la tarea A para la descarga del búfer, `msgFactory.addclass('MSG_A_REC_STR',true,CLS_BYTE,256,classID)` adiciona la clase que contendrá la cadena de caracteres. Los parámetros con los que se llama a esta rutina indican el registro de una clase colección de bytes (`CLS_BYTE` es un tipo primitivo) con un límite superior de 256 elementos. `Syncro.addoperator(1,operID)` se emplea para adicionar un nuevo permisos de ejecución, el argumento de valor 1 no tiene relevancia, pero es obligatorio en esta versión. `Syncro.addoprExclu(operID,TK_B_ACCES)` registra las relaciones de exclusión. `Channels.addChannel(3*1024,1024,chnID)` Adiciona los canales de comunicación; nótese que las constantes coinciden con el orden de los canales ingresados, es importante que en la definición de las estructuras este orden siempre se conserve. El canal posee una capacidad de 3kb e inicia su trabajo con 1kb.

En el módulo `tasks` la tarea `tCTask` se encarga de implementar el ciclo de trabajo controlador. Las cuatro primeras líneas de la rutina de ejecución (`execute`) constituyen la inicialización de la tarea. Seguidamente aparece el ciclo del manipuladores de mensaje, cuya primera instrucción es `GetNextMessage(nextMsg,head)` dirigida a obtener el próximo mensaje en cola. Luego una sentencia `case` determina cuál es el manipulador correspondiente a la clase del mensaje recuperado. La instrucción:

`putMsg(CHANNEL_ATASK, nextMsg.memory, nextMsg.address, DELAY)` reenvía este mensaje hacia el canal de la tarea A; y a continuación se solicita el permiso de ejecución para esta tarea `syncro.mountOper(TK_A_ACCESS)`. La tarea A antes de manipular el mensaje pide el permiso que ya debe haber sido solicitado por la tarea controladora, la instrucción que usa es `syncro.getOper(TK_A_ACCESS)`.

Esta instrucción bloquea la ejecución de la tarea A hasta que sean garantizadas las condiciones de ejecución requeridas para el permiso `TK_A_ACCESS`. Una vez que se ha manipulado el mensaje el permiso debe cancelarse para indicar que no se usará más, dando oportunidad a los demás manipuladores de ejecutar sus instrucciones. Esto se realiza con la instrucción `syncro.releaseOper(TK_B_ACCESS)`. Es importante después de cada *get* realizar el *release* correspondiente o el sistema podría entrar en un bloqueo (*deadlock).* La instrucción:

```
msgFactory.getproperty(nextMsg.memory, 'VALUE', str,nextMsg.address);
```

permite obtener la cadena de caracteres contenida en el mensaje procesado.

Anexo 3. Sitios Web de Interés.

A continuación mostramos un conjunto de direcciones electrónicas donde puede encontrarse información interesante sobre los temas tratados en el presente trabajo.

Ingeniería de Software

www.omg.org/uml/

www.enteract.com/~bradapp/docs/patterns-intro.html

Working Group. www.incose.org/tools/

http://www.rational.com

www.augustana.ab.ca/~mohrj/courses/2000.winter/csc220/papers/rup_best_practices/rup_bestpractices.html

