Evaluación del rendimiento de una implementación Cloud para un clasificador neuronal aplicado a imágenes hiperespectrales

Juan Mario Haut, Mercedes Paoletti, Javier Plaza y Antonio Plaza[1]

Resumen—
La tecnología actual permite a los sensores hiperespectrales capturar cientos de imágenes, en diferentes longitudes de onda, sobre una misma zona sobre la superficie de la Tierra. Las denominadas *imágenes hiperespectrales* se caracterizan por su gran volúmen y dimensionalidad, lo que complica considerablemente su almacenamiento y procesamiento. Si además tenemos en cuenta que gran parte de los algoritmos de procesamiento de este tipo de datos presentan una elevada complejidad computacional, resulta fácil justificar la aparición de implementaciones de este tipo de técnicas sobre arquitecturas de computación de altas prestaciones. En particular, las arquitecturas cloud permiten el procesamiento distribuido de los datos, que normalmente se encuentran ubicados en diferentes centros de procesamiento. En este artículo presentamos un estudio del rendimiento computacional de una implementación cloud (desarrollada utilizando Apache Spark) de una arquitectura de red neuronal orientada a la clasificación de datos hiperespectrales. Los resultados experimentales sugieren que las arquitecturas distribuidas tipo cloud permiten procesar de forma distribuida grandes conjuntos de datos hiperespectrales.

Palabras clave— **imágenes hiperespectrales, computación neuronal, arquitecturas distribuidas, computación cloud.**

I. Introducción

Las imágenes hiperespectrales contienen información en cientos de bandas espectrales contiguas, lo que incrementa significativamente su tamaño en relación con las imágenes tradicionales, imponiendo nuevos requerimientos en términos del almacenamiento y el procesado de este tipo de datos. El crecimiento exponencial de estos requerimientos, debido al avance en la tecnología de desarrollo de los instrumentos de adquisición y a la disponibilidad de cada vez más nuevas misiones que generan un flujo continuo de datos multi/hiperespectrales, está provocando la aparición de repositorios de datos hiperespectrales de grandes dimensiones [1]. Por ejemplo, el sensor AVIRIS (Airbone Visible/Infrared Imaging Spectrometer) operado por el Jet Propulsion Laboratory de la NASA, presenta unas tasas de adquisición de datos de 2.5 MB/s (casi 9 GB/hora). Un caso similar es el sensor Hyperion [1], que adquiere casi 71.9 GB/hora (over 1.6 TB/día). La mayor parte de las misiones satelite que estarán operativas en breve, como el programa de mapeo y analisis ambiental (EnMAP) [1] presentan tasas de adquisición de datos similares.

En la actualidad, las plataformas de computación cloud están siendo utilizadas con el fin de procesar datos adquiridos de forma remota en arquitecturas distribuidas. En este sentido, la computación cloud ofrece avanzadas capacidades para computación orientada a servicios y computación de altas prestaciones. El uso de computación cloud para el análisis de grandes repositorios de datos hiperespectrales puede considerarse una solución natural, resultado de la evolución de técnicas previamente desarrolladas para otros tipos de plataformas de computación [2], [3]. Sin embargo, existen pocos ejemplos en la literatura reciente orientados al uso de infraestructuras de computación cloud para la implementación de técnicas de análisis hiperespectral en general, y para la clasificación supervisada de datos hiperespectrales en particular.

Desde los años 90, las redes neuronales han atraído la atención de gran cantidad de investigadores pertenecientes al área del análisis hiperespectral [4], [5] y, especialmente, los dedicados a la clasificación de datos hiperespectrales [6], [7], como una consecuencia directa de su éxito en el campo de reconocimiento de patrones [8]. La principal ventaja de este tipo de aproximaciones sobre los métodos probabilísticos radica en el hecho de que no necesitan conocimiento previo sobre la distribución estadística de las clases. Además, son una posibilidad atractiva debido a la disponibilidad de múltiples técnicas de entrenamiento para datos linealmente no separables [9], a pesar de que estas técnicas se hayan visto tradicionalmente afectadas por su complejidad algorítmica y computacional [10], así como por el número de parámetros que necesitan ser ajustados para su correcta aplicación. Han sido muchos los algoritmos neuronales propuestos en la literatura para la clasificación de datos hiperespectrales, incluyendo enfoques no parametrizados, tanto supervisados como no supervisados [11], [12], [13], [14], [15], siendo las redes de propagación hacia delante (*feedforward networks*, FN) las más comúnmente utilizadas para tareas de clasificación de datos hyperespectrales.

En este artículo, exploramos la posibilidad de utilizar arquitecturas distribuidas para el procesado de datos hiperespectrales. Utilizamos un clasificador neuronal como caso de estudio,

[1]Dpto. Tecnología de los Computadores y de las Comunicaciones, Universidad de Extremadura, e-mail: juanmariohaut@unex.es, mpaolett@alumnos.unex.es, jplaza@unex.es, aplaza@unex.es

[1]http://www.enmap.org/

centrándonos en el uso de arquitecturas neuronales de clasificación supervisada basadas en redes FN para demostrar la posibilidad de utilizar tecnología de computación cloud para clasificar de forma eficiente datos hiperespectrales, acelerando la computación asociada a este proceso.

