COLEGIO DE POSTGRADUADOS

INSTITUCIÓN DE ENSEÑANZA E INVESTIGACIÓN
EN CIENCIAS AGRÍCOLAS

CAMPUS MONTECILLO

SOCIOECONOMÍA, ESTADÍSTICA E INFORMÁTICA
CÓMPUTO APLICADO

Redes neuronales regularizadas; un enfoque Bayesiano con cómputo paralelo

Eduardo Guzmán Hernández

T E S I S

PRESENTADA COMO REQUISITO PARCIAL PARA
OBTENER EL GRADO DE:

MAESTRO EN CIENCIAS

MONTECILLO,TEXCOCO, EDO. DE MÉXICO
2016

La presente tesis titulada: **Redes neuronales regularizadas; un enfoque Bayesiano con cómputo paralelo**, realizada por el alumno: **Eduardo Guzmán Hernández**, bajo la dirección del Consejo Particular indicado ha sido aprobada por el mismo y aceptada como requisito parcial para obtener el grado de:

MAESTRO EN CIENCIAS

SOCIOECONOMÍA, ESTADÍSTICA E INFORMÁTICA
CÓMPUTO APLICADO

CONSEJO PARTICULAR

CONSEJERO

Pérez Rdz.

Dr. Paulino Pérez Rodríguez

ASESOR

Dr. Mario Alberto Vázquez-Peña

ASESOR

Dr. David Hebert del Valle Paniagua

Redes neuronales regularizadas; un enfoque Bayesiano con cómputo paralelo

Eduardo Guzmán Hernández, MC

Colegio de Postgraduados, 2016

Resumen

Una Red Neuronal Artificial (RNA) es un paradigma de aprendizaje y procesamiento automático inspirado en el comportamiento biológico de las neuronas y en la estructura del cerebro. El cerebro es un sistema altamente complejo; su unidad básica de procesamiento son las neuronas, las cuales se encuentra distribuidas de forma masiva compartiendo múltiples conexiones entre ellas. Las RNAs intentan emular ciertas características propias de los humanos, pueden ser vistas como un sistema inteligente que lleva a cabo tareas de manera distinta a como lo hacen las computadoras actuales. Las RNAs pueden emplearse para realizar actividades complejas, por ejemplo: reconocimiento y clasificación de patrones, predicción del clima, predicción de valores genéticos, etc. Los algoritmos utilizados para entrenar las redes, son en general complejos, por lo cual surge la necesidad de contar con alternativas que permitan reducir de manera significativa el tiempo necesario para entrenar una red. En este trabajo se presenta una propuesta de algoritmos basados en la estrategia "divide y conquista" que permiten entrenar las RNAs de una sola capa oculta. Parte de los sub problemas del algoritmo general de entrenamiento se resuelven utilizando técnicas de cómputo paralelo, lo que permite mejorar el desempeño de la aplicación resultante. El algoritmo propuesto fue implementado utilizando el lenguaje de programación C++, así como las librerías Open MPI y ScaLAPACK. Se presentan algunos ejemplos de aplicación y se evalúa el desempeño del programa resultante. Los resultados obtenidos muestran que es posible reducir de manera significativa los tiempos necesarios para ejecutar el programa que implementa el algoritmo para el ajuste de la RNA.

Palabras clave: Open MPI, C++, ScaLAPACK, Cómputo paralelo.

Artificial Neuronal Networks; a Bayesian approach using paralell computing

Eduardo Guzmán Hernández, MC

Colegio de Postgraduados, 2016

Abstract

An Artificial Neural Network (ANN) is a learning paradigm and automatic processing inpired in the biological behaviour of neurons and the brain structure. The brain is a complex system; its basic processing unit are the neurons, which are distributed massively in the brain sharing multiple connections between them. The ANNs try to emulate some characteristics of humans, and can be thought as intelligent systems that perform some tasks in a different way that actual computer does. The ANNs can be used to perform complex activities, for example: pattern recognition and classification, weather prediction, genetic values prediction, etc. The algorithms used to train the ANN, are in general complex, so therefore there is a need to have alternatives which lead to a significant reduction of times employed to train an ANN. In this work we present an algorithm based in the strategy "divide and conquer" which allows to train an ANN with a single hidden layer. Part of the sub problems of the general algorithm used for training are solved by using parallel computing techniques, which allows to improve the performance of the resulting application. The proposed algorithm was implemented using the programming language C++, and the libraries Open MPI and ScaLAPACK. We present some application examples and we asses the application performance. The results shown that it is possible to reduce significantly the time necessary to execute the program that implements the algorithm to train the ANN.

Key words: Open MPI, C++, ScaLAPACK, Parallel computing.

AGRADECIMIENTOS

Al Consejo Nacional de Ciencia y Tecnología (CONACyT) por el apoyo económico brindado durante la realización de mis estudios.

Al Colegio de Postgraduados, por haberme brindado la oportunidad de seguir mi formación académica en sus aulas.

A los integrantes de mi Consejo Particular:

Al Dr. Paulino Pérez Rodríguez, le agradezco sinceramente por haber confiado en mi, por su atención, por su valioso apoyo tanto profesional, como personal y a sus indicaciones que fueron de suma importancia para concluir mi trabajo de investigación.

A los doctores Mario Alberto Vázquez Peña y David Hebert del Valle Paniagua por su orientación, apoyo y colaboración desinteresada en el presente trabajo de investigación.

A mis profesores, compañeras de clase y todas aquellas personas que de alguna manera fueron coparticipes de esta tarea, a todos gracias.

DEDICATORIA

A mi mamá, por alentarme a seguir y terminar esta etapa ...

A mi dulce esposa, quien me apoyó y alentó para continuar, cuando parecía que me iba a rendir.

Dedico esta tesis a todos aquellos que no creyeron en mí, a aquellos que esperaban mi fracaso en cada paso que daba hacia la culminación de mis estudios, a aquellos que nunca esperaban que lograra terminar esta maestría, a todos aquellos que le apostaban a que me rendiría a medio camino, a todos los que supusieron que no lo lograría, a todos ellos les dedico esta tesis.

Al Dr. Paulino por haber confiado en mí, por no haberse rendido ni desesperado al enseñarme y explicarme, a pesar de que muchas cosas no las entendía a la primera, ni a la segunda, ni a la ..., y algunas aún no las entiendo. Gracias.

Contenido

Lista de tablas

Lista de figuras

Capítulo 1

Introducción

El incremento en el poder de procesamiento de las computadoras, ha mostrado resultados palpables en varias áreas, desde la representación numérica y gráfica de grandes eventos físicos, hasta la simulación numérica de sistemas complejos como: dinámica de fluidos, modelación del clima, diseño de circuitos eléctricos, reacciones químicas, modelos ambientales y procesos de manufacturación. Estas y otras áreas han impulsado el desarrollo de computadoras cada vez más potentes (Vargas-Lombardo, 2012). En un inicio el modelo computacional fue de naturaleza secuencial. En la Figura 1.1, se ilustra como una computadora con un solo procesador tiene la capacidad de ejecutar una instrucción a la vez; sin embargo, desde los primeros días del cómputo secuencial se vislumbraba la necesidad de recurrir al cómputo paralelo para resolver los nuevos problemas que demandan mucho poder de cómputo, es por ello que se busca que las aplicaciones puedan realizar los cálculos lo más rápido posible aprovechando todos los recursos del equipo con la mayor eficiencia posible (Almeida, 2008).

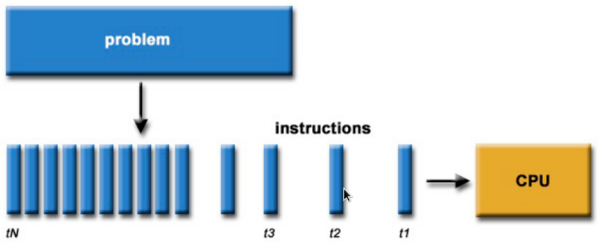

Figura 1.1: Cómputo serial, en donde un problema se divide en sub tareas las cuales se ejecutan en un solo procesador.
Fuente: Amilcar (2011).

En el pasado para cubrir con las necesidades del mercado se incrementaba la velocidad del procesador, aumentando el número de transistores al circuito integrado. En 1965, Gordon Moore afirmó que el número de transistores por centímetro cuadrado en un circuito

integrado se duplicaba cada año y que la tendencia continuaría durante las siguientes dos décadas. Más tarde, en 1975, Moore modificó su propia afirmación y predijo que el ritmo bajaría, y que la densidad de transistores se duplicaría aproximadamente cada 18 meses, como se muestra en la Figura 1.2 (Wong, 2005).

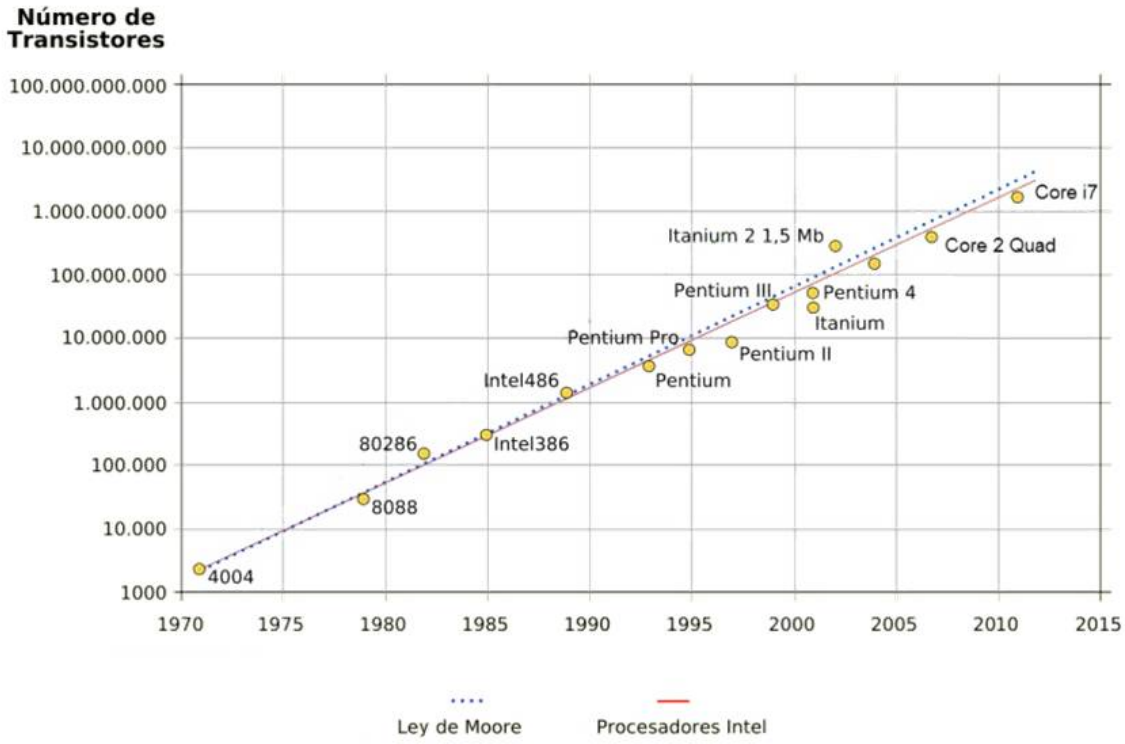

Figura 1.2: Número de transistores por procesador de 1970 al 2015. La línea punteada muestra las predicciones de la ley de Moore y la línea solida muestra las predicciones de Intel, los puntos representan el número real de transistores <https://tecnologiaraptor.wordpress.com/2014/09/10/proceso-de-manufactura-de-un-microprocesador> [Consulta: 15 de abril, 2015].

En la actualidad no es fácil disminuir el tamaño de los transistores. Para compensarlo se han construido procesadores que tienen más de un núcleo de procesamiento. Estas nuevas cualidades de las computadoras han creado la necesidad de desarrollar aplicaciones que aprovechen las nuevas características del procesamiento paralelo. La incorporación de más núcleos de procesamiento tiene la finalidad de aumentar la velocidad de ejecución de las aplicaciones.

La idea del procesamiento en paralelo es hacer que una tarea se ejecute rápidamente ya que existen varios procesadores que de forma simultánea y coordinada den solución a un problema. En la Figura 1.3 se muestra de forma teórica que una tarea se divide en subtareas, las cuales se sincronizan en los diferentes núcleos. Así con n procesadores podremos dividir la tarea en n subtareas y el problema se resolverá n veces más rápido,

sin embargo este proceso solo se lograría con tareas que se puedan ejecutar de forma independiente, la mayoría de los procesos dependen de otros, la velocidad para resolver un problema dependerá de que tanta dependencia tienen las subtareas.

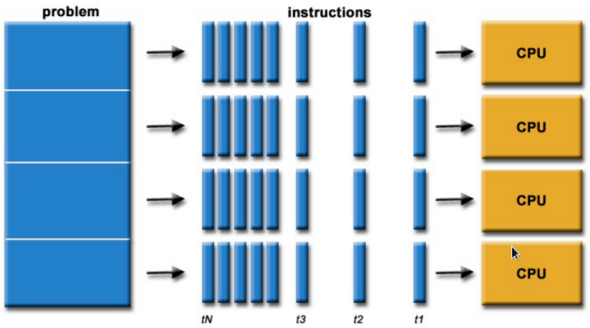

Figura 1.3: Cómputo paralelo. Un problema se divide en múltiples subtareas que son ejecutadas en varios procesadores.
Fuente: Amilcar (2011).

La meta de la división de las tareas es reducir al mínimo el tiempo total de cómputo distribuyendo la carga de trabajo entre los procesadores disponibles, esta es una situación ideal pero en la práctica, no resulta como dicta la teoría. Existe una sobrecarga extra al momento de transferir los datos a los diferentes núcleos y viceversa, por lo tanto, la velocidad de ejecución de una tarea no es lineal o proporcional al número de elementos de procesamiento.

En la Figura 1.4, se ilustra un ejemplo de multiplicación de matrices (Meyer y Tier, 2013), en el cual se puede observar que la GPU (96 cores) y CPU4 (4 cores) no presenta una gran diferencia en el tiempo de ejecución, el CPU1 (1 core) presenta un tiempo de ejecución más alto con respecto a los dos anteriores. Debido a que un solo núcleo realiza toda la tarea, sin embargo, la ejecución no depende únicamente del número de núcleos del procesador, sino que influyen otros factores como el tipo de procesador, tipo de hardware, el problema a resolver, etc.

Cabe mencionar que en la actualidad la mayoría de las aplicaciones son seriales, es decir se ejecutan solo en uno de los núcleos; estos programas no utilizan de manera eficiente los recursos con los que cuenta el equipo. Para contrarrestarlo surge la necesidad de escribir códigos que puedan ejecutarse en paralelo.

La velocidad de procesamiento no es la única razón para utilizar el paralelismo. La construcción de aplicaciones más complejas han requerido computadoras mas rápidas, y las limitaciones en el desarrollo de computadoras seriales han llegado a ser más evidentes.

En la actualidad, el costo del hardware disminuye rápidamente, por lo que el procesamiento paralelo se está empleando cada vez más en tareas de diversas áreas, debido al hardware que permiten llevar a cabo este tipo de procesamiento. Además es necesario

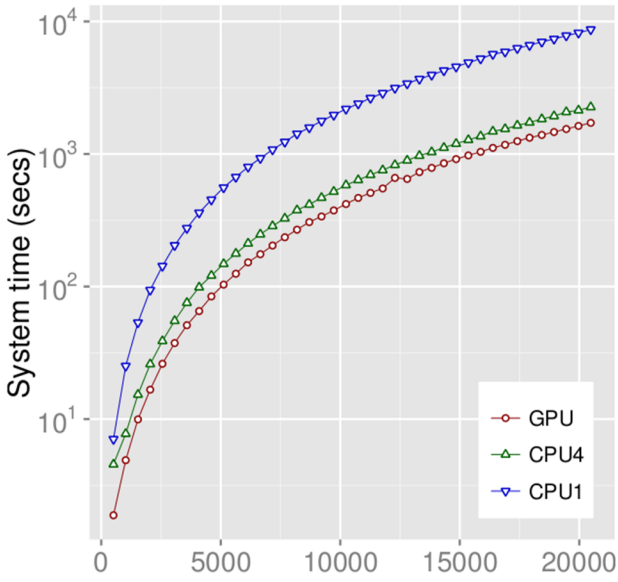

Figura 1.4: Tiempo de ejecución de rutinas que realizan el producto matricial $\boldsymbol{Z}\boldsymbol{Z}'$, $z_{ij} \in \{0, 1, 2\}$. El número de columnas de $\boldsymbol{Z}$ fue de 512,000, mientras el número de filas varió entre 512 y 20, 480.
Fuente: Meyer y Tier (2013).

contar con software que soporte la ejecución y la coordinación de procesos en forma paralela.

En el ámbito de la inteligencia artificial, podemos encontrar varios problemas en los que se involucran algoritmos como las redes neuronales artificiales como menciona Brío y Molina (2005), éstas constituyen un modelo computacional el cual esta inspirado en características generales de las redes neuronales biológicas, y que permiten resolver diversos problemas de la vida real. Por dar algunos ejemplos, podemos se puede mencionar la lectura de caracteres, el reconocimiento de patrones, la búsqueda de información en la web, la recuperación de imágenes en Internet, descubrimiento de información en grandes bases de datos, el reconocimiento musical y de voz, entre otros.

Para solucionar los problemas antes mencionados se puede recurrir a las técnicas de programación tradicional, sin embargo los resultados no serán eficientes, puesto que la ejecución de un programa podría llevar mucho tiempo y requerir demasiado poder de cómputo, lo cual puede implicar un problema si no se cuenta con el recurso computacional adecuado.

Para mitigar estos posibles problemas se puede recurrir al cómputo paralelo el cual proporcionara la arquitectura e infraestructura necesaria para resolver los problemas antes mencionados. Sin embargo, que un ordenador tenga una mayor capacidad de procesamiento no se consigue simplemente colocando un número mayor de procesadores, sino que supone una gran complejidad, tanto en el hardware como en el software.

Los objetivos del presente trabajo son los siguientes:

- Diseñar e implementar un algoritmo para ajustar una red neuronal regularizada Bayesiana usando cómputo paralelo.
- Implementar el algoritimo de estimación de parámetros en una red neuronal regularizada Bayesiana, usando Open MPI.
- Evaluar el desempeño del programa que implementa el algoritmo propuesto.
- Comparar la eficiencia del algoritmo propuesto usando varios arreglos de procesamiento en ScaLAPACK.

A continuación se describe la organización del trabajo. En el capítulo 2 se presenta el marco teórico. En el capítulo 3 se desarrollan los algoritmos para el ajuste de la red usando cómputo paralelo. En el capítulo 4 se presentan ejemplos de aplicación, así como los resultados de la evaluación del desempeño de un programa escrito en C++ que implementa el algoritmo descrito en el capítulo 3. En el capítulo 5 se presentan las conclusiones. Los anexos del trabajo presentan el código fuente de la aplicación en C++, así como algunas rutinas auxiliares en el paquete estadístico R (R Core Team, 2015).

Capítulo 2

Marco Teórico

2.1. Redes Neuronales Artificiales (RNAs)

2.1.1. Modelo Biológico

El sistema de comunicación neuronal se compone de tres partes fundamentales: 1) Receptores, 2) Sistema nervioso y 3) Organos afectores. El elemento estructural y funcional más esencial en el sistema de comunicación neuronal, es la célula nerviosa o neurona. La mayoría de las neuronas utilizan señales químicas (transmisores) para el envío de la información. Dicha información se envía, entre las distintas neuronas, a través de conexiones formando redes, en las cuales se elabora y almacena la información (Viñuela y León, 2004).

La mayoría de las neuronas codifican sus salidas como una serie de impulsos de tensión breves. Estos pulsos, comúnmente conocidos como potenciales de acción o picos, se originan en o cerca del cuerpo celular de las neuronas y luego se propagan a través de las neuronas individuales en velocidad y amplitud constante (Haykin, 1999). La misión de las neuronas comprende generalmente cinco funciones parciales:

1. Recoger la información (mensaje) que llega a ellas en forma de impulsos procedentes de otras neuronas.
2. Integración de un código de activación.
3. Transmisión del mensaje codificado en forma de impulsos a través de su axón.
4. Distribución de los mensajes a través del axón.
5. Transmisión de impulsos a las neuronas subsiguientes.

En la Figura 2.1 se observa que la neurona consta de un cuerpo celular y un núcleo, como el resto de las células del organismo, pero cuenta con algunos elementos específicos, el axón, que es una ramificación de salida de la neurona, a través de él se propaga una serie de impulsos electro-químicos. A demás, la neurona cuenta con un gran número de ramificaciones de entrada que se conocen como dendritas; cuya tarea es propagar la señal al interior de la neurona.

