UNIVERSIDAD DE EXTREMADURA

ESCUELA POLITÉCNICA

GRADO EN INGENIERÍA INFORMÁTICA EN INGENIERÍA DE COMPUTADORES

TRABAJO FIN DE GRADO

Implementación *cloud* de una red neuronal profunda para análisis de imágenes hiperespectrales

UNIVERSIDAD DE EXTREMADURA

ESCUELA POLITÉCNICA

GRADO EN INGENIERÍA INFORMÁTICA EN INGENIERÍA DE COMPUTADORES

TRABAJO FIN DE GRADO

Implementación *cloud* de una red neuronal profunda para análisis de imágenes hiperespectrales

Autor: Jose Antonio Gallardo Jaramago

Tutor: Antonio Plaza Miguel

Co-Tutor: Juan Mario Haut Hurtado

AGRADECIMIENTOS

En primer lugar, agradecer a mis tutores no solo por la guía proporcionada para la consecución de este proyecto, si no también por todos los consejos y enseñanzas que han compartido conmigo a lo largo de estos últimos meses. Estas enseñanzas, junto a lo aprendido a lo largo de estos cuatro años, no solo me han permitido adquirir los conocimientos necesarios para llevar a cabo este trabajo, si no que también han dejado una profunda huella en mi a nivel personal.

Por otra parte, agradecer a todas las personas que diariamente me animan a seguir adelante, pues sin ellos y sus palabras de apoyo quizás este trabajo nunca hubiera llegado a existir.

Resumen

Los avances en las técnicas de captación de imágenes hiperespectrales han dado lugar a un progresivo aumento de la resolución espectral de dichas imágenes. Esto genera la necesidad de desarrollar nuevas técnicas que permitan el almacenamiento, procesamiento y análisis de grandes volúmenes de datos hiperespectrales. La computación de altas prestaciones nos permite tratar estos datos mediante el uso de soluciones *cloud* distribuidas que dividen las tareas de procesamiento entre diferentes nodos de cómputo. Debido a su alta resolución espectral, las imágenes hiperespectrales se han convertido en un objetivo perfecto de este nuevo paradigma computacional. Para reducir el tamaño de almacenamiento y transferencia de dichas imágenes, en este trabajo se propone la implementación de un algoritmo no lineal de aprendizaje profundo, conocido como *autoencoder*. El *framework* distribuido *Apache Spark* es utilizado para interconectar los nodos de cómputo mediante una arquitectura maestro/esclavo. Esta nueva técnica ha sido evaluada sobre dos conjuntos de datos hiperespectrales, mostrando que la solución propuesta es muy adecuada para resolver este problema.

Abstract

Recent hyperspectral image adcquisition techniques have led to a significative increase in the spectral resolution of the captured images. This creates the need to develop new techniques to process, store and analyze big remotely sensed data volumes. High perfomance computing, in general, and distributed cloud computing, in particular, offer the potential to analyze huge data volumes by distributing processing tasks over a set of computing nodes. Due to the large amount of spectral bands present in hyperspectral images, the involved processing techniques represent a perfect application example for this new computing paradigm. In order to find alternative ways to reduce the transfer and storage times in hyperspectral data processing, we have developed a new distributed implementation of a deep learning-based nonlinear algorithm known as autoencoder. The Apache Spark framework serves as the backbone of our distributed environment by interconnecting the available computing nodes using a master/slave architechture. Our newly developed technique has been tested on two hyperspectral datasets, obtaining highly encouraging results.

Índice

1. INTRODUCCIÓN **1**

2. OBJETIVOS **6**

3. ESTADO DEL ARTE **7**

4. METODOLOGÍA **11**

4.1. *Framework* distribuido 12

4.2. Procesamiento de imágenes hiperespectrales mediante *autoencoders* . 16

4.3. Proceso de optimización del *autoencoder* 20

5. IMPLEMENTACIÓN Y DESARROLLO **21**

5.1. Flujo de ejecución distribuida del *autoencoder* 21

5.2. Implementación distribuida del *autoencoder* 27

6. RESULTADOS **41**

6.1. Entorno de pruebas 41

6.2. Imágenes hiperespectrales 43

6.3. Experimentación realizada 45

6.3.1. Experimento 1 46

6.3.2. Experimento 2 48

6.3.3. Experimento 3 49

7. CONCLUSIONES Y TRABAJO FUTURO **51**

Bibliografía **54**

Índice de tablas

1. Comparativa entre los paradigmas *cloud* y *grid computing*. 9
2. Características de diferentes algoritmos de análisis de imágenes de satélite que se basan en implementaciones *cloud*. 11
3. Características de las diferentes estructuras de datos distribuidas de *Spark*. 15
4. Topología del *autoencoder* propuesto. 23
5. Características de diferentes lenguajes de programación que poseen integración con *Spark*. 28
6. Formato de un píxel almacenado en el formato *svm*. Se observa la disposición (banda:valor), así como la omisión de los valores que son 0, se referencia la Imagen 1 donde se aprecia que cada banda tiene asociado un valor de reflectancia. 29
7. Resultados de la experimentación con la imagen *BIP* (Experimento 1). Se proporciona el error obtenido sobre el conjunto de validación, el tiempo de cómputo (en segundos), y la eficiencia con respecto a la versión serie. 47
8. Resultados de la experimentación con la imagen *HDS* (Experimentos 2 y 3). Se proporciona el error obtenido sobre el conjunto de validación, el tiempo de cómputo (en segundos), y la eficiencia con respecto a la versión serie. 52

Índice de figuras

1. Representación gráfica de tres firmas espectrales, de los materiales avena (a), maíz (b) y asfalto (c), respectivamente, obtenidas de las imágenes AVIRIS Indian Pines (a y b) y ROSIS Pavia University (c). . 2
2. Proceso de captación de imágenes hiperespectrales mediante un espectrómetro a bordo de un satélite. 3
3. Ilustración gráfica de un clúster *Spark*, compuesto por un maestro y varios esclavos. 13
4. Representación gráfica de un *autoencoder*. 18
5. *Pipeline* completo del autoencoder distribuido. Primero, el cubo de datos se convierte en una matriz que se particiona entre los nodos esclavos. Estas particiones se duplican para obtener un par entrada/salida idéntico, que sera utilizado mediante un entrenamiento iterativo de optimización del error basado en el algoritmo *L-BFGS*. . . 22
6. Proceso de entrenamiento distribuido en el instante *t*, tras desapilar los píxeles contenidos en un subconjunto. 24
7. Fragmento de código encargado de ejecutar el escalado paralelo de datos. La variable *rescale* contiene la *UDF* aplicada a cada columna. Estas funciones se reservan con la palabra reservada *UDF*. La ejecución de la misma se lleva a cabo mediante el uso de la función *withColumn*, contenida en la clase *dataframe* y en todos sus tipos heredados. 30
8. Esquema indicando la interconexión de las clases involucradas en el proceso de entrenamiento distribuido. 31
9. Fragmento de código encargado de instanciar la topología y la clase encargada del entrenamiento. 32
10. Fragmento de código encargado de inicializar la topología. 32
11. Funciones realizadas en el interior de las capas afines. 34

12. Declaración de la función de activación no lineal. 35

13. Funciones realizadas en el interior de las capas funcionales. 36

14. Conjunto de métodos utilizados para la aplicación de funciones de mapeo y reducción sobre vectores. 37

15. Función encargada de inicializar los pesos. 38

16. Función encargada de apilar las entradas para su procesamiento simúltaneo. 39

17. Proceso de mapeo-reducción cuando el conjunto de reducciones se realiza en forma de árbol. 40

18. Proceso de propagación hacia adelante de los datos. 41

19. Cálculo de la señal de error y de la matriz de incidencia. 42

20. Proceso de mapeo-reducción ejecutado sobre las diferentes particiones del conjunto de datos. 43

21. Proceso de asignación de recursos en un clúster *Spark*, utilizando una serie de recursos virtualizados en una plataforma *cloud* 44

22. Representación en falso color de la imagen hiperespectral AVIRIS Big Indian Pines (BIP). 45

23. Representación en falso color de la imagen hiperespectral Hyperion Data Set (HDS). 46

24. Escalabilidad de nuestra implementación en términos de eficiencia en base al número de nodos esclavos utilizados (experimento 1). La linea roja indica la eficiencia teórica, mientras que la barra azul indica la eficiencia obtenida. 47

25. Escalabilidad de nuestra implementación en términos de eficiencia en base al tamaño del problema (experimento 2). La linea roja indica la eficiencia teórica, mientras que la barra azul indica la eficiencia obtenida. 48

26. Escalabilidad de nuestra implementación en términos de eficiencia en base al número de nodos esclavos utilizados para problemas muy grandes (experimento 3). La linea roja indica la eficiencia teórica, mientras que la barra azul indica la eficiencia obtenida. Para una mejor lectura de los resultados, se presenta por separado el resultado de la experimentación para porcentajes de entrenamiento de 20 % y 40 % (a) y 60 % y 80 % (b). 50

1. INTRODUCCIÓN

Dada la inmensidad de longitudes de onda comprendidas dentro del espectro electromagnético, el estudio del mismo se ha centrado tradicionalmente en las longitudes de onda comprendidas entre 10^{-15}m y 10^{8}m, abarcando el denominado espectro visible, infrarrojo y ultravioleta [1]. El ojo humano solamente percibe las frecuencias contenidas dentro de este espectro visible (390-750 nm). Por tanto, es imposible percibir a simple vista una gran cantidad de información valiosa contenida en el resto de longitudes de onda. Las imágenes hiperespectrales [2] permiten resolver esta limitación, ya que contienen información mas allá del espectro visible.

Podemos clasificar las imágenes digitales según la cantidad de información espectral que contienen, existiendo así tres grandes tipos: (i) las imágenes RGB (*red-green-blue*), que contienen únicamente las bandas relativas a los colores rojo, verde y azul, (ii) las imágenes multiespectrales, que tradicionalmente cuentan con menos de 30 bandas espectrales, y (iii) las imágenes hiperespectrales, que cuentan con mas de 30 bandas espectrales contiguas (tradicionalmente, cientos de bandas). Al poseer estas últimas una gran cantidad de bandas contiguas en el espectro, ofrecen para cada material una huella única, la denominada firma espectral [3].

La Figura 1 muestra un ejemplo de tres firmas espectrales obtenidas a partir de dos imágenes hiperespectrales reales, obtenidas por los sensores *Airborne Visible Infra-Red Imaging Spectrometer* (AVIRIS) [2] y *Reflective Optics Spectrographic Imaging System* (ROSIS) [3]. Estas firmas pueden presentarse de forma pura, si representan de forma completa el material asociado, o de forma impura. En este último caso, puede ocurrir que co-existan varios materiales en un píxel debido a la baja resolución espacial de estas imágenes, con lo que la firma espectral obtenida se obtiene como una mezcla de firmas puras, dificultando su interpretación. En este último caso, existen herramientas de desmezclado como los llamados *denoising autoencoders* [4], capaces de eliminar el ruido generado por estas impurezas mediante técnicas de aprendizaje profundo [5].

En la actualidad, es posible encontrar repositorios abiertos donde se almacenan firmas espectrales puras de materiales comunes, como la librería ASTER [6] de NASA, que contiene firmas espectrales puras de materiales tanto terrestres como lunares.

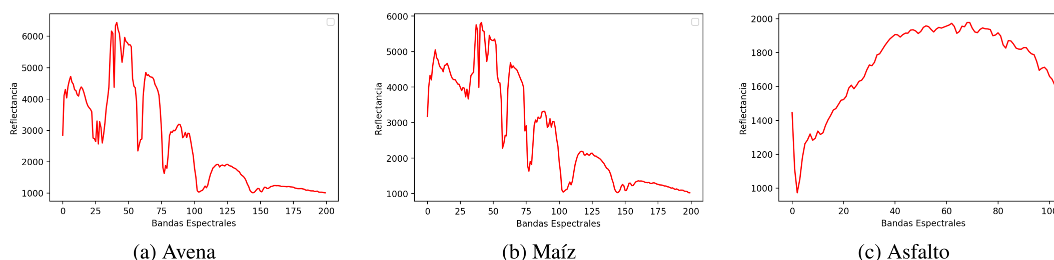


Figura 1: Representación gráfica de tres firmas espectrales, de los materiales avena (a), maíz (b) y asfalto (c), respectivamente, obtenidas de las imágenes AVIRIS Indian Pines (a y b) y ROSIS Pavia University (c).

Para la captura de imágenes hiperespectrales se utilizan sensores de observación remota de la Tierra llamados espectrómetros de imagen. El funcionamiento de estos sensores se basa en el concepto de reflectancia. Cuando la luz incide sobre un material concreto (previa absorción de parte de esta luz por parte de la atmósfera), genera una radiación reflejada que es diferente para cada material observado (similar a una huella digital). El proceso de captación de imágenes hiperespectrales aparece descrito de forma gráfica en la Figura 2.

Los espectrómetros de imagen son capaces de captar la radiación reflejada de fuentes externas, y se denominan sensores pasivos. Sin embargo, existen también sensores activos, capaces de generar la radiación por si mismos. Soluciones como el radar y los sensores LIDAR *(Light Detection And Ranging)* poseen ventajas como la capacidad de atravesar capas nubosas o de tomar imágenes con poca luz [1].

Estos avances en el instrumental utilizado para la captación de imágenes remotas han dado lugar a una notoria evolución en los métodos de observación remota del planeta [7]. Como se ha mencionado anteriormente, las imágenes hiperespectrales recogen y almacenan información en los espectros visible, infrarrojo cercano e infrarrojo de onda corta, lo que las hace idóneas para caracterizar parámetros biológicos, geológicos, químicos y físicos en la superficie terrestre. La información

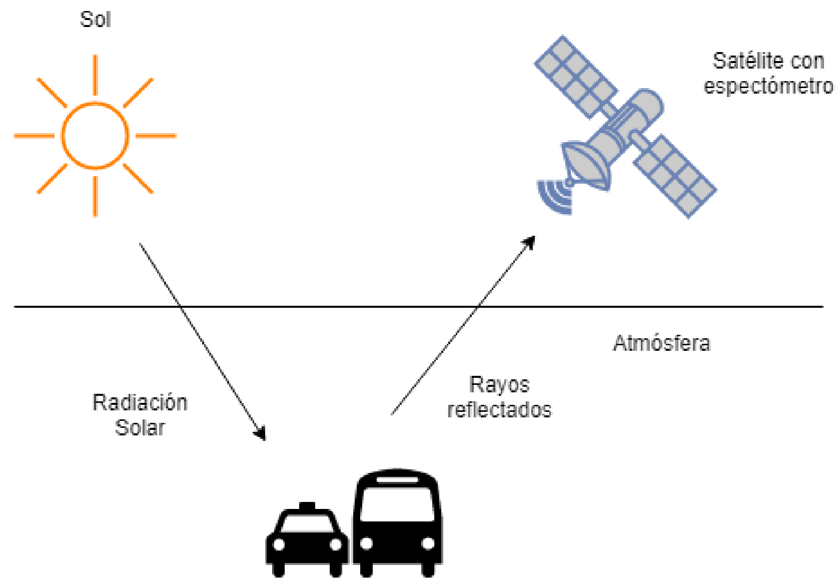


Figura 2: Proceso de captación de imágenes hiperespectrales mediante un espectrómetro a bordo de un satélite.

almacenada en dichas bandas espectrales ofrece numerosas aplicaciones en campos tan diversos como la agricultura, defensa, geología, monitorización de recursos naturales o aplicaciones urbanas [8, 9, 10].

Durante los últimos años, la gran mayoría de imágenes hiperespectrales han sido capturadas mediante espectrómetros de imagen alojados en diferentes medios aéreos. Por ejemplo, el sensor AVIRIS [2] fue uno de los primeros espectrómetros alojados en una plataforma aerotransportada. Este espectrómetro lleva en activo desde el final de la década de los 80. Otros ejemplos son *Airborn Prism Experiment* (APEX) [11] de la Agencia Espacial Europea, activo desde 2011 hasta 2016, o CASI *(Compact Airborne Spectographic Imager)* [12], entre otros.

La instalación de sensores hiperespectrales a bordo de satélites supuso un importante salto cualitativo. Por ejemplo, satélites como EO-1 Hyperion [13] de NASA o CHRIS [14] de la ESA son las principales fuentes actuales de imágenes hiperespectrales. En la actualidad, existen varias misiones hiperespectrales en desarrollo, como el *Prototype Research Instruments and Space Mission technology Advancement* (PRISMA) [15] de la Agencia Espacial Italiana, o el proyecto de

mapeado y análisis del entorno EnMAP [16] de la Agencia Espacial Alemana.