El resto del artículo se organiza de la siguiente forma. La Sección II presenta el diseño del entorno de trabajo distribuido que será utilizado en la implementación evaluada. La Sección III describe el algoritmo de clasificacion neuronal utilizado. En la Sección IV describimos la implementación distribuida. La Sección V evalúa el rendimiento de la implementación considerada bajo el punto de vista de su precisión y, especialmente, de su eficiencia computacional. Por último, la Sección VI expone una serie de conclusiones y consideraciones para posibles futuros trabajos.

II. Diseño del Framework Distribuido

Con el fin de desarrollar un marco distribuido para implementar el algoritmo del perceptrón multicapa en arquitecturas de computación *cloud*, hemos abordado dos cuestiones principales: 1) el modelo de programación distribuida y 2) el motor de la computación.

Para la programación distribuida, se recurre al modelo MapReduce [2], aprovechando al máximo las capacidades de alto rendimiento proporcionados por las arquitecturas de computación en *cloud*. En este modelo, una tarea es procesada por dos operaciones distribuidas: *map* y *reduce*. Los *datasets* están organizados como pares clave/valor, y la función map procesa los pares clave/valor con el fin de generar un conjunto de pares intermedios, dividiendo una tarea en varias subtareas independientes para que se ejecuten en paralelo. La función *reduce* se encarga de procesar todos esos valores intermedios asociados con la misma clave intermedia, a continuación, se juntan todos los resultados intermedios y se genera el resultado final.

En cuanto el motor de computación distribuida, la primera solución considerada fue Apache Hadoop[2] debido a su fiabilidad y escalabilidad, así como su naturaleza *open source*. Sin embargo, Apache Hadoop sólo es compatible con cálculos simples, que se realizan de una sola pasada, por tanto, en general, no es adecuado para algoritmos iterativos tales como el perceptrón multicapa. Apache Spark[3] es un motor de computación actual cuya finalidad es el procesamiento de grandes volúmenes datos en las arquitecturas de computación *cloud*, es tolerante a fallos, y proporciona, de una manera rápida, un procesamiento de datos generales sobre grandes plataformas distribuidas. Como hemos dicho anteriormente, no sólo es compatible con cálculos simples, también con los algoritmos iterativos. Por todo lo indicado anteriormente, el diseño de nuestro framework distribuido en paralelo para la clasificación de datos hiperespectrales utilizando Spark Apache es descrito gráficamente en la Fig. 1.

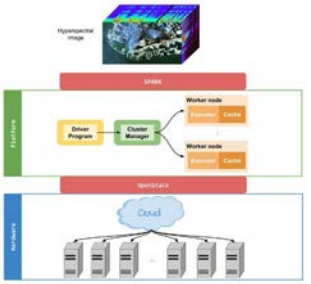

Fig. 1. Descripción de la arquitectura de Apache Spark utilizada en nuestros experimentos.

Como se muestra en la Fig. 1, la arquitectura tiene dos partes principales:

- La *zona de hardware*: contiene las máquinas físicas que soportan nuestras máquinas virtuales, que son creadas en la plataforma OpenStack[4], el sistema operativo que controla las máquinas del *cloud*, almacenamiento y recursos de red a través de un centro de procesamiento de datos, todo ello gestionado a través de un panel de control que permite a los administradores controlar a la vez que provisionar recursos a las máquinas a través de una interfaz web.
- La *zona de la plataforma*: el framework Apache Spark en la Fig. 1) está instalado sobre un conjunto de máquinas virtuales Linux Ubuntu, creadas en OpenStack. Nuestro clúster tiene diferentes tipos de nodos. El algoritmo MLP se ha diseñado utilizando la librería de *Machine Learning* MLIB[5], y la aplicación se integra en Spark y el framework OpenStack, como se ilustra gráficamente en la Fig. 2. Cuando lanzamos una instancia del clasificador MLP, el nodo maestro gestiona los recursos del clúster y los esclavos realizan tareas individuales en los datos. El maestro divide el trabajo en tareas y coordina la asignación de las mismas, siguiendo el modelo MapReduce.

[2]http://hadoop.apache.org
[3]http://spark.apache.org/
[4]https://wiki.openstack.org/wiki/Main_Page
[5]https://spark.apache.org/docs/latest/mllib-guide.html

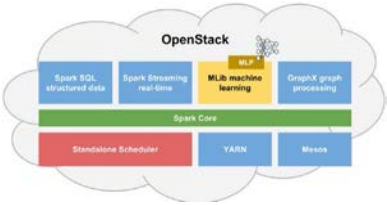

Fig. 2. Integración de Apache Spark y el framework OpenStack, ambos utilizados para la implementación del MLP.