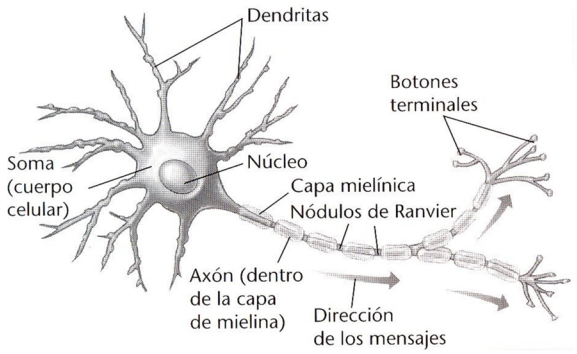

Figura 2.1: Red neuronal biológica. Una neurona es una unidad básica de procesamiento de información, el canal de entrada está compuesto por las dendritas, el procesador por el soma y la salida por el axón <https://filosert.wordpress.com/temas/4-bases-biologicas-del-comportamiento/ > [Consulta: 15 octubre, 2015].

Las sinapsis recogen información electro-químicas procedentes de las células vecinas a las que se encuentran conectadas; esta información llega al núcleo donde es procesada hasta generar una respuesta que es propagada por el axón. Más tarde, la señal única propagada por el axón se ramifica y llega a las dendritas de otras células a través de lo que se denomina sinapsis, las cuales son los elementos de unión entre el axón y las dendritas. El funcionamiento en general será el de una enorme malla que propaga señales electro-químicas de una célula a otra y que va modificando la información mediante la sinapsis, en este caso la sinapsis es una concentración iónica que cambia cuando pasa de una neurona a otra. La conectividad entre las células del cerebro es muy elevada; se calcula que en cada cerebro existen alrededor de 100, 000 millones de neuronas, conectadas cada una de ellas con alrededor de 10, 000, es decir, la salida de cada neurona afecta a otras 10, 000 neuronas (Viñuela y León, 2004).

2.1.2. Modelo Artificial

Una red neuronal, es un sistema de procesadores paralelos conectados entre sí en forma de grafo dirigido. Esquemáticamente cada elemento de procesamiento (neuronas) de la red se representa como un nodo. Estas conexiones establecen una estructura jerárquica que tratando de emular la fisiología del cerebro, busca nuevos modelos de procesamiento

para solucionar problemas concretos del mundo real. Un aspecto esencial de las RNA es su capacidad de aprender, reconocer y aplicar relaciones entre objetos y grupos de objetos propios del mundo real. En este sentido, las RNAs se consideran como herramientas que podrán resolver problemas difíciles.

McCulloch y Pitts (1943) desarrollaron la primera neurona artificial que imita las características de la neurona biológica. En esencia, es un conjunto de datos de entrada los cuales representa la salida de otra neurona. Cada entrada se multiplica por un peso correspondiente, análogo a una fuerza sináptica, y todas las entradas ponderadas se suman para determinar el nivel de activación de la neurona (Freeman y Skapura, 1993).

Una comparación clásica de las capacidades de procesamiento de información entre el ser humano y la computadora se destaca por un intento de automatizar el procesamiento de información; un ordenador convencional utiliza un algoritmo que opera en serie y es controlada por la unidad central de procesamiento y almacena la información en árboles para tener la ubicación de la memoria, en donde se encuentra almacenada la información. Por otro lado, el cerebro funciona con transformaciones altamente distribuidas que operan en paralelo, su control se distribuye a través de miles de neuronas interconectadas que almacenan la información de forma distribuida entre las neuronas.

Las RNAs constan de un bloque básico denominado neurón también se les conoce como neuronas artificiales. Estas neuronas se conectan entre sí en una o varias capas, las cuales están compuestas de elementos que actúan de forma similar a los elementos biológicos de las neuronas cerebrales (Khare y Nagendra, 2006). Las neuronas artificiales están inspiradas en el funcionamiento del cerebro humano, el cual realiza sus cálculos completamente diferentes a la de la computadora digital convencional (Simon, 1999).

2.1.3. La Neurona Artificial

Según Viñuela y León (2004) la neurona artificial, célula o autómata, es un elemento que posee un estado interno, llamado nivel de activación y recibe señales que le permiten, en su caso, cambiar de estado. El nivel de activación de una neurona depende de las entradas recibidas y de los valores sinápticos, pero no de los valores anteriores del estado de activación, se calcula primero la entrada total de la neuronal, este valor es la suma de todas las entradas ponderadas por ciertos valores.

La Figura 2.2 muestra un modelo que representa la idea de una neurona artificial. Aquí un grupo de entradas $x_1, x_2, \ldots, x_m$ son introducidas en una neurona artificial. Éstas entradas, definidas por un vector $\boldsymbol{x}$, corresponden a las señales de la sinapsis de una neurona biológica. Cada señal se multiplica por un peso asociado $w_1, w_2, \ldots, w_m$ antes de que se sumen las salidas ($\sum$). Cada peso corresponde a la fuerza de una conexión sináptica, es decir el nivel de concentración iónica de la sinapsis, y se representa por un vector $\boldsymbol{w}$.

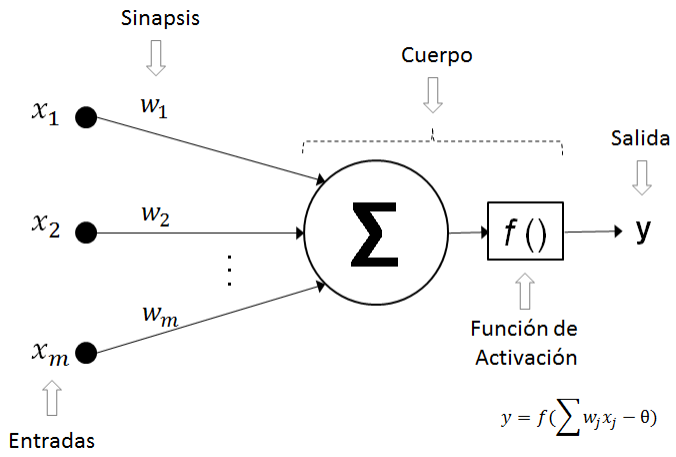

Figura 2.2: Modelo de una neurona artificial, la cual consta de un vector de entradas $\boldsymbol{x} = (x_1, x_2, \ldots, x_m)'$, un conjunto de pesos sinópticos w_j, con $j = 1, \ldots, m$, una función de activación, la cual proporciona el estado de activación de la neurona, una salida y en función del estado de activación (Haykin, 1999).

La sumatoria corresponde al cuerpo de la neurona, suma todas las entradas ponderadas algebraicamente, produciendo una salida que se denomina E, así:

$$E = x_1 w_1 + x_2 w_2 + \cdots + x_m w_m.$$

Esto puede ser definido en forma vectorial como:

$$E = \boldsymbol{x}'\boldsymbol{w}.$$

La señal E es procesada además por una función llamada función de activación o de salida F, que produce la señal de salida de la neurona s.

2.2. Regresión Lineal Múltiple

En la mayoría de los problemas de investigación donde se aplica el análisis de regresión es necesario incluir más de una variable independiente en el modelo de regresión. La complejidad de la mayor parte de los mecanismo científicos es tal, que para ser capaces de predecir una respuesta importante se necesita un modelo de **regresión múltiple**. Cuando este modelo es lineal en los coeficientes se denomina **modelo de regresión lineal múltiple** (Walpole *et al.*, 1999).

2.2. Regresión Lineal Múltiple

La definición dada por Triola (2004) menciona que una ecuación de regresión múltiple expresa una relación lineal entre una variable dependiente y y dos o más variables independientes $(x_1, x_2, ..., x_p)$, la forma general de una ecuación de regresión múltiple es:

$$y_i = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \cdots + \beta_p x_{ip} + \varepsilon_i, \tag{2.1}$$

donde $\boldsymbol{\beta} = (\beta_0, ..., \beta_p)'$ son denomidados **coeficientes de regresión**. La predicción de la variable respuesta se obtiene sustituyendo $\boldsymbol{\beta}$ por estimaciones de estos coeficientes, es decir, $\widehat{\boldsymbol{\beta}}$ en (2.1)

$$\widehat{y} = \widehat{\beta}_0 + \widehat{\beta}_1 x_1 + \widehat{\beta}_2 x_2 + \cdots + \widehat{\beta}_p x_p,$$

donde cada coeficiente de regresión $\beta_j, j = 1, ..., p$ se estima por $\widehat{\beta}_j$, generalmente $\widehat{\boldsymbol{\beta}}$ se obtiene por el método de mínimos cuadrados ordinarios. El ajuste de un modelo de regresión lineal múltiple se facilita enormemente usando vectores y matrices (Walpole *et al.*, 2012).

Note que el modelo (2.1) representa en esencia a n ecuaciones que describen cómo se generan los datos. Si usamos la notación de matrices el modelo (2.1) puede describirse como:

$$\boldsymbol{y} = \boldsymbol{X}\boldsymbol{\beta} + \boldsymbol{\varepsilon},$$

donde:

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \cdot \\ \cdot \\ \cdot \\ y_n \end{bmatrix}, \mathbf{X} = \begin{bmatrix} 1 & x_{11} & x_{12} & \cdot & \cdot & \cdot & x_{1p} \\ 1 & x_{21} & x_{22} & \cdot & \cdot & \cdot & x_{2p} \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ 1 & x_{n1} & x_{n2} & \cdot & \cdot & \cdot & x_{np} \end{bmatrix}, \boldsymbol{\beta} = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \cdot \\ \cdot \\ \cdot \\ \beta_p \end{bmatrix}, \boldsymbol{\varepsilon} = \begin{bmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \cdot \\ \cdot \\ \cdot \\ \varepsilon_n \end{bmatrix}$$

$\boldsymbol{\varepsilon}$ representa los errores aleatorios, es decir, $\boldsymbol{\varepsilon} \sim N(\mathbf{0}, \sigma_e^2 \boldsymbol{I})$.

Hay algunos casos especiales del modelo de regresión múltiple (2.1) que se utilizan frecuentemente. Uno es el **modelo de regresión polinomial**, en el cual las variables independientes son potencias de una sola variable, el cual está dado por:

$$y_i = \beta_0 + \beta_1 x_i + \beta_2 x_i^2 + \cdots + \beta_p x_i^p + \varepsilon_i. \tag{2.2}$$

Los modelos de regresión múltiple también se pueden hacer con potencias de diversas variables. Por ejemplo, un modelo de regresión polinomial de grado 2, también llamado **modelo cuadrático**, en dos variables x_1 y x_2 está dado por:

$$y_i = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \beta_3 x_{i1} x_{i2} + \beta_4 x_{i1}^2 + \beta_5 x_{i2}^2 + \varepsilon_i. \tag{2.3}$$

Una variable producto de las otras dos variables es llamada **interacción**. En la ecuación (2.3), la variable $x_{i1}x_{i2}$ es la **interacción** entre x_1 y x_2.

Los modelos (2.2) y (2.3) se consideran lineales, aunque contengan términos no lineales de las variables independientes. La razón de que continúen siendo modelos lineales es que son *lineales en los coeficientes* β_j, $j = 0, ..., p$ (Navidi, 2006).

Los cálculos que se requieren para ajustar un modelo de regresión múltiple son tan complicados que los usuarios depende de las computadoras y de software estadístico para resolver el problema.

2.2.1. Estimación de los coeficientes

En cualquier modelo de regresión múltiple, los estimadores $\widehat{\beta}_0, \widehat{\beta}_1, ..., \widehat{\beta}_p$, se calculan por medio de mínimos cuadrados. Defina $\widehat{y}_i$ como la coordenada y_i de la ecuación de mínimos cuadrados correspondientes a los valores $x_{i1}, ..., x_{ip}$. Los residuos representan las cantidades $e_i = y_i - \widehat{y}_i$, que constituyen las diferencias entre los valores observados y y los valores predichos $\widehat{y}$ que proporciona la ecuación. Para obtener $\widehat{\beta}_0, \widehat{\beta}_1, ..., \widehat{\beta}_p$, es necesario minimizar la suma de los cuadrados residuales $\sum_{i=1}^{n} e_i^2$. Con ese propósito se expresa e_i como función de $\beta_0, \beta_1, \ldots, \beta_p$:

$$e_i = y_i - \beta_0 - \beta_1 x_{i1} - \cdots - \beta_p x_{ip}.$$

Por lo tanto, se desea minimizar la suma

$$Q(\beta_0, \ldots, \beta_p) = \sum_{i=1}^{n} \left(y_i - \beta_0 - \beta_1 x_{i1} - \cdots - \beta_p x_{ip}\right)^2. \tag{2.4}$$

Para minimizar $Q(\beta_0, \cdots, \beta_p)$ es necesario obtener las derivadas parciales de la ecuacion (2.4) con respecto a $\beta_0, \beta_1, \ldots, \beta_p$, e igualarlas a 0 y resolver las $p + 1$ ecuaciones resultantes con $p + 1$ incógnitas, obteniendo $\widehat{\beta}_0, \ldots, \widehat{\beta}_p$. Las expresiones que se obtienen para $\widehat{\beta}_0, \widehat{\beta}_1, ..., \widehat{\beta}_p$, son complicadas. Para cada coeficiente estimado $\widehat{\beta}_j$, hay una desvia-

ción estándar estimada $s_{\widehat{\beta}_j}$. El vector $\widehat{\boldsymbol{\beta}}$ se puede obtener usando álgrebra de matrices y cálculo vectorial y está dado por $\widehat{\boldsymbol{\beta}} = \left(\boldsymbol{X}'\boldsymbol{X}\right)^{-1}\boldsymbol{X}'\boldsymbol{y}$.

2.3. Regresión No Lineal

Los modelos de regresión lineal simple y múltiple son ampliamente utilizados, debido en gran parte a sus aplicaciones y a la fácil interpretación de los resultados. Sin embargo, hay fenómenos observables que no pueden ser explicados por modelos lineales, en tales situaciones un modelo no lineal en los parámetros se puede ajustar mejor. En un modelo de regresión no lineal la función $f(\cdot)$ que relaciona la respuesta con los predictores no es necesariamente lineal y está dada por:

$$y_i = \beta_0 + f(\boldsymbol{\beta}^*, \boldsymbol{x}_i') + \varepsilon_i, \tag{2.5}$$

donde y_i es la variable respuesta, β_0 es el intercepto, $\boldsymbol{\beta}^* = (\beta_1, ..., \beta_p)'$, $\boldsymbol{x}_i' = (x_{i1}, ..., x_{ip})$ $f(\cdot)$ es una función que mapea el espacio de entrada (p-dimensional) a la recta real, ε_i es el error aleatorio asociado.

El teorema de Kolmogorov establece que cualquier función no lineal en p variables, $f(x_1, \ldots, x_p)$ puede ser representada de la forma siguiente:

$$f(\boldsymbol{x_i}) = f(x_1, ..., x_p) = \sum_{q=1}^{2p+1} g_q \left(\sum_{r=1}^{p} h_{qr}(x_r) \right), \tag{2.6}$$

donde $h_{qr}(x)$ son funciones continuas crecientes en $I = [0, 1]$, y $g_q(\cdot)$ son funciones continuas de una sola variable, para mayores detalles ver Girosi y Poggio (1989).

2.4. Redes Neuronales Artificiales (RNA)

El teorema de Kolmogorov establece las bases matemáticas que permiten interpretar una red neuronal de una sola capa oculta como un modelo de regresión no lineal. Considere el modelo de regresión no lineal dado en (2.5), si se utiliza (2.6), entonces (2.5) puede expresarse como:

$$y_i = \beta_0 + \sum_{k=1}^{s} w_k g_k \left(b_k + \sum_{j=1}^{p} x_{ij} \beta_j^{[k]} \right) + \varepsilon_i, \tag{2.7}$$

es decir, las predicciones se pueden obtener en dos pasos: 1) Las entradas se transforman en una forma no lineal en la capa oculta y 2) Las salida de la capa oculta se combinan linealmente para obtener la predicción.

El modelo dado en (2.7) corresponde a una **Red Neuronal Artificial de una sola capa oculta**, $(w_1, \ldots, w_s)'$ son pesos de las neuronas; $(b_1, \ldots, b_s)'$ son llamados interceptos (sesgos) en la máquina de aprendizaje; $(\beta_1^{[1]}, \ldots, \beta_p^{[1]}, \ldots, \beta_1^{[s]}, \ldots, \beta_p^{[s]}, \beta_0)'$ son coeficientes de regresión, donde $\beta_j^{[k]}$ es un coeficiente de regresión asociado a la variable $j = 1, \ldots, p$ y la neurona $k = 1, \ldots, s$, y $g_k(\cdot)$ es la función de activación, un ejemplo de esta función es (ver Pérez-Rodríguez *et al.* (2013)):

$$g_k(x) = \frac{\exp(2x) + 1}{\exp(2x) - 1},$$

donde $\exp(\cdot)$ es una función exponencial. La Figura 2.3 presenta gráficamente el modelo dado en la ecuación (2.7).

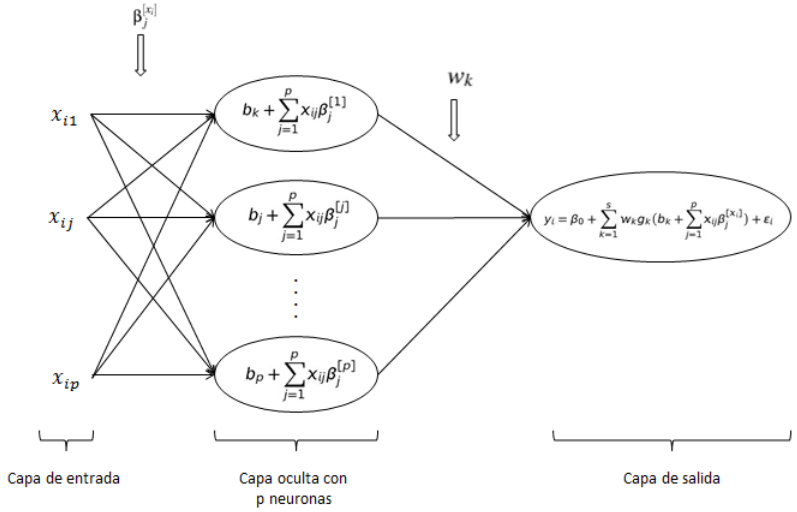

Figura 2.3: Ejemplo de la red neuronal de la ecuación (2.7), en donde se ilustra como la función de activación se aplica en cada neurona, como salida se tiene la suma de las salidas de las neuronas multiplicada por su peso w_k correspondiente.

2.4.1. Normalización de datos en las RNAs

Con el objetivo de mejorar la estabilidad numérica de los algoritmos utilizados para ajustar las RNAs se recomienda transformar todas las entradas y variables de salida de tal manera que se encuentren en el intervalo $[-1, 1]$. Una vez que el modelo es ajustado se tiene la posibilidad de obtener las predicciones en su escala original (Gianola *et al.*, 2011). Considere por ejemplo el vector de observaciones para alguna covariable, $(x_1, \ldots, x_n)' = \boldsymbol{x}$. Las entradas normalizadas de $\boldsymbol{x}$ se obtienen aplicando la expresión siguiente:

$$z_i = \frac{2(x_i - b)}{s} - 1, i = 1, \ldots, n, \tag{2.8}$$

donde $b = \text{mín}\{x_1, \ldots, x_n\}$, $s = b - \text{máx}\{x_1, \ldots, x_n\}$.

La función `normalize` de la librería de funciones `brnn` (Pérez-Rodríguez y Gianola, 2015) del paquete estadístico R implementa el proceso de normalización descrito anteriormente. Una vez que se han normalizado los datos es posible también regresar a la escala original, esto se logra despejando x_i, $i = 1, \ldots, n$ en la ecuación (2.8), es decir:

$$x_i = b + 0.5b(z_i + 1), i = 1, \ldots, n.$$

La función `un_normalize` del paquete `brnn` puede utilizarse para obtener los datos en la escala original.

2.5. Open MPI/C++

2.5.1. Conceptos de Paralelismo

Un proceso o tarea es un bloque de programa secuencial, con flujo de control propio. Es un concepto fundamental de la programación concurrente. Un procesador es el dispositivo físico sobre el cual se ejecuta el proceso. La concurrencia define la ejecución simultánea de dos o más procesos en uno o más procesadores.

Los objetivos principales del paralelismo están relacionados con la posibilidad de ajustar el modelo de arquitectura y software al mundo real y con incrementar la velocidad en la solución de problemas mediante la utilización de varios procesadores, además de mejorar la eficiencia de los mismos.

Dado que la concurrencia y el paralelismo implican la existencia de varios procesos que

interactúen en la solución de un problema, aparecen conceptos fundamentales que no existen en el mundo secuencial, como la comunicación (para intercambiar datos entre procesos) y la sincronización (para evitar colisiones entre procesos). El objetivo de la sincronización es restringir los accesos de un proceso o programa a un recurso compartido, esto se logra mediante la posesión de información acerca de otro proceso, con la finalidad de coordinar las actividades de cada proceso.

El modelo de comunicación se refiere a cómo se organiza y transmite la información compartida por los procesos concurrentes. Esto es, la manera en la cual un proceso provee datos u obtiene resultados de otro, los procesos pueden comunicarse principalmente por dos mecanismos: 1) Memoria compartida y 2) Pase de mensajes. Esto se relaciona dependiendo de la arquitectura de procesamiento utilizada (Naiouf, 2004).