La liberación de los datos tomados por estos satélites al publico general por parte de NASA [17] y ESA (mediante el programa Copernicus [18]) ha dado lugar a que los investigadores puedan acceder en la actualidad a grandes bases de datos de imágenes hiperespectrales, normalmente distribuidas en cuanto a su almacenamiento. Por ejemplo, el satélite *Sentinel* de la ESA provee una cantidad media diaria de 93.5 Tb diarios de datos[1], por lo que el análisis y tratamiento de datos hiperespectrales puede ser englobado en el término *big data* debido a la imperiosa necesidad de velocidad de procesamiento sobre grandes volúmenes de datos distribuidos, así como de una gran precisión en los resultados del análisis. [19, 20].

Debido a esto, las tecnologías de procesamiento de imágenes hiperespectrales basadas en soluciones serie, como MATLAB o *R*, están sufriendo limitaciones cada vez mas severas a medida que el volumen y el carácter distribuido de los datos crece, sobre todo cuando es necesaria la explotación de los datos en tiempo real Esto ha dado pie al estudio de nuevas alternativas basadas en sistemas paralelos, como tarjetas gráficas programables (GPUs), para la explotación de dichas imágenes. Estos sistemas proporcionan soluciones con un alto grado de aceleración, pero aún existen limitaciones a nivel de memoria y distribución de datos que estos sistemas no pueden resolver [21, 3].

Es por ello por lo que recientemente han surgido otras soluciones que proporcionan escalabilidad no solo a nivel de capacidad de cómputo, si no también a nivel de almacenamiento tanto volátil (RAM) como persistente (disco duro). Estas soluciones se basan en clústers [22], *grids* [23], o *cloud computing* [24, 25]. Además, el procesamiento de volúmenes masivos de datos se lleva a cabo usando modelos de programación preparados para ello. Un ejemplo seria el modelo *MapReduce* [26], modelo adoptado por los *frameworks* de *Apache Hadoop* [27] y *Apache Spark* [28]. Estas soluciones *cloud* ofrecen una alternativa perfecta para el procesamiento de grandes volúmenes de datos hiperespectrales, sobre todo cuando estos datos se

[1] https://sentinel.esa.int/web/sentinel/news/-/article/sentinel-data-access- annual-report-2017

encuentran distribuidos en diferentes localizaciones geográficas [29, 30, 31].

En el presente trabajo proponemos una nueva implementación que utiliza servicios remotos y tecnologías *cloud* para la compresión de imágenes hiperespectrales, facilitando así su posterior transferencia y tratamiento [32, 33, 8, 34]. Concretamente, se propone una solución no lineal utilizando un *autoencoder* profundo para reducir la dimensionalidad (número de bandas espectrales) de la imagen hiperespectral de entrada, asumiendo que las características principales de una imagen hiperespectral pueden obtenerse a partir de un subconjunto de bandas de la misma [35], lo que nos permite realizar el proceso de compresión sin pérdida de información [36].

La implementación se ha realizado utilizando el paradigma *MapReduce* ofrecido por *Apache Spark* [37, 29, 38, 39], y se presenta como una alternativa al conocido método *Principal Component Analysis* (PCA) que ya ha sido implementado en arquitecturas *cloud* [40]. La principal aportación de nuestro método frente a soluciones existentes badas en PCA es el carácter no lineal del *autoencoder* propuesto (que se ajusta mejor a las características no lineales presentes en el proceso de adquisición y análisis de imágenes hiperespectrales), frente al carácter estrictamente lineal del método PCA. Nuestra comparativa indica que la implementación del método *autoencoder* se ajusta perfectamente al paradigma de la computación *cloud*, acelerando el tiempo de procesamiento del algoritmo sin perder calidad en la compresión realizada.

El trabajo fin de grado se encuentra estructurado de la siguiente forma. En primer lugar, se proporciona una visión en profundidad de los objetivos perseguidos en la sección 2. La sección 3 ofrece un breve resumen de las técnicas utilizadas en el campo del análisis de grandes volúmenes de datos (*big data)* y en el campo de análisis de imágenes hiperespectrales. La sección 4 muestra una explicación general del *framework* distribuido desarrollado, la red utilizada, y su proceso de optimización. La explicación detallada de la implementación de la red y el flujo de datos completo del proceso de entrenamiento se detalla en la sección 5. Para demostrar la funcionalidad de la implementación distribuida, se proporciona un análisis preciso de los resultados de

validación obtenidos en la sección 6. Finalmente, la sección 7 describe las conclusiones extraídas y las futuras líneas de trabajo.

2. OBJETIVOS

El objetivo general del presente trabajo es la implementación de una red neuronal de tipo *autoencoder* ejecutada bajo el *framework* de cómputo *Apache Spark*, integrándose la misma dentro de la librería de *machine learning* MLLib. Este *autoencoder* pretende proporcionar una solución no lineal al problema de reducción de la dimensionalidad espectral de una imagen mediante el uso de una topología de compresión/descompresión, donde la salida generada es la reconstrucción de la entrada obtenida tras aumentar de nuevo la dimensionalidad de los datos reducidos. Esta solución pretende aprovechar todas las ventajas inherentes al uso del paradigma *cloud* mencionadas anteriormente, como la tolerancia a fallos o las facilidades a la hora de escalar recursos, lo que permite aplicar esta solución sobre grandes volúmenes de datos distribuidos y así lograr una alternativa capaz de desarrollar la operación de reducción en un tiempo mucho menor, posibilitando la explotación de grandes repositorios de imágenes hiperespectrales en tiempo real. Además, es necesario que la solución proporcionada sea robusta y ofrezca una reconstrucción fiable de las firmas espectrales contenidas en la imagen. Para demostrar esta funcionalidad, se lleva a cabo una experimentación consistente en la ejecución de la implementación propuesta sobre dos grandes conjuntos de datos hiperespectrales.

Para la consecución de este objetivo general, se ha llevado a cabo una serie de objetivos específicos:

1. Estudiar las principales técnicas de computación distribuida. Tras el análisis de las distintas soluciones existentes, se opta por el uso de una solución *cloud* debido a su mayor facilidad de gestión y tolerancia a fallos.

2. Estudiar las técnicas de análisis de imágenes hiperespectrales. Tras estudiar el estado del arte, se selecciona el método *autoencoder*, pues proporciona una

solución no lineal al problema de compresión de datos hiperespectrales, que presentan características claramente no lineales en su proceso de adquisición.

3. Seleccionar un *framework* de cómputo distribuido. En la actualidad, existen varias soluciones sobre las que realizar una implementación: *MapReduce*, como *Hadoop* o *Spark*. La facilidad de programación de este último y su capacidad de tolerancia a fallos hace que, en nuestra opinión, se ajuste mejor al problema considerado.

4. Asignar recursos de cómputo e instalación del *framework Apache Spark* en modo clúster. El uso de *Spark* sobre los recursos virtualizados permite interconectar los mismos siguiendo el modelo maestro/esclavo.

5. Realizar una implementación distribuida del *autoencoder*. Utilizando la API de *Spark* en el lenguaje de programación *Scala*, se ha desarrollado una solución distribuida y escalable del *autoencoder* sobre la librería de *machine learning* MLLib.

6. Analizar los resultados obtenidos. Para demostrar la fiabilidad de la implementación desarrollada, se ejecuta la misma sobre un clúster *Spark* y se determina su robustez en base a diferentes métricas de rendimiento, tanto a nivel de cómputo como a nivel de reconstrucción espectral, utilizando dos imágenes hiperespectrales de referencia en la literatura.

3. ESTADO DEL ARTE

El paradigma de computación distribuida se basa en la acumulación de recursos de cómputo enfocados a la resolución de un problema cuya complejidad computacional lo hace irresoluble en un entorno personal. Estos conjuntos de recursos pueden ser *homogéneos*, si todos los recursos son de un único tipo (por ejemplo, un clúster de procesadores o de *GPUs*), o *heterogéneos*, si recursos de diferente tipo actúan como una única entidad de cómputo, siendo necesario un gestor de comunicaciones a bajo

nivel capaz de traducir las diferentes llamadas a estos recursos a un lenguaje común. Como se ha mencionado en la introducción de este trabajo, las principales soluciones distribuidas son las basadas en los conceptos de *grid* y *cloud computing*.

Las soluciones *grid* están basadas en el paradigma de cómputo HTC *(High Throughput Computing)*, en el que una tarea es divida en diferentes trabajos que pueden ser asignados a un conjunto de recursos de cómputo. Algunas de las ventajas de la computación *grid* son la capacidad de interconectar recursos heterogéneos, y su independencia geográfica. La existencia de redes de interconexión de área global de gran velocidad permite interconectar de forma eficiente recursos separados entre si. Es necesario un *software* de gestión, llamado *middleware*, capaz de coordinar las tareas de procesamiento entre los nodos del *grid*. Un buen ejemplo de *grid computing* es el proyecto *SETI@Home* [41], en el que los usuarios ceden el tiempo ocioso de proceso de sus ordenadores para colaborar en ciertos cálculos de ámbito científico.

La tendencia actual ha seguido una evolución natural hacia el *cloud computing*, un nuevo paradigma que ofrece ciertas ventajas con respecto a las soluciones *grid* basadas en servidores [42]. Al derivar la carga de trabajo en un conjunto de servicios de cómputo virtualizados e interconectados por una red de alta velocidad [25, 43], se libera al usuario final de gestionar los recursos *hardware* [39]. Esto, unido a la centralización de dichos recursos, no solo evita la necesidad de un *middleware* para la gestión de interconexiones, sino que también proporciona una mayor tolerancia a fallos. En la Tabla 1 se proporciona una comparativa de los modelos descritos. Actualmente, grandes compañías tecnológicas ofrecen diferentes soluciones *cloud*, basadas en Infraestructura como Servicio (*IaaS*) o Software como Servicio (*SaaS*), destacando las plataformas *Amazon Web Services* (*AWS*) [44], *Microsoft Azure* [45], y *Google Cloud Platform* [46].

La evolución en las técnicas de computación distribuida han hecho de ella una alternativa cada vez mas deseable para el análisis de imágenes teledetectadas. Es por ello que, en la actualidad, es cada vez mas común encontrar implementaciones distribuidas en *cloud* de algoritmos cuyas versiones secuenciales han demostrado

	Paradigma de cómputo distribuido	
Características	Grid Computing	Cloud Computing
Arquitectura	Descentralizada	**Centralizada**
Necesidad de middleware	Si	**No**
Modelo de adquisición de recursos	Cesión del recurso	**Virtualización**
Cesión de cómputo	Todo el recurso	**Bajo demanda**
Escalabilidad	Mas compleja	**Mas sencilla**

Tabla 1: Comparativa entre los paradigmas *cloud* y *grid computing*.

una alta eficiencia en el análisis de este tipo de imágenes, pues dichas soluciones proporcionan un aumento en la aceleración que no afecta a los resultados del análisis.

Recientemente se han desarrollado varias implementaciones *cloud* de algoritmos de clasificación de imágenes hiperespectrales, donde cabe destacar las implementaciones del algoritmo de agrupamiento *k-medias* [37], un método de clasificación no supervisado (es decir, no necesita etiquetas que guíen el proceso de aprendizaje) basado en la utilización de centroides para la agrupación de características. El algoritmo genera *k* centroides y agrupaciones de datos, de forma que durante cada iteración va asignando a cada grupo los valores de entrada de manera que la distancia entre cada grupo sea mínima para posteriormente actualizar los centroides. Este proceso se realiza de forma iterativa tras seleccionar los centroides mas alejados entre si. La implementación *cloud* de este algoritmo paraleliza el cálculo de los nuevos centroides en los nodos de tal forma que cada nodo calcula su propio conjunto de centroides y ejecuta una reducción en el nodo maestro para obtener los mejores *k* centroides. Dicha implementación se ejecuta bajo el paradigma *MapReduce*, utilizando *Apache Spark* como *framework*. Por otro lado, destacamos también la implementación *cloud* del algoritmo de regresión logística multinomial [47] para clasificación de imágenes hiperespectrales.Este método, que a diferencia del anterior realiza la clasificación de manera supervisada, es una adaptación del método de la regresión logarítmica capaz de resolver problemas de categorización no binaria. Para ello, trata de predecir la probabilidad de pertenencia de cada dato de entrada a cada clase del conjunto mediante el uso de el coeficiente de similitud logarítmica. El

objetivo final del algoritmo *MLR* consiste en maximizar dicho coeficiente utilizando un proceso de estimación logarítmica.. En este caso, la implementación se ha realizado también sobre *Apache Spark*, aprovechando los nodos esclavos para realizar los cálculos de parámetros intermedios de forma paralela utilizando particiones mas pequeñas del conjunto de datos, una vez computados, estos parámetros son reducidos en el nodo maestro.

Es importante destacar también dos implementaciones *cloud* del algoritmo de reducción de dimensionalidad (*Principal Component Analisys)* (PCA) [40, 38]. Este algoritmo realiza, en primera instancia el cálculo de la matriz de covarianza utilizando para ello el valor medio de cada banda espectral implicada en la computación. Acto seguido se realiza la descomposición de la misma en autovalores (λ) y autovectores $(\mathbf{V})$, que son ordenados según el valor de cada autovector de forma descendente ($\lambda_0 \geq \lambda_1 ...$). Por ultimo se seleccionan una serie de autovectores que serán utilizados para reducir la dimensionalidad de los datos de entrada. La finalidad de estas propuestas es la misma que la del *autoencoder* desarrollado en el presente trabajo. Las mencionadas implementaciones *cloud* de *PCA* basan su funcionamiento distribuido en el cálculo de la matriz de covarianza, realizando el mismo en función de una serie de tareas mas pequeñas realizadas por los nodos de trabajo y reducidas en el nodo maestro. Sin embargo, la implementación en [40] se ejecuta sobre un *cluster Apache Hadoop*, mientras que el algoritmo distribuido presentado en [38] lo hace sobre *Apache Spark*, siendo este último mas eficiente en términos de rendimiento computacional.

A pesar de que todas las soluciones mencionadas en este apartado tienen en común el uso de implementaciones *cloud* para reducir el tiempo de cómputo, no es posible comparar la solución propuesta en este trabajo y los algoritmos de clasificación mencionados, pues sus finalidades son completamente distintas. Sin embargo, es importante mencionar las ventajas del *autoencoder* propuesto con respecto al algoritmo *PCA*, haciendo especial énfasis en su capacidad para encontrar relaciones no lineales entre los datos. Si nos remitimos a la Figura 1, podemos observar que no es difícil distinguir entre materiales con características espectrales muy dispares (por

ejemplo, avena y asfalto). Sin embargo pero, si los materiales poseen características espectrales similares, como la avena y el maíz, las relaciones no lineales se convierten en una herramienta poderosa que mejora la discriminación del algoritmo. Otra ventaja del *autoencoder*, al ser un método parametrizado, es que puede ser utilizado para comprimir otros conjuntos de datos similares sin necesidad de realizar el entrenamiento con esos datos. La Tabla 2 muestra un resumen de las características de los diferentes algoritmos de análisis de imágenes de satélite (basados en implementaciones *cloud*) que hemos descrito en el presente apartado.

	Algoritmos			
Características	**K-Means**	**Regresión logística**	**PCA**	**Autoencoder**
Objetivo	Clasificación	Clasificación	Regresión	Regresión
Supervisado	**No**	No	**No**	**No**
Precisión	Baja	Media	Media	**Alta**
Puede predecir	No	**Sí**	No	**Sí**
Linealidad de la solución	Lineal	Lineal	Lineal	**No lineal**

Tabla 2: Características de diferentes algoritmos de análisis de imágenes de satélite que se basan en implementaciones *cloud*.

4. METODOLOGÍA

La complejidad espectral de las imágenes hiperespectrales hace posible encontrar firmas espectrales de la misma clase pero con nivel de similitud muy bajo, debido a factores externos como la iluminación, la profundidad, la presencia de diferentes componentes a nivel sub-píxel, o la proximidad al agua. El uso de algoritmos de extracción de características reduce la varianza de firmas espectrales [48], posibilitando la representación de un determinado objeto o material mediante un conjunto de características comunes, eliminando así la dificultad de interpretación inherente a esta varianza en los datos hiperespectrales [49] y al denominado fenómeno de *Hughes* [50].

Los métodos descritos en la sección 3 de este trabajo se han utilizado tradicionalmente para proporcionar una solución lineal al problema de la interpretación de los datos hiperespectrales, siendo *PCA* el algoritmo mas ampliamente utilizado para

la reducción dimensional dichas imágenes. Como se ha comentado, *PCA* proporciona una solución no supervisada capaz de aplicar una transformación lineal a un conjunto de datos de alta dimensionalidad para extraer sus componentes principales (con un nivel menor de correlación) sin que se produzca una perdida significativa de información. El problema inherente a este algoritmo es que, al realizar una proyección lineal, no es capaz de inferir transformaciones no lineales.

Las técnicas no lineales, como los *autoencoders* [51], ofrecen una solución que se ajusta mejor a la distribución real de los datos. Esto se debe a su arquitectura basada en un conjunto de capas con funciones de activación no lineales, capaces de extraer representaciones más detalladas. A continuación describimos en detalle el *framework* distribuido que hemos desarrollado en el presente trabajo para realizar una implementación *cloud* eficiente del método *autoencoder* para reducción dimensional de imágenes hiperespectrales.