III. Algoritmo de clasificación basado en computación neuronal

A. Redes de propagación hacia delante

Como hemos mencionado en la Seción I, las FNs han sido estudiadas en profundidad y ampliamente utilizadas desde la aparición del famoso algoritmo de retropropagación (backpropagation, BP) [16], un método de optimización basado en el método del gradiente de primer orden, que presenta dos inconvenientes principales: su convergencia es bastante lenta y existe la posibilidad de que el método se quede atrapado en un mínimo local, especialmente si los parámetros de la red no se han ajustado adecuadamente. Con el objetivo de contrarestar los inconvenientes del algoritmo original, se han propuesto en la literatura diferentes estrategias de optimización de órdenes superiores, que se caracterizan por ser más rápidas y menos parametrizadas [17], [18].

En concreto, las redes de propagación hacia delante con una única capa oculta (*single layer feedforward network*, SLFN) son las más comunmente utilizadas en el ámbito de la clasificación de datos hiperespectrales. Sean $\{\langle \mathbf{x_i t_i} \rangle | \mathbf{x_i} \in \mathbf{R^d}, \mathbf{t_i} \in \mathbf{R^m}, \mathbf{i} = \mathbf{1}, \mathbf{2}, ..., \mathbf{k}\}$ K patrones de entrenamiento diferentes. Para una SLFN con L neuronas ocultas y función de activación $g_i(x)$, su salida o_j puede expresarse como:

$$o_j = \sum_{i=1}^{L} \beta_{ij} g_i(\mathbf{x}) = \sum_{\mathbf{i=1}}^{\mathbf{L}} \beta_{\mathbf{ij}} \mathbf{g}(\mathbf{w_i} \cdot \mathbf{x} + \mathbf{b_i}) \quad (1)$$

donde $\boldsymbol{w}_i$ es el vector de pesos que conecta la $i-ésima$ neurona oculta con las neuronas de la capa de entrada, $\beta_{\mathbf{ij}}$ es el vector de pesos que conecta la $i-ésima$ neurona oculta con la $j-ésima$ neurona de salida y b_i es el bias de la $i-ésima$ neurona oculta

Podemos definir la función de error de una SLFN como el error cuadrático medio (*Mean Square Error*, MSE):

$$\boldsymbol{E} = 1/2 \sum_{i=1}^{K} ||o_i - t_i||^2$$

El objetivo del aprendizaje es minimizar la distancia entre la salida de la red y la salida deseada ajustando los pesos de la red. Como hemos mencionado antes, tradicionalmente, dicho ajuste suele realizarse mediante algoritmos de aprendizaje basados en el descenso del gradiente, que suelen presentar dos problemas fundamentales: son costosos computacionalmente (especialmente ante datos de alta dimensionalidad) y pueden quedar atrapados en algún mínimo local de la superficie del error (crucial si el mínimo local está muy lejos del mínimo global). Sin embargo, la minimización de la función de error de una SLFN puede verse estrictamente como un problema de optimización. De esa forma, son muchas las alternativas basadas en métodos de optimización de segundo orden que pueden emplearse en combinación con el algoritmo BP [19], [20], [18], [21]. Diferentes implementaciones (Scikit, Spark, etc.) utilizan en su perceptrón el método de optimización L-BFGS, tambien llamado método de Broyden-Fletcher-Goldfarb-Shanno con límite de memoria [22].

B. Método de Broyden-Fletcher-Goldfarb-Shanno

Este método se clasifica dentro de los métodos de minimización multivariante quasi-newton, los cuales pretenden minimizas una función de error $E(x)$ con $x \in R^n$ y E función real $E : R^n \to R$.

Los métodos de descenso de gradiente y de gradiente conjugado pueden minimizar esa función, pero son lentos, con un orden de convergencia lineal. Por otra parte, el método de Newton es ligeramente más rápido, con un orden de convergencia cuadrático, gracias al uso de la información de la matriz Hessiana[6] de $E(x)$. Sin embargo, el cálculo de esta matriz es bastante costoso y a veces no es factible. Por ello, los métodos quasi-newton utilizan una aproximación al Hessiano mediante la información que obtienen del gradiente.

Tomemos $E(x) \in R^n$ contínua y con segundas derivadas parciales. Para los puntos x_k y x_{k+1} con gradiente $g_i = \nabla f(x_i)$ y matriz Hessiana constante F, se cumple:

$$q_k \equiv g_{k+1} - g_k, \;\; p_k \equiv x_{k+1} - x_k$$

Por lo que $q_k = F \cdot p_k$. Entonces podemos aproximar sucesivamente F mediante matrices aproximadas H a partir de n direcciones linealmente independientes (p) y sus respectivos gradientes (q): $H = Q \cdot P^{-1}$, donde Q es la matriz de los sucesivos q_k y P la matriz de los sucesivos p_k. Por otra parte, se necesita calcular la inversa del Hessiano, F^{-1}. A partir de $q_k = F \cdot p_k$ podemos obtenerlo como $H \cdot q_k = p_k$, entonces tenemos $H = F^{-1}$.

