UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf
Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

Implementación de un módulo para el entrenamiento y evaluación de redes neuronales mediante GPUs

Proyecto Final de Carrera

Ingeniería Informática

Autor: Adrián Palacios Corella

Directores: Salvador España Boquera
Francisco Zamora Martínez

17 de septiembre de 2012

A mis padres, que tanto han luchado para que yo pudiese perseguir mis sueños y no me distrajese con la realidad. A mis amigos, por aguantarme a lo largo de esta aventura, especialmente a Raül, que se ha ganado el cielo. Muchas gracias.

A Salvador España y Francisco Zamora, por todos los conocimientos adquiridos y la atención recibida al trabajar juntos en este proyecto tan personal. A María José Castro, por darme la oportunidad de colaborar con un grupo de investigación tan apasionado. Gracias.

Resumen

Las redes neuronales artificiales son un modelo matemático ampliamente conocido y utilizado en reconocimiento estadístico de formas para estimar funciones a partir de datos de entrenamiento supervisado. En particular, bajo ciertas condiciones bastante generales, pueden estimar distribuciones de probabilidad. Uno de sus mayores inconvenientes es el alto coste computacional para realizar el proceso de entrenamiento y, en menor medida, de evaluación, especialmente cuando aumenta el tamaño de la red o el número de muestras.

En este contexto se enmarca el trabajo que presentamos en esta memoria, cuyo objetivo es mejorar el entrenamiento de estas redes y así poder realizar entrenamientos para tareas más complejas que requieren redes con una topología de mayor tamaño y procesar corpus muy grandes.

El objetivo de este trabajo ha sido la mejora de la implementación de un toolkit de entrenamiento y evaluación de redes neuronales ya existente, incluyendo una versión en lenguaje CUDA para ser ejecutada en GPUs.

El toolkit de partida utilizado en este proyecto se denomina "April" (acrónimo de "A Pattern Recognizer In Lua") y ha sido desarrollado con los lenguajes C++ y Lua por los directores de este proyecto final de carrera y permite entrenar redes neuronales artificiales de tipo *feedforward* utilizando el algoritmo de retropropagación del gradiente. El uso de bibliotecas de cálculo matricial, como la biblioteca MKL de Intel, junto al uso del modo de entrenamiento conocido como "bunch" permite acelerar de manera sustancial las etapas de evaluación y entrenamiento de estas redes, lo cual resulta casi imprescindible en la práctica cuando los experimentos reales requieren periodos de tiempo que van de varios días a varias semanas de CPU.

A pesar de las características del toolkit original, resulta muy conveniente mejorar todavía más el rendimiento. Para tal fin, este trabajo explora una vía más prometedora para incrementar la velocidad de entrenamiento reside en el uso de las unidades gráficas de proceso (GPU) cuya arquitectura SIMD las hace idóneas para cálculos de tipo matricial como el requerido en el entrenamiento de este tipo de redes neuronales. Trabajos previos encontrados en la literatura reportan mejoras de incluso dos órdenes de magnitud al usar GPUs.

Para incluir estas mejoras, ha sido necesario un esfuerzo en la parte de desarrollo de C++ de cara a realizar un remodelado del diseño tanto para

integrar la parte GPU, como el uso de la biblioteca CUBLAS y el desarrollo de kernels CUDA. Este trabajo de diseño e implementación ha sido validado con un estudio experimental comparando las diversas versiones en función de diversos parámetros como el tamaño de las redes o el uso de modo bunch.

Palabras clave: redes neuronales, neural networks, backpropagation, gpgpu, gpu, momentum, weight decay, blas, cblas, mkl, cuda, cublas, april.

Índice general

Índice de figuras

Índice de cuadros

Glosario

ACML AMD Core Math Library

ANN Redes Neuronales Artificiales o "Artificial Neural Nets"

API Interfaz de Programación de Aplicaciones o "Application Programming Interface"

ATLAS Automatically Tuned Linear Algebra Software

BLAS Basic Linear Algebra Subprograms

BP Algoritmo de Retropropagación del Error o "Backpropagation"

CPU Unidad Central de Procesamiento o "Central Processing Unit"

CUDA Arquitectura de Dispositivos de Cómputo Unificado o "Compute Unified Device Architecture"

DTW Dynamic Time Warping

FANN Fast Artificial Neural Network Library

GPGPU General-Purpose Computing on Graphics Processing Units

GPU Unidad de Procesamiento Gráfico o "Graphics Processing Unit"

MKL Math Kernel Library

MLP Perceptrón Multicapa o "Multilayer Perceptron"

MSE Error cuadrático medio o "Mean Square Error"

NIST National Institute of Standards and Technology

OCR Reconocimiento óptico de carácteres o "Optical Character Recognition"

PNG Portable Network Graphics

SIMD Single Instruction Multiple Data

SNNS Stuttgart Neural Network Simulator

θ Umbral de una neurona.

η Factor de aprendizaje del BP (learning rate).

μ Factor del "momentum" del BP.

α Valor del Weight Decay.

Capítulo 1

Introducción

El principal objetivo de este trabajo ha sido el desarrollo de una implementación eficiente del algoritmo de retropropagación del gradiente (Algoritmo de Retropropagación del Error o "Backpropagation" (BP)) mediante el uso de una Unidad de Procesamiento Gráfico o "Graphics Processing Unit" (GPU). En la actualidad ya existen algunas implementaciones de este algoritmo con la ayuda de una GPU [SMU10, JPJ08], por lo que en este capítulo se explicarán las razones y motivaciones que nos han llevado a realizar un diseño propio y en qué contexto se ha desarrollado. Por otra parte, a lo largo del trabajo, se mostrarán las ventajas y los inconvenientes que nuestro proyecto presenta comparado con proyectos del mismo tipo.

1.1. Motivación

Las Redes Neuronales Artificiales o "Artificial Neural Nets" (ANN) son un modelo matemático empleado en el ámbito del aprendizaje automático para el modelado de funciones complejas de las que disponemos de conjuntos de entrenamiento supervisado (datos de la entrada y de la salida que se desea).

En este trabajo nos centramos en las redes no recurrentes o "*feedforward*". Estas redes se pueden considerar como funciones continuas de R^n a R^m, siendo n y m el número de entradas y de salidas, respectivamente. La principal característica que las hace tan atractivas en numerosos campos reside en la capacidad de aproximación "universal" que tienen, entendiendo por esto que la red puede aproximar otras funciones con una aproximación arbitrariamente buena bajo ciertas condiciones: la red ha de tener al menos una capa oculta, suficientes parámetros y disponer de suficientes datos de entrenamiento.

Entre otras aplicaciones, se ha demostrado que pueden aprender a estimar probabilidades condicionales, por lo que pueden ser útiles en el ámbito del reconocimiento estadístico de formas: clasificación, estimación de modelado de lenguaje, asociadas a modelos ocultos de Markov para el reconocimiento de voz, de escritura, etc.

Lo que hace realmente interesantes a las ANN es la gran variedad de algoritmos de entrenamiento, que viene a ser realizar el ajuste de los parámetros o pesos de la

ANN para aproximarse a los datos de entrenamiento. Más adelante comentaremos el problema del sobreentrenamiento o *overfitting*.

Entre los métodos más populares de entrenamiento podemos destacar los basados en descenso por gradiente. Para explicar el aprendizaje mediante descenso por gradiente conviene realizar la siguiente analogía: Podemos considerar todos los parámetros de la red como un punto en un espacio. Cada punto de este espacio es una configuración de la red para una misma topología y cada punto ofrecerá, ante un conjunto de entradas a la red, unas salidas que se ajustarán mejor o peor a la " salida deseada" (estamos en un marco de aprendizaje supervisado). Dada una forma de medir este error (existen varias, como el error cuadrático medio o la entropia cruzada, que se verán posteriormente) podemos imaginar una superfície que tiene como base el espacio de pesos de la red y como alturas los errores que la red presenta ante los datos de entrenamiento.

Aprender consiste en elegir el punto de menor error. Idealmente se trataría del mínimo global, pero el descenso por gradiente consiste en partir de un punto arbitrario (inicialización de la red con valores aleatorios) y de ir bajando en dirección de mayor pendiente, lo que suele resultar en un mínimo local. El algoritmo más conocido se llama retropropagación, en inglés, "backpropagation" (BP).

La popularidad de los métodos basados en descenso por gradiente, en contraposición a la de los métodos de segundo orden, se debe en gran parte al menor coste de actualización de los pesos con el tamaño de la red. Los métodos basados en descenso por gradiente tienen un coste de $O(W)$ operaciones por actualización, siendo W el número de pesos o parámetros de la red. Por otra parte, los métodos de segundo orden tienen un coste de $O(W^3)$ operaciones por actualización que, como mucho, puede reducirse hasta $O(W^2)$ si empleamos métodos de segundo orden menos costosos, como es el caso de los métodos *Quasi-Newton*. Por otra parte, los métodos de descenso por gradiente tienen como principal inconveniente una convergencia bastante lenta que obliga a realizar un gran número de presentaciones o ciclos de las muestras empleadas para el entrenamiento. A pesar de requerir más iteraciones, estos métodos presentan mejoras conforme aumenta el tamaño de las redes a entrenar.

El algoritmo BP puede funcionar en dos modos: El modo "batch" (conocido como modo *off-line*) y el modo "stochastic" (también conocido como modo *on-line*). El modo "batch" calcula la media de los errores producidos por todas las muestras de entrenamiento respecto a las salidas deseadas. Ese error es propagado hacia atrás de forma que los pesos son ajustados según el factor de aprendizaje escogido. En la práctica, el modo *off-line* tiene una convergencia hacia la solución muy lenta, por lo que su uso es prácticamente nulo. El modo "on-line", en cambio, consiste en calcular el error producido por una muestra y propagarlo hacia atrás para efectuar la actualización de los pesos. De esta manera, después de la ejecución del algoritmo BP en modo "on-line" sobre una ANN cualquiera, los pesos se habrán actualizado tantas veces como muestras de entrenamiento hayan sido presentadas. Este modo consigue acelerar la velocidad de convergencia del algoritmo BP respecto al modo "off-line", por lo que en la práctica éste es el modo de ejecución más extendido. Existe una solución intermedia denominada "modo bunch" (quizás se podría traducir "por lotes") consistente en realizar el modo batch sobre pequeños subconjuntos de datos para poder tener la ventaja del modo online y

permitir aprovechar su cálculo en forma matricial para acelerar sustancialmente los cálculos.

El algoritmo BP clásico requiere que especifiquemos un valor denominado *learning rate* o factor de aprendizaje que indica cuánto modificar los pesos en la dirección del gradiente tras la presentación de los datos (sea modo batch, online o bunch), pero el algoritmo puede incluir factores adicionales para producir variantes del algoritmo con ciertas propiedades que pueden resultar de gran interés. Este es el caso del algoritmo BP con "momentum". El momentum es un factor adicional cuyo objetivo es el de acelerar la convergencia del algoritmo. El factor expresa un porcentaje del valor de los pesos de la iteración anterior que se añade al cálculo del valor de los nuevos pesos, produciendo así un efecto de inercia en la red. El coste temporal no se ve afectado en gran medida por este factor, aunque el coste espacial sí que se incrementa,[1] pues es necesario almacenar los pesos de la iteración anterior en una zona de memoria dedicada a este fin.

Otra variante del algoritmo es el BP con *weight decay*. Este factor se añade a la fórmula clásica para la actualización de pesos restando un porcentaje del valor del peso en la iteración anterior, produciendo así un efecto de poda en los pesos de la red que no se vean reforzados de manera significativa por el propio algoritmo. Se puede considerar que actúa como un factor de regularización y que puede evitar en parte el sobreentrenamiento. En esta variante, los costes temporales y espaciales se ven incrementados de la misma manera que lo hacían en el caso del BP con momentum.

Hasta aquí se han presentado, de forma breve, las distintas variantes del algoritmo BP, junto a sus ventajas y desventajas principales. Las herramientas actuales para entrenar ANN (normalmente mediante BP) pueden clasificarse en dos tipos:

- Herramientas que permiten describir ANN priorizando la flexibilidad sobre la eficiencia. Estas herramientas plantean un diseño enfocado hacia a las neuronas, de forma que éstas funcionan como unidades de cálculo individuales. Estas neuronas se suelen almacenar en listas enlazadas, por lo que se introduce un coste temporal notable a la hora de realizar cualquier tipo de cálculos sobre la ANN. La ventaja de este plantemiento frente al otro es que se puede describir cualquier tipo de topología sin ningún tipo de esfuerzo adicional.

- Herramientas que permiten usar sólamente ANN con topologías predeterminadas (normalmente redes de tipo "capa a capa", donde la topología de la red consiste en una capa de entrada, un número determinado de capas ocultas y una capa de salida, estando las conexiones entre todas las neuronas de capas adyacentes). Estas herramientas plantean un diseño enfocado hacia las conexiones, en donde la importancia de las neuronas disminuye y los componentes importantes de la ANN son las capas (que en realidad son un conjunto de neuronas) y las conexiones (que son un conjunto de pesos). La ventaja de estas herramientas frente a las del otro tipo es que son mucho más eficientes. Esta aceleración se consigue gracias a la posibilidad de

[1] Aunque no asintóticamente, solo se dobla.

especializar el código para el tipo de topología seleccionada, normalmente utilizando cálculos de tipo matricial.

A primera vista parece que hay que escoger entre una de las dos alternativas, favoreciendo así la flexibidad para describir topologías o la eficiencia para realizar cálculos. El propósito de este proyecto es construir una herramienta con un buen compromiso entre flexibilidad y eficiencia. El resultado es un toolkit que permite utilizar redes mucho más complejas que las típicas "capa a capa" sin dejar de ser *tan eficiente* como las implementaciones especializadas. Esta flexibilidad se ha resuelto mediante el diseño de una jerarquía de clases extensible y variada. Para resolver el problema de la eficiencia será propuesto un nuevo modo de ejecución que trabaja con grupos de neuronas, lo que permite utilizar la forma matricial del algoritmo BP, pero que nos permite acelerar la aplicación mediante el uso de librerías de álgebra lineal y el uso de GPUs para realizar cálculos paralelos.

Con el paso del tiempo, las tarjetas gráficas (GPUs) se han convertido en uno de los componentes más importantes de los ordenadores personales, hasta el punto de ser el componente más caro y sofisticado de éstos. Las condiciones extremas, a las que las GPU se someten al procesar los gráficos de juegos recientes, obliga a los fabricantes a ingeniar arquitecturas más complejas y desarollar nuevas tecnologías para poder lidiar con esas aplicaciones.

Una de estas tecnologías es Arquitectura de Dispositivos de Cómputo Unificado o "Compute Unified Device Architecture" (CUDA), el modelo/lenguaje de programación de GPUs propuesto por Nvidia. Las GPU que disponen de la tecnología CUDA permiten al programador el acceso al dispositivo para ejecutar costosas operaciones de cálculo de forma paralela y eficiente, consiguiendo de este modo una aceleración sin precedentes en aquellos algoritmos cuya ejecución se basa en este tipo de cálculos complejos y altamente paralelizables. Nuestro objetivo es emplear la GPU en la ejecución del algoritmo BP, de forma que un período de aprendizaje que puede durar semanas o incluso meses, se convierta ahora en un entrenamiento que dure unos pocos días.

1.2. El toolkit April

April es un *toolkit* que reúne diversas utilidades para el reconocimiento de formas [Zam05, EZCG07]. Este paquete de herramientas, desarrollado por los directores de este proyecto desde 2004, incluye no solamente bibliotecas de entrenamiento de redes neuronales sino también diversos algoritmos para reconocimiento de secuencias (voz, escritura), *parsing*, traducción de lenguaje natural, manipulación de imágenes, etc. Ha sido desarrollado en C++ y en Lua, lo que da nombre al toolkit ("April" es acrónimo de "A Pattern Recognizer In Lua"). El motivo de realizar la parte de redes neuronales de dicho toolkit fue, inicialmente, tener una versión de BP más eficiente que otras herramientas populares de la época. Otro motivo nada despreciable fue la mejora en la descripción de los datos de entrenamiento, lo que hace muy sencillo y eficiente realizar entrenamientos utilizando imágenes (para entrenar filtros neuronales convolutivos para la limpieza de imágenes [HECP05, ZEnC07], o para normalizar escritura contínua [ECGZ11]), para realizar modelado de lenguaje [ZCE09], etc.

Con el paso de los años, April ha ido incorporando otros tipos de utilidades para el reconocimiento de formas. Estas utilidades se diseñan y programan en el lenguaje C++, para luego poder ser exportadas al lenguaje Lua, que es un lenguaje similar a Python pero especialmente diseñado para ser pequeño y empotrable dentro de un programa C o C++. De esta forma, podemos describir experimentos en programas o *scripts* realizados en Lua que acceden a la eficiencia de C++. Lua proporcionaría, en este caso, una mayor comodidad para manipular ficheros o utilizar estructuras de datos flexibles con recolección automática de basura y reemplazaría los scripts bash o en awk utilizados por otros toolkits alternativos. Podemos mencionar que esta aproximación ha sido utilizada también por muchos otros toolkits, casualmente el toolkit "Torch" [CBM02] también utiliza Lua como lenguaje de scripting.

1.3. Aportaciones

Tal y como evolucionan las tecnologías, es normal que algunas utilidades se queden anticuadas, como es el caso de la implementación del BP que ha sufrido diversas mejoras en April a lo largo del tiempo. Uno de los objetivo de este proyecto ha sido remodelar y actualizar la implementación del BP con momentum, aprovechando las nuevas tecnologías y los conocimientos e ideas que se han ido acumulando con el paso del tiempo. Otro objetivo ha sido la implementación de una versión para GPUs, y también podemos destacar como aportación la realización de una exhaustiva batería de experimentos que van más allá de la mera validación de la corrección de las implementaciones y que realizan un estudio de los parámetros de entrenamiento en diversas tareas.

Uno de los problemas más comunes al trabajar con redes neuronales es la elección entre una herramienta que permita representar redes generales o una herramienta eficiente para un tipo de redes concreto. También puede que nuestra elección se vea afectada por los factores que deseamos incluir en el entrenamiento, tales como el weight decay, cuya implementación no suele ser tan común como nosotros deseamos.

También es importante, a la hora de experimentar con redes, la adaptación de los conjuntos de datos para la experimentación y el aprendizaje. Aunque no forma parte de la parte central del algoritmo de aprendizaje, implementar o dominar las aplicaciones que se adaptan a nuestras necesidades suponen una inversión de tiempo cuantiosa.

En este proyecto nos hemos planteado la implementación de una nueva herramienta que nos ofrezca la posibilidad de representar redes generales y mantener un alto nivel de eficiencia al mismo tiempo.

Una importante restricción empírica que se impone es que la implementación de los algoritmos se adapte a una forma matricial, para poder beneficiarse de la eficiencia que las librerías de álgebra lineal (tanto para CPUs multicore como para GPUs) nos ofrecen. Esto se consigue mediante la adaptación del modelo clásico de las redes neuronales a un nuevo modo de ejecución: El modo *bunch*. El modo *bunch* empaqueta un número cualquiera de muestras para ser procesadas de forma simulatánea por el algoritmo BP.

Gracias a la jerarquía de clases diseñada, es posible implementar nuevas funciones de activación, funciones de error o cualquier elemento de las redes neuronales sin mucho esfuerzo, así como diseñar topologías mucho más variadas que las simples "capa a capa". Por ejemplo, existen acciones que permiten que distintas partes de la red compartan pesos, o conectar una capa a varias, etc. Los factores "momentum" y "weight decay" han sido incluídos en la implementación de manera exitosa, sin perjudicar la eficiencia de la herramienta.

Con el objetivo de elaborar una implementación elegante y fácil de extender, se formulan los conceptos necesarios para permitir el uso de la GPU de una forma cómoda. Esto resulta en la implementación de los bloques de memoria y los *wrappers* para el álgebra lineal, que automatizan las peticiones de memoria y la llamada a las funciones especializadas para cada dispositivo.

Al final de proyecto, la aplicación no sólo consigue ser eficiente, sino que lo hace con creces. El código de las funciones ha sido adaptado a las arquitecturas paralelas de las GPU, y aquellas funciones que no son tan fáciles de paralelizar han sido rediseñadas mediante algoritmos de reducción u otros mecanismos.

1.4. Estructura de la memoria

En este primer capítulo ya se han presentado los principales problemas a la hora de trabajar con ANN y las ideas que se aportan para tratar de solucionarlos. El capítulo 2 pretende hacer un repaso general de la teoría relacionada con las ANN y el algoritmo de entrenamiento BP. El tercer capítulo presenta las alternativas de diseño de la aplicación y las razones que nos llevan a escoger una de ellas, además de introducir algunas de las librerías o bibliotecas que se utilizan en el proyecto. El capítulo 4 profundiza en los aspectos relacionados con la inclusión de la GPU al diseño ya elaborado en el anterior capítulo.

En este punto finaliza la parte de memoria dedicada a la exposición de los conceptos relacionados con ANN y el diseño del proyecto. El capítulo 5 muestra un ejemplo de uso de la aplicación después de presentar los conceptos relacionados con la herramienta necesarios para comprenderlo. En el siguiente capítulo se detalla el proceso de experimentación con la herramienta, presentando los resultados y comparando su rendimiento con el de otras aplicaciones populares dentro del ámbito de las ANN. En el último capítulo se muestran las conclusiones alcanzadas tanto en la parte de diseño y de implementación como en la parte de experimentación. También se mencionan aquellos trabajos en los cuales se emplea esta aplicación y se ofrece una lista de posibles ampliaciones y desarrollos futuros relacionados con este proyecto.

Capítulo 2

El algoritmo Backpropagation

En este capítulo se pretende exponer el algoritmo Backpropagation descrito de manera muy superficial en la introducción. Para ello, primero será necesario explicar los conceptos básicos sobre redes neuronales artificiales y cómo se relacionan éstas con el aprendizaje automático.

2.1. El problema del aprendizaje

Para estudiar cómo se relacionan las ANN con el aprendizaje, vamos a considerar una red arbitraria como una caja negra, de forma que esta red neuronal dispone de n entradas y m salidas (figura 2.1). La red puede contener cualquier número de unidades ocultas y exhibir cualquier patrón de conexiones entre las unidades. Se nos proporciona un conjunto de entrenamiento $\{(x_1, t_1), \ldots, (x_p, t_p)\}$ formado por p pares ordenados de vectores, de dimensión n y m, llamados patrones de entrada y salida. Sean las funciones primitivas de cada nodo de la red continuas y diferenciables, y los pesos de los arcos números reales que parten de una configuración inicial (normalmente son escogidos aleatoriamente en un rango determinado, aunque existen diversas propuestas de algoritmos de inicialización de los pesos). Cuando se le presenta a la red el patrón de entrada x_i, se produce un salida o_i, que por lo general es diferente de la salida deseada t_i. Nuestro objetivo es hacer que o_i sea "lo más parecida" a t_i para todo $i = 1, \ldots, p$ utilizando un algoritmo de aprendizaje, en donde "parecida" viene dado por una función de error. Concretamente, queremos minimizar el valor de esta función de error de la red aplicada sobre los datos de entrenamiento, de forma que nos aproximemos más a la función objetivo de la red. Existe otro criterio relacionado con evitar el

Figura 2.1: Red neuronal como una caja negra.

$f(\sum_{i=1}^{n} w_i x_i - \theta) = z.$
$f(\sum_{i=0}^{n} w_i x_i) = z,$ incluyendo $-\theta$ como el peso w_0 y haciendo que $x_0 = 1.$

Cuadro 2.1: Cálculo realizado por una neurona.

problema del sobreaprendizaje y que está relacionado con términos de regularización (weight decay) o con el uso de un corpus de validación para poder decidir un criterio de parada.

Existen diversas funciones de error para cuantificar la aproximación de la red a la función objetivo, siendo la más popular de entre ellas la función del error cuadrático medio (abreviado MSE del inglés *Mean Squared Error*). En este proyecto se pone a disposición del usuario el siguiente conjunto de funciones de error que, por las características modulares de la implementación, se podría ampliar en un futuro:

- Error cuadrático medio (MSE).
- Tangente hiperbólica.
- Entropia cruzada.
- Entropia cruzada completa.

En el capítulo del diseño del BP serán comentados los aspectos relacionados con la implementación de estas funciones de error, así como la posibilidad de crear nuevas funciones a partir de una interfaz común que todas han de implementar.

2.2. Estructura de las redes neuronales

Una neurona es una unidad de proceso conectada con una serie de entradas y una única salida. La neurona produce una combinación lineal de las entradas, y propaga en su salida un valor que depende de esta combinación y la aplicación de una función de activación que, por lo general, no es lineal. Esta unidad se denomina perceptrón, y realiza las siguientes operaciones:

1. Un producto escalar del vector de entradas $\vec{x}$ por otro vector, cuyos valores representan los pesos, $\vec{w}$. A este valor se le resta un valor θ, denominado umbral. Al resultado se le denomina potencial: $\gamma = \vec{x} \cdot \vec{w} - \theta$.

2. Al potencial que hemos obtenido anteriormente, se le aplica una función de activación f no lineal, de forma que: $f(\gamma) = z$.

3. El resultado z, conseguido después de aplicar la función f, es propagado por la salida de la neurona.

Para simplificar los cálculos, se añade a la neurona una nueva entrada x_0, conectada a un valor fijo 1, de manera que el peso w_0 realiza la función del umbral θ que se resta en el punto 1.

En este proyecto podemos hacer uso de las siguientes funciones de activación:

- **Lineal**: El valor resultante no cambia, de forma que $f(x) = x$.
- **Sigmoide**: La función más popular de las ANN. El intervalo de valores resultantes se sitúa entre el 0 y el 1. El factor c, empleado en el cálculo de la función, hace que la sigmoide se parezca a la función escalón para valores altos de c (véase la figura 2.3), algunas versiones de BP permiten variar este factor pero, en general, se puede dejar fijo. El cálculo se realiza de la siguiente forma: $s_c(x) = \dfrac{1}{1 + e^{-cx}}$.
- **Tangente hiperbólica**: Esta función tiene la misma forma que la función sigmoide, pero proporciona valores entre $[-1, 1]$. Es posible relacionar ambas funciones de forma que, mediante una transformación lineal, se convierta una en la otra. La función se calcula de esta forma:
  $f(x) = tanh(x) = \dfrac{e^x - e^{-x}}{e^x + e^{-x}}$
  En el caso en que $s_c = s_1$, la transformación se obtiene como: $tanh(x) = 2s_1(2x) - 1$.
- **Softmax**: La ventaja de esta función frente a las otras es que se normaliza la salida de las neuronas de forma que la suma de todas ellas sea igual a 1, lo que suele dar mejores resultados cuando la salida de la red se interpreta como una probabilidad condicionada a los datos de la entrada. Dado un conjunto de n neuronas en la red, con una única salida, la fórmula para el cálculo de la salida k es: $f(x_k) = e^{x_k} / \sum_{k'=1}^{n} e^{x_{k'}}$.

Una ANN se forma gracias a la conexión de un grupo de neuronas. La topología clásica agrupa las neuronas por capas, de forma que las neuronas de una capa están conectadas solamente con neuronas de la anterior y de la siguiente capa. A este tipo de ANN se las conoce como red "capa a capa", y también como Perceptrón Multicapa o "Multilayer Perceptron" (MLP). Pero es posible generalizar más la topología de una red, conectando algunas neuronas con otras neuronas que no estén en la anterior o la siguiente capa, de forma que la red se convierta en un grafo acíclico. A estos tipos de redes se las denomina red hacia-adelante general (véase la figura 2.2).

Una red neuronal se ajusta modificando los valores de los pesos de las conexiones entre estas capas. Para poder modificarlos, pueden utlizarse varias técnicas, siendo las más populares de todas los métodos basados en descenso por gradiente. El gradiente se refiere al del error sobre los datos de entrenamiento respecto al espacio de pesos de la red. Este gradiente se calcula en base a una función de error (derivada), por lo que es necesario que las funciones de activación de las neuronas sean derivables. En el fondo, este algoritmo es un caso particular del clásico descenso por gradiente para localizar el mínimo local de una función con ciertas propiedades matemáticas.

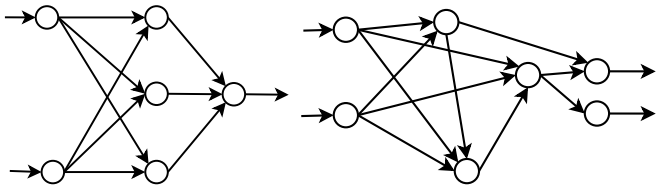

Figura 2.2: Diagrama de redes hacia-delante, una capa a capa y otra general.

2.3. Algoritmos de aprendizaje para ANN

Los algoritmos de aprendizaje para redes neuronales se pueden clasificar básicamente en dos tipos: El primer tipo es el de los métodos basados en el descenso por gradiente y el segundo tipo es el de métodos de segundo orden, los cuales suelen ser variaciones del primer tipo de métodos.

A continuación se presentan ambos tipos y se muestran los algoritmos representativos de éstos.

2.3.1. Métodos basados en el descenso por gradiente

Los métodos más populares de entre los basados en el descenso por gradiente son el algoritmo BP y sus variantes. Este tipo de métodos se basan en definir una función de error y calcular el gradiente respecto a los pesos de la red, los cuales serán modificados según la dirección del gradiente, tratando de encontrar así un mínimo en la función de error.