4.1. *Framework* distribuido

Se ha desarrollado un nuevo método escalable y tolerante a fallos capaz de ejecutar *autoencoders* de manera distribuida en entornos *cloud* para el análisis de imágenes hiperespectrales. En primer lugar, es necesario definir el *framework* de cómputo utilizado. Esta elección condicionará, entre otras cosas, el paradigma de computación distribuida utilizado.

En la actualidad, existen varios *framework* de computación distribuido para el análisis de grandes volúmenes de datos, entre los que podemos enumerar *Apache Spark*, *Apache Hadoop*, *Apache Storm* o *Apache Hive*. Al estar estos dos últimos enfocados a propósitos mas específicos (*Storm* proporciona un mayor control del flujo de datos en tiempo real, mientras que *Hive* encadena procesos de análisis sobre transacciones de bases de datos), sólamente hemos tenido en cuenta los dos primeros como alternativa. Mientras que *Hadoop* proporciona los datos directamente desde el soporte de almacenamiento, las operaciones en los clústers basadas en *Spark* se realizan sobre datos almacenados en memoria, lo que hace de este último una solución

mas rápida a la hora de abordar operaciones que comprenden gran cantidad de datos. Esto, acompañado de las facilidades para realizar tareas distribuidas de *machine learning* (gracias a su librería integrada MLLib) nos han llevado a escoger *Spark* como *framework* para el desarrollo de nuestra implementación.

En *Apache Spark*, la ejecución se realiza mediante un clúster de máquinas *standalone* compuesto por varios nodos con arquitectura maestro/esclavo. El *framework* se encarga de proporcionar características cruciales como flexibilidad y facilidad de cómputo para grandes volúmenes de datos de manera rápida y eficiente, así como tolerancia a fallos. Esta flexibilidad viene dada en gran parte por la arquitectura maestro/esclavo de los clústers *Spark*. En ellos, existe un nodo encargado de gestionar los recursos (el nodo maestro). Este nodo se ejecuta sobre una máquina virtual *Java* que también posee un planificador, que se encarga de mapear las tareas de cómputo sobre los nodos de trabajo, los cuales poseen un proceso *Java* encargado de ejecutar las tareas que el maestro les envías, y además almacenan las particiones de datos a tratar (ver Figura 3).

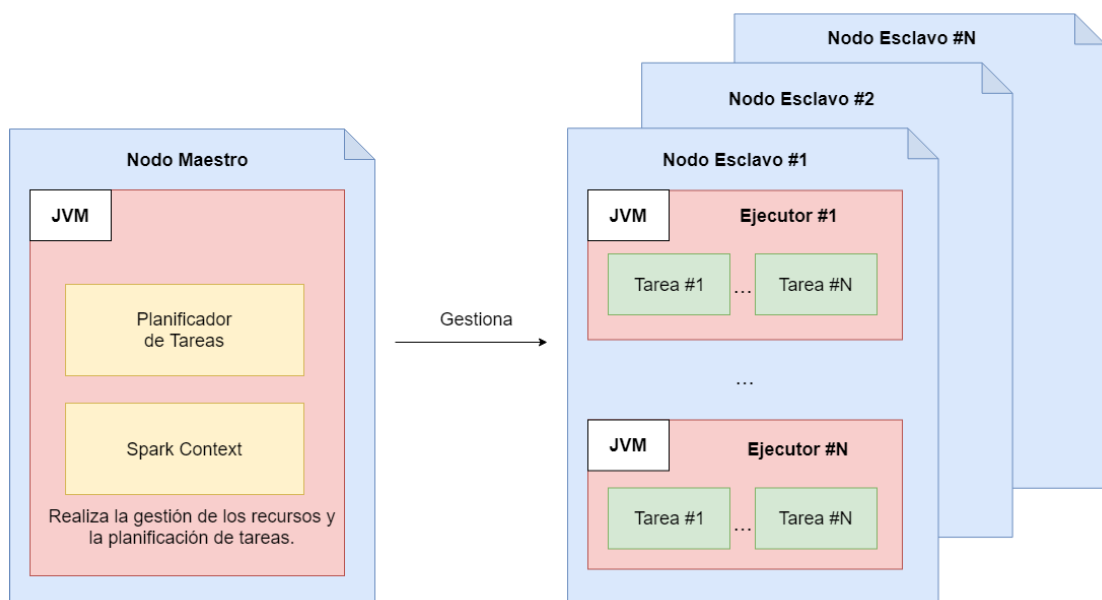


Figura 3: Ilustración gráfica de un clúster *Spark*, compuesto por un maestro y varios esclavos.

En nuestra implementación utilizamos el modelo *MapReduce* [38] que proporciona

el *framework Spark*. Los datos originales se agrupan en tuplas de tipo clave/valor, y las operaciones sobre las mismas se realizan basándose en dos operaciones fundamentales:

- Mapeado: Aplica operaciones a los elementos de las tuplas, generando nuevos valores.
- Reducción: Recopila los resultados intermedios obtenidos en la operación anterior para obtener el resultado final.

Para facilitar el tratamiento distribuido de los datos, el *framework Apache Spark* posee una serie de tipos de datos que proporcionan abstracción y tolerancia a fallos, llamados *Resilient Distributed Datasets* (*RDDs*). Los *RDDs* pueden verse como colecciones de datos abstractos de *Spark*, representados de forma serializable y distribuida, de modo que pueden ser repartidos entre los diferentes nodos del *cluster*. Esto da lugar a la posibilidad de realizar operaciones *MapReduce* de forma simultánea sobre cada subconjunto de los datos en cada nodo, acelerando el tiempo de cómputo. Las operaciones que se pueden realizar sobre los *RDDs* se dividen en dos tipos, dependiendo de si la salida es otro *RDD* o un valor o conjunto de valores:

- Transformación: Se aplica una operación a cada fila del *RDD* por separado, devolviendo un *RDD* con los valores retornados por la operación.
- Acción: Devuelve una serie de valores que pueden ser el propio *RDD*, o bien la salida de una transformación.

Con vistas a optimizar el modelo de ejecución, *Spark* proporciona un método de ejecución perezosa, es decir, las transformaciones son encoladas hasta que una acción no necesita ser invocada, momento en el que *Spark* ejecuta un proceso estructurado para la recolección del resultado. Esta acción se corresponde con la salida de un grafo dirigido acíclico, cuyos nodos intermedios serán las transformaciones a realizar sobre el conjunto de datos de entrada. A nivel computacional, esto genera un trabajo o *job* que el planificador se encarga de repartir entre los nodos esclavos. A su vez, este trabajo se

divide en *stages*, es decir, fragmentos encargados de ejecutar la misma transformación y, a su vez, estos fragmentos se particionan en forma de tareas, que serán las que finalmente ejecuten los nodos esclavos del clúster.

Para facilitar aún más el tratamiento de grandes volúmenes de datos (sobre todo si éstos poseen varios niveles de información), *Spark* proporciona un nivel de abstracción de datos superior al *RDD*, los denominados *DataFrames*. Estos *DataFrames* se organizan de forma que cada dato ocupa una fila del mismo pero, además, estas filas se organizan en un conjunto de columnas identificadas por su nombre, facilitando así el acceso a cada columna (al igual que en una base de datos relacional). Se adjunta una comparativa mas completa de ambos servicios en la tabla 3.

	Estructura de datos	
Características	**RDD**	**DataFrame**
Representación de datos	Abstracta	**Esquemática**
Nomenclatura de columnas	No permitida	**Permitida**
Filtros	Al conjunto de datos	**Flexibles (Por columna)**
Operaciones	Mapeo / Reducción	**Mapeo / Reducción / UDF**

Tabla 3: Características de las diferentes estructuras de datos distribuidas de *Spark*.

Teniendo esto en cuenta, se puede resumir el funcionamiento de un clúster *Spark* mediante una serie de pasos sencillos:

1. El nodo principal ejecuta un proceso *Java* que posee el contexto de la aplicación y el planificador de tareas.

2. El contexto, apoyado en el planificador, realiza una partición de los datos de entrada, asignando un subconjunto de datos a cada nodo esclavo. El tamaño del subconjunto depende de dos aspectos principales: el tamaño del bloque y cómo se encuentran mapeados los datos de entrada. Por último, se generan los ejecutores de tareas en los nodos esclavo, los cuales realizan operaciones sobre sus respectivos conjuntos de datos asignados.

3. Basándose en el método de ejecución perezosa, cuando en el código de la aplicación se ejecuta una acción, el grafo de dependencias generado es enviado

a los ejecutores etapa a etapa. *Spark* realiza esta ejecución de manera eficiente, asignando cada etapa a los ejecutores que poseen el subconjunto de datos sobre el que se realizan las operaciones.

4. Cuando una etapa termina con éxito, el contexto asigna la siguiente a través del planificador (si es una transformación), o bien traslada el resultado generado al nodo principal (si se trata de una acción).

El Algoritmo 1 muestra un pseudocódigo sencillo de las tareas anteriormente descritas.

Algorithm 1 Proceso iterativo de una tarea en *Spark*

```
1:  procedure OPERACION MapReduce EN Spark
2:      Particiones ← Spark.paralelizarDatos()
3:      t ← 0
4:      while t < n_iteraciones do
5:          enviarDatosIteracionAnterior()
6:          foreach particion ∈ Particiones do
7:              particion.accion()
8:          end for
9:          recogerDatosIteracion()
10:         t ← t + 1
11:     end while
12: end procedure
```

4.2. Procesamiento de imágenes hiperespectrales mediante *autoencoders*

Una imagen hiperespectral $\mathbf{X} \in \mathbb{R}^{n_1 \times n_2 \times n_{bandas}}$ se puede considerar como un espacio de datos tridimensional, donde la altura y anchura de la imagen se denotan respectivamente como n_1 y n_2 (siendo $n1 \times n_2$ el número de píxeles de la imagen) y el número de bandas de imagen se denota como n_{bandas}. A la hora de procesar la

imagen, puede entenderse como un conjunto de píxeles $[\mathbf{x}_{1 \ldots n_1 \times n_2}]$, donde cada píxel $\mathbf{x}_i = [x_{1 \ldots n_{bandas}}]$ contiene la firma espectral del material o materiales asociados a dicho píxel. En este ámbito, el objetivo de un método de reducción dimensional, como el *autoencoder*, es obtener para cada píxel $\mathbf{x}_i$, un vector de dimensiones reducidas ($\mathbf{c}_i$) capaz de conservar la mayor cantidad de información posible en un espacio dimensional menor.

La arquitectura del *autoencoder* considerado en el presente trabajo es del tipo *tied-autoencoder*, estas topologías consisten en un conjunto de capas intermedias junto a una capa de entrada y una de salida donde las capas intermedias se encargan de ajustar los datos hasta alcanzar el *ratio* de compresión deseado en la capa intermedia. Dicha capa, llamada cuello de botella, contendrá la representación dimensional reducida de los datos de entrada. Este tipo de topologías proporcionan una serie de características que las hacen muy deseables a la hora de solucionar problemas de reducción dimensional. Al utilizar el proceso de compresión y descompresión la misma estructura de capas invertida, resulta más sencillo ajustar los parámetros de la red y se reduce la probabilidad de *overfitting*. Esto facilita el proceso de entrenamiento utilizado para minimizar el error de reconstrucción de la entrada. En la Figura 4 se muestra una representación gráfica de la estructura descrita.

Esta arquitectura, de tipo *multi-layer perceptron* (MLP), aplica en cada capa una transformación a los datos de entrada $\mathbf{x}_i^{(l)}$, de tal forma que la salida de la capa siguiente $\mathbf{x}_i^{(l+1)}$ viene influenciada por un conjunto de pesos $\mathbf{W}^{(l)}$ y una serie de sesgos $b^{(l)}$, siguiendo la ecuación (1).

$$\mathbf{x}_i^{(l+1)} = \mathscr{H}\left(\mathbf{x}_i^{(l)} \cdot \mathbf{W}^{(l)} + b^{(l)}\right), \qquad (1)$$

De esta ecuación podemos deducir que $\mathbf{x}_i^{(l+1)}$ es el valor predicho de $\mathbf{x}_i$ a su paso por la capa $l+1$, obtenido tras aplicar los sucesivas productos escalares de las capas anteriores. Cada producto escalar pasa por una función de activación de tipo *ReLU (Rectified Linear Unit)* [52], cuyo funcionamiento viene dado por la ecuación (2).

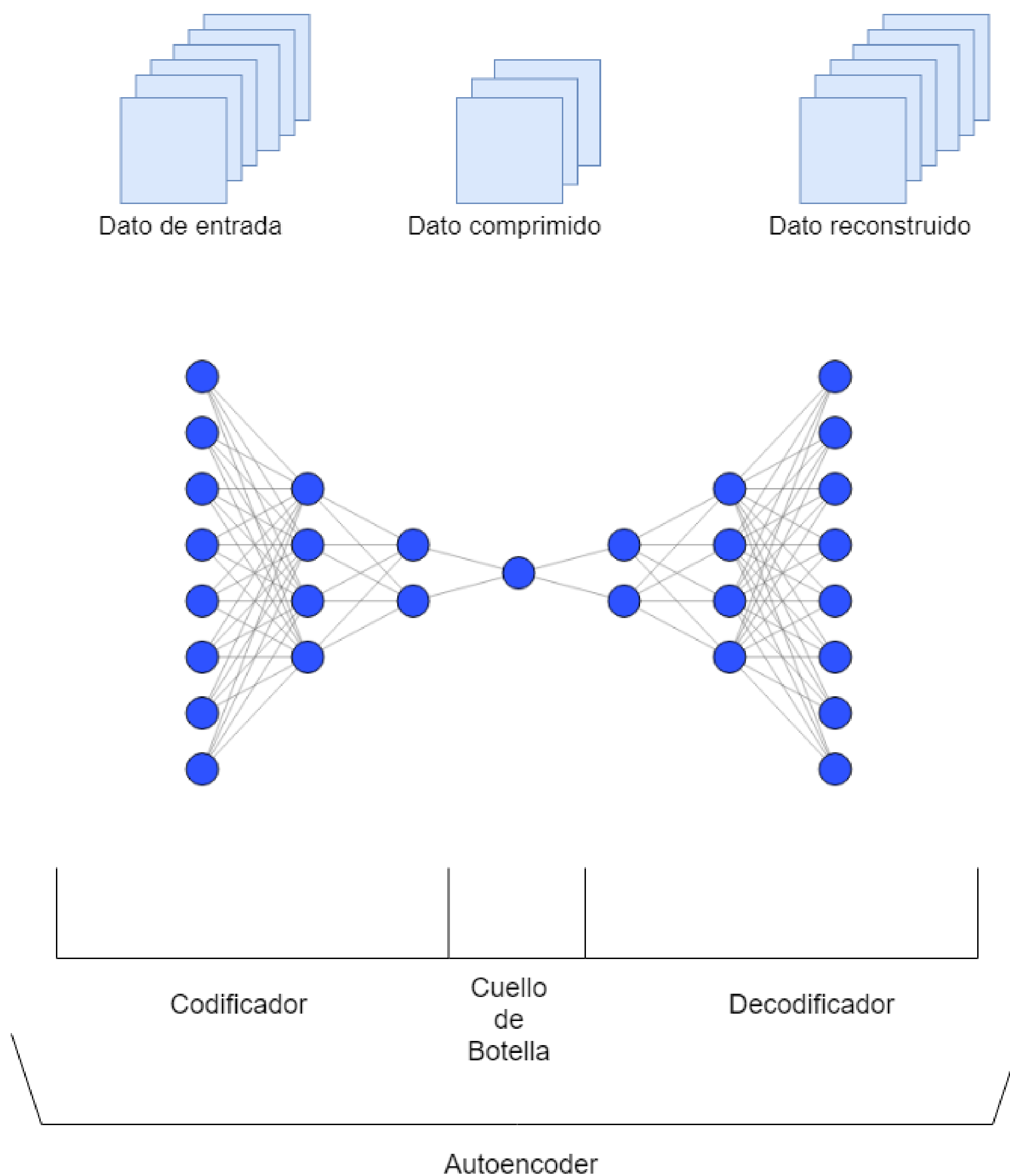


Figura 4: Representación gráfica de un *autoencoder*.

$$\mathscr{H}(x) = max(0,x). \tag{2}$$

Por tanto, la función para obtener la característica espectral k en $\mathbf{x}_i^{(l+1)}$ se puede definir como:

$$x_{i,k}^{(l+1)} = \mathscr{H}\left(\sum_{j=1}^{n^{(l-1)}} \left(x_{i,j}^{(l)} \cdot w_{k,j}^{(l)}\right) + b^{(l)}\right). \quad (3)$$

El *autoencoder* aplica un procesamiento en dos pasos a cada píxel de entrada $\mathbf{x}_i$. En el primer paso (codificación), realiza un mapeo de los datos del número de bandas original ($\mathbb{R}^{n_{bandas}}$) a un espacio latente reducido ($\mathbb{R}^{n_{nuevo}}$). Las capas codificadoras proyectan los datos a una dimensión menor siguiendo las ecuaciones (1) y (3) hasta llegar a la capa intermedia. Este cuello de botella contiene el resultado de proyectar cada píxel de entrada $\mathbf{x}_i \in \mathbb{R}^{n_{bandas}}$ a su espacio latente representado en $\mathbf{c}_i \in \mathbb{R}^{n_{nuevo}}$. En este sentido, conviene destacar que el *autoencoder* no solamente permite reducir las dimensiones de una imagen hiperespectral, sino que también permite ampliarlas o extenderlas.