[6]Matriz de segundas derivadas parciales de la función $E(x)$:

$$F = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \cdots & \cdots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}$$

Uno de los primeros métodos quasi-newton fue el Davidon-Fletcher-Powell (ver algoritmo 1).

Algorithm 1 Algoritmo Davidon-Fletcher-Powell

1: **procedure** DFP
2: $d_k = -H_k \nabla E(x_k)$
3: $p_k = \alpha_k \cdot d_k \leftarrow \alpha_k$ por búsqueda lineal
4: $x_{k+1} = x_k + p_k$
5: $q_k = \nabla E(x_{k+1}) - \nabla E(x_k)$
6: $H_{k+1} = H_k + \frac{p_k \cdot p_k^t}{p_k^t \cdot q_k} - \frac{H_k \cdot q_k \cdot q_k^t \cdot H_k}{q_k^t \cdot H_k \cdot q_k}$
7: **end procedure**

El método de Broyden-Fletcher-Goldfarb-Shanno aplica la fórmula $q_k = H \cdot p_k \rightarrow q_k = B_{k+1} \cdot p_k$, donde B_k es la k-ésima aproximación a la matriz hessiana. Esta expresión tiene la particularidad de que cualquier fórmula para para actualizar F^{-1} puede transformarse en una fórmula para actualizar la propia H, con tan solo sustituir F_k^{-1} por B_k e intercambiar q_k y p_k. Entonces, si intercambiamos esta forma en la fórmula de actualización de DFP obtendremos:

$$B_{k+1} = B_k + \frac{q_k \cdot q_k^t}{q_k^t \cdot p_k} - \frac{B_k \cdot p_k \cdot p_k^t \cdot B_k}{p_k^t \cdot B_k \cdot p_k}$$

Así pues, su inversa analítica será la actualización propiamente dicha de BFGS:

$$H_{k+1} = H_k + \left(1 + \frac{q_k^t H_k q_k}{q_k^t p_k}\right) \frac{p_k p_k^t}{q_k^t q_k} - \frac{p_k q_k^t H_k + H_k q_k p_k^t}{q_k^t p_k} \quad (2)$$

IV. Implementación distribuida del clasificador neuronal

A. Implementación Distrubida en Apache Spark

Apache Spark implementa una versión del MLP [23] que, básicamente, puede definirse como una FN que puede contener más de una capa oculta. En esta versión, se introduce el tamaño de bloque, *blocksize*, con el fin de acelerar el cálculo matricial. El algoritmo apila los datos en particiones de tamaño h, si el tamaño es mayor que los datos, entonces ajusta la partición a los datos. Esto se traduce en que la obtención tradicional de la salida de la capa oculta -ver ecuación (3) pasa a ser optimizado como se aprecia en la ecuación (4). Este proceso también es extrapolable a los cálculos realizados en la capa de salida.

$$y = g(W^T \cdot x + b) \quad (3)$$

Donde:

- **y** es un vector que representa la salida de la capa oculta, de tamaño L, siendo L el número de neuronas ocultas.
- W^T es la matriz de pesos de tamaño $m \times L$, siendo m la dimensionalidad de los datos de entrada.
- **b** es el vector bias con tamaño L.
- **g** es la funcion de activación.

$$Y = g(W^T \cdot X + B) \quad (4)$$

Donde:

- **Y** es la matríz que representa la salida con tamaño $L \times h$.
- W^T es la matriz de pesos de tamaño $m \times L$.
- **X** es el la matriz de entrada a la capa de tamaño $m \times h$.
- **b** es el vector bias con tamaño $L \times h$.
- **g** es la funcion de activación.

Inicialmente, los datos son cargados en estructuras de tipo *dataframe*; este tipo de datos son una colección distribuida organizadas por columnas, las cuales tienen asociado un nombre. Al igual que los RDD, este tipo tiene evaluación perezosa. Es decir el cálculo que queremos realizar solo se lleva a cabo si representa una acción. Lo interesante de los *dataframes* es que todos los datos que contienen, se distribuyen de manera automática por los *workers*.

Función de los nodos:

1. El máster tiene una tarea primordial en esta implementación, sus principales funciones son la del envío de los parámetros a los esclavos, computar el gradiente final y decidir si volver a iterar o parar en función de la condición de salida.
2. Los nodos esclavos, por otra parte, se encargan de la realización de los cálculos.

Entrenar el clasificador: Esta etapa comprende dos acciones:

1. Propagación hacia delante: En esta fase, los pesos, se establecen de manera aleatoria. A partir de ahí, en la segunda y sucesivas, lo hace mediante la siguiente ecuacion:

   $$W' = W - tamPasoIter \cdot (gradiente + regGradient(w))$$

   Por tanto, los pesos de la etapa k, W_k, serían los pesos de la etapa anterior, menos el tamaño del paso de la iteración (o paso del gradiente en k) multiplicado por la suma del gradiente, más el gradiente de la parte regularizada en la función de coste de los W_k.
2. Propagación hacia atrás: Una vez que los esclavos han calculado sus respectivos gradientes, el máster, obtiene el gradiente global mediante la siguiente ecuación:

   $$gradTotal = gradTotal - \frac{\sum_{i=1}^{K} gradientes}{numGradientes}$$

   De tal manera que el valor del gradiente se actualiza buscando la convergencia hacia el mínimo.