2.5.2. MPI

El pase de mensajes es un modelo de comunicación ampliamente usado en computación paralela. En años recientes se han logrado desarrollar aplicaciones importantes basadas en este paradigma. Dichas aplicaciones han demostrado que es posible implementar sistemas basados en el pase de mensajes de una manera eficiente y portable. El crecimiento en el volumen y diversidad de tales aplicaciones originaron la necesidad de crear un estándar, es así como surge MPI (Message Passing Interface, ver Barney, 2014). MPI es un estándar para la implementación de sistemas de pase de mensajes, para funcionar en una amplia variedad de computadores paralelos y de forma tal que los códigos sean portables. Su diseño está inspirado en máquinas con una arquitectura de memoria distribuida, tal como se puede observar en la Figura 2.4, donde cada procesador es propietario de cierta memoria y la única forma de intercambiar información es a través de mensajes. Originalmente, MPI fue diseñado para las arquitecturas de memoria distribuida.

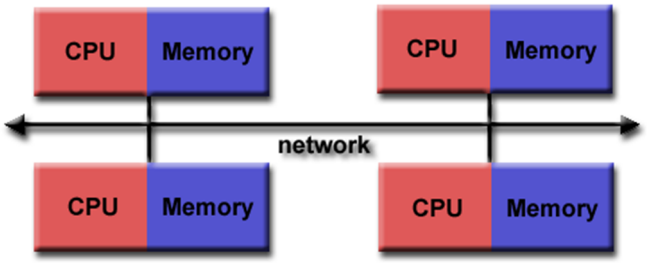

Figura 2.4: Arquitecturas de memoria distribuida, en el cual no existe el concepto de memoria global, los mecanismos de pase de mensajes se realizan sobre una red, son escalables y cuentan con alta disponibilidad es decir el fallo de un procesador no afecta el funcionamiento del sistema.
Fuente: Barney (2014).

En la actualidad MPI se ejecuta en prácticamente cualquier plataforma de hardware,

con memoria distribuida, compartida o bien híbrida. Los diseñadores de MPI tomaron las características más atractivas de varios sistemas de pase de mensajes existentes, en vez de adoptar a uno de ellos como el estándar. Así, MPI ha sido fuertemente influenciado por trabajos en T. J. Watson Research Center (IBM), NX/2 (Intel), Express (Parasoft), Vertex (nCUBE), P4 (ANL), y PARMACS (ANL); además de otras contribuciones de Zipcode (MSU), Chimp (Edinburg University), PVM (ORNL, UTK, Emory U.), Chameleon (ANL) y PICL (ANL) (Gropp *et al.*, 2014).

El objetivo principal de MPI es lograr la portabilidad a través de diferentes máquinas, tratando de obtener un grado de portabilidad comparable al de un lenguaje de programación que permita ejecutar, de manera transparente, aplicaciones sobre sistemas heterogéneos; objetivo que no debe ser logrado a expensas del rendimiento (Otto *et al.*, 1996).

MPI permita escribir programas usando pase de mensajes. Por lo tanto, la interface debe establecer un estándar práctico, portable, eficiente y flexible, MPI es una API (Application Interface) para programación multiproceso de memoria compartida, el cual permite agregar concurrencia a los programas escritos en C, C++ y Fortran. MPI se compone de un conjunto de directivas de compilador, biblioteca de rutinas, y variables de entorno que influyen el comportamiento en tiempo de ejecución (Pacheco, 1997).

Arquitectura de capas de abstracción de MPI

Open MPI cuenta con tres principales capas de abstracción (Squyres, 2014) que se muestran en la Figura 2.5 y se describen brevemente a continuación.

1. Open MPI (OMPI): es la capa de abstracción más alta, y es la única que interactúa con las aplicaciones. El API de MPI se implementa en esta capa, la portabilidad es una de sus más grandes exigencias, para lo cual esta capa soporta una amplia variedad de tipo protocolos y redes.

2. Open MPI Run-Time Environment (ORTE): es la capa de abstracción media, la cual implementa el proceso de pase de mensajes, también debe proporcionar un sistema de tiempos de ejecución, que se encargue de iniciar, supervisar y eliminar trabajos paralelos.

3. Open MPI Portable Access Layer (OPAL): es la capa de abstracción más baja. Esta capa proporciona lo siguiente:

    a) Servicios públicos para la comunicación.

    b) Mecanismos de portabilidad del núcleo de Open MPI para diferentes sistemas operativos.

    c) Manejo de memoria.

    d) Manejo del procesador y la memoria de afinidad de alta precisión.

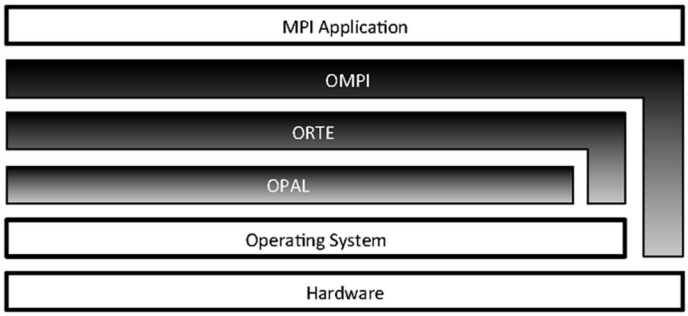

Figura 2.5: Vista de las capas de abstracción de Open MPI, mostrando sus tres capas principales: OPAL, ORTE y OMPI.
Fuente: Squyres (2014).

Elementos Open MPI

En este apartado se cuenta con un resumen de los elementos de Open MPI, publicado por la (University of Rochester, 2015).

El estándar de MPI incluye: 1) Comunicación punto-a-punto, 2) Operaciones colectivas, 3) Grupos de procesos, 4) Dominios de la comunicación, 5) Topologías de proceso, 6)Distribución de procesos de forma dinámica, 7) Tolerancia a fallos, 8) Soporte de redes heterogéneas. A continuación se describe brevemente algunos de las operaciones colectivas, paso de mensajes y comunicaciones punto a punto.

Operaciones de Difusión (Broadcast)

El tipo más sencillo de operación colectiva es la difusión. Esta operación se ilustra gráficamente en la Figura 2.6. Cada una de las fila representa un proceso diferente. Cada bloque de colores en una columna representa la ubicación de un fragmento de los datos, los bloques con el mismo color contienen copias de los mismos datos.

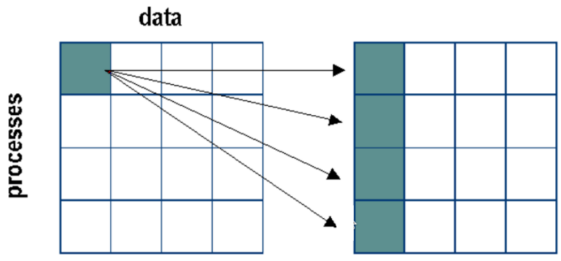

Figura 2.6: Operación de difusión (Broadcast operations). Un proceso envía una copia de algunos datos a los demás procesos de un grupo.

Operaciones Reunir y Dispersar (Scatter & Gather)

Es el mecanismo mediante el cual un nodo maestro establece una comunicación colectiva

entre los diferentes procesadores o grupos de procesadores para distribuir los datos y las tareas que deberán realizar para que al final se realice la recolección de los datos. El proceso se ilustra en la Figura 2.7.

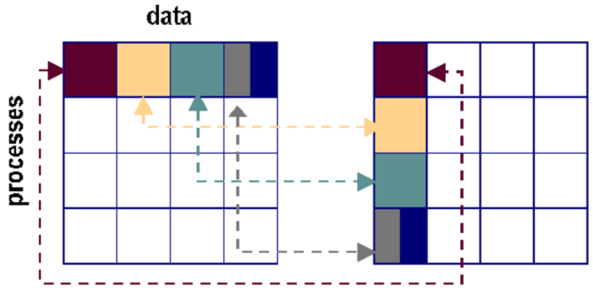

Figura 2.7: Operaciones reunir y Dispersar (Scatter & Gather operations). Todos los datos son accesibles solamente por un solo proceso (el lado izquierdo de la Figura). Después de la operación de dispersión, los fragmentos de datos se distribuyen en diferentes procesos (el lado derecho de la Figura).

Operaciones de Reducción (Reduction)

Una reducción es una operación colectiva en la que un único proceso (el proceso raíz) colecta los datos utilizados por los demás procesos. Después de está operación, los datos se encuentran situados en el proceso raíz (ver Figura 2.8).

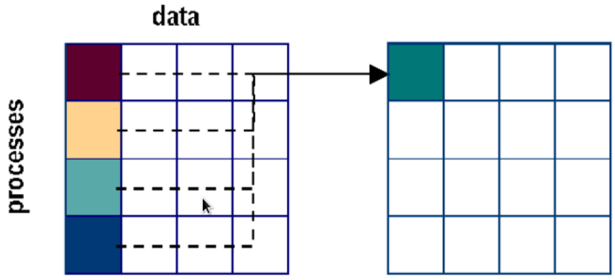

Figura 2.8: Operaciones de Reducción (Reduction operations). Inicialmente los datos se encuentran distribuidos entre varios procesadores y después de la operación se colocan en el proceso raíz.

Comunicación punto a punto (Point-to-Point Communications)

Los procesos de comunicación punto a punto proporcionados por la librería MPI operan de forma asíncrona, es decir los procesos tienen a menudo uno o más mensajes que se han enviado pero aún no recibidos. En MPI los mensajes pendientes no se mantienen en una cola FIFO (First Input-First Ouput) simple, si no que cada mensaje en espera tiene varios atributos, donde el proceso de destino (proceso de recepción) puede utilizar los atributos para determinar qué mensaje ha recibido.

Mensaje

Los mensajes constan de dos partes: la etiqueta y el cuerpo del mensaje. La envoltura de un mensaje de MPI es análoga a la envoltura de papel alrededor de una carta enviado por correo. El sobre de una carta típicamente tiene la dirección del destinatario, la dirección del remitente, y cualquier otra información necesaria para transmitir y entregar la carta, como la clase de servicio (correo aéreo, por ejemplo).

El sobre de un mensaje MPI tiene 4 partes:

1. Fuente.
2. Destino.
3. Comunicador. Especifica un grupo de procesos a los que la fuente y el destino pertenecen.
4. Etiqueta (tag). Es utilizada para clasificar los mensajes.

Las etiquetas de los mensaje son necesarias, pero su uso se deja en manos del programa. Un par de procesos que se comunican pueden utilizar valores de variables para distinguir clases de mensajes. Por ejemplo, un valor de la variable puede ser utilizado para los mensajes que contienen datos y otro valor para los mensajes que contienen información del estado.

Open MPI se puede descargar desde el sitios oficial, http://www.open-mpi.org/. El sitio tiene información detallada para realizar el proceso de compilación e instalación, se cuentan con librerías pre-compiladas para diferentes distribuciones Linux y para Cygwin, un emulador de Linux para el sistema operativo Windows, la ultima versión estable es la 1.10.0 (25/Ago/2015).

2.5.3. C++

El lenguaje C++ se comenzó a desarrollar en 1980. Su autor fue B. Stroustrup. Al comienzo era una extensión del lenguaje C que fue denominada *"C con clases"*. Este nuevo lenguaje comenzó a ser utilizado en 1983. El nombre C++ es también de ese año, y hace referencia al carácter del operador incremento de C (++).

El C++ mantiene las ventajas del C en cuanto a riqueza de operadores y expresiones, flexibilidad y eficiencia. Además, ha eliminado algunas de las dificultades y limitaciones del C original. La evolución de C++ ha continuado con la aparición de Java, un lenguaje creado simplificando algunas cosas de C++ y añadiendo otras, que se utiliza para realizar aplicaciones en Internet. El C++ es a la vez un lenguaje procedural (orientado a

algoritmos) y orientado a objetos. Como lenguaje procedural se asemeja al C (Deitel y Deitel, 2008).

2.5.4. Compiladores de C/C++

Existen multitud de compiladores, tanto libres como comerciales para las plataformas de cómputo más comunes. Para los sistemas Linux los compiladores estándar son proporcionados por GNU (http://gnu.org). Para instalar C/C++ en alguna distribución del sistema operativo Linux, se puede utilizar el gestor de repositorios de dicha distribución. Es necesario instalar las librerías de gcc, gcc-c++, make y el depurador de código (gdb).

Para instalar los compiladores de C/C++ en Windows, se pueden utilizar adaptadores de software Linux como MinGW (http://www.mingw.org/) y Cygwin (https://www.cygwin.com/).

2.6. ScaLAPACK

ScaLAPACK, es un acrónimo de librería de funciones de Álgebra lineal escalable (Scalable Linear Algebra PACKage), en este apartado se describirán los elementos principales del software, para lo cual se utilizó como referencia el sitio oficial de ScaLAPACK (Blackford *et al.*, 2012).

LAPACK (Linear Algebra PACKage)

LAPACK o paquete de álgebra lineal, es una colección de rutinas para la resolución de sistemas lineales, problemas de mínimos cuadrados, cálculo de eigen-vectores y de factorización de matrices. Se trata de rutinas altamente optimizadas que implementan algoritmos que usan la librería BLAS (Basic Linear Algebra Subprograms).

BLAS (Basic Linear Algebra Subprograms)

Son subrutinas para cálculos de álgebra lineal comunes, cuenta con mecanismos para construir bloques de operaciones vectoriales y matriciales, el nivel 1 BLAS permite realizar operaciones escalares, vectoriales y vector-vector, el nivel 2 BLAS realiza operaciones matriz-vector y el nivel 3 BLAS realiza operaciones matriz-matriz, un objetivo importante de la BLAS es proporcionar una capa de portabilidad para el cálculo.

ScaLAPACK

ScaLAPACK es una biblioteca de rutinas de álgebra lineal de alto rendimiento para equi-

pos MIMD (Multiple Instruction, Multiple Data), es un sistema con un flujo de múltiples instrucciones que operan sobre múltiples datos de paso de mensajes de memoria distribuida y redes de estaciones de trabajo. Las rutinas de ScaLAPACK se basan en algoritmos de bloque de particiones con el fin de minimizar la frecuencia de movimiento de datos entre los diferentes niveles de la jerarquía de memoria (Yoginath *et al.*, 2005). ScaLAPACK es la continuación del proyecto LAPACK, el cual diseña software para estaciones de trabajo, super computadoras vectoriales y computadoras paralelas de memoria compartida.

2.6.1. Estructura Funcional

ScaLAPACK se ejecutará en cualquier máquina donde el BLAS y el BLACS están disponibles. La biblioteca está actualmente escrita en Fortran 77 y cuenta con rutinas escritas en C para el manejo de problemas aritméticos.

En la Figura 2.9 se describe la jerarquía del software ScaLAPACK. Los componentes por debajo de la línea, etiquetados como 'Local', son llamados en un solo procesador, con argumentos almacenados en un único procesador. Los componentes encima de la línea, etiquetados como 'Global', son rutinas paralelas síncronas, cuyos argumentos incluyen matrices y vectores distribuidos a través de múltiples procesadores, para mayores detalles ver Blackford *et al.* (2012).

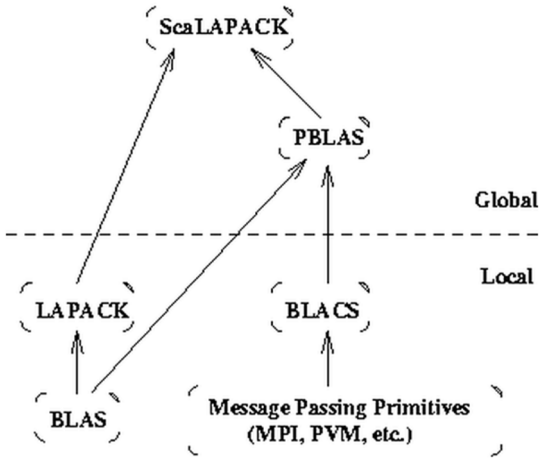

Figura 2.9: Jerarquía software de ScaLAPACK.

A continuación se describen brevemente cada unos de los elementos en la Figura 2.9.

PBLAS (Parallel Basic Linear Algebra Subprograms)

Para simplificar el diseño de ScaLAPACK, y por que las rutinas incluidas en BLAS han demostrado ser herramientas útiles fuera de LAPACK, se optó por construir un conjunto paralelo de BLAS, llamado PBLAS, que optimizan el paso de mensajes y cuya interfaz

es lo similar a la de BLAS. Esta decisión ha permitido que el código ScaLAPACK sea bastante similar con el código de LAPACK.

BLACS (Basic Linear Algebra Communication Subprograms)

BLACS es una biblioteca de paso de mensajes diseñado para el álgebra lineal. El modelo computacional consiste en mallas de procesos bidimensionales, donde cada proceso almacena fragmentos de las matrices y vectores. La biblioteca incluyen sincronización de envío/recepción de rutinas para comunicarse con una matriz o submatriz de un proceso a otro, para transmitir submatrices hacia muchos procesos, o para calcular las reducciones globales (sumas, máximos y mínimos). También hay rutinas para construir, cambiar o consultar la malla de procesos.

La librería ScaLAPACK se puede descargar del sitio oficial http://www.netlib.org, el sitio tiene información detallada para realizar el proceso de compilación e instalación, cuenta además con paquetes pre-compilados para diferentes distribuciones de Linux y Cygwin. La ultima versión estable es la 2.0.2 (01/May/2012).

La Tabla 2.1 muestra las direcciones web desde donde se pueden obtener las librerías BLACS, BLAS, LAPACK y MPI.

Tabla 2.1: Resumen de direcciones web para obtener las librerías de ScaLAPACK y Open MPI.

Librerías	Dirección Web	Versión
BLACS	http://www.netlib.org/blacs/	5-May-1997 (Versión 1.1)
LAPACK	http://www.netlib.org/lapack/	16-Nov-2013 (Versión 3.5.0)
BLAS	http://www.netlib.org/blas/	Nov-2013 (Versión 3.5.0)
MPI	http://www.open-mpi.org/	25-Ago-2015 (Versión 1.10.0)

Capítulo 3

Algoritmos para entrenamiento de una RNA

En este capítulo se presenta la propuesta de los algoritmos para entrenamiento de una RNA de una sola capa oculta. Se describe también brevemente las rutinas en C++ necesarias para implementar dichos algoritmos. Esta sección está dividida en las partes siguientes: 1) Descripción del problema, 2) Algoritmos para el entrenamiento, 4) Cómputo paralelo y 4) Implementación MPI/C++/ScaLAPACK.

3.1. Descripción del problema

Dados un vector de observaciones $\boldsymbol{y}$ y una matriz de incidencias $\boldsymbol{X}$, considere el problema de ajustar el modelo de regresión no lineal dado por:

$$y_i = \beta_0 + \overbrace{\sum_{k=1}^{s} w_k \underbrace{g_k\left(b_k + \sum_{j=1}^{p} x_{ij}\beta_j^{[k]}\right)}_{\text{Salida de la capa oculta}}}^{\text{Combinación de las salidas de la capa oculta}} + e_i \tag{3.1}$$

donde $(w_1, \ldots, w_s)'$ son pesos de las neuronas; $(b_1, \ldots, b_s)'$ son llamados interceptos (sesgos) en la máquina de aprendizaje; $(\beta_1^{[1]}, \ldots, \beta_p^{[1]}, \ldots, \beta_1^{[s]}, \ldots, \beta_p^{[s]}, \beta_0)'$ son coeficientes de regresión, donde $\beta_j^{[k]}$ es un coeficiente de regresión asociado a la variable $j = 1, \ldots, p$ y la neurona $k = 1, \ldots, s$, y $g_k(\cdot)$ es la función de activación y se supone que ε_i se distribuyen como variables aleatorias normales independientes e idénticamente distribuidas con media cero y varianza σ_e^2.

El modelo dado en (3.1) se presenta esquemáticamente en la Figura 3.1 y corresponde a una RNA con una sola capa oculta descrita brevemente en el Capítulo 2. De la ecuación (3.1) y la Figura 3.1 es claro que la predicción se puede hacer en 2 pasos: 1) Las entradas se transforman en forma no lineal en la capa oculta con s neuronas, 2) Las salidas de cada una de las neuronas se ponderan y se combinan linealmente.