El segundo paso realiza una función de decodificación de los datos. En este paso, se intenta reconstruir el píxel original basado en la información contenida en el cuello de botella [5]. Para ello, cada capa del decodificador realiza la reconstrucción del vector codificado $\mathbf{c}_i$, hasta llegar a la ultima capa, cuya salida es el píxel reconstruido $\mathbf{x}'_i$. El proceso de codificación-decodificación viene dado por la ecuación (4).

$$\begin{aligned} \mathbf{c}_i \leftarrow \text{For } l \text{ in } n_{codificador}&: \mathbf{x}_i^{(l+1)} = \mathscr{H}\left(\mathbf{x}_i^{(l)} \cdot \mathbf{W}^{(l)} + b^{(l)}\right) \\ \mathbf{x}'_i \leftarrow \text{For } ll \text{ in } n_{decodificador}&: \mathbf{c}_i^{(ll+1)} = \mathscr{H}\left(\mathbf{c}_i^{(ll)} \cdot \mathbf{W}^{(ll)} + b^{(ll)}\right) \end{aligned} \quad (4)$$

Para mejorar la representación latente obtenida por el *autoencoder*, se realiza un proceso iterativo de ajuste de parámetros no supervisado donde el optimizador trata de minimizar el error de reconstrucción entre el píxel de entrada $\mathbf{x}_i$ y el de salida $\mathbf{x}'_i$. Normalmente, la función de error utilizada es el error cuadrático medio o *Mean Square Error* (MSE), dado por la ecuación (5).

$$E(\mathbf{X}) = \text{mín} \parallel \mathbf{X} - \mathbf{X}' \parallel_2 = \text{mín} \sum_{i=1}^{n_1 \cdot n_2} \parallel \mathbf{x}_i - \mathbf{x}'_i \parallel_2 . \quad (5)$$

4.3. Proceso de optimización del *autoencoder*

Como se ha mencionado anteriormente, es necesario un optimizador para ajustar de manera progresiva los parámetros del *autoencoder* hasta mejorar la salida proporcionada por el mismo. Al tratarse de una red neuronal artificial, el optimizador trata de encontrar una configuración de parámetros proporcionen el error mínimo para la ecuación (5).

Nuestro optimizador consiste en una función de evaluación que infiere el grado de precisión de la predicción realizada por la red neuronal para un conjunto de datos $\mathbf{X}$, lo cuál depende de una serie de variables de aprendizaje $\mathscr{W}$. Por tanto, esta función de aprendizaje se puede entender como $E(\mathbf{X}, \mathscr{W})$ y, al ser esta no lineal, la optimización debe realizarse de forma iterativa, mediante reducciones progresivas del error, hasta alcanzar una condición de parada.

Para ello, las funciones de optimización realizan un proceso de propagación hacia atrás del error, calculando el gradiente de cada parámetro (es decir, en qué dirección y cuánto debe desplazarse dicho parámetro para minimizar el error final). Este gradiente también puede interpretarse como la importancia del parámetro en el total del error. La actualización de $\mathscr{W}$ en una iteración t se calcula como indica la ecuación (6):

$$\begin{aligned} \mathscr{W}_{t+1} &= \mathscr{W}_t + \Delta\mathscr{W}, \\ \text{siendo } \Delta\mathscr{W} &= \mu_t \cdot \mathbf{p}_t, \end{aligned} \tag{6}$$

donde μ_t el ratio de aprendizaje *(learning rate)* y $\mathbf{p}_t$ la dirección de búsqueda de descenso [53]. El optimizador intenta obtener la dirección de búsqueda que le permita disminuir este error iterativamente, hasta alcanzar un mínimo global.

En el estado del arte existen varios algoritmos de optimización muy utilizados, como por ejemplo el algoritmo de descenso estocástico del gradiente (*SGD*) o una de sus extensiones mas conocidas, el algoritmo de estimación del momento adaptable (*Adam*). No obstante, la implementación propuesta utiliza el algoritmo de *Broyden–Fletcher–Goldfarb–Shanno* (*BFGS*) que, a pesar de tener una complejidad

computacional mayor que la de los algoritmos basados en el descenso estocástico del gradiente, es capaz de realizar el proceso de optimización de manera mas rápida y eficiente que estos, al inferir μ_t de forma automática basándose en los parámetros de entrada. El problema de este algoritmo es que, para redes con una cantidad significativa de parámetros, es necesaria una gran cantidad de memoria a la hora de calcular la matriz de gradientes [54]. Existe una variante llamada *L-BFGS* que limita el uso de memoria del algoritmo. En la siguiente sección se propone la implementación de un *autoencoder* distribuido con optimizador *L-BFGS* en entornos *cloud*.

5. IMPLEMENTACIÓN Y DESARROLLO

5.1. Flujo de ejecución distribuida del *autoencoder*

Comenzamos explicando el flujo completo que siguen los datos en el *autoencoder*, desde su carga en los nodos distribuidos y pasando por el conjunto de transformaciones que sufren durante el proceso de entrenamiento, haciendo hincapié en los beneficios inherentes a una implementación distribuida. La Figura 5 proporciona una visión general de este proceso.

El primer paso consiste en convertir el cubo de datos hiperespectral $\mathbf{X} \in \mathbb{R}^{n_1 \times n_2 \times n_{bandas}}$ en una matriz en la que cada fila representa un píxel con sus n_{bandas} asociadas. Para ello se reduce el cubo a una matriz $\mathbf{X} \in \mathbb{R}^{n_{pixeles} \times n_{bandas}}$, siendo $n_{pixeles} = n_1 \times n_2$. Esta matriz $\mathbf{X}$ se particiona en un conjunto de subconjuntos P, siendo estos subconjuntos distribuidos entre los nodos esclavos en forma de *RDDs*.

Para aliviar el alto coste de memoria derivado del tamaño de los datos y las operaciones involucradas, *Spark* proporciona un hiperparámetro extra, el tamaño de bloque (*Tb*). Utilizando este parámetro, *Spark* agrupa un número *Tb* de píxeles en una única fila, que será procesada de forma conjunta para minimizar así el número de transformaciones.

Estas consideraciones nos permiten deducir que un subconjunto de datos no es mas que una matriz ${}^{(p)}\mathbf{D} \in \mathbb{R}^{n_{filas} \times (Tb \cdot n_{bandas})}$, en la que cada fila almacena un total de *Tb*

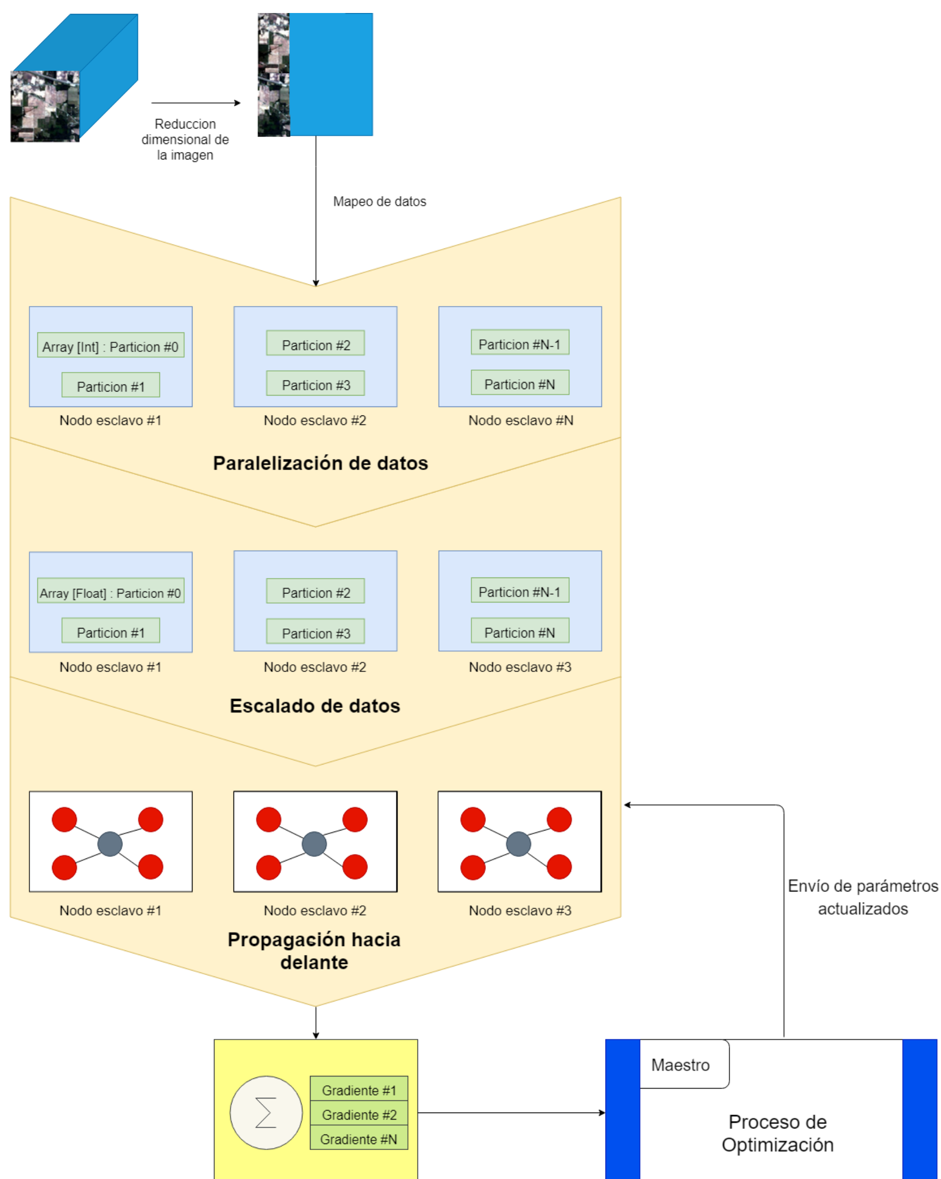


Figura 5: *Pipeline* completo del autoencoder distribuido. Primero, el cubo de datos se convierte en una matriz que se particiona entre los nodos esclavos. Estas particiones se duplican para obtener un par entrada/salida idéntico, que sera utilizado mediante un entrenamiento iterativo de optimización del error basado en el algoritmo *L-BFGS*.

píxeles consecutivos, dando lugar a un total de $Tb \cdot n_{bandas}$ píxeles por subconjunto. Estos subconjuntos son distribuidos entre los nodos, cuyos ejecutores les aplican operaciones en base a lo dispuesto por el planificador.

Para mejorar la precisión del *autoencoder*, los datos de cada subconjunto son escalados de forma distribuida. Para ello, se utiliza un escalado basado en el máximo

y mínimo global del conjunto total $\mathbf{X}$ y de la columna ${}^{(p)}\mathbf{D}$, como se recoge en la ecuación (7).

$$
\begin{aligned}
{}^{(p)}\hat{\mathbf{d}}_j &= \frac{{}^{(p)}\mathbf{d}_j - {}^{(p)}\mathbf{d}_{min}}{{}^{(p)}\mathbf{d}_{max} - {}^{(p)}\mathbf{d}_{min}} \\
{}^{(p)}\mathbf{d}_j &= {}^{(p)}\hat{\mathbf{d}}_j \cdot (\mathbf{x}_{max} - \mathbf{x}_{min}) + \mathbf{x}_{min}
\end{aligned} \tag{7}
$$

En (7), ${}^{(p)}\mathbf{d}_{max}$ y ${}^{(p)}\mathbf{d}_{min}$ son los mínimos locales de la columna y $\mathbf{x}_{max}$ y $\mathbf{x}_{min}$ son los mínimos globales del conjunto de la imagen.

Una vez tenemos el conjunto de datos particionado y escalado, se procede a entrenar el *autoencoder* distribuido. Para entender como funciona este proceso, es importante describir la topología del modelo. Como se expuso anteriormente, el modelo se puede dividir en: (i) un codificador que consta de dos capas ($l^{(1)}$ y $l^{(2)}$), donde la entrada de la capa $l^{(1)}$ es un píxel $\mathbf{x}_i$, además de una tercera capa (el cuello de botella $l^{(3)}$), donde se recogen los píxeles comprimidos y (ii) un decodificador, compuesto por dos capas: $l^{(4)}$ y $l^{(5)}$. La salida de $l^{(5)}$, denotada como $\mathbf{x}'_i$, es la reconstrucción del píxel $\mathbf{x}_i$. La finalidad del *autoencoder* es minimizar el error de reconstrucción entre $\mathbf{x}_i$ y $\mathbf{x}'_i$. La topología (número de neuronas por capa) de la arquitectura propuesta se muestra en la Tabla 4.

Capa	$l^{(1)}$	$l^{(2)}$	$l^{(3)}$	$l^{(4)}$	$l^{(5)}$
Neuronas por capa	n_{bandas}	140	60	140	n_{bandas}

Tabla 4: Topología del *autoencoder* propuesto.

Llegados a este punto, insistimos en el hecho de que el rendimiento de la red se vería significativamente afectado si el proceso de optimización se llevase a cabo con un conjunto de píxeles apilados. Por tanto, una vez realizado el grueso de las operaciones elementales, los píxeles son desapilados para introducirlos en la red.

Teniendo estos aspectos en cuenta, podemos entender el proceso de entrenamiento como un proceso compuesto por t iteraciones, siguiendo el modelo de propagación

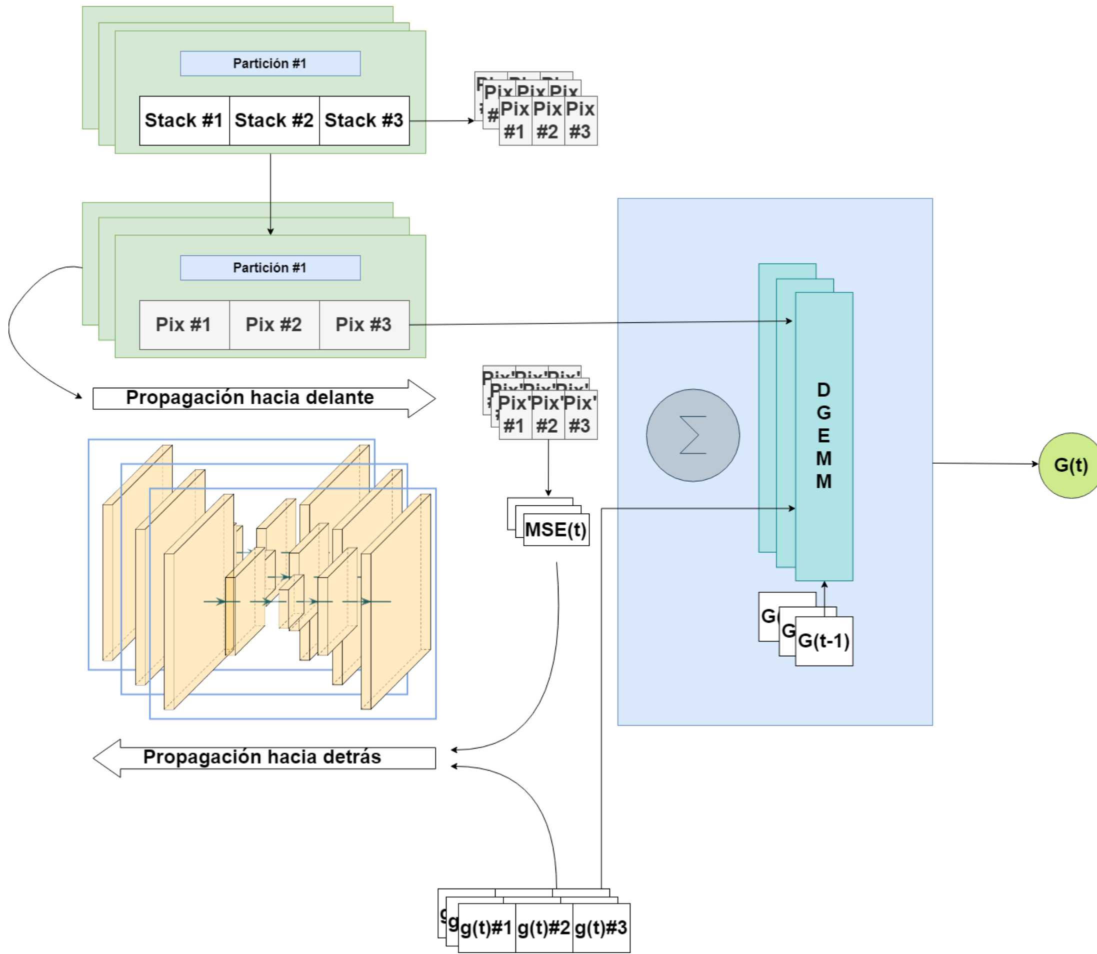


Figura 6: Proceso de entrenamiento distribuido en el instante *t*, tras desapilar los píxeles contenidos en un subconjunto.

hacia adelante y hacia atrás de las redes neuronales. Al tratarse de un proceso distribuido, cada ejecutor calcula su propia matriz de pesos, propagando la entrada $^{(p)}\mathbf{X}$ a través de las capas y comparándola con la salida de la capa $l^{(5)}$, siguiendo la ecuación (5) y obteniendo (para cada partición) su *MSE* en el instante *t*, según la fórmula: $^{(p)}\mathrm{MSE}_t = E(^{(p)}\mathbf{X}, \mathscr{W}_t)$.