http://www.ai.sri.com/~cheyer/papers/aai/oaa.html

Redes Neuronales Recurrentes

http://www.bcs.rochester.edu/people/robbie/robbie.html#publications

http://gunther.smeal.psu.edu/9077.html

http://www.informatik.uni-osnabrueck.de/barbara/papers/pub_hammer.html

http://citeseer.ist.psu.edu/cis?q=P+Baldi&submit=Search+Documents&cs=1

http://gruyere.ucd.ie/~gianluca/papers.html

http://www.dsi.unifi.it/~paolo/papers.html

http://ipnweb.in2p3.fr/~lptms/membres/mezard/lipub.ps

http://www.enm.bris.ac.uk/research/reports/

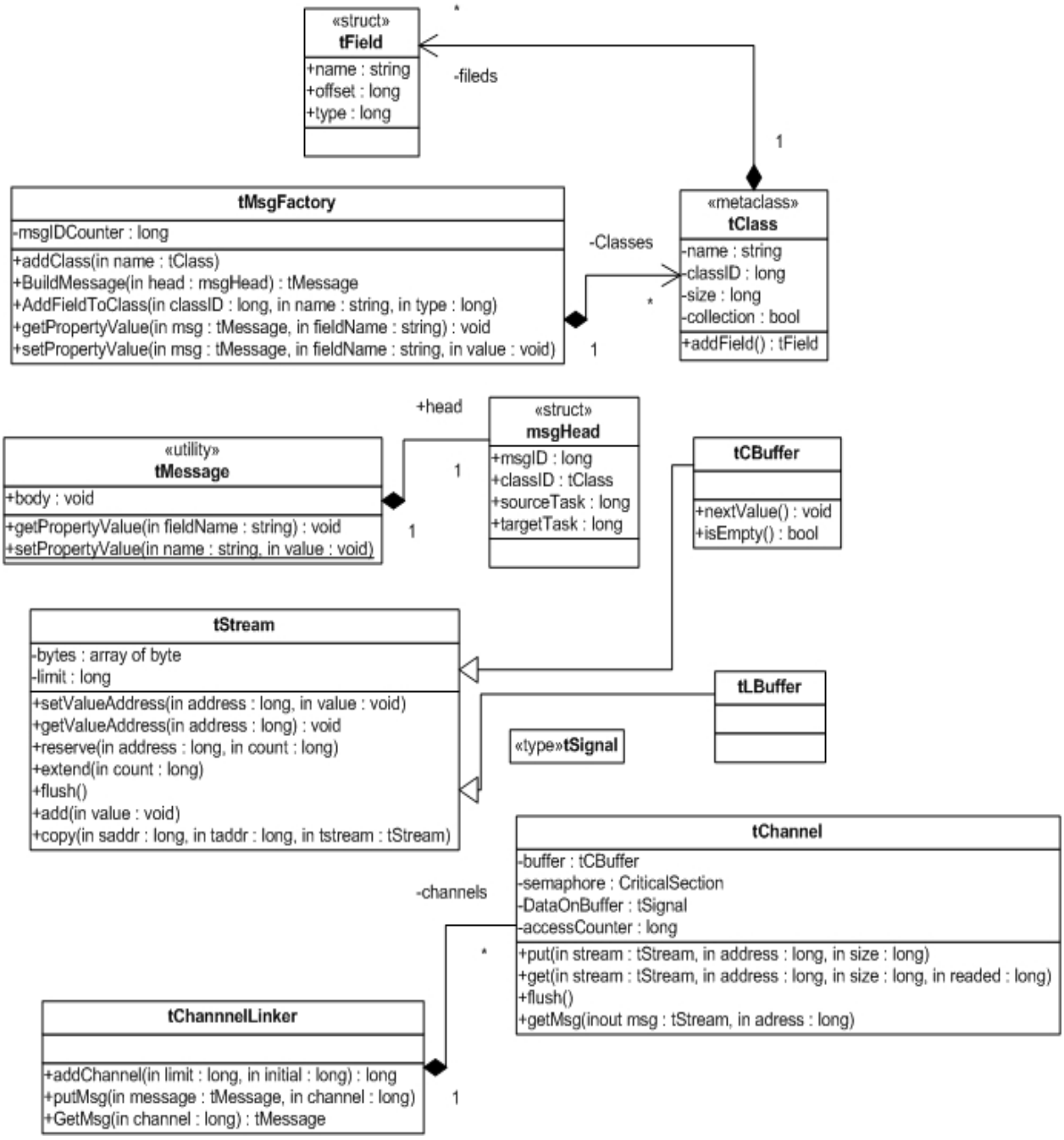
http://www.informatik.uni-osnabrueck.de/barbara/papers/pub_hammer.html

http://www.imse.cnm.es/~lumi/neuroprogr.html

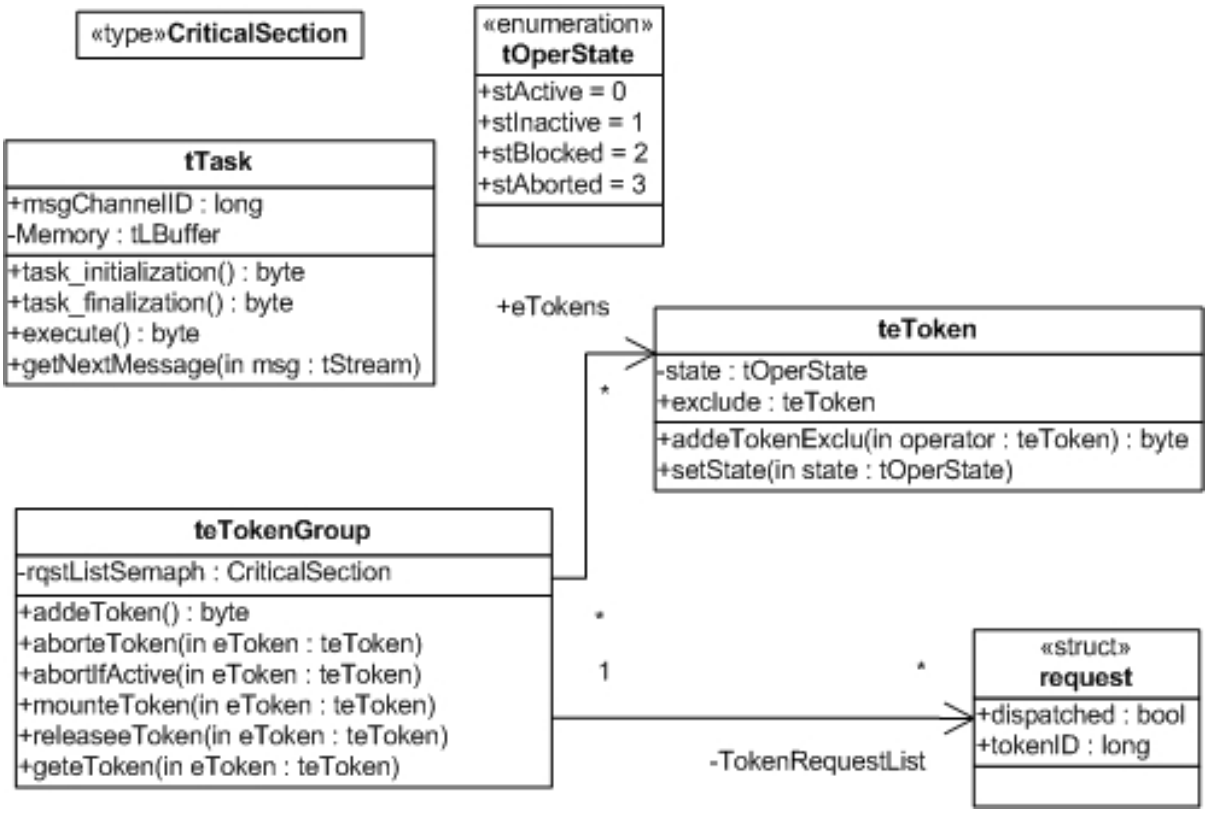
http://www.ncrg.aston.ac.uk/netlab/over.php