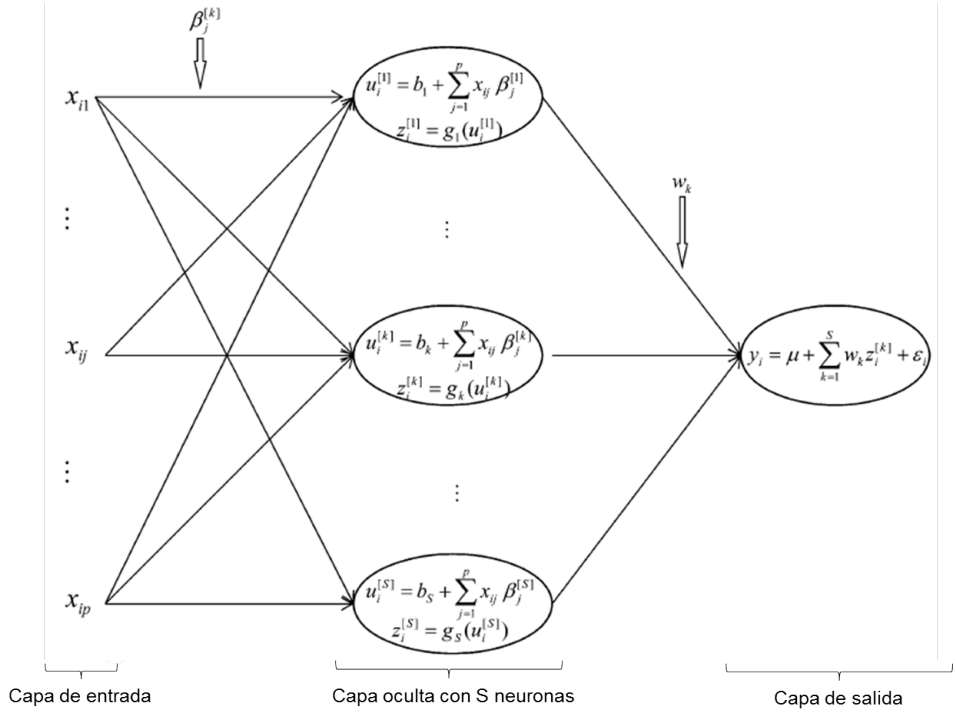

Figura 3.1: Ejemplo de la red neuronal de la ecuación (3.1).

El problema de "entrenamiento" de una red como la descrita anteriormente consiste en encontrar valores "optimos" para todas las cantidades desconocidas, es decir, $(w_1, \ldots, w_s)'$, $(b_1, \ldots, b_s)'$, $(\beta_1^{[1]}, \ldots, \beta_p^{[1]}, \ldots, \beta_1^{[s]}, \ldots, \beta_p^{[s]}, \beta_0)'$ y σ_e^2. Desde el punto de vista estadístico se trata de resolver un problema de "estimación".

3.2. Algoritmos de Estimación

Existen varias estrategias que permiten resolver el problema de estimación, por ejemplo mínimos cuadrados, técnicas de estadística Bayesiana, etc. En el presente trabajo nos enfocamos a resolver el problema utilizando técnicas de estadística Bayesiana, mismas que han mostrado ser de gran utilidad en este tipo de problemas donde regularmente, dado el gran número de parámetros a estimar se presenta el problema de "sobre ajuste". El problema de estimación en este tipo de redes ha sido revisado ampliamente por varios autores, por ejemplo Foresee y Hagan (1997), Gianola *et al.* (2011), Pérez-Rodríguez *et*

al. (2013), entre muchos otros. Sin embargo, no existen en la literatura propuestas para resolver el problema usando técnicas de cómputo paralelo. En lo que sigue, se describe el algoritmo que permite resolver el problema de estimación, gran parte del trabajo está basada en la publicación de Pérez-Rodríguez *et al.* (2013).

Sea $\boldsymbol{\theta} = \left(w_1, \ldots, w_s, b_1, \ldots, b_s, \beta_1^{[1]}, \ldots, \beta_p^{[1]}, \ldots, \beta_1^{[s]}, \ldots, \beta_p^{[s]}, \beta_0\right)'$ con p el número de predictores y s el número de neuronas. Suponga que $\theta_j \sim N(0, \sigma_\theta^2)$. El conjunto de parámetros a estimar para la RNA de una sola capa oculta son $\{\boldsymbol{\theta}, \sigma_e^2, \sigma_\theta^2\}$.

Foresee y Hagan (1997) demostraron que es posible resolver el problema de estimación usando dos pasos, mismos que se repiten de forma cíclica hasta alcanzar la convergencia. En el paso 1 se suponen conocidos los componentes de varianza $(\sigma_e^2, \sigma_\theta^2)$ y se actualiza $\boldsymbol{\theta}$. En el paso 2, se supone conocido θ y se actualizan los componentes de varianza. La Figura 3.2 presenta esquemáticamente el proceso descrito.

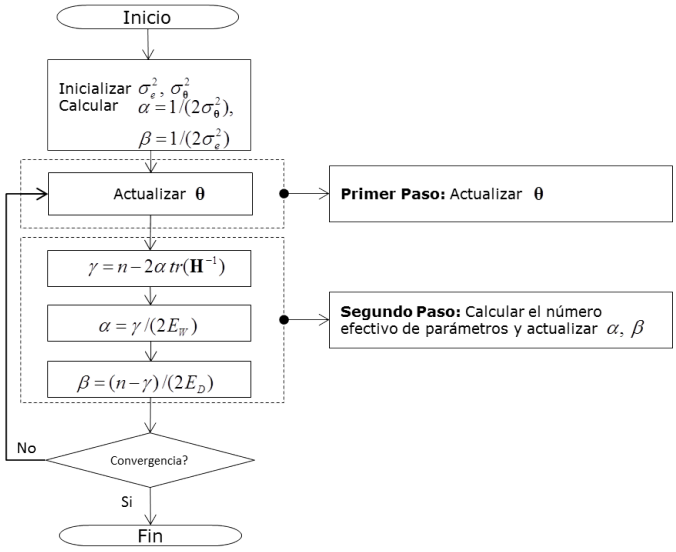

Figura 3.2: Diagrama de flujo del algoritmo empleado para el ajuste de la red neuronal.

Estrategia general de actualización de parámetros.

A continuación se describe detalladamente la estrategia para actualizar $\{\boldsymbol{\theta}, \sigma_e^2, \sigma_\theta^2\}$.

1. Dado σ_e^2 y σ_θ^2, actualizar $\boldsymbol{\theta}$. Esto se puede realizar minimizando la "suma de cuadrados del error aumentada", esto es:

$$F(\boldsymbol{\theta}) = \frac{1}{2\sigma_e^2}\sum_{i=1}^{n} e_i^2 + \frac{1}{2\sigma_\theta^2}\sum_{j=1}^{m}\theta_j^2 \; = \beta E_D + \alpha E_w,$$

donde $\beta = \frac{1}{2\sigma_e^2}$ y $\alpha = \frac{1}{2\sigma_\theta^2}$. En el presente trabajo se utilizó el algoritmo de Levenberg-Marquardt para minimizar $F(\boldsymbol{\theta})$.

2. Dado $\boldsymbol{\theta}$, actualizar σ_e^2 y σ_θ^2. Para esto, se utilizan las expresiones siguientes:

$$\alpha = \frac{\gamma}{2E_w} \quad \beta = \frac{n-\gamma}{2E_D} \quad \gamma = m - 2\alpha tr(\boldsymbol{H})^{-1},$$

donde γ es un estimador del número efectivo de parámetros, E_w es la suma de los cuadrados de los sesgos y pesos $\left(\sum_{j=1}^{m} \theta_j^2\right)$, E_D es la suma de los cuadrados de las diferencias entre los valores observados y predichos $(\sum_{i=1}^{n} e_i^2)$, m es el número total de los parámetros de la red, $\boldsymbol{H}$ es la matriz Hessiana la cual se aproxima mediante la ecuación $\boldsymbol{H} = \nabla^2 F(w) \approx 2\beta \boldsymbol{J^t J} + 2\alpha \boldsymbol{I}_N$, $\boldsymbol{J}$ es una matriz jacobiana de los errores del conjunto de entrenamiento, e $\boldsymbol{I}$ la matriz identidad (Foresee y Hagan, 1997).

La Figura 3.3 presenta el algoritmo de Levenberg-Marquardt.

3.3. Cómputo Paralelo

Todo algoritmo que se utiliza para resolver un problema, tiene asociado un orden de complejidad. Este orden de complejidad está dado por el número máximo de pasos que deberá ejecutar el algoritmo para resolver el problema en función del tamaño de la entrada. No es suficiente conocer la cantidad de pasos que utilizará el algoritmo para cada tamaño de instancia o entrada, es necesario conocer el orden de crecimiento de la cantidad de pasos que realizará el algoritmo, este orden de crecimiento se describe con la llamada notación O.

La notación O se utiliza para medir la complejidad de un algoritmo, el cual se representa como $O(f(n))$, esta función define el orden de complejidad, donde $f(n)$ es la razón de crecimiento del algoritmo (Mañas, 1997). Con esta definición, si $f(n)$ es un polinomio de grado k, el algoritmo correspondiente tendrá una complejidad de $O(n^k)$, independientemente de cual sean los coeficientes y términos de menor grado del polinomio. Si un algoritmo tiene una complejidad $O(n^k)$, se dice que es polinomial (en el caso de $k = 1$ se le llama lineal, para $k = 2$ cuadrática y para $k = 34$ cúbica); el resto de algoritmos se conocen como no-polinomiales (Ruz, 2003). La Tabla 3.1, muestran los distintos órdenes de complejidad de los algoritmos.

El algoritmo desarrollado en el presente trabajo realiza varias operaciones con matrices, por ejemplo productos e inversas. Los procesos que involucran operaciones con matrices tiene una complejidad de $O(n^3)$, mientras que la factorización e inversa por el método de Cholesky tiene una complejidad doble, por una parte la factorización de orden $(1/3)n^3$

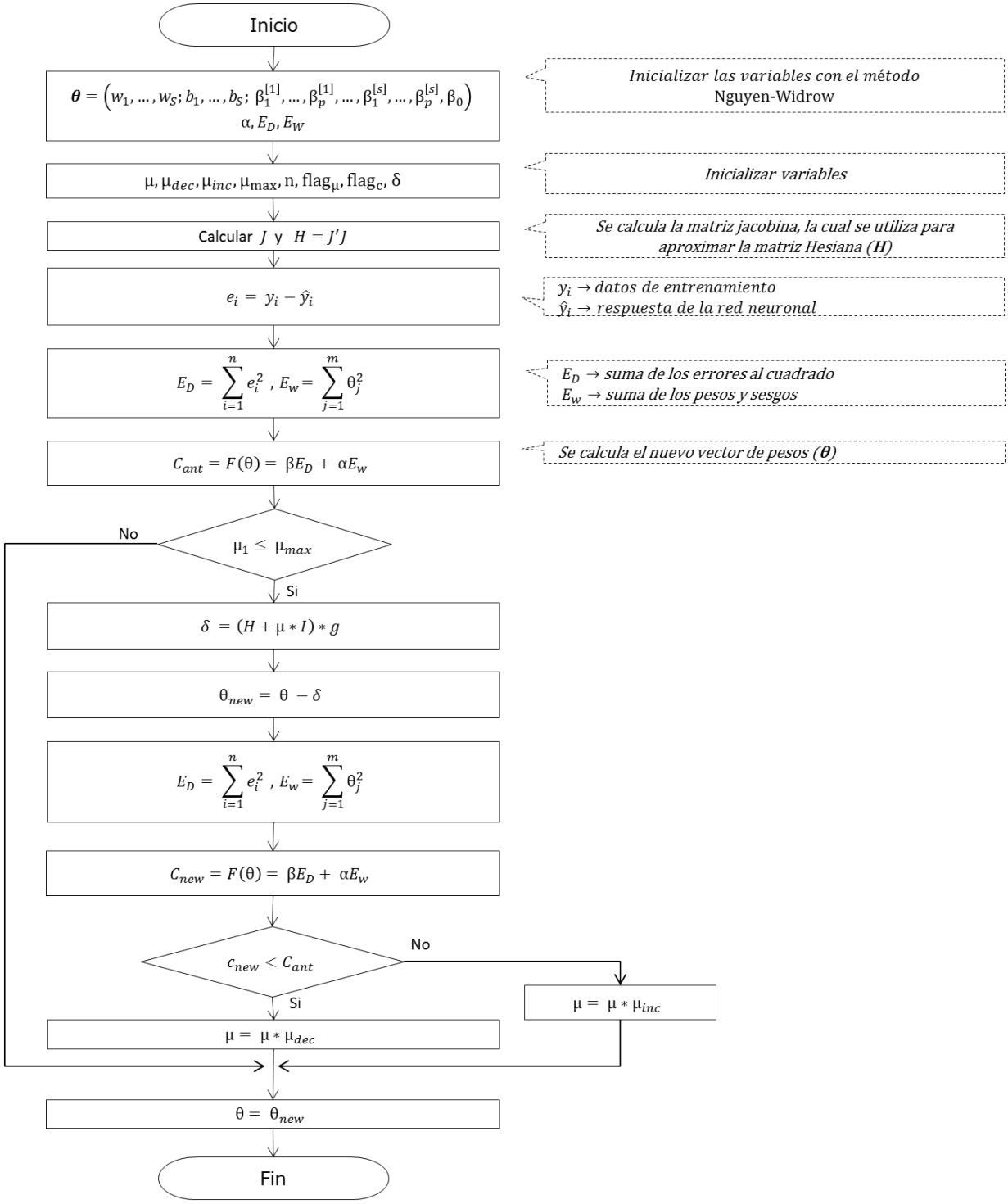

Figura 3.3: Algoritmo de Levenberg-Marquardt, donde se muestra como se realiza la actualización del vector $\boldsymbol{\theta}$.

y la solución del sistema triangular de ecuaciones de complejidad $2n^2$, mientras que las operaciones vector-matriz tiene un orden de $O(n^2)$ (Pérez-Bordas, 2000).

Tabla 3.1: Órdenes de complejidad de un algoritmo.

Orden	Descripción
$O(1)$	orden constante
$O(\log n)$	orden logaritmico
$O(n)$	orden lineal
$O(n^2)$	orden cuadrático
$O(n^a)$	orden polinomial (a>2)
$O(n^n)$	orden exponencial (n>2)
$O(n!)$	orden factorial

El algoritmo cuenta con diferentes órdenes de complejidad lo que afecta el rendimiento y provoca que el tiempo de ejecución aumente a medida que se incrementa el número de datos, es por ello que se propone usar cómputo paralelo como una alternativa para mejorar el rendimiento del programa.

Las operaciones matriciales son utilizadas ampliamente en distintas áreas de cómputo, donde la cantidad de cálculos en un corto tiempo es crucial, ya que esto conlleva un alto costo de procesamiento, y más si de estos cálculos dependen otros procesos por lo que se busca una respuesta ágil a estas operaciones. Bajo este contexto la búsqueda de mayor eficiencia a partir de la paralelización, ha sido el centro de un gran esfuerzo a través del tiempo (Zenón *et al.*, 2014).

En la Figura 3.4, se muestra el diagrama de flujo del algoritmo de ajuste del modelo (3.1), en el cual se identificaron tres partes que se pueden paralelizar para mejorar el rendimiento del algoritmo, en estas tres partes se realizan operaciones con matrices, de tal manera que el aumento en el tamaño de las matrices implica realizar más operaciones, por lo tanto el tiempo de ejecución se incrementa considerablemente. En la Figura 3.4 se identifican 3 bloques de operaciones que pueden parelizarse y que son las operaciones más costosas en el algoritmo general. A continuación se describen brevemente los elementos de cada uno de estos bloques, así como las estrategias generales para realizar las operaciones de forma paralela.

Bloque 1

El primer elemento que se puede paralelizar es un producto de la matriz jacobiana $(\boldsymbol{H} = \boldsymbol{J}'\boldsymbol{J})$, que es utilizado para realizar una aproximación de la matriz Hessiana, el cual esta remarcado en la Figura 3.4 con el número **1**. Una matriz jacobiana es una matriz formada por las derivadas parciales de primer orden de una función. Una de las aplicaciones más interesantes de esta matriz es la posibilidad de aproximar linealmente a la función en un punto. En este sentido, el Jacobiano representa la derivada de una función multivariable (Andújan *et al.*, 2004). La matriz Hesiana es una matriz cuadrada de $n \times n$, de las segundas derivadas parciales, sirven para verificar si el punto crítico que se esta tratando es un máximo, mínimo, punto silla o simplemente no puede determinarse.

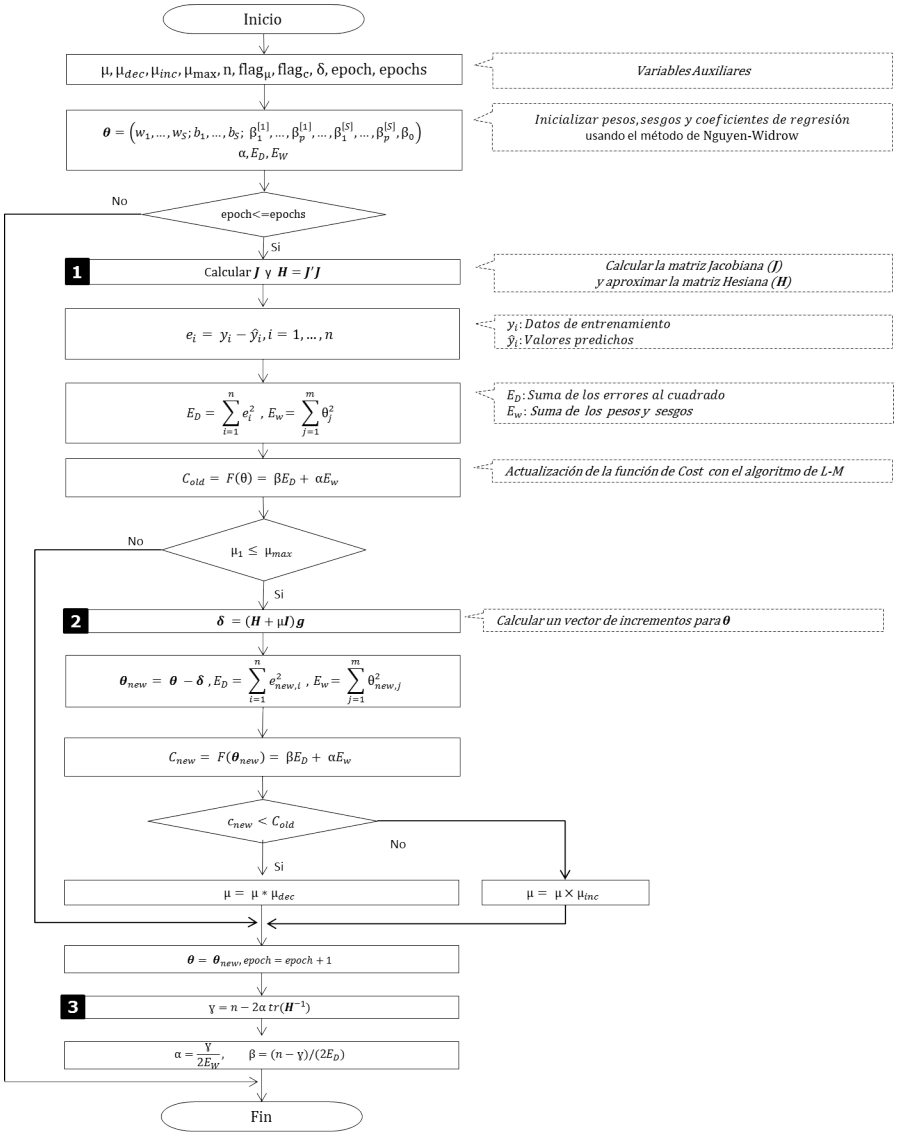

Figura 3.4: Diagrama de flujo del algoritmo, identificando los elementos paralelizables, en este diagrama se identificaron los tres bloques de operaciones [1,2,3] que se pueden paralelizar, estos bloques son los que demandan más tiempo de ejecución, para lo cual se utilizara ScaLAPACK y MPI para resolver estas tareas en paralelo.

Con la finalidad de obtener $\boldsymbol{H}$ se utilizan operaciones matriciales por bloques.

Matrices por bloques

Para realizar el producto matricial se pueden utilizar las matrices particionadas o en bloques. Para realizar ciertos cálculos resulta conveniente, repartir los elementos de una matriz $\boldsymbol{A}$, mediante rectas verticales y horizontales, en submatrices que denominaremos bloques de $\boldsymbol{A}$.

$$\boldsymbol{A} = \left[\begin{array}{cccccc} 1 & 2 & \vdots & 3 & 4 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ 5 & 6 & \vdots & 7 & 8 \\ 9 & 1 & \vdots & 2 & 3 \end{array}\right] = \begin{bmatrix} A_{11} & \vdots & A_{12} \\ \cdots & \cdots & \cdots \\ A_{21} & \vdots & A_{22} \end{bmatrix} \Rightarrow \begin{cases} A_{11} = (1\ 2) \\ A_{12} = (3\ 4) \\ A_{21} = \begin{bmatrix} 5 & 6 \\ 9 & 1 \end{bmatrix} \\ A_{22} = \begin{bmatrix} 7 & 8 \\ 2 & 3 \end{bmatrix} \end{cases}$$

Las operaciones entre matrices por bloques se realizan de forma análoga a las operaciones entre matrices, con la única restricción de que los bloques, sean conformables. Para realizar la multiplicación de dos matrices $\boldsymbol{A}$ y $\boldsymbol{B}$ por bloques es necesario tener en cuenta las siguientes restricciones:

1. El número de columnas del bloque $\boldsymbol{A}$, debe de ser igual al número de filas del bloque $\boldsymbol{B}$, es decir los bloques necesitan ser conformables.
2. Los bloques correspondientes en la suma deben de ser conformables para que se pueda realizar la multiplicacion de los elementos.

$$\boldsymbol{A} = \begin{bmatrix} A_{11} & \vdots & A_{12} \\ \cdots & \cdots & \cdots \\ A_{21} & \vdots & A_{22} \end{bmatrix}, y\ \boldsymbol{B} = \begin{bmatrix} B_{11} & \vdots & B_{12} & \vdots & B_{13} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ A_{21} & \vdots & A_{22} & \vdots & B_{23} \end{bmatrix},\ \text{entonces,}$$

$$\boldsymbol{AB} = \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & \vdots & A_{11}B_{12} + A_{12}B_{22} & \vdots & A_{11}B_{13} + A_{12}B_{23} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ A_{21}B_{11} + A_{22}B_{21} & \vdots & A_{21}B_{12} + A_{22}B_{22} & \vdots & A_{21}B_{13} + A_{22}B_{23} \end{bmatrix}$$

El dividir una matriz en bloques $\boldsymbol{A}_{p\times q}$ implica tener un conjunto de q columnas cada una de las cuales es un vector de p elementos; o bien, un conjunto de p filas cada una de las cuales es un vector de q componentes,

$$\boldsymbol{A}\ \boldsymbol{B} = \begin{bmatrix} \boldsymbol{a_1} \\ \boldsymbol{a_2} \\ \vdots \\ \boldsymbol{a_n} \end{bmatrix} \begin{bmatrix} b_1 & b_2 & \cdots & b_n \end{bmatrix},$$

de manera que cada elementos del producto es el producto escalar de la fila i-ésima por la columna j-ésima.