Las variantes más importantes del algoritmo son:

- BP en modo "batch": El error de la red se calcula como la media de los errores producidos en cada muestra de entrenamiento, por lo que la actualización de los pesos se realiza después de cada época.
- BP en modo "on-line": El error de la red se calcula como el error producido por una muestra individual, actualizando así los pesos después de visitar cada muestra.
- BP con momentum: Se trata de una variación de los anteriores modos en la cual se añade un factor llamdo momentum (representado por μ) a la fórmula. Este factor contribuye a suavizar el entrenamiento al sumar un porcentaje del valor de los pesos en la iteración anterior a los de la iteración actual.

2.3.2. Métodos de segundo orden

En esta sección se presentan algunos de los métodos de segundo orden más populares, que tratan de acelerar la convergencia mediante el uso de nuevas técnicas:

- Algoritmo *Quickprop*: Este algoritmo procede de la misma forma que el algoritmo BP, pero se diferencia en el cálculo del incremento de los pesos.

El algoritmo asume que la función de error tiene un superfície (localmente) parecida a una función cuadrática, la cual trata de aproximar mediante el cálculo de la derivada de la función de error en la iteración anterior y la iteración actual. Una vez hecha la aproximación, se modifican los pesos buscando el mínimo de la función cuadrática, por lo que se produce un descenso mucho más rápido hacia el mínimo de la función de error.

- Algoritmo de *Newton*: Para el estudio de este algoritmo, se requiere un nuevo operador, la *matriz Hessiana*:

$$H = \frac{\partial^2 E}{\partial wji \partial w_{lk}}.$$

Esta matriz está formada por la derivada segunda de la función de error respecto a cada uno de los pesos. Es posible calcular, a partir de la inversa de esta matriz, la dirección exacta del mínimo de la función de error E. El problema de este método es que el cálculo exacto de la matriz inversa de H tiene un coste computacional de $O(W^3)$, en donde W es el número de pesos en la red.

- Algoritmos *quasi-Newton*: Este tipo de algoritmos se basan en el algoritmo de *Newton*, pero el cálculo de la inversa de la matriz H se efectúa de forma aproximada. De este modo, se acelera la velocidad del algoritmo reduciendo el coste computacional, que pasa a ser $O(W^2)$, pero que introduce errores de imprecisión en el cálculo. La parte restante del algoritmo sigue siendo exactamente igual.

2.4. Introducción al algoritmo BP

El algoritmo BP es un algoritmo de aprendizaje supervisado que se usa para entrenar redes neuronales artificiales. El algoritmo trata de encontrar un punto mínimo en una función de error mediante la técnica del descenso por gradiente. Los conceptos que se van a presentar han sido extraídos de los libros [Roj96, Bis96, Rip96].

El algoritmo aplica una función de activación a las neuronas de cada capa después de ser calculadas. Estas funciones de activación necesitan ser derivables, de modo que al realizar paso de actualización de los pesos podemos efectuar el cálculo inverso de las funciones de activación en cada capa.

Existen unas cuantas funciones de activación para el algoritmo BP muy populares. De entre ellas, vamos a escoger la función sigmoide para explicar cómo funciona este algoritmo. La función sigmoide se define como:

$$s_c(x) = \frac{1}{1 + e^{-cx}}$$

El valor c es una constante que ha de tener un valor mayor que 0. Este valor modifica la forma de la función, de manera que cuando $c \to \infty$ la función sigmoide se asemeja a la función escalón en el origen de coordenadas. En la figura 2.3

podemos observar como cambia la forma de la función sigmoide en función del valor elegido para c.

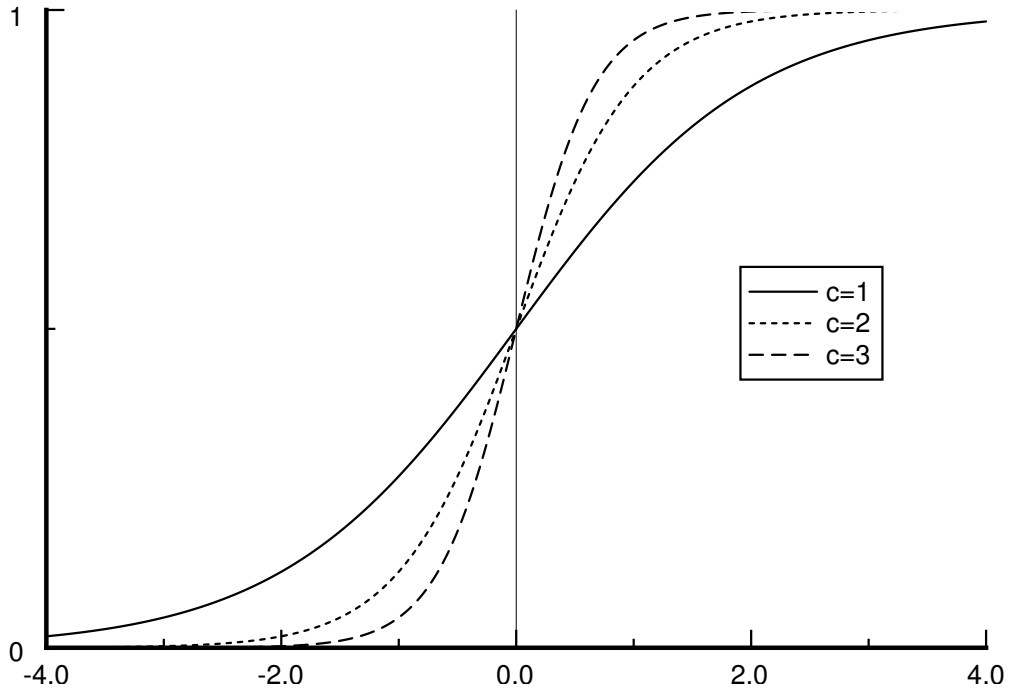

Figura 2.3: Tres sigmoides (con c=1, c=2 y c=3).

Con el objetivo de simplificar las expresiones, a partir de ahora asumiremos que $c = 1$, de manera que $s_1(x)$ pasa a ser $s(x)$. El hecho de realizar esta simplificación no supondrá ningún cambio en los resultados. La función $s(x)$ quedará como:

$$s_1(x) = \frac{1}{1 + e^{-x}}$$

La derivada de esta función respecto a x será la siguiente:

$$\frac{d}{dx}s(x) = \frac{e^{-x}}{(1 + e^{-x})^2} = s(x)(1 - s(x))$$

2.4.1. El problema del aprendizaje

Las redes hacia-adelante pueden verse como un grafo computacional cuyos nodos son unidades de cómputo y cuyas aristas dirigidas transmiten información numérica de un nodo a otro. Cada unidad de cómputo es capaz de calcular una función por lo general no lineal de sus entradas, de manera que la red calcula una composición de funciones que transforma un espacio de entrada en un espacio de salida. La función que realiza esta transformación es llamada función de la red. El problema del aprendizaje consiste en encontrar la combinación óptima de pesos de forma que la función de la red se aproxime lo más posible a una cierta función

f. No obstante, esta función f es desconocida para nosotros, pero disponemos de ejemplos que la definen implícitamente. Por tanto, como ya se ha explicado de forma general en el primer apartado de este capítulo, necesitamos ajustar los parámetros de la red (pesos) para minimizar un función de error determinada, que nos da una medida de lo buena que es la aproximación a la función f.

Después de minimizar la función de error para el conjunto de entrenamiento, al presentar nuevos patrones de entrada a la red, desconocidos hasta el momento, se espera que la red interpole y acierte con la salida. La red debe aprender a reconocer si los nuevos patrones de entrada son similares a los otros patrones ya aprendidos y producir una salida similar para ellos.

Ahora vamos a extender la red para que cada vez que llegue un nuevo patrón, se calcule la función de error E de forma automática para el mismo. Para ello, conectamos cada unidad de salida a un nuevo nodo que evalúa la función $\frac{1}{2}(o_{ij} - t_{ij})^2$, en donde o_{ij} y t_{ij} representan la componente j del vector de salidas o_i y del objetivo t_i, respectivamente. La salida de estas m unidades es recolectada por un nodo final que suma todas las entradas recibidas, de manera que la salida de este nodo es la función E_i. La misma extensión de la red ha de ser construida para cada patrón t_i. La salida de la red extendida es la función de error E.

Ahora tenemos una red capaz de calcular el error total para un conjunto de entrenamiento. Los pesos de la red son los únicos parámetros modificables para minimizar la función de error E. Dado que E es calculada a partir de la composición de funciones de los nodos, E será una función continua y derivable de los l pesos $w_1, w_2, \ldots, w_l$ que componen la red. Podemos entonces minimizar E utilizando un proceso iterativo de descenso por gradiente. El gradiente se define como:

$$\nabla E = (\frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, \ldots, \frac{\partial E}{\partial w_l}).$$

Cada peso es actualizado por el incremento:

$$\triangle w_i = -\eta \frac{\partial E}{\partial w_i}, \text{ para } i = 1, .., l.$$

En donde η representa el factor de aprendizaje, un parámetro de proporcionalidad que define la longitud del paso de cada iteración en la dirección del gradiente.

Con esta extensión de la red original, el problema del aprendizaje se reduce a calcular el gradiente de la función de la red respecto a los pesos que la componen. Una vez tengamos un método para el cálculo de este gradiente, podemos ajustar los pesos de forma iterativa, esperando encontrar un mínimo local de la función de error E en la cual $\nabla E = 0$.

2.4.2. Derivada de la función de la red

Vamos a dejar de lado todo lo visto hasta el momento, para centrarnos en encontrar la derivada de la función de la red extendida respecto a cada uno de los pesos, o dicho de otro modo, como obtener ∇E.

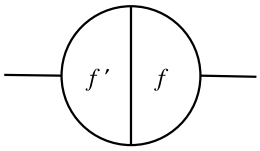

Figura 2.4: Los dos lados de una unidad de proceso.

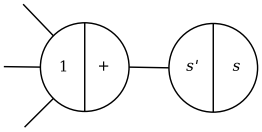

Figura 2.5: Sumatorio y función de activación separadas en dos unidades.

La red puede realizar dos pasos: El paso hacia-delante y el paso de retropropagación. En el paso hacia-delante, las unidades calculan una función de sus entradas y propagan el resultado en la salida. Al mismo tiempo, la derivada de la función es calculada para la misma entrada, y el resultado se guarda para su posterior uso. Podemos ver las unidades como separadas en dos partes, una con el resultado obtenido al aplicar la función y otra con el resultado de la derivada, como se ilustra en la figura 2.4. En el paso de retropropagación, las unidades utilizarán este último resultado para calcular la derivada de la función de la red.

Ahora separaremos la unidad en dos partes independientes (figura 2.5). La primera es la que calcula, para una unidad k, el sumatorio de todas sus entradas, obteniendo $\gamma_k = \sum_i w_{ki} \cdot x_{ki}$. La segunda calculará la función primitiva de la misma unidad, obteniendo: $f(\gamma_k) = z_k$.

De esta forma, tenemos tres casos en los que estudiar cómo calcular la derivada:

Primer caso: Composición de funciones

La composición de funciones se consigue mediante la unión de la salida de una unidad con la entrada de otra. La función que calcula el primer nodo es g, y la que calcula el segundo nodo es f, de forma que se realiza la composición $f(g(x))$. El resultado de esta composición es el obtenido por la salida de la última

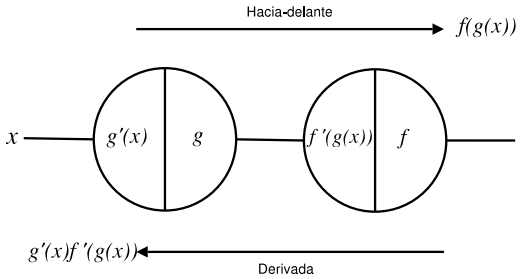

Figura 2.6: Composición de funciones.

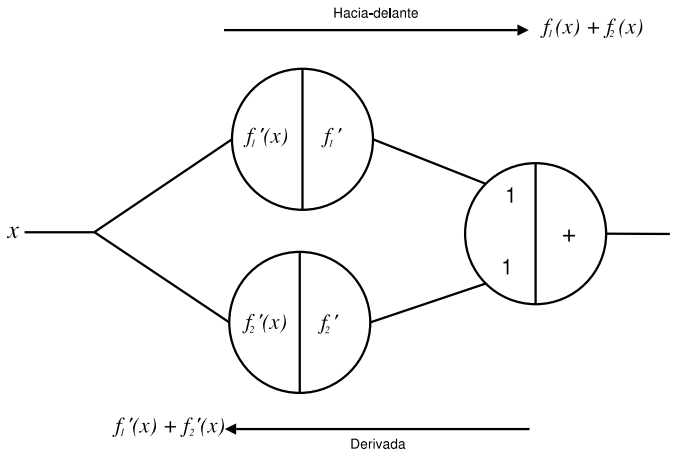

Figura 2.7: Adición de funciones.

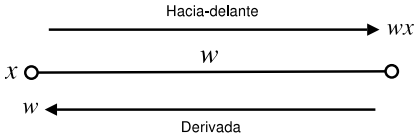

Figura 2.8: Pesos en las aristas.

unidad. De forma paralela, cada unidad guarda el valor de la derivada en su lado izquierdo. En el paso de retropropagación entra un 1 por la parte derecha de la última unidad, que es multiplicado por el valor guardado en el lado izquierdo de cada unidad. El resultado obtenido es transmitido hacia la siguiente unidad por la izquierda. Por tanto, el cálculo que la red está efectuando es $f'(g(x))g'(x)$, que es la derivada de $f(g(x))$. Cualquier secuencia de composición de funciones puede ser evaluada de esta manera, y su derivada puede ser obtenida el paso de retropropagación.

Segundo caso: Adición de funciones

En este caso, tenemos una serie de nodos en paralelo conectados a un nodo de salida que realiza la suma de todas sus entradas. Para dos nodos de entrada, que calculan las funciones f_1 y f_2, la suma calculada por el nodo de salida en el paso hacia-delante será $f_1(x) + f_2(x)$. La derivada parcial de la función de adición respecto a cualquiera de sus entradas es 1. En el paso de retropropagación entra un 1 por la parte derecha de la unidad encargada de realizar la suma, por lo que el resultado final de la red será $f_1'(x) + f_2'(x)$, que es la derivada de la función evaluada en el paso hacia-delante. Se puede demostrar, mediante inducción, que la derivada de la adición de cualquier número de funciones puede ser tratada del mismo modo.

Tercer caso: Pesos en las aristas

Las aristas pueden tener un peso asociado que podría ser tratado de la misma forma que un caso de composición de funciones, pero hay una forma diferente de tratar con ellas que resulta ser más fácil. En el paso hacia-delante, la información entrante x es multiplicada por el peso w, de forma que el resultado es wx. En el paso de retropropagación, el valor 1 es multiplicado por el peso de la arista. El resultado es w, que es la derivada de wx respecto a x.

2.4.3. Demostración formal del algoritmo BP

Ahora ya podemos formular el algoritmo BP y probar por inducción que funciona en una red hacia-delante arbitraria con funciones de activación en los nodos. Vamos a asumir que estamos tratando con una red que únicamente tiene una unidad de entrada y otra de salida.

Consideraremos una red con una única entrada real x y la función de red F. La derivada $F'(x)$ es calculada en dos fases:

- Hacia-delante: La entrada x se introduce en las entradas de la red. Las funciones de activación y sus derivadas son evaluadas en cada nodo. La derivada es guardada.
- Retropropagación: La constante 1 es introducida por la unidad de salida y la red se ejecuta hacia atrás. La información pasa de nodo a nodo y es multiplicada por el valor guardado en la parte izquierda de cada nodo. El resultado recogido en la unidad de entrada es la derivada de la función de la red respecto a x.

Para demostrar que el algoritmo es correcto aplicaremos un razonamiento inductivo. Supongamos que el algoritmo BP funciona para cualquier red hacia-delante con n nodos. Ahora podemos considerar una red hacia-delante con $n+1$ nodos. El paso hacia-delante es ejecutado en primera instancia y el resultado obtenido en la única unidad de salida es la función de la red F evaluada con x. Las m unidades conectadas a esa unidad de salida producen $F_1(x), \ldots, F_m(x)$ por su salida, y teniendo en cuenta que la función de activación de la unidad de salida es φ, tenemos que:

$$F'(x) = \varphi(w_1 F_1(x) + w_2 F_2(x) + \cdots + w_m F_m(x))$$

Por lo que la derivada de F con x es:

$$F'(x) = \varphi'(s)(w_1 F_1'(x) + w_2 F_2'(x) + \cdots + w_m F_m'(x))$$

En donde s representa $F(x)$. Si introducimos un 1 por el nodo salida de la red y ejecutamos el paso de retropropagación, podremos calcular la derivada de las funciones $F_1(x), \ldots, F_m(x)$. Si en lugar de introducir un 1, introducimos la constante $\varphi'(s)$ y la multiplicamos por los pesos asociados a los nodos, entonces obtendremos $w_1 F_1'(x)\varphi'(s)$ para el nodo que calcula $F_1(x)$ en la unidad de entrada, y lo mismo para los otros nodos hasta el nodo m. El resultado de ejecutar el algoritmo de retropropagación será, por tanto:

$$\varphi'(s)(w_1 F_1'(x) + w_2 F_2'(x) + \cdots + w_m F_m'(x))$$

Que es la derivada de F evaluada con x. Nótese que la introducción de las constantes $w_1\varphi'(s), \ldots, w_m\varphi'(s)$ en las m unidades conectadas la unidad de salida puede ser realizada introduciendo un 1 en la unidad de salida, multiplicándolo por el valor almacenado $\varphi'(s)$ y distribuyendo el resultado a las m unidades conectadas a través de las aristas con pesos $w_1, \ldots, w_m$. De hecho, estamos ejecutando la red hacia atrás tal y como el algoritmo BP exige, con lo cual se prueba que el algoritmo funciona para $n + 1$ nodos y la demostración concluye.

2.4.4. Aprendiendo con el BP

Recuperemos ahora el problema del aprendizaje en las redes neuronales. Queremos minimizar una función de error E, la cual depende de los pesos de la red. El paso hacia-delante es efectuado de la misma forma en que se ha hecho hasta el momento, pero ahora vamos a guardar, además del valor de la derivada en el lado izquierdo de cada unidad, el valor de la salida en el lado derecho. El paso de retropropagación será ejecutado en la red extendida que calcula la función de error y fijaremos nuestra atención en uno de los pesos, que denominaremos w_{ij}, cuyas arista relaciona la unidad i con la unidad j. En el primer paso calcularemos $x_j = \sum o_i w_{ij}$, de manera que x_j será el valor que llegue a la unidad j. También tendremos que $o_j = f_j(x_j)$, siendo f_j la función de activación que se aplica en el nodo j. La unidad i guardará en su salida el valor o_i. Las derivadas parciales de E se calculan como:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial x_j}\frac{\partial x_j}{\partial w_{ij}} = o_i \frac{\partial E}{\partial x_j} = o_i f_j'(x_j)\frac{\partial E}{\partial o_j} = o_i \delta_j, \text{ con } \delta_j = f_j'(x_j)\frac{\partial E}{\partial o_j}.$$

La primera igualdad es una consecuencia de la dependencia entre E y los pesos con los que se relaciona en las salidas. La segunda se obtiene a partir de $x_j = \sum o_i w_{ij}$. Denominaremos a δ_j el *error retropropagado* para el nodo j. Por tanto, la información que cada nodo ha de guardar para poder realizar estos cálculos son:

- La salida o_i del nodo en el paso hacia-delante.
- El resultado acumulativo tras ejecutar el paso de retropropagación en ese nodo, δ_i.

Para concluir el cálculo es necesario definir cúal va a ser el valor de δ_j para cada unidad de la red. Podemos observar que $\delta_j = f_j'(x_j)\dfrac{\partial E}{\partial o_j} = \dfrac{\partial E}{\partial x_j}$. De esta expresión, podemos calcular $f_j'(x_j)$ para cualquier unidad, ya que el valor x_j es conocido para todas las unidades durante el paso de retropropagación. Despejaremos cúal debe ser el valor del otro factor de la expresión, distinguiendo entre dos casos. En las unidades de salida de la red, $\dfrac{\partial E}{\partial o_j}$ se calcula de forma directa. Para las unidades de las capas ocultas, teniendo en cuenta la unidad j y las K que le suceden ($j \to K$):

$$\frac{\partial E}{\partial o_j} = \sum_{k:j\to K} w_{jk}\frac{\partial E}{\partial x_k} = \sum_{k:j\to K} w_{jk}\delta_k.$$

Una vez todas las derivadas parciales han sido calculadas, podemos añadir a cada peso w_{ij} el incremento:

$$\triangle w_{ij} = -\eta o_i \delta_j$$

Con todo esto, ya tenemos definido el funcionamiento del algoritmo para redes neuronales generales, las cuales son capaces de describir cualquier topología hacia-delante.

A continuación se ofrece una traza paso a paso del algoritmo completo para una muestra de entrenamiento del problema consistente en aprender la función lógica "o exclusiva" o *xor*. El problema *xor* es un problema clásico en el cual se entrena una red para aprender la función mostrada en el cuadro 2.2.

i_1	i_2	t_1
0	0	0
0	1	1
1	0	1
1	1	0

Cuadro 2.2: Función a aprender en el problema de la *xor*.

En la siguiente traza se muestra la ejecución del algoritmo BP para la muestra (0, 1).

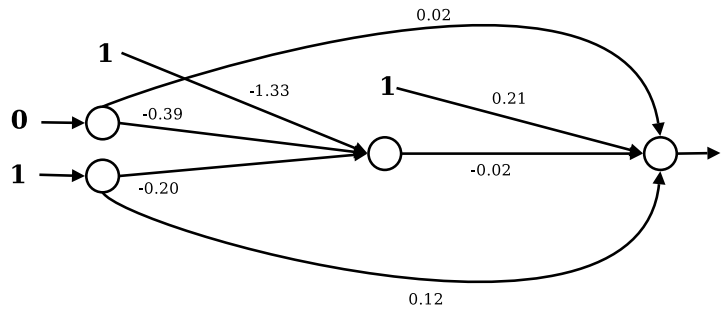

Figura 2.9: Traza del BP, para resolver el problema de la *xor*. Configuración inicial de la red.

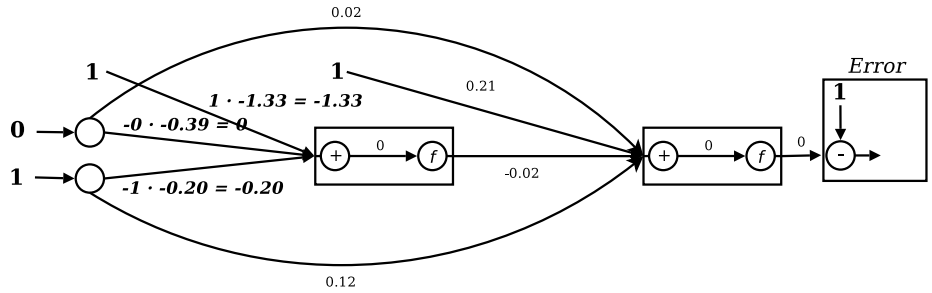

Figura 2.10: Traza del BP, para resolver el problema de la *xor*. Paso hacia-delante, 1/4.

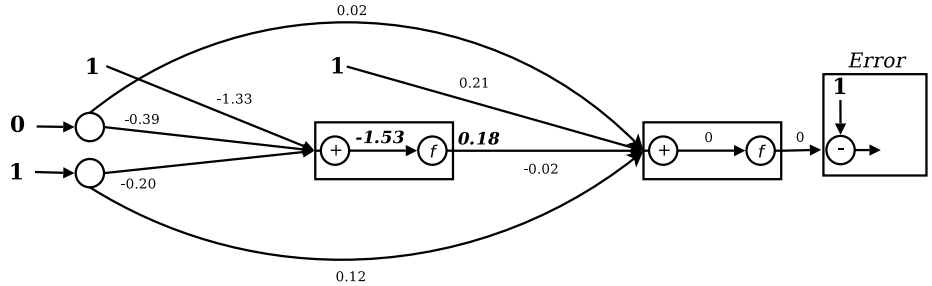

Figura 2.11: Traza del BP, para resolver el problema de la *xor*. Paso hacia-delante, 2/4.

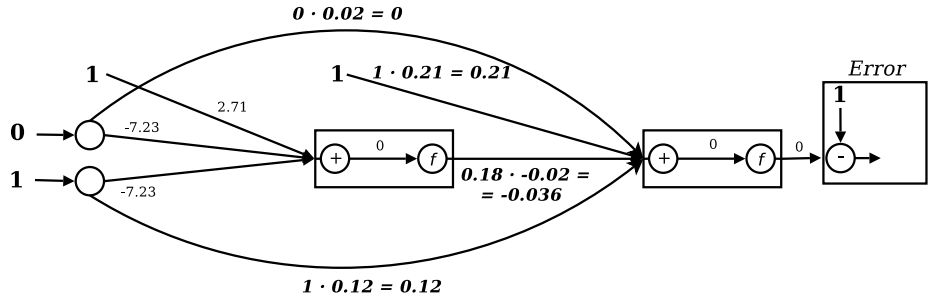

Figura 2.12: Traza del BP, para resolver el problema de la *xor*. Paso hacia-delante, 3/4.

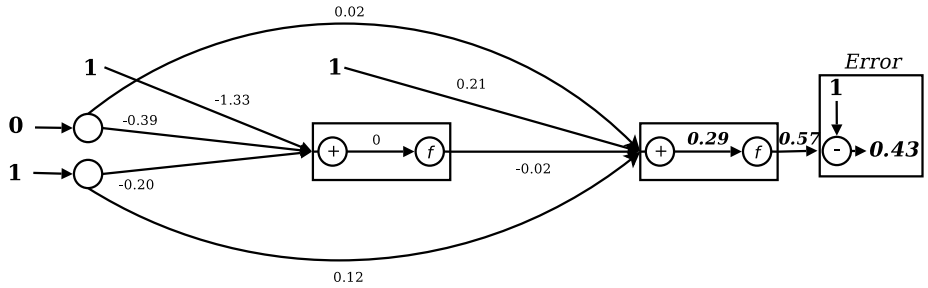

Figura 2.13: Traza del BP, para resolver el problema de la *xor*. Paso hacia-delante, 4/4.

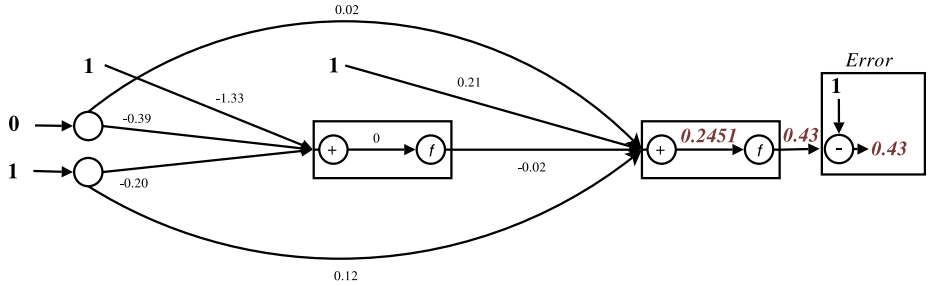

Figura 2.14: Traza del BP, para resolver el problema de la *xor*. Paso de retropropagación, 1/6 (Cálculo de la derivada de la neurona de salida).

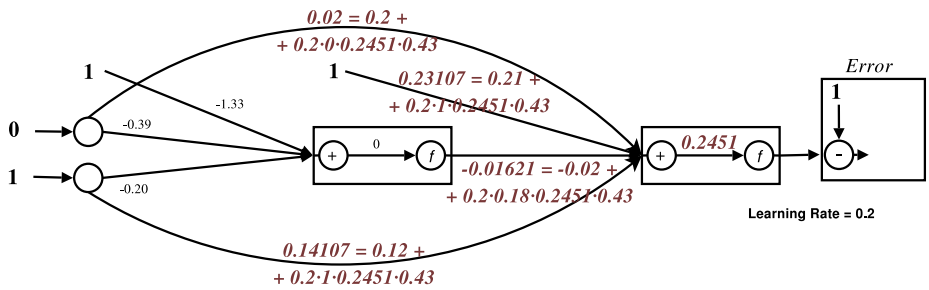

Figura 2.15: Traza del BP, para resolver el problema de la *xor*. Paso de retropropagación, 2/6 (Ajuste de los pesos conectados a la neurona de salida).

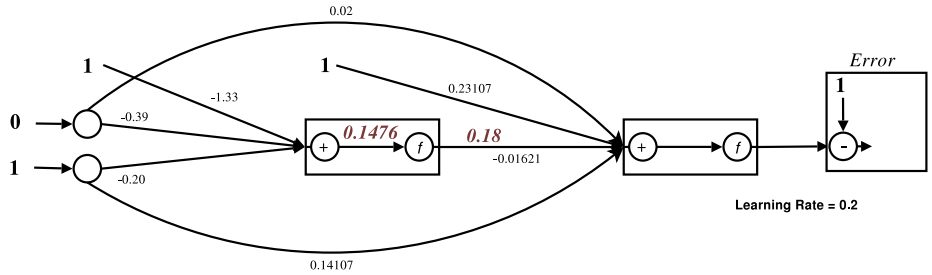

Figura 2.16: Traza del BP, para resolver el problema de la *xor*. Paso de retropropagación, 3/6 (Cálculo de la derivada de la neurona oculta).

