Proyecto final de carrera para la obtención del Título de Ingeniero en Informática

Estudio de la tecnología CUDA en la implementación de algoritmos basados en redes neuronales y modelos estadísticos

Escuela Politécnica Superior y Facultad de Ciencias Experimentales

Universidad de Almería

Alumna: *Aida López Túnez*

Director: *Dr. José Antonio Torres Arriaza*

Almería, Febrero de 2013

A mis padres.

Agradecimientos

A mi madre, por ser mi incondicional y estar en varios sitios a la vez para permitirme estudiar y aprender tantas cosas como quise.

A mi hermana, por ser mi competidora más feroz y retarme a superarme continuamente.

A José Manuel, por ser mi compañero, mi apoyo y mi empuje en los buenos y en los malos momentos.

A Carmen, por ser mi amiga y la mejor compañera de prácticas que podría tener. Sin ti nada habría sido lo mismo.

A mis profesores, a todos aquellos que creyeron en mí y me animaron a seguir adelante, y a los que no creyeron y con ello me animaron a demostrarles que se equivocaban.

A José Antonio Torres, mi profesor, director de proyecto, jefe y sobre todo amigo, por confiar tanto en mí.

Y sobre todo, quiero agradecerle a mi padre, por su trabajo, su constancia, su profesionalidad, su paciencia y su pasión por la ciencia, la cultura y el conocimiento, que durante toda mi vida ha sido una gran inspiración para mí.

A todos ellos, gracias.

Prólogo

La creación de los computadores permitió al mundo científico un importantísimo avance tanto en fiabilidad como en la rapidez operativa de datos. Cada año las mejoras en los componentes de los ordenadores mejoran estas cualidades.

Cálculos que una persona tardaría horas en realizar son resueltos por un computador en segundos. Sin embargo, determinados problemas pueden tardar en resolverse días, semanas, meses,… pese a ser realizados en un ordenador (predicción meteorológica, reconocimiento de patrones, etc.). Para solucionar esta problemática se creó el concepto de paralelización.

Las redes neuronales artificiales son algoritmos que simulan aspectos básicos del funcionamiento del cerebro humano. De esta forma, los problemas son resueltos de una forma parecida a la que lo haría una persona, pero de forma más rápida y eficaz, pero como ya hemos dicho anteriormente, determinado tipo de problemas pueden tardar mucho tiempo en resolverse, y para las redes neuronales artificiales también se da este caso.

La paralelización consiste en la subdivisión de un problema, en subproblemas, cada uno de los cuales es resuelto de forma simultánea y por separado, ofreciendo reducciones importantes en cuanto al tiempo de ejecución. Sin embargo, no todos los problemas son aptos para el empleo de la paralelización, y los que sí lo son, no tienen porqué responder de forma positiva a su resolución paralela (presentar mejoras en el tiempo). Lo mismo ocurre con el empleo de la paralelización en redes neuronales artificiales, no siempre es posible y/o positivo, todo depende del caso de estudio.

Índice general

Capítulo 1: Introducción..16
1. Motivación para el desarrollo 16
2. Objetivos 16
2.1. Objetivos formativos 17
2.2. Objetivos generales del proyecto 17
3. Metodología 17
4. GPU: Un procesador de varios núcleos, paralelismo alto y multihilo. 18
Capítulo 2: Tecnología CUDA de NVIDIA............................21
1. CUDA: Un modelo de programación paralela 21
2. JCuda 22
2.1. Características 23
2.2. Librerías 24
2.2.1. Librería base 24
2.2.2. Otras librerías 24
2.3. Utilidades. 25
2.4. Ejemplos. 26
Capítulo 3: Redes neuronales artificiales. 27
1. ¿Qué son las RNA? 27
2. Estructura de una RNA 29
3. Conexiones 29
4. Aplicaciones 30
5. Entrenamiento de la red 31
6. Aprendizaje 31
6.1. Aprendizaje supervisado. 32
6.2. Aprendizaje no supervisado. 33
Capítulo 4: Redes neuronales artificiales RBF.......................35
1. Redes neuronales artificiales de funciones de base radial (RBF) 35
1.1. Topología de las RBF 35
1.2. Método de aprendizaje de las RBF 36

1.3. Resumen esquemático **36**
1.3.1. Inicialización de la red 36
1.3.2. Entrenamiento de la red 37
1.3.3. Operación de la red 38
2. RBF Lineal 39
2.1. Codificación de la red **40**
2.1.1. Inicialización de la red 40
2.1.2. Entrenamiento de la red 42
2.1.3. Operación de la red 47
2.2. Estadísticas de simulación **48**
3. RBF CUDA 50
3.1. Codificación de la red **50**
3.1.1. Inicialización de la red 50
3.1.2. Entrenamiento de la red 50
3.1.3. Operación de la red 60
3.2. Estadísticas de la simulación **61**
4. Comparativa RBF 63
4.1. Comparativa de las gráficas **63**
4.2. Resumen **66**
Capítulo 5: Redes neuronales artificiales SOM 67
1. Mapas autoorganizativos (SOM) 67
1.1. Aprendizaje competitivo **68**
1.2. Topología de las redes SOM **68**
1.3. Método de aprendizaje de las redes SOM **69**
1.4. Resumen esquemático **70**
1.4.1. Inicialización de la red 70
1.4.2. Entrenamiento de la red 70
1.4.3. Operación de la red 72
2. SOM Lineal 72
2.1. Codificación de la red **73**
2.1.1. Inicialización de la red 73
2.1.2. Entrenamiento de la red 75
2.1.3. Operación de la red 78

2.2. Estadísticas de la simulación 79
3. SOM CUDA 81
3.1. Codificación de la red 82
3.1.1. Inicialización de la red 82
3.1.2. Entrenamiento de la red 82
3.1.3. Operación de la red 86
3.2. Estadísticas de la simulación 86
4. Comparativa SOM 88
4.1. Comparativa de las gráficas 88
4.2. Resumen 93
Capítulo 6: Conclusiones finales. 94
Bibliografía 97

Índice de figuras

Figura 1. Comparativa esquemática entre CPU y GPU. 19

Figura 2. Paralelización por bloques. 20

Figura 3. Componentes software de CUDA. 22

Figura 4. Red multicapa con propagación hacia delante. 28

Figura 5. Red neuronal artificial con realimentación y competición. 30

Figura 6. Gráfica completa de las simulaciones de RBF lineal. 49

Figura 7. Gráfica completa de las simulaciones de RBF CUDA. 61

Figura 8. Curvatura inicial de las simulaciones de RBF CUDA. 62

Figura 9. Datos finales de las simulaciones de RBF CUDA. 62

Figura 10. Gráficas completas de las simulaciones de RBF lineal y RBF CUDA. 64

Figura 11. Gráfica completa conjunta/comparativa de las simulaciones de RBF lineal y RBF CUDA…………………………………………………………65

Figura 12. Topología de una red SOM. 69

Figura 13. Gráfica completa de las simulaciones de SOM lineal. 80

Figura 14. Gráfica sin valor extremo de las simulaciones de SOM lineal. 80

Figura 15. Gráfica de la curvatura de las simulaciones de SOM lineal. 81

Figura 16. Gráfica completa de las simulaciones de SOM CUDA. 86

Figura 17. Gráfica sin valor extremo de las simulaciones de SOM CUDA. 87

Figura 18. Gráfica de la curvatura de las simulaciones de SOM CUDA. 87

Figura 19. Gráfica completa conjunta/comparativa de las simulaciones de SOM lineal y SOM CUDA………………………………………………………...88

Figura 20. Gráfica conjunta/comparativa sin valor extremo de las simulaciones de SOM lineal y SOM CUDA………………………………………………89

Figura 21. Gráfica de la curvatura conde la curvatura conjunta/comparativa de las simulaciones de SOM lineal y SOM CUDA………………………………89

Figura 22. Gráfica de la tasa de crecimiento de las simulaciones de SOM lineal. 90

Figura 23. Gráfica de la tasa de crecimiento de las simulaciones de SOM CUDA. 91

Figura 24. Gráfica parcial de la tasa de crecimiento de las simulaciones de SOM lineal. 91

Figura 25. Gráfica parcial de la tasa de crecimiento de las simulaciones de SOM CUDA. 92

Índice de tablas

Tabla 1. Resumen de la comparativa de las simulaciones de RBF lineal y RBF CUDA. 66

Tabla 2. Resumen de la comparativa de las simulaciones de SOM lineal y SOM CUDA. 93

Capítulo 1: Introducción.

1. Motivación para el desarrollo

Desde que comencé mi carrera como Ingeniera Informática me di cuenta de algo que desconocía completamente al inicio: la Informática se divide en tal variedad de campos y cada uno de ellos posee tal variedad de especializaciones, que resulta muy complicado conocer con gran profundidad cada uno de los mismos.

Conforme fueron transcurriendo los años desarrollé una gran pasión por la programación, que se convirtió en mi pilar como informática, llegando a trabajar posteriormente como programadora e incluso impartiendo cursos y clases sobre la misma. La programación se convirtió en mi red de seguridad, donde me sentía cómoda y experimentada, pero… ¿realmente, en tan poco años, sabía tanto de ella? La respuesta era no.

He de reconocer que a lo largo de mi carrera universitaria he llegado a aprender más de aquellas asignaturas que me resultaron duras, que me hicieron cuestionarme a mí misma y sacar fuerzas desde donde no sabía que las tenía. Esas asignaturas supusieron un reto para mí. Por todo ello, con el apoyo de mi director, buscamos un proyecto en el que pudiera aprender aún más cosas y no sólo mostrar los conocimientos que ya tenía.

Había algo dentro del campo de la programación que para mí suponía un reto: la paralelización. Cierto era que ya había trabajado anteriormente con este concepto, pero nunca se me había dado la oportunidad de acceder al uso de GPUs. Finalmente escogimos la tecnología CUDA de NVIDIA, que actualmente es relativamente novedosa y está a la vanguardia del mercado, para llevar a cabo este proyecto, así como el desarrollo de redes neuronales artificiales para explotar las capacidades de dicha tecnología.

2. Objetivos

El objetivo principal de este proyecto es usar la tecnología CUDA de NVIDIA para desarrollar algoritmos de paralelización para simular redes neuronales artificiales. Vamos a diferenciar los objetivos en dos grupos principales: objetivos formativos y objetivos generales del proyecto.

2.1. Objetivos formativos

Desde el punto de vista formativo, la alumna se beneficiará de la experiencia y enriquecerá sus conocimientos en el campo de la programación, y con ello conseguirá:

- Mejorar el aprendizaje sobre los procesos de paralelización, concretamente los llevados a cabo mediante el empleo de GPUs.
- Conocer el funcionamiento de la tecnología CUDA de NVIDIA.
- Conocer el funcionamiento de distintos tipos de redes neuronales.

2.2. Objetivos generales del proyecto

Dentro de los objetivos generales para este proyecto podemos encontrar los siguientes puntos:

- Emplear los conocimientos de programación anteriormente adquiridos.
- Emplear la tecnología CUDA de NVIDIA.
- Desarrollar algoritmos de simulación de redes neuronales artificiales.
- Desarrollar algoritmos de paralelización para simular redes neuronales artificiales.
- Comparar resultados estadísticos de las simulaciones lineales y paralelas de distintos tipos de redes neuronales.

La finalidad de este proyecto es la comparación de explotación de las GPUs de CUDA de NVIDIA, por lo que los datos utilizados para los cálculos en las redes neuronales artificiales simuladas RBF y SOM no persiguen hallar un resultado para una situación real, sino solamente simular dichas redes y comparar los tiempos de cómputo.

3. Metodología

Los algoritmos de simulaciones de los distintos tipos de redes neuronales se llevarán a cabo en el lenguaje de programación Java [6][7], con la ayuda de una serie de librerías de CUDA, que serán explicadas en siguientes puntos.

Para este proyecto se ha utilizado un ordenador que posee las siguientes características:

- Procesador Intel Core i5 modelo 660.
- 4 GB de RAM.
- Windows 7 Professional.

Para realizar el paralelismo se ha instalado una tarjeta gráfica CUDA de NVIDIA modelo GT 420. Esta tarjeta soporta la tecnología CUDA.

La implementación de los programas de simulación de las redes neuronales artificiales, que como ya hemos indicado se crearán en Java, se dará en el entorno integrado de desarrollo Netbeans [8].

4. GPU: Un procesador de varios núcleos, paralelismo alto y multihilo.

Impulsada por la insaciable demanda del mercado de tiempo real, los gráficos de alta definición en 3D, la GPU programable se ha convertido en un procesador de varios núcleos, con paralelismo alto y multihilo, con una tremenda potencia de cálculo y un gran ancho de banda de memoria.

Sin embargo, existe una discrepancia con respecto a la capacidad de punto flotante entre la CPU y la GPU debido a que la GPU está especializada para la computación intensiva, la computación de paralelismo alto, y por lo tanto diseñada de tal manera que más transistores se dedican al procesamiento de datos en lugar de al almacenamiento en caché de datos y al control de flujo, por lo que en ocasiones la cantidad de accesos a memoria puede jugar en su contra.

Más específicamente, la GPU es especialmente adecuada para abordar los problemas que se pueden expresar como cálculos paralelos de datos (el mismo programa de ejecuta en muchos elementos de datos en paralelo) con alta intensidad aritmética. Como el mismo programa se ejecuta para cada elemento de datos, hay un menor requerimiento de control de flujo sofisticado, y como se ejecuta en muchos elementos de datos y tienen alta intensidad aritmética, la latencia de acceso a memoria puede ser ocultada con cálculos en lugar de cachés de datos grandes.

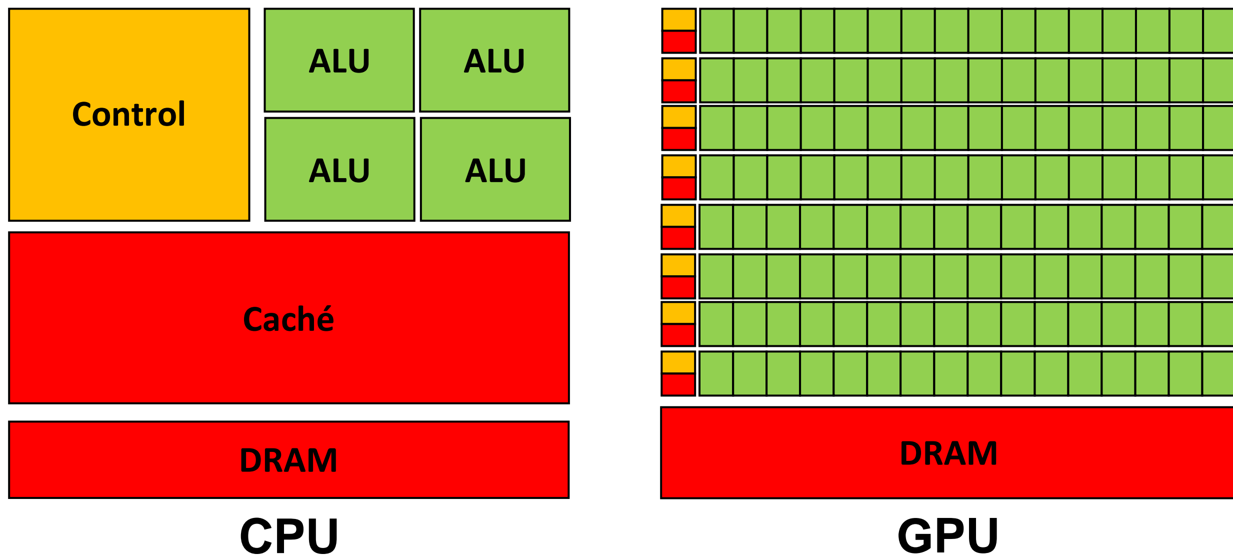

Figura 1. Comparativa esquemática entre CPU y GPU.

Muchas aplicaciones que procesan grandes conjuntos de datos pueden utilizar un modelo de programación de datos paralelo para acelerar los cálculos. En la representación 3D de grandes conjuntos de píxeles y vértices están asignados a hilos paralelos. De hecho, muchos algoritmos fuera del campo de procesamiento de imágenes son acelerados por el procesamiento paralelo de datos, procesamiento de señal de forma general o simulación de la física a las finanzas computacionales o la biología computacional.

Nvidia ha desarrollado, bajo el paraguas de la tecnología CUDA, los procesadores Tesla de alto rendimiento en operaciones de cálculo paralelo masivo. Sin embargo, las tarjetas gráficas también se benefician de esta tecnología permitiendo al programador utilizar GPUs de esta manera similar.

Para la simulación de este proyecto vamos a utilizar un procesador de múltiples núcleos, y para explotar al máximo su capacidad de procesamiento nos centraremos en la paralelización dentro de cada uno de los conjuntos de entrenamiento, mientras que cada conjunto serán procesados secuencialmente.

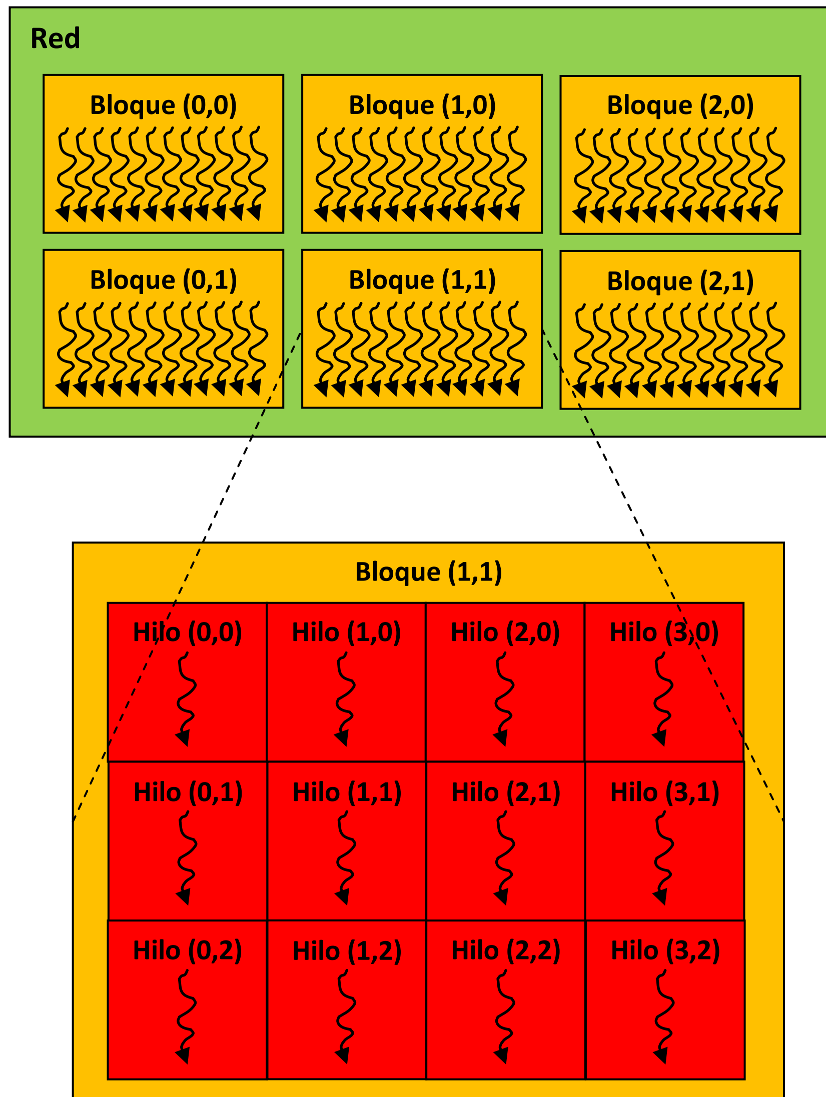

Figura 2. Paralelización por bloques.

A partir de los resultados finales podremos realizar estadísticas y análisis de los mismos, para obtener las conclusiones finales de nuestro proyecto.

Capítulo 2: Tecnología CUDA de NVIDIA.