Bloque 2

Note que el bloque 2 incluye la solución de un sistema de ecuaciones lineales, estos sistemas se pueden expresar en forma matricial como:

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}, \tag{3.2}$$

donde dada una matriz $\boldsymbol{A}$ de orden $N \times N$ y un vector $\boldsymbol{b}$, se busca determinar el vector $\boldsymbol{x}$.

El sistema lineal de ecuaciones (3.2), puede solucionarse utilizando la factorización de Cholesky para matrices simétricas positivas (Tinetti y Denham, 2012). Note que $\boldsymbol{A} = \boldsymbol{L}\boldsymbol{L}'$, donde el factor de Cholesky $\boldsymbol{L}$ es una matriz triangular inferior con elementos diagonales positivos. Con esta factorización, el vector solución del sistema $\boldsymbol{x}$, puede ser calculado a través de sustituciones sucesivas hacia adelante (Forward Substitution) y hacia atrás (Backward Substitution) para resolver los sistemas triangulares $\boldsymbol{L}\boldsymbol{y} = \boldsymbol{b}$ y $\boldsymbol{L}'\boldsymbol{x} = \boldsymbol{b}$ (Santamaría, 1999).

El segundo elemento que se paralelizará será la solución del siguiente sistema de ecuaciones:

$$\boldsymbol{\delta} = \left(\boldsymbol{H} + \mu\boldsymbol{I}\right) \times \boldsymbol{g}, \tag{3.3}$$

donde $\boldsymbol{H}$ es la matriz Hessiana, μ es una constante e $\boldsymbol{I}$ la matriz identidad.

La ecuación (3.3) se resuelve utilizando la factorización de Cholesky. Note que (3.3) puede expresarse en términos de un sistema similar a (3.2), es decir: $\boldsymbol{A} := \boldsymbol{H} + \mu \times \boldsymbol{I}$, $\boldsymbol{x} = \boldsymbol{g}$, y $\boldsymbol{b} = \boldsymbol{\delta}$.

El sistema de ecuaciones (3.3) es resuelto utilizando rutinas de ScaLAPACK, la cual utiliza rutinas de BLAS y LAPACK para realizar los cálculos aritméticos y MPI para la comunicación. La solución paralela propuesta consiste en dividir en bloques la matriz de coeficientes $\boldsymbol{A}$ como se muestra en la Figura 3.5.

$$\overbrace{\begin{pmatrix} a_{11} & a_{21} & a_{31} & a_{41} & \dots \\ a_{21} & a_{22} & a_{32} & a_{42} & \dots \\ a_{31} & a_{32} & a_{33} & a_{43} & \dots \\ a_{41} & a_{42} & a_{43} & a_{44} & \dots \\ \dots & \dots & \dots & \dots & \ddots \end{pmatrix}}^{A} = \overbrace{\begin{pmatrix} l_{11} & 0 & \dots & \dots & \dots \\ l_{21} & l_{22} & 0 & \dots & \dots \\ l_{31} & l_{32} & l_{33} & 0 & \dots \\ l_{41} & l_{42} & l_{43} & l_{44} & \dots \\ \dots & \dots & \dots & \dots & \ddots \end{pmatrix}}^{L} \times \overbrace{\begin{pmatrix} l_{11} & l_{21} & l_{31} & l_{41} & \dots \\ 0 & l_{22} & l_{32} & l_{42} & \dots \\ \dots & 0 & l_{33} & l_{43} & \dots \\ \dots & \dots & 0 & l_{44} & \dots \\ \dots & \dots & \dots & \dots & \ddots \end{pmatrix}}^{L^T}$$

Figura 3.5: División por bloques de la matriz **A=LL′**.

Los elementos de la matriz **L** se calculan con $O\left(\frac{1}{3}n^3\right)$ operaciones las cuales son:

$$l_{ij} = \left(a_{ij} - \sum_{k=1}^{j-1} l_{ik} \times l_{jk}\right)/l_{jj} \; ; \quad 1 \leq j < i \leq n \tag{3.4}$$

$$l_{ii} = \sqrt{a_{ii} - \sum_{k=1}^{i-1} l_{ik}^2} \; ; \quad 1 \leq i \leq n \tag{3.5}$$

La ecuación (3.5) determina los valores de la diagonal principal, la ecuación (3.4) determina los valores de la parte triangular inferior. El patrón de dependencia en los cálculos es claro a partir de las ecuaciones anteriores. Según la ecuación (3.4), para el cálculo de l_{ij} con $j < i$ se utilizan los datos de la fila i y la fila j hasta la columna $i-1$ de ambas filas. Según la ecuación (3.5), para el cálculo de l_{ii} se utilizan los datos de la misma fila i desde la primera columna hasta la columna $i-1$. De otra manera, la primera fila no tiene ninguna dependencia de datos, y teniendo una fila k calculada se puede calcular todos los elementos de la columna k de las filas siguientes y luego el elemento de la diagonal principal de la fila $k+1$ (Tinetti y Romero, 2010).

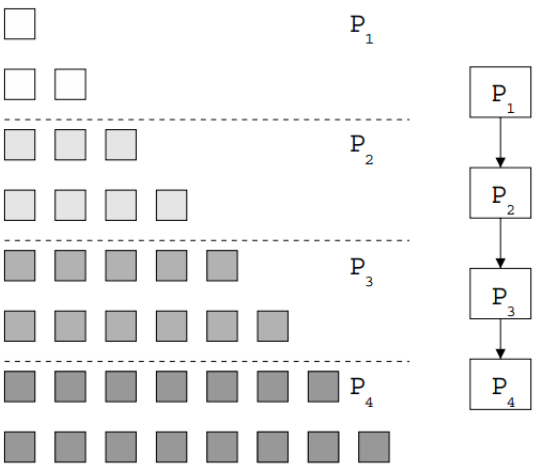

Figura 3.6: Ejemplo de una matriz de 8×8 elementos distribuidos entre cuatro núcleos, interconectados en un arreglo unidimiensional de núcleos, para este caso los datos (filas de $\boldsymbol{L}$) se comunican en una sola dirección, tal como lo indican las flechas (Tinetti y Romero, 2010).

Se puede conseguir una paralelización, avanzando por filas de la matriz, considerando a los procesos interconectados en un arreglo unidimensional. Distribuyendo la matriz a factorizar en bloques de filas, todos los núcleos tendrán datos de las columnas de la matriz original (y a factorizar) de acuerdo con las filas que tengan asignada. La Figura 3.6 se ilustra un ejemplo de la matriz en bloques paralelos.

Bloque 3

El tercer elemento que se paralelizará es la inversa de la matriz $\boldsymbol{H} = \boldsymbol{A}$, el problema consiste en encontrar una matriz $\boldsymbol{A}^{-1}$ tal que $\boldsymbol{A}\boldsymbol{A}^{-1} = \boldsymbol{I}$. La condición para que una matriz se pueda invertir es que no sea singular, es decir, que su determinante sea diferente de 0. Si hacemos, $\boldsymbol{A}\boldsymbol{A}^{-1} = \boldsymbol{I}$, entonces $\boldsymbol{A^{-1}} = \left[\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n\right]$, el cual resulta en un sistema matricial de $\boldsymbol{A}$ e $\boldsymbol{I}$, es decir:

$$\boldsymbol{A}\boldsymbol{A}^{-1} = \boldsymbol{A}\begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1n} \\ x_{21} & x_{22} & \cdots & x_{2n} \\ x_{31} & x_{32} & \cdots & x_{3n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{nn} \end{bmatrix} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix}.$$

Note que se tienen n sistemas de ecuaciones, los cuales se pueden resolver de forma independiente, esta característica es la que permite obtener de forma paralela la inversa de la matriz Hessiana, es decir:

$$\underbrace{\boldsymbol{A}\overbrace{\begin{bmatrix} x_{11} \\ x_{21} \\ x_{31} \\ \vdots \\ x_{n1} \end{bmatrix}^{\boldsymbol{x}_1} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}}_{Sistema\ 1}, \underbrace{\boldsymbol{A}\overbrace{\begin{bmatrix} x_{12} \\ x_{22} \\ x_{32} \\ \vdots \\ x_{n2} \end{bmatrix}^{\boldsymbol{x}_2} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}}_{Sistema\ 2}, \cdots, \underbrace{\boldsymbol{A}\overbrace{\begin{bmatrix} x_{1n} \\ x_{2n} \\ x_{3n} \\ \vdots \\ x_{nn} \end{bmatrix}^{\boldsymbol{x}_n} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 1 \end{bmatrix}}_{Sistema\ n}.$$

Al sustituir $\boldsymbol{A} = \boldsymbol{L}\boldsymbol{L}'$, se obtienen sistemas de ecuaciones que permiten obtener la inversa de una forma más sencilla, es decir,

$$\underbrace{\boldsymbol{A}\overbrace{\begin{bmatrix} x_{11} \\ x_{21} \\ x_{31} \\ \vdots \\ x_{n1} \end{bmatrix}^{\boldsymbol{x}_1} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}}_{Sistema\ 1}, \text{donde} \quad \boldsymbol{L}\boldsymbol{L}'\begin{bmatrix} x_{11} \\ x_{21} \\ x_{31} \\ \vdots \\ x_{n1} \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix},$$

donde $\boldsymbol{y}$ es un vector donde se almacenaran los resultados de la solución de los sistemas de ecuaciones triangular inferiores:

$$\boldsymbol{y_1} = \boldsymbol{L'} \begin{bmatrix} x_{11} \\ x_{21} \\ x_{31} \\ \vdots \\ x_{n1} \end{bmatrix}.$$

Para la solución es necesario resolver el siguiente sistema de ecuaciones, que como resultado proporcionara el vector de $\boldsymbol{y}$, con los valores necesarios para resolver el sistema triangular superior:

$$\boldsymbol{Ly_1} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \text{ sistema triangular inferior a resolver: } \boldsymbol{L} \begin{bmatrix} y_{11} \\ y_{21} \\ y_{31} \\ \vdots \\ y_{n1} \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}.$$

La solución de este sistema de ecuaciones proporcionara un vector de resultados $\boldsymbol{y}$, los cuales son empleados para resolver el sistema triangular superior de la siguiente manera:

$$\boldsymbol{L'} \underbrace{\begin{bmatrix} x_{11} \\ x_{21} \\ x_{31} \\ \vdots \\ x_{n1} \end{bmatrix}}_{x_k} = \begin{bmatrix} y_{11} \\ y_{21} \\ y_{31} \\ \vdots \\ y_{n1} \end{bmatrix}.$$

donde el vector $\boldsymbol{x}_k$, es la columna k-ésima inversa.

Este proceso se realiza de forma similar en todos los sistemas de ecuaciones a resolver. La Figura 3.7 muestra un esquema general del proceso de paralelización para obtener la solución de la inversa de una matriz.

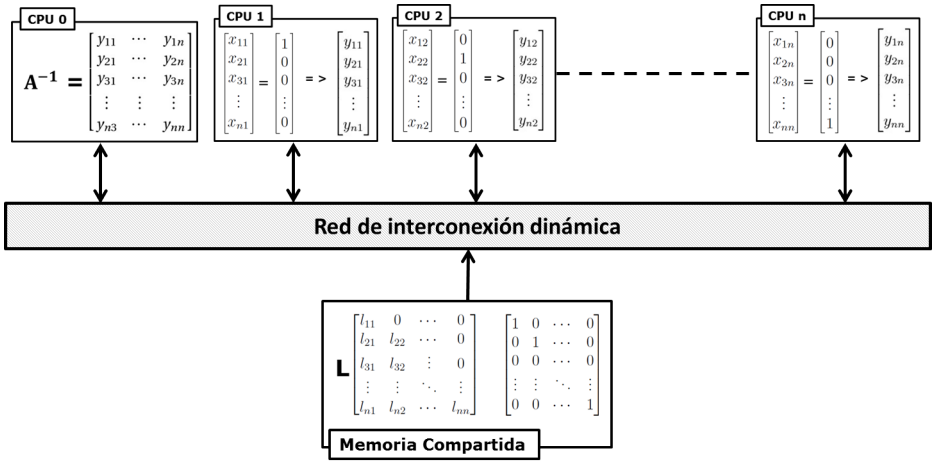

Figura 3.7: Inversión de una matriz mediante el algoritmo de Cholesky en paralelo. Las matrices $\boldsymbol{L}$, e $\boldsymbol{I}$, se cargan a memoria y los nodos resuelven sistemas de ecuaciones triangulares superiores e inferiores hasta obtener la inversa.

3.4. Implementación MPI/C++/ScaLAPACK

En esta sección, se describirán las funciones que integran la implementación del modelo de ajuste de la red neuronal artificial de la ecuación (3.1), se encuentra codificado en el lenguaje de programación C++, el cual utiliza las librerías de MPI para dividir y ejecutar las tareas en paralelo, para las operaciones de álgebra de matrices se utiliza la librería ScaLAPACK, esta última ayudará a ejecutar todas y cada una de las operaciones que involucren matrices, tales como productos de matrices, inversas, etc. Estas operaciones se realizan en paralelo para mejorar la eficiencia del algoritmo.

Funciones para implementar el algoritmo.

a) Correlación de Pearson (`cor_pearson`): Es un índice que mide la relación lineal entre dos variables cuantitativas, en este trabajo se utiliza para medir el grado de asociación lineal entre los valores de la variable respuesta en el conjunto de datos de entrenamiento, así como los valores predichos.

b) Máximo (`max`): Función que obtiene el numero máximo de un vector.

c) Leer matrices (`read_matrix`): Función para leer una matriz almacenada en un archivo de texto separado por comas (csv).

d) Leer vector (`read_vector`): Función para leer un vector columna almacenado en un archivo de texto.

e) Tangente hiperbólica (`tasing`).

f) Predicciones (`prediction_nn`): Función que obtiene las predicciones de la red neuronal con s neuronas.

g) Matriz jacobiana (`jacobian`).

h) Suma de cuadrados (`sc`): Función en la cual se realiza la suma de los cuadrados de los pesos, coeficientes de regresión e interceptos en una red.

i) Inicializando (`initnw`): Función que implementa el algoritmo de Nguyen y Widrow (1990) para inicializar los pesos, coeficientes de regresión e interceptos en una red.

j) Producto de Matrices (`matrix_crossprod`): Esta función tiene la capacidad de ejecutar productos de matrices en paralelo, esta tarea se realiza utilizando la librería de ScaLAPACK, la cual divide las matrices en submatrices y son enviadas a diferentes nodos con ayuda de MPI. Cuando se terminan de realizar los cálculos en los diferentes nodos, ScaLAPACK se encarga de reunir y agrupar los resultados, regresando al final el producto de las matrices. El producto de matrices sin procesamiento paralelo, implica tener un costo de ejecución de $O(n^3)$ (González *et al.*, 2012).

k) Producto Vector-Matriz (`matrix_vector_product`): Esta función al igual que la anterior, cuenta con la capacidad de ejecutar operaciones en paralelo, en donde se utilizan las mismas librerías (ScaLAPACK y MPI) y el mismo procedimiento, con la diferencia de que esta función realiza operaciones entre un vector y una matriz. El producto de un vector y una matriz sin procesamiento paralelo, implica tener un costo de ejecución de $O(n^2)$ (González *et al.*, 2012).

l) Solución de Cholesky (`solve_cholesky`): Resuelve un sistema de ecuaciones lineales con el método de Cholesky en paralelo.

m) Inversa de una matriz (`inverse`): Realiza la inversión de una matriz usando el algoritmo de Cholesky y procesamiento en paralelo.

Función principal (main)

La función principal invoca las funciones descritas anteriormente con la finalidad de ajustar una RNA regularizada Bayesiana. A continuación se describe brevemente las tareas que se realizan dentro de esta función:

1. Inicialización de MPI y ScaLAPACK.
2. Lectura de datos para entrenamiento y prueba.
3. Ajuste de la red.
4. Resumen de resultados.

El anexo A, muestra el código fuente del programa resultante, se supone que el usuario tiene instalado y configurado un compilador de C++, las librerías de OpenMPI y ScaLAPACK. El anexo B muestra las instrucciones necesarias para compilar la aplicación desde la línea de comandos de Unix/Linux.

Ejecución del programa

El programa resultante funciona desde la línea de comandos de Unix/Linux. Para ejecutarlo es necesario contar con la información siguiente.

1. Nombre del archivo ejecutable del programa.
2. Número de procesadores a utilizar.
3. Filas y columnas de la matriz de observaciones para las variables predictoras.
4. Número de neuronas.
5. Nombre del archivo con la matriz de observaciones para las variables predictoras.
6. Nombre del archivo con las observaciones para la variable respuesta.
7. Tamaño de la malla de cómputo.
8. Tamaño de bloques para dividir matrices en ScaLAPACK.
9. Nombre del archivo de variables predictoras los cuales pueden ser utilizados como datos de prueba. Este archivo es opcional.

La Figura 3.8, muestra la línea de comandos usada para ejecutar el programa.

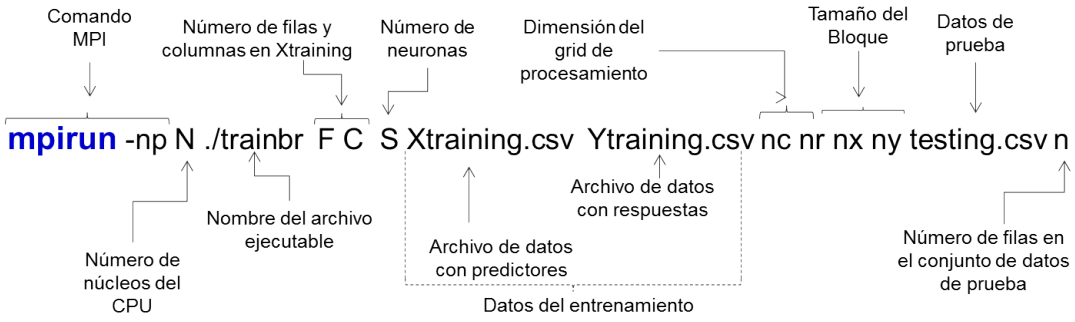

Figura 3.8: Comando de MPI para ajustar una red neuronal artificial en paralelo. La aplicación utiliza ScaLAPACK para realizar operaciones matriciales en paralelo y OpenMPI para realizar la comunicación entre procesos.

Capítulo 4

Aplicación y Desempeño del Software

En este capítulo se presentan dos ejemplos de aplicación de las RNA regularizadas Bayesianas. En el primer ejemplo se predice el consumo de electricidad utilizada en los sistemas de calefacción (calentamiento y enfriamiento) en edificios. Las predicciones de los consumos se realizan utilizando como variables predictoras algunas covariables ambientales, por ejemplo: temperatura, humedad, radiación solar, viento, etc. Los datos fueron analizados previamente por MacKay (1994). En el segundo ejemplo se analizan un conjunto de datos de rendimiento de trigo del Centro Internacional de Mejoramiento de Maíz y Trigo (CIMMyT, `http://www.cimmyt.org`). El objetivo en este caso es predecir el rendimiento de las líneas de trigo utilizando 1, 279 marcadores moleculares. Los datos fueron analizados previamente por Crossa *et al.* (2010) y muchos otros autores posteriormente. Finalmente se presentan los resultados de la evaluación del desempeño de la aplicación.

4.1. Ejemplo 1 (Consumos de energía)

Para este ejemplo se han seleccionado los datos reportados en el artículo de MacKay (1994). Los datos se descargaron del sitio `www.inference.phy.cam.ac.uk/mackay/`, en donde se encuentra una recopilación de sus investigaciones, así como los datos utilizados en sus artículos.