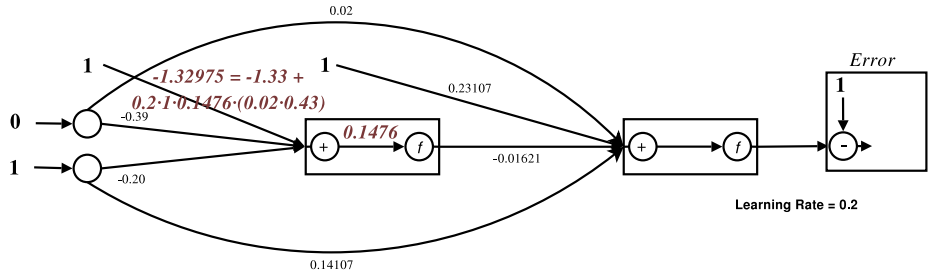

Figura 2.17: Traza del BP, para resolver el problema de la *xor*. Paso de retropropagación, 4/6 (Ajuste del peso umbral de la neurona oculta).

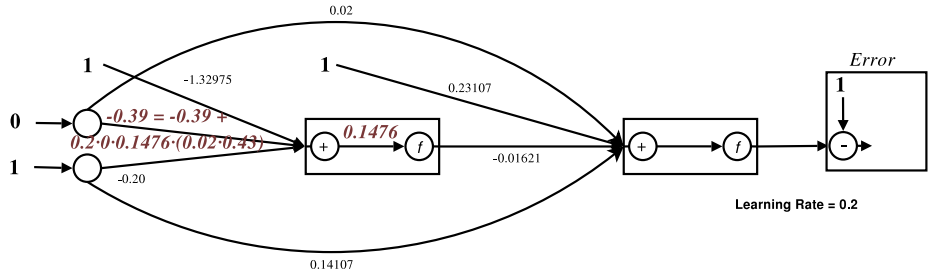

Figura 2.18: Traza del BP, para resolver el problema de la *xor*. Paso de retropropagación, 5/6 (Ajuste del primer peso de la neurona oculta).

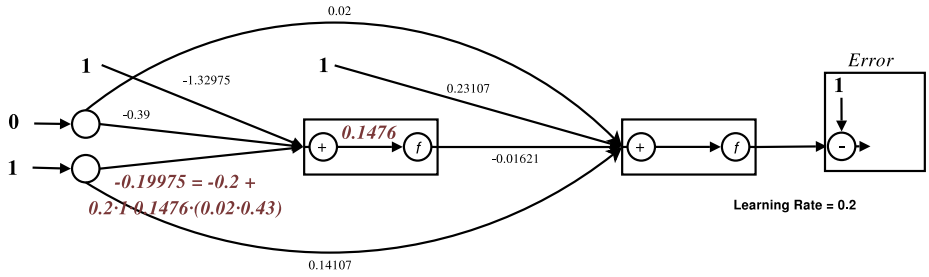

Figura 2.19: Traza del BP, para resolver el problema de la *xor*. Paso de retropropagación, 6/6 (Ajuste del segundo peso de la neurona oculta).

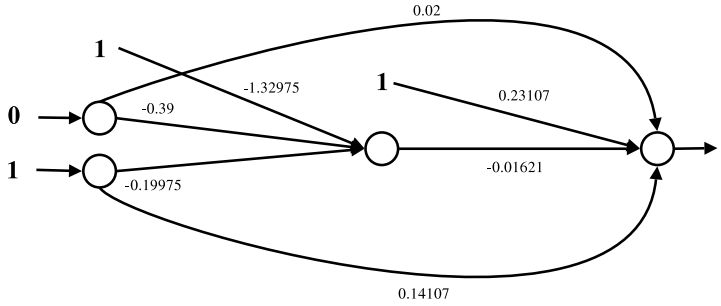

Figura 2.20: Traza del BP, para resolver el problema de la *xor*. Configuración final de la red.

La iteración del algoritmo BP con la muestra (0, 1) finaliza aquí. En este punto, se supone que la red proporcionará salidas más cercanas a la salida deseada para la muestra (0, 1). Por lo general, será necesario un mayor número de presentaciones de muestras con tal de obtener una mejor aproximación a las salidas deseadas.

2.5. Forma matricial del BP

El enfoque empleado a lo largo de este capítulo con grafos computacionales para la prueba del algoritmo BP puede sernos útil para el caso de las redes con una topología general. No obstante, en otro tipo de redes como las redes capa a capa, puede resultar de interés utilizar diferentes enfoques, como es el caso de la forma matricial del BP. Esta forma matricial nos permite organizar los elementos de la red para sacar un mayor provecho de los procesadores vectoriales u otras máquinas especializadas en el álgebra lineal.

Para mostrar su funcionamiento, se empleará una red con una capa de entrada, una capa oculta y otra de salida (con n, k y m unidades). La capa de entrada se representa con un vector o de dimensiones $1 \times n$, y las capas siguientes se denotan con o^i, en donde i es la posición que toman las capas después de la capa de entrada. Las matrices de pesos son representadas con W_i, siendo W_1 la matriz de conexiones entre la capa o y la capa o^1, cuyas dimensiones son $n \times k$, lógicamente.

El paso hacia-delante producirá $o^2 = s(o^1 W_2)$ en el vector de salidas, en donde $o^1 = s(oW_1)$. Las derivadas almacenadas en el paso hacia-delante por las k unidades ocultas y las m unidades de salida pueden ser escritas como la matriz diagonal:

$$D_2 = \begin{pmatrix} o_1^2(1-o_1^2) & 0 & \cdots & 0 \\ 0 & o_2^2(1-o_2^2) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & o_m^2(1-o_m^2) \end{pmatrix}$$

La misma matriz se puede confeccionar para las derivadas entre la capa de entrada y la capa oculta:

$$D_1 = \begin{pmatrix} o_1^1(1-o_1^1) & 0 & \cdots & 0 \\ 0 & o_2^1(1-o_2^1) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & o_k^1(1-o_k^1) \end{pmatrix}$$

El vector e, que guarda las derivadas de las diferencias entre las salidas obtenidas y las salidas deseadas, se define como:

$$e = \begin{pmatrix} (o_1^2 - t_1) \\ (o_2^2 - t_2) \\ \vdots \\ (o_m^2 - t_m) \end{pmatrix}$$

El vector m-dimensional δ^2, que representa el *error retropropagado* por las unidades de salida, se obtiene mediante la expresión:

$$\delta^2 = D_2 e$$

El vector k-dimensional δ^1, que representa el *error retropropagado* por las unidades ocultas, se obtiene mediante la expresión:

$$\delta^1 = D_1 W_2 \delta^2$$

Los incrementos de las matrices W_1 y W_2 vienen dados por:

$$\triangle W_2^T = -\eta \delta^2 o^1$$

$$\triangle W_1^T = -\eta \delta^1 o$$

Es fácil generalizar estas ecuaciones para el caso de una red capa a capa con l capas. La expresión para la obtención de los *errores retropropagados* se puede definir de forma recursiva:

$$\delta^i = D_i W_{i+1} \delta^{i+1}, \text{ para } i = 1, .., l-1.$$

O podemos desplegar la expresión para obtener:

$$\delta^i = D_i W_{i+1} \cdots W_{l-1} D_{l-1} W_l D_l e$$

Los incrementos de las matrices de pesos se calculan de la misma forma para el caso con l capas. Este punto de vista resultará ser esencial para la elaboración de este proyecto, y será ampliado más adelante para permitir el cálculo de varias entradas de forma simultánea, consiguiendo así una aceleración notable al utilizar librerías especializadas en el cálculo de productos de matrices.

2.6. El algoritmo BP con momentum

Ya se ha mencionado que una de las variantes del algoritmo BP es el algoritmo BP con momentum.

La introducción del factor momentum pretende suavizar el ajuste de los parámetros, evitando así algunos problemas del algoritmo clásico. El momentum provoca un efecto de inercia en el entrenamiento de la red, evitando cambios bruscos al realizar la actualización de los pesos.

El cálculo de los pesos en el algoritmo clásico se realizaba conforme a la siguiente fórmula:

$$\triangle w_{ij} = -\eta o_i \delta_j.$$

A esta fórmula pretendemos añadirle el momentum, que será representado por μ a partir de ahora. Como el momentum se define como un factor del incremento de los pesos, es necesario introducir dos valores para cada peso: w_{ij}^k representa el valor del peso en la posición ij para la iteración actual, mientras que $w_i^{k-1}j$ representa el valor de ese mismo peso en la iteración previa. La diferencia entre estos valores estará determinando el incremento del peso, por lo que la fórmula queda como sigue:

$$\triangle w_{ij}^k = -\eta o_i \delta_j - \mu(w_{ij}^k - w_{ij}^{k-1})$$

El valor μ debería tener valores entre $[0, 1]$, dado que se define como un porcentaje. Resulta obvio que si $\mu = 0$, nos encontramos de nuevo con el algoritmo BP clásico.

El uso de esta técnica tiene sus ventajas y desventajas. La principal ventaja es que se logra suavizar el entrenamiento de la red. No obstante, la inclusión de los valores necesarios para actualizar los pesos en la red exige que se guarden los pesos de la iteración anterior, por lo que se requiere una porción mayor de memoria a la hora de realizar el entrenamiento.

Otra ventaja que nos proporciona el momentum es que puede propiciar el descubrimiento de nuevas zonas en la función de error a partir de una misma inicialización, pudiendo encontrar así nuevos locales mínimos mejores que los anteriores.

2.7. El algoritmo BP con weight decay

Los algoritmos de poda se utilizan para ajustar las arquitecturas de las redes neuronales a los problemas que se tratan de resolver. Estos algoritmos tratan de detectar qué partes de la red no contribuyen al cálculo del BP, y se dividen principalmente en dos tipos: Los algoritmos de poda de unidades y los algoritmos de poda de pesos. En esta sección se presenta el BP con weight decay [MHL95], uno de los algoritmos de poda de pesos más populares, que cae dentro del grupo de algoritmos con factor de penalización.

En este algoritmo se introduce el término weight decay (representado por α) en la fórmula para la actualización de los pesos del algoritmo BP clásico:

$$\triangle w_{ij}^k = -\eta o_i \delta_j - \alpha w_{ij}^{k-1}$$

El término completo que se ha añadido a la fórmula representa la resta de una porción del peso en la anterior iteración al peso en la iteración actual. Concretamente, el weight decay indica qué porcentaje del peso en la anterior iteración se restaría al peso en la iteración actual.

Al restar este término en la actualización de los pesos, aquellos pesos que no sean reforzados de forma significativa por los incrementos efectuados en el paso de retropropagación se aproximarán más rápidamente hacia el valor 0, produciendo así un efecto de poda en el peso.

De la misma forma que ocurría con el momentum, hemos de ser capaces de recuperar el valor que el peso tenía en la anterior iteración para poder realizar este cálculo, por lo que tendremos que guardar los pesos de la iteración anterior cada vez que queremos actualizar los pesos. Si se pretende aplicar el algoritmo BP con momentum para el entrenamiento de una red, no es una mala idea usar también la técnica del weight decay, ya que de todos modos hemos de guardar los pesos en la iteración anterior.

2.8. Complejidad algorítmica del BP

El entrenamiento se realiza en dos pasos: El paso hacia-delante y el paso de retropropagación. El primer paso tiene un coste de $O(W)$, siendo W el número de pesos de la red. El paso de retropropagación calcula primero las derivadas de las neuronas, lo cual tiene un coste de $O(N)$, siendo N el número de neuronas en la red. A continuación, se actualizan los pesos según el valor obtenido en la fórmula del BP. Por tanto, el coste final del algoritmo será de $O(2(W + N))$, pero como $N \leq M$, nos queda un coste lineal con el número de conexiones en la red, $O(W)$.

Este coste no es reducible ya que es necesario visitar todos los pesos de la red para modificarlos. No obstante, es posible acelerar la ejecución del algoritmo en implementaciones especializadas para una determinada topología. Esto es lo que se pretende hacer en este proyecto mediante el uso de arquitecturas Single Instruction Multiple Data (SIMD) que las Unidad Central de Procesamiento o "Central Processing Unit" (CPU) y GPU actuales implementan. Gracias a estas arquitecturas, seremos capaces de paralelizar procesos como la aplicación de las

funciones de activación sobre las neuronas de las capas o la modificación de pesos en el paso de retropropagación.

2.9. Inicialización de pesos

El valor inicial de los pesos es un factor clave en la obtención de un mínimo local de la función de error de la red. Por esto mismo, la capacidad de poder reproducir un mismo experimento con una misma inicialización resulta bastante importante. Para lograr esta reproducibilidad, hemos utilizado un generador de números pseudo-aleatorios que pueda guardar y recuperar su estado. El generador que se utiliza en este proyecto es el *Mersenne Twister* [MN98]. Este generador nos permite la reproducción exacta de experimentos a partir de una misma semilla, como también nos permite su reanudación a partir del estado en el que se encontraba cuando fue detenido.

2.10. Sobreentrenamiento

El sobreentrenamiento (también conocido como *overfitting*) es uno de los problemas que presentan las tećnicas del reconocimiento de formas. El sobreentrenamiento se produce cuando los modelos para el reconocimiento se ajustan demasiado al conjunto de muestras con el cual se realiza el entrenamiento, de forma que el modelo no consigue generalizar las clases lo suficiente y falla en la clasificación durante la fase de test cuando le presentamos muestras que no estaban en el conjunto de entrenamiento.

Por este motivo, los conjuntos de muestras suelen ser separados en tres subconjuntos de muestras, cada uno de los cuales tiene un objetivo:

- Conjunto de entrenamiento: Es el subconjunto de muestras con el cual se pretende entrenar la red.
- Conjunto de validación: Es el subconjunto de muestras con el cual se pretende ajustar los parámetros de entrenamiento de la red. Este subconjunto no es usado por el algoritmo de entrenamiento, por lo que los resultados de clasificación son más fiables. A veces se pueden considerar 2 conjuntos de validación (si hay datos suficientes): uno para elegir el criterio de parada para una red concreta y otro para elegir la mejor red (en caso de probar diversas topologías o configuraciones) antes de realizar la medición del test.
- Conjunto de test: Es el subconjunto de muestras con el cual se realiza la clasificación con la red ya entrenada para obtener el error final de la red y se utilizaría una única vez cuando se ha decidido la mejor red (topología y pesos).

Existen variantes de esta idea que realizan diversas particiones de la parte de entrenamiento y validación (conocidos como “leaving one out”, etc.) cuando el tamaño de estos conjuntos no es suficientemente grande. La idea consiste en realizar diversas particiones, entrenar con ellas y promediar los resultados.

Para realizar un entrenamiento, es recomendable fijar un criterio de parada relacionado con el conjunto de validación, consiguiendo así evitar el sobreentrenamiento en el caso de que las muestras de los conjuntos para el entrenamiento y la validación tengan la suficiente variabilidad. El entrenamiento suele detenerse cuando el error en validación no ha mejorado durante un número determinado de épocas (presentaciones de las muestras de entrenamiento). Al conseguir un buen resultado de clasificación respecto al conjunto de validación es más probable que los resultados de clasificación para el conjunto de test sean buenos.

Otras técnicas como el "shuffle", que consiste en escoger la muestra de entrenamiento de forma aleatoria, o el reemplazamiento, que permite a una muestra ser escogida aleatoriamente repetidas veces, también suelen mejorar la calidad del entrenamiento.

Capítulo 3

Diseño del BP

A continuación se van a exponer todas las decisiones tomadas a la hora de realizar el diseño para la implementación del BP. De este modo podremos comprender mejor cómo deberían funcionar los cálculos, los problemas que surgen debido a estas decisiones de diseño y las soluciones que se proponen para resolverlos.

3.1. Consideraciones previas

Las ANN son un modelo para el aprendizaje cuyo entrenamiento puede llegar a ser muy costoso temporalmente: Las redes complejas pueden tardar semanas o meses en entrenarse antes de poder ser usadas para el reconocimiento. Es por esto por lo que se pretende crear una implementación que sea eficiente.

Para poder acelerar la ejecución de las operaciones que el BP desarrolla a lo largo de su ejecución, el uso de las librerías de álgebra lineal en combinación con el uso de las GPU resulta fundamental.

La Interfaz de Programación de Aplicaciones o "Application Programming Interface" (API) estándar de Basic Linear Algebra Subprograms (BLAS) permite al usuario realizar la llamada de funciones para ejecutar cálculos de álgebra lineal optimizadas para las CPU más comunes mediante el uso de una librería que las implemente, no siendo necesario el uso de otros componentes especializados como la GPU. Además, algunos fabricantes de CPU también nos dan la posibilidad de emplear sus propias librerías de cálculo y compiladores para optimizar todavía más la ejecución de los programas en sus arquitecturas.

Los fabricantes de tarjetas gráficas también ponen a nuestra disposición librerías y compiladores para el cálculo de operaciones de álgebra lineal mediante el uso de las GPU, aunque la API puede llegar a diferenciarse bastante debido a que las peculiaridades del diseño de las mismas no hacen sencillo abstraer todos los detalles cuando se programan, por ello se requiere un esfuerzo adicional para la programación de las mismas, como veremos más adelante.

Con el objetivo de poder hacer uso de estas tecnologías hemos adoptado un modo de ejecución de BP basado en operaciones vectoriales y matriciales [SCG+10]. Este es el llamado modo *bunch*, que procesa las muestras por lotes, consiguiendo así una implementación parecida a la descrita en la sección en donde se explica la forma matricial del BP, en el capítulo 2.

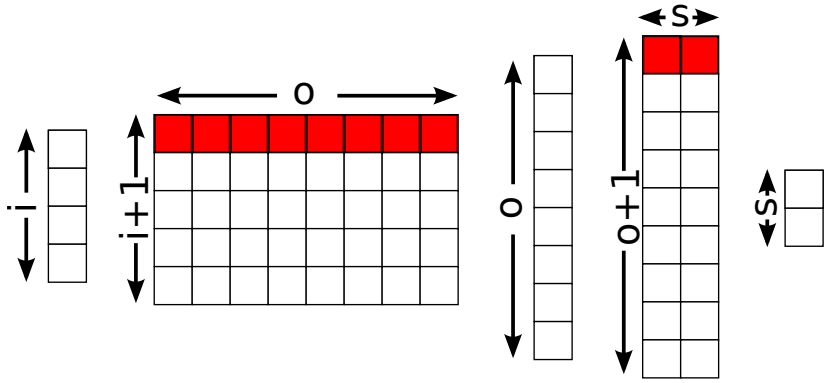

Figura 3.1: Estructura de la red para ejecutar el algoritmo en modo "on-line". Cada iteración procesa una muestra.

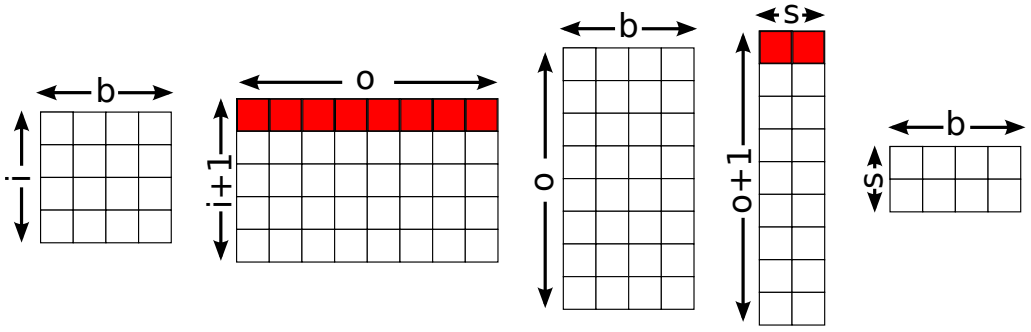

Figura 3.2: Estructura de la red para ejecutar el algoritmo en modo "bunch". Cada iteración procesa b muestras.

En la figura 3.1 se muestra a qué nos referimos. El modo *on-line* del algoritmo BP procesa una muestra con i entradas. La única capa oculta de la red tiene o neuronas, y la capa de salida se compone de s neuronas. Las conexiones entre estas capas se forman con $(i+1) \times o$ y $(o+1) \times s$ pesos (los pesos correspondientes al umbral θ están marcados en rojo). Al agrupar b muestras (figura 3.2), conseguimos una estructura matricial (las activaciones de cada capa, puesto que manipulamos varios patrones en paralelo, pasan de considerarse vectores a matrices). Esto nos permite emplear operaciones cuyos operandos son matrices, de forma que la productividad de la herramienta se ve aumentada.

3.2. Estructuras de datos

Antes de empezar a tratar con los aspectos más cercanos a la máquina y su programación, es necesario realizar un diseño que, en nuestra implementación en C++, se corresponde con una jerarquia de clases. Una de las particularidades de este diseño, heredado del diseño original de la herramienta April, es que permite describir redes con unas topologías muy flexibles mediante el uso de una serie de objetos entre los que destacamos agrupaciones de neuronas (para formar capas, por ejemplo), matrices de pesos (para representar conexiones en capas, normalmente cuando se trata de conexiones de "todos a todos") y, muy importante, objetos de tipo "accion" que permiten, vía polimorfismo, describir diversos de cálculos a realizar con las piezas descritas anteriormente. La red contiene una secuencia de acciones a realizar durante el forward (durante la fase backward se ejecutan la parte backward de dichas acciones en el orden contrario). De este modo es posible combinar diversas secuencias de acciones para modelar arquitecturas muy diversas sin tener que modificar la jerarquía de clases. En caso de querer soportar tipos de red que no se pueden construir mediante las piezas ya existentes, basta con añadir las subclases necesarias reutilizando lo anterior y sin tener que modificar la parte central del diseño. En general, el diseño de clases que mostramos a continuación facilita la creación de nuevas subclases para la extensión hacia nuevos tipos de componentes y arquitecturas para las ANN, nuevos criterios de error en el aprendizaje, etc.

A continuación se van a mostrar las clases genéricas, cuyas interfaces han de ser implementadas por las clases derivadas de éstas. Después de las clases genéricas, se enunciarán las subclases más empleadas y se explicará qué elementos de las ANN tratan de modelar.

3.2.1. La estructura ANNConfiguration

Esta estructura se relaciona con la mayoría de clases que se van a exponer, por lo que es aconsejable comentarla en primera instancia. La estructura *ANNConfiguration* es una estructura que agrupa los diversos parámetos que necesitan acceder la mayoría de los elementos de la red y que está compuesta por tres valores:

- **cur_bunch_size**: El valor del parámetro *bunch* actual, que marca el número de muestras de entrenamiento que se procesan en cada iteración del

algoritmo BP.

- **max_bunch_size**: El valor del parámetro *bunch* máximo, que marca el número máximo de muestras de entrenamiento procesadas por las iteraciones del algoritmo BP hasta la iteración actual.

  Este valor será igual al valor *cur_bunch_size* normalmente, aunque es necesario guardar este número para poder realizar correctamente los cálculos en las últimas muestras de entrenamiento presentadas en cada época.

  Imagínese que se diseña un experimento con 100 muestras de entrenamiento sobre una red que procesa 32 muestras por iteración. Al no ser 100 múltiplo de 32, en la cuarta iteración del algoritmo quedarán por procesar 4 muestras, por lo que se necesita ajustar el valor del *bunch*. No obstante, si lo ajustamos y no guardamos el valor previo, estaríamos perdiendo la información sobre cómo se organizan las matrices de valores de la red en memoria.

- **use_cuda_flag**: El *flag* para utilizar CUDA se marca con este atributo. Por defecto, su valor es *false*, con lo cual todas las operaciones se ejecutan en CPU. Si el valor de este *flag* se pone a *true*, entonces se utilizará la GPU para efectuar la mayoría de operaciones.

La información que esta estructura proporciona es requerida por prácticamente todos los elementos de la red, los cuales guardan una referencia de ésta. Es por ello que, en las próximas secciones, no se considerará la referencia como un atributo de esas clases.

3.2.2. La clase ANNBase

La clase *ANNBase* es la clase utilizada para representar una ANN de un tipo bastante general, hereda de otra clase *TrainableSupervised* todavía más general que únicamente asume que existen métodos para proporcionar pares de entrada y salida en un paradigma de aprendizaje supervisado. *ANNBase* contiene atributos para almacenar los parámetros del entrenamiento y valores relacionados con las entradas y las salidas de la red. También contiene varias referencias a los bloques de memoria para almacenar:

- Los valores de entrada de la red.
- Los valores de salida de la red.
- Los valores deseados para las salidas de la red.
- Los errores producidos entre las salidas obtenidas y las salidas deseadas en la red.

Además de estos atributos, la clase *ANNBase* dispone de tres vectores para guardar:

- Las referencias a las capas de activación de la red, que son objetos que representan conjuntos de neuronas.

- Las referencias a las conexiones entre las capas de activación (matrices de pesos). Observemos que nuestro diseño permite tener "pesos ligados", es decir, que distintas conexiones entre capas compartan el mismo vector de pesos que se trata de manera cosistente y adecuada durante el entrenamiento.

- Las referencias a las acciones a ejecutar sobre la red. Que es la forma genérica de poder añadir una lista de cosas a ejecutar de manera totalmente heterogénea. Algunas acciones se corresponden con la conexión entre capas usando las conexiones, pero otras acciones permiten hacer que una capa se divida en varias, que existan pesos ligados, etc.

Adicionalmente, la clase incorpora métodos para realizar el registro de estos componentes en sus correspondientes vectores. Las funciones para ejecutar el paso hacia-delante (*doForward()*) y el paso de retropropagación (*doBackward()*) no tienen implementación: Cada topología ha de implementar su forma particular de realizarlos, utilizando las clases adecuadas para ello.

Si se pretende extender la aplicación para trabajar con nuevas topologías de red, se recomienda que su implementación se realice a partir de la extensión de esta clase, ya que los métodos base que cualquier topología de red necesita tener están ya implementados.

La clase MLP

La clase MLP es una subclase de *ANNBase* y su nombre proviene de la abreviatura de "MultiLayer Perceptron" o, en castellano, Perceptrón Multicapa. Esta clase representa el tipo de red construido a base de unir capas de neuronas y que normalmente se corresponde con una red de tipo "capa a capa". Esta red ofrece el interfaz de *ANNBase* que permite entrenar mediante pares de entrada salida al tiempo que incluye la lógica necesaria para recurrir de manera transparente al modo bunch, almacenando patrones hasta llenar un "bunch" a menos que le indiquemos que ya no hay más muestras para procesar en esta época. La red guarda los siguientes atributos además de los valores de los parámetros relacionados con el entrenamiento:

- **num_patterns_processed**: El número de muestras de entrenamiento procesadas hasta el momento.

- **cur_bunch_pos**: El número de muestras de entrenamiento ya cargadas en la capa de entrada. Se utiliza como un índice para cargar las siguientes muestras de entrenamiento.

- **error_func**: Un puntero al objeto función de error que se utiliza en la red.

- **batch_error_sum**: Un atributo para almacenar el valor devuelto por el cálculo de la función de error de la red.

La clase implementa las funciones ya enunciadas en la clase *ANNBase* y muchas otras:

- **trainPattern**: Añade una muestra de entrenamiento al *bunch* actual. En caso de completar el *bunch*, se llama a la función *doTraining()* para ejecutar el entrenamiento.
- **validatePattern**: Añade una muestra de validación al *bunch* actual. En caso de completar el *bunch*, se llama a la función *doForward()* para ejecutar el paso hacia-delante y obtener los resultados de clasificación de la red para ese conjunto de muestras.
- **beginTrainingBatch**: Inicializa los atributos de la red tales como el número de muestras procesadas o la suma de los errores devueltos por la función de error.
- **endTrainingBatch**: Se examina si quedan muestras de entrenamiento por procesar, en cuyo caso se ajusta el tamaño del bunch y se procesan las muestras restantes. Luego se comprueba que los valores de los pesos de la red sean correctos y se devuelve la suma total de los errores devueltos por la función de error.
- **beginValidateBatch**: Inicializa los atributos de la red tales como el número de muestras procesadas o la suma de los errores devueltos por la función de error.
- **endValidateBatch**: Se examina si quedan muestras de validación por procesar, en cuyo caso se ajusta el tamaño del bunch y se procesan las muestras restantes. Luego se devuelve la suma total de los errores devueltos por la función de error.
- **doTraining**: Primero se inicializan los vectores de error de las capas de la red y se ejecuta el paso hacia-delante. A continuación, se obtiene la suma de los errores devueltos por la función de error para el conjunto de muestras de entrenamiento procesadas en el *bunch* actual. Finalmente, se ejecuta el paso de retropropagación y se modifican los pesos de la red.

Como puede observarse, la mayoría de las funciones están relacionadas con la organización del entrenamiento de la red. Es este proyecto, ése es el cometido de las redes: proporcionar mecanismos de automatización del entrenamiento para simplificar el diseño de experimentos.

La clase *AllAllMLP* representa un MLP cuyas capas son objetos de la clase *RealActivationUnits* y cuyas conexiones son objetos de la clase *AllAllConnections*. Estas clases son comentadas posteriormente en la memoria.

3.2.3. La clase ActivationUnits

Las capas que agrupan conjuntos de neuronas son implementadas en la clase *ActivationUnits*. Esta clase no contiene ningún tipo de atributo, ni tampoco implementa ninguna función genérica para las capas, pues no son necesarias (en este conteo y en los próximos, los *setters*, los *getters* y las funciones para obtener una réplica del objeto no se consideran funciones, dispongan o no de una implementación en la clase para la que se cuentan).

Las funciones sin implementación que define la interfaz de la clase *ActivationUnits* son:

- **activate**: Se aplica la función de activación asignada a la capa sobre sus unidades.
- **multiplyDerivatives**: Se multiplican las derivadas de las unidades de la capa por los resultados del vector de error y se guardan en ese mismo vector.
- **reset**: Se ponen a 0 los valores del vector de errores de la capa.