1. CUDA: Un modelo de programación paralela

La llegada de CPUs multinúcleo y GPUs de varios núcleos significa que los chips de procesadores convencionales son ahora sistemas paralelos. Además, su paralelismo continúa la ampliación de la ley de Moore. El reto es desarrollar aplicaciones software que de forma transparente escalen su paralelismo para aprovechar el creciente número de núcleos del procesador.

CUDA [1][2][3][4] intenta explotar las ventajas de las GPUs frente a las CPUs de propósito general utilizando el paralelismo que ofrecen sus múltiples núcleos, que permiten el lanzamiento de un altísimo número de hilos simultáneos.

CUDA (Compute Unified Device Architecture) es, desde el punto de vista de un programador, un compilador y un conjunto de herramientas de desarrollo creadas por NVIDIA que permiten usar una variación del lenguaje de programación C para codificar algoritmos en GPUs utilizando controladores NVIDIA.

En esencia son tres abstracciones clave: una jerarquía de grupos de hilos, memorias compartidas y sincronización de barrera; que son simplemente expuestos al programador como un conjunto mínimo de extensiones C.

Estas abstracciones proporcionan paralelismo de datos de grano fino y paralelismo de hilos, anidado dentro de paralelismo de datos de grano grueso y paralelismo de tareas. Estos guían al programador a dividir el problema en sub-problemas secundarios que se pueden resolver de forma independiente en paralelo y, a continuación en trozos más finos que se pueden resolver de forma cooperativa en paralelo. Tal descomposición preserva la expresividad del lenguaje permitiendo hilo para cooperar en la resolución de cada sub-problema, y al mismo tiempo permite escalabilidad transparente ya que cada sub-problema puede ser programado para ser resuelto en cualquiera de los núcleos disponibles del procesador, y sólo el Runtime del sistema necesita conocer el número de procesadores físicos.

El conjunto de software CUDA se compone de varias capas como se ilustra en la Figura 3: un controlador de dispositivo, una interfaz de programación de aplicaciones (API) y su runtime, y dos bibliotecas matemáticas de nivel superior de uso común, CUFFT y CUBLAS, que se describen más adelante.

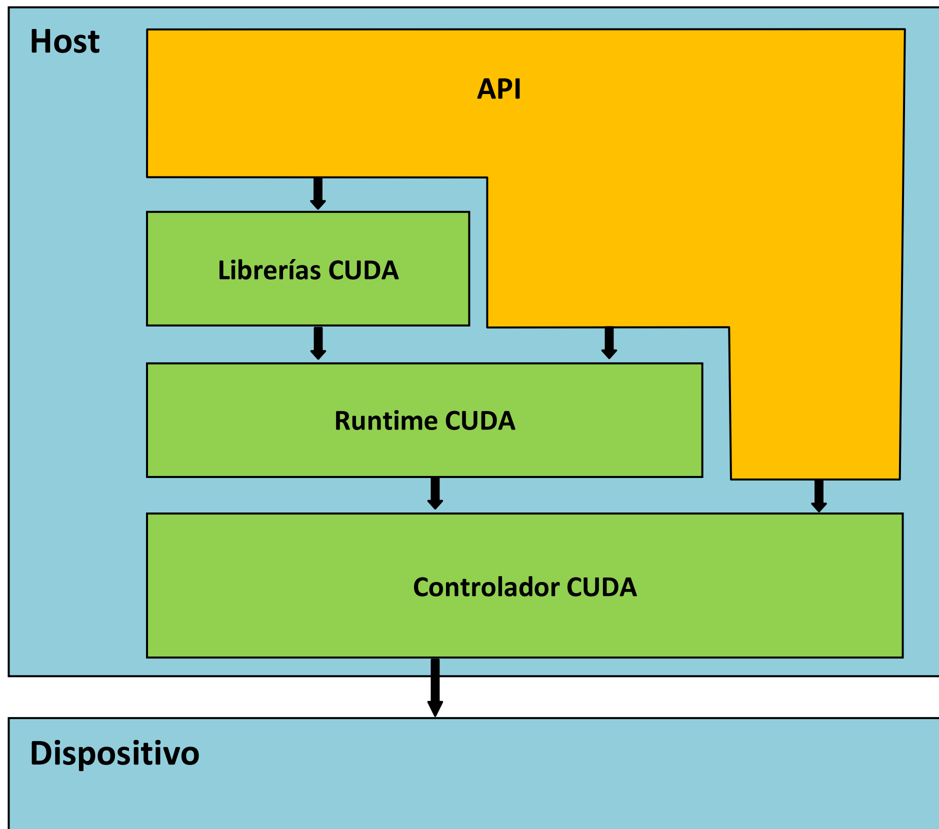

Figura 3. Componentes software de CUDA.

Dicho de otra forma, CUDA extiende C permitiendo al programador definir funciones C, llamados *kernels*, que, cuando son llamados, son ejecutados N veces en paralelo por N hilos CUDA diferentes, en lugar de sólo una vez como en las funciones regulares C.

Sin embargo, en los últimos años CUDA ha introducido una gran novedad para sus programadores, JCuda, que permite programar en Java en lugar de en C. Todo ello se explicará mejor en el siguiente punto.

2. JCuda

JCuda [5] consiste en un conjunto de librerías Java relacionadas, desarrolladas para el uso de GPUs de CUDA. Todas estas librerías se pueden descargar de la página oficial *jcuda.org*.

Las API de las librerías de JCuda se han mantenido cerca de la API original, que está desarrollada en el lenguaje de programación C. Las funciones de todas las librerías se proporcionan como métodos estáticos, y la semántica y las firmas de estos métodos se han

mantenido coherentes con la funciones de la librería original, salvo limitaciones específicas del lenguaje Java (como el uso de punteros).

Para utilizar dichas librerías, además de un dispositivo habilitado con CUDA GPU, serán necesarios el controlador de NVIDIA con soporte CUDA y el CUDA Toolkit que podemos adquirir en la página web de NVIDIA.

2.1. Características

Actualmente JCuda proporciona una serie de características, entre las que podemos encontrar las siguientes:

- Soporte para el controlador CUDA de la API.
- Posibilidad de cargar módulos propios en el controlador de la API.
- Soporte para la ejecución CUDA de la API.
- Interoperabilidad completa entre las diferentes librerías basadas en CUDA (JCublas, JCufft, JCudpp, JCurand, JCusparse y JNpp).
- Amplia documentación de la API extraída de las documentaciones de las librerías nativas.
- Interoperabilidad OpenGL.
- Control de errores conveniente.

Sin embargo, pese a todas las ventajas que proporciona JCuda, existen algunas limitaciones:

- No todas las funcionalidades se han probado extensamente en todos los sistemas operativos, dispositivos GPU y arquitecturas de acogida.
- Las operaciones de copia de memoria asíncrona que implique almacenamiento de memoria pueden causar la corrupción de la memoria.
- Pueden existir otro tipo de errores aún no detectados.

2.2. Librerías

2.2.1. Librería base

La principal librería también recibe el nombre de JCuda. Permite la interacción con un dispositivo CUDA, proporcionando métodos para la gestión de dispositivos y eventos, la asignación de memoria en el dispositivo y la copia de la memoria entre el dispositivo y el sistema anfitrión. Además, la librería cuenta con enlaces para el controlador de CUDA API, que permite cargar y ejecutar archivos PTX- y archivos CUBIN, y lanzar kernels CUDA desde Java.

2.2.2. Otras librerías

Existen otras librerías de propósito especial que utilizan JCuda como una plataforma común:

- **JCublas**

Enlaces Java para la librería CUDA BLAS de NVIDIA.

Esta librería permite utilizar CUBLAS, lo que implica la implementación NVIDIA CUDA de subprogramas de álgebra lineal básica para aplicaciones Java.

- **JCufft**

Enlaces Java para CUFFT, la librería CUDA FFT de NVIDIA.

Esta librería proporciona métodos para utilizar CUFFT, lo que implica la implementación de transformaciones rápidas de Fourier para aplicaciones Java.

- **JCudpp**

Enlaces Java para CUDPP, la librería CUDA de Datos Paralelos Primitivos.

Esta librería permite a las aplicaciones Java el uso de la librería CUDPP, que contiene métodos para realizar multiplicaciones de vectores-matrices dispersas, escaneos paralelos y clasificaciones.

- **JCurand**

Enlaces Java para la librería CURAND, el generador de números aleatorios CUDA de NVIDIA.

JCurand ofrece la generación de números aleatorios acelerados con GPU para Java.

- **JCusparse**

Enlaces Java para CUSPARSE, la librería CUDA de matriz dispersa de NVIDIA.

Con JCusparse es posible utilizar las funciones de matriz dispersa de niveles 1, 2 y 3, y rutinas de conversión de matrices dispersas.

- **JNpp**

Enlaces Java para NPP, librerías de NVIDIA para rendimiento primitivo (NVIDIDA Performace Primitive).

JNpp ofrece método de procesamiento de imágenes NPP.

2.3. Utilidades.

Además de las anteriores librerías, que son propias de CUDA, nos serán de gran ayuda una librería denominada JCudaUtils, que no forma parte del corazón de la API de JCuda, pero que puede llegar a ser de gran utilidad.

Una de las clases más usuales es *KernelLauncher*, que simplifica la configuración y puesta en marcha de los kernels mediante el controlador JCuda de la API. Permite la creación de archivos de código fuente PTX dentro del código fuente que es dado como un String o a partir de un archivo fuente de CUDA. Los archivos PTX- o CUBIN pueden ser cargados y los kernels pueden ser llamados más convenientemente debido a la configuración automática de los argumentos del kernel.

Por otro lado, la librería contiene algunas clases que ofrecen funcionalidades que son similares a las funciones “CUTIL” de la SDK CUDA, tales como:

- Analizar los argumentos de línea de comandos.
- Comparación de las matrices.

- Archivos simples de entrada/salida.
- Funciones de reloj.

Todas estas clases están destinadas principalmente para simplificar el proceso de portar los ejemplos de código existentes de NVIDIA CUDA a Java. También pueden ser útiles para la depuración o la creación de pruebas unitarias.

2.4. Ejemplos.

La página oficial JCuda.org también proporciona una serie de ejemplos del empleo de las librerías de JCuda y derivadas.

Algunos de los ejemplos aún utilizan archivos CUBIN, como es el caso del ejemplo de la inversión de matrices, el cual será empleado en el desarrollo de este proyecto fin de carrera, tal y como explicaremos más adelante.

Capítulo 3: Redes neuronales artificiales.

1. ¿Qué son las RNA?

Las redes neuronales artificiales (RNA) [9][10][11][12] son algoritmos implementados en forma de programa informático o modelo electrónico, basados en el funcionamiento del cerebro humano. Son un campo de investigación que estudia la creación de modelos complejos a partir de elementos simples, idénticos, no lineales y paralelos, con la intención de solucionar problemas como, por ejemplo, clasificaciones, regresiones y predicciones, entre otros.

La mecánica de los ordenadores es sencilla, siguen un esquema de procesamiento basado en la realización de una instrucción sobre un dato, sin embargo, tienen grandes dificultades en otros problemas relacionados con el reconocimiento de patrones, la generalización de futuros eventos a partir de acciones pasadas, etc., que requieren un procesamiento paralelo.

Las redes neuronales artificiales persiguen una similitud al cerebro humano. Éste almacena la información en forma de patrones, los cuales tienen distintos niveles de complejidad. El proceso de almacenar información en patrones, utilizarlos y resolver problemas con ellos es el que tratan de imitar las redes neuronales artificiales, para intentar reproducir el comportamiento humano sobre las computadoras.

Actualmente las redes neuronales artificiales intentan únicamente reproducir de forma simplificada los mecanismos más básicos del cerebro animal. No se trata de reproducir el cerebro humano, sólo se centran en mecanismos de resolución de problemas individuales. La finalidad de las RNA actuales es, por tanto, la de entender la forma en la que los humanos resuelven los problemas y utilizarla para complementar las capacidades de los sistemas de inteligencia artificial.

El cerebro humano tiene una estructura tridimensional y sus neuronas son capaces de interconectarse libremente; las redes neuronales artificiales proporcionan un esquema de procesamiento alternativo basado en la operación de un número determinado de unidades de procesamiento elemental denominadas neuronas, que están interconectadas entre sí.

Entre las distintas arquitecturas, existe un tipo, comúnmente utilizado, que tiene una estructura en capas, que generalmente son tres:

- **Capa de entrada:** Las entradas de las RNA aceptan datos de tipo numérico (tanto con un rango de variación continuo como discreto). Cuando los datos manejados en un problema no son numéricos, se necesita obtener una representación numérica de los mismos (de acuerdo con un sistema de codificación), antes de poder ser procesados automáticamente por la misma.
- **Capa intermedia:** Puede haber una o varias capas intermedias. En la mayoría de las redes neuronales artificiales, cada neurona de la capa intermedia (o capas intermedias) recibe una señal de cada una de las neuronas de la capa anterior a través de un conjunto de interconexiones ponderadas. Todas estas señales de entrada se combinan en una sola, sobre la que se realizar una operación elemental y cuyo resultado se transmite a todas las neuronas de la capa siguiente. De esta forma, la neurona actúa como la unidad de procesamiento elemental sobre la que se fundamenta la operación de la RNA, en el sentido en que la combinación de dichas operaciones elementales es la que permite a la red desempeñar una tarea relevante.
- **Capa de salida:** Las neuronas de la capa de salida generan los valores que la red cree que se corresponden con los valores de entrada a ella, o bien los valores que la red cree que son la predicción adecuada de los valores de entrada.

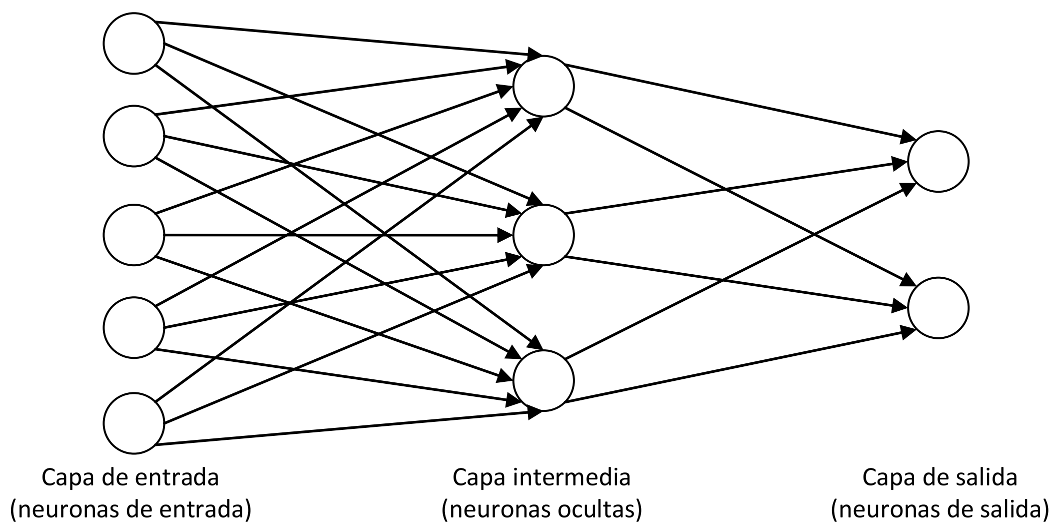

Figura 4. Red multicapa con propagación hacia delante.

Las RNA no utilizan conceptos de programación propios de otros sistemas de inteligencia artificial, sino que utilizan mecanismos de procesamiento paralelo, entrenamiento de pesos, etc.

2. Estructura de una RNA

Una red neuronal artificial puede verse como un conjunto de neuronas interconectadas, de tal forma que la salida de una neurona cualquiera sirve, generalmente, como entrada de otras neuronas. Existen varios aspectos importantes a la hora de estudiar una red neuronal artificial.

Por un lado, el número de neuronas, la disposición de las mismas y las conexiones existentes entre ellas determinan su componente estructural, también denominada arquitectura o topología de la red.

Por otro lado, ya que las neuronas están interconectadas, tiene sentido hablar de los caminos o rutas a través de los cuales se propagan las señales con el fin de procesar una información. Este procesamiento se realizará mediante el cambio de estado de las neuronas en dicho camino. El estado de todas y cada una de las neuronas se denomina estado de la red, y su variación en el tiempo se realiza de acuerdo con un determinado modo de operación.

Por último, los pesos asociados a cada una de las conexiones representan la configuración de la red. Esta configuración se modifica en el tiempo gracias a un proceso de entrenamiento que determina la capacidad de adaptación o aprendizaje del modelo.

3. Conexiones

En la figura anterior podemos apreciar que las conexiones entre las neuronas siempre son hacia delante (propagación hacia delante), sin embargo, las redes neuronales artificiales también se pueden diseñar de modo que las neuronas de una misma capa estén interconectadas. Este tipo de RNA se utiliza, normalmente, en problemas de reconocimiento de patrones. En este caso, las neuronas de la capa de entrada compiten de tal forma que sólo "triunfe" una neurona (aquella que tenga más probabilidad de éxito), inhibiéndose el resto de neuronas.

Otro tipo de conexión es la realimentación (feedback). Se da en aquellos casos en que la salida de una neurona de un nivel realimenta las neuronas de un nivel anterior.

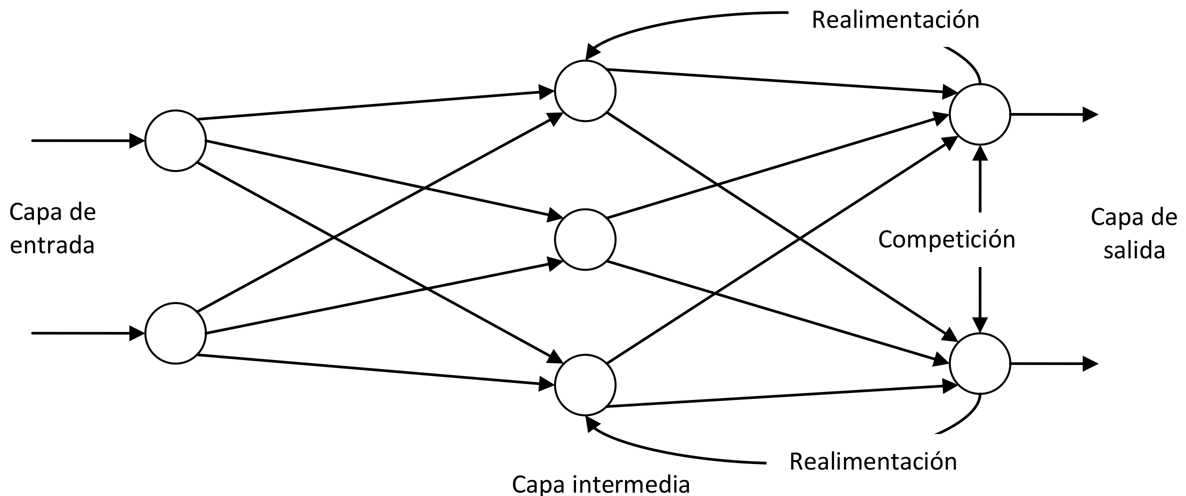

Figura 5. Red neuronal artificial con realimentación y competición.

Por otro lado, una de las características más especial de las RNA es su capacidad de adaptar su comportamiento mediante un proceso que genéricamente se denomina entrenamiento de la red.

4. Aplicaciones

Existen diversas aplicaciones potenciales de las redes neuronales artificiales, entre las que destacamos:

- **Asociación de patrones:** El problema consiste en la memorización de un conjunto de patrones de interés, de forma que, posteriormente, sea posible la recuperación de los mismos, bien a partir de una descripción parcial, bien a partir de una descripción completa pero deformada.
- **Reconocimiento de patrones:** En este caso se trata de clasificar un patrón de interés o la entrada en una clase de un conjunto de categorías predefinidas.
- **Aproximación de funciones:** Este problema se plantea en situaciones en las que existe una aplicación (no lineal) entre dos conjuntos de datos o espacios, y ésta se desconoce. En este problema se pretende que la red aproxime dicha aplicación con una precisión arbitraria.
- **Construcción de filtros:** Un filtro es un dispositivo o algoritmo utilizado para la extracción de una característica particular de un conjunto de datos ante la presencia de ruido. En este contexto, el ruido puede definirse como las variaciones aleatorias

de una o más características de los datos, por lo que el modelado y el tratamiento del ruido son fundamentales a la hora de resolver dichos problemas.