El conjunto de datos a analizar contiene información a nivel horario de temperatura, humedad, radiación solar y viento, para el periodo del 01 de septiembre de 1989 al 31 de diciembre de 1989. También se cuenta con información de consumo de electricidad, así como consumos de energía para los sistemas de calefacción (enfriamiento y calentamiento).

Se tiene un total de 2,929 registros. MacKay (1994) dividió los datos en dos conjuntos: entrenamiento y prueba. El conjunto de datos para la prueba constan de 1,282 registros. El interés del investigador en este caso se centró en predecir los consumos de energía utilizando como predictores las variables ambientales. La Figura 4.1 muestra la serie con los datos observados, así como las predicciones obtenidas por MacKay (1994) para el caso de electricidad. MacKay preproceso los datos originales y agregó variables que le permitieron obtener diferentes representaciones del tiempo y dias festivos, moviendo los promedios de las variables ambientales, etc., de manera que al final del preprocesamiento de los datos originales obtuvo un conjunto de datos con 25 variables que utilizo como entradas, el conjunto de datos preprocesados se pueden descargar del sitio web de MacKay. A modo de ejemplo, en el presente trabajo se realizan las predicciones de los consumos de energía utilizando el programa desarrollado.

La Figura 4.1 muestra los consumos de electricidad observados y predichos que reporta MacKay. En la gráfica también se incluyen los residuales, los cuales se obtienen como la diferencia entre valores observados y predichos por la red.

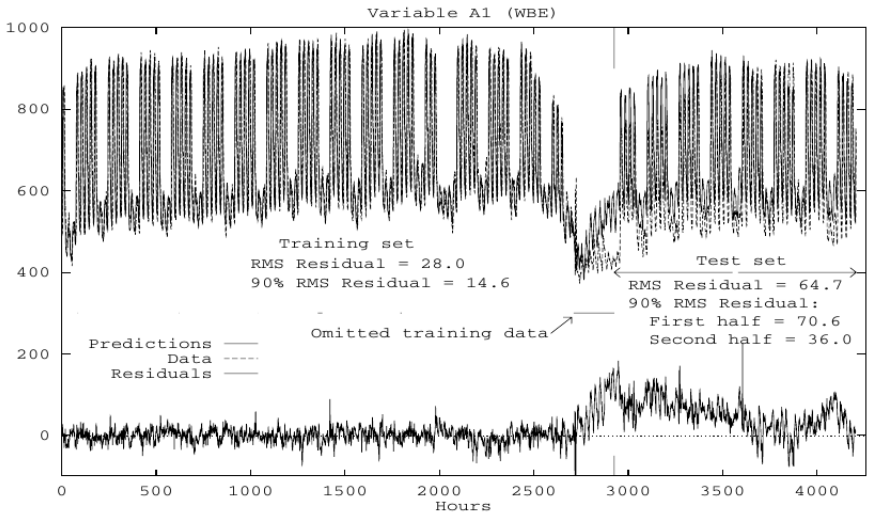

Figura 4.1: Consumo de electricidad, se muestra los valores observados y predichos con el algoritmo de MacKay (1994) y los residuos los cuales se representan como la diferencia de los valores observados y predichos.

Utilizando los datos analizados por MacKay se ajustó la RNA descrita con 8 neuronas. Los datos fueron previamente normalizados tanto para entrenamiento como para prueba. Una vez ajustado el modelo se realizaron las predicciones correspondientes, re-escalando las salidas para obtener la respuesta en la escala original. La RNA fue ajustada usando 4 núcleos de procesamiento y el tiempo de cómputo necesario para ejecutar la aplicación fue de 2 seg. La Figura 4.2 muestra los datos de entrenamiento y de prueba y en la Figura 4.3 se muestran los datos observados, predichos y los residuales. En general se observa que la red predice bastante bien los datos en el conjunto de prueba.

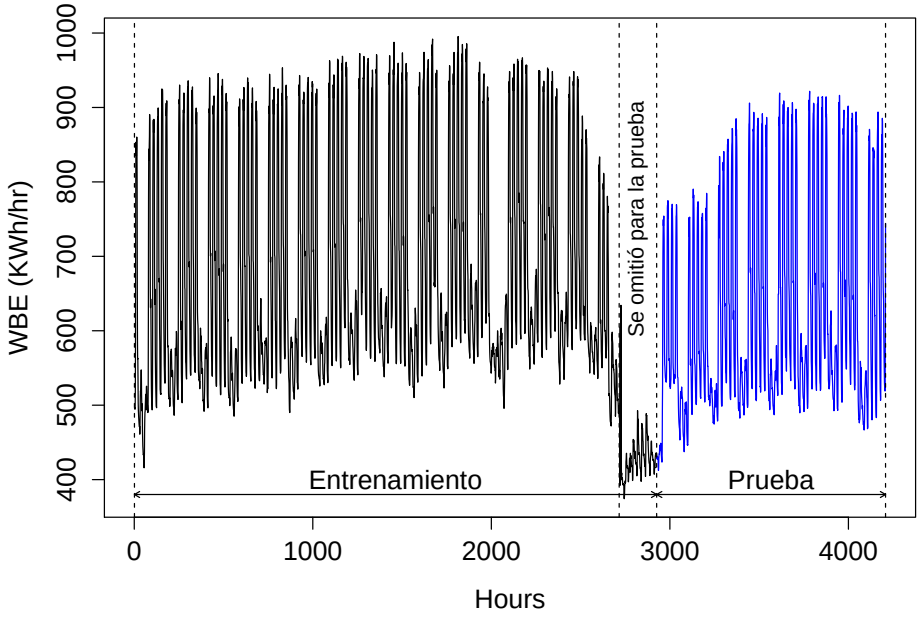

Figura 4.2: Consumo de electricidad, se muestran los datos de entrenamiento y prueba

4.2. Ejemplo 2 (Predicción de rendimiento en trigo)

Los datos utilizados en este ejemplo corresponden a rendimiento de 599 líneas de trigo. Las líneas fueron evaluadas en el programa de mejoramiento de trigo del Centro Internacional de Mejoramiento de Maíz y Trigo (CIMMyT, http://www.cimmyt.org). Las líneas de trigo fueron evaluadas en 4 mega ambientes. Las líneas fueron genotipeadas utilizando 1,279 marcadores moleculares (DArT) generados Triticarte Pty. Ltd., Canberra, Australia; http://www.triticarte.com.au. Los marcadores DArT toman solo dos valores: 0 y 1. Los datos fueron analizados inicialmente por Crossa *et al.* (2010) y pueden descargarse de http://www.genetics.org/content/suppl/2010/09/02/genetics.110.118521.DC1.

Desde el punto de vista de estadística se tiene un problema de predicción, la variable respuesta (y) corresponde al rendimiento y como predictores se tienen los marcadores moleculares ($x_1, \ldots, x_{1279}$). Crossa *et al.* (2010) utilizaron el modelo de regresión lineal múltiple para predecir los rendimientos, es decir,

$$y_i = \beta_0 + \sum_{j=1}^{1279} x_{ij}\beta_j, i = 1, \ldots, 599.$$

Con la finalidad de evaluar el poder predictivo del modelo propuesto, Crossa *et al.* (2010)

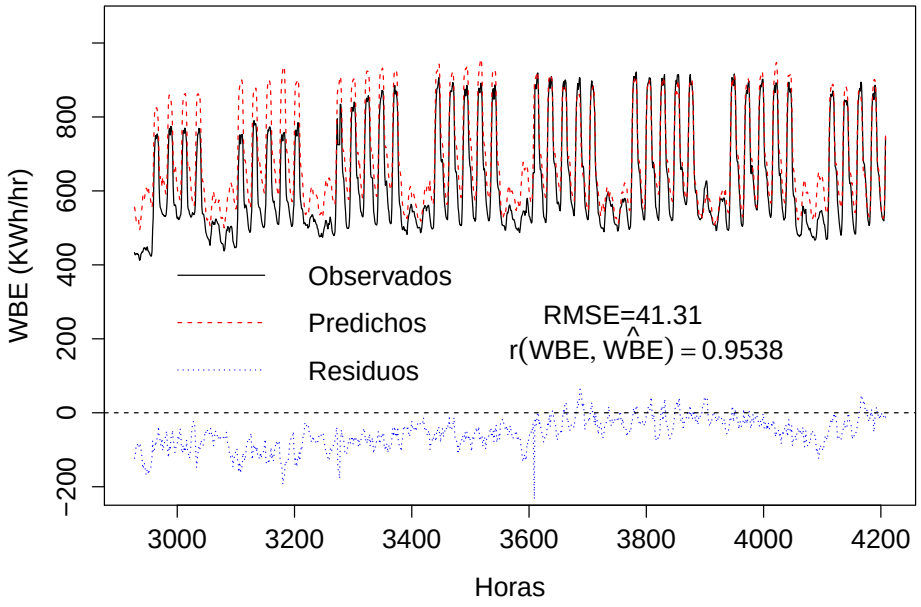

Figura 4.3: Consumo de electricidad, se muestra los valores observados y predichos así como los residuos los cuales se representan como la diferencia de los valores observados y predichos.

utilizaron validación cruzada con 10 grupos. La idea básica es ajustar el modelo utilizando una parte de los datos y predecir el resto. En el caso de validación cruzada con 10 grupos los datos se dividen 10 conjuntos disjuntos, $\{S_1, \ldots, S_{10}\}$. Por ejemplo, un posible conjunto de datos de entrenamiento se obtiene usando los datos en los conjuntos $\{S_2, \ldots, S_{10}\}$ y posteriormente se predice el rendimiento utilizando los marcadores en $\{S_1\}$, con lo cual se podrá obtener la correlación entre valores observados y predichos tanto para los datos en entrenamiento como para los datos en prueba. El mismo ejercicio se puede repetir para el resto de los subconjuntos.

A manera de ejercicio, en el presente trabajo se ajustó una RNA con 4 neuronas y usando 4 núcleos de procesamiento. El poder predictivo del modelo se estudió utilizando el esquema de validación cruzada con 10 grupos descrito anteriormente. La Tabla 4.1, presentan los resultados de la validación. El poder predictivo es bastante bueno, los resultados pueden ser comparados directamente con los reportados por Crossa *et al.* (2010).

Tabla 4.1: Correlaciones de Pearson para observaciones en los conjuntosde datos de entrenamiento y prueba, así como los tiempos de ejecución.

Grupo	r_{trn}	r_{test}	Tiempo (hrs)
1	0.9781	0.5900	2.8
2	0.9748	0.6474	1.9
3	0.9752	0.4692	2.5
4	0.9711	0.3642	1.9
5	0.9842	0.3298	4.3
6	0.9742	0.5569	2.9
7	0.9876	0.4070	3.2
8	0.9724	0.4362	6.1
9	0.9768	0.2276	2.9
10	0.9782	0.5312	3.8
Prom.	**0.9781**	**0.4559**	**3.23**

4.3. Evaluación del desempeño

Con el objetivo de evaluar el desempeño del programa desarrollado, se contemplaron diferentes escenarios. Se ajustó la RNA utilizando los datos de rendimiento de trigo descritos en la sección anterior. El modelo fue ajustado con s=1,2,3,4 neuronas usando N=1,2,4,6,8 núcleos de procesamiento.

Dado que los valores de inicialización para el algoritmo de ajuste de la red son aletorios (Nguyen y Widrow, 1990), cada una de las pruebas se repitió 5 veces, con la finalidad cuantificar la variabilidad y obtener un estimado de la media de los tiempos de ejecución de la aplicación. La evaluación se llevó a cabo en una estación de trabajo Linux con un procesador AMD Opteron (tm) con una velocidad de procesador @ 2.6 GHz y 160 GB de memoria RAM, en el cual se instaló y configuró la librería de OpenMPI y ScaLAPACK. La Figura 4.4, muestra el resultado de pruebas para evaluar el desempeño del algoritmo.

En la Figura 4.5 se muestran los tiempos de ejecución de los cuatro modelos. Se observa que el tiempo que tardó en ajustarse el modelo con una neurona fue de 1, 372 minutos, al aumentar el número de núcleos el tiempo disminuye considerablemente, en la prueba se alcanza un tiempo mínimo de ejecución al utilizar los ocho núcleos de procesamiento.

En la Figura 4.6, se muestra el tiempo de ejecución vs el número de neuronas. Se observa que a medida que aumentan el número de neuronas aumenta también el tiempo de ejecución del programa. Esto es ocasionado por el aumento en los parámetros que se necesitan calcular, el número de parámetros está determinado por la siguiente ecuación:

$$\text{Número de parámetros} = s \times (2 + p), \tag{4.1}$$

				Test 1	Test 2	Test 3	Test 4	Test 5	Promedio		
N°	Núcleos	Neuronas	Rejilla de Núcleos	seg	seg	seg	seg	seg	Segundos	Minutos	Horas
1	1	1	1 , 1	137	133	174	145	142	146	2	0.04
2		2		6468	6682	6140	10655	10533	8095	135	2.25
3		3		30888	42977	44381	47825	26835	38581	643	10.72
4		4		129658	56508	74624	51733	99346	82374	1373	22.88
5	2	1	1 , 2	93	90	90	84	81	88	1	0.02
6		2		5569	3516	2297	4512	2874	3754	63	1.04
7		3		16037	15337	18457	15867	13911	15922	265	4.42
8		4		41622	31874	64261	44104	45525	45477	758	12.63
9	4	1	2 , 2	41	43	40	41	43	41	1	0.01
10		2		1409	2313	1635	3460	3265	2417	40	0.67
11		3		6847	7436	5499	7671	6804	6852	114	1.90
12		4		9778	8835	10619	9380	10090	9740	162	2.71
13	6	1	2 , 3	16	15	16	16	16	16	0	0.00
14		2		612	541	374	511	539	515	9	0.14
15		3		2619	2268	1987	3280	2820	2595	43	0.72
16		4		5544	5302	7927	6519	5501	6158	103	1.71
17	8	1	2 , 4	13	13	15	13	12	13	0	0.00
18		2		485	382	582	515	307	454	8	0.13
19		3		1976	2049	1838	894	3567	2065	34	0.57
20		4		4136	4678	5072	5130	3667	4537	76	1.26

Figura 4.4: Escenarios de prueba de la aplicación.

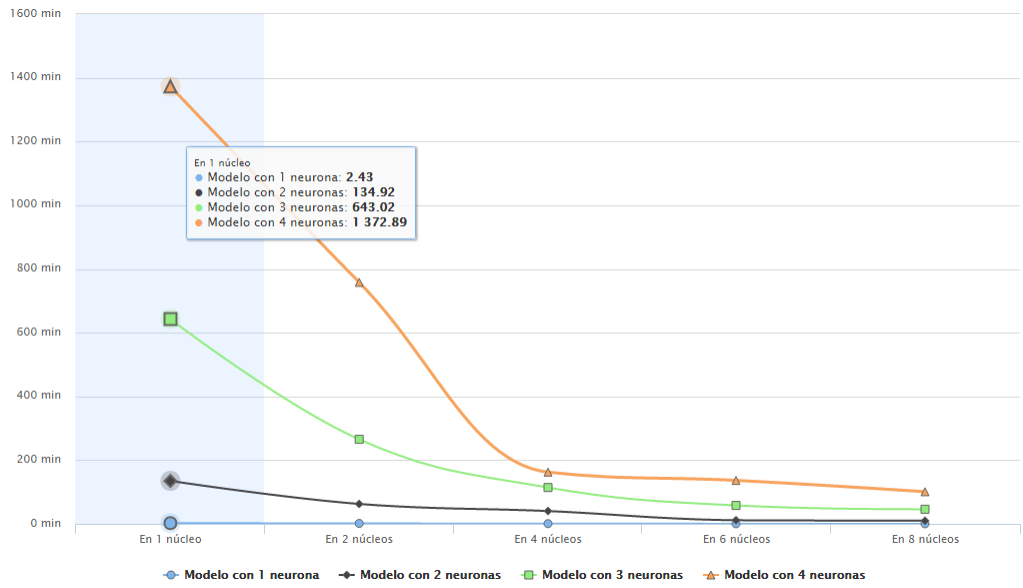

Figura 4.5: Tiempos de cálculo por modelo.

donde s es el número de neuronas del modelo que utiliza el algoritmo y p es el número de predictores.

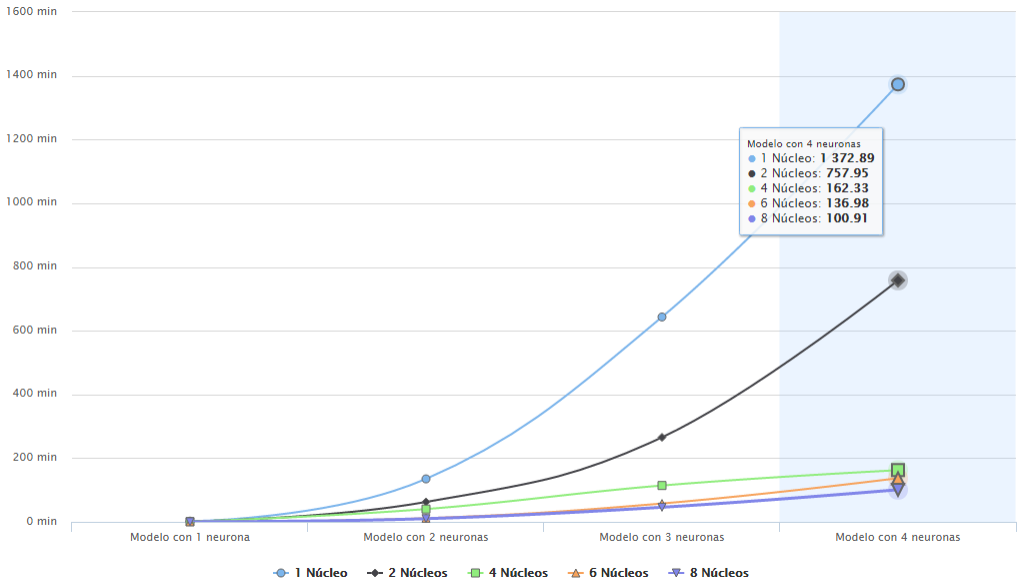

Figura 4.6: Tiempos de ejecución en función del número de neuronas.

Capítulo 5

Conclusiones y Recomendaciones

En el presente trabajo se propuso un algoritmo para el ajuste de una red neuronal regularizada utilizando cómputo paralelo. Los resultados de la evaluación del desempeño muestran que el algoritmo es eficiente, debido a que disminuye el tiempo de ejecución del programa que lo implementa al realizar más rápido las operaciones cuando se aumenta el número de procesadores.

La reducción del tiempo de ejecución se debe a que la carga de las operaciones se divide en los diferentes procesadores, sin embargo, es necesario realizar una buena planificación que permita detectar los elementos del algoritmo que se pueden paralelizar, esta es una tarea muy costosa para el programador, puesto debe de considerar los procesos que dependen de otros y los que demandan más tiempo de procesamiento.

El cómputo paralelo es una técnica que tiene el potencial de poder aumentar la eficiencia y velocidad de los programas. En la actualidad todas las computadoras cuentan con el hardware necesario para ejecutar tareas de forma paralela, sin embargo son pocas las aplicaciones que realmente aprovechan todo el potencial de la computadora, es necesario contar con más investigaciones de este tipo para demostrar que estas técnicas no son exclusivas de grandes centros de investigación o de grandes problemas científicos.

Temas de investigación futuros

Existen muchas librerías y software que permite implementar el cómputo paralelo, para dar seguimiento a este tema, se sugiere realizar una investigación la cual permita conocer las diferentes herramientas e implementaciones del cómputo paralelo.

Referencias

Almeida, F. (2008). *Introducción a la Programación Paralela*. Paraninfo Cengage Learning.

Amilcar, M. V. (2011). Introducción a gpu's y programación cuda para hpc. Departamento de Computación CINVESTAV-IPN / LUFAC Computación.

Andújan, J. M., Barragán, A. J., Gegúndez, M. E. y J, F. (2004). Aplicación de la matriz jacobiana de un sistema de control borroso a la obtención de sus estados de equilibrio. Inf. téc., Universidad de Huelva, E.P.S. La Rábida.

Barney, B. (2014). Message passing interface (mpi). Website. https://computing.llnl.gov/tutorials/mpi/#What,[15-03-2015].

Blackford, L. S., Choi, J., Cleary, A., DÁzevedo, E., Demmel, J., Dhillon, I., Dongarra, J., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D. y Whaley, R. C. (2012). Scalapack users'guide. Sito Oficial netlib. http://netlib.org/scalapack/slug/index.html,[14-03-2015].

Brío, B. y Molina, A. (2005). *Redes neuronales y sistemas difusos*. Alfaomega.

Crossa, J., de-los Campos, G., Pérez-Rodríguez, P., Gianola, D., Burgueño, J., Araus, J. L., Makumbi, D., Singh, R. P., Dreisigacker, S., Yan, J., Arief, V., Banziger, M. y Braun, H.-J. (2010). Prediction of Genetic Values of Quantitative Traints in Plant Breeding Using Pedigree and Molecular Markers. *Genetics*, 186, 2, 714–724.

Deitel, H. M. y Deitel, P. J. (2008). *C++ Cómo Programar*. Pearson Educación, 6ª edición.