La clase RealActivationUnits

Esta clase hereda de *ActivationUnit* e implementa una capa de unidades, sin ningún tipo de particularidad numérica, a excepción de que sus valores son números reales representados en coma flotante. Es decir, implementa la forma más habitual de una capa de neuronas, con lo que sería conveniente considerar otras alternativas para entender por qué se ha distiguido entre las *ActivationUnit* y las *RealActivationUnits*. De hecho, existen al menos otras dos subclases de *ActivationUnit*, una de ellas es *LocalActivationUnits* que representa una capa donde una sola neurona está activada a 1 y el resto a 0, muy conveniente en determinadas tareas. Los atributos de la clase *RealActivationUnits* son:

- **activations**: Una referencia a un bloque de memoria para almacenar los valores de las neuronas de la capa.
- **error_vector**: Una referencia a un bloque de memoria para almacenar los valores de los *errores retropropagados*. La talla de este bloque será exactamente igual a la talla del vector de activaciones.
- **act_func**: Una referencia al objeto que implementa la función de activación de esta capa. Es decir, todas las neuronas de la misma capa tienen la misma función de activación, lo cual no es la forma más general pero, por otra parte, simplifica enormemente el uso de funciones de activación como la *softmax* que resulta más compleja de implementar en diseños basados en neuronas individuales en lugar de en capas.

Dada la simplicidad de esta clase, no se necesita incorporar nuevas funciones, excepto aquellas que exigen su implementación desde la clase *ActivationUnits*.

3.2.4. La clase Connections

Las "conexiones" es la palabra con la que usualmente nos referimos a los pesos que relacionan las unidades de una capa con las de la capa siguiente. En este proyecto, las conexiones se vuelven mucho más relevantes que en la mayoría de implementaciones del algoritmo BP, pues los cálculos más pesados se organizan a partir de ellas. A continuación, se comentan las funciones de la interfaz abstracta que todo tipo de conexión debe implementar:

- **forward**: A esta función se le pasan como parámetros las referencias de una capa de entrada y otra de salida. Las conexiones se encargan de obtener los valores de las unidades y, a continuación, realizan el producto matricial de estas unidades con los pesos de las conexiones. Los resultados se guardan en las unidades de la capa de salida, y luego se llama a la función de activación desde esta capa para aplicarla sobre sus unidades.

- **updateWeights**: Esta función recibe los mismos parámetros que la anterior. En primera instancia, se llama a la función para multiplicar las derivadas en la capa de salida. Acto seguido, se retropropagan los errores hacia la capa de entrada y se modifican los pesos de acuerdo al incremento que se ha calculado, aplicando el factor momentum o weight decay sobre ellos y guardando los pesos para la siguiente iteración si cualquiera de estos parámetros está activo.

- **loadWeights**: La carga de pesos puede hacerse con esta función a partir de una *matrix*, que es una clase disponible en el toolkit April y que se puede manipular en el *script* que describe el experimento. En el capítulo de experimentación se describe qué es una *matrix* y cómo se usan en los experimentos con redes.

- **copyWeightsTo**: La copia de los pesos a fichero se realiza con esta función, la cual actúa de un modo reverso a la anterior, salvando los pesos de la red en una *matrix* para almacenarla posteriormente en un fichero.

- **randomizeWeights**: La inicialización de los pesos de las conexiones se realiza a partir de un generador de números aleatorios (que también se analizará en el capítulo de experimentación), inicializado a su vez con una semilla declarada en el *script* de *Lua* que describe el experimento. Esta función realiza las llamadas para obtener esos números aleatorios y asignarlos a los pesos de las conexiones.

- **checkInputOutputSizes**: El nombre de esta función ya nos indica que su cometido es el de comprobar que el número de entradas y el número de salidas en las capas concuerdan con el número de pesos en esta conexión.

- **pruneSubnormalAndCheckNormal**: Los pesos cuyo valor estén por debajo de un cierto umbral son "podados", es decir, su valor se fija a 0. En particular, los números en coma flotante conocidos como "subnormal" se pueden convertir a 0 sin afectar apenas la precisión del aprendizaje mientras que su manipulación por parte de la mayoría de la CPUs es sustancialmente más lenta que con el resto de números en coma flotante. También se comprueba que los números sean correctos, es decir, que no sean valores tipo *NaN* (not a number) o infinito. En caso de que se encuentre alguno, se alerta al usuario de ello y se aborta el programa.

Como la mayoría de funciones no tienen implementación, la clase genérica de conexiones tampoco dispone de atributo alguno, excepto el umbral para la detección de números subnormales.

La clase AllAllConnections

Esta clase implementa un conjunto de conexiones que relacionan todas las unidades de la capa de entrada a la conexión con todas las unidades de la capa de destino de la misma conexión. Éste es uno de las tipos más comunes de conexiones en el diseño de redes, a pesar de que también implica la realización de un número elevado de cálculos, por lo que se ha tratado de obtener una eficiencia máxima en su implementación.

La clase *AllAllConnections* contiene los siguientes atributos:

- **weights_block**: Una referencia al bloque de memoria en donde se guardan los valores de los pesos de las conexiones.
- **prev_weights_block**: Una referencia al bloque de memoria en donde se guardan los valores de los pesos de las conexiones en la iteración anterior.
- **update_weights_calls**: El número de veces que se ha llamado la función *updateWeights*. Este valor es necesario para distinguir si es la primera vez que se llama a la función o la última.

Además de estos atributos, también se guardan los valores de los parámetros relacionados con el entrenamiento:

- **learning_rate**: El valor del factor de aprendizaje.
- **momentum**: El valor del parámetro momentum.
- **weight_decay**: El valor del parámetro weight decay.
- **c_weight_decay**: El valor que representa el porcentaje del parámetro weight decay que no se aplica: $1 - weight\_decay$.

Uno de los motivos para tener copias locales de estos valores, en lugar de simplemente una referencia a los mismos valores en la clase *MLP*, es la posibilidad de utilizar valores diferentes en capas diferentes de la red.

Además de las funciones que esta clase hereda de *Connections*, se han creado nuevas funciones para simplificar el código de los otros métodos:

- **forwardSingleInput**: Esta función realiza los mismos cálculos que la función *forward* de la clase *Connections* para ejecutar el modo *on-line*.
- **backpropagateErrors**: La retropropagación de errores se realiza con esta función. Básicamente, se realiza un producto de las derivadas ya multiplicadas de las unidades en la capa de salida por los valores de los pesos en las conexiones, guardando los resultados en el vector de errores de la capa de entrada.
- **computeMomentumOnPrevVectors**: Se incrementan los valores de los pesos de la anterior iteración de acuerdo con el factor momentum.

- **computeBPUpdateOnPrevVectors**: Se incrementan los valores de los pesos de la anterior iteración de acuerdo con la fórmula completa del BP. Estas dos últimas funciones realizan los cambios sobre el vector previo de pesos, ya que al final de *updateWeights* se intercambian los punteros a los bloques de memoria que guardan los pesos de la iteración anterior y la actual, en caso de que sea la última llamada a *updateWeights* para esas conexiones.

Con la adición de estas funciones, se logra obtener un buen rendimiento en el cálculo del incremento de pesos para los algoritmos BP con momentum y BP con weight decay. Un gran número de implementaciones descarta la inclusión de estos parámetros del entrenamiento ya que su programación resulta tediosa e ineficiente. En este proyecto hemos logrado incluir ambos factores en la implementación del algoritmo, ayudándonos del atributo que guarda el número de llamadas a updateWeigths y el número de referencias del objeto conexiones para distinguir en qué casos hemos de aplicar estos parámetros y cómo.

3.2.5. La clase ActivationFunction

La clase *ActivationFunction* es otra clase genérica cuyo objetivo es proporcionar una interfaz para implementar las funciones de activación en nuestro proyecto. Como ya sucede en las otras clases genéricas, esta clase no contiene ningún atributo propio. Las funciones que debe implementar cualquier clase descendiente de la clase *ActivationFunction* son sólo dos:

- **applyActivation**: Se aplica la función de activación sobre las neuronas de la capa.
- **multiplyDerivatives**: Se multiplican la derivada de la función de activación respecto al valor de la neurona por el valor de las neuronas de la capa.

Se ha creado una clase que hereda de *ActivationFunction* para cada una de las funciones de activación utilizadas en este proyecto (logística, tanh, lineal, softmax) y resulta muy sencillo añadir nuevas funciones de activación en caso de necesidad.

3.2.6. La clase Action

Las acciones que la red debe realizar son representadas como objetos de la clase *Action*. Estos objetos son guardados en un vector de forma ordenada (que es un atributo de la clase *ANNBase*) y la red se encarga de recorrer el vector para ejecutar las funciones adecuadas de los objetos de acción. La interfaz de la clase *Action* declara dos funciones sin implementación:

- **doForward**: Se realizan las acciones que deben efectuarse al ejecutar el paso hacia-delante del algoritmo BP. En el caso más simple, esto significa llamar a la función *forward* de las conexiones con las que está asociada.

- **doBackward**: Se realizan las acciones que deben efectuarse al ejecutar el paso de retropropagación del algoritmo BP. En el caso más simple, esto significa llamar a la función *updateWeights* de las conexiones con las que está asociada.

Esta interfaz puede resultar confusa en un principio y nos hace plantearnos la utilidad de esta clase dentro de la aplicación. No obstante, la inclusión de la clase *Action* está más que justificada para el caso de arquitecturas de redes complejas, proporcionando un mecanismo de abstracción para ejecutar los dos pasos del algoritmo sin tener que preocuparnos de lo que sucede por debajo. Como hemos comentado anteriormente, mediante acciones se puede no sólo especificar en qué orden se realizan los cálculos de cada capa, sino que también se puede ejecutar el código necesario para ligar pesos, para copiar partes de una capa en otra, etc. En el apartado *Ampliaciones futuras* del capítulo sexto se expondrán un par de ampliaciones en las cuales los objetos de la clase *Action* jugarían un papel importante.

3.2.7. La clase ErrorFunc

La literatura relacionada con las ANN y el algoritmo BP suele emplear la función Error cuadrático medio o "Mean Square Error" (MSE) como función de error en todos los ejercicios propuestos. No obstante, existen otras muchas funciones disponibles para valorar el error producido por la red respecto a la función de la red que se trata de aproximar.

Por esto mismo, se facilita una clase genérica a partir de la cual podemos implementar nuevas funciones de error para nuestras redes. Las funciones que éstas deben implementar son las siguientes:

- **computePatternErrorFunction**: Esta función calcula el error que se comete en cada unidad de la capa de salida respecto a las salidas deseadas, y deposita el resultado en el vector de errores de la capa de salida.
- **computeBatchErrorFunction**: Esta función calcula la función de error de la red E. Para ello, utiliza los valores depositados en el vector de errores y devuelve el valor calculado.

3.3. La API de BLAS

BLAS es la API estándar para la publicación de librerías que implementen las operaciones básicas de álgebra lineal tales como la multiplicación de vectores y matrices.

Hay bastante variedad a la hora de elegir alguna de las implementaciones de BLAS. Podemos elegir entre una librería de código abierto como Automatically Tuned Linear Algebra Software (ATLAS), que suele tener un buen rendimiento independientemente de la CPU para la cual se esté optimizando el código, o también podemos usar una implementación facilitada por los fabricantes, como es el caso de la librería Intel Math Kernel Library (MKL) de Intel o la librería AMD

Core Math Library (ACML) de AMD. Estas últimas implementaciones suelen tener un rendimiento mucho mayor al de librerías como ATLAS al estar especializadas para los procesadores de los fabricantes, aunque puede haber problemas a la hora de emplearlas en otras arquitecturas.

La API de BLAS clasifica las operaciones en tres niveles de forma que en el primer nivel se sitúan las operaciones sencillas del álgebra lineal y en los niveles más altos se sitúan las operaciones con un grado de complejidad mayor. A continuación se enumerará cada uno de estos niveles, algunas de las operaciones que lo componen y se remarcarán aquellas operaciones que han sido de utilidad para la implementación del proyecto.

3.3.1. El nivel 1 de BLAS

El primer nivel de la API de BLAS está compuesto por una gran variedad de operaciones que trabajan sobre uno o más vectores. Las llamadas de este nivel nos permiten:

- Generar y aplicar rotaciones sobre el plano.
- Operaciones sencillas como la copia o el intercambio de vectores.
- Cálculos de vectores resultado para la multiplicación escalar y la suma de vectores.
- Operaciones para la obtención del máximo, el mínimo o la raíz de la suma de cuadrados de un vector.
- Productos escalares entre dos vectores.

Las operaciones de este nivel, al igual que todas las operaciones de la API de BLAS, pueden trabajar sobre vectores de números reales representados en coma flotante simple (*float*), doble (*double*) o números complejos.

De entre las operaciones que contiene el primer nivel de BLAS, el proyecto usa las siguientes funciones:

- *scopy*: Realiza la copia de una zona de memoria en otra zona de memoria. Se usa para copiar las muestras de entrada a la red o los pesos de la actual iteración al bloque de memoria en donde se guardarán para la siguiente iteración.
- *sscal*: Multiplica un vector x por un factor α y lo guarda en la misma zona de memoria:

  $$x \leftarrow \alpha x$$

  Sirve, por ejemplo, para ejecutar la multiplicación escalar del momentum sobre los pesos de la red.

- *saxpy*: Multiplica un vector x por un escalar α y le suma el vector y. El resultado se guarda en y:

$$y \leftarrow \alpha x + y$$

  Se emplea en el cálculo de neuronas en las capas de salida afectadas por el *bias* de las capas de entrada.

Los ejemplos de uso que acompañan a las funciones empleadas en el proyecto son sólo algunos de sus usos, pues la implementación del BP está en gran parte escrita empleando llamadas a funciones de BLAS.

3.3.2. El nivel 2 de BLAS

El segundo nivel de la API de BLAS está compuesto en su totalidad por operaciones que involucran una matriz y un vector. El gran número de llamadas que la API de BLAS ofrece en este nivel nos hace dudar que sólo se estén efectuando este tipo de operaciones, pero hay que aclarar que esto se debe a que tenemos llamadas especializadas según el tipo de la matriz.

Esto nos posibilita el acelerar todavía más nuestro código si sabemos que vamos a trabajar con este tipo de matrices en determinadas partes de un proyecto: La API nos permite alertarla de que se está usando una matriz simétrica, hermitiana o triangular. En el caso de que nuestra matriz no tenga ninguna de las características mencionadas, usaremos las operaciones de tipo general.

En este proyecto se utilizan dos funciones de este nivel para realizar los cálculos del algoritmo BP para el modo "on-line" (el caso en el cual sólo se procesa una muestra por cada iteración del algoritmo). En este modo, las unidades de activación toman la forma de un vector de unidades, y los cálculos se realizan de forma análoga a como se ha mostrado en la sección *Forma matricial del BP* (los vectores pueden ser considerados matrices con una sóla fila). Las funciones utilizadas son:

- *sgemv*: Este cálculo realiza el producto de una matriz genérica A multiplicada por un escalar α y un vector x, produciendo otro vector al cual se le suma el vector y multiplicado por un escalar β. El resultado se guarda en y:

$$y \leftarrow \alpha Ax + \beta y$$

  La función es empleada para obtener los resultados de las unidades de las capas a partir del producto de las capas anteriores a éstas y las conexiones que las relacionan.

- *sger*: Este cálculo realiza el producto de dos vectores de la misma talla x e y, aplicando un factor α sobre el primer vector y trasponiendo el segundo, de forma que se obtiene una matriz que se añade a otra matriz A, que le pasamos como parámetro a la función, guardando el resultado en la zona de memoria de A:

$$A \leftarrow \alpha x y^T + A$$

Esta operación es empleada por el algorimo BP con momentum para calcular el incremento de los pesos de la red.

En la parte final de este mismo capítulo, se realiza un estudio sobre la aceleración producida al implementar el cálculo del BP en modo "on-line" con estas rutinas del nivel 2 de BLAS.

3.3.3. El nivel 3 de BLAS

En el tercer nivel de la API de BLAS tenemos reunidas las operaciones necesarias para efectuar un producto entre dos matrices. Como ya ocurría en el anterior nivel, podemos emplear las llamadas especializadas para las matrices simétricas, hermitianas y triangulares, además de tener una llamada para matrices sin ninguna particularidad.

En el proyecto sólo se emplea la función *sgemm* de entre las funciones de este nivel. La operación *sgemm* realiza el producto matricial de dos matrices genéricas, A y B, aplicando un factor α sobre la primera. El resultado se suma la matriz C, a la cual también se le aplica un factor β. La fórmula completa es:

$$C \leftarrow \alpha AB + \beta C$$

Esta operación resulta esencial para el cálculo del BP ya que podemos emplearla para la obtención de los resultados de una capa partir del producto de la capa anterior por los pesos que las conectan. También podemos emplear esta operación para la actualización de los pesos en la segunda parte del algoritmo.

3.4. La biblioteca ATLAS

La biblioteca ATLAS ya ha sido introducida anteriormente. Esta biblioteca implementa las funciones descritas en la API de BLAS para los lenguajes C y Fortran77, además de otras funciones que no forman parte de BLAS [WPD01].

El proyecto ATLAS tiene como objetivo proveer al usuario de una biblioteca portable y eficiente para la resolución de cálculos de la álgebra lineal. De este modo, el usuario puede obtener un buen rendimiento en su máquina independientemente de la arquitectura que ésta tenga.

De las funciones adicionales que no forman parte de BLAS y que implementa esta biblioteca, resulta de especial interés la función *catlas_sset*. Esta función puede realizar la inicialización de los valores de un vector a un valor determinado.

3.5. La biblioteca Intel MKL

La biblioteca Intel MKL es una biblioteca que facilita el fabricante de CPUs Intel. Esta biblioteca implementa todo tipo de utilidades para el cálculo matemático entre las que se incluyen la implementación de la API de BLAS [Int07].

En contraste con la biblioteca ATLAS, cuyo objetivo es ofrecer un alto grado de portabilidad entre diferentes arquitecturas, la Intel MKL está optimizada para obtener el máximo rendimiento posible en una gran variedad de arquitecturas de procesadores Intel.

En el proyecto no se emplea solamente aquellas funciones que forman parte de la API de BLAS, sino que se ha optado por el uso de funciones matemáticas, como la función exponencial, para obtener un rendimiento y precisión todavía mayores.

Además, la biblioteca Intel MKL nos ofrece la posibilidad de ajustar el número de núcleos que deseamos emplear para efectuar los cálculos en nuestro procesador. Esto supone una gran aceleración en los ordenadores actuales, ya que la mayoría de ordenadores de sobremesa disponen de procesadores de entre cuatro y ocho núcleos, pero pocas son las aplicaciones que hacen un uso total de ellos.

3.6. Otras decisiones de diseño

Aunque las decisiones de diseño más importantes ya han sido tomadas en cuenta, todavía quedan pendientes otros factores que pueden influir en la calidad final del proyecto. A continuación se discuten cuáles son estas decisiones y las razones que nos llevan a pronunciarnos sobre la opciones finalmente incluídas en el diseño.

3.6.1. Cálculos en coma flotante

Existen dos alternativas a la hora de realizar cálculos con números reales, ya que podemos escojer entre una representación de números reales en coma flotante simple (denominados tipos de valor *float* en la mayoría de lenguajes de programación) o doble (en este caso denominados tipos de valor *double*).

Ya se ha dejado entrever en anteriores secciones, por las operaciones de BLAS que se emplean en el proyecto (*scopy*, *saxpy*, etcétera...) que el tipo de valor empleado para la representación de los números reales será el *float*. Varios artículos han demostrado que este tipo es suficiente para trabajar con redes neuronales. El tipo *double* (cuyo tamaño en memoria ocupa el doble que un *float*) puede llegar a aumentar considerablemente el número de fallos en la memoria caché, lo cual implica un número mayor de transferencias de memoria principal a caché. Además, las operaciones que se ejecutan con valores del tipo *double* son más costosas por lo que el tiempo requerido para efectuarlas es mayor.

No obstante, la realización de algunos experimentos con ANN podrían requerir el uso de valores de tipo *double*. Los bloques de memoria, que serán introducidos más adelante en este trabajo, están implementados como un *template* cuyo valor está actualmente fijado a *float*, pero es posible modificar el código para que se empleen valores de tipo *double* en estos bloques.

3.6.2. Representación de matrices

La API de BLAS nos da la opción de escoger entre dos tipos de representación de matrices para realizar operaciones en las cuales uno de los operandos es una

matriz. La representación de una matriz tiene relación con la forma de almacenar los componentes de ésta en memoria y por tanto, puede influir en el rendimiento de la aplicación final.

La representación *row-major* es la representación clásica, en la cual las filas son almacenadas en memoria de forma consecutiva. La representación *column-major*, en cambio, almacena los valores de la matriz por columnas.

Para que quede claro cómo se organiza en memoria una matriz en función de su representación, a continuación se presenta la siguiente matriz:

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{pmatrix}$$

Para almacenar los valores de esta matriz, será necesario reservar un vector de 8 componentes, ya que las dimensiones de la matriz son 2 filas y 4 columnas.

El ordenamiento en memoria, en caso de emplear la representación *row-major*, será el siguiente:

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \end{bmatrix}$$

Como puede observarse, la primera fila es almacenada entera en primer lugar, y luego sucede lo mismo para la siguiente fila. En esta representación, para acceder al siguiente elemento de la matriz dentro de una misma fila, basta con incrementar el puntero en una unidad. En caso de encontrarnos en el último elemento de una fila y querer acceder al primer miembro de la siguiente fila, el método empleado anteriormente también funciona en caso de que no haya un hueco de memoria entre ellos (lo que se suele llamar un *offset*).

En cambio, si utilizamos la representación *column-major* para estos propósitos, la zona de memoria quedaria de este modo:

$$\begin{bmatrix} 1 & 5 & 2 & 6 & 3 & 7 & 4 & 8 \end{bmatrix}$$

En este caso, sucede lo contrario: Primero se almacena la primera columna, luego la siguiente, y así hasta la última. En esta representación, para acceder al siguiente elemento dentro de una misma fila, tenemos que incrementar el puntero en tantas unidades como columnas tenga la matriz. Para acceder al elemento que se encuentra en la siguiente fila y en la misma columna, tan sólo es necesario un incremento de una unidad sobre el puntero.

Las primeras versiones del proyecto, que sólo implementaban la ejecución del BP mediante CPU, empleaban *row-major* para la representación de matrices en memoria. No obstante, al llegar el momento de implementar las llamadas a CUBLAS para la realización de cálculos mediante GPU, se forzó el cambio a la representación *column-major*, ya que CUBLAS sólo acepta matrices representadas de este modo para efectuar los cálculos.

El cambio tuvo un efecto colateral beneficioso para el proyecto: Las mismas ejecuciones, realizadas anteriormente con la representación *row-major*, eran ahora más veloces por el simple hecho de cambiar a *column-major*.

3.6.3. Guardado temporal de los pesos

El almacenamiento de los pesos de la iteración previa a la iteración actual es necesario para la implementación del BP con parámetros tales como el momentum o el weight decay, por lo que se emplean dos bloques de memoria en cada conexión para guardar los pesos de la iteración actual y los pesos de la iteración previa a la actual.

3.6.4. El entrenamiento on-line

El empleo de las bibliotecas de álgebra lineal puede resultar provechoso para el entrenamiento paralelo de una cantidad elevada de muestras. No obstante, para valores pequeños del parámetro *bunch*, el uso de estas herramientas puede resultar contraproducente. El ejemplo más claro de esto es el del entrenamiento "on-line", el cual puede simularse procesando una muestra por iteración (por lo que el valor del parámetro *bunch* es 1).

En las primeras versiones de este proyecto, los cálculos se realizaban sin distinguir entre el modo "on-line" y el modo *bunch*, por lo que se utilizaban las funciones del nivel 3 de BLAS, tratando el vector de unidades como una matriz de dimensiones $1 \times n$, en donde n era el número de unidades de esa capa. Los resultados no fueron satisfactorios, por lo que decidimos no utilizar ninguna de las funciones de BLAS para realizar estos cálculos. En la parte final del proyecto nos decantamos por utilizar funciones del nivel 2 de BLAS y, después de experimentar con el modo "on-line", los resultados mostraban una aceleración notable respecto a la versión que no empleaba rutinas de BLAS.

Capítulo 4

Implementación del BP con CUDA

Las decisiones de diseño tomadas en el anterior capítulo nos permiten ahora dirigir nuestra atención hacia los detalles finales de la implementación del algoritmo BP.

El paso de llamadas BLAS (que hasta el momento sólo utilizan la CPU para realizar sus cálculos) a GPU sería la mejora obvia que daría cuenta del principal cuello de botella de la aplicación. Para llevar a cabo esta mejora, nos ayudaremos de la biblioteca CUBLAS, que ya implementa las funciones de la API de BLAS para el lenguaje CUDA.

La conservación de las llamadas BLAS con CPU nos obligará a rediseñar la manipulación las zonas de memoria con los bloques de memoria, los cuales implementan una copia *lazy* (perezosa) de datos entre memoria principal y la memoria GPU. Los *wrappers* para el álgebra lineal trabajarán con estos bloques de memoria para efectuar llamadas BLAS independientemente de la biblioteca que las implemente.

El acoplamiento de esta nueva biblioteca (CUBLAS) también nos obliga a cambiar el tipo de representación de las matrices, que antes era *row-major*, para poder realizar operaciones con CUDA, que trabaja con una representación *column-major*.

Posteriormente, hemos decidido mejorar todavía más la implementación pasando a CUDA otras partes del diseño, con el objetivo de evitar innecesarias tranferencias de memoria y acelerar procesos como la aplicación de las funciones de activación.

4.1. Las tarjetas gráficas Nvidia y la tecnología CUDA

El fabricante de tarjetas gráficas Nvidia puede ser considerado uno de los fabricantes más importantes de tarjetas gráficas a nivel mundial. Especializada en este tipo de componentes, Nvidia siempre ha apostado por la innovación dentro del campo del procesamiento gráfico, aportando nuevas tecnologías como *PhysX*,

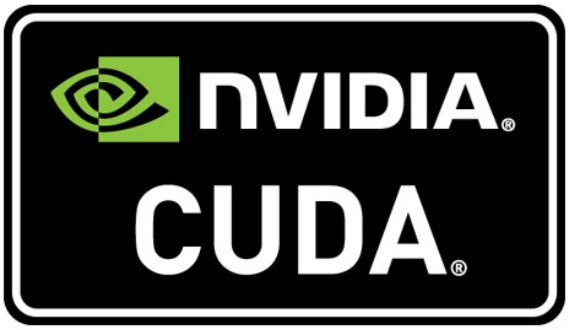

Figura 4.1: El logo de CUDA.

3D Vision o *SLI*.

Una de las tecnologías desarrolladas por Nvidia es la tecnología CUDA. La tecnología CUDA es una plataforma de cómputo paralelo de las GPU de Nvidia con una arquitectura compatible con ésta, accesible para los desarrolladores de *software* a través de extensiones de los lenguajes de programación más empleados en la indústria. Este sublenguaje, que se incorpora a lenguajes como C++, es conocido también como lenguaje CUDA.

Los dispositivos CUDA disponen de una arquitectura SIMD que permite la aplicación de operaciones de forma simultánea sobre un conjunto de datos almacenado en la memoria GPU.

Dado el grado de aceleración que supone la adaptación de esta tecnología en algunos programas, su uso ya está presente en campos como el análisis de flujo del tráfico aéreo o la identificación de placas ocultas en el cuerpo humano gracias a la simulación del flujo de sangre en el cuerpo humano.

4.2. El modelo de ejecución de CUDA

El modelo de ejecución de CUDA nos permite realizar cálculos de forma paralela sobre una conjunto de datos, con lo que se consigue una aceleración notable respecto a los cálculos secuenciales que solemos implementar en los programas corrientes, que hacen un uso mínimo de los núcleos de la CPU.

En esta sección se explicarán los aspectos básicos de la tecnología CUDA, tratando de relacionar las estructuras diseñadas para organizar los cálculos con el *hardware* que los ejecuta. Luego, se comentarán las aportaciones del lenguaje CUDA necesarias para programar una transformación simple sobre una matriz. De este modo, el lector tendrá una idea sobre cómo funcionan estos dispositivos y cómo los manejamos para hacer lo que deseamos.

4.2.1. Elementos de CUDA

Las GPUs que disponen de la tecnología CUDA están formadas por un número limitado de *cores* o núcleos. Estos núcleos pueden verse como versiones pequeñas de una CPU, capaces de ejecutar un conjunto de instrucciones simple, sin llegar

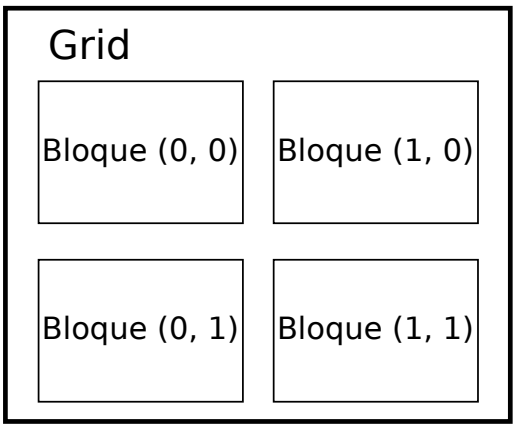

Figura 4.2: Un grid de dimensiones 2×2.

a la potencia de cálculo de las CPU de hoy en día. El número de núcleos varía de una tarjeta gráfica a otra, aunque lo normal es que este número se sitúe entre los 100 y los 300 núcleos. Los núcleos se agrupan para formar multiprocesadores de flujo (de ahora en adelante, los llamaremos procesadores).

El modelo de ejecución de CUDA se entenderá mejor una vez explicados los elementos que participan en él. Se dejarán de lado los aspectos más avanzados de la tecnología ya que el proyecto no hace uso de ellos. Para conocer más sobre el lenguaje CUDA y sus componentes, se recomienda la lectura de los manuales proporcionados por Nvidia [Cor12a, Cor12c, Cor12b].