5. Entrenamiento de la red

Antes de ser utilizadas, las redes neuronales artificiales necesitan ser entrenadas. Este proceso comienza con la asignación de valores (que suelen ser aleatorios) a los pesos asociados a cada conexión y con la definición de los parámetros de aprendizaje utilizados. Tras la inicialización, se lleva a cabo el entrenamiento de la red, lo cual conlleva la adaptación de estos pesos.

La adaptación de los pesos o ajuste de los pesos debe realizarse de acuerdo con alguna regla de modificación o regla de aprendizaje. La más común y conocida es la regla de aprendizaje de Hebb (1949). La idea básica es que si dos neuronas j y k están activas simultáneamente, su interconexión debe ser fortalecida. Si la neurona k recibe como entrada la salida de la neurona j, en su versión más sencilla, la regla de Hebb establece que el peso sináptico w_{jk} debe modificarse de acuerdo con:

$$\Delta w_{jk} = n y_j y_k$$

donde n es una constante proporcional positiva que se denomina tasa de aprendizaje.

Existen otros tipos de reglas de aprendizaje, así como una amplia diversidad de algoritmos de aprendizaje. Haykin considera cinco tipos básicos de reglas de aprendizaje:

- Aprendizaje basado en la corrección de error.
- Aprendizaje basado en memoria.
- Aprendizaje Hebbiano.
- Aprendizaje competitivo.
- Aprendizaje Boltzmann.

6. Aprendizaje

Como hemos visto anteriormente, dentro del entrenamiento de una red se lleva a cabo el aprendizaje de la misma. La principal propiedad de una red neuronal artificial es la capacidad de aprender del entorno en el que opera y mejorar su funcionamiento.

El concepto de aprendizaje en el contexto de las redes neuronales artificiales puede definirse de la siguiente forma: el aprendizaje es el proceso mediante el cual los parámetros de una red neuronal artificial se adaptan como consecuencia de un proceso de estimulación llevado a cabo por el entorno en el que la red opera. El tipo de aprendizaje vendrá determinado por la forma en que cambie la configuración de la red.

Existe una clasificación del proceso de aprendizaje ampliamente extendida según el modelado del entorno en el que opera la red neuronal artificial:

- **Aprendizaje supervisado (Supervised Learning):** Existen pares de vectores de entrenamiento, en el que cada par está compuesto por los valores de entrada a la red y los valores de salida esperados para tales entradas. Es decir, se trata de un aprendizaje controlado por un supervisor o por un conjunto de reglas.
- **Aprendizaje no supervisado (Unsupervised Learning):** Sólo se utilizan vectores de entrada para llevar a cabo el entrenamiento la red y las salidas son determinados por la red durante el aprendizaje. Este tipo de aprendizaje persigue la búsqueda de similitudes entre los valores de los vectores de entrada.
- **Aprendizaje por grados o con fortalecimiento (Reinforcement Learning):** No se facilitan vectores de entrenamiento (entradas y salidas esperadas) para la red, por lo que para determinar la bondad de la red es necesario la actuación de un crítico que la valora de vez en cuando. Este es el único paradigma de aprendizaje que no desarrollaremos en este proyecto.

6.1. Aprendizaje supervisado.

El aprendizaje supervisado es un tipo de aprendizaje que requiere tanto las entradas como las salidas esperadas de la red durante la fase de entrenamiento. La red neuronal artificial procesa las entradas y compara los resultados obtenidos con los valores esperados, cuya diferencia es propagada hacia capas intermedias haciendo que el sistema reajuste los pesos.

Por tanto, para este tipo de aprendizaje se aportan tanto los datos de entrada como los datos de salida esperados, y estos datos se denominan conjunto de entrenamiento (training set). Durante el aprendizaje se procesan varias veces los mismos pares de vectores de entrenamiento y en cada paso del aprendizaje se van reajustando sus pesos.

El tiempo de aprendizaje puede durar segundos o semanas, terminando cuando la red alcanza los resultados esperados. Las redes pueden llegar a no aprender, como en el caso de

que los vectores de entrada utilizados para entrenarlas no contengan la información idónea a partir de la cual se deriven los vectores de salida.

En el caso de que una red no pueda resolver un problema, hay que revisar uno o varios de los siguientes puntos: los vectores de entrada y de salida, el número de capas, el número de neuronas en cada capa, las interconexiones entre capas, las funciones de activación utilizadas, etc.

Las redes con aprendizaje supervisado se utilizan en problemas de reconocimiento de códigos de barras, en problemas de predicción de series temporales, en problemas de control mediante robots, en el reconocimiento óptico de caracteres, etc.

Las redes neuronales artificales más comunes que utilizan el aprendizaje supervisado son las siguientes:

- MLP (Multilayer Perceptron).
- RBF (Radial-Basis Function Network).
- BAM (Bi-directional Associative Memory).

6.2. Aprendizaje no supervisado.

En el aprendizaje no supervisado la red sólo se entrena con vectores de entrada. Busca similitudes entre los valores de entrada, creando figuras que determinan el agrupamiento de los vectores de entrada, y así genera la salida. Por tanto, no hay valores deseados (no hay supervisión).

El propósito de un algoritmo de aprendizaje no supervisado es descubrir modelos o características significativas a partir únicamente de los datos de entrada. Para ello, los algoritmos de aprendizaje se basan en un conjunto de reglas de vecindad, lo que permite calcular la proyección entrada-salida preservando la estructura de los datos de entrada.

Las redes neuronales artificiales con aprendizaje no supervisado se utilizan en aplicaciones de reconocimiento de voz, análisis estadístico de datos, reconocimiento de caracteres, reducción de la dimensión (como el número de colores de un fichero imagen), etc.

Entre las redes neuronales artificiales con aprendizaje no supervisado podemos encontrar:

- SOM (*Self-Organizing Maps*, desarrollados por Kohonen).

- PCA (*Principal Components Analysis*).
- Redes de Hopfield.
- *Learning Vector Quantization*.
- ART (*Adaptive Resonance Theory*).
- FIR (*Finite Impulse Response*).

Capítulo 4: Redes neuronales artificiales RBF.

1. Redes neuronales artificiales de funciones de base radial (RBF)

Las redes neuronales artificiales de funciones de base radial (RBF) se engloban dentro de las redes de aprendizaje supervisado. Su topología es similar a la utilizada por las redes MLP (perceptrón multicapa), sin embargo, su forma de aprendizaje es completamente diferente.

El perceptrón multicapa es capaz de aproximar una función continua o modelar un problema. Sin embargo, este tipo de redes requiere unos tiempos de entrenamiento que en ocasiones son excesivos. Con RBF se reducen los tiempos de entrenamiento de las redes neuronales con aprendizaje supervisado.

Las RBF son excelentes aproximadores universales. Utilizan varios métodos como el método exacto o el método de los mínimos cuadrados medios para ajustar los pesos. Las bases son normalmente funciones Gaussianas cuyas medias y desviaciones estándar pueden calcularse teniendo en cuenta los vectores del espacio de entrada.

1.1. Topología de las RBF

Las redes neuronales artificiales RBF constan de tres capas: una capa de entrada, una capa intermedia y una capa de salida.

- **Capa de entrada:** La capa de entrada de la red de funciones de base radial es un receptor para los datos de entrada.
- **Capa intermedia:** La capa intermedia realiza una transformación no lineal del espacio de entrada al espacio del nivel intermedio. Las neuronas del nivel intermedio son las funciones base para los vectores de entrada.
- **Capa de salida:** La capa de salida calcula la combinación lineal ponderada de las salidas de las neuronas del nivel intermedio.

1.2. Método de aprendizaje de las RBF

Como ya hemos dicho al inicio, las RBF son excelentes aproximadores universales, y los parámetros que definen dicho proceso de aproximación son los siguientes:

- Los pesos entre los centros y las neuronas del nivel de salida.
- La posición de los centros.
- Las funciones Gaussianas de los centros.

Al igual que en el MLP, se presentan los pares de vectores de entrenamiento sucesivamente en la red. El valor de los pesos, los centros y las funciones gaussianas se ajusta en cada uno de estos pasos.

La adaptación de los pesos de la red durante el proceso de aprendizaje puede realizarse minimizando el error cuadrático medio, o mediante la utilización de otros algoritmos como el método de pseudoinversa.

1.3. Resumen esquemático

1.3.1. Inicialización de la red

<table>
<tr><td colspan="2">Conjunto E</td><td colspan="2">Conjunto C</td></tr>
<tr><td colspan="2">$E = \begin{Bmatrix} \overrightarrow{e_1} \\ \vdots \\ \overrightarrow{e_n} \end{Bmatrix}$</td><td colspan="2">$C = \begin{Bmatrix} \overrightarrow{c_1} \\ \vdots \\ \overrightarrow{c_n} \end{Bmatrix} = E$</td></tr>
<tr><td>Vector X</td><td>Vector RBF</td><td>Vector Y</td><td>Vector W</td></tr>
<tr><td>$\vec{x} = \begin{Bmatrix} x_1 \\ \vdots \\ x_L \end{Bmatrix}$</td><td>$\overrightarrow{RBF} = \begin{Bmatrix} RBF_1 \\ \vdots \\ RBF_n \end{Bmatrix}$</td><td>$\vec{y} = \begin{Bmatrix} y_1 \\ \vdots \\ y_n \end{Bmatrix}$</td><td>$\vec{w} = \begin{Bmatrix} w_1 \\ \vdots \\ w_n \end{Bmatrix}$</td></tr>
</table>

1º. Cargamos los valores del archivo:

- **N:** Cantidad de vectores del conjunto de datos de entrada E y del conjunto C. Longitud de los vectores $\overrightarrow{RBF}$, $\vec{y}$, $\vec{w}$.
- **L:** Longitud de los vectores del conjunto E, conjunto C y el vector X.
- **Conjunto E:** Conjuntos de vectores de entrada $\vec{e}$.

- **Vector X:** Vector de entrada X.
- **Vector Y:** Vector de salida esperada $\vec{y}$.

2º. Inicializamos:

- **Vector RBF:** Vector de neuronas RBF. Inicializamos todo sus elementos a valores aleatorios entre [-1,1].
- **Conjunto C:** Conjuntos de vectores $\vec{c}$ para el entrenamiento. Lo inicializamos igualándolo al conjunto E.
- **Vector W:** Vector de resultado $\vec{w}$.
- **Salida Y:** Resultado final de la red. Un número con decimales.

1.3.2. Entrenamiento de la red

En esta fase el propósito final es hallar el vector resultado $\vec{w}$. Para ello se realiza un transformación de los vectores implicados en matrices y la creación de la matriz de neuronas RBF de dimensiones NxN. En resumen, la ecuación para hallar el vector resultado $\vec{w}$ es la siguiente:

$$[\,\vec{w}\,] = [\,\vec{y}\,] \cdot [Matriz\ RBF]^{-1}$$

1º. Creamos dos matrices de dimensiones 1xN a partir de los vectores $\vec{w}$ e $\vec{y}$.

$$\vec{w} \Rightarrow [w_1 \quad \cdots \quad w_n]$$

$$\vec{y} \Rightarrow [y_1 \quad \cdots \quad y_n]$$

2º. Generamos la matriz de neuronas RBF. Como ya hemos indicado anteriormente, esta matriz tiene una dimensión de NxN. Para la matriz las neuronas RBF, éstas se representarán mediante ϕ y la matriz se presentaría del siguiente modo:

$$\begin{bmatrix} \phi_1(\overrightarrow{e_1}) & \phi_1(\overrightarrow{e_2}) & \cdots & \phi_1(\overrightarrow{e_n}) \\ \phi_2(\overrightarrow{e_1}) & \phi_2(\overrightarrow{e_2}) & & \phi_2(\overrightarrow{e_n}) \\ \vdots & & \ddots & \vdots \\ \phi_n(\overrightarrow{e_1}) & \phi_n(\overrightarrow{e_2}) & \cdots & \phi_n(\overrightarrow{e_n}) \end{bmatrix}$$

Cada una de estas neuronas se calcula durante la fase de entrenamiento a partir de los vectores del conjunto E y los vectores del conjunto C, mediante la siguiente fórmula:

$$\phi_i(\vec{e_j}) = e^{\frac{-\|\vec{e_j}-\vec{c_i}\|^2}{\sigma}}$$

El valor de sigma σ se establece por el usuario. En los entrenamientos realizados para este proyecto se le asignará el valor 0'05.

3º. Una vez tenemos generada la matriz RBF, calculamos su inversa. Este supone el punto más crítico en la explotación de redes neuronales artificiales RBF. En esta fase se consumirá el mayor tiempo para la ejecución de la red y una gran cantidad de memoria, llegando a colapsarla.

Por ello, este será el centro de nuestro estudio comparativo para las redes neuronales artificiales RBF, tal y como se expondrá en los dos puntos siguientes.

4º. Hallamos la matriz w mediante la operación indicada al principio del apartado de entrenamiento.

$$[w_1 \quad \cdots \quad w_n] = [y_1 \quad \cdots \quad y_n] \cdot \begin{bmatrix} \phi_1(\vec{e_1}) & \phi_1(\vec{e_2}) & \cdots & \phi_1(\vec{e_n}) \\ \phi_2(\vec{e_1}) & \phi_2(\vec{e_2}) & & \phi_2(\vec{e_n}) \\ \vdots & & \ddots & \vdots \\ \phi_n(\vec{e_1}) & \phi_n(\vec{e_2}) & \cdots & \phi_n(\vec{e_n}) \end{bmatrix}^{-1}$$

5º. Finalmente transformamos la matriz w en el vector $\vec{w}$.

$$[w_1 \quad \cdots \quad w_n] \Rightarrow \vec{w}$$

1.3.3. Operación de la red

En esta fase calcularemos el resultado final, la salida Y, que es un único número. La fórmula para realizar su cálculo precisa la transformación de vectores en matrices, tal y como se muestra a continuación:

$$salida\ Y = [\ \overrightarrow{RBF}\ ] \cdot [\ \vec{w}\ ]$$

Para realizar este cálculo, llevaremos a cabo los siguientes pasos:

1º. Calculamos los elementos del vector $\overrightarrow{RBF}$ a partir del vector de entrada $\vec{x}$ y de los vectores del conjunto C, siguiendo la siguiente fórmula:

$$RBF_i = \mathrm{e}^{\frac{-\|\vec{x}-\vec{c_i}\|^2}{\sigma}}$$

2º. Transformamos el vector $\overrightarrow{RBF}$ en una matriz de dimensiones 1xN, mientras que el vector $\vec{w}$ será transformado en una matriz de dimensiones Nx1.

$$\overrightarrow{RBF} \Rightarrow [RBF_1 \quad \cdots \quad RBF_n]$$

$$\vec{w} \Rightarrow \begin{bmatrix} w_1 \\ \vdots \\ w_n \end{bmatrix}$$

3º. Finalmente realizamos el producto de la matrices, tal y como indicamos al principio, para halla la salida Y.

$$salida\ Y = [RBF_1 \quad \cdots \quad RBF_n] \cdot \begin{bmatrix} w_1 \\ \vdots \\ w_n \end{bmatrix}$$

2. RBF Lineal

Para este proyecto, a la hora de simular una red neuronal artificial RBF, se ha desarrollado un programa en lenguaje de programación Java en el que se crea, se entrena y se opera una red neuronal artificial RBF.

En el punto anterior se ha explicado tanto desarrolladamente como a nivel esquemático el funcionamiento de las redes neuronales artificiales RBF, y seguiremos dichas pautas para explicar el programa que se ha desarrollado para llevar a cabo las simulaciones de este tipo de redes.

En este punto, y siguiendo el esquema de funcionamiento de las redes RBF expuesto en el punto anterior, vamos a explicar y resumir el código Java desarrollado para esta red. Posteriormente se mostrarán las estadísticas obtenidas de las diferentes simulaciones.

2.1. Codificación de la red

2.1.1. Inicialización de la red

1º. Declaramos las variables globales que vamos a utilizar:

```
int N = 0; //Cantidad de vectores del conjunto de datos de entrada E.
int L = 0; //Longitud de los vectores.

ArrayList conjuntoE; //Conjunto de vectores de entrada E.
double vectorX[]; //Vector de entrada X.
double vectorRBF[]; //Vector de neuronas RBF.
ArrayList conjuntoC; //Conjunto de vectores C de las RBFs.
double vectorW[]; //Vector de resultado W.
double vectorY[]; //Vector de salida esperada Y. Lo facilitamos por archivo.
double salidaY; //Salida Y final.

File archivo = null;
```

2º. Cargamos los valores del archivo:

- **N:** Cantidad de vectores del conjunto de datos de entrada E y del conjunto C. Longitud de los vectores $\overrightarrow{RBF}$, $\vec{y}$, $\vec{w}$.
- **L:** Longitud de los vectores del conjunto E, conjunto C y el vector X.
- **Conjunto E:** Conjuntos de vectores de entrada $\vec{e}$.
- **Vector X:** Vector de entrada X.
- **Vector Y:** Vector de salida esperada $\vec{y}$.

```
BufferedReader entrada = new BufferedReader(new FileReader(archivo));
String linea;
StringTokenizer espacio;

entrada.readLine();//Linea 1 - Comentario

N = Integer.parseInt(entrada.readLine()); //Linea 2.

entrada.readLine();entrada.readLine();//Linea 3-7 - Comentarios

L = Integer.parseInt(entrada.readLine()); //Linea 8.

entrada.readLine();//Linea 9 - Comentarios

//Conjunto de vectores de entrada E.
conjuntoE = new ArrayList();
double vectorE[];
for(int i=0;i<N;i++)
{
    linea = entrada.readLine();
    espacio=new StringTokenizer(linea," ");

    vectorE = new double[L];
    for(int j=0;j<L;j++)
    {
        //Leer elementos de archivo.
        double elemento = Double.parseDouble(espacio.nextToken());
        vectorE[j]=elemento;
    }
    conjuntoE.add(vectorE);
}

entrada.readLine();entrada.readLine();//Comentarios

//Vector de entrada X
linea = entrada.readLine();
espacio=new StringTokenizer(linea," ");
vectorX = new double[L];
for(int j=0;j<L;j++)
{
    //Leer elementos de archivo.
    double elemento = Double.parseDouble(espacio.nextToken());
    vectorX[j]=elemento;
}

entrada.readLine();entrada.readLine();//Comentarios

//Vector de salida esperada Y. Lo facilitamos por archivo
linea = entrada.readLine();
espacio=new StringTokenizer(linea," ");
vectorY = new double[N];
for(int j=0;j<N;j++)
{
    //Leer elementos de archivo.
    double elemento = Double.parseDouble(espacio.nextToken());
    vectorY[j]=elemento;
}
```

3º. Inicializamos:

- **Vector RBF:** Vector de neuronas RBF. Inicializamos todo sus elementos a valores aleatorios entre [-1,1].
- **Conjunto C:** Conjuntos de vectores $\vec{c}$ para el entrenamiento. Lo inicializamos igualándolo al conjunto E.
- **Vector W:** Vector de resultado $\vec{w}$.
- **Salida Y:** Resultado final de la red. Un número con decimales. En realidad no llega a inicializarse en sí, ya que se ha declarado como variable global y una vez se halle su valor, se mostrará.

```
        //Vector de neuronas RBF
        vectorRBF = new double[N];
        for(int i=0;i<N;i++)
        {
            double elemento = (double)(Math.random()*(2.0))+(-1.0);
            vectorRBF[i]=elemento;
        }

        //Conjunto de vectores de entrada C.
        conjuntoC = new ArrayList();
        double vectorC[];
        for(int i=0;i<N;i++)
        {
            vectorC = (double[])conjuntoE.get(i);
            conjuntoC.add(vectorC);
        }

        //Vector de resultado W
        vectorW = new double[N];
    }
```

2.1.2. Entrenamiento de la red

En esta fase el propósito final es hallar el vector resultado $\vec{w}$. Para ello se realiza un transformación de los vectores implicados en matrices y la creación de la matriz de neuronas RBF de dimensiones NxN. En resumen, la ecuación para hallar el vector resultado $\vec{w}$ es la siguiente:

$$[\,\vec{w}\,] = [\,\vec{y}\,] \cdot [Matriz\ RBF]^{-1}$$

1º. Creamos dos matrices de dimensiones 1xN a partir de los vectores $\vec{w}$ e $\vec{y}$.

$$\vec{w} \Rightarrow [w_1 \quad \cdots \quad w_n]$$

$$\vec{y} \Rightarrow [y_1 \quad \cdots \quad y_n]$$

En el cuadro de código anterior vemos como declaramos los vectores, que pueden ser utilizados por matrices, por lo que a nivel de código no es necesario realizar ninguna operación.