La Figura 3 muestra un esquema resumiendo el funcionamento del algoritmo distribuido.

Algorithm 2 Algoritmo SLFN

```
procedure CLASIFICADOR_SLFN(maxIter, capas, tamBloque, maxTol semilla)
    for iteracion < maxIter & toleracia < maxTol do
        Máster envía parámetros a esclavos
            ▷ + esclavos, - cómputo + comunicación
                ▷ - tamaño de bloque, - cómputo + comunicación
        Esclavos calculan el gradiente.
        Esclavos envían el grandiente al máster.
        Máster calcula el gradiente final.
    end for
end procedure
```

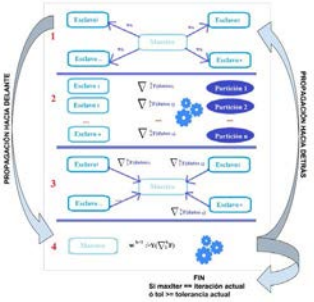

Fig. 3. Esquema de funcionamiento del MLP en Spark.

Número óptimo de workers: Para la elección del número de workers, se puede aplicar la siguiente expresión:

$$nWorkers = max\left(\left\lfloor \frac{m \cdot S \cdot \ln 2}{p \cdot (\frac{128 \cdot S}{2 \cdot b} + 2 \cdot c)} \right\rfloor, 1\right)$$

Siendo:

- m = Tamaño de los datos
- S = Número de patrones * Número de clases
- p = FLOPS
- b = Velocidad de la red
- c = Congestión de la red

B. Implementación iterativa del Perceptrón Multicapa

Scikit-Learn [24] es una librería de *Machine Learning* para Python. Esta librería contiene su propia versión del perceptrón multicapa, *sklearn.neural_network*, las principales opciones configurables son el número de capas y neuronas en cada capa, *hidden_layer_sizes*, función de activación de las capas ocultas, *activation*, la tolerancia para el proceso de optimización, *tol*, el número máximo de iteraciones, *max_iter* y el algoritmo de optimización, *algorithm*. [7]

V. Validación experimental

A. Imágenes hiperspectrales

Las imágenes [25] utilizadas en los experimentos, fueron adquiridas por el sensor AVIRIS [26]. El conocido dataset Indian Pines fue obtenido en 1992 y comprende una zona agrícola con múltiples tipos de vegetación en múltiples zonas:

1. La primera escena (Fig. 4) tiene un tamaño de 145x145 píxeles, que mezcla zona forestal con agricultura. Tiene 220 bandas en el rango de 400 a 2500 nm, con una resolución espectral de 10 nm, resolución espacial moderada de 20nm y 16 bits de resolución radiométrica. Después de un análisis inicial, 8 bandas han sido eliminadas debido al ruido existente en los datos, quedando finalmente, 212 bandas útiles. En torno a la mitad de los píxeles de la imagen (10366 de 21025) tienen información del *ground-truth* comprendida en 16 clases diferentes.

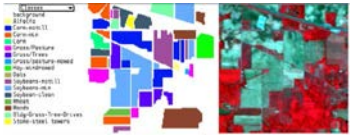

Fig. 4. Imagen en falso color y *grown-truth*

2. La segunda escena (Fig. 5) tiene un tamaño de 2678x614 píxeles, recogida sobre el mismo área pero es mucho más extensa. Contiene 220 bandas en el rango de 400 a 2500 nm, con una resolución espectral de 10 nm, resolución espacial moderada de 20nm y 16 bits de resolución radiométrica. Después de un análisis inicial, 8 bandas han sido eliminadas debido al ruido existente en los datos, quedando finalmente, 212 bandas útiles. Existen un total de 58 clases diferentes. El porcentaje de píxeles con *ground-truth* es de 20.33% (334245 de 1644292 píxeles).