El grid

El *grid* (en español, la *rejilla*) es el elemento primario del modelo de ejecución de CUDA. Un *grid* se compone de un número determinado de bloques, hasta un máximo de 65535 bloques.

Dado que la tecnología CUDA suele aplicarse para el tratamiento de matrices de 2 dimensiones, el número de bloques que componen el *grid* suele declararse como un par de números que especifican el tamaño de las dimensiones del *grid*, aunque es posible emplear desde 1 hasta 3 dimensiones para ello. En la figura podemos observar un *grid* declarado con unas dimensiones de 2×2 bloques.

El bloque

Los bloques no tienen un correspondencia uno a uno con los procesadores, aunque la ejecución de un bloque se da en uno de ellos. El número de bloques declarados en el *grid* se reparte entre los procesadores disponibles, siendo necesario hacer varias iteraciones de reparto en caso de que el número de bloques exceda el número de procesadores, hasta procesar el número total de bloques.

Los bloques, a su vez, se componen de hilos. Como ya sucedía con los bloques, es posible especificar el número de hilos como si de las dimensiones de una matriz se tratase. Como mucho, pueden emplearse hasta 3 dimensiones en su declaración, y el producto de los valores de esas dimensiones no puede ser mayor que 1024 para

las GPUs de última generación. En la tabla de la figura 4.3 se adjuntan los valores máximos para la declaración de los elementos de CUDA según la versión de la arquitectura de la GPU.

	Compute Capability				
Technical Specifications	**1.0**	**1.1**	**1.2**	**1.3**	**2.x**
Maximum x- or y-dimension of a grid of thread blocks	65535				
Maximum number of threads per block	512				1024
Maximum x- or y-dimension of a block	512				1024
Maximum z-dimension of a block	64				

Figura 4.3: Dimensiones máximas de los elementos de CUDA en función de la versión de la GPU.

Dentro de cada hilo, podemos acceder a los atributos x, y y z de la estructura *blockDim*, que nos dan a conocer el tamaño de los bloques (en hilos) de la dimensión correspondiente. También podemos acceder a los mismos atributos de la estructura *blockIdx* para saber la posición que ocupa el bloque dentro del *grid* en cada una de las dimensiones. En la figura 4.4 se puede observar un bloque declarado con unas dimensiones 2×3.

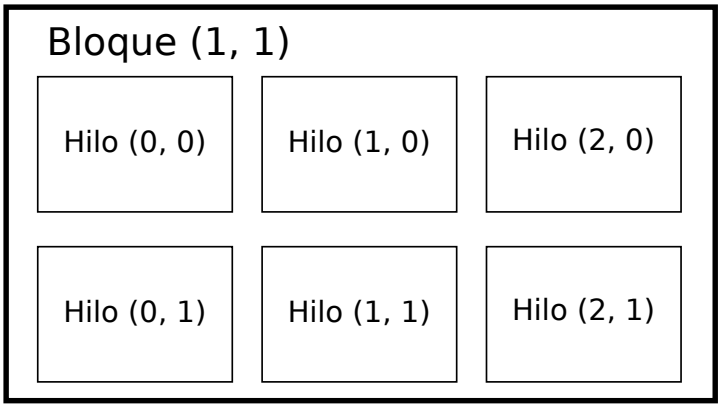

Figura 4.4: Un bloque de dimensiones 2×3.

El hilo

El hilo es el elemento final de CUDA. La ejecución de un conjunto de hilos perteneciente a un mismo bloque es llevada a cabo por un procesador de forma concurrente. Físicamente esto no funciona así, pues en realidad los hilos son agrupados para formar *warps* de 32 hilos. La ejecución de los *warps* creados (cuyo número es limitado por los recursos del procesador) se va alternando dentro de un mismo procesador. Como nuestra intención es no entrar en detalles, asumiremos que todos los hilos de un mismo bloque se ejecutan de forma paralela en un mismo procesador.

Cada hilo ejecuta un *kernel*. Un *kernel* es una función implementada en lenguaje CUDA que aplica alguna transformación sobre la matriz de forma local al hilo. Los hilos también disponen de una estructura *threadIdx* para identificar al hilo dentro de un bloque, accediendo a las coordenadas mediante la consulta de los atributos x, y y z. Con la ayuda de todas estas estructuras, es posible obtener el índice exacto que apunta al elemento de la matriz para el cual se está ejecutando el hilo. Por ejemplo, para una matriz bidimensional:

$$i = blockIdx.x * blockDim.x + threadIdx.x$$

$$j = blockIdx.y * blockDim.y + threadIdx.y$$

Cuando se han ejecutado todos los hilos que se generan a partir de la totalidad de bloques dentro del *grid*, puede considerarse que el *kernel* se ha aplicado de forma exitosa sobre todos los elementos de la matriz.

4.2.2. El lenguaje CUDA

El lenguaje CUDA añade al lenguaje C++ los elementos necesarios para poder escribir programas que interactúen con la GPU de forma exitosa. Estos elementos son básicamente estructuras, operadores y palabras clave. Para compilar un programa escrito con lenguaje CUDA, es necesario utilizar el compilador *nvcc*.

Este lenguaje impone algunas limitaciones en su programación: No se pueden declarar funciones recursivas o con un número variable de parámetros, ni tampoco se pueden utilizar punteros a funciones o variables estáticas dentro del cuerpo de las funciones.

De forma breve, se van a detallar los elementos necesarios para poder crear un sencillo programa CUDA que aplique una transformación sobre una matriz de 2 dimensiones.

La estructura dim3

Las estructura *dim3* es una estructura que contiene tres atributos: x, y y z. Estos atributos pueden ser ajustados accediendo a ellos de forma directa, mediante el operador habitual para acceder a los atributos de una estructura o clase de C++:

```
dim3 num_blocks;

num_blocks.x = 2;
num_blocks.y = 3;
```

No obstante, es posible inicializar estas estructuras mediante un constructor, que admite hasta un máximo de tres parámetros enteros. Los parámetros cuyo valor no sea declarado, serán inicializados a 1 por defecto. En el siguiente ejemplo, x es inicializado a 2, y es inicializado a 3, pero z es inicializado a 1 porque no se ha declarado un valor para esta dimensión:

```
dim3 num_blocks(2, 3);
```

El uso de esta estructura está recomendado para declarar las dimensiones del grid y las dimensiones de los bloques del grid, que de otra forma deberían ser declarados con números enteros, teniendo que pasar como parámetro información sobre como se organiza la matriz en memoria. También será posible identificar, desde un hilo, el bloque en el que nos encontramos, gracias al uso de estas estructuras.

Declaración de kernels

La declaración de *kernels* es muy parecida a la declaración de funciones en C++. A la declaración del tipo de la función, se le ha de anteponer el tipo calificador de la función. Existen tres tipos de calificadores que sirven para especificar si una función es ejecutable desde el *host* (el anfitrión que, en este caso, es la CPU) o el *device* (el dispositivo, refiriéndose a la GPU), y si es posible llamar estas funciones desde el anfitrión o el dispositivo. Estos tipos son:

- **__device__**: Este calificador declara una función que se ejecuta sobre el dispositivo y sólo puede ser llamada desde el dispositivo.
- **__global__**: Este calificador declara una función que se ejecuta sobre el dispositivo y sólo puede ser llamada desde el anfitrión. Este el tipo de calificador empleado para la declaración de *kernels*. Su tipo de retorno sólo puede ser *void*.
- **__host__**: Este calificador declara una función que se ejecuta sobre el anfitrión y sólo puede ser llamada desde el anfitrión. Es el equivalente a declarar una función sin ningún tipo de calificador.

Es posible realizar una combinación entre los calificadores **__host__** y **__device__** si pretendemos compilar la función tanto para el anfitrión como para el dispositivo. Como sólamente nos interesa la declaración de *kernels*, el único calificador que usaremos será **__global__**.

Desde dentro de la función, podemos acceder a los atributos de las estructuras *blockDim*, *blockIdx* y *threadIdx*, para saber las coordenadas del hilo dentro del bloque y el *grid*. Estas estructuras están declaradas de forma local e implícita, por lo que no es necesario utilizar otros mecanismos para orientarnos.

Supongamos que queremos programar un *kernel* sencillo que multiplique por un factor α los elementos de una matriz, Si *stride* es la separación entre dos columnas (CUDA usa la representación *column-major*), entonces el siguiente *kernel* nos serviría:

```
__global__  void scaleMatrix(float alpha,
                             float *matrix,
                             unsigned int stride)
{
```

```
  unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;
  unsigned int j = blockIdx.y * blockDim.y + threadIdx.y;
  unsigned int index = i * stride + j;

  matrix[index] = alpha * matrix[index];
}
```

Como puede observarse, podemos calcular el índice del elemento para el cual se lanza el hilo, obteniendo primero los índices i y j. Luego sólo hace falta multpilicar i por el *stride* y le sumamos j para obtener el índice.

Ejecución de kernels

La llamada de los *kernels* se efectúa de forma especial. Es necesario interponer los operadores <<< y >>> entre el nombre del *kernel* y la lista de parámetros para especificar las dimensiones del *grid* y los bloques. Por ejemplo, para llamar al *kernel* que acabamos de mostrar, con un *grid* de dimensiones 2×2 y bloques de dimensiones 2×3, podemos usar el siguiente trozo de código:

```
dim3 grid(2, 2);
dim3 block(2, 3);
unsigned int stride = 6;

scaleMatrix<<<grid, block>>>(2.0, (float *)matrix_gpu, stride);
```

El *kernel* se aplicará sobre los elementos de una matriz de dimensiones 4×6 de forma exitosa después de la ejecución de esta llamada.

Un ejemplo de uso de CUDA

Ahora que ya hemos repasado todos los elementos básicos de la programación en CUDA, vamos a mostrar el ejemplo completo utilizando el kernel y las estructuras que se han ido proponiendo a lo largo de esta sección:

```
#include <cstdio>
#include <cstdlib>
#include <cuda.h>

// Dimensiones de la matriz (M x N)
#define M 4
#define N 6

__global__  void scaleMatrix(float alpha, float *matrix)
{
  unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;
  unsigned int j = blockIdx.y * blockDim.y + threadIdx.y;
```

```
  unsigned int index = i * N + j;

  matrix[index] = alpha * matrix[index];
}

int main(int argc, char* argv[])
{
  float *matrix = (float *) malloc (sizeof(float) * (M * N));
  CUdeviceptr matrix_gpu;
  CUresult result;
  CUdevice cuDevice;
  CUcontext cuContext;

  // Inicializacion de los datos de la matriz
  for (int i = 0; i < M; i++)
    for (int j = 0; j < N; j++)
      matrix[i * N + j] = 2.0f;

  // Las siguientes instrucciones son de inicializacion
  result = cuInit(0);
  if (result != CUDA_SUCCESS) printf("Init error\n");
  result = cuDeviceGet(&cuDevice, 0);
  if (result != CUDA_SUCCESS) printf("Device error\n");
  result = cuCtxCreate(&cuContext, 0, cuDevice);
  if (result != CUDA_SUCCESS) printf("Context error\n");

  // Reservamos memoria GPU y copiamos la matriz en ella
  result = cuMemAlloc(&matrix_gpu, sizeof(float) * (M * N));
  if (result != CUDA_SUCCESS) printf("Allocate error\n");
  result = cuMemcpyHtoD(matrix_gpu, matrix, sizeof(float) * (M * N));
  if (result != CUDA_SUCCESS) printf("CopyHtoD error\n");

  // Declaramos las dimensiones
  dim3 grid(2, 2);
  dim3 block(2, 3);

  // Ejecutamos el kernel
  scaleMatrix<<<grid, block>>>(2.0, (float *)matrix_gpu);

  // Copiamos la memoria de la GPU a memoria principal
  result = cuMemcpyDtoH(matrix, matrix_gpu, sizeof(float) * (M * N));
  if (result != CUDA_SUCCESS) printf("CopyDtoH error\n");

  // Liberamos la memoria
  cuMemFree(matrix_gpu);

  return 0;
```

```
}
```

En programas que procesan matrices de un tamaño sólamente conocido en tiempo de ejecución, las dimensiones del *grid* y los bloques deben ajustarse según estos parámetros. Este código nos permite procesar una matriz de dimensiones arbitrarias:

```
dim3 block(min(M, 16), min(N, 16));
dim3 grid(M / block.x + (M % block.x) ? 1 : 0),
          N / block.y + (N % block.y) ? 1 : 0));

scaleMatrix<<<grid, block>>>(2.0, (float *)matrix_gpu);
```

El resto de la división entre las dimensiones totales de la matriz y el número de bloques nos indican que hemos de añadir un bloque para procesar los valores restantes al final de cada dimensión. Es por esto que también se recomienda realizar una comprobación de los índices i y j que se obtiene en el *kernel* para no alterar zonas de memoria que no pertenecen a la matriz que estamos modificando.

4.2.3. Algoritmos de reducción

Si bien el uso de operaciones matriciales es el cuello de botella de la aplicación, limitar el uso de la GPU a este tipo de cálculos supondría tener que mover las activaciones de la memoria GPU a memoria principal y viceversa. De ahí surge el interés de implementar la lógica de las activaciones o las funciones de error como *kernels* CUDA.

Además, en muchas aplicaciones prácticas de este *toolkit* se requieren topologías con salidas de tipo *softmax* con decenas e incluso cientos de miles de salida (por ejemplo, en tareas de modelado del lenguaje conexionista), por lo que también resulta importante realizar este cálculo eficientemente. Por motivos de estabilidad numérica, la función de activación *softmax* requiere calcular el máximo, el mínimo y la suma del conjunto de neuronas sobre las que se aplique. Estas operaciones son fáciles de implementar de forma secuencial, aunque puede resultar extremadamente ineficiente hacerlo de este modo.

Estos operadores binarios pueden implementarse de forma paralela, mejorando la eficiencia respecto a la ejecución secuencial, mediante los denominados algoritmos de reducción [HSJ86, SHZO07]. Las operaciones de reducción son aquellas en las cuales una misma operación binaria asociativa (como la suma, el mínimo, el producto) se aplica reiteradamente sobre un conjunto de datos hasta obtener un único resultado.

En este proyecto se han implementado algoritmos de reducción para calcular sumatorios, mínimos y máximos de forma paralela sobre un *bunch*, ya que éstos no suelen aplicarse sobre un conjunto de vectores sino sobre un único vector.

Estos algoritmos asumen que el vector sobre el cual se va a aplicar el operador tiene un número de componentes que es potencia de dos. Los resultados se van acumulando sobre los valores del mismo vector, de forma que al final del algoritmo

podemos consultar el primer componente para saber el resultado total. Todos estos factores nos obligaron a implementar nuestros propios *kernels* para aplicar la reducción, ya que no siempre tendremos vectores de talla potencia de dos ni tampoco queremos modificar los datos de la matriz al aplicarla.

Los algoritmos de reducción aplican el operador sobre dos componentes del vector cuya distancia entre ellos es la mitad de la longitud del vector. El resultado de aplicar ese operador se guarda en el miembro que se encuentra más cerca del inicio del vector. Luego, se vuelve a aplicar la reducción sobre la primera mitad del vector del mismo modo. Este proceso termina cuando sólo queda un valor por reducir.

A continuación se muestra una traza para reducir un vector de 8 componentes, en donde el operador aplicado es la suma:

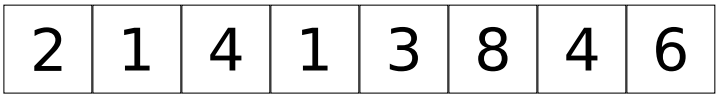

Figura 4.5: Traza de reducción (1/7). Estado inicial del vector.

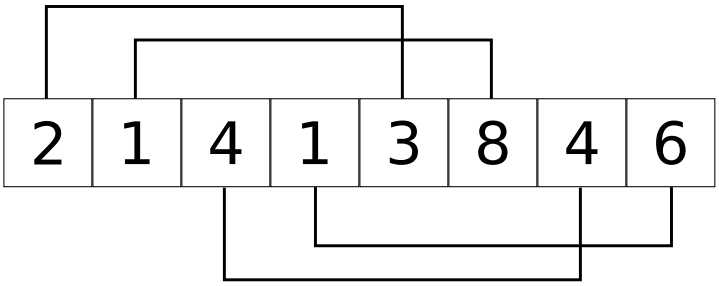

Figura 4.6: Traza de reducción (2/7). Se consultan los valores a reducir para el vector con 8 componentes.

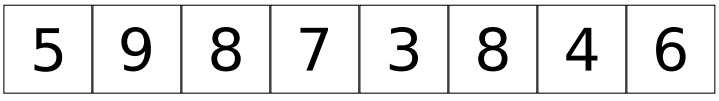

Figura 4.7: Traza de reducción (3/7). Se guardan los valores resultantes de aplicar la suma en las 4 primeras componentes.

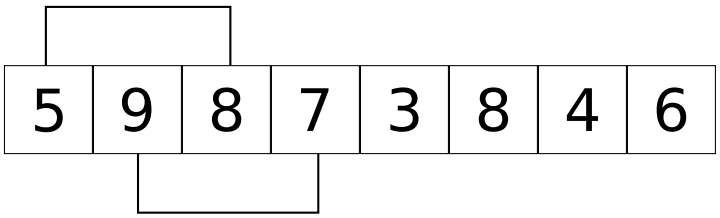

Figura 4.8: Traza de reducción (4/7). Se consultan los valores a reducir para el vector con 4 componentes.

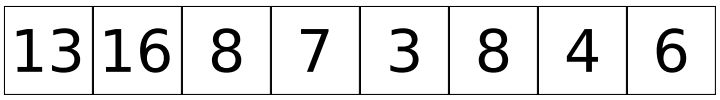

Figura 4.9: Traza de reducción (5/7). Se guardan los valores resultantes de aplicar la suma en las 2 primeras componentes.

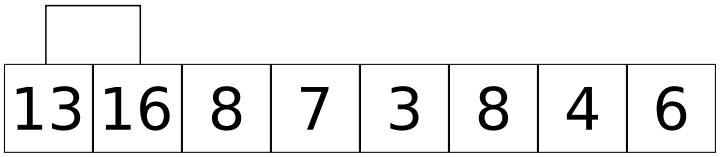

Figura 4.10: Traza de reducción (6/7). Se consultan los valores a reducir para el vector con 2 componentes.

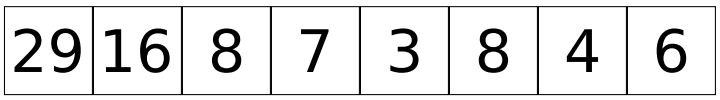

Figura 4.11: Traza de reducción (7/7). Estado final del vector, el valor total de la suma es el que se guarda en el primer componente.

De esta manera, un algoritmo que tendría un coste lineal al ser ejecutado de forma secuencial, logra tener un coste logarítmico al ser ejecutado de forma paralela.

La traza que se ha mostrado es muy sencilla. En nuestro proyecto, se mantiene la esencia del algoritmo, aunque ha sido necesario adaptarlo a nuestras necesidades para no sobreescribir valores o trabajar con vectores cuya talla no es potencia de dos.

4.3. La librería CUBLAS

La librería CUBLAS es una biblioteca que implementa funciones parecidas a las de la API de BLAS en lenguaje CUDA, para poder realizar cálculos de álgebra lineal empleando la GPU para ello [Cor12d].

La librería CUBLAS también contiene las funciones necesarias para poder trabajar con la GPU, relacionadas con la inicialización y apagado del entorno, la gestión de memoria, copia de zonas de memoria entre memoria principal y memoria de la GPU y viceversa. Estas funciones no son más que conjuntos de instrucciones CUDA.

4.3.1. La clase CublasHelper

La clase *CublasHelper* es un clase estática diseñada para controlar el encendido y apagado del entorno CUDA, además de realizar otras gestiones. La clase sólo es compilada si se realiza una compilación con CUDA, al igual que lo es la única instancia de esta clase.

La clase implementa dos funciones para inicializar y apagar el entorno CUDA, realizando las comprobaciones necesarias antes de cada acción y emitiendo diálogos de error si algo va mal. La nueva API de CUBLAS exige que el primer parámetro que se pase a sus funciones sea un *handle* (un objeto de la clase *cublasHandle_t* que declara CUBLAS), el cual se incluye dentro de esta clase y se dispensa a aquellas funciones que lo necesiten para su ejecución.

Además de estas funcionalidades, la clase *CublasHelper* está preparada para gestionar el uso de múltiples GPU y múltiples hilos de ejecución en una GPU (llamados *streams*).

4.4. El bloque de memoria

Con la adición de la tecnología CUDA al proyecto se presentan dificultades a la hora de mantener un código simple y claro, por lo que se propuso realizar una abstracción sobre las zonas de memoria principal y de la GPU para poder trabajar sobre ellas de una forma más cómoda, sin tener que preocuparse por la gestión o las transferencias de memoria.

Estos bloques de memoria se implementan en la clase *GPUMirroredMemoryBlock*, que es una *template class* que proporciona al desarrollador la opción de trabajar con los tipos de datos *float* o *double*, pues cabe la posibilidad de que en el futuro alguien pretenda ampliar la aplicación para conseguir un mayor grado de precisión en los cálculos, ahorrando así el esfuerzo extra de replicar esta clase para trabajar con el tipo de dato *double*.

La clase *GPUMirroredMemoryBlock* representa un bloque de memoria general, con un atributo que representa la zona de memoria reservada en memoria principal y otro atributo representando la zona de memoria del mismo tamaño localizada en la memoria de la GPU, tal y como se ilustra en la figura 4.12. La construcción de este bloque de memoria implica una reserva de memoria principal alineada.

Idealmente, estas zonas de memoria alamacenan los mismos valores para los elementos que contienen, aunque al implementarlas se adopta una estrategia de actualización *lazy*, de forma que se copia el contenido de la zona de memoria que estábamos manipulando a la zona de memoria a la que solicitamos acceso (ya sea para lectura o escritura) si ésta lo requiere.

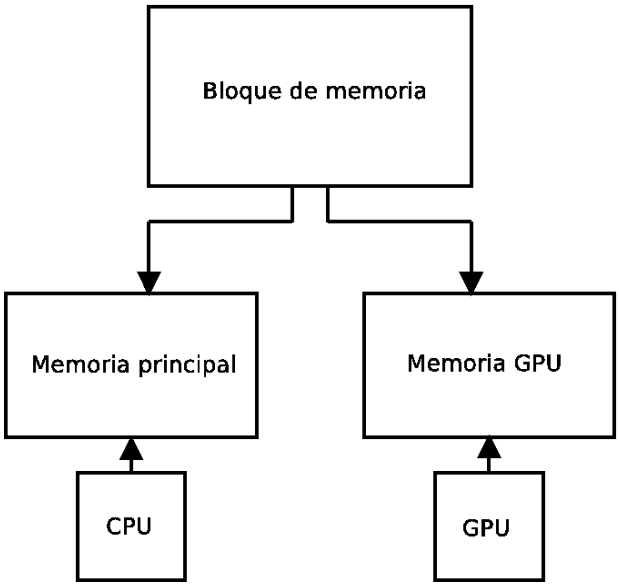

Figura 4.12: Diagrama del bloque de memoria.

Por ejemplo, si llegamos a un punto en el que queremos usar la GPU para efectuar algún cálculo, tendremos que utilizar la función del bloque de memoria que nos entrega un puntero a una zona de memoria GPU. El bloque se encargará, en ese momento, de revisar si esa zona de memoria GPU está actualizada. Si no lo está, realizará la copia de memoria principal a memoria GPU de forma automática. Esto sucede cuando, hasta el momento, se estaba usando la CPU para realizar los cálculos. En caso contrario, la memoria GPU será la memoria con la que se ha trabajado hasta el momento, por lo que se nos devolverá el puntero a la zona de memoria con la que ya estábamos trabajando anteriormente.

4.5. Wrappers para el álgebra lineal

Los *wrappers* para el álgebra lineal se presentan como una herramienta complementaria a los bloques de memoria descritos en la anterior sección.

Un *wrapper* se define como una función cuyo objetivo final es el de llamar a una segunda función con objeto de ofrecer a quien la llama una vista más simple y resolver problemas de interfaz (compatibilidad entre bibliotecas). Estos *wrappers* reemplazan las llamadas a funciones de BLAS para poder trabajar con ellas de forma independiente al componente utilizado para efectuar los cálculos de álgebra lineal.

Los *wrappers* se encargan de solicitar los datos necesarios para realizar los cálculos a los bloques de memoria, que facilitan estos datos actualizados en la memoria en la cual se vayan a emplear (que puede ser memoria principal o memoria de la GPU). Más tarde, los mismos *wrappers* se encargan de llamar a las funciones correspondientes para realizar las operaciones de álgebra lineal necesarias.

El código de los *wrappers* utiliza directivas de precompilación para generar un código óptimo y sencillo en función del modo de compilación (presentados en la

siguiente sección). De esta forma se consigue realizar una abstracción total sobre el tipo de memoria e instrucciones para realizar cálculos de álgebra lineal, lo cual permite al desarrollador trabajar sin necesidad de preocuparse por las librerías empleadas al compilar la aplicación.

El siguiente fragmento de código, que forma parte de la versión final de la aplicación, es el *wrapper* diseñado para ejecutar la llamada a la función *sscal* correspondiente. Téngase en cuenta que *USE_ CUDA* sólo está definido para las compilaciones con CUDA:

```
void doSscal(unsigned int size,
             float alpha,
             FloatGPUMirroredMemoryBlock *x,
             unsigned int shift,
             unsigned int inc,
             bool use_gpu) {
  float *x_mem;
#ifdef USE_CUDA
  if (use_gpu) {
    cublasStatus_t status;
    cublasHandle_t handle = CublasHandler::getHandler();
    x_mem = x->getGPUForReadAndWrite() + shift;

    status = cublasSscal(handle, size, &alpha, x_mem, inc);

    checkCublasError(status);
  }
  else {
#endif
    x_mem = x->getPPALForReadAndWrite() + shift;
    cblas_sscal(size, alpha, x_mem, inc);
#ifdef USE_CUDA
  }
#endif
}
```

4.6. Compilación y ejecución

Este proyecto presenta varias opciones de compilación y ejecución para su uso, ya que se presenta la posibilidad de que un usuario de la aplicación no disponga de una tarjeta gráfica compatible con CUDA pero sí quiera emplear la versión que trabaja con librerías que implementan las funciones de BLAS mediante el uso de la CPU.

Por tanto, existen varias alternativas a la hora de compilar la aplicación:

- Compilación con ATLAS: Tan sólo se necesita tener instalada la librería ATLAS para compilar el proyecto de este modo. La aceleración conseguida

al ejecutar la aplicación es la menor de entre las tres opciones, pero también es la alternativa más fácil de compilar dado que la instalación de la librería ATLAS puede realizarse cómodamente a través de los repositorios de cualquier distribución de *GNU/Linux*.

Esta es también la opción que deberíamos escoger para compilar en caso de que se desconozca el *hardware* en el cual se va a emplear la aplicación o no se disponga de los componentes requeridos para trabajar con ella en cualquiera de las siguientes opciones.

- Compilación con Intel MKL: El *software* requerido para realizar este tipo de compilación incluye la librería Intel MKL, que al contrario que la librería ATLAS, no está disponible en los repositorios de las distribuciones de *GNU/Linux*, sino que es necesario descargar la librería desde la página web del fabricante, teniendo que registrarse previamente en ella para conseguir un código de instalación.

  Este modo de compilación es el recomendado para trabajar con máquinas que dispongan de un procesador de la marca Intel, que consigue acelerar notablemente la aplicación mediante el uso intensivo de la CPU. También se nos permite emplear más de un núcleo de la GPU, por lo que el tiempo de ejecución disminuirá todavía más en caso de indicar por la línea de comandos que se desea utilizar más de un núcleo de la GPU al ejecutar la aplicación.

- Compilación con CUDA e Intel MKL: La compilación con CUDA e Intel MKL requiere tener instalada la biblioteca necesaria para el anterior tipo de compilación, además de la versión 4.2 del *toolkit* de CUDA. Del mismo modo que sucedía con la biblioteca de Intel, también tendremos que visitar la página web de Nvidia para poder descargar el paquete de instalación del *toolkit* de CUDA, aunque no es necesario el registro para realizar la descarga.

  Una vez instalada y compilada la aplicación, será necesario disponer de una tarjeta gráfica Nvidia compatible con CUDA y un procesador de la marca Intel para poder hacer uso de ella. Es necesario indicar de manera explícita, mediante el script creado para describir los experimentos, que se desea emplear la GPU al realizar la ejecución. Esto se consigue poniendo a *true* el flag para usar CUDA. En caso de no indicarlo, la aplicación utilizará la CPU para efectuar los cálculos, como si se hubiese compilado utilizando el anterior modo de compilación.

Capítulo 5

Uso de la aplicación

En esta sección se explican de forma general las herramientas con las que se trabaja para la realización de experimentos con April. El objetivo de estas herramientas es facilitar la descripción de experimentos, no siendo necesario el preprocesamiento previo de los datos o la construcción de las redes mediante un editor de redes externo.

5.1. Matrices y datasets

Los conjuntos de datos son representados matricialmente mediante un objeto denominado *matrix*. Este objeto permite la creación de matrices de datos a partir de ficheros o tablas *Lua*. Estas matrices no están limitadas dimensionalmente, por lo que es posible crear objetos *matrix* con un número cualquiera de dimensiones.

Los conjuntos de datos se definen con la ayuda de los datasets. Es posible definir un dataset a partir de una matriz con un número cualquiera de dimesiones en la cual podemos hacer el recorrido de un fragmento arbitrario mediante la especificación de datos como: La posición inicial, el orden del recorrido, la longitud entre un patrón y el siguiente, etc. También es posible emplear operaciones sobre conjuntos para obtener, por ejemplo, nuevos conjuntos de datos a partir de la unión o la partición. Esto permite al diseñador describir conjuntos de datos de un modo sencillo y flexible.