2º. Generamos la matriz de neuronas RBF. Como ya hemos indicado anteriormente, esta matriz tiene una dimensión de NxN. Para la matriz las neuronas RBF, éstas se representarán mediante ϕ y la matriz se presentaría del siguiente modo:

$$\begin{bmatrix} \phi_1(\vec{e_1}) & \phi_1(\vec{e_2}) & \cdots & \phi_1(\vec{e_n}) \\ \phi_2(\vec{e_1}) & \phi_2(\vec{e_2}) & & \phi_2(\vec{e_n}) \\ \vdots & & \ddots & \vdots \\ \phi_n(\vec{e_1}) & \phi_n(\vec{e_2}) & \cdots & \phi_n(\vec{e_n}) \end{bmatrix}$$

La función encargada de generar cada una de las posiciones (neuronas) de la matriz de neuronas RBF se denomina *calcularRBF* y como parámetros les pasamos la coordenada de la neurona.

```
double matrizRBFi[][] = new double [N][N];

//x será horizontal
//y será vertical
//Por tanto vamos rellenando la matriz por filas
for(int x=0;x<N;x++)
{
    for(int y=0;y<N;y++)
    {
        matrizRBFi[x][y] = calcularRBF(x,y);
    }
}
```

Cada una de estas neuronas se calcula con la función anteriormente mencionada *calcularRBF* durante la fase de entrenamiento, a partir de los vectores del conjunto E y los vectores del conjunto C, mediante la siguiente fórmula:

$$\phi_i(\vec{e_j}) = \mathrm{e}^{\frac{-\|\vec{e_j}-\vec{c_i}\|^2}{\sigma}}$$

El valor de sigma σ se establece por el usuario. En los entrenamientos realizados para este proyecto se le asignará el valor 0’05.

```
private double calcularRBF(int x, int y)
{
    double vectorResta[] = new double[L];
    double vectorC[] = (double[])conjuntoC.get(x);
    double vectorE[] = (double[])conjuntoE.get(y);
    for(int i=0;i<L;i++)
    {
        vectorResta[i] = vectorE[i] - vectorC[i];
    }

    double modulo = 0;
    for(int i=0;i<L;i++)
    {
        modulo = modulo + (vectorResta[i]*vectorResta[i]);
    }
    modulo = (double)Math.sqrt(modulo);

    double division = - ( modulo / 0.05); //El 0,05 es el SIGMA

    double euler = Math.exp(division);

    return euler;
}
```

3º. Una vez tenemos generada la matriz RBF, calculamos su inversa. Este supone el punto más crítico en la explotación de redes neuronales artificiales RBF. En esta fase se consumirá el mayor tiempo para la ejecución de la red y una gran cantidad de memoria, llegando a colapsarla.

Por ello, este será el centro de nuestro estudio comparativo para las redes neuronales artificiales RBF, tal y como se expondrá en los dos puntos siguientes.

A nivel de código, para calcular la inversa utilizamos el método de Faddeev-Leverrier.

```
private double[][] calcularInversaMatrizFaddeevLeverrier(double[][] matriz){
    int n = matriz.length;
    double c[] = new double[n];
    double matrizInversa[][] = new double[n][n];
    double s[][][] = (double[][][]) faddeevLeverrier(matriz, c);
    for (int i=0; i<n; ++i)
    {
        for (int j=0; j<n; ++j)
        {
            matrizInversa[i][j] = -s[n-1][i][j]/c[0];
        }
    }
    return matrizInversa;
}
// Método para completar la recursión Faddeev-Leverrier.
public static double[][][] faddeevLeverrier(double a[][], double c[]) {
    int n = c.length;
    double s[][][] = new double[n][n][n];
    for (int i=0; i<n; ++i)
    {
        s[0][i][i] = 1;
    }
    for (int k=1; k<n; ++k)
    {
        s[k] = mm(a,s[k-1]);
        c[n-k] = -tr(mm(a,s[k-1]))/k;
        for (int i=0; i<n; ++i)
        {
            s[k][i][i] += c[n-k];
        }
    }
    c[0] = -tr(mm(a,s[n-1]))/n;
    return s;
}
// Método para calcular la traza de una matriz.
public static double tr(double a[][]) {
    int n = a.length;
    double sum = 0;
    for (int i=0; i<n; ++i) sum += a[i][i];
    return sum;
}
// Método para calcular el producto de dos matrices.
public static double[][] mm(double a[][], double b[][]) {
    int n = a.length;
    double c[][] = new double[n][n];
    for (int i=0; i<n; ++i)
        for (int j=0; j<n; ++j)
            for (int k=0; k<n; ++k)
                c[i][j] += a[i][k]*b[k][j];
    return c;
}
```

4º. Hallamos la matriz w mediante la operación indicada al principio del apartado de entrenamiento.

$$[w_1 \quad \cdots \quad w_n] = [y_1 \quad \cdots \quad y_n] \cdot \begin{bmatrix} \phi_1(\vec{e_1}) & \phi_1(\vec{e_2}) & \cdots & \phi_1(\vec{e_n}) \\ \phi_2(\vec{e_1}) & \phi_2(\vec{e_2}) & & \phi_2(\vec{e_n}) \\ & \vdots & \ddots & \vdots \\ \phi_n(\vec{e_1}) & \phi_n(\vec{e_2}) & \cdots & \phi_n(\vec{e_n}) \end{bmatrix}^{-1}$$

```
        //Realizamos el producto de matrices
        double matrizProducto[][] = productoMatrices(matrizY, matrizRBFinversa);
```

```
    //Multiplica matrices de dimensiones compatibles
    private double[][] productoMatrices(double[][] matrizA, double[][] matrizB){
        int lonxA = matrizA.length;
        int lonyA = matrizA[0].length;
        int lonxB = matrizB.length;
        int lonyB = matrizB[0].length;

        if(lonyA != lonxB)
        {
            return null;
        }

        double[][] matrizResultado = new double[lonxA][lonyB];

        //i será la dimensión en común que recorrerá elementos de A y B.
        //x,y recorreran matrizResultado
        for(int x=0;x<lonxA;x++)
        {
            for(int y=0;y<lonyB;y++)
            {
                double sumatoria = 0;
                for(int i=0;i<lonyA;i++)
                {
                    sumatoria = sumatoria + (matrizA[x][i] * matrizB[i][y]);
                }
                matrizResultado[x][y] = sumatoria;
            }
        }
        return matrizResultado;
    }
```

5º. Finalmente transformamos la matriz w (*matrizProducto*) en el vector $\vec{w}$.

$$[w_1 \quad \cdots \quad w_n] \Rightarrow \vec{w}$$

```
//Transformamos esa matrizProducto que tendrá dimensión 1xN al vectorW
for(int i=0;i<matrizProducto[0].length;i++)
{
    vectorW[i] = matrizProducto[0][i];
}
```

2.1.3. Operación de la red

En esta fase calcularemos el resultado final, la salida Y, que es un único número. La fórmula para realizar su cálculo precisa la transformación de vectores en matrices, tal y como se muestra a continuación:

$$salida\ Y = [\ \overrightarrow{RBF}\ ] \cdot [\ \vec{w}\ ]$$

Para realizar este cálculo, llevaremos a cabo los siguientes pasos:

1º. Calculamos los elementos del vector $\overrightarrow{RBF}$ a partir del vector de entrada $\vec{x}$ y de los vectores del conjunto C, siguiendo la siguiente fórmula:

$$RBF_i = \mathrm{e}^{\frac{-\|\vec{x}-\vec{c_i}\|^2}{\sigma}}$$

```
for(int x=0;x<N;x++)
{
    double vectorResta[] = new double[L];
    double vectorC[] = (double[])conjuntoC.get(x);
    for(int i=0;i<L;i++)
    {
        vectorResta[i] = vectorX[i] - vectorC[i];
    }
    double modulo = 0;
    for(int i=0;i<L;i++)
    {
        modulo = modulo + (vectorResta[i]*vectorResta[i]);
    }
    modulo = (double)Math.sqrt(modulo);

    double division = - ( modulo / 0.05); //El 0,05 es el SIGMA

    vectorRBF[x] = Math.exp(division);
}
```

2º. Transformamos el vector $\overrightarrow{RBF}$ en una matriz de dimensiones 1xN, mientras que el vector $\vec{w}$ será transformado en una matriz de dimensiones Nx1.

$$\overrightarrow{RBF} \Rightarrow [RBF_1 \quad \cdots \quad RBF_n]$$

$$\vec{w} \Rightarrow \begin{bmatrix} w_1 \\ \vdots \\ w_n \end{bmatrix}$$

```
double matrizRBF[][] = new double[1][N];
double matrizW[][] = new double [N][1];

for(int i=0;i<N;i++)
{
    matrizRBF[0][i] = vectorRBF[i];
    matrizW[i][0] = vectorW[i];
}
```

3º. Finalmente realizamos el producto de la matrices, tal y como indicamos al principio, para halla la salida Y.

$$salida\ Y = [RBF_1 \quad \cdots \quad RBF_n] \cdot \begin{bmatrix} w_1 \\ \vdots \\ w_n \end{bmatrix}$$

```
double matrizY[][] = productoMatrices(matrizRBF, matrizW);

salidaY = matrizY[0][0];
```

2.2. Estadísticas de simulación

En este proyecto se han creado, entrenado y operado redes RBF, a lo largo de diversas simulaciones. A lo largo de dichas simulaciones se han ido aumentando la cantidad de vectores del conjunto de datos de entrada E (variable *N*), para explotar hasta sus máximas capacidades el ordenador y analizar la respuesta del mismo. Recordemos que esta variable *N* determina además la dimensión de la matriz RBF sobre la que se realizará la inversión y que constituye la mayor carga de trabajo a nivel de tiempo de ejecución en las simulaciones.

El número de vectores ha ido aumentando en múltiplos de 16[1], donde el ordenador experimenta un crecimiento más o menos constante, hasta llegar a la máxima capacidad del equipo utilizado con 304 vectores del conjunto de datos de entrada E (variable *N*). Para valores superiores, el ordenador se queda sin memoria de procesamiento.

A continuación se muestran las gráficas de algunas de las simulaciones llevadas a cabo:

Figura 6. Gráfica completa de las simulaciones de RBF lineal.

Como podemos ver, la gráfica de las simulaciones muestra una curvatura en la que podemos decir que el tiempo crece exponencialmente a la dimensión de la matriz RBF.

En el punto siguiente se explicará las mejoras introducidas para el desarrollo de redes neuronales artificiales RBF con CUDA, en donde podremos observar las diferencias con la ejecución lineal.

[1] Esto es así por una particularidad del algoritmo de paralelización que se usa después, y que requiere datos múltiplos de 16.

3. RBF CUDA

Se ha desarrollado un programa en lenguaje de programación Java, que al igual que en el punto anterior, se crea, se entrena y se opera una red neuronal artificial RBF, pero esta vez se recurre al empleo de la tecnología CUDA de NVIDIA.

La diferencia básicamente radica en que se va a incluir alguna mejora en la ejecución lineal de la red, mediante el empleo de funciones y librerías JCuda de CUDA de NVIDIA. En concreto, vamos a emplear la clase JCuda *MatrixInvert.java*, que realiza la inversión de una matriz mediante el empleo de la tecnología CUDA. Esta clase está accesible en la página web de JCuda.

Por tanto, el funcionamiento de la red neuronal artificial RBF será el mismo, puesto que el programa será igual en esencia, sólo que habrá una parte del código en la que introduciremos una mejora. Básicamente sustituiremos el método de inversión de matrices de Faddeev-Leverrier por la clase de JCuda *MatrixInvert.java.* Recordemos que la inversión de la matriz RBF se lleva a cabo en el proceso de entrenamiento de la red, y por tanto nos centraremos en el mismo.

Al igual que hemos hecho en el punto anterior, vamos a explicar y resumir el código Java nuevo desarrollado para esta red siguiendo el esquema de funcionamiento de las redes RBF que también hemos seguido para explicar el programa RBF lineal. Posteriormente se mostrarán las estadísticas obtenidas de las diferentes simulaciones.

3.1. Codificación de la red

3.1.1. Inicialización de la red

Igual que en el programa para la red neuronal artificial RBF de tipo lineal.

3.1.2. Entrenamiento de la red

En esta fase el propósito final es hallar el vector resultado $\vec{w}$. Para ello se realiza un transformación de los vectores implicados en matrices y la creación de la matriz de neuronas RBF de dimensiones NxN. En resumen, la ecuación para hallar el vector resultado $\vec{w}$ es la siguiente:

$$[\,\vec{w}\,] = [\,\vec{y}\,] \cdot [Matriz\ RBF]^{-1}$$

1º. Creamos dos matrices de dimensiones 1xN a partir de los vectores $\vec{w}$ e $\vec{y}$.

$$\vec{w} \Rightarrow [w_1 \quad \cdots \quad w_n]$$

$$\vec{y} \Rightarrow [y_1 \quad \cdots \quad y_n]$$

En el cuadro de código anterior vemos como declaramos los vectores, que pueden ser utilizados por matrices, por lo que a nivel de código no es necesario realizar ninguna operación.

2º. Generamos la matriz de neuronas RBF. Como ya hemos indicado anteriormente, esta matriz tiene una dimensión de NxN. Para la matriz las neuronas RBF, éstas se representarán mediante ϕ y la matriz se presentaría del siguiente modo:

$$\begin{bmatrix} \phi_1(\vec{e_1}) & \phi_1(\vec{e_2}) & \cdots & \phi_1(\vec{e_n}) \\ \phi_2(\vec{e_1}) & \phi_2(\vec{e_2}) & & \phi_2(\vec{e_n}) \\ & \vdots & \ddots & \vdots \\ \phi_n(\vec{e_1}) & \phi_n(\vec{e_2}) & \cdots & \phi_n(\vec{e_n}) \end{bmatrix}$$

La función encargada de generar cada una de las posiciones (neuronas) de la matriz de neuronas RBF se denomina *calcularRBF* y como parámetros les pasamos la coordenada de la neurona.

```
double matrizRBFi[][] = new double [N][N];

//x será horizontal
//y será vertical
//Por tanto vamos rellenando la matriz por filas
for(int x=0;x<N;x++)
{
    for(int y=0;y<N;y++)
    {
        matrizRBFi[x][y] = calcularRBF(x,y);
    }
}
```

Cada una de estas neuronas se calcula con la función anteriormente mencionada *calcularRBF* durante la fase de entrenamiento, a partir de los vectores del conjunto E y los vectores del conjunto C, mediante la siguiente fórmula:

$$\phi_i(\vec{e_j}) = \mathrm{e}^{\frac{-\|\vec{e_j}-\vec{c_i}\|^2}{\sigma}}$$

El valor de sigma σ se establece por el usuario. En los entrenamientos realizados para este proyecto se le asignará el valor 0'05.

```
private double calcularRBF(int x, int y)
{
    double vectorResta[] = new double[L];
    double vectorC[] = (double[])conjuntoC.get(x);
    double vectorE[] = (double[])conjuntoE.get(y);
    for(int i=0;i<L;i++)
    {
        vectorResta[i] = vectorE[i] - vectorC[i];
    }

    double modulo = 0;
    for(int i=0;i<L;i++)
    {
        modulo = modulo + (vectorResta[i]*vectorResta[i]);
    }
    modulo = (double)Math.sqrt(modulo);

    double division = - ( modulo / 0.05); //El 0,05 es el SIGMA

    double euler = Math.exp(division);

    return euler;
}
```

3º. Una vez tenemos generada la matriz RBF, calculamos su inversa. Este supone el punto más crítico en la explotación de redes neuronales artificiales RBF. En esta fase se consumirá el mayor tiempo para la ejecución de la red y una gran cantidad de memoria, llegando a colapsarla.

Por ello, este será el centro de nuestro estudio comparativo para las redes neuronales artificiales RBF, tal y como se expondrá en los dos puntos siguientes.

A nivel de código, en esta ocasión, en lugar de emplear el método de Faddeev-Leverrier, vamos a recurrir a la clase *MatrixInvert.java* de JCuda.

Esta clase tiene un particularidad, y es que es necesario especificarle el tamaño de bloque para la computación en paralelo. Este tamaño de bloque determinará las dimensiones de las matrices sobre las que puede operar, ya que dichas dimensiones deberán ser múltiplos del tamaño de bloque impuesto a la clase. Para este proyecto, se ha decido emplear un tamaño de bloque de 16.