Acto seguido, se realiza el cálculo del gradiente local, propagando hacia atrás la señal de error obtenida y obteniendo (para cada partición y para cada instante) una matriz $^{(p)}\mathbf{G}_t$.

Por último, el nodo maestro se encarga de recoger las matrices de cada partición y reducirlas en una única matriz $\Delta\mathscr{W}_t$. Esta matriz indica (para cada neurona de la red,

en cada instante) el impacto que dicha neurona tiene en el cálculo del error, lo cuál determina en qué medida debe ser modificada para minimizar al máximo el error.

Si tenemos en cuenta que existe un número P de subconjuntos, cada conjunto desapilado se puede entender como ${}^{(p)}\mathbf{X} \in \mathbb{R}^{(Tb \cdot n_{filas}) \times n_{bandas}}$, conteniendo $Tb \times n_{filas}$ píxeles normalizados. Cada capa de la red está formada por $n^{(l)}_{neuronas}$. La salida en la capa $l+1$ para la partición p se denota como ${}^{(p)}\mathbf{X}^{(l+1)}$, y se puede calcular modificando la ecuación (1) de la siguiente forma:

$$ {}^{(p)}\mathbf{X}^{(l+1)} = \mathscr{H}\left({}^{(p)}\mathbf{X}^{(l)} \cdot \mathbf{W}^{(l)} + b^{(l)}\right), \quad (8) $$

dónde:

- ${}^{(p)}\mathbf{X}^{(l+1)} \in \mathbb{R}^{(Tb \times n_{filas}) \times n^{(l)}_{neuronas}}$ representa la salida de la capa l con tamaño $(Tb \cdot n_{filas}) \times n^{(l)}_{neuronas}$.

- ${}^{(p)}\mathbf{X}^{(l)} \in \mathbb{R}^{(Tb \times n_{filas}) \times n^{(l-1)}_{neuronas}}$ es la matriz de entrada de la capa l, que contiene $(Tb \cdot n_{filas})$ píxeles en el espacio formado por las neuronas de la capa $l-1$, formado por $n^{(l-1)}_{neuronas}$.

- $\mathbf{W}^{(l)} \in \mathbb{R}^{n^{(l-1)}_{neuronas} \times n^{(l)}_{neuronas}}$ es la matriz de pesos que conecta las capas $lyl-1$ y $b^{(l)}$ son los sesgos asociados a la capa l.

- $\mathscr{H}$ es la función de activación. En nuestra implementación, utilizamos la función $\mathrm{ReLU}(x) = \mathrm{máx}(0, x)$.

Tras propagar los datos a través de las capas en base a la ecuación (8), se obtiene la salida reconstruida ${}^{(p)}\mathbf{X}'$ de la partición p. Esta salida se compara con la entrada original ${}^{(p)}\mathbf{X}$ para calcular el *MSE* de la reconstrucción. El nodo maestro recoge los datos de error de cada partición, y calcula el error global como la media de los mismos, como se indica en la ecuación (9):

$$
\begin{aligned}
{}^{(p)}\mathrm{MSE}_t &= \frac{1}{(Tb \times n_{filas})} \sum_{k=1}^{(Tb \times n_{filas})} \|{}^{(p)}\mathbf{x}_k - {}^{(p)}\mathbf{x}'_k\|_2 \\
\mathrm{MSE}_t &= \frac{1}{P} \sum_{p=1}^{P} {}^{(p)}\mathrm{MSE}_t,
\end{aligned} \tag{9}
$$

dónde ${}^{(p)}\mathbf{x}_k \in {}^{(p)}\mathbf{X}$ y ${}^{(p)}\mathbf{x}'_k \in {}^{(p)}\mathbf{X}'$ son, respectivamente, los datos originales y los reconstruidos en la partición.

Llegados a este punto, el error de cada partición se propaga hacia atrás para calcular el gradiente ${}^{(p)}\mathbf{G}_t$ de cada partición en el instante t. Es posible calcular, para cada neurona, el impacto de esta en el error, basándonos en la derivada de la función de activación:

$$
\mathscr{H}'(x) = \begin{cases} 0, & \text{si } x \leq 0 \\ 1, & \text{si } x > 0 \end{cases} \tag{10}
$$

Podemos estimar el impacto de todas las neuronas mediante ${}^{(p)}g_t^L = [{}^{(p)}g_t^{(1)}, \cdots, {}^{(p)}g_t^{(5)}]$, donde ${}^{(p)}g_t^L$ denota el impacto de cada neurona de la capa l en el instante t.

Para calcular la partición de gradientes, se utiliza una operación básica del álgebra lineal, la multiplicación de matrices en doble precisión (*DGEMM*) [55]. Esta multiplicación se realiza de la siguiente forma:

$$
\mathbf{C} = \alpha * \mathbf{A} * \mathbf{B} + \beta * \mathbf{C}. \tag{11}
$$

Si especificamos cada término de esta operación para que concuerde con el cálculo del gradiente, se deriva la siguiente ecuación:

$$
{}^{(\mathbf{p})}\mathbf{G_t} = \frac{1}{n_{filas}} * {}^{(\mathbf{p})}\mathbf{X} * {}^{(\mathbf{p})}\mathbf{g_t^L} + 1 * {}^{(\mathbf{p})}\mathbf{G_{t-1}}. \tag{12}
$$

Por último, y al igual que ocurre con el error, la matriz de gradientes $\mathbf{G_t}$ se calcula

en el nodo maestro como la media de las matrices gradiente de cada partición. El proceso completo se ilustra en las Figuras 5 y 6.

Una vez calculada la matriz de gradientes, esta matriz se utiliza en el paso de optimización. Como se mencionó anteriormente, el optimizador elegido (*L-BFGS*) es una variante del algoritmo *BFGS* que limita el uso de recursos de memoria para minimizar el error en cada iteración.

5.2. Implementación distribuida del *autoencoder*

Una vez analizado el flujo que siguen los datos durante el proceso de entrenamiento, se procede a explicar el funcionamiento interno de la implementación propuesta, desde la generación del contexto de *Spark* hasta la evaluación de la solución entrenada.

En primer lugar, es importante introducir algunas de las características del código, pues influyen en la forma en la que éste ha sido programado. El lenguaje de programación escogido ha sido *Scala* [56]. A pesar de que *Spark* proporciona interfaces de programación en *R*, *Java* y *Python*. La presencia de tipos estáticos y sus carencias en el paradigma de la programación funcional nos llevaron descartar rápidamente *Java*. Por otro lado, *Scala* (a diferencia de *Python* y *R*) no sólo es compilado y ejecutado sobre una *JVM*, sino que, además, es el lenguaje sobre el que están escritas las librerías de *Apache Spark*, proporcionando mayor acceso y control sobre las librerías. Este punto es clave para la elección de este lenguaje, precisamente porque nos permite modificar de manera sencilla la librería *MLLib* para añadir las funciones y procedimientos necesarios para el funcionamiento del *autoencoder*. La Tabla 5 muestra las características generales de los diferentes lenguajes considerados antes de seleccionar *Scala* como base para nuestra implementación.

Otra decisión importante que influye en la estructura y diseño del código es la estructura elegida para almacenar los datos durante el proceso de entrenamiento. Ya se hizo hincapié en la sección 4 acerca de la existencia de dos estructuras de datos distribuidas dentro de *Spark*: los *RDDs* y los *DataFrames*, que a su vez traduce en

	Lenguaje			
Característica	**Scala**	**Java**	**Python**	**R**
Paradigma de programación	**Funcional**	Orientado a objetos	**Funcional**	**Funcional**
Tipado de datos	**Dinamico**	Estatico	**Dinamico**	**Dinamico**
Tipo	**Compilado**	**Compilado**	Interpretado	Interpretado
Integración de librerias de Spark	**Natural**	Wrapper	Wrapper	Wrapper

Tabla 5: Características de diferentes lenguajes de programación que poseen integración con *Spark*.

dos librerías de *machine learning* contenidas dentro de *MLLib*: una para soluciones basadas en *RDDs* y otra para la generación de flujos de análisis sobre datos contenidos en *DataFrames*. Nuestra implementación se basa en esta última debido a la mayor flexibilidad y control ofrecido por los *DataFrames*. En particular, la posibilidad de realizar consultas basadas en filtros ejecutados sobre columnas específicas da lugar a un mayor control en las operaciones de transformación de características, lo que genera mayor flexibilidad a la hora de trabajar con datos almacenados en dichos *DataFrames*.

Teniendo en cuenta estas consideraciones, a continuación se realiza un análisis en profundidad del código implementado. Lo primero es generar el contexto sobre el que se ejecutará la aplicación, como podemos observar en la Figura 3. Este contexto se almacena en el nodo maestro y es el encargado de gestionar las comunicaciones entre los diferentes nodos. Una vez tenemos el soporte sobre el que ejecutar la aplicación distribuida, el siguiente paso es realizar la carga del conjunto de datos desde el soporte de almacenamiento a la memoria. Para ello, la librería *MLLib* proporciona una serie de funciones capaces de leer datos almacenados en distintos formatos, como *csv* o *svm*, siendo este último el formato recomendado por *Spark* para el almacenamiento y carga de datos puesto que, gracias a su formato (columna: valor), omite las columnas cuyo valor es 0, liberando de esta forma espacio en el soporte de almacenamiento. En la Tabla 6 el ejemplo de un píxel almacenado en este formato.

Etiqueta	Características					
2	1:40	3:21	4:2	7:9	8:1	10:4

Tabla 6: Formato de un píxel almacenado en el formato *svm*. Se observa la disposición (banda:valor), así como la omisión de los valores que son 0, se referencia la Imagen 1 donde se aprecia que cada banda tiene asociado un valor de reflectancia.

Una vez que *Spark* ha realizado la carga de los datos a memoria, se ejecuta un preprocesado sobre los mismos con el objetivo de mejorar la estabilidad y el rendimiento del *autoencoder*. En primer lugar se escalan las características en el rango entre 0 y 1, utilizando el método de escalado propuesto en la ecuación (7). Este proceso se realiza de forma distribuida, aprovechando la distribución en forma de *DataFrame* que poseen los datos almacenados en memoria, paralelizando las operaciones de cómputo del máximo y el mínimo local de cada columna (gracias a las facilidades ofrecidas para el tratamiento de columnas). Al igual que ocurre con el cálculo de los parámetros, el proceso de escalado se lleva a cabo de forma paralela en las columnas, gracias al uso de lo que *Spark* denomina funciones definidas por el usuario o *user-defined functions* (*UDFs*). Estas *UDFs* permiten definir transformaciones por columnas a los *DataFrames*, aprovechando las operaciones relacionales que se pueden realizar sobre los mismos. La Figura 7 muestra el fragmento de código encargado de la transformación de los datos.

Una vez que los datos han sido escalados, es necesario dividir el conjunto de entrada en dos subconjuntos: uno utilizado para entrenar el modelo y otro para validar la solución. Dentro de la clase *dataframe*, existen métodos que (tomando como entrada una serie de porcentajes y un número que actuará como semilla del generador de números aleatorios) separan el conjunto de datos en dos conjuntos disjuntos, cuyo tamaño coincide con los porcentajes proporcionados.

Tras el preprocesado de los datos, se realizan las llamadas necesarias para comenzar el proceso de entrenamiento distribuido. Como dicho involucra una gran cantidad de clases e interacciones entre las mismas, se proporciona un esquema completo de la estructura interna del código en la Figura 8.

```
  override def transform(dataset: Dataset[_]): DataFrame = {
  transformSchema(dataset.schema, logging = true)
  val originalRange = (originalMax.asBreeze
     - originalMin.asBreeze).toArray
  val minArray = originalMin.toArray

  val reScale = udf { (vector: Vector) =>
    val scale = $(max) - $(min)

    // 0 in sparse vector will probably be rescaled to non-zero
    val values = vector.toArray
    val size = values.length
    var i = 0
    while (i < size) {
      if (!values(i).isNaN) {
        val raw = if (originalRange(i) != 0){
        (values(i) - minArray(i)) / originalRange(i)
        } else 0.5
        values(i) = raw * scale + $(min)
      }
      i += 1
    }
    Vectors.dense(values)
  }

  dataset.withColumn($(outputCol), reScale(col($(inputCol))))
}
```

Figura 7: Fragmento de código encargado de ejecutar el escalado paralelo de datos. La variable *rescale* contiene la *UDF* aplicada a cada columna. Estas funciones se reservan con la palabra reservada *UDF*. La ejecución de la misma se lleva a cabo mediante el uso de la función *withColumn*, contenida en la clase *dataframe* y en todos sus tipos heredados.

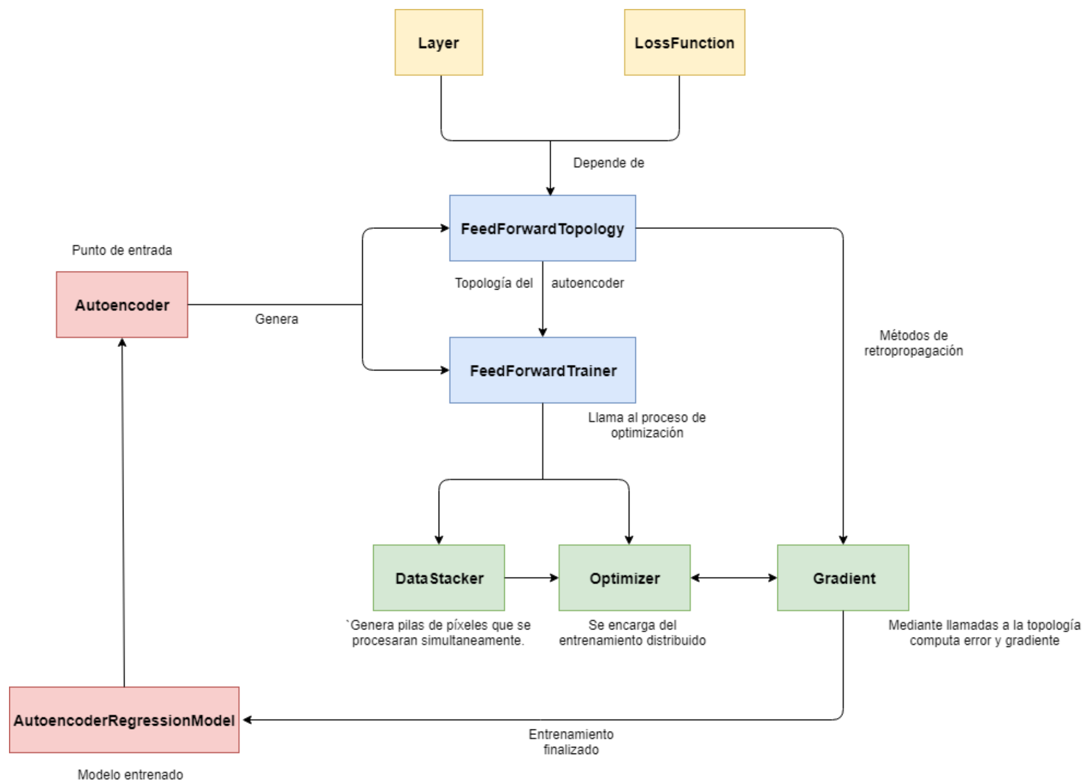


Figura 8: Esquema indicando la interconexión de las clases involucradas en el proceso de entrenamiento distribuido.