De esta forma, los datasets ofrecen un interfaz general para representar un corpus o un conjunto de patrones de una misma dimensionalidad, aunque el dataset más relevante utiliza una matriz y permite iterar sobre submatrices de la misma especificando un versátil conjunto de parámetros. Existe toda una biblioteca de estilo funcional para construir nuevos datasets que resulta de gran importancia para describir de manera sencilla y eficiente un gran número de tareas reales sobre imágenes, secuencias, documentos... Este tipo de facilidades resulta muy importante a nivel pragmático para un uso eficiente de la aplicación.

5.2. Generador de números aleatorios

El objeto *random* nos permite obtener instancias de generadores de números aleatorios a partir de una semilla dada. Un generador de estas características es necesario para poder asegurar la reproducibilidad de los experimentos, sin la cual no sería posible la comparación de algunos experimentos como los realizados en este proyecto. Además de esto, el objeto *random* nos permite guardar el estado actual del generador en disco duro, lo cual facilita la reanudación de experimentos que, de otra forma, deberían ser ejecutados desde el principio hasta alcanzar el estado guardado y continuar su ejecución.

5.3. Uso de AllAllMLP

Para construir un MLP desde Lua es necesario crear tanto el objeto MLP (correspondiente al mismo en C++) como las diversas componentes que contiene (capas de neuronas, conexiones, etc.). Para simplificar el uso de un tipo de red utilizado muy frecuentemente (las redes con conexiones capa a capa) se dispone de una subclase de MLP denominada *AllAllMLP*.

La clase *AllAllMLP*, que representa un MLP en el cual las capas se relacionan de forma que todas las neuronas de la capa de entrada están conectadas a todas las neuronas de la capa de salida, puede ser instanciada en los *scripts* de experimentación para crear un objeto de esta clase. El método *generate*, al cual podemos darle una serie de parámetros para especificar las características de la red, construye la red con esos parámetros.

Los datasets, que ya han sido generados a partir de las *matrices* de datos, pueden ser pasados como paramétros a las funciones de la red diseñadas para el entrenamiento y la validación.

5.4. Ejemplo de uso

La descripción de los experimentos se hace con el lenguaje de *scripting* Lua [IdFC03, lua03]. La ejecución de estos *scripts* es posible ya que previamente se realiza un *binding* a Lua de las clases programadas en C++. De esta forma, no se depende de más elementos que el programa compilado y el *script* que describe el experimento para experimentar. De manera opcional, podemos usar *scripts* de *bash* para ejecutar un mismo experimento con diferentes configuraciones, ya que es posible el paso de parámetros a los *scripts* en Lua.

A continuación se va a mostrar un ejemplo de uso de April para entrenar un problema de Reconocimiento óptico de carácteres o "Optical Character Recognition" (OCR). Esta tarea consiste en la clasificación de dígitos manuscritos normalizados a 16×16 píxeles en blanco y negro. El corpus está formado por 1000 dígitos y se almacena en una única imagen en formato Portable Network Graphics (PNG). Esta imagen ordena los dígitos en 100 filas, de forma que cada columna contiene 100 ejemplos de un mismo dígito del 0 al 9 (véase la figura 5.1).

Como la red recibe un dígito como entrada, la capa de entrada estará formada por un total de 256 neuronas (tamaño total en píxeles de las imágenes). La

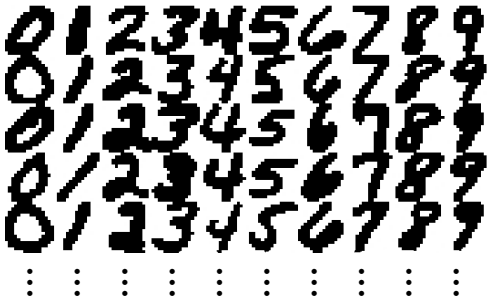

Figura 5.1: Fragmento de la imagen digits.png que contiene 100 filas de 10 muestras, cada una de ellas formada por 16×16 píxeles.

capa de salida estará formada por 10 neuronas (una neurona por cada clase). Los experimentos para este problema suelen utilizar una o dos capas ocultas de tamaños no mayores de 512 neuronas, pero dada la potencia de cálculo que nuestra aplicación ofrece, vamos a construir la capa oculta con 1024 neuronas.

En este ejemplo de uso se emplean cuatro datasets para generar:

- Un conjunto de muestras de entrada para el entrenamiento. Este conjunto se compone de las primeras 80 filas de dígitos de la imagen cargada.
- Un conjunto de muestras de entrada para la validación. Este conjunto se compone de las últimas 20 filas de dígitos de la imagen cargada.
- Un conjunto de muestras de salida deseadas para el entrenamiento. La generación de este conjunto se realiza directamente a partir de una *matrix* declarada en el script (*m2*).
- Un conjunto de muestras de salida deseadas para la validación. Al igual que el anterior conjunto, éste también se genera a partir de la matriz *m2*.

Una vez aclarados estos conceptos, ya podemos presentar el *script* utilizado para la experimentación con esta tarea:

```
-- Establecimiento de parametros
-- El bunch sera 64 si no se pasa como argumento
bunch_size    = tonumber(arg[1]) or 64
semilla       = 1234
aleat         = random(semilla) -- Un generador de numeros aleatorios
description   = "256 inputs 1024 tanh 10 softmax" -- Topologia
inf           = -0.1
sup           =  0.1
otrorand      = random(5678)
 -- Parametros relacionados con el entrenamiento
learning_rate = 0.01
momentum      = 0.0
weight_decay  = 0.0
```

```
max_epochs    = 10

-----------------------------------------------------------------

-- Carga de la matriz de pixels
m1 = matrix.loadImage("digits.png", "gray")

-- Generamos el conjunto de entrenamiento a partir de la matriz
train_input = dataset.matrix(m1,
           {
             patternSize = {16,16}, -- Tamanyo de cada muestra
             offset      = {0,0},   -- Desplazamiento inicial
             numSteps    = {80,10}, -- Numero de pasos en cada direccion
             stepSize    = {16,16}, -- Tamany de cada paso
             orderStep   = {1,0}    -- Direccion del paso
            })

-- Generamos el conjunto de validacion a partir de la matriz
val_input  = dataset.matrix(m1,
          {
            patternSize = {16,16},
            offset      = {1280,0},
            numSteps    = {20,10},
            stepSize    = {16,16},
            orderStep   = {1,0}
          })

-- Esta matriz es pequenya, por lo que la cargamos directamente
m2 = matrix(10,{1,0,0,0,0,0,0,0,0,0})

-- El conjunto de salidas deseadas se genera a partir de m2
train_output = dataset.matrix(m2,
             {
        patternSize = {10},
        offset      = {0},
        numSteps    = {800},
        stepSize    = {-1},  -- Desplazamos la ventana hacia la izquierda
        circular    = {true} -- Es un dataset circular
             })

-- De forma similiar, generamos ahora para validacion
val_output   = dataset.matrix(m2,
             {
        patternSize = {10},
        offset      = {0},
        numSteps    = {200},
        stepSize    = {-1},
```

```
        circular    = {true}
            })

-- Creamos la red con los parametros dados
lared = ann.mlp.all_all.generate{
  bunch_size  = bunch_size,
  topology    = description,
  random      = aleat,
  inf         = inf,
  sup         = sup,
}

-- Enpaquetamos los datos para entrenar y validar
datosentrenar = {
  input_dataset  = train_input,
  output_dataset = train_output,
  shuffle        = otrorand,
}

datosvalidar = {
  input_dataset  = val_input,
  output_dataset = val_output,
}

-- Configuramos el entrenamiento
lared:set_option("learning_rate", learning_rate)
lared:set_option("momentum",      momentum)
lared:set_option("weight_decay",  weight_decay)

lared:set_error_function(ann.error_functions.mse())

totalepocas = 0

clock = util.stopwatch()
clock:go()

-- Un bucle que para cuando se alcanza max_epochs
print("Epoch Training  Validation")
for epoch = 1,max_epochs do
  totalepocas = totalepocas+1
  -- Entrenamos y validamos
  errortrain  = lared:train_dataset(datosentrenar)
  errorval    = lared:validate_dataset(datosvalidar)
  -- Imprimimos la epoca y los errores
  printf("%4d  %.7f %.7f\n",
         totalepocas,errortrain,errorval)
end
```

```
clock:stop()
cpu,wall = clock:read()

-- Imprimimos las medidas de tiempo para el entrenamiento
printf("Wall total time: %.3f    per epoch: %.3f\n", wall, wall/max_epochs)
printf("CPU  total time: %.3f    per epoch: %.3f\n", cpu, cpu/max_epochs)

-- Guardamos la red
ann.mlp.all_all.save(lared, "ejemplo.net", "ascii", "old")
```

La salida obtenida al ejecutar la aplicación compilada con la librería ATLAS es el siguiente:

```
Epoch Training  Validation
   1  0.6388747 0.5456210
   2  0.7807434 0.6630745
   3  0.5221108 0.1941401
   4  0.1985256 0.1110188
   5  0.0531882 0.0840450
   6  0.0381000 0.0650603
   7  0.0700443 0.0551373
   8  0.0344651 0.0283038
   9  0.0443663 0.0431527
  10  0.0214009 0.0773187
Wall total time: 4.358    per epoch: 0.436
CPU  total time: 7.392    per epoch: 0.739
```

Como puede observarse, la red se entrena en 10 iteraciones consiguiendo un error final del 2.14 % para las muestras de entrenamiento y un 7.73 % para las muestras de validación.

El *script* también guarda el tiempo Wall y el tiempo CPU, total y por época, empleados en la ejecución del algoritmo. El valor numérico del tiempo Wall representa el tiempo según la percepción humana. El tiempo CPU es la parte de ese tiempo que el procesador ha estado activo (ejecutando instrucciones). Si se está usando más de un núcleo del procesador, este valor se expresa como la suma de los tiempos de cada núcleo activo.

Capítulo 6

Experimentación

En este capítulo van a mostrarse los resultados de varios experimentos realizados con esta herramienta. Éstos se realizarán con todos los modos de compilación disponibles, comprobando primero que la implementación del algoritmo BP y sus variantes sea correcta. Luego, se procederá a compararar la eficiencia de las distintas *builds* (los programas resultantes de un modo de compilación determinado) para resolver distintos problemas. Finalmente, se muestran dos estudios adicionales que justifican la inclusión del factor weight decay y el desarrollo del algoritmo con CUDA.

6.1. Criterios de comparación

El lector podrá comprobar que no se comparan los resultados experimentales con otras herramientas comunes del ámbito de las redes neuronales como Stuttgart Neural Network Simulator (SNNS), *Weka* o Fast Artificial Neural Network Library (FANN) [ZMH+95, wek99, Nis03].

Esto ocurre por muchas razones, siendo la principal de ellas que nuestro proyecto está diseñado para sacar el máximo partido al modo *bunch*, mientras que la mayoría de estas herramientas suelen trabajar en modo *on-line* (procesando sólamente una muestra por cada iteración del BP). Estos dos modos funcionan de formas distintas, por lo que cualquier tipo de comparación no sería justo. Además, algunos factores como el *momentum* o el *weight decay* no son incorporados en la mayoría de implementaciones del BP, ya que suponen un gran esfuerzo de diseño y programación si se pretende conseguir una herramienta eficaz. En este proyecto estos factores sí que se consideran y además su implementación ha resultado ser eficiente.

También es conveniente recordar que algunas de estas herramientas (SNNS, por ejemplo) utilizan valores de tipo *double* para la representación de los número reales, en contraposición a April, que trabaja con valores de tipo *float*.

Herramientas como SNNS y *Weka* dejan mucho que desear en lo que eficiencia se refiere. SNNS se plantea como una herramienta para usos didácticos y su antigüedad se empieza a notar en las máquinas actuales. *Weka*, en cambio, es una aplicación más reciente, y aunque su rendimiento sea aceptable, no se puede

decir que su implementación esté optimizada para la experimentación con redes neuronales.

En las versiones iniciales de la herramienta April ya se mostraba un factor de aceleración superior a 8 respecto a SNNS y un rendimiento similar al obtenido con FANN [Zam05].

6.2. Parámetros experimentales

En esta sección se exponen los componentes de la máquina con la cual se han realizado los experimentos. También se especifican los programas y otros recursos necesarios para la programación, compilación y ejecución de esta herramienta. Después se introducen las tareas a resolver en las distintas partes de la experimentación con la herramienta.

6.2.1. Hardware utilizado

El ordenador utilizado para compilar la aplicación y ejecutar los experimentos se compone de:

- Un procesador Intel Core i7-870 funcionando a 2.93 GHz.
- 8 GBs de memoria RAM.
- Una tarjeta gráfica nVidia GeForce 9800 GTX+, con 128 núcleos CUDA.

6.2.2. Software empleado

Las versiones de las librerías y otros programas que permiten la compilación y ejecución de esta herramienta son:

- Sistema operativo *GNU/Linux* de 64 bits.
- Versión 4.6 del compilador *gcc*.
- Versión 4.2 del compilador *nvcc*.
- Versión 4.2 de la librería CUDA.
- Versión 4.2 de la librería CUBLAS.

6.2.3. La tarea xor

Esta tarea ya se ha mostrado en el capítulo 2, y es una tarea clásica de las redes neuronales introducida en el libro [MS69] para demostrar que el problema no es resoluble sin el uso de capas ocultas en las redes neuronales. Se compone de 4 muestras de entrenamiento (todas las posibles combinaciones de 2 *bits* binarios) y el entrenamiento hace que la red aproxime una función parecida a la función *xor*.

6.2.4. La tarea dígitos

La tarea *dígitos* también se ha introducido en el capítulo 5. El corpus está formado por 1000 dígitos manuscritos en blanco y negro, normalizados a 16×16 píxeles, de forma que se tienen 100 muestras para cada dígito. Las redes que pretendan resolver este problema deberán tener un total de 256 neuronas de entrada y 10 neuronas de salida (una por cada dígito).

6.2.5. La tarea MNIST

La tarea *MNIST* [LC98] es parecida a la tarea *dígitos*, se basa en dígitos manuscritos en blanco y negro, aunque el número de muestras que contiene es mucho mayor. Este corpus está formado por 60000 muestras para el entrenamiento, siendo cada una de estas muestras una imagen de dimensiones 28×28 píxeles. Además de estas muestras, también se facilitan otras 10000 muestras como conjunto de test. El corpus *MNIST* es una versión reducida y modificada de dos de las bases de datos publicadas por el National Institute of Standards and Technology (NIST).

6.3. Corrección de las implementaciones

Las pruebas de corrección se han hecho en base a la versión de April previa a este proyecto. Esta versión ya garantizaba su corrección inspirándose en el artículo de J. Gibb y L. Hamey [GH95] para comprobar el buen funcionamiento del algoritmo BP y sus variantes, definir una serie de tests y probarlos con SNNS y April.

Lo que se ha hecho fue comprobar que los resultados para la *build* con ATLAS fuesen parecidos a los resultados obtenidos para esta versión previa con las tareas *xor* y *dígitos*. Luego, los resultados de la *build* con ATLAS fueron comparados con los de la *build* con MKL, y esta última *build* fue empleada para verificar la corrección de los resultados de la *build* con CUDA.

6.3.1. Resultados para xor

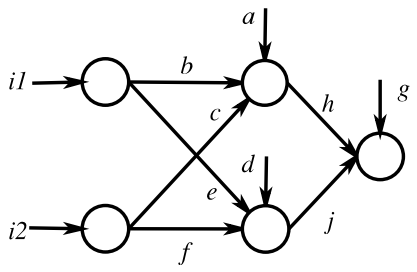

Figura 6.1: Red para resolver el problema de la xor.

La primera prueba consiste en entrenar una red con 2 neuronas en la capa de entrada, 2 neuronas en la capa oculta y 1 neurona en la capa de salida. En la figura

6.1 podemos observar una ilustración que representa la topología de esta red, en donde también se etiquetan los pesos con una letra del abecedario. La función de activación utilizada en las capa oculta y de salida es la *tangente hiperbólica*. La función de error de la red es la función MSE. Los pesos iniciales son los mostrados en el cuadro 6.1, en donde se relaciona la etiqueta del peso con su valor inicial.

Peso	Valor inicial
a	-0.5
b	-1.2
c	1.0
d	-2.0
e	4.0
f	-4.0
g	-1.0
h	2.0
j	2.0

Cuadro 6.1: Pesos iniciales de la red de la figura 6.1.

Para el entrenamiento, se fijan los siguientes parámetros:

- learning rate(η) = 0.4
- momentum(μ) = 0.1
- weight decay(α) = 10^{-5}

El criterio de parada para este entrenamiento consiste en procesar 30000 ciclos completos de todas las muestras, con un *bunch* que agrupa 64 muestras de entrenamiento.

Los resultados obtenidos después de realizar el entrenamiento utilizando la *build* con ATLAS, con MKL y con CUDA se muestran en los cuadros 6.2, 6.3 y 6.4.

Peso	Valor final
a	-1.74390077590942
b	-3.16053938865662
c	3.13186645507812
d	-2.60031056404114
e	4.35672664642334
f	-4.29912185668945
g	6.63821458816528
h	4.37812709808350
j	4.13424348831177

i_1	i_2	Activación de la salida
0	0	0.00021910667419434
0	1	0.98837041854858
1	0	0.98946917057037
1	1	0.00018572807312012

Cuadro 6.2: Pesos y activaciones finales para la *build* de ATLAS.

Puede observarse que cada librería nos proporciona unos resultados diferentes al final de los experimentos. Esto ocurre porque cada una de ellas tiene un modo

Peso	Valor final
a	−1.74440097808838
b	−3.16140174865723
c	3.13308095932007
d	−2.60152125358582
e	4.35702896118164
f	−4.30008983612061
g	6.63708400726318
h	4.37579965591431
j	4.13343763351440

i_1	i_2	Activación de la salida
0	0	0.00021946430206299
0	1	0.98836177587509
1	0	0.98946464061737
1	1	0.00018608570098877

Cuadro 6.3: Pesos y activaciones finales para la *build* de MKL.

Peso	Valor final
a	−1.74588787555695
b	−3.16202616691589
c	3.13313961029053
d	−2.60048341751099
e	4.35337018966675
f	−4.29543352127075
g	6.63744926452637
h	4.37538480758667
j	4.13293027877808

i_1	i_2	Activación de la salida
0	0	0.00021898746490479
0	1	0.98834645748138
1	0	0.98943901062012
1	1	0.00018680095672607

Cuadro 6.4: Pesos y activaciones finales para la *build* de CUDA.

diferente de operar con el tipo *float*, que varía en función de cómo estén implementadas las operaciones de este tipo, ya sea a nivel *hardware* o a nivel *software*. Esta imprecisión no tiene por qué afectar negativamente al entrenamiento, sino que en ocasiones pueden observarse resultados mejores para un mismo entrenamiento al utilizar las *builds* más sofisticadas.

6.3.2. Resultados para dígitos

En esta prueba se entrena una red mucho más compleja que la del problema *xor*. En el capítulo 5 ya se experimentó con el mismo corpus, aunque la red era diferente (antes teníamos una única capa oculta de 1024 neuronas y parámetros diferentes). Esta nueva red se compone de una capa de entrada con 256 neuronas, a la que siguen dos capas ocultas con 256 y 128 neuronas, cuya función de activación para ambas es la *tangente hiperbólica*. La última capa se compone de 10 neuronas, y su función de activación es la *softmax*. La función de error de la red es la función MSE.

Los valores de los parámetros son asignados con $\eta = 0.01$, $\mu = 0.01$ y $\alpha = 10^{-5}$. En este entrenamiento, se realiza un número de ciclos completos igual a 10. Para cada ciclo (también conocido como época), se imprimen los errores que se obtienen con el conjunto de entrenamiento y el conjunto de validación según la función de

error de la red (un error menor será mejor). El valor para el parámetro *bunch* se ha ajustado a 32.

La salida producida al ejecutar el *script* de dígitos con los parámetros especificados y la *build* generada al compilar con ATLAS es la siguiente:

```
Epoch Training  Validation
   1  0.3923072 0.2862245
   2  0.1836092 0.1210741
   3  0.0765181 0.0765358
   4  0.0460878 0.0588698
   5  0.0366127 0.0522985
   6  0.0247615 0.0345044
   7  0.0216336 0.0322793
   8  0.0181125 0.0253723
   9  0.0159277 0.0294236
  10  0.0138948 0.0398200
Wall total time: 2.321    per epoch: 0.232
CPU  total time: 3.032    per epoch: 0.303
```

Al ejecutar el mismo *script*, pero con la *build* de MKL, se obtienen los siguientes resultados:

```
Epoch Training  Validation
   1  0.3923072 0.2862245
   2  0.1836092 0.1210741
   3  0.0765181 0.0765358
   4  0.0460878 0.0588698
   5  0.0366127 0.0522986
   6  0.0247615 0.0345044
   7  0.0216336 0.0322793
   8  0.0181125 0.0253723
   9  0.0159277 0.0294236
  10  0.0138948 0.0398200
Wall total time: 0.307    per epoch: 0.031
CPU  total time: 1.216    per epoch: 0.122
```

Y si utilizamos la *build* con CUDA:

```
Epoch Training  Validation
   1  0.3923073 0.2862249
   2  0.1836097 0.1210745
   3  0.0765184 0.0765361
   4  0.0460880 0.0588701
   5  0.0366129 0.0522987
   6  0.0247617 0.0345046
```

```
   7  0.0216338 0.0322796
   8  0.0181127 0.0253726
   9  0.0159279 0.0294239
  10  0.0138950 0.0398202
Wall total time: 0.264    per epoch: 0.026
CPU  total time: 0.216    per epoch: 0.022
```

Como puede observarse, los resultados son muy parecidos para todas las *builds*.

6.4. Rendimiento

Para las pruebas de rendimiento se utiliza el corpus *MNIST*. Este corpus es adecuado para poner a prueba nuestra aplicación, ya que el entrenamiento para resolver este problema es temporalmente muy costoso.

Este experimento se realiza en 3 pasos, los cuales nos permitirán elegir un valor adecuado para los parámetros del entrenamiento (entre los cuales se incluye el *bunch*), un número de neuronas en las capas ocultas que nos garanticen un entrenamiento con una buena tasa de error y un número eficiente de hilos para ejecuciones paralelas del mismo experimento. Estos 3 pasos se describen a continuación:

1. El experimento se parametriza para los valores que definen el entrenamiento. El número de neuronas para la primera y la segunda capa oculta se fijará a 256 y 128 neuronas, respectivamente. La función de activación de las capas ocultas es la *tangente hiperbólica*, y la de la capa de salida es *softmax*. La función de error de la red es la función de entropia cruzada.

   Para el valor del *bunch*, se prueban los valores 1, 4, 8, 16, 32, 64, 96, 128, 256, 512 y 1024. El factor de aprendizaje variará entre los valores 0.01, 0.02 y 0.04. El parámetro momentum tendrá 2 posibles valores: 0.0 y 0.02. De esta forma, se examina el efecto que produce la inclusión de este factor en el entrenamiento, asignando un valor común en caso de que este se incluya. Para el parámetro weight decay ocurre lo mismo: Su valor será 0.0 ó 10^{-6}. En esta parte del experimento ya se plantean un total de 132 subexperimentos (396 si se realiza una comparativa entre *builds*), que surgen de combinar todos los posibles valores para estos 4 parámetros.

   De estas pruebas, extraemos el tiempo Wall que han tardado en ejecutarse. También extraeremos el error conseguido en validación. Después de analizar los resultados obtenidos, escogeremos un experimento cuyo compromiso entre tiempo y error sea de los mejores.

2. Los valores de los parámetros del experimento destacado en la primera parte serán fijados para esta parte del experimento.

   El objetivo ahora será conseguir un número de neuronas en las capas ocultas que favorezca el entrenamiento con esos parámetros. El número de neuronas que pondremos en la primera capa oculta será 32, 64, 128, 256, 512 y

1024. Para la segunda capa oculta, se emplearan 0, 32, 64, 128 y 256 neuronas. De esta forma, con el valor 0 en la segunda capa, se está probando al mismo tiempo si nos beneficia tener una segunda capa oculta para los valores posibles del número de neuronas en la primera capa. El número de combinaciones posibles para este experimento es 30.

Una vez finalizados los experimentos, se revisa el error de validación y el error de test conseguidos en cada uno de ellos. De estos resultados, se escogen los valores de las capas cuyo experimento haya tenido los mejores resultados tanto en error de test como en error de validación.

3. En esta parte del experimento fijamos los valores de los parámetros ya fijados para ejecutar la segunda parte experimental y también fijamos el número de neuronas de las capas escogido en el paso anterior.

   Ahora probaremos cómo funciona la aplicación cuando son lanzadas una o varias instancias del mismo problema al mismo tiempo. La variable de entorno *OMP_NUM_THREADS* puede fijarse en las *builds* que compilen con MKL para limitar el número de procesadores que la aplicación puede utilizar en su ejecución. En los anteriores experimentos, este valor era 4, por lo que se utilizaban todos los núcleos de la CPU y la aplicación mejoraba su rendimiento.[1]

   Vamos a iterar sobre el valor que puede tomar esta variable con 1, 2 y 4. Para cada uno de estos valores, se lanzarán 1, 2 y hasta 4 instancias de la ejecución. Nótese que esta parte del experimento no podrá ser efectuada por la *build* con ATLAS, ya que no es posible ajustar el número de núcleos para su ejecución.

Como puede observarse, éste puede considerarse un experimento muy completo. No sólo se realiza un barrido de parámetros y topologías adecuados para nuestro problema, sino que además se puede valorar la productividad de la aplicación al lanzar varios hilos de ejecución.

6.4.1. Comparación de eficiencia

Primera parte

Los resultados de la primera parte experimental han sido procesados y condensados en las gráficas 6.2 y 6.3.

La gráfica 6.2 muestra los errores obtenidos al clasificar el conjunto de validación después del entrenamiento respecto al valor del parámetro *bunch*, para cada una de las *builds* compiladas. La gráfica inferior muestra los mismos resultados pero para un rango más estrecho, que nos permita analizar la calidad de los mejores resultados.

Algunas de las observaciones que se pueden hacer al analizar la gráfica 6.2 son las siguientes:

[1] La CPU utilizada dispone de 4 núcleos.

- El error mínimo obtenido se sitúa alrededor del 1.7 %, Muchos de los experimentos han obtenido resultados cercanos a esta marca.
- La mayoría de los valores de error en validación obtenidos se sitúan entre el 2 % y el 3 %, lo que significa que el entrenamiento se ha realizado con éxito para la mayoría de estos experimentos.
- Para valores de *bunch* superiores a 128, los errores en validación obtenidos son muy altos, por lo que se descarta la elección de alguno de estos valores de *bunch* para la próxima etapa, ya que podrían producir inestabilidades numéricas.
- Cuando sólo es procesada una muestra por iteración (*bunch* igual a 1) los resultados de error en validación son más dispersos que en los otros casos (pueden observarse errores repartidos entre el 2 % y el 10 %).
- Estos datos son insuficientes para decidirnos por un valor concreto. Se deberán analizar los resultados temporales para encontrar aquel experimento que mejor relación entre el error obtenido y el tiempo invertido.

La gráfica 6.3 muestra los tiempos por época obtenidos al ejecutar los experimentos respecto al valor del parámetro *bunch*, para cada una de las *builds* compiladas. Las conclusiones que se sacan después de analizarla son:

- Los mejores resultados están alrededor de los 1.2 segundos y todos ellos son conseguidos por la *build* de CUDA para los casos en que el *bunch* es mayor o igual a 32.
- Las configuraciones óptimas (es decir, aquellas que tienen un buen compromiso entre error de validación y tiempo por época) son aquellas para las cuales el valor del *bunch* está entre 64 y 256.
- Los resultados para la *build* de CUDA suelen estar concentrados en el mismo punto, mientras que los resultados para MKL o ATLAS están más dispersos.
- Para la *build* con ATLAS, se puede comprobar que la ejecución del modo *bunch* disminuye el tiempo invertido cuando mayor el número de muestras procesadas por iteración.

Después de haber analizado estas gráficas, decidimos escoger un valor para el *bunch* que estuviese entre el 64 y 128 (los errores en validación para valores superiores no eran convincentes, como tampoco lo eran los tiempos por época para valores inferiores). De entre estos valores, escogimos el menor de ellos (64), con tal de mantener una mejor estabilidad numérica en las siguientes etapas del experimento.

La mejor configuración (para error en validación) para un *bunch* de 64 era:

- learning rate(η) = 0.02
- momentum(μ) = 0.02

- weight decay(α) = 10^{-6}

Por lo que estos serán los valores que fijaremos para la siguiente parte del experimento.

Segunda parte

Las gráficas 6.4, 6.5 y 6.6 presentan los errores obtenidos en los conjuntos de validación y test para esta tarea, en función del número de neuronas de la primera capa. Las líneas relacionan los resultados obtenidos con el número de neuronas de la segunda capa.

Se puede observar que:

- Las *builds* ATLAS y MKL obtienen unos resultados muy parecidos.
- La *build* con CUDA parece obtener resultados bastante peores para ciertos experimentos.
- A primera vista, los resultados con una sola capa con muchas neuronas (512 y 1024) son los más favorables. No obstante, la *build* con CUDA obtiene sus peores resultados con estos mismos valores.
- En general, los mejores resultados se encuentran con 256 ó 128 neuronas en la primera capa y 128 ó 64 neuronas en la segunda capa.