Es necesario apuntar además que la clase *MatrixInvert.java* trabaja con matrices de tipo *Float*, por lo que será necesario realizar conversiones.

```
public double[][] calcularInversaMatriz(double[][] matriz)
{
    MatrixInvert.BLOCKSIZE=16;

    double m[][] = matriz;

    float origen [] = new float[N * N];
    float fin [] = new float[N * N];

    //Matriz origen
    int pos = 0;
    for(int i=0;i<m.length;i++)
    {
        for(int j=0;j<m[0].length;j++)
        {
            origen[pos] = Float.parseFloat(""+m[i][j]);
            pos++;
        }
    }

    //Inversión de la matriz
    MatrixInvert.invert(origen, fin, N);


    //Matriz inversa
    pos=0;
    for(int i=0;i<m.length;i++)
    {
        for(int j=0;j<m[0].length;j++)
        {
            m[i][j] = Double.parseDouble(""+fin[pos]);
            pos++;
        }
    }
    return m;
}
```

Y a continuación mostramos la clase *MatrixInvert.java* por partes (para verla con más claridad)*:*

- Declaración de variables globales:

```
public class MatrixInvert
{
    // The block size that is used in the kernels
    public static int BLOCKSIZE = 16;

    // The kernel launchers for the individual kernel functions
    private static KernelLauncher adjustRowL;
    private static KernelLauncher adjustRowU;
    private static KernelLauncher eliminateBlockL;
    private static KernelLauncher eliminateBlockU;
    private static KernelLauncher eliminateColL;
    private static KernelLauncher eliminateColU;
    private static KernelLauncher eliminateRestL;
    private static KernelLauncher eliminateRestU;
    private static KernelLauncher normalizeDiag;
    private static KernelLauncher setIdentity;
```

- Método de inicialización:

```
/**
 * Initializes all kernel launchers
 */
private static void init()
{
    if (adjustRowL != null)
    {
        return;
    }
    JCudaDriver.setExceptionsEnabled(true);
    JCuda.setExceptionsEnabled(true);

    String kernelsPath = "./kernels/";
    String args =
        "-D BLOCKSIZE=" + BLOCKSIZE + " " +
        "-D BLOCKSIZEMINUS1=" + (BLOCKSIZE - 1) + " " +
        "-D AVOIDBANKCONFLICTS=" + 0 + " ";

    /*Modo CREATE*/
    //Loading kernels from GPUadjustRow_kernel.cu
    adjustRowL = KernelLauncher.create(
        kernelsPath + "GPUadjustRow_kernel.cu",
        "adjustRowL_kernel", args);
    adjustRowU = adjustRowL.forFunction(
        "adjustRowU_kernel");

    //Loading kernels from GPUeliminateBlock_kernel.cu
    eliminateBlockL = KernelLauncher.create(
        kernelsPath + "GPUeliminateBlock_kernel.cu",
        "eliminateBlockL_kernel", args);
    eliminateBlockU = eliminateBlockL.forFunction(
        "eliminateBlockU_kernel");

    //Loading kernels from GPUeliminateCol_kernel.cu
    eliminateColL = KernelLauncher.create(
        kernelsPath + "GPUeliminateCol_kernel.cu",
        "eliminateColL_kernel", args);
    eliminateColU = eliminateColL.forFunction(
        "eliminateColU_kernel");

    //Loading kernels from GPUeliminateRest_kernel.cu
    eliminateRestL = KernelLauncher.create(
        kernelsPath + "GPUeliminateRest_kernel.cu",
        "eliminateRestL_kernel", args);
    eliminateRestU = eliminateRestL.forFunction(
        "eliminateRestU_kernel");

    //Loading kernels from GPUnormalizeDiag_kernel.cu
    normalizeDiag = KernelLauncher.create(
        kernelsPath + "GPUnormalizeDiag_kernel.cu",
        "normalizeDiag_kernel", args);

    //Loading kernels from GPUsetIdentity_kernel.cu
    setIdentity = KernelLauncher.create(
        kernelsPath + "GPUsetIdentity_kernel.cu",
        "GPUsetIdentity", args);
}
```

- Método de inversión:

```
/**
 * Invert the matrix A with 'size' rows and 'size' columns, and store the
 * result in the in invA.<br />
 * <br />
 * Note that the size must be a multiple of the BLOCKSIZE!
 *
 * @param A The input matrix
 * @param invA The inverted matrix
 * @param size The number of rows and columns of A
 */
public static void invert(float A[], float invA[], int size)
{
    if (adjustRowL == null)
    {
        init();
    }

    Pointer dDataIn = new Pointer();
    Pointer dDataInv = new Pointer();
    int size2InBytes = size * size * Sizeof.FLOAT;

    // Allocating memory for the data matrix and
    // identity matrix
    cudaMalloc(dDataIn, size2InBytes);
    cudaMalloc(dDataInv, size2InBytes);

    // Prepare the calculation of the identity matrix
    cudaMemset(dDataInv, 0, size2InBytes);

    // Transfer the matrix from host to device
    cudaMemcpy(dDataIn, Pointer.to(A),
        size2InBytes, cudaMemcpyHostToDevice);

    // Used SP/MP for calculations
    dim3 one = new dim3(1, 1, 1);
    dim3 idyThreads = new dim3(BLOCKSIZE, 1, 1);
    dim3 idyBlocks = new dim3(size / BLOCKSIZE, 1, 1);
    dim3 nThreads = new dim3(BLOCKSIZE, BLOCKSIZE, 1);
    dim3 nBlocks = new dim3(size / BLOCKSIZE, 1, 1);
    dim3 nBlocksRest = new dim3(size / BLOCKSIZE, size / BLOCKSIZE, 1);

    // Calculate the identity matrix
    // Original call:
    // GPUsetIdentity <<< idyBlocks, idyThreads >>> (dDataInv, size);
    setIdentity.setup(idyBlocks, idyThreads).call(dDataInv, size);
    cudaThreadSynchronize();
```

```
// Calculate the left diagonal Matrix (L)
for (int i = 0; i < size; i += BLOCKSIZE)
{
    int offset = i * size + i;

    // Step 1:
    // Calculate the triangle matrix
    // Store the pivot elements to left part of the triangle
    eliminateBlockL.setup(one, nThreads).call(
        at(dDataIn, offset), size);
    cudaThreadSynchronize();

    // Step 2:
    // Calculate the rest of the rows with the pivot
    // elements from step 1
    adjustRowL.setup(nBlocks, nThreads).call(
        at(dDataIn, i * size), at(dDataIn, offset),
        at(dDataInv, i * size), size, i);
    cudaThreadSynchronize();

    // Step 3:
    // Fill the columns below the block with the pivot elements. They
    // are used to get the columns to zero and multiply with the row
    eliminateColL.setup(nBlocks, nThreads).call(
        at(dDataIn, i), size, i);
    cudaThreadSynchronize();

    // Step 4:
    // Adjust the rest of the Matrix with the calculated pivot
    // elements: El_new_0 -= (p0+p1+p2..+p15) * El_piv_0
    eliminateRestL.setup(nBlocksRest, nThreads).call(
        dDataIn, dDataInv, size, i);
    cudaThreadSynchronize();
}

// Set the left lower diagonal matrix to zero
for (int i = 1; i < size; i++)
{
    int offset = i * size;
    cudaMemset(at(dDataIn, offset), 0, i * Sizeof.FLOAT);
}
cudaThreadSynchronize();
```

```
        // Calculate the right diagonal Matrix (U)
        for (int i = (size - BLOCKSIZE); i >= 0; i -= BLOCKSIZE)
        {
            int offset = i * size + i;

            // Step 1:
            // Calculate the triangle matrix
            // Store the pivot elements to right part of the triangle
            eliminateBlockU.setup(one, nThreads).call(
                at(dDataIn, offset), size);
            cudaThreadSynchronize();

            // Step 2:
            // calculate the rest of the rows with the pivot
            // elements from step 1
            adjustRowU.setup(nBlocks, nThreads).call(
                at(dDataIn, offset), at(dDataInv, i * size), size, i);
            cudaThreadSynchronize();

            // Step 3:
            // Fill the columns below the block with the pivot elements. They
            // are used to get the columns to zero and multiply with the row
            eliminateColU.setup(nBlocks, nThreads).call(
                at(dDataIn, i), size, i);
            cudaThreadSynchronize();

            // Step 4:
            // Adjust the rest of the Matrix with the calculated pivot
            // elements: El_new_0 -= (p0+p1+p2..+p15) * El_piv_0
            eliminateRestU.setup(nBlocksRest, nThreads).call(
                dDataIn, dDataInv, size, i);
            cudaThreadSynchronize();
        }

        // Force the diagonal entries to 1
        for (int i = 0; i < size; i += BLOCKSIZE)
        {
            int rowOffset = i * size;

            normalizeDiag.setup(nBlocks, nThreads).call(
                at(dDataIn, rowOffset), at(dDataInv, rowOffset), size, i);
            cudaThreadSynchronize();
        }

        // Copy the results back and clean up
        cudaMemcpy(Pointer.to(invA), dDataInv,
            size2InBytes, cudaMemcpyDeviceToHost);
        cudaFree(dDataIn);
        cudaFree(dDataInv);
    }
```

- Método de obtención de punteros:

```
    /**
     * Returns a pointer which is computed from the given pointer by
     * adding the given number of float elements to its address.
     *
     * @param p The input pointer
     * @param floatOffset The offset in number of float elements
     * @return The resulting pointer
     */
    private static Pointer at(Pointer p, int floatOffset)
    {
        return p.withByteOffset(floatOffset * Sizeof.FLOAT);
    }
}
```

4º. Hallamos la matriz w mediante la operación indicada al principio del apartado de entrenamiento.

$$[w_1 \quad \cdots \quad w_n] = [y_1 \quad \cdots \quad y_n] \cdot \begin{bmatrix} \phi_1(\vec{e_1}) & \phi_1(\vec{e_2}) & \cdots & \phi_1(\vec{e_n}) \\ \phi_2(\vec{e_1}) & \phi_2(\vec{e_2}) & & \phi_2(\vec{e_n}) \\ & \vdots & \ddots & \vdots \\ \phi_n(\vec{e_1}) & \phi_n(\vec{e_2}) & \cdots & \phi_n(\vec{e_n}) \end{bmatrix}^{-1}$$

```
        //Realizamos el producto de matrices
        double matrizProducto[][] = productoMatrices(matrizY, matrizRBFinversa);
```

```
//Multiplica matrices de dimensiones compatibles
private double[][] productoMatrices(double[][] matrizA, double[][] matrizB){
    int lonxA = matrizA.length;
    int lonyA = matrizA[0].length;
    int lonxB = matrizB.length;
    int lonyB = matrizB[0].length;

    if(lonyA != lonxB)
    {
        return null;
    }

    double[][] matrizResultado = new double[lonxA][lonyB];

    //i será la dimensión en común que recorrerá elementos de A y B.
    //x,y recorreran matrizResultado
    for(int x=0;x<lonxA;x++)
    {
        for(int y=0;y<lonyB;y++)
        {
            double sumatoria = 0;
            for(int i=0;i<lonyA;i++)
            {
                sumatoria = sumatoria + (matrizA[x][i] * matrizB[i][y]);
            }
            matrizResultado[x][y] = sumatoria;
        }
    }
    return matrizResultado;
}
```

5º. Finalmente transformamos la matriz w (*matrizProducto*) en el vector $\vec{w}$.

$$[w_1 \quad \cdots \quad w_n] \Rightarrow \vec{w}$$

```
//Transformamos esa matrizProducto que tendrá dimensión 1xN al vectorW
for(int i=0;i<matrizProducto[0].length;i++)
{
    vectorW[i] = matrizProducto[0][i];
}
```

3.1.3. Operación de la red

Igual que en el programa para la red neuronal artificial RBF de tipo lineal.

3.2. Estadísticas de la simulación

A lo largo de diversas simulaciones hemos creado, entrenado y operado redes RBF, a lo largo de las cuales, al igual que hicimos con RBF lineal, se han ido aumentando la cantidad de vectores del conjunto de datos de entrada E (variable *N*), para explotar hasta sus máximas capacidades el ordenador y analizar la respuesta del mismo.

El número de vectores han ido aumentando en múltiplos de 16 hasta llegar a la máxima capacidad de CUDA con 3856 vectores del conjunto de datos de entrada E (variable *N*). Para valores superiores, el ordenador se queda sin memoria de procesamiento.

Las gráficas que muestran el comportamiento de las simulaciones se muestran a continuación:

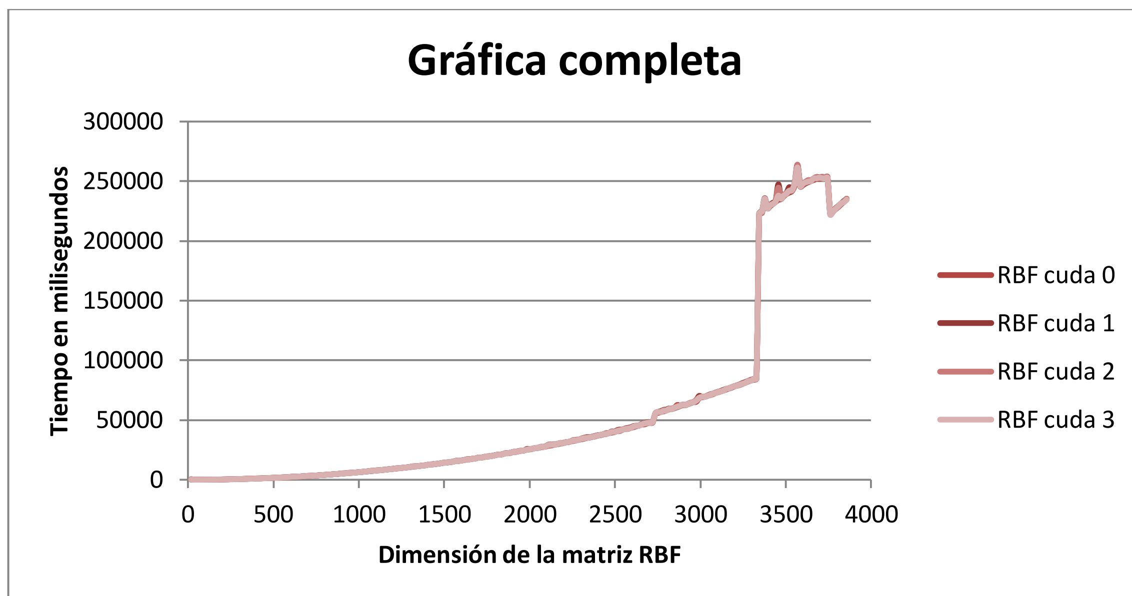

Figura 7. Gráfica completa de las simulaciones de RBF CUDA.

En la gráfica podemos ver como el crecimiento exponencial es progresivo y proporcional, hasta llegar entorno a dimensiones 3300. A partir de entonces se disparan los tiempos de ejecución (debido al aumento de tráfico que existe entre la memoria de la tarjeta gráfica y la RAM) aunque dentro de dicha subida vuelven a presentar un crecimiento progresivo (con algunos picos).

A continuación mostramos la misma gráfica en dos partes: su crecimiento inicial y los datos finales, para ver más claramente lo que ya habíamos podido observar en la gráfica completa:

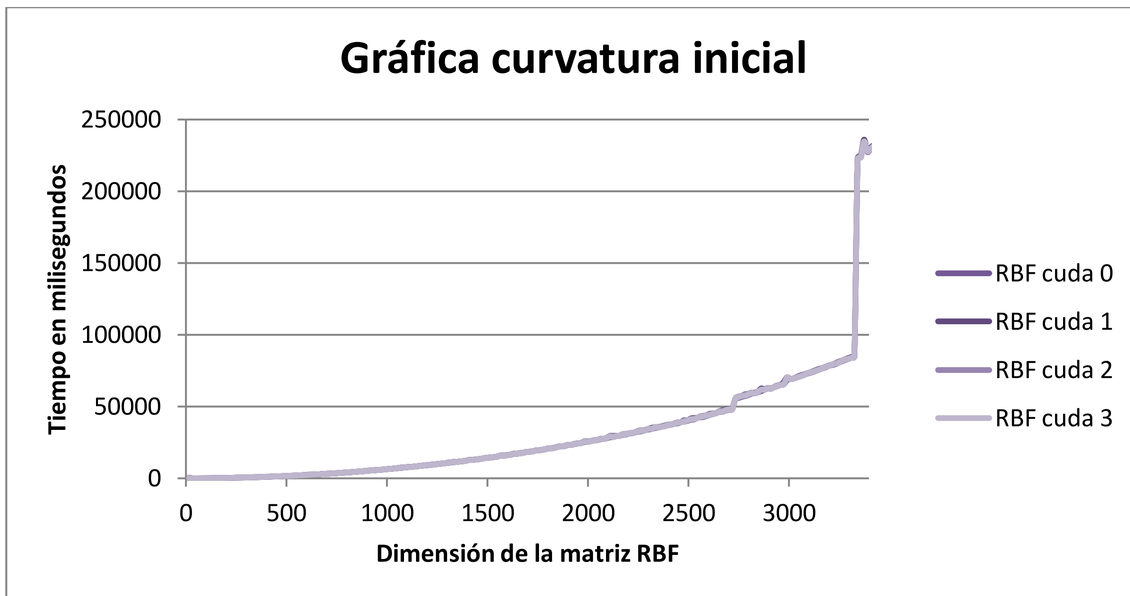

Figura 8. Curvatura inicial de las simulaciones de RBF CUDA.

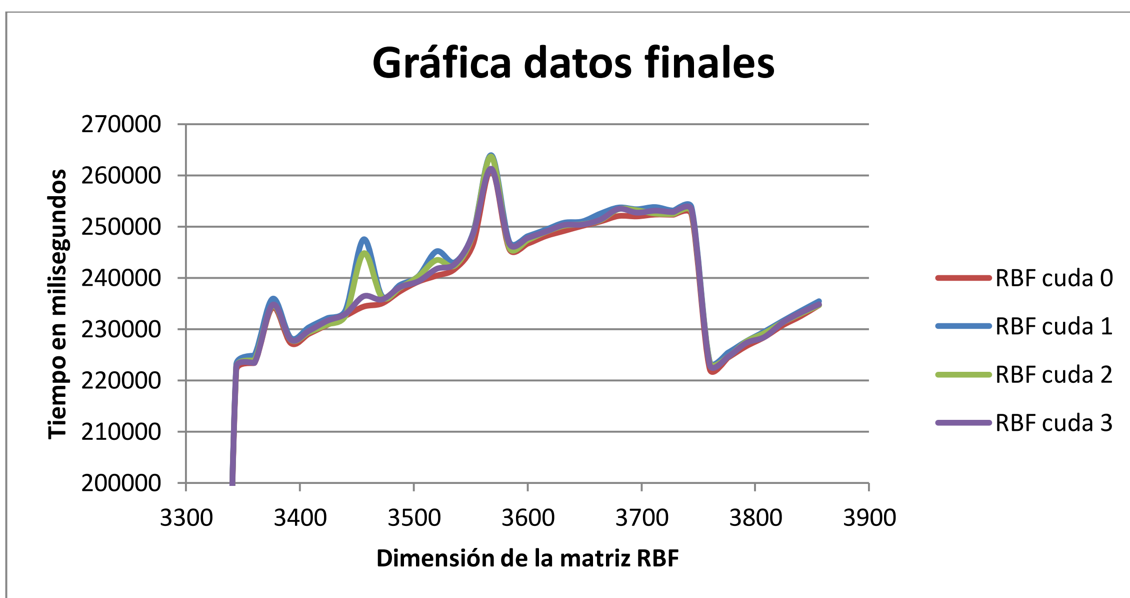

Figura 9. Datos finales de las simulaciones de RBF CUDA.

Es curioso notar que con dimensiones a partir de los 3760 vuelven a bajar los tiempos de ejecución, aunque posteriormente siguen experimentando el mismo índice de crecimiento hasta colapsar la memoria de procesamiento.

En el siguiente punto se lleva a cabo una comparativa de las simulaciones con RBF lineal y RBF CUDA.

4. Comparativa RBF

El fin principal de este proyecto es analizar las mejoras que podemos obtener utilizando la tecnología CUDA de NVIDIA en el funcionamiento de redes neuronales artificiales. Por ello, es necesario llevar a cabo una comparativa de las simulaciones realizadas para las redes neuronales artificiales RBF de forma lineal y las redes neuronales artificiales RBF con la ayuda de CUDA.

En este punto nos vamos a centrar en comparar las simulaciones realizadas para las redes neuronales artificiales SOM de forma lineal y con la ayuda de CUDA.

4.1. Comparativa de las gráficas

En primer lugar volvamos a ver las gráficas completas de ambos tipos de simulaciones:

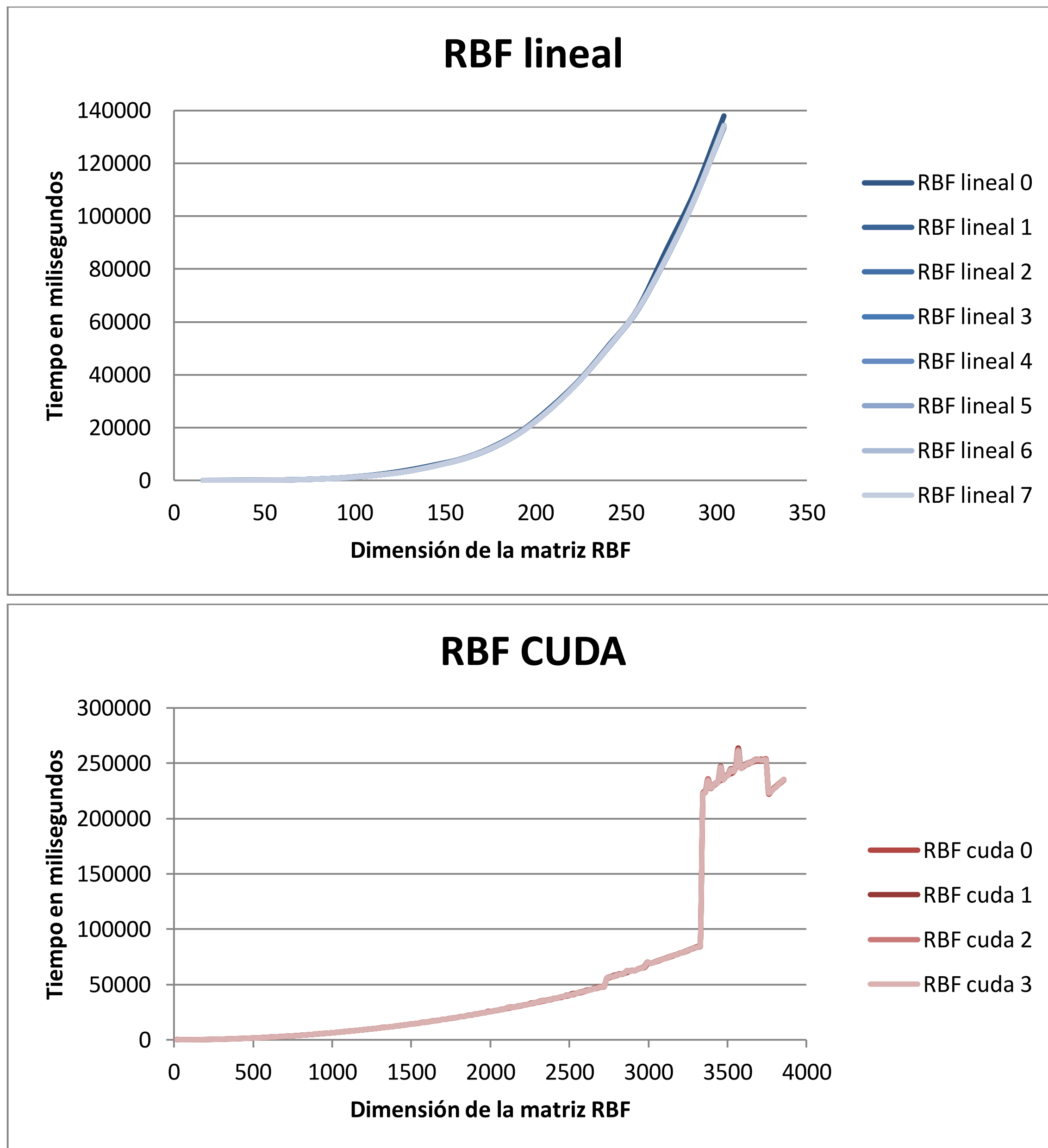

Figura 10. Gráficas completas de las simulaciones de RBF lineal y RBF CUDA.