Foresee, F. D. y Hagan, M. T. (1997). Gauss Newton Approximation to Bayesian Learning. *International Conference on*, 3, 1931–1933.

Freeman, J. y Skapura, D. (1993). *Redes Neuronales. Algoritmos, aplicaciones y técnicas de propagación*. Addison-Wesley.

Gianola, D., Okut, H., Weigel, K. A. y Rosa, G. J. (2011). Predicting complex quantitative traits with Bayesian neural networks: a case study with Jersey cows and wheat. *BMC Genetics*, 87, 1–14.

Girosi, F. y Poggio, T. (1989). Representation Properties of Networks: Kolmogorov's Theorem is Irrelevant. *Neural Computation*, 1, 4, 465–469. ISSN 0899-7667.

González, L. P. G., Cuenca, J. y Giménez, D. (2012). Técnicas de modelado y optimización del tiempo de ejecución de rutinas paralelas de álgebra lineal. Inf. téc., Universidad de Murcia.

Referencias

Gropp, W., Lusk, E. y Skjellum, A. (2014). *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. Scientific and Engineering Computation. MIT Press. ISBN 9780262527392.

Haykin, S. (1999). *Neural Networks: A Comprehensive Foundation*. International edition. Prentice Hall. ISBN 9780139083853.

Khare, M. y Nagendra, S. S. (2006). *Artificial Neural Networks in Vehicular Pollution Modelling*. Springer.

MacKay, D. J. (1994). Bayesian Non-Linear Modelling for the Prediction Competition. *ASHRAE transactions*, 100, 5–12.

Mañas, J. A. (1997). Análisis de algoritmos: Complejidad. Website. http://www.lab.dit.upm.es/~lprg/material/apuntes/o/, [25-10-2015].

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

Meyer, K. y Tier, B. (2013). Utility of graphics processing units for dense matrix calculations in computing and inverting genomic relationship matrices. Inf. téc., Animal Genetics and Breeding Unit, University of New England, Armidale.

Naiouf, R. M. (2004). *Procesamiento Paralelo. Balance de Cargas Dinámico en Algoritmos de Sorting*. Tesis Doctoral, Facultad de Ciencias Exactas Universidad Nacional de La Plata.

Navidi, W. (2006). *Estadística para ingenieros y científicos*. Mc Graw Hill, 9ª edición.

Nguyen, D. y Widrow, B. (1990). Improving the learning speed of 2-layer neural networks by choosing initial values of the adaptive weights. En *International Symposium on Neural Networks*.

Otto, S., Huss-Lederman, S. y Walker, D. (1996). *MPI: The complete Reference*. The MIT Press, segunda edición.

Pacheco, P. (1997). *Parallel Programming with MPI*. Morgan Kaufmann Publishers.

Pérez-Bordas, F. (2000). Implementación y evaluación de la factorización de Cholesky mediante tbb y threads en arquitecturas multicore. Website. http://upcommons.upc.edu/bitstream/handle/2099.1/10988/PFC.Cholesky.pdf?sequence=1, [25-10-2015].

Pérez-Rodríguez, P. y Gianola, D. (2015). brnn (bayesian regularization for feed-forward neural networks). R-project. https://cran.r-project.org/web/packages/brnn/index.html, [01-08-2015].

Pérez-Rodríguez, P., Gianola, D., Weigel, K. A., Rosa, G. J. M. y Crossa, J. (2013). Technical Note: An R package for fitting Bayesian regularized neural networks with applications in animal breeding. *Journal of Animal Science*, 91, 3522–3525.

R Core Team (2015). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria.

Referencias

Ruz, V. V. (2003). *Análisis de Algoritmos*. INACAP Copiapó.

Santamaría, M. J. M. (1999). Factorización de Cholesky modificada de matrices dispersas sobre multiprocesadores. Inf. téc., Universidad de Santiago de Compostela, Facultad de Física.

Simon, H. (1999). *Neuronal Networks*. Prentice Hall International, Inc.

Squyres, J. M. (2014). Open mpi. Website. http://www.aosabook.org/en/openmpi.html, [22-04-2015].

Tinetti, F. G. y Denham, M. (2012). Álgebra lineal en paralelo: Factorizaciones en clusters heterogéneos. Inf. téc., Instituto de Investigación en Informática LIDI.

Tinetti, F. G. y Romero, F. (2010). Factorización de matrices Cholesky: Paralelización y balanceo de cargas. Inf. téc., Facultad de Informática, UNLP, Argentina.

Triola, M. F. (2004). *Estadística*. Pearson Addison Wesley, 9ª edición.

University of Rochester, D. o. P. S. (2015). Message passing fundamentals. Website. http://www.rochester.edu/college/psc/thestarlab/help/MPI_Course.pdf,[14-03-2015].

Vargas-Lombardo, I. R. M. (2012). Principios y Campos de Aplicación en CUDA Programación paralela y sus potencialidades. *Nexo*, 25, 40–41.

Viñuela, P. y León, I. (2004). *Redes de neuronas artificiales: un enfoque práctico*. Pearson Educación - Prentice Hall. ISBN 9788420540252.

Walpole, R. E., Myers, R. H. y Myers, S. L. (1999). *Probabilidad y Estadística para ingenieros*. Pearson Educación, 6ª edición.

Walpole, R. E., Myers, R. H., Myers, S. L. y Ye, K. (2012). *Probabilidad y Estadística para ingeniería y ciencias*. Pearson, 9ª edición.

Wong, J. C. C. (2005). Ley de Moore, nanotecnología y nanociencia: síntesis y modificación de nanopartículas. *Revista Digital Universitaria, DGSCA-UNAM*, 6, 3–5.

Yoginath, S., Bauer, D., Samatova, N., Kora, G., Fann, G. y Geist, A. (2005). RScaLAPACK: High-Performance Parallel Statistical Computing with R and ScaLAPACK. *Journal of Physics*, 1–4.

Zenón, G. R., Apaza, R. G. y Tejerina, W. C. (2014). Programación paralela: implementación de clúster para resolución de multiplicación de matrices. Inf. téc., Facultad de Ingenieria, Universidad Nacional de Jujuy.