[7] http://scikit-learn.org/dev/modules/generated/sklearn.neural_network.MLPClassifier.html

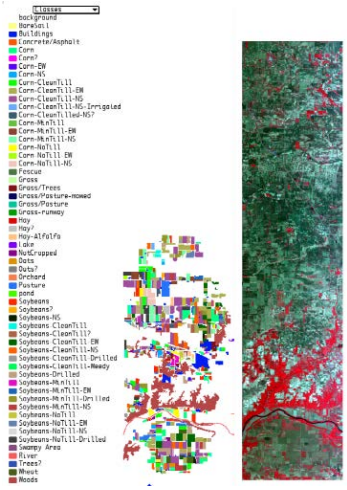

Fig. 5. Imagen en falso color y *grown-truth*

B. Descripción de los experimentos

El entorno distribuido con el que hemos probado nuestra implementación está descrito en la sección II. Como hemos mencionado en esa sección, la plataforma de *cloud computing* utilizada para nuestra evaluación experimental es OpenStack. Este software es un conjunto de tecnologías *Open Source* que proveen un desarrollo escalable en el entorno de *cloud computing* . Nuestro entorno está compuesto por Intel(R) Xeon(R) CPUs E5430 @ 2.66GHz (8 cores), 16 GB RAM, Shared storage, NetApp FAS3140. Los nodos virtuales que forman parte de del clúster construido con Apache Spark en el hardware especificado tienes dos CPUs virtuales, 4GB de RAM y 40GB de disco duro cada uno. La máquina que ejecuta el algoritmo secuencial ha necesitado 8GB de RAM debido al desbordamiento de memoria causado por el procesamiento de la imagen grande de Indian Pines, el resto de parámetros mantiene la misma configuración que las del clúster. En nuestros experimentos, hemos usado Java 1.8.0_92-b14, Ubuntu 14.04 x64 LTS como sistema operativo, Python 2.7.10, las versiones actuales, en desarrollo, Scikit Learn 0.18.dev0 y Apache Spark 2.11.

Cabe destacar que, como el principal objetivo de este trabajo es analizar el rendimiento computacional de la versión distribuida del algoritmo, hemos fijado un error objetivo muy bajo y forzamos la detención del clasificador por número de iteraciones (a pesar de que el número de iteraciones pueda resultar excesivo en el contexto del problema). Así, logramos ejecutar siempre el mismo número de iteraciones, haciendo que el análisis de la aceleración tenga mayor consistencia.

B.1 Algoritmo iterativo

En este primer experimento hemos lanzado 5 ejecuciones con la imagen pequeña de Indian Pines (en el caso de la imagen grande, solo hemos realizado una clasificación en serie debido a su elevado tiempo de ejecución), imagenes descritas en el apartado V-A. En estos tests, hemos establecido una configuración común de tolerancia de 1e-07, un número de iteraciones de 4000, sigmoide como función de activación de las neuronas de la capa oculta, el algoritmo de optimización 'l-bfgs'. Para la imagen pequeña, hemos establecido una capa con 135 neuronas ocultas mientras que para la imagen grande, una capa con 180 neuronas ocultas. La Tabla I muestra los tiempos medios de ejecución y la precisión media obtenidas al ejecutar el algoritmo de clasificación 5 veces sobre la imagen pequeña y 1 sobre la imagen grande.

Imagen	AVG Tiempo	Std Tiempo	AVG Precisión	Std Precisión
(145 x 145)	535.588	16.729	0.845	0.010
(2678 x 614)	38730.236	-	0.559	-

TABLA I
TIEMPOS DE EJECUCIÓN Y RESULTADOS DE PRECISIÓN OBTENIDOS SOBRE LOS DOS CONJUNTOS DE DATOS PROCESADOS UTILIZANDO LA IMPLEMENTACIÓN SERIE.

B.2 Algoritmo distribuido

En este experimento hemos lanzado 5 ejecuciones con las dos imágenes de Indian Pines descritas en el apartado V-A, variando el número de nodos con 1, 2, 4 y 8 nodos esclavos[8]. En estos tests, hemos establecido una configuración común de tolerancia de 1e-07, un número de iteraciones de 4000, sigmoide como función de activación de las neuronas de la capa oculta, el algoritmo de optimización 'l-bfgs'. Para la imagen pequeña, hemos establecido una capa con 135 neuronas ocultas mientras que para la imagen grande, una capa con 180 neuronas ocultas.

C. Discusión de Resultados

La Tabla II muestra los tiempos de ejecución y los resultados de precisión obtenidos sobre las dos imágenes consideradas utilizando diferentes números de nodos. Como puede observarse, los resultados de precisión son muy similares con respecto a los obtenidos por la implementación serie, mientras que el tiempo de procesamiento disminuye sensiblemente al utilizar la arquitectura distribuida considerando un número suficiente de nodos de computación.

[8]Excepto con Indian Pines pequeña que ha sido 1, 2, 4 y 7 nodos

Imagen	Nodos	AVG Tiempo	Std Tiempo	AVG Precisión	Std Precisión
(145 x 145)	1	4306.1049	89.4959	0.8104	0.0
	2	2265.4825	228.8360	0.8216	0.0026
	4	1476.4164	80.4546	0.8220	0.0021
	7	1388.5527	14.6431	0.8322	0.0023
(2678 x 614)	1	141769.2273	9572.8992	0.5550	0.0
	2	76782.8940	178.7860	0.5569	0.0008
	4	49348.3759	313.6937	0.5562	0.0016
	8	18488.2113	221.1524	0.5501	0.0010

TABLA II
TIEMPOS DE EJECUCIÓN Y PRECISIÓN OBTENIDOS SOBRE LOS DOS CONJUNTOS DE DATOS PROCESADOS UTILIZANDO LA IMPLEMENTACIÓN DISTRIBUIDA Y DIFERENTES NÚMEROS DE NODOS DE COMPUTACIÓN.