A simple vista, comparando las gráficas de RBF lineal y de RBF CUDA, podemos observar que el crecimiento que experimentan las simulaciones de RBF lineal es un crecimiento exponencial. Sin embargo, el crecimiento descrito por RBF CUDA es un exponencial más suave, aunque con algunos picos, tal y como hemos indicado en el punto anterior.

La forma más sencilla de comparar dichas simulaciones es mediante un gráfica conjunta, por ello a continuación mostramos una gráfica en la que se incluyen las simulaciones de RBF lineal en tonos azules y las de RBF CUDA en tonos granate:

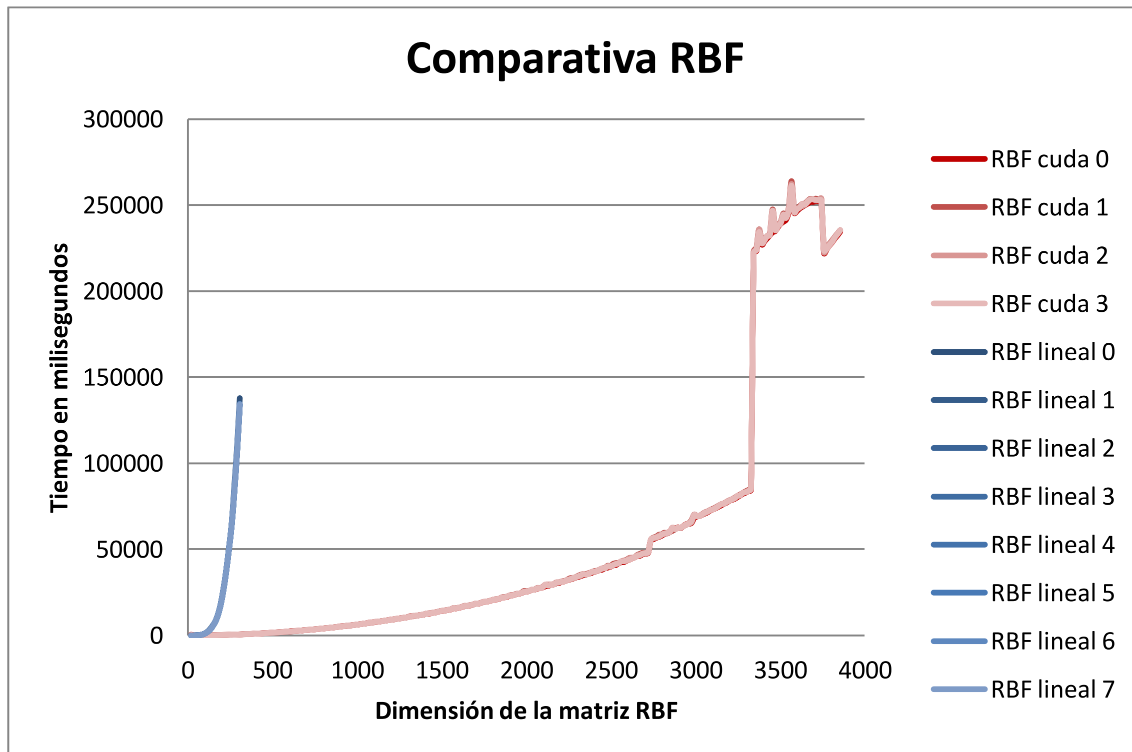

Figura 11. Gráfica completa conjunta/comparativa de las simulaciones de RBF lineal y RBF CUDA.

Las simulaciones de RBF mediante el empleo de la tecnología CUDA ofrecen mejores tiempos de ejecución y además, una mayor capacidad de cómputo, llegando a operar con matrices de hasta 3856x3856. Como podemos observar en esta última gráfica, tal y como perseguíamos en este proyecto, para la simulación de redes neuronales artificiales RBF la tecnología CUDA constituye una herramienta clave y muy eficiente.

4.2. Resumen

RBF Lineal	RBF CUDA
• Capacidad máxima de ejecución: matrices de 304x304.	• Capacidad máxima de ejecución: matrices de 3856x3856.
• Crecimiento de tipo exponencial.	• Crecimiento más constante, exponencial suave. • Presenta picos en los tiempos de ejecución para valores finales.
• Mayores tiempos de ejecución para una red neuronal artificial RBF con un determinado número de vectores de entrada.	• Menores tiempos de ejecución para una red neuronal artificial RBF con un determinado número de vectores de entrada.
• Mayor tiempo de ejecución en la inversión de matrices. • Método de Faddeev-Leverrier.	• Menor tiempo de ejecución en la inversión de matrices. • Clase *MatrixInvert.java* de JCuda.

Tabla 1. Resumen de la comparativa de las simulaciones de RBF lineal y RBF CUDA.

Capítulo 5: Redes neuronales artificiales SOM

1. Mapas autoorganizativos (SOM)

Una de las propiedades del cerebro humano es su capacidad de asociar conceptos con categorías. En la década de los 80, Kohonen mostró cómo construir una tipo de red neuronal artificial capaz de extraer la estructura, en términos de categorías, de un conjunto de datos. Estas redes llevan su nombre, pero también son conocidas como SOM (*Self-Organizing Maps* o Mapas Autoorganizativos). De esta forma se desarrolló una red neuronal capaz de realizar tareas de clasificación.

Las características principales de las redes neuronales artificiales SOM son las siguientes:

- Poseen un solo nivel con muchas conexiones.
- Se necesita iniciar los pesos de estas conexiones.
- Las neuronas compiten de forma que las ganadoras son las únicas que modifican los pesos asociados a sus conexiones.
- Estas redes agrupan las neuronas en diferentes clases.
- El aprendizaje es posible con la identificación de la estructura de los datos: correlaciones, agrupaciones, redundancia, etc.

En los mapas autoorganizativos, las neuronas de salida se disponen habitualmente de forma matricial, aunque esto depende de la aplicación concreta. Esta disposición realizada por el usuario determina qué neuronas de salida son vecinas.

Los patrones de entrenamiento que están próximos entre sí en el espacio de entrada (donde la proximidad se determina por alguna métrica, normalmente la distancia euclídea), deben proyectarse en las neuronas de salida que están también próximas unas a otras. Con este tipo de entrenamiento se consigue que, sin necesidad de conocer la salida deseada, la respuesta de la red sea similar para vectores similares en el espacio de entrada (aprendizaje no supervisado).

Las redes neuronales artificiales de tipo SOM tiene múltiples aplicaciones, algunas de las cuales son:

- **Agrupamiento (*Clustering*):** Los datos de entrada deben ser agrupados en “clusters” (grupos), y el sistema de procesamiento de datos tiene que encontrar

esos grupos inherentes en los datos de entrada. La salida del sistema debería proporcionar el grupo al que pertenece el patrón de entrada.

- **Reducción de la dimensión:** Los datos de entrada se agrupan en un subespacio que tiene una dimensión más pequeña que la de los datos. El sistema tiene que aprender a realizar una proyección óptima, de tal manera que la mayoría de las variaciones (diferencias) en los datos de entrada se mantengan en los datos de salida.
- **Extracción de características:** El sistema tiene que extraer las características útiles de la señal de entrada.

1.1. Aprendizaje competitivo

Las redes SOM utilizan el aprendizaje competitivo. Este método de aprendizaje agrupa vectores de entrada próximos entre sí de forma que cada grupo represente una clase o categoría.

La red actúa como un clasificador, que toma un vector de entrada y devuelve un índice correspondiente al nodo o neurona que mejor encaja en una determinada clase. La red se organiza a sí misma de forma que las neuronas que corresponden a una determinada clase tiene respuestas similares, y todas las neuronas representan a alguna de estas clases.

Para llevar a cabo el proceso de clasificación es necesario definir una distancia o medida de similitud para evaluar el grado de semejanza de los patrones, como, por ejemplo: distancia euclídea, absoluta, Manhattan, Voronoi, etc. Normalmente la más utilizada es la distancia euclídea (que es la que vamos a utilizar en nuestras simulaciones), que dados dos puntos se define como:

$$d(x_i, x_j) = \|x_i - x_j\|_2 = \sqrt{\sum_k [x_i(k) - x_j(k)]^2}$$

1.2. Topología de las redes SOM

Las redes neuronales artificiales SOM constan de dos capas: una capa de entrada y una capa de salida.

- **Capa de entrada:** Las neuronas en la capa de entrada únicamente distribuyen los valores de la señal de entrada a las neuronas de la siguiente capa, dispuestas en un

vector unidimensional o bidimensional de forma circular, triangular, rectangular, etc., con un cierto orden topológico que depende de la aplicación. Las señales de entrada x se aplican a todas las neuronas de salida en paralelo.

- **Capa de salida:** Una neurona de salida cualquiera k recibe su entrada a través de un conjunto de conexiones ponderadas. Los pesos de estas conexiones se representan por el correspondiente vector de pesos w_k. Todas las neuronas de la capa de salida están conectadas a todas la neuronas de la capa da entrada con pesos w_{jk}.

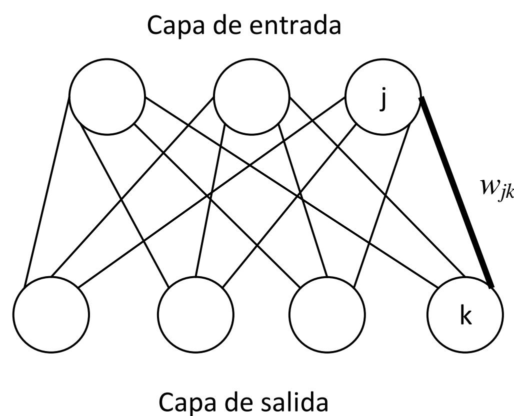

Figura 12. Topología de una red SOM.

Para cada vector de entrada x, sólo se activa una neurona de la capa de salida ("neurona ganadora"). En un entrenamiento correcto de la red, todos los vectores de entrada de una misma clase tendrán como ganadora a la misma neurona de salida.

1.3. Método de aprendizaje de las redes SOM

Las redes SOM incorporan a la regla de aprendizaje competitivo un cierto grado de sensibilidad con respecto al conjunto de las neuronas vecinas a la ganadora. Por tanto, el proceso de aprendizaje es local (la modificación de los pesos se realiza en un mayor o menor grado dependiendo de la proximidad a la neurona ganadora).

Kohonen propuso que las neuronas de salida interactuaran lateralmente, llegando así a los mapas de características autoorganizativos. La propiedad más importante del modelo es el concepto de aprendizaje de un vecindario próximo a la neurona ganadora. El tamaño del vecindario decrece en cada iteración.

En el resumen esquemático de funcionamiento se explicarán los pasos y las fórmulas a llevar a cabo en el entrenamiento de este tipo de RNA.

1.4. Resumen esquemático

1.4.1. Inicialización de la red

Conjunto E	Conjunto R	Vector X
$E = \begin{Bmatrix} \overrightarrow{e_1} \\ \vdots \\ \overrightarrow{e_n} \end{Bmatrix}$	$R = \begin{Bmatrix} \overrightarrow{r_1} \\ \vdots \\ \overrightarrow{r_n} \end{Bmatrix}$	$\vec{x} = \begin{Bmatrix} x_1 \\ \vdots \\ x_L \end{Bmatrix}$

1º. Cargamos los valores del archivo:

- **L:** Longitud de los vectores del conjunto E, conjunto C y el vector X.
- **tamE:** Cantidad de vectores del conjunto de datos de entrada E.
- $\boldsymbol{\eta}$**:** Definido por el usuario con un valor [0,1].
- **Conjunto E:** Conjuntos de vectores de entrada $\vec{e}$.
- **Vector X:** Vector de entrada X.

2º. Inicializamos:

- **tamR:** Cantidad de vectores de tipo neurona del conjunto R.
- **Conjunto R:** Conjuntos de vectores de tipo neurona $\vec{r}$. Valores aleatorios de rango [-0’01, 0’01].
- **Neurona ganadora:** Resultado final de la red. Mostraremos su distancia y su posición.

1.4.2. Entrenamiento de la red

Esta fase se repetirá un número variable de veces (de 1 a k veces) y cuyo propósito final es entrenar la red para que se produzca un aprendizaje en la misma.

Para ello, en cada iteración, llevaremos a caso los siguientes pasos:

1º. Para cada vector $\vec{e}$ del conjunto de entrada E, hallamos el vector tipo neurona $\vec{r}$ ganador. La neurona $\vec{r}$ ganadora será aquella que tenga una menor distancia con dicho vector $\vec{e}$, y para determinar dicha distancia emplearemos la fórmula de la distancia euclídea anteriormente expuesta:

$$\vec{r}_{ganadora}: \quad d(\vec{e}_i, \vec{r}) = \|\vec{e}_i - \vec{r}\|_2 = \sqrt{\sum_l [\, e_i[l] - r[l]\,]^2}$$

2º. Una vez hemos hallado la neurona $\vec{r}$ ganadora para el vector $\vec{e}_i$, procedemos a reajustar el conjunto R de neuronas a partir del vector $\vec{e}_i$ y la neurona $\vec{r}_{ganadora}$. Para ello para cada neurona $\vec{r_j}$ realizaremos los siguientes pasos:

a) Hallar el vector distancia existente entre el vector $\vec{e}_i$ y el vector $\vec{r}_{ganadora}$, que no es otra cosa que el resultado de restar dos vectores:

$$\overrightarrow{dis}(\vec{e}_i, \vec{r}_{ganadora}) = \{(e_i[0] - r_{ganadora}[0]), \ldots, (e_i[L-1] - r_{ganadora}[L-1])\}$$

b) Hallar la distancia que hay entre el vector neurona $\vec{r_j}$ y el vector $\vec{r}_{ganadora}$:

$$d(\vec{r}_{ganadora}, \vec{r_j}) = \|\vec{r}_{ganadora} - \vec{r_j}\|_2 = \sqrt{\sum_l [r_{ganadora}[l] - r_j[l]\,]^2}$$

c) A partir de la distancia anterior obtendremos la función vecindad:

$$\nabla(\vec{r}_{ganadora}, \vec{r_j}) = e^{-\frac{[d(\vec{r}_{ganadora}, \vec{r_j})]^2}{\sigma}}$$

El valor de sigma σ se establece por el usuario. En los entrenamientos realizados para este proyecto se le asignará el valor 0’05.

d) Finalmente reasignamos el vector $\vec{r_j}$ con la siguiente función:

$$\vec{r_j} = \vec{r_j} + \eta \cdot \overrightarrow{dis}(\vec{e}_i, \vec{r}_{ganadora}) \cdot \nabla(\vec{r}_{ganadora}, \vec{r_j})$$

3º. Una vez que hemos reasignado todos los vectores $\vec{r}$ de tipo neurona, seleccionamos el siguiente vector de entrada del conjunto E y volvemos a realizar los mismos pasos.

Cuando hayamos recorrido todos los vectores del conjunto E, tantas iteraciones como se nos haya indicado (de 1 a k veces), daremos por finalizado el aprendizaje de nuestra red, y por tanto, también supondrá el fin de la fase de entrenamiento.

1.4.3. Operación de la red

En esta fase calcularemos el vector $\vec{r}$ de tipo neurona ganador, y mostraremos su distancia y su posición dentro de la red.

Para hallar dicho vector $\vec{r}_{ganadora}$, llevaremos a cabo los siguientes pasos:

1º. Para cada vector $\vec{r}$ de tipo neurona, hallamos la distancia con el vector de entrada $\vec{x}$:

$$d(\vec{x},\vec{r}) = \|\vec{x} - \vec{r}\|_2 = \sqrt{\sum_l [x[l] - r[l]\,]^2}$$

Elegimos la neurona ganadora. La neurona $\vec{r}_{ganadora}$ será aquella que tenga una menor distancia con dicho vector $\vec{x}$.

2º. Mostramos la distancia de la neurona $\vec{r}_{ganadora}$ y su posición dentro de la red.

2. SOM Lineal

Al igual que se hiciese con las redes RBF, a la hora de simular una red neuronal artificial SOM, se ha desarrollado un programa en lenguaje de programación Java en el que se crea, se entrena y se opera una red neuronal artificial SOM.

Para ello se han seguido las pautas expuestas en el punto anterior, donde se ha explicado tanto desarrolladamente como a nivel esquemático el funcionamiento de este tipo de redes neuronales artificiales.

En este punto, y siguiendo el esquema del punto anterior, vamos a explicar y resumir el código Java desarrollado para esta red. Posteriormente se mostrarán las estadísticas obtenidas de las diferentes simulaciones.

2.1. Codificación de la red

2.1.1. Inicialización de la red

1º. Declaramos las variables globales que vamos a utilizar y definimos η.

```
int L = 0; //Leer archivo para saber longitud de los vectores.
int tamE = 0; //Leer archivo para saber cantidad de vectores del conjunto
              //de datos de entrada E.
int tamR = 0; //Cantidad de vectores de tipo neurona R.

double nu = (double)Math.random()* 1; // Definido por el usuario [0,1]

ArrayList conjuntoE; //Conjunto de vectores de entrada E.
ArrayList conjuntoR; //Conjunto de vectores de tipo neurona R.
double vectorX[]; //Vector de entrada.

File archivo = new File("EntrenamientoSOM.txt");
```

2º. Cargamos los valores del archivo:

- **L:** Longitud de los vectores del conjunto E, conjunto C y el vector X.
- **tamE:** Cantidad de vectores del conjunto de datos de entrada E.
- **Conjunto E:** Conjuntos de vectores de entrada $\vec{e}$.
- **Vector X:** Vector de entrada X.

```
BufferedReader entrada = new BufferedReader(new FileReader(archivo));

entrada.readLine();//Línea 1 - Comentario

L = Integer.parseInt(entrada.readLine()); //Línea 2.

entrada.readLine();entrada.readLine();
entrada.readLine();//Línea 3-5 - Comentarios

tamE = Integer.parseInt(entrada.readLine()); //Línea 6.

entrada.readLine();//Línea 7 - Comentario

conjuntoE = new ArrayList();
double vectorE[];
for(int i=0;i<tamE;i++)
{
    String linea = entrada.readLine();
    StringTokenizer espacio=new StringTokenizer(linea," ");

    vectorE = new double[L];
    for(int j=0;j<L;j++)
    {
        //Leer elementos de archivo.
        double elemento = Double.parseDouble(espacio.nextToken());
        vectorE[j]=elemento;
    }
    conjuntoE.add(vectorE);
}

entrada.readLine();entrada.readLine();//Comentarios

String linea = entrada.readLine();
StringTokenizer espacio=new StringTokenizer(linea," ");
vectorX = new double[L];
for(int j=0;j<L;j++)
{
    //Leer elementos de archivo.
    double elemento = Double.parseDouble(espacio.nextToken());
    vectorX[j]=elemento;
}

entrada.readLine();entrada.readLine();//Comentarios
```

3º. Inicializamos:

- **tamR:** Cantidad de vectores de tipo neurona del conjunto R. En el código vendrá definido según un bucle *for* que irá incrementando el número de neuronas a lo largo de las simulaciones de redes SOM.
- **Conjunto R:** Conjuntos de vectores de tipo neurona $\vec{r}$. Valores aleatorios de rango [-0’01, 0’01].