Anexos

Anexo A: Implementación C++/MPI/ScaLAPACK

```
/*
File: trainbr.cpp
*/

#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include <sys/time.h>

#include "mpi.h"

#ifdef F77_WITH_NO_UNDERSCORE
  #define   numroc_       numroc
  #define   descinit_     descinit
  #define   pdgemm_       pdgemm
  #define   pdgeadd_      pdgeadd
  #define   pdgemv_       pdgemv
  #define   pdgesv_       pdgesv
  #define   pdpotrf_      pdpotrf
  #define   pdpotri_      pdpotri
  #define   pdpotrs_      pdpotrs
  #define   Cblacs_barrier_ Cblacs_barrier
#endif

extern "C"
{
    extern void Cblacs_pinfo( int* mypnum, int* nprocs);
    extern void Cblacs_get( int context, int request, int* value);
    extern int Cblacs_gridinit( int* context, const char * order,
           int np_row, int np_col);
    extern void Cblacs_gridinfo( int context, int*  np_row,
           int* np_col, int*  my_row, int*  my_col);
    extern void Cblacs_gridexit( int context);
    extern void Cblacs_exit( int error_code);
    extern void Cdgesd2d(int ictxt, int m, int n, double* A,
```

```
36              int ld, int rdest, int cdest);
37      extern void Cdgerv2d(int ictxt, int m, int n, double* A,
38              int ld, int rsrc, int csrc);
39      extern int numroc_( int *n, int *nb, int *iproc,
40              int *isrcproc, int *nprocs);
41      extern void descinit_( int *desc, int *m, int *n,
42              int *mb, int *nb, int *irsrc, int *icsrc,
43              int *ictxt, int *lld, int *info);
44      extern void pdgemm_( const char *TRANSA, const char *TRANSB,
45              int * M, int * N, int * K, double * ALPHA, double * A,
46              int * IA, int * JA, int * DESCA, double * B, int * IB,
47              int * JB, int * DESCB, double * BETA, double * C,
48              int * IC, int * JC, int * DESCC );
49      extern void pdgeadd_(const char *TRANSA, int * M, int * N,
50              double * ALPHA, double * A, int * IA, int * JA,
51              int * DESCA, double * BETA, double * C, int * IC,
52              int * JC, int * DESCC );
53      extern void pdgemv_ (const char *TRANS, int *M, int *N,
54              double *ALPHA, double *A, int *IA, int *JA,
55              int *DESCA, double *X, int *IX, int *JX,
56              int *DESCX, int *INCX, double *BETA,
57              double *Y, int *IY, int *JY, int *DESCY, int *INCY);
58      extern void pdgesv_( int *n, int *nrhs, double *A, int *ia,
59              int *ja, int *desca, int* ipiv,double *B, int *ib,
60              int *jb, int *descb, int *info);
61      extern void pdpotrf_(const char *UPLO, int *N,
62              double *A, int *IA, int *JA, int *DESCA, int *INFO);
63      extern void pdpotri_(const char *UPLO, int *N,
64              double *A, int *IA, int *JA, int *DESCA, int *INFO);
65      extern void Cblacs_barrier  (int, const char * );
66      extern void pdpotrs_(const char *uplo, int *n, int *nrhs,
67              double *a, int *ia, int *ja, int *desc_a,
68              double *b, int *ib, int *jb, int *desc_b, int *info);
69  };
70
71
72  void welcome()
73  {
74    printf("\n");
75    printf("# trainbr-EXTENDED                        #\n");
76    printf("# C/Fortran/Scalapack version             #\n");
77    printf("# Bayesian regularized neural networks    #\n");
78    printf("#_________________________________________#\n");
79    printf("\n");
80  }
81
82  int max( int a, int b )
83  {
84      if (a>b) return(a); else return(b);
85  }
86
87  double cor_Pearson(double *x, double *y, int n)
88  {
89     double sumx=0;
```

```
90     double sumy=0;
91     double sumxy=0;
92     double sumx2=0;
93     double sumy2=0;
94     int i;
95
96     for(i=0; i<n;i++)
97     {
98        sumx+=x[i];
99        sumy+=y[i];
100       sumxy+=x[i]*y[i];
101       sumx2+=x[i]*x[i];
102       sumy2+=y[i]*y[i];
103    }
104    return((sumxy-sumx*sumy/(double (n)))
105            /(pow((sumx2-sumx*sumx/(double (n)))
106           *(sumy2-sumy*sumy/(double (n))),0.5)));
107 }
108
109 void read_matrix(double *X, int rows, int columns, char *Input_file)
110 {
111     const int MAX_LEN = 32767;
112     char Buffer[MAX_LEN];
113     char *token=NULL;
114     FILE *ptr;
115     int i,j;
116
117     ptr=fopen(Input_file,"r");
118     if(ptr!=NULL)
119     {
120             i=0;
121         printf("Loading incidence matrix...");
122                while(fgets(Buffer, MAX_LEN, ptr) != NULL)
123             {
124                 token=strtok(Buffer,",");
125           j=0;
126                 while(token!=NULL)
127           {
128             X[i+(j*rows)]=atof(token);
129             token = strtok(NULL,",");
130             j++;
131                }
132                i++;
133             }
134             printf("Done\n");
135         if(i!=rows)
136         {
137           printf("The file has MORE/LESS lines that
138                   those indicated by the user\n");
139           exit(1);
140         }
141         fclose(ptr);
142     }else{
143         printf("Unable to open input file with incidence matrix\n");
```

```
144         exit(1);
145     }
146 }
147
148 void read_vector(double *y, int rows,char *Input_file)
149 {
150       const int MAX_LEN=256;
151       char Buffer[MAX_LEN];
152       int i;
153       FILE *ptr;
154       ptr=fopen(Input_file,"r");
155       if(ptr!=NULL)
156       {
157         i=0;
158         printf("Loading response vector...");
159         while(fgets(Buffer, MAX_LEN, ptr) != NULL)
160         {
161             y[i]=atof(Buffer);
162             i++;
163             }
164             printf("Done\n");
165         if(i!=rows)
166         {
167             printf("The file has MORE/LESS lines that
168                     those indicated by the user\n");
169             exit(1);
170         }
171         fclose(ptr);
172       }else{
173           printf("Unable to open file with response vector\n");
174           exit(1);
175       }
176 }
177
178 double tansig(double x)
179 {
180     return(2.0/(1.0+exp(-2.0*x)) - 1.0);
181 }
182
183 double sech(double x)
184 {
185     return(2.0*exp(x)/(exp(2.0*x)+1.0));
186 }
187
188 double predictions_nn(double *X, unsigned int rows,
189                       unsigned int columns, double *theta,
190                       unsigned int neurons,double *yhat, double *y,
191                       double *residuals)
192 {
193    unsigned int i,j,k;
194    double suma,z;
195    double error=0;
196
197    for(i=0;i<rows;i++)
```

```
198     {
199        suma=0;
200        for(k=0;k<neurons;k++)
201        {
202           z=0;
203           for(j=0;j<columns;j++)
204           {
205              z+=X[i+(j*rows)]*theta[(columns+2)*k+j+2];
206           }
207           z+=theta[(columns+2)*k+1];
208           suma+=theta[(columns+2)*k]*tansig(z);
209        }
210        yhat[i]=suma;
211        residuals[i]=y[i]-yhat[i];
212        error+=pow(residuals[i],2.0);
213     }
214     return(error);
215  }
216
217  void predictions_nn(double *X, unsigned int rows,
218                      unsigned int columns, double *theta,
219                      unsigned int neurons,double *yhat)
220  {
221     unsigned int i,j,k;
222     double suma,z;
223
224     for(i=0;i<rows;i++)
225     {
226        suma=0;
227        for(k=0;k<neurons;k++)
228        {
229           z=0;
230           for(j=0;j<columns;j++)
231           {
232              z+=X[i+(j*rows)]*theta[(columns+2)*k+j+2];
233           }
234           z+=theta[(columns+2)*k+1];
235           suma+=theta[(columns+2)*k]*tansig(z);
236        }
237        yhat[i]=suma;
238     }
239  }
240
241  void jacobian(double *X, unsigned int n,
242                unsigned int p,double *theta,
243                unsigned int neurons,double *J)
244  {
245     unsigned int i,j,k;
246     double z,dtansig;
247
248     printf("Calulating jacobian...");
249     for(i=0; i<n; i++)
250     {
251       for(k=0; k<neurons; k++)
```

```
252      {
253           z=0;
254           for(j=0;j<p;j++)
255           {
256               z+=X[i+(j*n)]*theta[(p+2)*k+j+2];
257           }
258           z+=theta[(p+2)*k+1];
259           dtansig=pow(sech(z),2.0);
260           J[i+(((p+2)*k)*n)]=-tansig(z);
261           J[i+(((p+2)*k+1)*n)]=-theta[(p+2)*k]*dtansig;
262
263          for(j=0; j<p;j++)
264          {
265              J[i+(((p+2)*k+j+2)*n)]=-theta[(p+2)*k]*dtansig*X[i+(j*n)];
266          }
267      }
268    }
269    printf("Done\n");
270 }
271
272 double sc(double *theta, int npar)
273 {
274    double suma=0;
275    for(int i=0; i<npar; i++) suma+=pow(theta[i],2.0);
276    return(suma);
277 }
278
279 void initnw(double *theta, int n, int p, int neurons, int npar)
280 {
281   double scaling_factor,dx, suma;
282   int k,j;
283
284   srand(time(NULL));
285   for(int i=0; i<npar;i++)
286   theta[i]=((double) rand()) / ((double) RAND_MAX)-0.5;
287
288   printf("Initializing using the Nguyen-Widrow method\n");
289   if(p==1)
290   {
291       scaling_factor=0.7*neurons;
292       for(k=0;k<neurons;k++)
293       {
294           theta[(p+2)*k+1]=scaling_factor*(1.0-2.0*(((double) rand())
295                / ((double) RAND_MAX)));
296           theta[(p+2)*k+2]=scaling_factor;
297       }
298   }else{
299      scaling_factor=0.7*pow(neurons,(1.0/double (n)));
300      dx=2.0/double(neurons);
301      for(k=0; k<neurons;k++)
302      {
303           suma=0;
304           for(j=0;j<p;j++)
305           {
```

```
306             suma+=pow(theta[(p+2)*k+2+j],2.0);
307           }
308           for(j=0; j<p; j++)
309           {
310             theta[(p+2)*k+2+j]=scaling_factor*theta[(p+2)*k+2+j]/pow(suma,0.5);
311           }
312           theta[(p+2)*k+1]=(-1.0+dx*k)*scaling_factor;
313       }
314     }
315   }
316
317   void matrix_crossprod(double *A, int row_a, int col_a, double *C,
318                         int row_c, int col_c, int nprow,
319                         int npcol, int mb, int nb, int myrow,
320                         int mycol, int ictxt )
321   {
322       int one=1;
323       int zero=0;
324       int info;
325       double alpha=1.0;
326       double beta=0.0;
327       int descA[9], descA_distr[9];
328       int descC[9], descC_distr[9];
329       int mp1 = numroc_( &row_a, &mb, &myrow, &zero, &nprow );
330       int nq1 = numroc_( &col_a, &nb, &mycol, &zero, &npcol );
331       double *A_distr = (double *) malloc(mp1*nq1*sizeof(double));
332       int lld1 = max(numroc_( &row_a, &row_a, &myrow, &zero, &nprow), 1 );
333
334       descinit_( descA, &row_a, &col_a, &row_a, &col_a,
335                  &zero, &zero, &ictxt, &lld1, &info );
336       int lld_distr1 = max( mp1, 1 );
337
338       descinit_( descA_distr, &row_a, &col_a, &mb, &nb, &zero,
339                  &zero, &ictxt, &lld_distr1, &info );
340       pdgeadd_( "N", &row_a, &col_a, &alpha, A, &one, &one,
341                 descA, &beta, A_distr, &one, &one, descA_distr);
342
343       int mp3 = numroc_(&row_c, &mb, &myrow, &zero, &nprow);
344       int nq3 = numroc_(&col_c, &nb, &mycol, &zero, &npcol);
345       double *C_distr = (double *) malloc(mp3*nq3*sizeof(double));
346       int lld3 = max( numroc_( &row_c, &row_c, &myrow, &zero,
347                       &nprow ), 1);
348       descinit_(descC, &row_c, &col_c, &row_c, &col_c,
349                  &zero, &zero, &ictxt, &lld3, &info);
350
351       int lld_distr3 = max(mp3, 1);
352       descinit_(descC_distr, &row_c, &col_c, &mb, &nb,
353                  &zero, &zero, &ictxt, &lld_distr3, &info);
354
355       pdgeadd_("N", &row_c, &col_c, &alpha, C, &one, &one,
356                 descC, &beta, C_distr, &one, &one, descC_distr);
357       pdgemm_("T", "N", &row_c, &col_c, &row_a, &alpha,
358               A_distr, &one, &one, descA_distr, A_distr,
359               &one, &one, descA_distr, &beta, C_distr,
```

```
360             &one, &one, descC_distr);
361     pdgeadd_("N",&row_c, &col_c, &alpha, C_distr, &one,
362              &one, descC_distr, &beta, C, &one, &one, descC);
363
364
365     free(A_distr);
366     A_distr=NULL;
367     free(C_distr);
368     C_distr=NULL;
369 }
370
371 void matrix_vector_product(double *A, int row_a,
372                            int col_a, double *b,  double *sol,
373                            int nprow, int npcol, int mb, int nb,
374                            int myrow, int mycol, int ictxt)
375 {
376     int one=1;
377     int zero=0;
378     int info;
379     double alpha=1.0;
380     double beta=0.0;
381
382     int descA[9], descA_distr[9];
383     int descb[9], descb_distr[9];
384     int descsol[9], descsol_distr[9];
385     int mp1 = numroc_( &row_a, &mb, &myrow, &zero, &nprow );
386     int nq1 = numroc_( &col_a, &nb, &mycol, &zero, &npcol );
387     double *A_distr = (double *) malloc(mp1*nq1*sizeof(double));
388     int lld1 = max(numroc_( &row_a, &row_a, &myrow, &zero, &nprow), 1 );
389
390     descinit_( descA, &row_a, &col_a, &row_a, &col_a, &zero,
391               &zero, &ictxt, &lld1, &info );
392     int lld_distr1 = max( mp1, 1 );
393     descinit_( descA_distr, &row_a, &col_a, &mb, &nb,
394               &zero, &zero, &ictxt, &lld_distr1, &info );
395     pdgeadd_( "N", &row_a, &col_a, &alpha, A, &one, &one,
396              descA, &beta, A_distr, &one, &one, descA_distr);
397
398     int ma=max(row_a,col_a);
399
400     int mp2 = numroc_( &ma, &mb, &myrow, &zero, &nprow );
401     int nq2 = numroc_( &one, &nb, &mycol, &zero, &npcol );
402     double *b_distr = (double *) malloc(mp2*nq2*sizeof(double));
403
404     int lld2 = max(numroc_( &ma, &ma, &myrow, &zero, &nprow), 1 );
405     descinit_( descb, &ma, &one, &ma, &one, &zero, &zero, &ictxt,
406                &lld2, &info );
407     int lld_distr2 = max( mp2, 1 );
408     descinit_( descb_distr, &ma, &one, &mb, &nb, &zero, &zero,
409                &ictxt, &lld_distr2, &info );
410     pdgeadd_( "N", &ma, &one, &alpha, b, &one, &one, descb,
411                &beta, b_distr, &one, &one, descb_distr);
412
413     double *sol_distr=(double *) malloc(mp2*nq2*sizeof(double));
```

```
414
415
416     descinit_( descsol, &ma, &one, &ma, &one, &zero, &zero,
417                &ictxt, &lld2, &info);
418     descinit_( descsol_distr, &ma, &one, &mb, &nb, &zero,
419                &zero, &ictxt, &lld_distr2, &info);
420     pdgeadd_( "N", &ma, &one, &alpha, sol, &one, &one,
421               descsol, &beta, sol_distr, &one, &one, descsol_distr);
422     pdgemv_("T",&row_a,&col_a,&alpha,A_distr,&one,&one,
423              descA_distr,b_distr,&one,&one,descb_distr,
424              &one,&beta,sol_distr,&one,&one,descsol_distr,&one);
425     Cblacs_barrier(ictxt,"A");
426     pdgeadd_("N",&ma, &one, &alpha, sol_distr, &one, &one,
427              descsol_distr, &beta, sol, &one, &one, descsol);
428     Cblacs_barrier(ictxt,"A");
429
430     free(A_distr);
431     A_distr=NULL;
432     free(b_distr);
433     b_distr=NULL;
434     free(sol_distr);
435     sol_distr=NULL;
436 }
437
438 void solve_Cholesky(double *A, int row_a, int col_a,
439                     double *b, int nprow, int npcol,
440                     int mb, int nb, int myrow,
441                     int mycol, int ictxt)
442 {
443     int one=1;
444     int zero=0;
445     int info;
446     double alpha=1.0;
447     double beta=0.0;
448
449     int descA[9], descA_distr[9];
450     int descb[9], descb_distr[9];
451     int mp1 = numroc_( &row_a, &mb, &myrow, &zero, &nprow );
452     int nq1 = numroc_( &row_a, &nb, &mycol, &zero, &npcol );
453     double *A_distr = (double *) malloc(mp1*nq1*sizeof(double));
454     int lld1 = max(numroc_( &row_a, &row_a, &myrow, &zero, &nprow), 1 );
455
456     descinit_( descA, &row_a, &row_a, &row_a, &row_a, &zero,
457                &zero, &ictxt, &lld1, &info );
458     int lld_distr1 = max( mp1, 1 );
459     descinit_( descA_distr, &row_a, &row_a, &mb, &nb, &zero,
460                &zero, &ictxt, &lld_distr1, &info );
461     pdgeadd_( "N", &row_a, &row_a, &alpha, A, &one, &one, descA,
462               &beta, A_distr, &one, &one, descA_distr);
463
464     int mp2 = numroc_( &row_a, &mb, &myrow, &zero, &nprow );
465     int nq2 = numroc_( &one, &nb, &mycol, &zero, &npcol );
466     double *b_distr = (double *) malloc(mp2*nq2*sizeof(double));
467     int lld2 = max(numroc_( &row_a, &row_a, &myrow, &zero, &nprow), 1 );
```

```
468     descinit_( descb, &row_a, &one, &row_a, &one, &zero,
469                &zero, &ictxt, &lld2, &info );
470     int lld_distr2 = max( mp2, 1 );
471     descinit_( descb_distr, &row_a, &one, &mb, &nb, &zero, &zero,
472                &ictxt, &lld_distr2, &info );
473     pdgeadd_( "N", &row_a, &one, &alpha, b, &one, &one, descb, &beta,
474               b_distr, &one, &one, descb_distr);
475     pdpotrf_("U", &row_a,  A_distr, &one, &one, descA_distr, &info);
476
477     Cblacs_barrier(ictxt,"A");
478     pdpotrs_("U", &row_a, &one, A_distr, &one, &one, descA_distr,
479               b_distr, &one, &one, descb_distr, &info);
480     pdgeadd_("N",&row_a, &one, &alpha, b_distr, &one, &one,
481              descb_distr, &beta, b, &one, &one, descb);
482
483     Cblacs_barrier(ictxt,"A");
484
485     free(A_distr);
486     A_distr=NULL;
487     free(b_distr);
488     b_distr=NULL;
489 }
490
491 void inverse(double *A, int row_a,double *Ainv, int nprow,
492              int npcol, int mb, int nb, int myrow,
493              int mycol, int ictxt)
494 {
495     int one=1;
496     int zero=0;
497     int info;
498     double alpha=1.0;
499     double beta=0.0;
500     int descA[9], descA_distr[9];
501     int descAinv[9];
502     int mp1 = numroc_( &row_a, &mb, &myrow, &zero, &nprow );
503     int nq1 = numroc_( &row_a, &nb, &mycol, &zero, &npcol );
504     double *A_distr = (double *) malloc(mp1*nq1*sizeof(double));
505     int lld1 = max(numroc_( &row_a, &row_a, &myrow, &zero, &nprow), 1 );
506
507     descinit_( descA, &row_a, &row_a, &row_a, &row_a, &zero,
508                &zero, &ictxt, &lld1, &info );
509     int lld_distr1 = max( mp1, 1 );
510     descinit_( descA_distr, &row_a, &row_a, &mb, &nb, &zero,
511                &zero, &ictxt, &lld_distr1, &info );
512     pdgeadd_( "N", &row_a, &row_a, &alpha, A, &one, &one,
513               descA, &beta, A_distr, &one, &one, descA_distr);
514     pdpotrf_("U", &row_a,  A_distr, &one, &one, descA_distr, &info);
515     Cblacs_barrier(ictxt,"A");
516     pdpotri_("U", &row_a, A_distr, &one, &one, descA_distr, &info);
517     Cblacs_barrier(ictxt,"A");
518
519     int lld3 = max( numroc_( &row_a, &row_a, &myrow, &zero, &nprow ), 1);
520     descinit_(descAinv, &row_a, &row_a, &row_a, &row_a, &zero,
521               &zero, &ictxt, &lld3, &info);
```

```
522     pdgeadd_("N",&row_a, &row_a, &alpha, A_distr, &one, &one,
523               descA_distr, &beta, Ainv, &one, &one, descAinv);
524
525     Cblacs_barrier(ictxt,"A");
526     free(A_distr);
527     A_distr=NULL;
528 }
529
530 int main(int argc, char **argv)
531 {
532     int i,j;
533     double *X, *y,*yhat, *residuals,*theta, *theta_new,
534            *J, *H, *delta, *Hinv, *g, *diag_H, *F_history;
535
536     int rank_mpi;
537     int nprocs_mpi;
538
539     int n=atoi(argv[1]);
540     int p=atoi(argv[2]);
541     int neurons=atoi(argv[3]);
542     int nprow=atoi(argv[6]);
543     int npcol=atoi(argv[7]);
544     int mb=atoi(argv[8]);
545     int nb=atoi(argv[9]);
546
547     int npar=neurons*(2+p);
548
549     MPI_Init( &argc, &argv);
550     MPI_Comm_rank(MPI_COMM_WORLD, &rank_mpi);
551     MPI_Comm_size(MPI_COMM_WORLD, &nprocs_mpi);
552
553     MPI_Barrier(MPI_COMM_WORLD);
554
555     if (nprow*npcol>nprocs_mpi)
556     {
557       if (rank_mpi==0)
558           printf(" **** ERROR : we do not have enough processes
559                  available to make a p-by-q process grid ***\n");
560           printf(" **** Bye-bye ***\n");
561           MPI_Finalize();
562           exit(1);
563     }
564
565     if(nprow>npcol)
566     {
567         printf(" ****  Error number of rows  can not be larger
568                than the number of columns in the process grid ***\n");
569         printf(" ****  Bye-bye ***\n");
570         MPI_Finalize();
571         exit(1);
572     }
573
574      if(nprow>npcol)
575      {
```

```
576           printf(" ****  Error number of rows  can not be
577                   larger than the number of columns in
578                   the process grid ***\n");
579           printf(" ****  Bye-bye ***\n");
580           MPI_Finalize();
581           exit(1);
582       }
583
584       int ictxt;
585       int myrow, mycol;
586
587       Cblacs_get( -1, 0, &ictxt);
588       Cblacs_gridinit(&ictxt,"Row",nprow,npcol);
589       Cblacs_gridinfo(ictxt,&nprow,&npcol,&myrow,&mycol);
590
591
592       double gamma;
593       double Ed, Ed_new;
594       double Ew, Ew_new;
595       double alpha;
596       double beta;
597       double C;
598       double C_new;
599       int epoch=0;
600       int max_epochs=1000;
601       double mu=0.005;
602       double mu_inc=10.0;
603       double mu_dec=0.1;
604       double mu_max=1e10;
605       double change=0.001;
606       double trace;
607       double t1, t2, t3, t4;
608
609       if(rank_mpi==0)
610       {
611           t1=MPI_Wtime();
612           welcome();
613
614           printf("Predictors: %d\n",p);
615           printf("Neurons: %d\n",neurons);
616           printf("Number of parameters to estimate: %d\n",npar);
617
618           X = (double *) malloc(n*p*sizeof(double));
619           y = (double *) malloc(n*sizeof(double));
620           yhat=(double *) malloc(n*sizeof(double));
621           residuals=(double *) malloc(n*sizeof(double));
622           J=(double *) malloc(n*npar*sizeof(double));
623           H=(double *) malloc(npar*npar*sizeof(double));
624           Hinv=(double *) malloc(npar*npar*sizeof(double));
625           theta=(double *) malloc(npar*sizeof(double));
626           theta_new=(double *) malloc(npar*sizeof(double));
627           delta=(double *) malloc(npar*sizeof(double));
628           g=(double *) malloc(npar*sizeof(double));
629           diag_H=(double *) malloc(npar*sizeof(double));
```

```
630         F_history=(double *) malloc(max_epochs*sizeof(double));
631
632         read_matrix(X,n,p,argv[4]);
633         read_vector(y,n,argv[5]);
634
635         initnw(theta,n,p,neurons,npar);
636
637         gamma=npar;
638         Ed=predictions_nn(X,n,p,theta,neurons,yhat,y,residuals);
639         beta=(n-gamma)/(2.0*Ed);
640         if(beta<0) beta=1.0;
641         Ew=sc(theta,npar);
642         alpha=gamma/(2.0*Ew);
643     }
644
645     MPI_Barrier(MPI_COMM_WORLD);
646
647     int flag_C;
648     int flag_mu=1;
649     int flag_change_F=1;
650
651     while(epoch<max_epochs && flag_mu && flag_change_F)
652     {
653       if(rank_mpi==0)
654       {
655             printf("epoch=%d\n",epoch);
656          printf("alpha=%4.4f\tbeta=%4.4f\n",alpha,beta);
657
658          jacobian(X,n,p,theta,neurons,J);
659
660          Ed=predictions_nn(X, n, p, theta,neurons,yhat,y,residuals);
661          Ew=sc(theta,npar);
662          printf("Ed=%4.4f\tEw=%4.4f\n",Ed,Ew);
663       }
664
665       matrix_crossprod(J,n,npar,H, npar, npar, nprow,
666                        npcol,mb,nb, myrow, mycol,ictxt);
667       matrix_vector_product(J,n,npar,residuals,g,nprow,
668                        npcol, mb, nb, myrow, mycol, ictxt);
669
670       if(rank_mpi==0)
671       {
672         for(i=0; i<npar;i++)
673         {
674           g[i]=g[i]+alpha/beta*theta[i];
675           diag_H[i]=H[i+(i*npar)];
676         }
677         C=beta*Ed+alpha*Ew;
678       }
679
680       flag_C=1;
681       while(flag_C && flag_mu)
682       {
683         if(rank_mpi==0)
```

```
684         {
685           for(i=0; i<npar;i++)
686           {
687             H[i+(i*npar)]=diag_H[i]+alpha/beta+mu/(2.0*beta);
688             delta[i]=g[i];
689           }
690         }
691 
692         solve_Cholesky(H,npar,npar,delta,
693                        nprow,nprow,mb,nb,myrow,mycol,ictxt);
694 
695         if(rank_mpi==0)
696         {
697            for(i=0; i<npar;i++)
698            {
699               theta_new[i]=theta[i]-delta[i];
700            }
701            Ed_new=predictions_nn(X, n,
702                                  p, theta_new,neurons,yhat,y,residuals);
703            Ew_new=sc(theta_new,npar);
704            C_new=beta*Ed_new+alpha*Ew_new;
705            printf("C_new=%f\tmu=%f\n",C_new,mu);
706            if(C_new<C)
707            {
708              mu=mu*mu_dec;
709              if (mu < 1e-20) mu = 1e-20;
710              flag_C=0;
711              F_history[epoch]=C_new;
712              if(epoch>3)
713              {
714                if((fabs(F_history[epoch]-F_history[epoch-1])<change)
715                    && (fabs(F_history[epoch-1]-F_history[epoch-2])<change)
716                    && (fabs(F_history[epoch-2]-F_history[epoch-3])<change))
717                {
718                   flag_change_F=0;
719                   printf("Changes in F = beta*SCE + alpha*Ew in
720                           last 3 iterations less than 0.001\n");
721                }
722              }
723            }else{
724              mu=mu*mu_inc;
725              if(mu>mu_max)
726              {
727                flag_mu=0;
728                printf("Maximum mu reached\n");
729              }
730            }
731         }
732 
733         MPI_Bcast(&flag_mu,1,MPI_INT,0,MPI_COMM_WORLD);
734         MPI_Bcast(&flag_C,1,MPI_INT,0,MPI_COMM_WORLD);
735         MPI_Bcast(&flag_change_F,1,MPI_INT,0,MPI_COMM_WORLD);
736       }
737 
```

```
738         if(rank_mpi==0)
739         {
740            for(i=0; i<npar;i++)
741            {
742              for(j=0; j<npar;j++)
743              {
744                 if(i==j){
745                   H[i+(i*npar)]=2.0*beta*diag_H[i]+2.0*alpha;
746                 }else{
747                   H[i+(j*npar)]=2.0*beta*H[i+(j*npar)];
748                 }
749              }
750            }
751            t3=MPI_Wtime();
752         }
753
754         inverse(H,npar,Hinv,nprow,nprow,mb,nb,myrow,mycol,ictxt);
755
756         if(rank_mpi==0)
757         {
758           t4=MPI_Wtime();
759           trace=0;
760           for(i=0; i<npar; i++)
761           {
762             theta[i]=theta_new[i];
763             trace+=Hinv[i+(i*npar)];
764           }
765           gamma=npar-2.0*alpha*trace;
766           alpha=gamma/(2.0*Ew_new);
767           beta=(n-gamma)/(2.0*Ed_new);
768           printf("gamma=%4.4f\talpha=%4.4f\tbeta=%4.4f\n",gamma,alpha,beta);
769           printf("Elapsed time inverse=%4.4f\n",t4-t3);
770           printf("#---------------------------------------- #\n");
771           epoch++;
772         }
773
774         MPI_Bcast(&epoch,1,MPI_INT,0,MPI_COMM_WORLD);
775
776       }
777
778       MPI_Barrier(MPI_COMM_WORLD);
779
780       Cblacs_gridexit(0);
781
782       if(rank_mpi==0)
783       {
784         printf("PREDICTIONS FOR THE TRAINING SET\n");
785         printf("Case\t y \t yhat \n");
786         for(i=0; i<n; i++)
787           printf("%d\t %f\t%f\n",i+1,y[i],yhat[i]);
788         printf("Pearson's Correlation=%4.4f\n",cor_Pearson(y,yhat,n));
789         printf("#---------------------------------------------- #\n");
790
791         if(argc>10)
```

```
792         {
793           int ntest=atoi(argv[11]);
794           double *Xtest = (double *) malloc(ntest*p*sizeof(double));
795           double *yhattest= (double *) malloc(ntest*sizeof(double));
796           read_matrix(Xtest,ntest,p,argv[10]);
797           predictions_nn(Xtest,ntest,p,theta,neurons,yhattest);
798 
799           printf("PREDICTIONS FOR THE TESTING SET\n");
800 
801           if(argc==12)
802           {
803                   printf("Case\t yhat \n");
804                   for(i=0; i<ntest; i++)
805                             printf("%d\t%f\n",i+1,yhattest[i]);
806                   printf("#------------------------------------ #\n");
807           }
808           if(argc==13)
809           {
810             double *ytest=(double *) malloc(ntest*sizeof(double));
811             read_vector(ytest,ntest,argv[12]);
812             printf("Case\t y\t yhat \n");
813              for(i=0; i<ntest; i++)
814                printf("%d\t%f\t%f\n",i+1,ytest[i],yhattest[i]);
815                printf("#------------------------------------------------------ #\n");
816                printf("Pearson's Correlation=%4.4f\n",cor_Pearson(ytest,yhattest,ntest));
817                free(ytest);
818           }
819 
820           free(Xtest);
821           free(yhattest);
822         }
823 
824         free(X);
825         free(y);
826         free(yhat);
827         free(residuals);
828         free(J);
829         free(H);
830         free(Hinv);
831         free(theta);
832         free(theta_new);
833         free(delta);
834         free(g);
835         free(diag_H);
836         free(F_history);
837 
838         t2=MPI_Wtime();
839         printf("Elapsed time=%4.2f seconds\n",t2-t1);
840       }
841 
842       MPI_Finalize();
843 
844       exit(0);
845   }
```

Anexo B: Script shell para compilación

```
#!/bin/bash

#if mpi executables and libraries are in the path it is not necessary
echo "Be sure that mpi executables and library are in the path";

#Compile the code
mpic++ -c -Wall trainbr.cpp
echo "compiling code...done";

#link the code, in SUSE it is necesary to link against lgfortran
mpic++ -o trainbr trainbr.o /opt/lib/libscalapack.a -lblas -llapack
echo "linking... done";
```

Anexo C: Rutinas de R para Normalizar y Desnormalizar

C.1. Normalizar datos

```
rm(list=ls())
setwd("~/Investigacion/Datos")


normalizar=function(Xatrain,base,spread){
  if(is.matrix(Xatrain)){
    return(sweep(sweep(Xatrain,2,base,"-"),2,2/spread,"*")-1)
  }else{
    return(2*(Xatrain-base)/spread-1)
  }
}

# Normalizar el archivo Xtrain.csv, completo

Xtrain = read.csv("Xtrain.csv", header=FALSE)
Xtrain=as.matrix(Xtrain)

base=apply(Xtrain,2,"min")
maximo=apply(Xtrain,2,"max")
spread=maximo-base
DatosNormalizados_Xtrain = normalize(Xtrain,base,spread)
write.table(DatosNormalizados_Xtrain,file="XtrainN.csv")

# Normalizar el archivo Ytrain.csv

YtrainE = read.csv("Ytrain.csv", header=FALSE)
```

```
YtrainE=as.matrix(YtrainE)

base=min(YtrainE)
maximo=max(YtrainE)
spread=maximo-base

DatosNormalizados_YtrainE = normalize(YtrainE,base,spread)
write.table(DatosNormalizados_YtrainE,file="Ytrain.csv")

# Normalizar el archivo Xtest.csv, Completo

Xtrain = read.csv("Xtrain.csv", header=FALSE)
Xtrain=as.matrix(Xtrain)

base=apply(Xtrain,2,"min")
maximo=apply(Xtrain,2,"max")
spread=maximo-base

Xtest = read.csv("Xtest.csv", header=FALSE)
Xtest=as.matrix(Xtest)

DatosNorm_Xtest = normalize(Xtest,base,spread)
write.table(DatosNorm_Xtest,file="XtestN.csv")
```

C.2. Desnormalizar datos

```
un_normalize=function (Xatrain,base,spread) {
  if (is.matrix(Xatrain)) {
    return(sweep(sweep(Xatrain + 1, 2, 0.5 * spread, "*"), 2, base, "+"))
  }
  else {
    return(base + 0.5 * spread * (Xatrain + 1))
  }
}

Ytrain = read.csv("Ytrain.csv", header=FALSE)
Ytrain=as.matrix(Ytrain)
base=min(Ytrain)
maximo=max(Ytrain)
spread=maximo-base

XRes = read.csv("Resultados.csv", header=FALSE)
XRes=as.matrix(XRes)

DatosDesNormalizados_Res_Y = un_normalize(XRes[,1],base,spread)      #observados
DatosDesNormalizados_Res_Yhat = un_normalize(XRes[,2],base,spread)   #predichos


# des-normalizar el archivo Y predicho, y observados. Completo

Ytrain = read.csv("Ytrain.csv", header=FALSE)
```

```
26 Ytrain=as.matrix(Ytrain)
27
28 base=min(YtrainE)
29 maximo=max(YtrainE)
30 spread=maximo-base
31
32 YtestHat = read.csv("Yhat_test.csv", header=FALSE)
33 YtestHat=as.matrix(YtestHat)
34 Ytestin_hat = un_normalize(YtestHat,base,spread)
35
36
37 Ytest = read.csv("Yhat_ans.csv", header=FALSE)
38 Ytest=as.matrix(Ytest)
```