Para la siguiente parte experimental fijaremos los tamaños a 256 neuronas para la primera capa oculta y 128 para la segunda capa oculta, ya que la diferencia de tiempo por época empleado para ejecutar el experimento es despreciable.

Tercera parte

Las gráficas 6.7 y 6.8 nos muestran el tiempo Wall[2] y el tiempo CPU[3] por época al ejecutar los experimentos con los parámetros anteriormente fijados, en función del valor OMP_NUM_THREADS con el que se hayan lanzado (de ahora en adelante, nos referiremos a este parámetro como OMP). Como ya se ha explicado anteriormente, esta parte experimental solamente es válida para las *builds* con MKL y CUDA, ya que sólo éstas permiten regular el número de núcleos para su ejecución.

Después de observar cómo la regularización del parámetro OMP y el lanzamiento de varios procesos en paralelo afectan al tiempo por época, podemos decir que:

- Es casi irrelevante regular el número de núcleos de la CPU que la tarea debe utilizar en el caso de que la aplicación se compile y ejecute con CUDA. Esto es lógico, ya que los cálculos son delegados a la GPU y el procesador solamente se encarga de crear los objetos y llamar las funciones necesarias, además de realizar las transferencias de memoria a la GPU.

[2]El tiempo real aproximado que transcurre al ejecutarse experimento.

[3]La suma de los tiempos de cada núcleo activo del procesador.

- El tiempo CPU es la suma de los tiempos que cada núcleo ha invertido en la ejecución de instrucciones CPU, por lo que su incremento se ve afectado más notablemente al analizar el tiempo CPU que el tiempo Wall.
- Para un único hilo en ejecución, el incremento del parámetro OMP mejora notablemente el tiempo Wall empleado, llegando a necesitar sólo la mitad de tiempo para ejecutarse si fijamos este parámetro a 4 en lugar de a 1.

  Esto significa que la CPU logra paralelizar algunas instrucciones de forma automática al ejecutarse con más de un núcleo, por lo que se hace uso de todos los núcleos en algunas partes de la ejecución.
- Para más de un hilo en ejecución, en cambio, el ajuste del parámetro sí que puede ser perjudicial, como cabría esperar. Esto sucede porque la CPU no puede repartir las tareas entre los núcleos de una manera efectiva, y ha de determinar intervalos de tiempo para la ejecución concurrente de estas tareas.

  La máquina utilizada para los experimentos tiene 4 núcleos. Podemos deducir esto a través de la gráfica, ya que para OMP = 2 y 2 hilos en ejecución, el tiempo Wall empleado es menor que con OMP = 1. Esto se debe a que el procesador es capaz de asignar 2 núcleos a cada tarea, sin causar una sobrecarga por tener que repartir las tareas entre los núcleos. En cambio, si ponemos OMP = 4, la ejecución de estos 2 hilos cuesta más tiempo, ya que el la CPU ha de ir dando "turnos" a cada tarea para que se pueda ejecutar con los 4 núcleos de los que se disponen.

El estudio de estas gráficas no aporta una información suficiente como para decidirse por una configuración u otra, pues no se ha valorado el trabajo que realizan éstas. Por tanto, hemos de plantear la resolución de una tarea cuyos experimentos puedan ser ejecutados de manera concurrente, y valorar si nos conviene lanzar las ejecuciones de forma secuencial o no según su productividad.

La productividad se valora dividiendo el tiempo por época empleado en la ejecución de un experimento (secuencial o concurrente) por el número de trabajos ejecutados a la vez. En el cuadro pueden observarse los valores de productividad de la herramienta.

Como puede observarse, si necesitamos ejecutar un número elevado de trabajos parecidos al que hemos ido diseñando nosotros a lo largo de este experimento, se recomienda lanzar los trabajos de 4 en 4 fijando la variable OMP a 1. Para obtener el modo de ejecución que más nos beneficia a partir de un procesador concreto, se recomienda elaborar un cuadro similar al cuadro 6.5 y escoger aquella configuración cuya productividad sea menor.

Conclusión

Llegados a este punto, hemos resuelto la tarea *MNIST* con muy buenos resultados cualitativos y temporales.

Los valores fijados en cada parte del experimento han sido los mismos para poder comparar las diferencias entre las distintas *builds* compiladas. Esto significa que algunas de estas *builds* podrían haberse beneficiado más si se hubiesen

Valor de OMP_NUM_THREADS	Hilos concurrentes	Tiempo por época	Productividad por época
1	1	4.017	4.017
2	1	2.671	2.671
4	1	2.062	2.062
1	2	4.266	2.133
2	2	2.893	1.4465
4	2	3.496	1.748
1	4	4.733	1.18325
2	4	5.158	1.2895
4	4	8.074	2.0185

Cuadro 6.5: Productividad de la herramienta al ejecutar varios experimentos de forma concurrente.

utilizado aquellos parámetros que las favorecían a ellas, pero entonces no hubiese sido posible compararlas entre ellas.

El usuario de la aplicación debería seguir un proceso experimental parecido para resolver sus tareas, de forma que se realice:

1. Un barrido de parámetros que beneficie al error de clasificación y minimicen el tiempo invertido en los experimentos.
2. Un barrido de topologías que beneficie al error de clasificación.
3. Un barrido de configuraciones de ejecución que minimice el tiempo invertido en los experimentos.

Los conjuntos a clasificar en cada una de las fases quedan a elección del usuario. En este experimento se han utilizado errores de validación en la primera y segunda parte, y errores de test en la segunda parte, pero es el usuario quien finalmente decide los criterios de calidad, del mismo modo que selecciona los criterios de parada al diseñar un experimento.

6.4.2. Estudio sobre ejecuciones concurrentes

Este estudio pretende ampliar el estudio realizado en la tercera parte del experimento sobre rendimiento. En esa parte, se han realizado ejecuciones concurrentes para una misma *build*, pero no se ha estudiado qué ocurre cuando se ejecutan los mismos experimentos con *builds* diferentes al mismo tiempo.

En esta sección vamos a ver el efecto que produce ejecutar la *build* con MKL sobre el experimento diseñado para la tercera parte del estudio sobre el rendimiento. Se ejecutará el programa dos veces, de manera que:

- Esta ejecución se realizará primero sin que ningún otro proceso influya en su rendimiento.
- La segunda ejecución se realizará de forma concurrente a otra instancia del mismo experimento para la *build* con CUDA.

La ejecución concurrente de dos experimentos similares con las *builds* de MKL y CUDA nos permitirán valorar la productividad de la herramienta al ejecutar al mismo tiempo dos experimentos usando la CPU y la GPU. El número de núcleos de la CPU que la aplicación pueda usar será fijado a 4. Los resultados son los mostrados en el cuadro 6.6.

Tipo de ejecución	Tiempo total empleado	Productividad total
Ejecución única	78.325	78.325
Ejecución concurrente	97.131	48.5655

Cuadro 6.6: Estudio de la productividad conseguida al ejecutar el mismo experimento en CPU y GPU de forma concurrente.

La productividad conseguida es mucho mejor que la de una ejecución concurrente de dos experimentos usando la *build* con MKL. Esto es lógico, ya que podemos considerar la GPU como un procesador aparte que no está siendo usando. El incremento del tiempo total empleado para realizar la tarea se debe a que la GPU no es totalmente independiente de la CPU, y por tanto, es necesario ejecutar algunas operaciones, como la llamada de funciones o la tranferencia de datos, con la CPU.

6.5. El efecto weight decay

El weight decay es un parámetro cuya inclusión en las herramientas típicas para entrenar redes neuronales no es usual. No obstante, algunos estudios reportan que la adición del weight decay en los entrenamientos mejora los valores obtenidos al clasificar cualquier conjunto de muestras, lo cual hemos querido comprobar por nosotros mismos.

Con el objetivo de estudiar el efecto producido por el weight decay en las ejecuciones del algoritmo BP, vamos a revisar los experimentos realizados en la primera parte del experimento anterior y vamos a contar aquellas configuraciones para las cuales el weight decay beneficia el error en validación.

El proceso que se ha seguido para realizar este conteo es el siguiente:

1. Se han reunido los valores del error en validación conseguidos en la primera parte del experimento anterior para la *build* con ATLAS, correspondientes a un total de 132 configuraciones.

2. A continuación, se han formado las parejas de valores conseguidas al procesar los experimentos cuyos valores de *bunch*, factor de aprendizaje y momentum son los mismos.

3. De estas 66 parejas, el primer valor de error en validación corresponde al experimento realizado sin weight decay. El segundo valor corresponde al mismo experimento, pero con un weight decay igual a 10^{-6}.

4. Ahora analizamos los resultados: Si el primer valor es superior a un umbral, entonces se examina el segundo valor. Si ese valor también es mayor que

el umbral, entonces la inclusión del weight decay no afecta al experimento. En caso contrario, lo beneficia.

5. Si ninguno de los dos valores es superior a ese umbral, se busca el valor mínimo. Si el valor mínimo resulta ser el segundo, entonces la inclusión del weight decay ha beneficiado el experimento. En caso de que el mínimo sea el primero, su inclusión ha producido el efecto inverso y ha empeorado el entrenamiento.

Este proceso ha sido realizado con un umbral del 20 %, obteniendo los resultados mostrados en el cuadro 6.7.

Efecto producido	Número de experimentos
Mejora	28
Empeora	13
Es indiferente	25
Total	66

Cuadro 6.7: Estudio del efecto que el weight decay produce sobre los experimentos de la primera parte.

Los resultados dejan claro que la inclusión del parámetro suele beneficiar la ejecución de nuestros experimentos. Los experimentos para los cuales esta inclusión es irrelevante son aquellos cuyos parámetros tienen un valor extremo y producen que los pesos de la red neuronal acaben siendo anormales, con lo que se obtienen errores de clasificación superiores al umbral declarado.

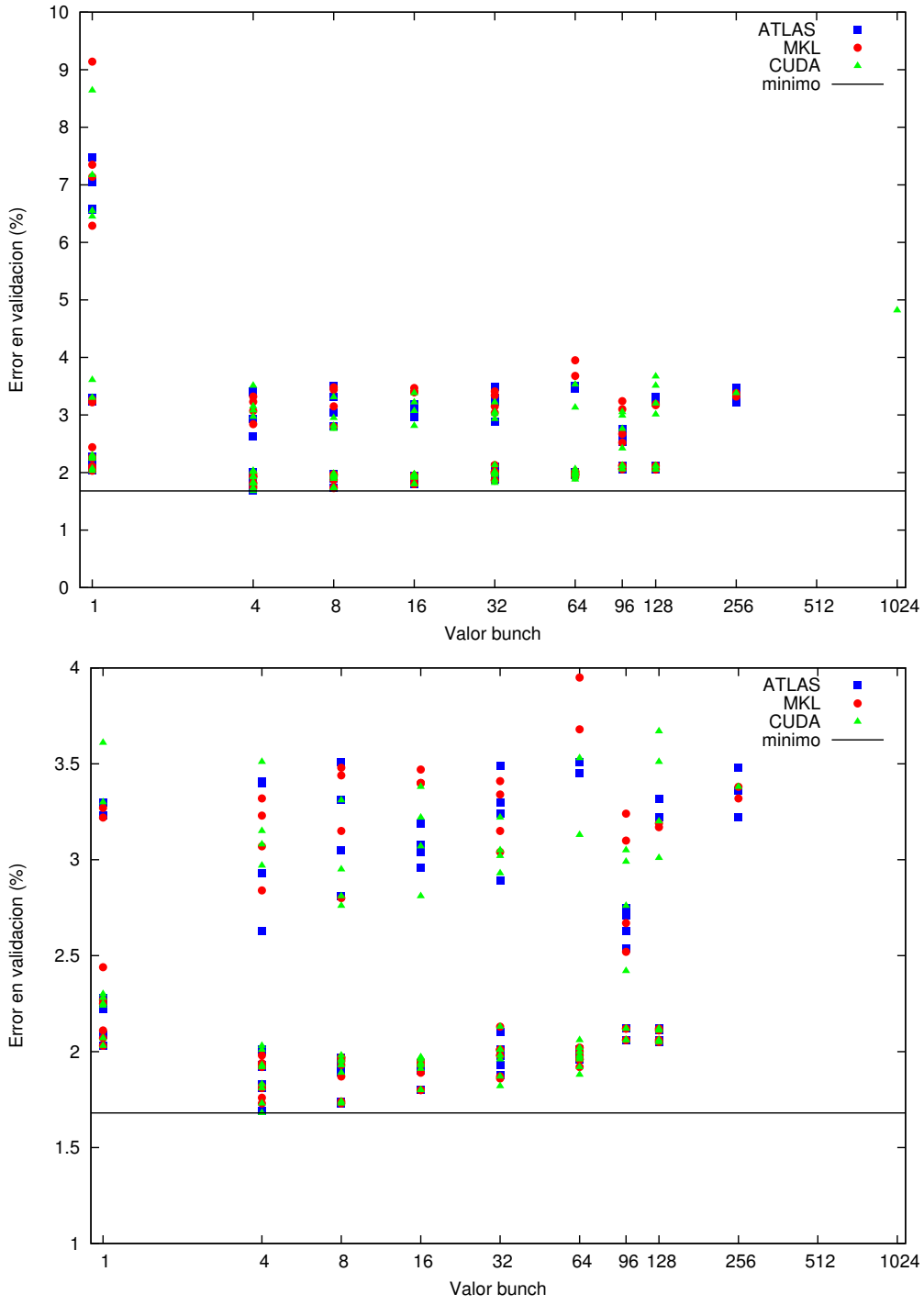

Figura 6.2: Errores de validación obtenidos en función del valor *bunch*.

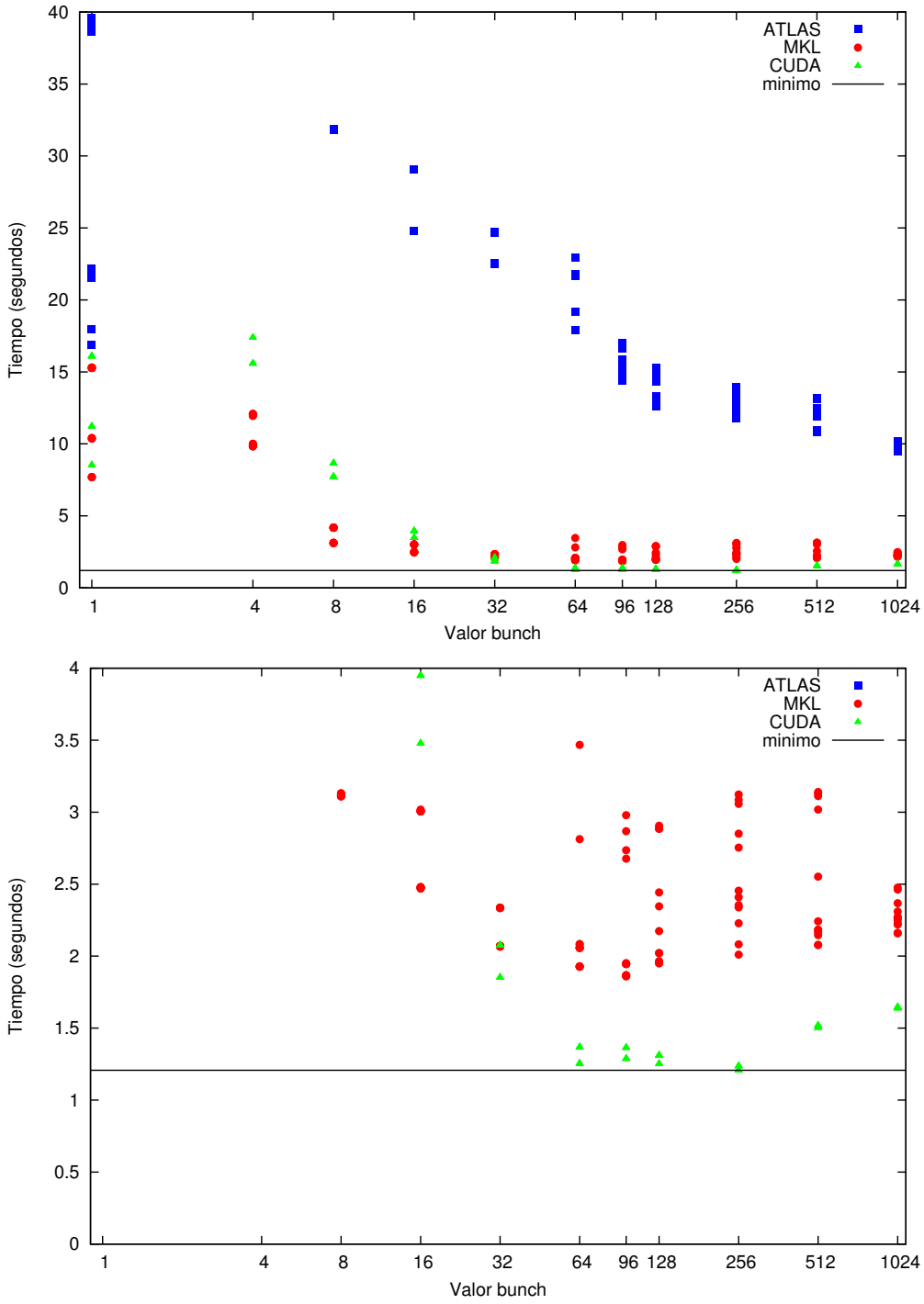

Figura 6.3: Tiempos por época invertidos en la ejecución en función del valor *bunch*.

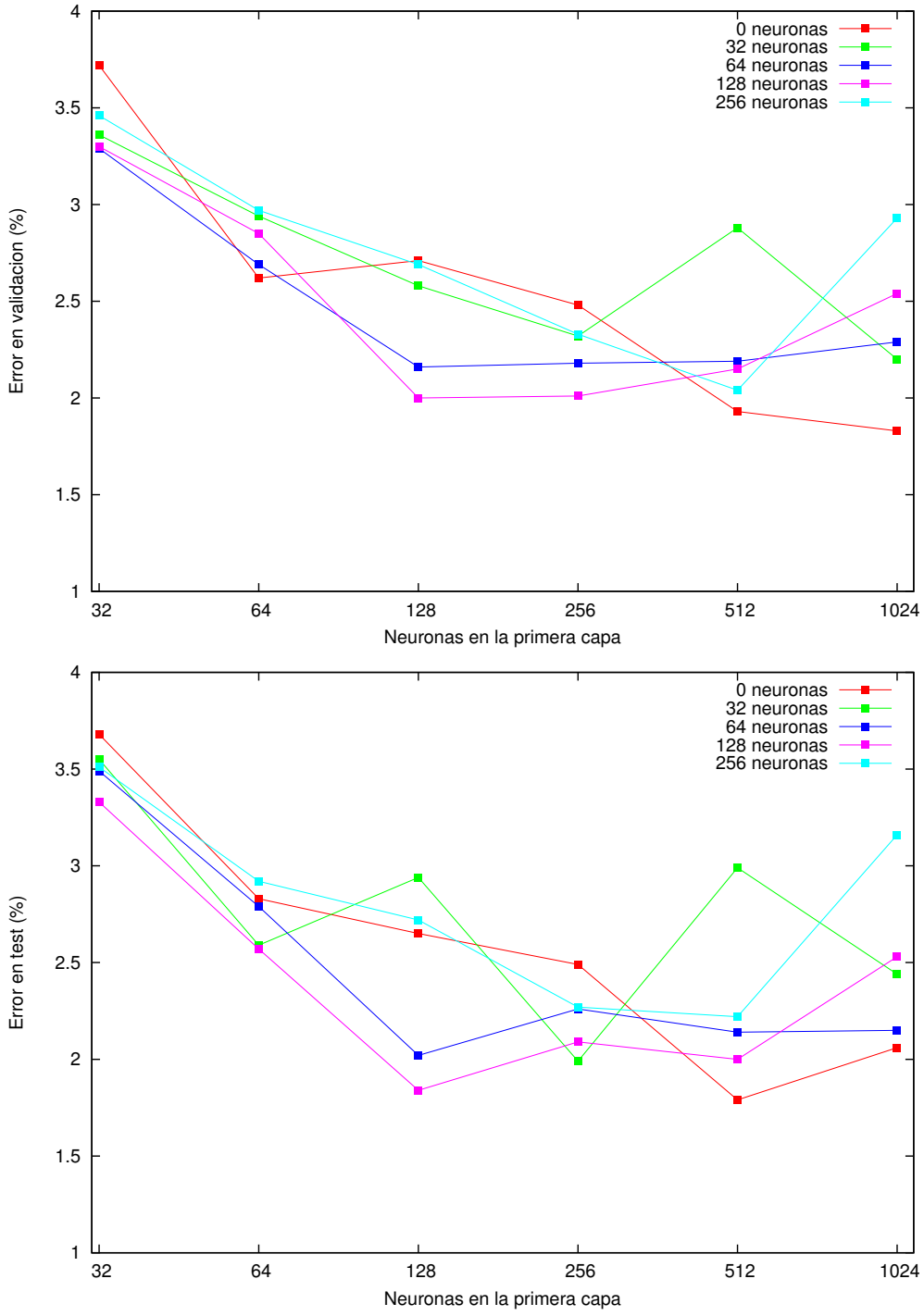

Figura 6.4: Errores de validación y test obtenidos en función del número de neuronas de la primera y segunda capa, para la *build* con ATLAS.

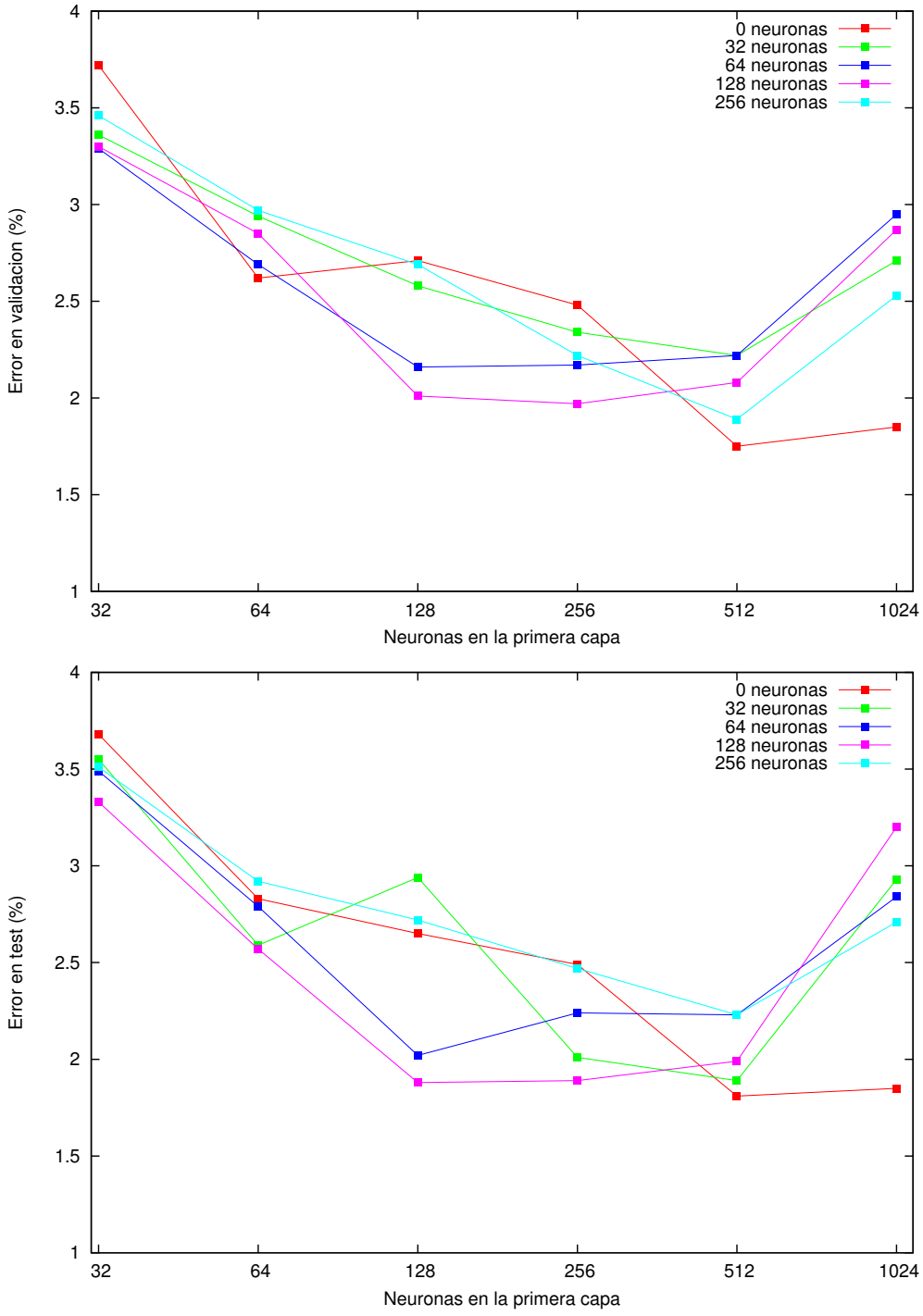

Figura 6.5: Errores de validación y test obtenidos en función del número de neuronas de la primera y segunda capa, para la *build* con MKL.

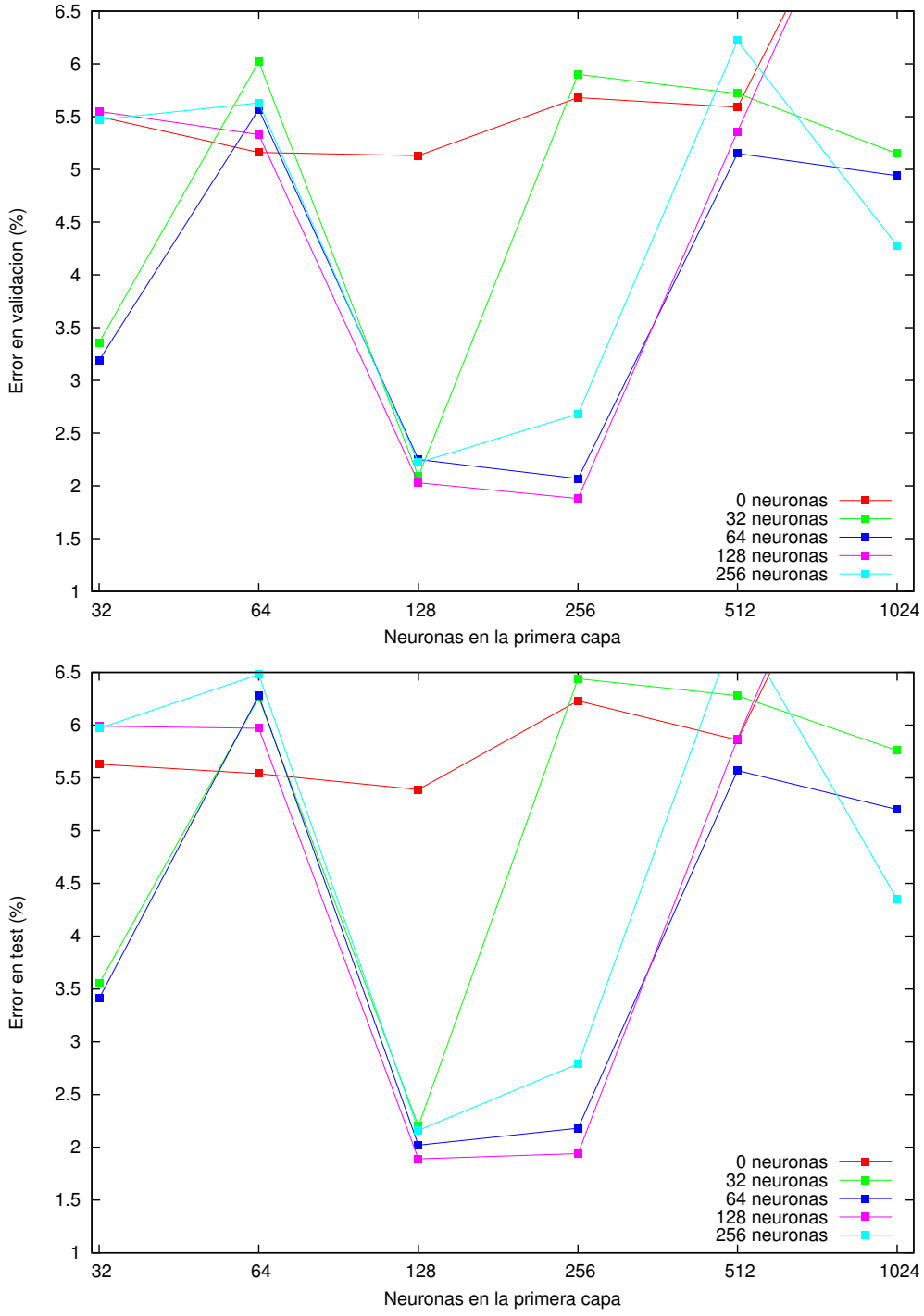

Figura 6.6: Errores de validación y test obtenidos en función del número de neuronas de la primera y segunda capa, para la *build* con CUDA.