Como se puede apreciar en la Figura 8, la clase *Autoencoder* actúa como punto de entrada del proceso distribuido, tomando como parámetros la topología del *autoencoder*, el tamaño de bloque utilizado para apilar conjuntos de píxeles (que serán procesados de forma conjunta), el número máximo de iteraciones utilizado como condición de parada, y la semilla utilizada para la inicialización de los pesos. Una vez instanciada la clase, se realiza la llamada al método *train* de la misma. Este método se encarga, en primer lugar, de generar las etiquetas duplicando las características de entrada, para proceder a continuación a crear (utilizando los datos proporcionados en la topología) el conjunto de capas del *autoencoder* como una instancia de la clase *FeedForwardTopology*, encargada de llamar al proceso de optimización como una instancia de *FeedForwardTrainer*. El fragmento de código encargado de realizar estas acciones se muestra en la Figura 9.

La clase *FeedForwardTopology* general las capas de forma iterativa, alternando

las capas encargadas de realizar transformaciones afines [ver ecuación (1)] y las que contienen las funciones de activación no lineales [ver ecuación (2)]. Por último, se instancia la capa final encargada de calcular el error mediante la ecuación (5), como se puede apreciar en la Figura 10.

```
val myLayers = $(layers)
val labels = myLayers.last
val lpData = extractLabeledPoints(dataset)
val data = lpData.map(lp => LabelConverter.duplicateFeatures(lp))
val topology = FeedForwardTopology.autoencoder(myLayers)
val trainer = new FeedForwardTrainer(topology, myLayers(0)
    , myLayers.last)
```

Figura 9: Fragmento de código encargado de instanciar la topología y la clase encargada del entrenamiento.

```
def autoencoder(layerSizes: Array[Int]): FeedForwardTopology = {
   val layers = new Array[Layer]((layerSizes.length - 1) * 2)
   for (i <- 0 until layerSizes.length - 1) {
     layers(i * 2) = new AffineLayer(layerSizes(i)
       , layerSizes(i + 1))
     layers(i * 2 + 1) =
       if (i == layerSizes.length - 2) {
         new EmptyLayerWithSquaredError()
       } else {
         new FunctionalLayer(new ReluFunction())
       }
   }
   FeedForwardTopology(layers)
  }
}
```

Figura 10: Fragmento de código encargado de inicializar la topología.

Los dos tipos de capas anteriormente mencionadas están programadas como clases heredadas de *Layer*. Las transformaciones afines se recogen en la clase *AffineLayer* (ver Figura 11), que contiene los métodos necesarios para realizar el cálculo de la ecuación (1) y el cálculo del impacto de cada neurona y el gradiente en el proceso de retropropagación mediante el uso de transformaciones matriciales de tipo *DGEMM*

[ver ecuación (11)], mientras que la clase *FunctionalLayer* (ver Figura 12) contiene el código necesario para aplicar a cada elemento del vector salida de la capa i la función *ReLU*, descrita en la ecuación (2), y también para calcular el impacto de cada neurona en la salida en función de la derivada de *ReLU*, descrita en la ecuación (10). Tanto la función *ReLU* como su derivada se encuentran recogidas en la clase *ReluFunction*, cuyo código se muestra en la Figura 13.

Por último, la capa encargada del error también se encuentra programada como una clase heredada de *Layer*, que implementa la interfaz *LossFunction* para añadir un método *loss* a la clase encargada de realizar el cálculo de la señal de error *MSE* mediante la ecuación (5).

Llegados a este punto, es necesario también mencionar la función *ApplyInPlace*. Dado un conjunto de valores numéricos y una función de transformación del conjunto, dicha función se encarga de aplicar la operación de transformación sobre todos los valores del mismo. Existen dos variantes de esta función: una capaz de realizar transformaciones sobre los datos y otra que, recibidos dos conjuntos de datos, realiza una operación de reducción sobre los mismos (ver Figura 14).

Una vez aclarados estos aspectos, retomamos el proceso de entrenamiento en el punto en el que la función *train* de la clase *Autoencoder* realiza la llamada a la función homónima, alojada en la clase *FeedForwardTrainer*. Esta función se encarga, en primer lugar, de recoger en una variable los pesos de las capas inicializados mediante la función *randomWeights* de la clase *AffineLayer*, la cuál inicializa los pesos de forma aleatoria en el rango de valores 0 y 1. El código de esta función se ilustra en la Figura 15. Acto seguido, se genera una instancia de la clase *DataStacker* que se encarga de realizar un proceso de apilado de los datos de entrada mediante una función de mapeado encargada de generar bloques de píxeles que serán procesados de forma conjunta cuyo tamaño se indica mediante el hiperparámetro *BlockSize*, dicho parámetro influirá en la velocidad de la computación al realizar de forma simultanea el proceso de entrenamiento para un conjunto de píxeles. La funcionalidad de esta clase queda descrita en la Figura 16. Por último, esta función llama a la clase *Optimizer* para

```
private[scaladl] class AffineLayerModel private[scaladl] (
   val weights: BDV[Double],
   val layer: AffineLayer) extends LayerModel {
   val w = new BDM[Double](layer.numOut, layer.numIn
       , weights.data, weights.offset)
   val b = new BDV[Double](weights.data, weights.offset +
       (layer.numOut * layer.numIn), 1, layer.numOut)

   private var ones: BDV[Double] = null

   override def eval(data: BDM[Double], output: BDM[Double]): Unit = {
       output(::, *) := b
       BreezeUtil.dgemm(1.0, w, data, 1.0, output)
   }

   override def computePrevDelta(
       delta: BDM[Double],
       output: BDM[Double],
       prevDelta: BDM[Double]): Unit = {
       BreezeUtil.dgemm(1.0, w.t, delta, 0.0, prevDelta)
   }

   override def grad(delta: BDM[Double], input: BDM[Double]
       , cumGrad: BDV[Double]): Unit = {
   // compute gradient of weights
   val cumGradientOfWeights = new BDM[Double](w.rows, w.cols
       ,cumGrad.data, cumGrad.offset)
   BreezeUtil.dgemm(1.0 / input.cols, delta
       , input.t, 1.0, cumGradientOfWeights)
   if (ones == null || ones.length != delta.cols) ones =
       BDV.ones[Double](delta.cols)
   // compute gradient of bias
   val cumGradientOfBias = new BDV[Double](cumGrad.data, cumGrad.offset
   + w.size ,1, b.length)
   BreezeUtil.dgemv(1.0 / input.cols, delta, ones, 1.0
       , cumGradientOfBias)
   }
}
```

Figura 11: Funciones realizadas en el interior de las capas afines.

```
private[scaladl] class FunctionalLayerModel private[scaladl]
  (val layer: FunctionalLayer) extends LayerModel {

   // empty weights
   val weights = new BDV[Double](0)

   override def eval(data: BDM[Double], output: BDM[Double]): Unit = {
       ApplyInPlace(data, output, layer.activationFunction.eval)
   }

   override def computePrevDelta(
       nextDelta: BDM[Double],
       input: BDM[Double],
       delta: BDM[Double]): Unit = {
       ApplyInPlace(input, delta, layer.activationFunction.derivative)
       delta :*= nextDelta
   }

   override def grad(delta: BDM[Double], input: BDM[Double]
    ,cumGrad: BDV[Double]): Unit = {}
 }
```

Figura 12: Declaración de la función de activación no lineal.

comenzar el entrenamiento iterativo.

Esta clase genera una instancia específica del optimizador escogido (en nuestro caso, *L-BFGS*) compuesta de dos partes: la función a optimizar, de tipo *costFun*, y la clase encargada de calcular los valores optimizados con *L-BFGS*. Esta última clase se encuentra encapsulada dentro de la librería de cálculo numérico y algebraico de *Scala*, denominada *Breeze*, ejecutándose esta optimización al final de cada iteración de forma local en el nodo maestro sobre la salida del cálculo de la función de coste (de manera distribuida).

El proceso para calcular esta función de coste en cada iteración consiste en los siguientes pasos. En primer lugar se envía a cada nodo los pesos actualizados de la iteración anterior mediante la primitiva *broadcast* de *Spark*, y seguidamente se almacena una copia local de la clase *Gradient* que se encarga del proceso de retropropragación.

```
private[scaladl] class ReluFunction extends ActivationFunction {

  override def eval: (Double) =>
      Double = x => if (x > 0) x else 0

  override def derivative: (Double) =>
      Double = z => if (z > 0) 1 else 0
}
```

Figura 13: Funciones realizadas en el interior de las capas funcionales.

El cálculo del error y del gradiente distribuido se realiza mediante una operación de agregación en árbol (*treeAgreggate*) disponible en *Spark*. Este proceso se basa en dos funciones. La primera de ellas, llamada *seqOp*, se encarga de realizar un mapeo paralelo sobre cada partición de datos, con el fin de obtener la transformación deseada. Acto seguido, el grueso de las transformaciones obtenidas se reduce mediante la función *combOp*, efectuando una serie de sumas en árbol, las cuales aparecen ilustradas de forma gráfica en la Figura 17. El uso de *treeAgreggate* con respecto a otros métodos de reducción deriva en un menor envió de información al nodo maestro, lo cuál supone una gran ventaja cuando se realizan operaciones sobre gran cantidad de datos.

Específicamente, en la implementación propuesta, la operación *seqOp* obtiene el error y el gradiente correspondiente a una partición del conjunto de datos de forma local. Para ello, lo primero es calcular la salida de la red en base a la entrada. Este proceso se realiza mediante el método *forward* de la clase *FeedForwardModel*, que contiene la topología y los pesos. Este método se encarga, en primera instancia, de asignar memoria para una matriz que almacena las salidas para cada capa de la red y, acto seguido, calcula la salida de cada capa, aplicando sobre la entrada de la capa anterior las ecuaciones (1) y (2) mediante la función *eval* de las clases *AffineLayer* y *FunctionLayer*, cuyo funcionamiento se recoge en las figuras 11 y 12, respectivamente. Una vez obtenido el conjunto de salidas de cada capa, el método se encarga de calcular el error de reconstrucción aplicando la ecuación (5) –implementada en la

```
private[scaladl] object ApplyInPlace {

    // TODO: use Breeze UFunc
    def apply(x: BDM[Double], y: BDM[Double]
      , func: Double => Double): Unit = {
      var i = 0
      while (i < x.rows) {
        var j = 0
        while (j < x.cols) {
          y(i, j) = func(x(i, j))
          j += 1
        }
        i += 1
      }
    }

    // TODO: use Breeze UFunc
    def apply(
      x1: BDM[Double],
      x2: BDM[Double],
      y: BDM[Double],
      func: (Double, Double) => Double): Unit = {
      var i = 0
      while (i < x1.rows) {
        var j = 0
        while (j < x1.cols) {
          y(i, j) = func(x1(i, j), x2(i, j))
          j += 1
        }
        i += 1
      }
    }
}
```

Figura 14: Conjunto de métodos utilizados para la aplicación de funciones de mapeo y reducción sobre vectores.

```
def randomWeights(
  numIn: Int,
  numOut: Int,
  weights: BDV[Double],
  random: Random): Unit = {
  var i = 0
  val sqrtIn = math.sqrt(numIn)
  while (i < weights.length) {
    weights(i) = (random.nextDouble * 4.8 - 2.4) / sqrtIn
    i += 1
  }
}
```

Figura 15: Función encargada de inicializar los pesos.

clase *LossFunction*– entre la entrada y la salida de la última capa. A continuación, el método calcula la matriz de incidencia de cada neurona en el cálculo de error mediante propagación inversa, haciendo uso de la función *computePrevDelta*. El proceso de propagación hacia adelante y los cálculos del error y la matriz de incidencia se detallan en las figuras 18 y 19, respectivamente.

El último paso de esta operación consiste en el cálculo de los gradientes locales en función de los datos de entrada, de la matriz de incidencia de las neuronas en el error, y del conjunto local de gradientes actual en base a la ecuación (12).

Una vez realizado el conjunto de transformaciones que componen la función *seqOp*, se procede al cálculo del error y el gradiente global mediante el uso de una función de reducción *combOp* encargada de sumar los errores y gradientes de cada partición. El bloque de código encargado de gestionar la operación *treeAgreggate* se muestra en la Figura 20.

Para finalizar la fase de entrenamiento, el nodo maestro (una vez obtenidos los parámetros globales) calcula el error final dividiendo el error sumado entre el número de muestras y el gradiente final aplicando una regularización dependiente del gradiente anterior.

Una vez hemos obtenido el modelo entrenado, es necesario realizar una validación

```
def stack(data: RDD[(Vector, Vector)]): RDD[(Double, Vector)] = {
  val stackedData = if (stackSize == 1) {
    data.map { v =>
      (0.0,
        Vectors.fromBreeze(BDV.vertcat(
          v._1.asBreeze.toDenseVector,
          v._2.asBreeze.toDenseVector))
        ) }
  } else {
    data.mapPartitions { it =>
      it.grouped(stackSize).map { seq =>
        val size = seq.size
        val bigVector = new Array[Double](inputSize * size
          + outputSize * size)
        var i = 0
        seq.foreach { case (in, out) =>
          System.arraycopy(in.toArray, 0, bigVector
              , i * inputSize, inputSize)
          System.arraycopy(out.toArray, 0, bigVector,
            inputSize * size + i * outputSize, outputSize)
          i += 1
        }
        (0.0, Vectors.dense(bigVector))
      }
    }
  }
  stackedData
}
```

Figura 16: Función encargada de apilar las entradas para su procesamiento simúltaneo.

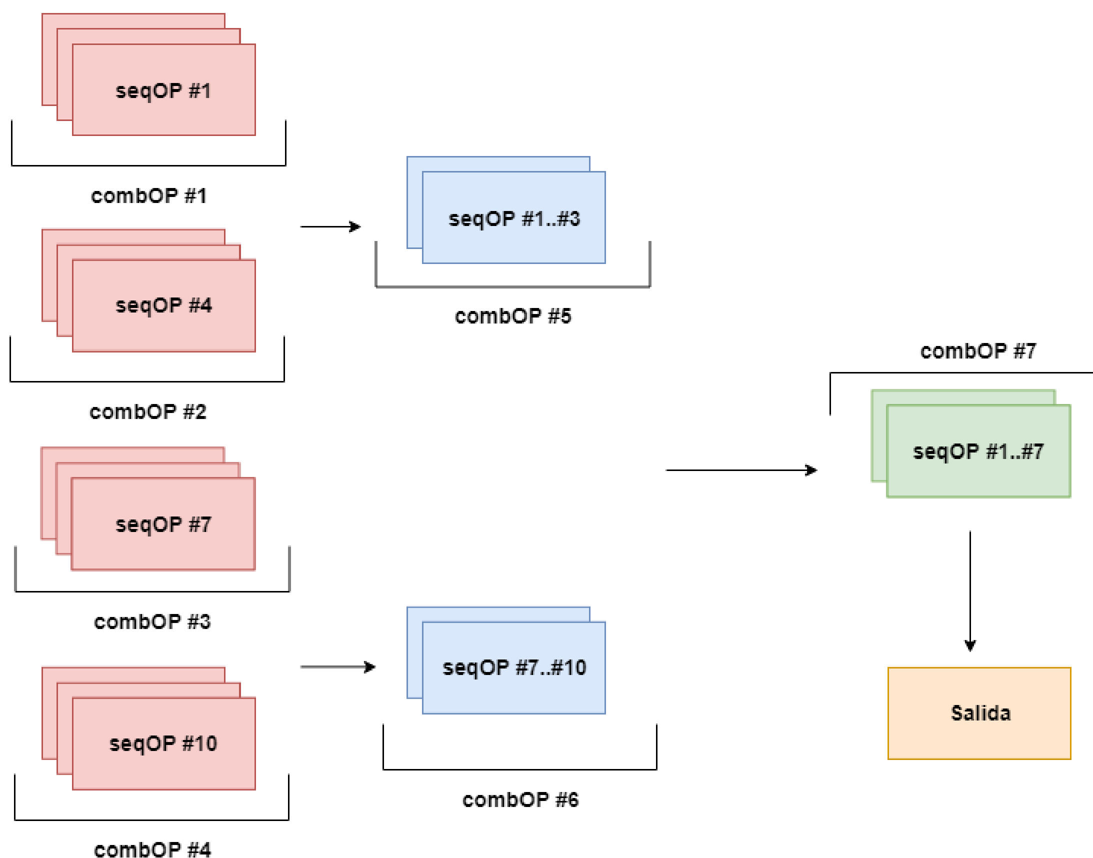


Figura 17: Proceso de mapeo-reducción cuando el conjunto de reducciones se realiza en forma de árbol.

del entrenamiento llevado a cabo, para lo cuál se ha implementado una clase evaluadora (*RegressionEvaluator*), encargada de calcular el *MSE* sobre el conjunto de validación y las salidas predichas del mismo. Esta predicción se lleva a cabo utilizando el método *predict*, recogido en la clase *Autoencoder*, que a su vez ejecuta el proceso *forward* mostrado en la Figura 18.

```
override def forward(data: BDM[Double]): Array[BDM[Double]] = {
  // Initialize output arrays for all layers.
  // Special treatment for InPlace
  val currentBatchSize = data.cols
  // TODO: allocate outputs as one big array and then
  // create BDMs from it
  if (outputs == null || outputs(0).cols != currentBatchSize) {
    outputs = new Array[BDM[Double]](layers.length)
    var inputSize = data.rows
    for (i <- 0 until layers.length) {
      if (layers(i).inPlace) {
        outputs(i) = outputs(i - 1)
      } else {
        val outputSize = layers(i).getOutputSize(inputSize)
        outputs(i) = new BDM[Double](outputSize, currentBatchSize)
        inputSize = outputSize
      }
    }
  }
  layerModels(0).eval(data, outputs(0))
  for (i <- 1 until layerModels.length) {
    layerModels(i).eval(outputs(i - 1), outputs(i))
  }
  outputs
}
```

Figura 18: Proceso de propagación hacia adelante de los datos.