- **Neurona ganadora:** Resultado final de la red. Mostraremos su distancia y su posición. En realidad no llega a inicializarse en sí, puesto que sus datos se extraerán y mostrarán de una neurona del conjunto R que sea elegida.

```
conjuntoR = new ArrayList();
double vectorR[];
for(int i=0;i<tamR;i++)
{
    vectorR = new double[L];
    for(int j=0;j<L;j++)
    {
        double elemento = (double)(Math.random()*(0.02))+(-0.01);
        vectorR[j]=elemento;
    }
    conjuntoR.add(vectorR);
}
```

2.1.2. Entrenamiento de la red

Esta fase se repetirá un número variable de veces (de 1 a k veces) y cuyo propósito final es entrenar la red para que se produzca un aprendizaje en la misma.

Para ello, en cada iteración, llevaremos a caso los siguientes pasos:

1º. Para cada vector $\vec{e}$ del conjunto de entrada E, hallamos el vector tipo neurona $\vec{r}$ ganador. La neurona $\vec{r}$ ganadora será aquella que tenga una menor distancia con dicho vector $\vec{e}$, y para determinar dicha distancia emplearemos la fórmula de la distancia euclídea anteriormente expuesta:

$$\vec{r}_{ganadora}: \quad d(\vec{e}_i, \vec{r}) = \|\vec{e}_i - \vec{r}\|_2 = \sqrt{\sum_{l} [\, e_i[l] - r[l]\,]^2}$$

De esta manera la función distancia sería:

```
private double distancia(double[] vector1, double[] vector2)
{
    double sumatoria = 0;
    for(int i=0;i<L;i++)
    {
        double a = vector1[i] - vector2[i];
        double b = a * a;
        sumatoria = sumatoria + b;
    }
    double dis = Math.sqrt(sumatoria);
    return dis;
}
```

Y hallaríamos la neurona ganadora así (para posteriormente llamar a la función de reajuste):

```
for(int i=0;i<tamE;i++)
{
    double vectorE[] = (double[])conjuntoE.get(i);

    //La distancia ganador es la mínima. Inicializamos la posición
    // de la neurona ganadora y su distancia al primer elemento.
    int posGanador = 0;
    double[] vectorR = (double[])conjuntoR.get(0);
    double distanciaGanador = distancia(vectorE,vectorR);

    //Hallamos la posición de la neurona con la menor distancia.
    for(int j=1;j<tamR;j++)
    {
        vectorR = (double[])conjuntoR.get(j);
        double dis = distancia(vectorE, vectorR);
        if (dis < distanciaGanador)
        {
            posGanador = j;
            distanciaGanador = dis;
        }
    }
    //Ya tenemos la posición de la neurona ganadora y extraemos
    // dicha neurona.
    double ganador[] = (double[])conjuntoR.get(posGanador);

    //Función de reajuste
    asignacionR(ganador, vectorE);
}
```

2º. Una vez hemos hallado la neurona $\vec{r}$ ganadora para el vector $\vec{e}_i$, procedemos a reajustar el conjunto R de neuronas a partir del vector $\vec{e}_i$ y la neurona $\vec{r}_{ganadora}$. Este método lo denominamos *asignacionR*. Para ello, primero hallamos el vector distancia existente entre el vector $\vec{e}_i$ y el vector $\vec{r}_{ganadora}$, que no es otra cosa que el resultado de restar dos vectores:

$$\overrightarrow{dis}(\vec{e}_i, \vec{r}_{ganadora}) =$$
$$= \{(e_i[0] - r_{ganadora}[0]), \dots, (e_i[L-1] - r_{ganadora}[L-1])\}$$

```
private void asignacionR(double[] ganador, double[] vectorE)
{
    //Hallamos el vector distancia
    double vectorDistancia[] = hallarVectorDistancia(vectorE,ganador);
```

La función del vector distancia sería:

```
private double[] hallarVectorDistancia(double[] vector1, double[] vector2)
{
    int longitud = vector1.length;
    double vectorDistancia[] = new double[longitud];
    for(int i=0;i<longitud;i++)
    {
        vectorDistancia[i] = vector1[i] - vector2[i];
    }
    return vectorDistancia;
}
```

Y posteriormente para cada neurona $\vec{r_j}$ realizaremos los siguientes pasos:

a) Hallar la distancia que hay entre el vector neurona $\vec{r_j}$ y el vector $\vec{r}_{ganadora}$:

$$d(\vec{r}_{ganadora},\vec{r_j}) = \|\vec{r}_{ganadora} - \vec{r_j}\|_2 = \sqrt{\sum_l [r_{ganadora}[l] - r_j[l]\,]^2}$$

```
for(int j=0;j<tamR;j++)
{
    double vectorR[] = (double[])conjuntoR.get(j);
    double nuevoVectorR[] = new double[L];

    //Hallamos la distancia
    double dis = distancia(ganador, vectorR);
```

b) A partir de la distancia anterior obtendremos la función vecindad:

$$\nabla(\vec{r}_{ganadora},\vec{r_j}) = \mathrm{e}^{-\frac{[d(\vec{r}_{ganadora},\vec{r_j})]^2}{\sigma}}$$

El valor de sigma σ se establece por el usuario. En los entrenamientos realizados para este proyecto se le asignará el valor 0’05.

```
    //Hallamos la función de vecindad
    double funcionVecindad = Math.exp(-((dis*dis)/0.05));
```

c) Finalmente reasignamos el vector $\vec{r_j}$ con la siguiente función:

$$\vec{r_j} = \vec{r_j} + \eta \cdot \overrightarrow{dis}(\vec{e_i},\vec{r}_{ganadora}\,) \cdot \nabla(\vec{r}_{ganadora},\vec{r_j})$$

```
//Hallamos el reajuste del vector R.j
for(int w=0;w<L;w++)
{
    nuevoVectorR[w] = vectorR[w] +
            (nu*vectorDistancia[w]*funcionVecindad);
}
//Sustituimos el vector R.j por el reajustado
conjuntoR.set(j, nuevoVectorR);
```

3º. Una vez que hemos reasignado todos los vectores $\vec{r}$ de tipo neurona, seleccionamos el siguiente vector de entrada del conjunto E y volvemos a realizar los mismos pasos.

Cuando hayamos recorrido todos los vectores del conjunto E, tantas iteraciones como se nos haya indicado (de 1 a k veces), daremos por finalizado el aprendizaje de nuestra red, y por tanto, también supondrá el fin de la fase de entrenamiento.

2.1.3. Operación de la red

En esta fase calcularemos el vector $\vec{r}$ de tipo neurona ganador, y mostraremos su distancia y su posición dentro de la red.

Para hallar dicho vector $\vec{r}_{ganadora}$, llevaremos a cabo los siguientes pasos:

1º. Para cada vector $\vec{r}$ de tipo neurona, hallamos la distancia con el vector de entrada $\vec{x}$:

$$d(\vec{x},\vec{r}) = \|\vec{x} - \vec{r}\|_2 = \sqrt{\sum_{l} [x[l] - r[l]\,]^2}$$

Elegimos la neurona ganadora. La neurona $\vec{r}_{ganadora}$ será aquella que tenga una menor distancia con dicho vector $\vec{x}$.

2º. Mostramos la distancia de la neurona $\vec{r}_{ganadora}$ y su posición dentro de la red.

```
private void operacion()
{
    //Iniciamos el ganador al primer elemento
    int posGanador = 0;
    double vectorR[] = (double[])conjuntoR.get(0);
    double disGanador = distancia(vectorX,vectorR);

    //Hallamos la neurona R ganadora
    for(int i=1;i<tamR;i++)
    {
        vectorR = (double[])conjuntoR.get(i);
        double dis = distancia(vectorX,vectorR);

        if(dis < disGanador)
        {
            disGanador = dis;
            posGanador = i;
        }
    }

    //Mostramos la posición y la distancia de la nuerona ganadora
    System.out.println("La neurona ganadora está en la posición "
            +posGanador+" con una distancia de "+disGanador);
}
```

2.2. Estadísticas de la simulación

Se han llevado a cabo varias simulaciones, en cada una de las cuales se han creado, entrenado y operado redes SOM, cuyas cantidades de vectores tipo neurona R han ido aumentando hasta explotar las máximas capacidades del ordenador y analizar la respuesta del mismo.

El número de neuronas han ido aumentando en múltiplos de 10.000, donde el ordenador experimenta un crecimiento más o menos constante, hasta llegar a la máxima capacidad del equipo con 2.560.000 neuronas donde el equipo presenta una gran subida en los tiempos de ejecución, llegando a no poder realizar más ejecuciones a partir de 2.570.000 neuronas por agotamiento de la memoria interna.

A continuación se muestran las gráficas de algunas de las simulaciones llevadas a cabo:

Figura 13. Gráfica completa de las simulaciones de SOM lineal.

Podemos observar como para el valor extremo de 2.560.000 neuronas el tiempo de ejecución se dispara.

A continuación veremos la gráfica ignorando este último valor extremo, para ver mejor el comportamiento de los tiempos de ejecución:

Figura 14. Gráfica sin valor extremo de las simulaciones de SOM lineal.

Y finalmente tomaremos valores más pequeños para poder apreciar con mayor claridad la curvatura que experimenta:

Figura 15. Gráfica de la curvatura de las simulaciones de SOM lineal.

Todos estos datos, podrán analizarse mejor cuando contrastemos la información con los datos de ejecución de las redes neuronales artificiales SOM con la ayuda de CUDA de NVIDIA. La comparativa podremos apreciarla en los dos próximos puntos.

3. SOM CUDA

Al igual que para simular una red neuronal artificial SOM de forma lineal, se ha desarrollado un programa en lenguaje de programación Java en el que se crea, se entrena y se opera una red neuronal artificial SOM mediante el empleo de CUDA.

Lo que se persigue es incluir alguna mejora en la ejecución lineal de la red, mediante el empleo de funciones y librerías JCuda de CUDA de NVIDIA. Vamos a necesitar tanto librerías de JCuda genéricas, como las especiales de JCublas, cuya función explicábamos al inicio de este documento.

El programa en sí es el mismo, puesto que los resultados serán iguales, sin embargo en una parte del código se ha llevado a cabo dicha mejora que íbamos buscando.

Recordemos que las fases de desarrollo de la simulación de una red SOM son: inicialización, entrenamiento y ejecución. Puesto que la parte del programa con mayor tiempo de ejecución es el entrenamiento, allí es donde vamos a llevar a cabo la mejora y por tanto nos vamos a centrar sólo en la explicación de dicha fase.

3.1. Codificación de la red

3.1.1. Inicialización de la red

Igual que en el programa para la red neuronal artificial SOM de tipo lineal.

3.1.2. Entrenamiento de la red

Recordamos que esta fase se repetirá un número variable de veces (de 1 a k veces) y cuyo propósito final es entrenar la red para que se produzca un aprendizaje en la misma.

La mejora será introducida en el método de reajuste de las neuronas *asignaciónR* que hemos renombrado como *asignacionRCUDA*, por lo que el resto de métodos no volverán a mostrarse en esta explicación, ya que son exactamente los mismos.

Los pasos a seguir son los siguientes:

1º. Para cada vector $\vec{e}$ del conjunto de entrada E, hallamos el vector tipo neurona $\vec{r}$ ganador. La neurona $\vec{r}$ ganadora será aquella que tenga una menor distancia con dicho vector $\vec{e}$, y para determinar dicha distancia emplearemos la fórmula de la distancia euclídea anteriormente expuesta:

$$\vec{r}_{ganadora}: \quad d(\vec{e}_i, \vec{r}) = \|\vec{e}_i - \vec{r}\|_2 = \sqrt{\sum_l [\, e_i[l] - r[l]\,]^2}$$

2º. Una vez hemos hallado la neurona $\vec{r}$ ganadora para el vector $\vec{e}_i$, procedemos a reajustar el conjunto R de neuronas a partir del vector $\vec{e}_i$ y la neurona $\vec{r}_{ganadora}$. Para ello, primero hallamos el vector distancia existente entre el vector $\vec{e}_i$ y el vector $\vec{r}_{ganadora}$, que no es otra cosa que el resultado de restar dos vectores:

$$\overrightarrow{dis}(\vec{e}_i, \vec{r}_{ganadora}) = \\ = \{(e_i[0] - r_{ganadora}[0]), \dots, (e_i[L-1] - r_{ganadora}[L-1])\}$$

```
private void asignacionRCUDA(double[] ganador, double[] vectorE)
{
    //Hallamos el vector distancia
    double vectorDistancia[] = hallarVectorDistancia(vectorE,ganador);
```

Y posteriormente para cada neurona $\vec{r_j}$ realizaremos los siguientes pasos:

a) Hallar la distancia que hay entre el vector neurona $\vec{r_j}$ y el vector $\vec{r}_{ganadora}$:

$$d(\vec{r}_{ganadora},\vec{r_j}) = \|\vec{r}_{ganadora} - \vec{r_j}\|_2 = \sqrt{\sum_l [r_{ganadora}[l] - r_j[l]\,]^2}$$

b) A partir de la distancia anterior obtendremos la función vecindad:

$$\nabla(\vec{r}_{ganadora},\vec{r_j}) = e^{-\frac{[d(\vec{r}_{ganadora},\vec{r_j})]^2}{\sigma}}$$

El valor de sigma σ se establece por el usuario. En los entrenamientos realizados para este proyecto se le asignará el valor 0'05.

```
for(int j=0;j<tamR;j++)
{
    double vectorR[] = (double[])conjuntoR.get(j);

    //Hallamos la distancia
    double dis = distancia(ganador, vectorR);
    //Hallamos la función de vecindad
    double funcionVecindad = Math.exp(-((dis*dis)/0.05));
```

c) Finalmente reasignamos el vector $\vec{r_j}$ con la siguiente función:

$$\vec{r_j} = \vec{r_j} + \eta \cdot \overrightarrow{dis(\vec{e_i},\vec{r}_{ganadora}\,)} \cdot \nabla(\vec{r}_{ganadora},\vec{r_j})$$

Si observamos el siguiente trozo de código, en la ejecución de la red neuronal artificial SOM de forma lineal, el reajuste se hacía mediante un bucle:

```
//Hallamos el reajuste del vector R.j
for(int w=0;w<L;w++)
{
    nuevoVectorR[w] = vectorR[w] +
            (nu*vectorDistancia[w]*funcionVecindad);
}
//Sustituimos el vector R.j por el reajustado
conjuntoR.set(j, nuevoVectorR);
```

Sin embargo, en esta ocasión para evitarnos dicho bucle recurrimos a la capacidad de paralelización de las GPUs de CUDA de NVIDIA ayudándonos de las librerías JCuda y JCublas.

Si nos fijamos bien, la fórmula de reajuste no es más que un vector ($\vec{r_j}$) al que se le suma el resultado de la multiplicación de un número (η), por un vector ($\overrightarrow{dis}(\vec{e}_i,\vec{r}_{ganadora})$), por otro número ($\nabla(\vec{r}_{ganadora},\vec{r_j})$). O lo que es lo mismo, basándonos en la propiedad conmutativa de la multiplicación, según la cual el orden de los factores no altera el producto, la fórmula de reajuste no es más que un vector ($\vec{r_j}$) al que se le suma el resultado de la multiplicación de un número (resultado de $\eta \cdot \nabla(\vec{r}_{ganadora},\vec{r_j})$), por un vector ($\overrightarrow{dis}(\vec{e}_i,\vec{r}_{ganadora})$). Es decir:

$$\overrightarrow{vector_1} = \overrightarrow{vector_1} + num \cdot \overrightarrow{vector_2}$$

donde:

$$\overrightarrow{vector_1} = \vec{r_j}$$

$$num = \eta \cdot \nabla(\vec{r}_{ganadora},\vec{r_j})$$

$$\overrightarrow{vector_2} = \overrightarrow{dis}(\vec{e}_i,\vec{r}_{ganadora})$$

De esta forma, podemos utilizar un método de las librerías de JCublas que realizará esta operación de forma paralelizada. Este método se denomina *cublasDaxpy* y se encarga de sumar un vector1 al resultado del producto de un número por otro vector2 y guardar el resultado final en el vector1. Su llamada se realiza de la siguiente forma:

JCublas.cublasDaxpy(L, alpha, d_vector2, 1, d_vector1, 1);

donde:

L: Longitud de los vectores.

alpha: Número de tipo double. Aquí introducimos el producto de *nu* con la función vecindad.

d_vector2, d_vector1: Los punteros a los vectores.

Los números 1 indican el incremento de las posiciones en los vectores.

Una vez dicho todo esto, a continuación mostramos el código desarrollado con la ayuda de CUDA para la realización del reajuste mediante la función *cublasDaxpy* y finalmente la sustitución del vector $\vec{r_j}$ reajustado.

```
            //Hallamos el reajuste de R.j a través de CUDA
            // **************** CUDA ********************
            Pointer d_vectorDistancia = new Pointer();
            Pointer d_vectorR = new Pointer();
            double alpha = nu * funcionVecindad;

            //Reservamos memoria
            JCublas.cublasAlloc(L, Sizeof.DOUBLE, d_vectorDistancia);
            JCublas.cublasAlloc(L, Sizeof.DOUBLE, d_vectorR);

            //Inicializamos los vectores en CUDA
            JCublas.cublasSetVector(L, Sizeof.DOUBLE,
                    Pointer.to(vectorDistancia), 1, d_vectorDistancia, 1);
            JCublas.cublasSetVector(L, Sizeof.DOUBLE,
                    Pointer.to(vectorR), 1, d_vectorR, 1);

            //Operamos  alpha*vectorDistancia + vector R
            JCublas.cublasDaxpy(L, alpha, d_vectorDistancia, 1, d_vectorR, 1);

            //Recuperamos el vector
            JCublas.cublasGetVector(L, Sizeof.DOUBLE, d_vectorR, 1,
                    Pointer.to(vectorR), 1);
            // ***************** FIN CUDA **********************

            //Sustituimos el vector R.j una vez reajustado
            conjuntoR.set(j, vectorR);
        }
    }
```

Como podemos observar y leer en los comentarios, hemos de crear punteros a los vectores, reservarles memoria en CUDA, y una vez terminada la operación, volver a recuperar el vector a partir de su puntero asignado.

3º. Una vez que hemos reasignado todos los vectores $\vec{r}$ de tipo neurona, seleccionamos el siguiente vector de entrada del conjunto E y volvemos a realizar los mismos pasos.

Al igual que en el programa de SOM lineal, una vez hayamos recorrido todos los vectores del conjunto E, tantas iteraciones como se nos haya indicado (de 1 a k veces), daremos por finalizado el aprendizaje de nuestra red, y por tanto, también supondrá el fin de la fase de entrenamiento.

3.1.3. Operación de la red

Igual que en el programa para la red neuronal artificial SOM de tipo lineal.

3.2. Estadísticas de la simulación

Al igual que en la simulaciones de SOM lineal, para probar CUDA con este tipo de redes neuronales artificiales se han creado, entrenado y operado redes SOM, cuyas cantidades de vectores tipo neurona R han ido aumentando para explotar hasta sus máximas capacidades el ordenador y analizar la respuesta del mismo.