Por razones ilustrativas, la Fig. 6 compara gráficamente el tiempo de ejecución entre la versión iterativa y la versión distribuida considerando la imagen pequeña (145×145, entrenando con un total de 1554 patrones de entrenamiento para las 16 clases presentes). De forma similar, la Fig. 7 compara el tiempo de ejecución entre la versión iterativa y la versión distribuida utilizando la imagen grande (2678×614, entrenando con un total de 50136 patrones de entrenamiento entre sus 58 clases). En el caso de la imagen pequeña, se observa como el tiempo de ejecucion del entrenamiento se estabiliza en torno a los 1500 segundos. La carga de trabajo (patrones de entrenamiento) resulta insuficiente para ocultar los retardos de las comunicaciones en este caso, en el que podemos ver como la versión iterativa tiene mejor rendimiento aunque aumentemos el número de nodos de computación. Parece razonable pensar que debemos aumentar la carga de trabajo para hacer que el rendimiento de una versión distribuida pueda mejorar su rendimiento en relación a la versión iterativa.

En este sentido, si consideramos la imagen grande, podemos observar como la versión distribuida llega a mejorar a la versión iterativa al utilizar un número suficiente de nodos de computación.

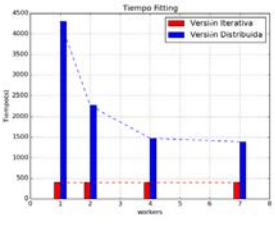

Fig. 6. Comparativa del tiempo de ejución entre la versión iterativa y la versión distribuida aumentando el número de workers sobre la imagen pequeña

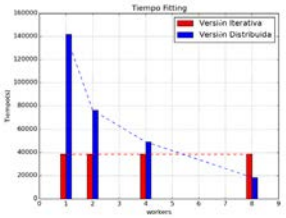

Fig. 7. Comparativa del tiempo de ejución entre la versión iterativa y la versión distribuida aumentando el número de workers sobre la imagen grande

Finalmente, la Fig. 8 muestra los factores de aceleración (speedup) obtenidos por la versión distribuida frente a la versión iterativa, considerando diferentes números de nodos. Aunque los factores de aceleración obtenidos no son demasiado significativos, conviene destacar que la implementación cloud permite procesar grandes volúmentes de datos de forma distribuida y es escalable, por lo que en el futuro realizaremos experimentos adicionales con un mayor porcentaje de patrones de entrenamiento y un mayor número de nodos, analizando en mayor detalle los patrones de computación y comunicación.

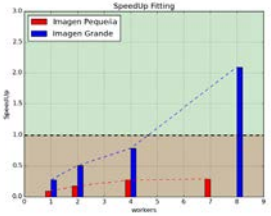

Fig. 8. Aceleración conseguida en la versión distribuida sobre la versión iterativa

VI. CONCLUSIONES Y LÍNEAS FUTURAS

En el presente trabajo se ha analizado la posibilidad de utilizar arquitecturas de computación *cloud* para el procesamiento de imágenes hiperspectrales. Como caso de estudio, hemos seleccionado una implementación *cloud computing* de un algoritmo de clasificación neuronal basado en SLFNs implementado en el framework Spark. La validación experimental evalúa la eficienda de la implementación distribuida propuesta, no solo en términos de precisión, sino también en términos de complejidad computacional, demostrándo que es posible trabajar con este tipo de implementaciones si procesamos grandes cantidades de datos. Como trabajo futuro, se planteará la evaluación de implementaciones cloud de otros algoritmos de

procesamiento de imágenes hiperespectrales.

Agradecimientos

Este trabajo ha sido subvencionado por la Junta de Extremadura (a través del decreto 297/2014, ayudas para la realización de actividades de investigación y desarrollo tecnológico, de divulgación y de transferencia de conocimiento por los Grupos de Investigación de Extremadura, Ref. GR15005). Además, el presente trabajo ha sido llevado a cabo haciendo uso de la infraestructura de computación facilitada por el Centro Extremeño de Tecnologías Avanzadas (CETA-CIEMAT), financiado por el Fondo Europeo de Desarrollo Regional (FEDER). El CETA-CIEMAT pertenece al CIEMAT y al Gobierno de España.

Referencias

[1] A. Plaza, J. Plaza, A. Paz, and S. Sanchez, "Parallel Hyperspectral Image and Signal Processing," *IEEE Signal Processing Magazine*, vol. 28, pp. 196–218, 2011.

[2] Z. Wu, Y. Li, A. Plaza, J. Li, F. Xiao, and Z. Wei, "Parallel and Distributed Dimensionality Reduction of Hyperspectral Data on Cloud Computing Architectures," *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, vol. PP, no. 99, pp. 1–9, 2016.