6. RESULTADOS

6.1. Entorno de pruebas

El entorno para la realización de las pruebas de validación en el presente trabajo ha sido proporcionado por *Jetsream Cloud Services*[2] del instituto de tecnología de la universidad de Indiana. Este entorno utiliza *Atmosphere*[3] como plataforma de cómputo y *Openstack*[4] como entorno operacional.

Sobre este entorno, se ha reservado un conjunto dedicado de *hardware*, formado

[2]https://jetstream-cloud.org/
[3]https://www.atmosphereiot.com/platform.html
[4]https://www.openstack.org/

```
val loss = layerModels.last match {
  case levelWithError: LossFunction =>
    levelWithError.loss(outputs.last, target, deltas(L - 1))
  case _ =>
    throw new UnsupportedOperationException("Top layer is required
    to have objective.")
}
for (i <- (L - 2) to (0, -1)) {
  layerModels(i + 1).computePrevDelta(deltas(i + 1)
    , outputs(i + 1), deltas(i))
}
```

Figura 19: Cálculo de la señal de error y de la matriz de incidencia.

por una serie de nodos de cómputo homogéneos. Concretamente, se ha dispuesto de un conjunto de nueve nodos (Un maestro y ocho esclavos), cada uno con seis núcleos a 2.5 Ghz, 16 GB de RAM, y 60 GB de almacenamiento de ficheros por bloques. La interconexión de los nodos se realiza mediante *Apache Spark*, gracias a una red local capaz de transferir hasta 160 (4x40) Gb de datos por segundo. En la Figura 21 se muestra un esquema ilustrando el proceso de la asignación de recursos en el clúster *Spark*.

En cuanto al entorno *software*, cada nodo ejecuta *Spark* 2.1.1 sobre una *Java Virtual Machine* en *Java* 8 y *Ubuntu Server* 16.04 como sistema operativo. *Spark* posee un conjunto de librerías de propósito específico, incluyendo la librería de *machine learning* (*MLLib*[5]). Esta librería proporciona un conjunto de herramientas y funciones que actúan como base de la implementación propuesta. Nuestra implementación ha sido desarrollada en *Scala* 2.11, posteriormente compilada a código interpretable por una máquina virtual de *Java*. En cuanto a los cálculos matemáticos, destacamos el uso de dos librerías de *Java/Scala*: *Breeze* 0.12, que proporciona operaciones matemáticas y de cálculo numérico, y *Netlib* 1.1.2, que proporciona enlaces de bajo nivel para las operaciones de álgebra lineal, acelerando el cálculo de las mismas.

[5]https://spark.apache.org/mllib

```
val (gradientSum, lossSum) = data.treeAggregate
  ((Vectors.zeros(n), 0.0))(
  seqOp = (c, v) => (c, v) match { case ((grad, loss)
    , (label, features)) =>
        val l = localGradient.compute(
          features, label, bcW.value, grad)
        (grad, loss + l)
  },
  combOp = (c1, c2) => (c1, c2) match { case ((grad1, loss1)
    , (grad2, loss2)) =>
        axpy(1.0, grad2, grad1)
        (grad1, loss1 + loss2)
  })
```

Figura 20: Proceso de mapeo-reducción ejecutado sobre las diferentes particiones del conjunto de datos.

6.2. Imágenes hiperespectrales

Para proporcionar datos cuantitativos de la implementación, se ha evaluado el rendimiento de la misma sobre dos imágenes hiperespectrales. La primera de se denomina *Big Indian Pines* (*BIP*) y fue obtenida por el sensor aerotransportado *AVIRIS*, y la segunda se denomina *Hyperion Data Set* (*HDS*), compuesta por seis imágenes capturadas por el satélite *EO-1* de *NASA*. A continuación describimos en más detalle ambas imágenes:

- La imagen *BIP* (ver Figura 22) fue recogida en 1992 por el satélite *AVIRIS* [2] y recoge un conjunto de campos agrícolas del noroeste de Indiana. La imagen tiene un tamaño de 2678×614 píxeles con resolución espacial de 20 metros por píxel y resolución espectral de 220 bandas que cubren el espectro desde 400 a 2500 nm.

- La imagen *HDS* (ver Figura 23) es un conjunto de seis imágenes tomadas en 2016 por el espectrómetro *EO-1 Hyperion* [57], capaz de capturar un rango de 357 a 2576 nm mediante 220 bandas espectrales. El tamaño estándar de una imagen es $7{,}7 \times 42 km^2$, dando lugar a un conjunto de seis imágenes

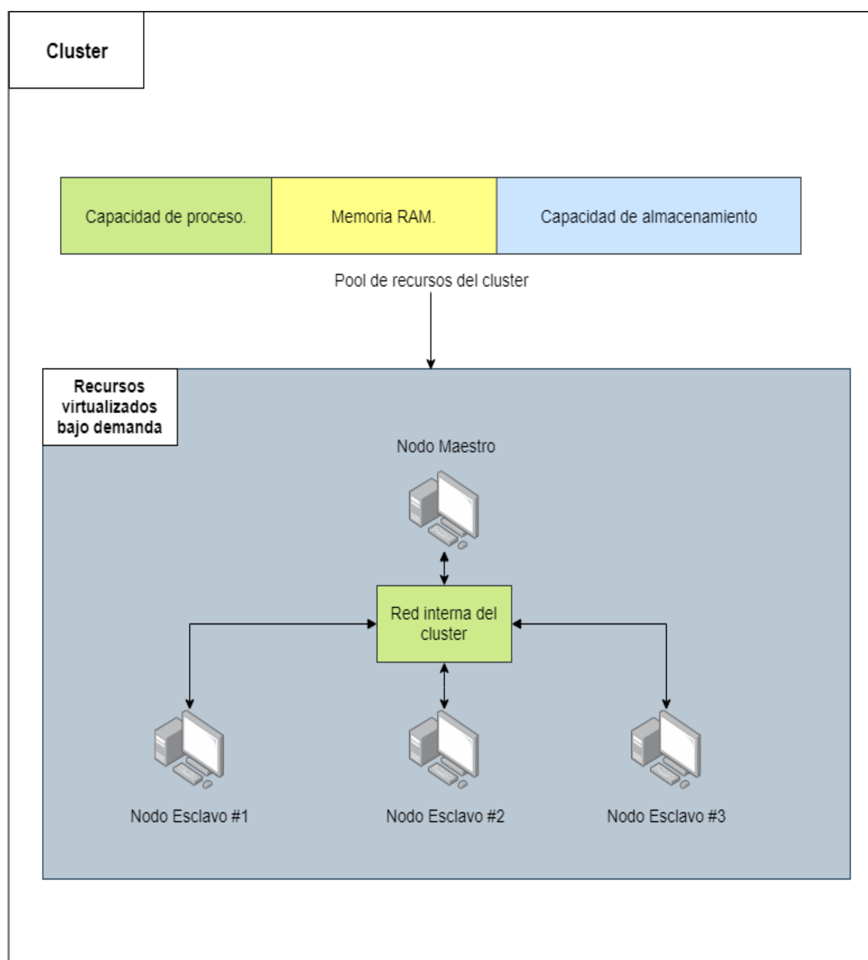


Figura 21: Proceso de asignación de recursos en un clúster *Spark*, utilizando una serie de recursos virtualizados en una plataforma *cloud*

.

que, una vez agrupadas, tienen un tamaño total de 20401×256 píxeles. Estas imágenes, proporcionadas por el Centro de Observación de Recursos Terrestres (EROS) en formato *GEOTIFF*, poseen un código identificador con formato *YDDDXXXML*, donde *DDD* nos indica el día del año en que la imagen fue capturada, *XXX* el sensor encargado de capturar dicha imagen, *M* la región sobre la que apunta el sensor (en las imágenes utilizadas *K* indicaría una región fuera del camino marcado) y *L*, la longitud de las imágenes. En concreto, las seis imágenes utilizadas poseen los identificadores siguientes: 065110KU, 035110KU, 212110KR, 247110KW, 261110KR y 321110KR.

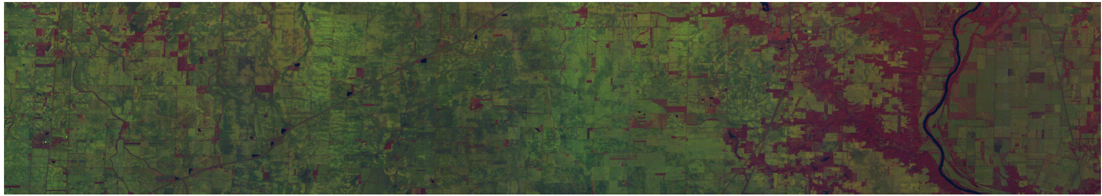


Figura 22: Representación en falso color de la imagen hiperespectral AVIRIS Big Indian Pines (BIP).

6.3. Experimentación realizada

Se han realizado tres experimentos para comprobar que la implementación distribuida, aprovecha de manera eficiente todas las ventajas inherentes a este paradigma de cómputo:

1. El primer experimento se ha realizado sobre el dataset *BIP*, y su objetivo es evaluar el rendimiento del *autoencoder* en base al número de nodos esclavos utilizados en *datasets* de tamaño grande. Para ello, se ha entrenado el *autoencoder* con 1, 2, 4, 8, 12 y 16 nodos esclavos, manteniendo los mismos parámetros de entrenamiento en todas las ejecuciones.

2. El segundo experimento intenta aprovechar el paralelismo en el interior de los nodos esclavos, utilizando una configuración de esclavos fija, se entrenan diferentes porcentajes del *dataset HDS*, para demostrar que la escalabilidad se mantiene lineal con respecto al crecimiento del tamaño del problema.

3. Por último, se ha llevado a cabo un tercer experimento (similar al primero, pero utilizando un *dataset* muy grande como es *HDS*). La configuración de nodos utilizada en este caso ha sido 2, 4, 8, 12 y 16, debido a la imposibilidad de ejecutar un problema semejante en un solo nodo por sus grandes requerimientos en cuanto a memoria.

Figura 23: Representación en falso color de la imagen hiperespectral Hyperion Data Set (HDS).

6.3.1. Experimento 1

En este experimento, se pone a prueba la capacidad de explotar el paralelismo a nivel de nodo del *autoencoder*. Para ello, utilizando la imagen *BIP*, se selecciona de forma aleatoria un 80 % de los píxeles de la imagen para entrenar la red, utilizándose el 20 % restante para validar el resultado de la experimentación. Como se ha mencionado anteriormente, se han utilizado diferentes configuraciones de nodos esclavos, todas con un nodo maestro encargada de distribuir y recibir los datos. Para obtener resultados estadísticamente relevantes, se han llevado a cabo cinco ejecuciones por configuración, siendo el resultado final de la experimentación la media y la desviación estándar de las

mismas.

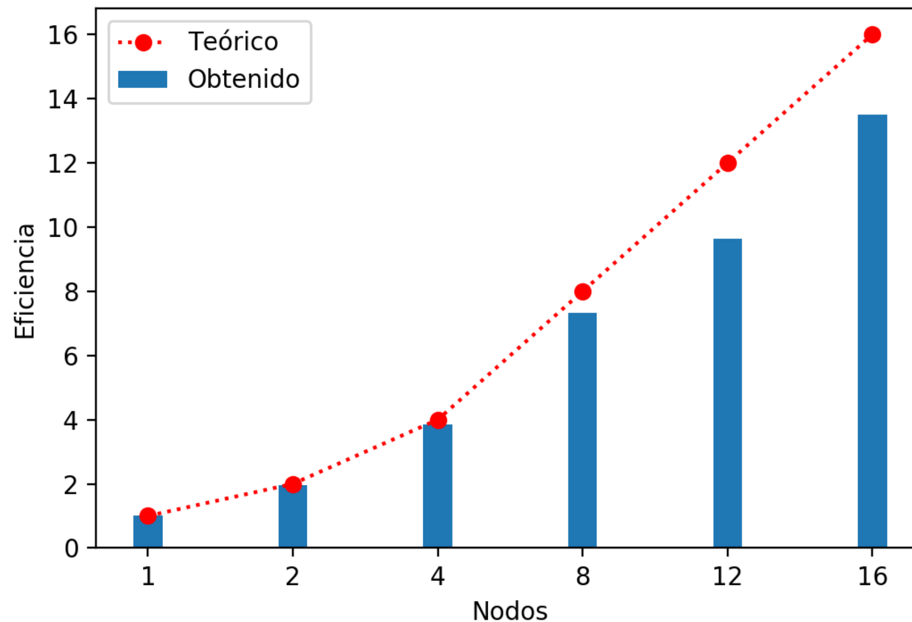


Figura 24: Escalabilidad de nuestra implementación en términos de eficiencia en base al número de nodos esclavos utilizados (experimento 1). La linea roja indica la eficiencia teórica, mientras que la barra azul indica la eficiencia obtenida.

Nodos	1	2	4	8	12	16
Error (MSE)	7.93e-5	7.92e-5	8.35e-5	9.51e-5	8.60e-5	8.35e-5
Tiempo (s)	17398.74	8991.12	4518.39	2354.91	1803.27	1288.69
Eficiencia	1	1.9308	3.8506	7.3882	9.6484	13.5011

Tabla 7: Resultados de la experimentación con la imagen *BIP* (Experimento 1). Se proporciona el error obtenido sobre el conjunto de validación, el tiempo de cómputo (en segundos), y la eficiencia con respecto a la versión serie.

En primer lugar analizamos la eficiencia de nuestra implementación, resultado de dividir el tiempo base (1 nodo) con el tiempo obtenido con varios nodos de cómputo. Como muestran los resultados en la Tabla 7 y en la Figura 24, la eficiencia se mantiene cercana al pico teórico en todas las configuraciones utilizadas, disminuyendo ligeramente a partir del uso de 12 nodos de cómputo. Esto es debido a la introducción de un tiempo de comunicación más significativo entre el maestro y los esclavos. Al aumentar el número de nodos (no el volumen de datos), estas comunicaciones generan

un cuello de botella, minimizando un tanto la ganancia en términos de tiempo de cómputo.

Es importante destacar también que el error se mantiene muy similar en todas las configuraciones de nodos utilizadas. Por tanto, aunque existen grandes mejoras en cuanto a tiempo de cómputo, estas no acarrean una pérdida de rendimiento a la hora de realizar la compresión de los datos hiperespectrales.

6.3.2. Experimento 2

Este segundo experimento evaluamos si el paralelismo interno de cada nodo esclavo escala de forma lineal con respecto al tamaño del problema. Para ello, se ha utilizado una única configuración de ejecución, con ocho nodos esclavos, sobre los que se han ejecutado entrenamientos con diferentes porcentajes de entrenamiento sobre la imagen *HDS* (en concreto, se han considerado porcentajes de entrenamiento que van desde el 20 % hasta el 80 %).