Anexo 4. Diagramas de la Estructura de Clases para la Gestión de Procesos Concurrentes.

4.1 Diagrama de Clases del Paquete *Communication Infrastructure*

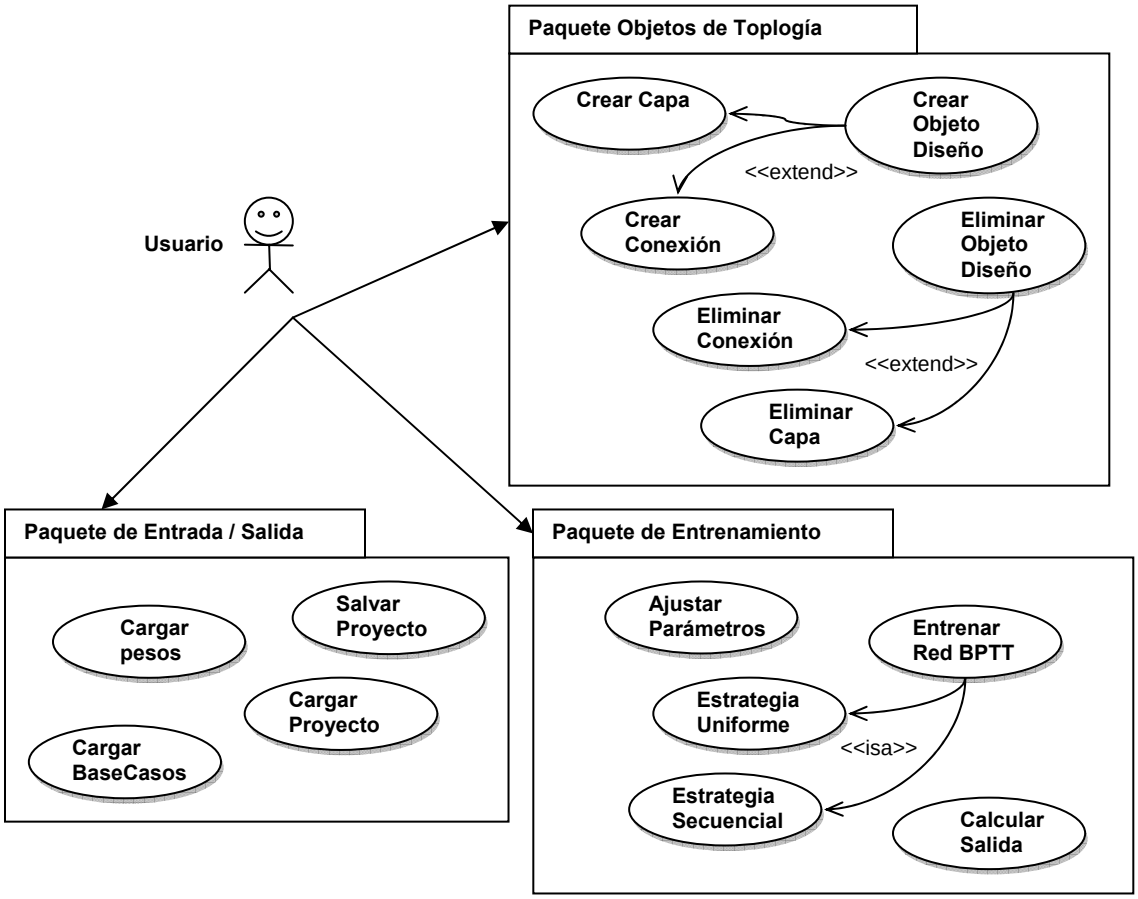


4.2 Diagrama de Clases del Paquete *Scheduling Infrastructure*



Anexo 5. Diagramas de la Herramienta NEngine

5.1 Diagrama de Casos de Uso.



5.2 Diagrama de Clases para Manipular Topologías