[3] J. Plaza, R. Pérez, A. Plaza, P. Martínez, and D. Valencia, "Parallel morphological/neural processing of hyperspectral images using heterogeneous and homogeneous platforms," *Cluster computing*, vol. 11, no. 1, pp. 17–32, 2008.

[4] J. Plaza and A. Plaza, "Spectral mixture analysis of hyperspectral scenes using intelligently selected training samples," *IEEE Geoscience and Remote Sensing Letters*, vol. 7, no. 2, pp. 371, 2010.

[5] J. Plaza, R. Pérez, A. Plaza, P. Martínez, and D. Valencia, "Mapping oil spills on sea water using spectral mixture analysis of hyperspectral image data," in *Optics East 2005*. International Society for Optics and Photonics, 2005.

[6] J. A. Benediktsson, P. H. Swain, and O. K. Ersoy, "Conjugate gradient neural networks in classification of very high dimensional remote sensing data," *Int. Jour. Remote Sens.*, vol. 14, no. 15, pp. 2883–2903, 1993.

[7] H. Yang, F. V. D. Meer, W. Bakker, and Z. J. Tan, "A backpropagation neural network for mineralogical mapping from AVIRIS data," *Int. Jour. Remote Sens.*, vol. 20, no. 1, pp. 97–110, 1999.

[8] C. Bishop, *Pattern Recognition and Machine Learning*, Springer-Verlag, New York, NY, USA, 2006.

[9] J. A. Benediktsson, *Statistical Methods and Neural Network Approaches for Classification of Data from Multiple Sources*, Ph.D. thesis, PhD thesis, Purdue Univ., School of Elect. Eng. West Lafayette, IN, 1990.

[10] J. A. Richards, "Analysis of remotely sensed data: The formative decades and the future," *IEEE Trans. Geos. Remote Sens.*, vol. 43, no. 3, pp. 422–432, 2005.

[11] J. A. Benediktsson, P. H. Swain, and O. K. Ersoy, "Neural network approaches versus statistical methods in classification of multisource remote sensing data," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 28, no. 4, pp. 540–552, 1990.

[12] E. Merényi, W. H. Farrand, J. V. Taranik, and T. B. Minor, "Classification of hyperspectral imagery with neural networks: comparison to conventional tools," *Eurasip Journal on Advances in Signal Processing*, vol. 2014, no. 1, pp. 1–19, 2014.

[13] F. Del Frate, F. Pacifici, G. Schiavon, and C. Solimini, "Use of neural networks for automatic classification from high-resolution images," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 45, no. 4, pp. 800–809, 2007.

[14] F. Ratle, G. Camps-Valls, and J. Wetson, "Semisupervised neural networks for efficient hyperspectral image classification," *IEEE Transactions on Geosciences and Remote Sens.*, vol. 48, no. 5, pp. 2271–2282, 2010.

[15] Y. Zhong and L. Zhang, "An adaptive artificial immune network for supervised classification of multi-/hyperspectral remote sensing imagery," *IEEE Transactions on Geosciences and Remote Sens.*, vol. 50, no. 3, pp. 894–909, 2012.

[16] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[17] M. Moller, "A scaled conjugate gradient algorithm for fast supervised learning," *Neural Networks*, vol. 6, no. 4, pp. 525–533, 1993.

[18] M. T. Hagan and M. Menhaj, "Training feed-forward networks with the marquardt algorithm," *IEEE Transactions on Neural Networks*, vol. 5, no. 6, pp. 989–993, 1994.

[19] K. Levenberg, "A method for the solution of certain problems in least squares," *Quart. Appl. Math*, vol. 54, pp. 164–168, 1944.

[20] D. Marquardt, "An algorithm for least-squares estimation of nonlinear parameters," *SIAM J. Appl. Math.*, vol. 11, pp. 431–441, 1963.

[21] R. Battiti and F. Masulli, "Fgs optimization for faster automated supervised learning," in *Int. Neural-Network Conf*, 1990, vol. 2, pp. 757–760.

[22] J. Nocedal, "Updating quasi-newton matrices with limited storage," *Math. of Computation*, vol. 35, pp. 773–782, 1980.

[23] A. Ulanov, "A Scalable Implementation of Deep Learning on Spark," Tech. Rep., Spark Summit, San Francisco, 2013.

[24] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine Learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[25] M. Baumgardner, L. Biehl, and D. Landgrebe, "220 Band AVIRIS Hyperspectral Image Data Set: June 12, 1992 Indian Pine Test Site 3," 2015.

[26] R. O. Green, M. L. Eastwood, C. M. Sarture, T. G. Chrien, M. Aronsson, B. J. Chippendale, J. A. Faust, B. E. Pavri, C. J. Chovit, M. Solis, M. R. Olah, and O. Williams, "Imaging spectroscopy and the airborne visible/infrared imaging spectrometer (AVIRIS)," *Remote Sensing of Environment*, vol. 65, pp. 227–248, 1998.