Las neuronas han ido aumentando en múltiplos de 10.000 hasta llegar a la máxima capacidad del equipo con 2.560.000 neuronas, a partir de ahí el ordenador no podía seguir computando por agotamiento de la memoria interna. El crecimiento temporal es, en la mayor medida, constante.

A continuación se muestran las gráficas de algunas de las simulaciones llevadas a cabo:

Figura 16. Gráfica completa de las simulaciones de SOM CUDA.

La gráfica muestra como para el valor extremo de 2.560.000 neuronas el tiempo de ejecución se dispara.

Vamos a mostrar la gráfica sin este valor extremo para poder apreciar más el crecimiento de los tiempos de ejecución:

Figura 17. Gráfica sin valor extremo de las simulaciones de SOM CUDA.

A continuación mostramos más de cerca la curvatura que describe la gráfica:

Figura 18. Gráfica de la curvatura de las simulaciones de SOM CUDA.

Aunque a primera vista no se aprecien tantas diferencias entre las curvaturas que describen las gráficas de SOM lineal y SOM CUDA, cabe destacar que los tiempos de ejecución con CUDA son mucho mayores, por lo que parece ser que no hemos conseguido

nuestro objetivo de mejora. Daremos las razones del porqué ocurre esto en el siguiente punto, donde se realiza una comparativa de ambos tipos de simulaciones de las redes neuronales artificiales SOM.

4. Comparativa SOM

En este punto nos vamos a centrar en comparar las simulaciones realizadas para las redes neuronales artificiales SOM de forma lineal y con la ayuda de CUDA.

4.1. Comparativa de las gráficas

En los dos puntos anteriores nos hemos centrado en exponer 3 tipos de gráficas: gráfica completa, gráfica sin valor extremo y gráfica viendo la curvatura más de cerca. En esta comparativa vamos a intentar seguir estas pautas para que nos sea más fácil apreciar sus diferencias.

Figura 19. Gráfica completa conjunta/comparativa de las simulaciones de SOM lineal y SOM CUDA.

A simple vista no se aprecia de forma clara que el tiempo de ejecución con CUDA es mucho mayor que la simulación de forma lineal, aunque sí se nota una gran diferencia en el valor extremo.

Figura 20. Gráfica conjunta/comparativa sin valor extremo de las simulaciones de SOM lineal y SOM CUDA.

Figura 21. Gráfica de la curvatura conde la curvatura conjunta/comparativa de las simulaciones de SOM lineal y SOM CUDA.

En cambio, en estas dos gráficas sí que podemos ver la gran diferencia de tiempo que ejecución que existe entre SOM lineal y SOM CUDA. Conforme aumenta el número de neuronas, SOM CUDA experimenta una mayor subida en su tiempo de ejecución.

Pero, si lo que nosotros buscábamos era una mejoría, y en teoría con CUDA y la función *cublasDaxpy* nos ahorrábamos un bucle, ¿por qué ocurre esto?

La razón es muy simple. En realidad el tiempo de ejecución de la función *cublasDaxpy* roza los 0.0 milisegundos, por lo que sí mejoraría el tiempo con CUDA con respecto al lineal. Sin embargo, para la utilización de CUDA hemos de utilizar punteros a los vectores con los que vamos a operar, lo que conlleva escrituras en memoria, y una vez hayamos realizado el cálculo, debemos recuperar dicho vector resultado, lo que conlleva una lectura de memoria. Es decir, lo que realmente consume tiempo de ejecución de las redes neuronales artificiales SOM son los accesos de memoria.

Dicho de otra forma, el tiempo de ejecución del bucle para reajustar una neurona R para SOM lineal (donde recorremos y calculamos cada una de sus L posiciones), es mucho menor al tiempo de acceso a memoria que realizamos para realizar el reajuste para SOM CUDA, aunque el tiempo de ejecución de la operación de reajuste en sí es menor.

De hecho, es tan simple demostrar estas afirmaciones, como mostrar las gráficas de crecimiento que experimentan las redes conforme se aumenta el número de neuronas.

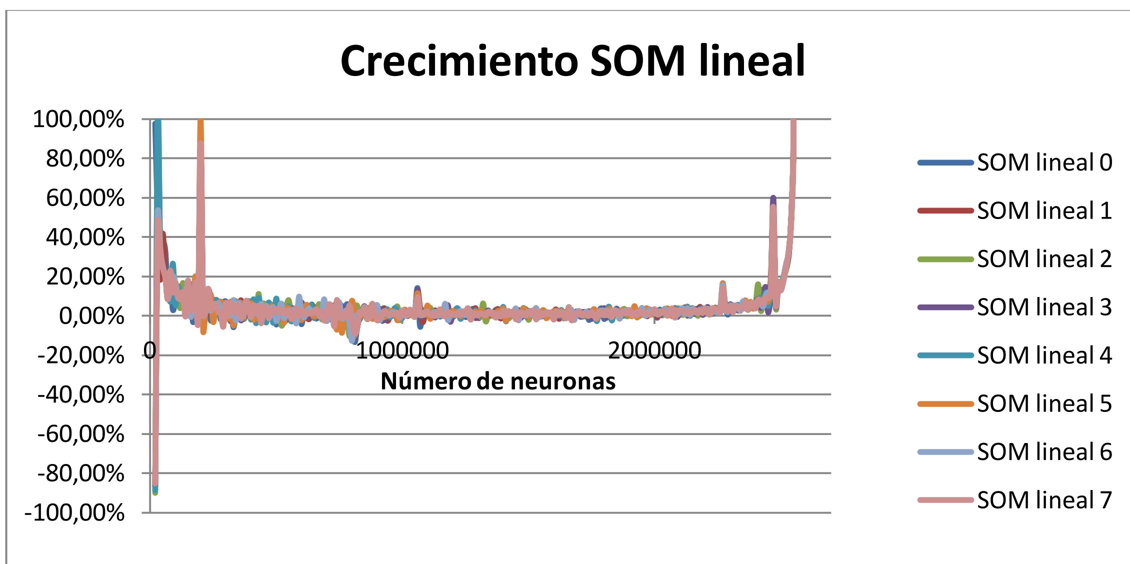

Figura 22. Gráfica de la tasa de crecimiento de las simulaciones de SOM lineal.

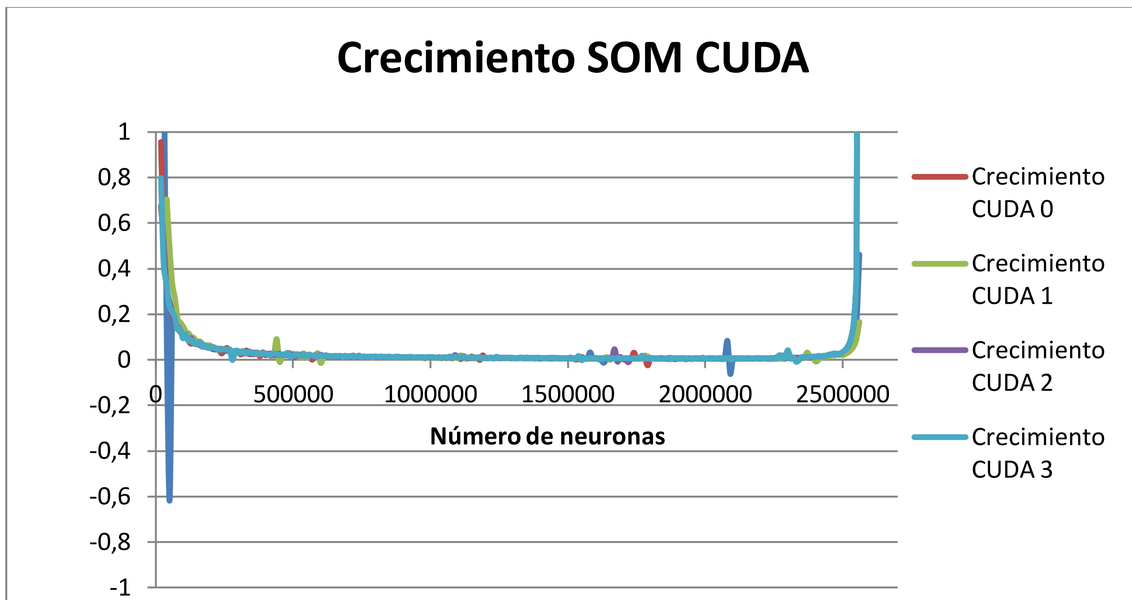

Figura 23. Gráfica de la tasa de crecimiento de las simulaciones de SOM CUDA.

Como vemos el crecimiento de SOM CUDA, excepto en valores extremos, es mucho más constante que en SOM lineal, que varía mucho más. Esto es debido a que los tiempos de acceso a memoria con CUDA son constantes, pero se realizan más accesos al haber más neuronas.

Vamos a ver estas gráficas de crecimiento mucho más de cerca, donde se aprecia mucho mejor esta diferencia:

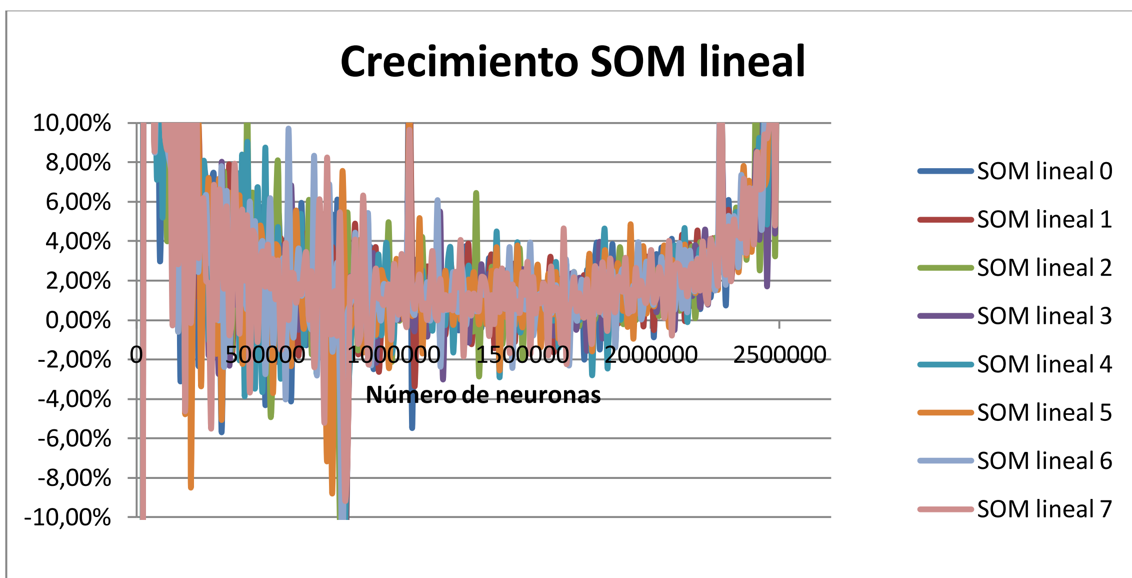

Figura 24. Gráfica parcial de la tasa de crecimiento de las simulaciones de SOM lineal.

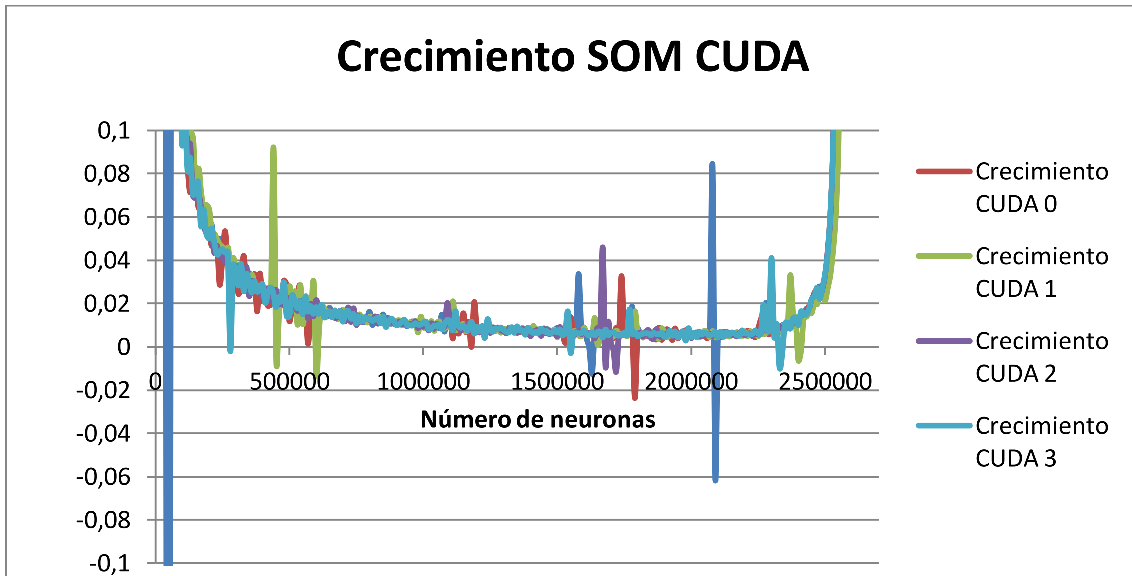

Figura 25. Gráfica parcial de la tasa de crecimiento de las simulaciones de SOM CUDA.

4.2. Resumen

SOM Lineal	SOM CUDA
• Capacidad máxima de ejecución 2.560.000 neuronas.	• Capacidad máxima de ejecución 2.560.000 neuronas.
• Pico en los tiempos de ejecución para el valor extremo.	• Pico en los tiempos de ejecución para el valor extremo.
• Menores tiempos de ejecución para una red neuronal artificial SOM con un determinado número de neuronas.	• Mayores tiempos de ejecución para una red neuronal artificial SOM con un determinado número de neuronas.
• Mayor tiempo de ejecución de la operación de reajuste.	• Menor tiempo de ejecución de la operación de reajuste (pero mucho gasto en los accesos a memoria).
• Porcentaje de crecimiento alto para valores neuronales pequeños y muy grandes.	• Porcentaje de crecimiento alto para valores neuronales pequeños y muy grandes.
• Media del porcentaje de crecimiento: 9’21%.	• Media del porcentaje de crecimiento: 6’03%.
• Porcentaje de crecimiento muy variable.	• Porcentaje de crecimiento más constante.

Tabla 2. Resumen de la comparativa de las simulaciones de SOM lineal y SOM CUDA.

Capítulo 6: Conclusiones finales.

Objetivos

El objetivo principal de este proyecto era usar la tecnología CUDA de NVIDIA para desarrollar algoritmos de paralelización para simular diferentes redes neuronales artificiales y analizar su comportamiento.

Para ello, se han escogido dos redes neuronales distintas, ya que se perseguía obtener resultados de aplicación totalmente diferentes y opuestos.

Simulaciones

Se han desarrollado dos tipos de redes neuronales artificiales: redes neuronales RBF y redes neuronales SOM. El mayor tiempo de ejecución en ambos tipos de redes se da en la fase de entrenamiento y por ello, ésta ha sido objeto de estudio para la aplicación de la tecnología CUDA de NVIDIA.

Las redes neuronales artificiales RBF, en su fase de entrenamiento, para llevar a cabo su aprendizaje han de realizar la inversión de una matriz, y este paso constituye la mayor carga en tiempo de ejecución, debido a la gran cantidad de llamadas recursivas y operaciones que han de realizarse. La aplicación de la tecnología CUDA de NVIDIA para la realización de la inversión de matrices supone una mejora abismal en los tiempos de ejecución de las redes neuronales artificiales RBF.

Las redes neuronales artificiales SOM en su fase de entrenamiento llevan a cabo un gran número de operaciones mediante el empleo de bucles. En este caso, la aplicación de la tecnología CUDA de NVIDIA se aplica en el reajuste de neuronas de tipo R, y aunque la operación de ajuste mejora los tiempos de ejecución lineales, el coste de acceso a memoria es tan grande, que para las redes neuronales artificiales SOM no supone una mejora en los tiempos de ejecución.

Resultados

Con todo ello, podemos concluir que el empleo de la tecnología CUDA de NVIDIA para llevar a cabo las simulaciones de redes neuronales artificiales, no siempre es beneficioso para la mejora de los tiempos de ejecución de dichas redes.

Aunque las operaciones en sí en las que se emplea la tecnología CUDA, mejoran las operaciones en ejecución lineal, los costes de tiempo en accesos a memoria son muy altos, por lo que podemos afirmar que el empleo de CUDA beneficia a aquellos tipos de redes neuronales artificiales en los que se realiza un gran número de llamadas recursivas, muchos cálculos y/o pocos accesos a memoria.

En conclusión, la tecnología CUDA de NVIDIA mejora sustancialmente los tiempos de ejecución de algoritmos de simulación de redes neuronales artificiales, pero sólo en determinado tipo de casos.

Trabajos futuros

Este proyecto se ha centrado desde un principio en el empleo de la tecnología CUDA de NVIDIA para la aceleración en la resolución de algoritmos explotando su paralelismo, en concreto, algunos algoritmo de redes neuronales artificiales. Para trabajos futuros, siguiendo esta línea de trabajo, podríamos extender el estudio a otros tipos de redes neuronales, tanto de aprendizaje supervisado como de aprendizaje no supervisado.

Por otra parte, se ha demostrado la gran influencia que la memoria de la tarjeta gráfica puede ejercer sobre los tiempos de ejecución de los algoritmos. Dicha influencia depende además de dos casos muy concretos: la capacidad de memoria y el tiempo de accesos a dicha memoria (lecturas y escrituras). Por todo ello, para la realización de otros trabajos futuros relacionados con este proyecto, podríamos centrarnos en:

- Determinar la influencia de la capacidad de las memorias de las tarjetas en el rendimiento de los algoritmos (probar distintas tarjetas gráficas NVIDIA con la tecnología CUDA).
- Medir con precisión los tiempos de retardo por la comunicación entre la tarjeta gráfica y el ordenador (accesos a memoria).

Objetivos cumplidos a nivel formativo

Desde el punto de vista formativo, a lo largo de todo este proyecto la alumna ha ampliado sus conocimientos en el campo de la programación y ha conseguido cumplir una serie de objetivos formativos:

- Mejorar el aprendizaje sobre los procesos de paralelización mediante el empleo de GPUs.
- Aprender el funcionamiento de la tecnología CUDA de NVIDIA.
- Emplear la tecnología CUDA de NVIDIA.
- Aprender el funcionamiento de distintos tipos de redes neuronales.
- Simular redes neuronales artificiales de tipo RBF y de tipo SOM.

Bibliografía

[1] Página web de NVIDIA.
http://www.nvidia.es/object/cuda_what_is_es.html

[2] NVIDIA CUDA Compute Unified Device Architecture. Guía de programación.
http://developer.download.nvidia.com/compute/cuda/1_0/NVIDIA_CUDA_Programming_Guide_1.0.pdf

[3] Artículos sobre CUDA
http://www.hardwarezone.com/

[4] David Luebke, NVIDIA Research
"CUDA Fundamentals"
SIGGRAPH2008

[5] Página web de JCuda
http://jcuda.org/

[6] Eckel, Bruce (2003)
"Thinking of Java"
Prentice Hall

[7] Java 2 Plataform (para consulta de componentes Java)
http://docs.oracle.com/javase/1.4.2/docs/api/

[8] Página del Entorno de Desarrollo Integrado (IDE) Netbeans (para la descarga de su plataforma, consulta, etc.).
www.netbeans.org

[9] Monografía sobre redes neuronales
http://www.monografias.com/trabajos/redesneuro/redesneuro.shtml

[10] J.M. Corchado, F. Díaz, L. Borrajo, F. Fernández (2000)
"Redes Neuronales artificiales. Un enfoque práctico"
Universidad de Vigo

[11] Simon Haykin (1999)
"Neural Networks"
Prentice Hall

[12] R. Flórez López, J. M. Fernández Fernández (2008)
“Las redes neuronales artificiales: Fundamentos teóricos y aplicaciones prácticas”
Netbiblio