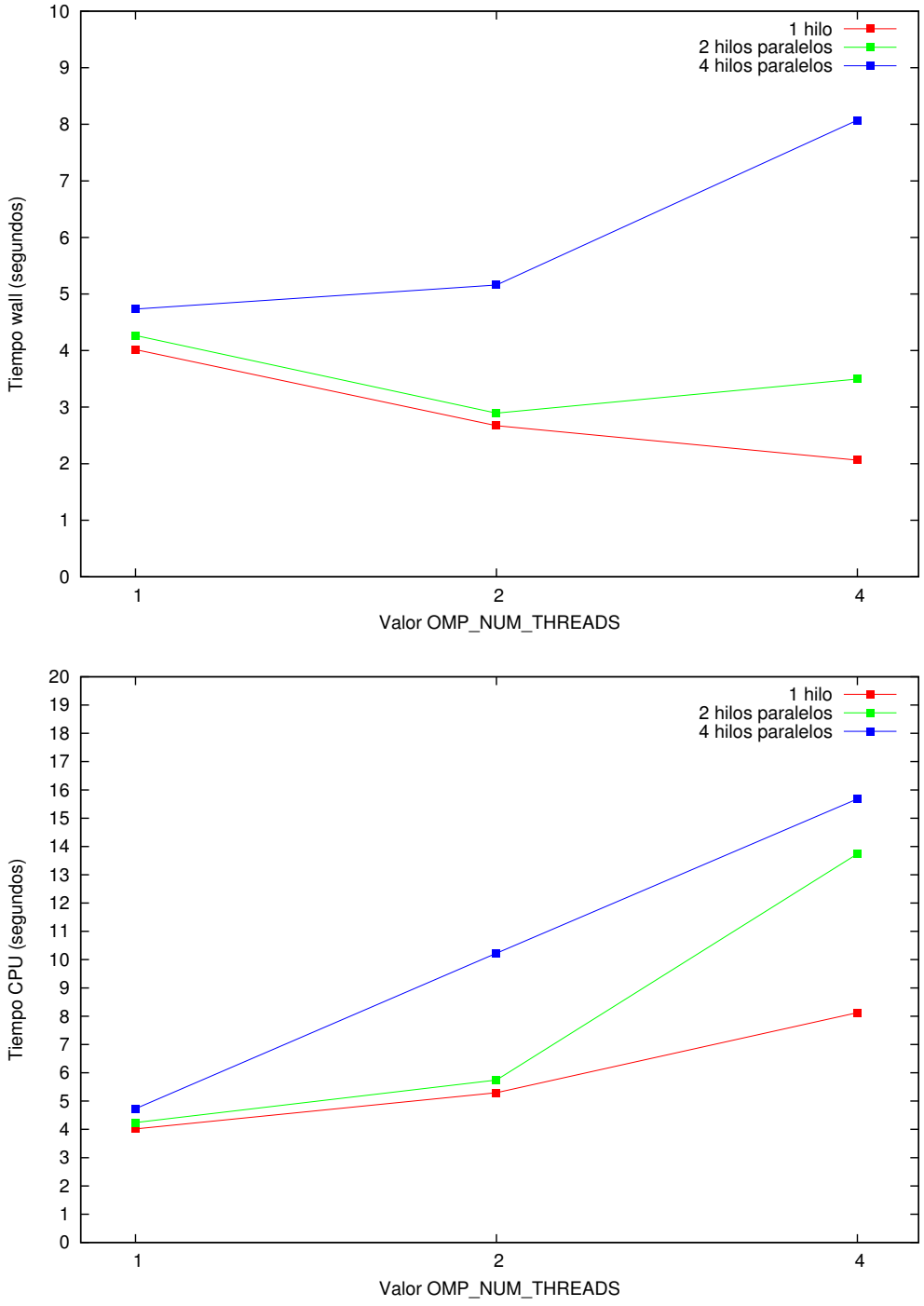

Figura 6.7: Tiempos Wall y CPU por época invertidos en la ejecución en función del parámetro OMP, para la *build* con MKL.

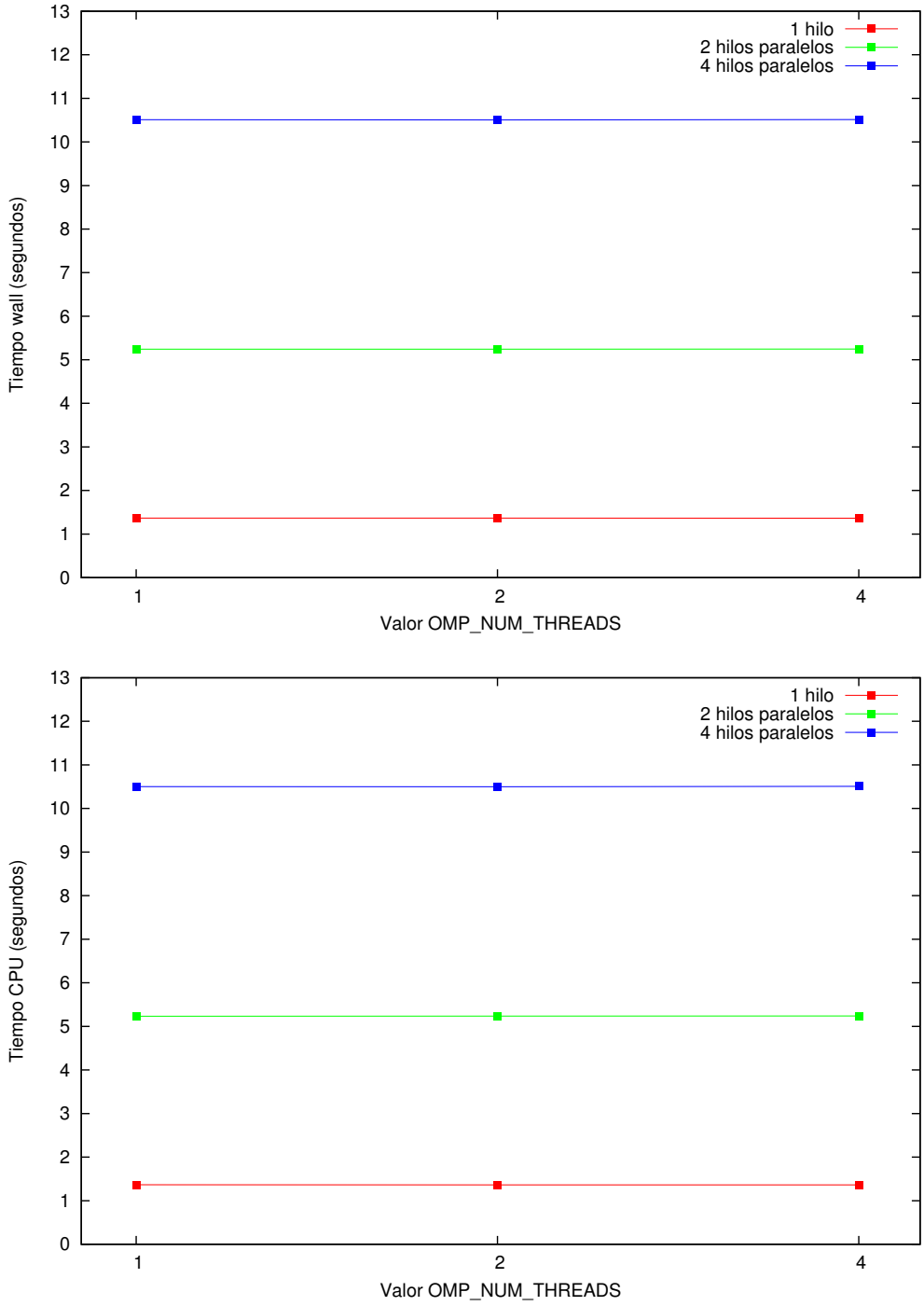

Figura 6.8: Tiempos Wall y CPU por época invertidos en la ejecución en función del parámetro OMP, para la *build* con CUDA.

Capítulo 7

Conclusiones y trabajos futuros

Después de mostrar cómo se ha elaborado el proyecto en los capítulos de diseño y haber realizado una serie de experimentos con él para poder calificar las mejoras que plantea en el capítulo de experimentación, en este capítulo vamos a presentar un resumen con las conclusiones que podemos extraer a partir de los resultados y otros temas relacionados con el uso real de la aplicación en otros proyectos o las futuras extensiones que se pueden realizar sobre ella.

7.1. Conclusiones

Podemos decir con certeza que los objetivos marcados para este proyecto han sido cumplidos con éxito. Se ha mejorado la implementación del algoritmo BP para entrenar y evaluar redes neuronales, con la opción de utilizar la GPU para efectuar estos cálculos.

Los experimentos realizados en el capítulo 6 han despejado nuestras dudas sobre si la implementación desarrollada era válida (es decir, ejecutaba el algoritmo BP como debe) y eficiente. El diseño de estos experimentos nos ha ayudado a probar un elevado número de combinaciones para los parámetros que definen el experimento, y también nos ha permitido evaluar el rendimiento de la aplicación, que ha resultado ser satisfactorio.

A pesar de que una de las principales ventajas de las redes neuronales es su enorme capacidad de paralelización (cada neurona podría calcularse de manera independiente a las demás), en la práctica, se trata de un paralelismo de "grano demasiado fino" para ser explotado por las actuales arquitecturas basadas en CPU *multi-cores*. La aproximación más eficiente para acelerar el cálculo en estas arquitecturas se basa en aprovechar la descripción matricial y utilizar bibiotecas de álgebra lineal que utilizan instrucciones especializadas (como las SSE3), arquitecturas específicas (GPUs), alineamientos de memoria, etc.

7.2. Uso de la aplicación

Las redes neuronales de April han sido utilizadas en diversas aplicaciones relacionadas con el reconocimiento estadístico de formas, tratamiento de imágenes,

procesamiento del lenguaje natural (modelado de lenguaje y traducción). En particular, hasta la fecha, se han utilizado en la tareas siguientes:

- Para clasificación de dígitos en reconocimiento de placas de matrícula y en tareas de clasificación de caracteres aislados [LPM+08] e incluso para clasificar palabras completas dadas como una imagen [ZCEG09].
- Para estimar las probabilidades de emisión en modelos ocultos de Markov hibridados con redes neuronales (HMM/ANN) tanto para realizar modelos acústicos en tareas de reconocimiento automático del habla [ZECdM11] como en reconocimiento de escritura contínua manuscrita [ECGZ11].
- Para limpiar imágenes históricas o documentos degradados utilizando un filtro neural convolucional. La idea consiste en aprender a estimar el pixel limpio a partir de un contexto del mismo pixel en la imagen original [HECP05, ZEnC07]. Para ello, es necesario disponer de pares de imágenes sucias y limpias. Una vez entrenada la red, ésta se aplica de manera independiente a cada pixel de la imagen, lo que permite sacar el máximo provecho del modo bunch no solamente en fase de entrenamiento sino en fase de explotación de la red.
- Para clasificar puntos de una imagen manuscrita y clasificarlos como partes de las líneas de referencia de cara a normalizar la imagen y mejorar el reconocimiento [GEZC08, ECGZ11].
- En tareas de lingüística computacional se han utilizado las redes neuronales para clasificar el tipo de etiqueta "part of speech" de las palabras de un documento [ZCTE09].
- Para modelado de lenguaje se entrena una red para estimar la probabilidad de aparición de una palabra dadas las $n-1$ palabras anteriores (modelo de lenguaje conocido como n-grama). Estos modelos de lenguaje habitualmente se aprenden por técnicas estadísticas (conteo de n-gramas y el uso de técnicas de suavizado para dar cuenta de los eventos no observados en entrenamiento), si bien se ha demostrado que la capacidad de las redes neuronales de interpolar valores y el uso de una representación contínua del vocabulario (palabras parecidas se asocian a representaciones parecidas) puede dar lugar a modelos con menor perplejidad (la medida estadística más utilizada para evaluar los modelos de lenguaje) así como mejores tasas de reconocimiento en las tareas en las que son utilizadas. De manera similar, las redes son también utilizadas en diversas etapas de un sistema de traducción automática (en la tesis doctoral de Francisco Zamora, codirector de este proyecto y en fase de evaluación en la actualidad). En este contexto, las redes utilizadas son enormes, con entradas y salidad de hasta cientos de miles de neuronas, donde cualquier técnica de aceleración es fundamental [ZEnGC06, ZCE09].

Las mejoras realizadas en este proyecto son automáticamente trasladables a estas tareas porque mantiene el mismo interfaz que la librería a la que reemplaza

y permitirán acelerar los entrenamientos y el uso de las mismas con redes más grandes o realizar barridos de experimentos en menor tiempo. En la actualidad, la versión desarrollada en este proyecto está siendo ya utilizada con éxito para tareas de limpieza de imágenes y de traducción automática.

7.3. Ampliaciones futuras

Este proyecto se ha elaborado tratando de dotar al usuario de una aplicación eficiente, sin dejar de ofrecer al desarrollador una interfaz capacitada para la extensión o la especialización de sus componentes. A continuación se plantean nuevas vías de trabajo surgidas a partir de este proyecto:

- **Interfaz gráfica**: Programas como SNNS disponen de una interfaz gráfica utilizada con fines didácticos para la experimentación con ANN hecha por usuarios principiantes. Desgraciadamente, este tipo de *software* deja mucho que desear después de unas pocas sesiones de trabajo, siendo un potencial sustituto este mismo proyecto con una interfaz añadida.

- **Ampliación de tipos de redes**: En este proyecto se ha optado por implementar una de las topologías de ANN más populares: Los MLP. No obstante, puede que el usuario desee emplear otros tipos de ANN para resolver ciertos problemas, por lo que resultaría de gran interés disponer de las implementaciones de otras topologías como las redes recurrentes [Pin87, LD09] para que el usuario no tenga que simularlas utilizando otras topologías o implementarlas él mismo.

- **Adaptación al uso de otras librerías**: Además de las librerías ya comentadas a lo largo de esta memoria, existen librerías alternativas para la realización de cálculos del álgebra lineal, así como componentes para los cuales esas librerías producen un código eficiente.

  En este proyecto se ha apostado por especializar código para CPUs de la marca Intel y tarjetas gráficas con tecnología CUDA, aunque hubiese sido posible trabajar con componentes de otras marcas como las CPU o las tarjetas gráficas de AMD [OJ04], para las cuales ya proporciona el fabricante librerías como la ACML, que es similar a la Intel MKL.

  Por tanto, puede ser interesante ampliar el proyecto para poder trabajar con estas librerías alternativas, ya sean de ámbito general (como la biblioteca ATLAS) o especializadas para ciertas marcas de componentes. Dado el alto grado de abstracción de la herramienta, esta modificación no supondría el rediseño completo de la aplicación, ya que tan sólo sería necesario añadir código a los bloques de memoria y los *wrappers* que con ellos trabajan.

- **Implementación del BP con paralelismo entre capas**: La topología MLP, implementada en este proyecto, es una de las topologías más populares para redes. No obstante, existen otras topologías en las cuales las capas pueden no estar conectadas de forma directa con la siguiente capa. En este caso, las redes se asemejan más a un grafo computacional sin ciclos.

Se propone la creación de un módulo que detecte las dependencias entre capas de este tipo de redes y construya automáticamente un grafo de acciones a realizar. De esta forma, sería posible la ejecución de pasos de forma simultánea para capas cuyas dependencias ya están resueltas y pueden continuar su ejecución de forma independiente a las otras. Esto supone una aceleración respecto a otras implementaciones cuyos cálculos se realizan de forma secuencial. La figura 7.1 muestra un ejemplo de este tipo de redes, en donde los resultados para las dos primeras capas ocultas pueden obtenerse de forma paralela, y una vez calculados, tamién pueden obtenerse los resultados de las capas siguientes de forma simultánea.

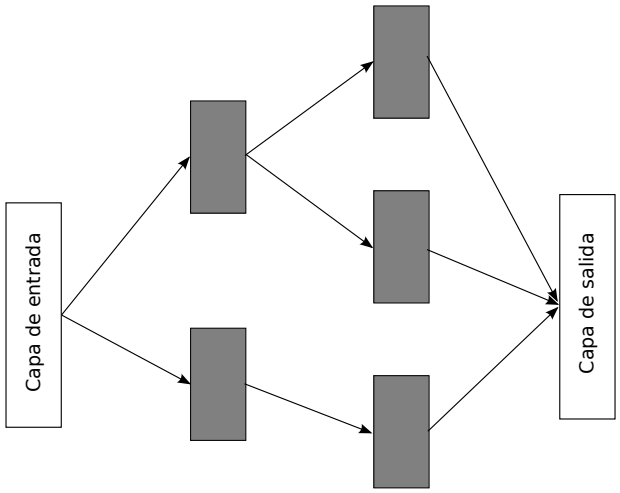

Figura 7.1: Red neuronal con capas procesables de forma paralela.

El mecanismo de acciones resulta muy general y flexible para implementar multitud de características, pero su naturaleza secuencial hace necesario un recorrido en orden topológico de las dependencias de los cálculos necesarios para simular la red. En lugar de esto, sería más eficiente que las acciones se dispusieran en un DAG (grafo acíclico dirigido) de modo que acciones independientes pudieran ejecutarse en paralelo.

- **Adaptación al uso de múltiples GPUs**: La tecnología *Scalable Link Interface* (abreviada *SLI*) es una tecnología desarrollada por Nvidia para facilitar el uso de varias GPU en un mismo ordenador.

  En el capítulo 4 se ha hablado sobre el atributo *handle*, contenida dentro de la clase estática *CublasHelper*. Este atributo, utilizado en la creación de un contexto para la librería CUBLAS, nos permite asociar una GPU con un hilo de ejecución mediante la función *cudaSetDevice()*. De esta forma, es posible llamar a las funciones CUBLAS sin tener que preparar ningún otro tipo de configuraciones, para que las funciones sean ejecutadas por la GPU asociada al *handle*.

- **Representación de valores en coma flotante doble**: Como ya se ha mencionado, la representación de valores en coma flotante utilizada en este

proyecto es la simple.

Sería interesante estudiar qué efecto tiene la adaptación de la herramienta para que funcione con valores de tipo *double*. De esta forma, podrían reproducirse los experimentos expuestos en la literatura, e investigar los resultados que se obtienen sobre éstos o sobre otros experimentos que puedan sernos de interés.

- **Paralelizar la preparación de los datos y el uso de los mismos en el cálculo del BP**: La carga y descarga de entradas, salidas y salidas deseadas es uno de los procesos que se puede realizar de forma independiente a la ejecución del algoritmo, sobre todo si estos datos han de ser transportados desde el disco duro.

  Se plantea un módulo que realice estas operaciones de carga y descarga de forma paralela a la ejecución del algoritmo. Esto puede ser conseguido mediante la creación de un nuevo tipo de capa (a partir de la clase genérica *ActivationUnits* o alguna de las subclases ya implementadas) que contenga dos bloques de memoria para sus neuronas, de forma que uno de ellos es el que sirve para la ejecución del algoritmo mediante GPU, mientras que el otro es el que sirve para realizar la carga o la descarga de la información pertinente mediante el uso de la CPU, alternando el uso de estos bloques en cada iteración. Las capas replicadas están marcadas con el color gris en la figura 7.2.

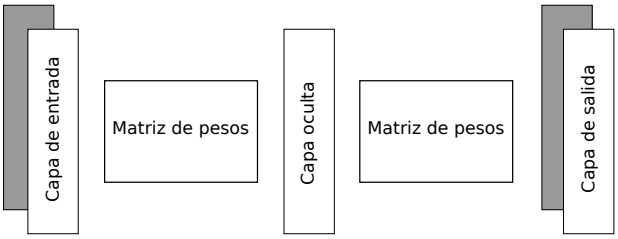

Figura 7.2: Red neuronal con capas replicadas.

- **Implementación de otros algoritmos de reconocimiento de formas mediante GPU**: Otros algoritmos para el reconocimiento de formas pueden beneficiarse del uso de la GPU para efectuar sus cálculos [LR10]. El *toolkit* April, en el cual se incluye este proyecto, ya dispone de otros modelos para el reconocimiento de formas tales como los *Modelos Ocultos de Markov* [ZZH$^+$09, LCL09], Dynamic Time Warping (DTW) [SMN$^+$10] o los algoritmos de búsqueda por los $k-$vecinos más próximos [GDB08].

  Los algoritmos utilizados en estos modelos pueden suponer una carga computacional muy pesada. Algunos de ellos ya cuentan con una formulación enfocada a matrices, como por ejemplo, el algoritmo de *Viterbi* aplicado sobre los *Modelos Ocultos de Markov*, que busca de forma aproximada la secuencia de estados más probable para una secuencia de salidas concreta. Se sugiere el rediseño de los módulos ya existentes para que implementen este tipo de algoritmos de forma eficiente mediante el uso de la GPU, aprovechando las

utilidades para el tratamiento de datos que suministra el *toolkit* April, tal y como se ha hecho en este proyecto con el algoritmo BP.

- **Adaptación para el cálculo en *grid***: Esta propuesta es similar a la adaptación para el uso de múltiples GPU, aunque en esta ocasión se pretende repartir la carga computacional entre un conjunto de máquinas (a esto se le llama computación en *grid*, que es una forma de sistema distribuido). De esta forma, es posible entrenar redes con distintas topologías o parámetros utilizando varias máquinas para ello.

Bibliografía

[Bis96] C. M. Bishop. *Neural networks for pattern recognition*. Oxford University Press, 1996.

[CBM02] R. Collobert, S. Bengio, and J. Mariéthoz. Torch: a modular machine learning software library. Technical report, Technical Report IDIAP-RR 02-46, IDIAP, 2002.

[Cor12a] NVIDIA Corporation. CUDA API Reference Manual. `http://developer.nvidia.com/cuda/nvidia-gpu-computing-documentation`, 2012. Version 4.2.

[Cor12b] NVIDIA Corporation. CUDA Best Practices Guide. `http://developer.nvidia.com/cuda/nvidia-gpu-computing-documentation`, 2012. Version 4.2.

[Cor12c] NVIDIA Corporation. CUDA C Programming Guide. `http://developer.nvidia.com/cuda/nvidia-gpu-computing-documentation`, 2012. Version 4.2.

[Cor12d] NVIDIA Corporation. CUDA Toolkit 4.2 CUBLAS Library. `http://developer.nvidia.com/cuda/cublas`, 2012. Version 4.2.

[ECGZ11] S. España, M.J. Castro, J. Gorbe, and F. Zamora. Improving offline handwritten text recognition with hybrid hmm/ann models. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 33(4):767–779, april 2011.

[EZCG07] S. España, F. Zamora, M.J. Castro, and J. Gorbe. Efficient BP Algorithms for General Feedforward Neural Networks. In *Bio-inspired Modeling of Cognitive Tasks*, volume 4527 of *Lecture Notes in Computer Science*, pages 327–336. Springer, 2007.

[GDB08] V. Garcia, E. Debreuve, and M. Barlaud. Fast k nearest neighbor search using gpu. In *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW'08. IEEE Computer Society Conference on*, pages 1–6. Ieee, 2008.

[GEZC08] J. Gorbe, S. España, F. Zamora, and M. J. Castro. Handwritten Text Normalization by using Local Extrema Classification. In *Pattern Recognition in Information Systems: 8th International Workshop on*

Pattern Recognition in Information Systems (PRIS 2008), pages 164–172, Barcelona (Spain), June 2008. INSTICC PRESS.

[GH95] J. Gibb and L. Hamey. A comparison of back propagation implementations. Technical report, Macquarie University, 1995.

[HECP05] José Luis Hidalgo, Salvador España, María José Castro, and José Alberto Pérez. Enhancement and cleaning of handwritten data by using neural networks. In J. S. Marques et al., editors, *Pattern Recognition and Image Analysis*, volume 3522 of *Lecture Notes in Computer Science*, pages 376–383. Springer-Verlag, 2005. ISSN 0302-9743.

[HSJ86] W.D. Hillis and G.L. Steele Jr. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, 1986.

[IdFC03] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes. Lua 5.0 reference manual. Technical report, PUC-Rio, 2003.

[Int07] MKL Intel. Intel math kernel library, 2007.

[JPJ08] H. Jang, A. Park, and K. Jung. Neural network implementation using cuda and openmp. In *Computing: Techniques and Applications, 2008. DICTA'08. Digital Image*, pages 155–161. Ieee, 2008.

[LC98] Y. LeCun and C. Cortes. The mnist database of handwritten digits, 1998.

[LCL09] J. Li, S. Chen, and Y. Li. The fast evaluation of hidden markov models on gpu. In *Intelligent Computing and Intelligent Systems, 2009. ICIS 2009. IEEE International Conference on*, volume 4, pages 426–430. IEEE, 2009.

[LD09] U. Lotrič and A. Dobnikar. Parallel implementations of recurrent neural network learning. *Adaptive and Natural Computing Algorithms*, pages 99–108, 2009.

[LPM+08] D. Llorens, F. Prat, A. Marzal, J.M. Vilar, M.J. Castro, J.C. Amengual, S. Barrachina, A. Castellanos, S. Espana, J.A. Gómez, J. Gorbe, A. Gordo, V. Palazón, G. Peris, R. Ramos-Garijo, and F. Zamora. The UJIpenchars Database: A Pen-Based Database of Isolated Handwritten Characters. In *Proceedings of the 6th International Conference on Language Resources and Evaluation (LREC 2008)*, Marrakech, Morocco, May 2008. European Language Resources Association (ELRA).

[LR10] L. Lopes and B. Ribeiro. Gpumlib: An efficient open-source gpu machine learning library. *International Journal of Computer Information Systems and Industrial Management Applications ISSN*, pages 2150–7988, 2010.

[lua03] *Programming in Lua.* Published by Lua.org, December 2003.

[MHL95] JE Moody, SJ Hanson, and RP Lippmann. A simple weight decay can improve generalization. *Advances in neural information processing systems*, 4:950–957, 1995.

[MN98] M. Matsumoto and T. Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30, 1998.

[MS69] M. Minsky and P. Seymour. Perceptrons. 1969.

[Nis03] Steffen Nissen. Implementation of a fast artificial neural network library, 2003.

[OJ04] Kyoung-Su Oh and Keechul Jung. Gpu implementation of neural networks. *Pattern Recognition*, 37(6):1311 – 1314, 2004.

[Pin87] F.J. Pineda. Generalization of back-propagation to recurrent neural networks. *Physical Review Letters*, 59(19):2229–2232, 1987.

[Rip96] B. D. Ripley. *Pattern Recognition and Neural Networks*. Cambridge University Press, 1996.

[Roj96] Raul Rojas. *Neural Networks - A Systematic Introduction*. Springer-Verlag, 1996.

[SCG+10] S. Scanzio, S. Cumani, R. Gemello, F. Mana, and P. Laface. Parallel implementation of artificial neural network training. In *Acoustics Speech and Signal Processing (ICASSP), 2010 IEEE International Conference on*, pages 4902–4905. IEEE, 2010.

[SHZO07] S. Sengupta, M. Harris, Y. Zhang, and J.D. Owens. Scan primitives for gpu computing. In *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 97–106. Eurographics Association, 2007.

[SMN+10] D. Sart, A. Mueen, W. Najjar, E. Keogh, and V. Niennattrakul. Accelerating dynamic time warping subsequence search with gpus and fpgas. In *Data Mining (ICDM), 2010 IEEE 10th International Conference on*, pages 1001–1006. IEEE, 2010.

[SMU10] X. Sierra, F. Madera, and V. Uc. Parallel training of a back-propagation neural network using cuda. In *Machine Learning and Applications (ICMLA), 2010 Ninth International Conference on*, pages 307–312. IEEE, 2010.

[wek99] *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*. Morgan Kaufmann, October 1999.

[WPD01] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimizations of software and the atlas project. *Parallel Computing*, 27(1-2):3 – 35, 2001. New Trends in High Performance Computing.

[Zam05] Francisco Zamora. Implementación eficiente del algoritmo de retropropagación del error con momentum para redes hacia delante generales. Proyecto Final de Carrera, 2005.

[ZCE09] F. Zamora, M.J. Castro, and S. España. Fast Evaluation of Connectionist Language Models. In Joan Cabestany, Francisco Sandoval, Alberto Prieto, and Juan M. Corchado, editors, *International Work-Conference on Artificial Neural Networks*, volume 5517 of *Lecture Notes in Computer Science*, pages 33–40. Springer, 2009. 10th International Work-Conference on Artificial Neural Networks, IWANN 2009, Salamanca, Spain, June 10-12, 2009. Proceedings.

[ZCEG09] F. Zamora, M. J. Castro, S. España, and J. Gorbe. Mejoras del reconocimiento de palabras manuscritas aisladas mediante un clasificador específico para palabras cortas. In *Conferencia de la Asociación Española para la Inteligencia Artificial*, pages 539–548, 2009.

[ZCTE09] F. Zamora, M.J. Castro, S. Tortajada, and S. España. Adding morphological information to a connectionist Part-Of-Speech tagger. In *Conferencia de la Asociación Española para la Inteligencia Artificial*. 2009.

[ZECdM11] F. Zamora, S. España, M.J. Castro, and R. de Mori. Neural Net Language Models based on Long-Distance Dependencies for a Spoken Dialog System. In *Neural Information Processing Systems*, 2011. submitted.

[ZEnC07] F. Zamora, S. España, and M. J. Castro. Behaviour-based clustering of neural networks applied to document enhancement. In *IWANN'07: Proceedings of the 9th international work conference on Artificial neural networks*, pages 144–151, Berlin, Heidelberg, 2007. Springer-Verlag.

[ZEnGC06] Francisco Zamora, Salvador España, Jorge Gorbe, and María José Castro. Entrenamiento de modelos de lenguaje conexionistas con grandes vocabularios. In *IV Jornadas en Tecnologia del Habla*, pages 141–144, Zaragoza, Spain, November 2006.

[ZMH+95] Andreas Zell, Niels Mache, Ralf Huebner, Michael Schmalzl, Tilman Sommer, and Thomas Korb. SNNS: Stuttgart Neural Network Simulator. User manual Version 4.1. Technical report, Stuttgart, 1995.

[ZZH+09] D. Zhang, R. Zhao, L. Han, T. Wang, and J. Qu. An implementation of viterbi algorithm on gpu. In *Information Science and Engineering (ICISE), 2009 1st International Conference on*, pages 121–124. IEEE, 2009.