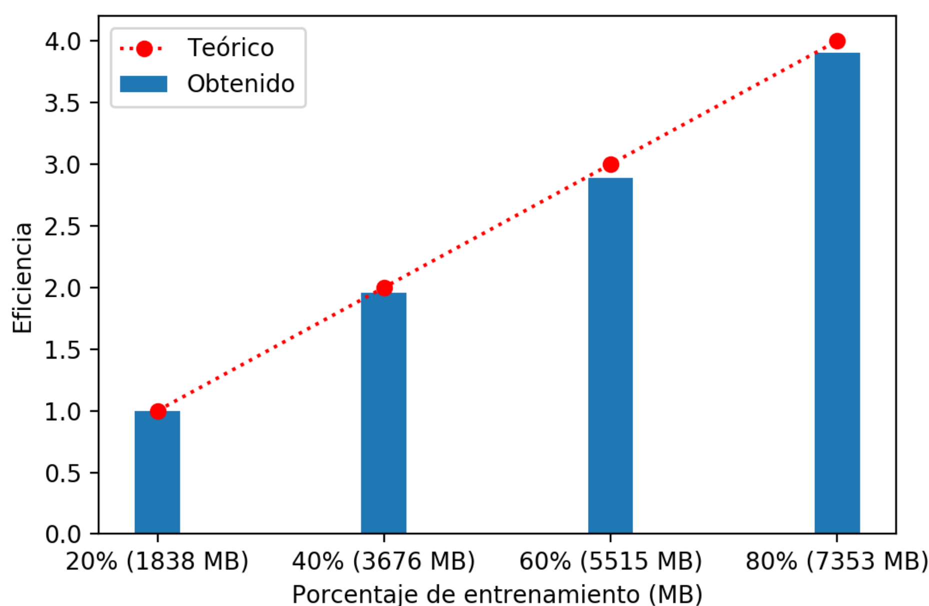


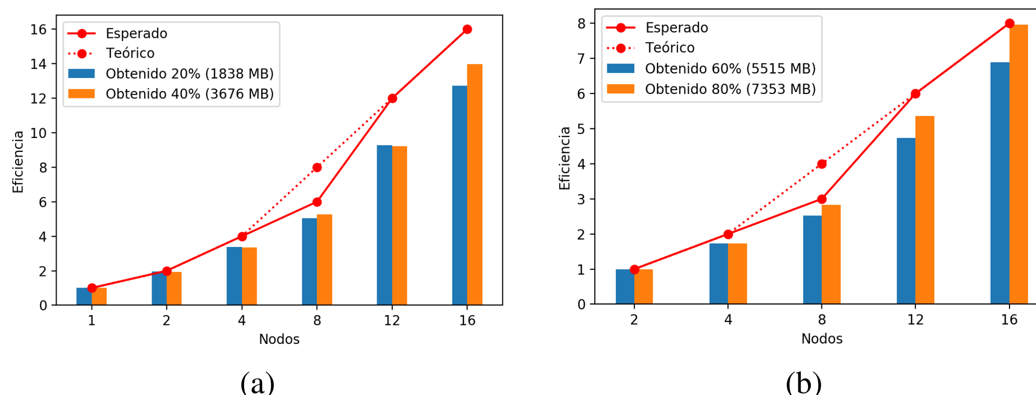
Figura 25: Escalabilidad de nuestra implementación en términos de eficiencia en base al tamaño del problema (experimento 2). La linea roja indica la eficiencia teórica, mientras que la barra azul indica la eficiencia obtenida.

La Figura 25 indica claramente que, cuando consideramos un crecimiento del tamaño del problema constante, el tiempo de cómputo del mismo también crece de forma manera lineal en base a esa constante. Esta observación es muy importante, pues confirma que nuestra implementación explota el paralelismo interno de cada nodo de forma completa, independientemente del tamaño del problema.

6.3.3. Experimento 3

Este tercer experimento proporciona datos acerca del comportamiento del *autoencoder* implementado cuando el tamaño del problema crece enormemente. Para ello, se ha utilizado la imagen hiperespectral HDS, con diferentes particionamientos de la imagen para controlar así el tamaño del problema y demostrar las mejoras en escalabilidad a medida que el tamaño del problema y el numero de nodos crecen. Como se puede apreciar en la Figura 26, se presenta por separado el resultado de la experimentación para porcentajes de entrenamiento de 20 % [ver Figura 26(a)] y 40 % (a) y 60 % y 80 % [ver Figura 26(b)]. En los resultados mostrados en la Figura 26(a), el *autoencoder* se ejecuta en primer lugar sobre un sólo nodo y se va aumentando el número de nodos hasta llegar a 16. En los resultados mostrados en la Figura 26(b), el *autoencoder* se ejecuta en primer lugar sobre dos nodos. Esto se debe a los porcentajes de entrenamiento más elevados en este último caso (60 % y 80 %), que requieren una mayor cantidad de memoria que la que posee un único nodo.

A diferencia de los experimentos 1 y 2, en este tercer experimento existe una tercera línea en las gráficas mostradas en la Figura 26: la eficiencia esperada (además de la obtenida y la teórica). Esta eficiencia esperada es la máxima obtenible por una configuración, y no coincide con la teórica debido a cómo *Spark* realiza el particionado de los datos en los ejecutores, expresándolo de forma cuantitativa. Por ejemplo, cuando tenemos una configuración con 8 nodos, tenemos un total de 48 núcleos de cómputo (6 por nodo). Esto supone que *Spark* puede realizar un total de 48 tareas simultáneas. Como el particionado automático de *Spark* ha resuelto que la tarea secuencial es divisible en un número de tareas T, tal que $48(n-1) < T < 48n$ (siendo n es un



(a) (b)

Figura 26: Escalabilidad de nuestra implementación en términos de eficiencia en base al número de nodos esclavos utilizados para problemas muy grandes (experimento 3). La linea roja indica la eficiencia teórica, mientras que la barra azul indica la eficiencia obtenida. Para una mejor lectura de los resultados, se presenta por separado el resultado de la experimentación para porcentajes de entrenamiento de 20 % y 40 % (a) y 60 % y 80 % (b).

número entero positivo), las tareas se están realizando en lotes de 48 tareas simultáneas, existiendo en este caso un último lote donde no se están utilizando todos los núcleos. De este modo, se genera así un número de núcleos ociosos que afectan a la eficiencia a nivel interno de cada nodo.

Para calcular el impacto real de dichos núcleos ociosos en la eficiencia total, se han tenido en cuenta la eficiencia a nivel de numero de nodos y a nivel interno de cada nodo de forma separada, siendo la eficiencia esperada el resultado de multiplicar estas eficiencias como se muestra en la ecuación (13).

$$\frac{T_1^{Nodos}}{T_n^{Nodos}} \cdot \frac{T_1^{nucleos}}{T_n^{nucleos}}, \tag{13}$$

Teniendo este importante aspecto en cuenta, podemos apreciar que (al igual que ocurre con el experimento 1) los valores obtenidos con porcentajes de entrenamiento del 20 % y 40 % [ver Figura 26(a)] muestran una caída gradual de la eficiencia en configuraciones con un número elevado de nodos, debido a que el tamaño del problema no es suficientemente grande como para explotar de forma completa este tipo de configuraciones masivas.

Para estudiar este aspecto en más detalle se ha realizado el experimento mostrado en la Figura 26(b), el cuál (al utilizar tanto un 60 % como un 80 % del total de los datos) explota completamente una configuración donde el número de nodos comienza a ser significativo. También es importante destacar, como en las configuraciones con mayor número de nodos, que los experimentos llevados a cabo con un 80 % del total de entrenamiento son más eficientes que el resto. En este caso, no sólo estamos aprovechando un numero de nodos mayor, si no que además aumenta significativamente el tamaño del problema, con lo que el tiempo de cómputo adquiere mayor peso que el de comunicación, incrementándose así la eficiencia.

Por último, al igual que ocurre en el experimento 1, destacamos que el error de reconstrucción en este experimento se mantiene similar sin importar el número de nodos. Por tanto, concluimos que la mejora en el rendimiento en cuanto al tiempo computacional no tiene ningún impacto negativo en el funcionamiento de la red. Por otro lado, al estar la imagen *HDS* formada por varias imágenes con un grado de similitud elevado, la variación en el porcentaje de entrenamiento tampoco genera diferencias significativas en cuanto al error de reconstrucción, lo cuál es también un aspecto altamente positivo. En general, los resultados mostrados en los experimentos 2 y 3 (ver Tabla 8) indican un comportamiento muy adecuado de la implementación desarrollada en la evaluación realizada con dos imágenes hiperespectrales reales de gran dimensionalidad y tamaño.

7. CONCLUSIONES Y TRABAJO FUTURO

En este trabajo de fin de grado se ha realizado una nueva implementación distribuida de una red neuronal artificial de tipo *autoencoder* para la compresión de imágenes hiperespectrales de la superficie terrestre. Nuestro método, al contrario que los métodos tradicionales de compresión disponibles en la literatura, proporciona una solución no lineal a la hora de reducir la dimensionalidad de los datos. La implementación distribuida desarrollada permite que el método sea más eficiente

	Porcentaje de entrenamiento (MB)	Nodos 1	2	4	8	12	16
Error (MSE)	20 (1838 MB)	6.09e-5	6.12e-05	6.01e-05	5.92e-05	6.47e-05	5.60e-05
	40 (3676 MB)	6.56e-5	5.80e-05	6.30e-05	6.44e-05	6.67e-05	6.13e-05
	60 (5515 MB)	N/A	5.88e-05	6.49e-05	6.27e-05	6.44e-05	5.70e-05
	80 (7353 MB)	N/A	6.59e-05	6.29e-05	6.25e-05	6.30e-05	6.20e-05
Tiempo (s)	20 (1838 MB)	14919.73	7632.64	4433.11	2952.27	1606.94	1171.87
	40 (3676 MB)	30526.79	15709.24	9087.66	5721.97	3311.36	2182.60
	60 (5515 MB)	N/A	21505.27	12458.54	8456.84	4536.73	3122.92
	80 (7353 MB)	N/A	32645.44	18900.49	11633.75	6084.69	4103.02
Eficiencia	20 (1838 MB)	1	1.9547	3.3655	5.0536	9.2845	12.7315
	40 (3676 MB)	1	1.9432	3.3591	5.2796	9.2187	13.9864
	60 (5515 MB)	N/A	1	1.7247	2.5191	4.7402	6.8862
	80 (7353 MB)	N/A	1	1.7268	2.8304	5.3651	7.9564

Tabla 8: Resultados de la experimentación con la imagen *HDS* (Experimentos 2 y 3). Se proporciona el error obtenido sobre el conjunto de validación, el tiempo de cómputo (en segundos), y la eficiencia con respecto a la versión serie.

computacionalmente cuando trabaja con imágenes de gran dimensionalidad y tamaño, como las empleadas en la experimentación llevada a cabo.

Entre las ventajas de nuestro método, no sólo se observa una mejora significativa en cuanto a la solución del problema de compresión, sino también en cuanto al tiempo de cómputo del mismo, como se puede ver en los niveles de escalabilidad alcanzados por la implementación desarrollada en *Apache Spark*.

Otra ventaja inherente a la implementación desarrollada es la capacidad de analizar conjuntos de datos que, de otra forma (debido a su tamaño, almacenamiento distribuido, etc.), serían imposibles de analizar. En este sentido, conviene destacar que la mayor parte de los repositorios de imágenes hiperespectrales disponibles se empiezan a caracterizar por su carácter distribuido (es decir, se encuentran almacenados en varios *datacenters*) y cada vez es más necesario acceder a grandes cantidades de datos, ubicadas en diferentes centros de investigación, para realizar estudios temporales que involucran grandes cantidades de datos hiperespectrales, obtenidos en diferentes localizaciones e intervalos temporales.

Para solventar estos problemas, que general grandes requerimientos en cuanto a tiempo de cómputo y (sobre todo) memoria, están surgiendo nuevos paradigmas englobados dentro del marco de la computación de altas prestaciones. En este sentido, las soluciones basadas en *GPUs* necesitan arquitecturas más potentes, tales como

clústers de *GPUs* capaces de realizar también almacenamiento masivo de datos. En nuestro caso, la implementación *cloud* desarrollada en *Apache Spark* ofrece prestaciones muy satisfactorias cuando la carga de trabajo es muy grande.

En el futuro, nos planteamos la posibilidad de incorporar nodos de procesamiento de tipo *manycore*, tales como *GPUs*, para incluir aún más recursos en cuanto a memoria y procesamiento al sistema desarrollado. En particular, los sistemas *GPU* (con gran cantidad de núcleos y memoria) son capaces de gestionar eficientemente problemas muy grandes, por lo que su incorporación al sistema desarrollado podría ayudar a solventar los problemas observados en la implementación actual en cuanto a la pérdida de rendimiento debida a la existencia de núcleos ociosos en nuestra implementación *Spark*.

Finalmente, destacamos que el presente trabajo ha sido enviado para su publicación a la revista de impacto *IEEE Transactions on Geoscience and Remote Sensing*, habiendo sido recibido como una contribución altamente innovadora en la primera ronda de revisión. Tras haber enviado la segunda versión del artículo implementando las modificaciones propuestas en la primera ronda de revisión, estamos a la espera de la siguiente comunicación referente a nuestro envío por parte del Comité Editorial de dicha revista.

Referencias

[1] J. M. Bioucas-Dias, A. Plaza, G. Camps-Valls, P. Scheunders, N. Nasrabadi, and J. Chanussot. Hyperspectral Remote Sensing Data Analysis and Future Challenges. *IEEE Geoscience and Remote Sensing Magazine*, 1(2):6–36, 2013.

[2] Robert O. Green, Michael L. Eastwood, Charles M. Sarture, Thomas G. Chrien, Mikael Aronsson, Bruce J. Chippendale, Jessica A. Faust, Betina E. Pavri, Christopher J. Chovit, Manuel Solis, Martin R. Olah, and Orlesa Williams. Imaging spectroscopy and the Airborne Visible/Infrared Imaging Spectrometer (AVIRIS). *Remote Sensing of Environment*, 65(3):227–248, 1998.

[3] Antonio Plaza, Jon Atli Benediktsson, Joseph W Boardman, Jason Brazile, Lorenzo Bruzzone, Gustavo Camps-Valls, Jocelyn Chanussot, Mathieu Fauvel, Paolo Gamba, Anthony Gualtieri, Mattia Marconcini, James C Tilton, and Giovanna Trianni. Recent advances in techniques for hyperspectral image processing. *Remote Sensing of Environment*, 113(1):S110–S122, 2009.

[4] Chen Xing, Li Ma, and Xiaoquan Yang. Stacked denoise autoencoder based feature extraction and classification for hyperspectral images. *Journal of Sensors*, 2016:1–10, 01 2016.

[5] Geoffrey E Hinton and Richard S Zemel. Autoencoders, Minimum Description Length and Helmholtz Free Energy. In *Proceedings of the 6th International Conference on Neural Information Processing Systems*, pages 3–10, Denver, Colorado, 1993. Morgan Kaufmann Publishers Inc.

[6] C.I. Grove G. RiveraJet A.M. Baldridge, S.J. Hook. The aster spectral library version 2.0. *Remote Sensing of Environment*, 2009.

[7] William Emery and Adriano Camps. *Chapter 2 - Basic Electromagnetic Concepts and Applications to Optical Sensors*, pages 43–85. Elsevier, 2017.

[8] A. Plaza, J. Plaza, A. Paz, and S. Sanchez. Parallel Hyperspectral Image and Signal Processing. *IEEE Signal Processing Magazine*, 28(3):119–126, 2011.

[9] Mustafa Teke, Hüsne Seda Deveci, Onur Haliloğlu, Sevgi Zübeyde Gürbüz, and Ufuk Sakarya. A Short Survey of Hyperspectral Remote Sensing Applications in Agriculture. In *Recent Advances in Space Technologies (RAST)*, 2013.

[10] Alexander F.H. Goetz, Gregg Vane, Jerry E. Solomon, and Barrett N. Rock. Imaging Spectrometry for Earth Remote Sensing. *Science*, 228(4704):1147–1153, 1985.

[11] Klaus I. Iteen and et al. APEX - the Hyperspectral ESA Airborne Prism Experiment. *Sensors*, 2008.

[12] Clifford D. Anger Stephen K. Babey. Compact Airborne Spectrographic Imager (CASI): a Progress Review. In *Proc.SPIE*, volume 1937, pages 1937 – 1937 – 12, 1993.

[13] Stephen G. Ungar, Jay S. Pearlman, Jeffrey A. Mendenhall, and Dennis Reuter. Overview of the Earth Observing One (EO-1) mission. *IEEE Transactions on Geoscience and Remote Sensing*, 2003.

[14] M. J. Barnsley, J. J. Settle, M. A. Cutter, D. R. Lobb, and F. Teston. The PROBA/CHRIS Mission: a Low-Cost Smallsat for Hyperspectral Multiangle Observations of the Earth Surface and Atmosphere. *IEEE Transactions on Geoscience and Remote Sensing*, 42(7):1512–1520, July 2004.

[15] C. Galeazzi, A. Sacchetti, A. Cisbani, and G. Babini. The PRISMA Program. In *IGARSS 2008 - 2008 IEEE International Geoscience and Remote Sensing Symposium*, volume 4, pages IV – 105–IV – 108, July 2008.

[16] Luis Guanter and et al. The EnMAP spaceborne imaging spectroscopy mission for earth observation, 2015.

[17] Michael A. Wulder, Jeffrey G. Masek, Warren B. Cohen, Thomas R. Loveland, and Curtis E. Woodcock. Opening the archive: How free data has enabled the science and monitoring promise of Landsat. *Remote Sensing of Environment*, 2012.

[18] Josef Aschbacher. *ESA's Earth Observation Strategy and Copernicus*, pages 81–86. Springer Singapore, Singapore, 2017.

[19] Yan Ma, Haiping Wu, Lizhe Wang, Bormin Huang, Rajiv Ranjan, Albert Zomaya, and Wei Jie. Remote sensing big data computing: Challenges and opportunities. *Future Generation Computer Systems*, 51(Supplement C):47 – 60, 2015.

[20] Mingmin Chi, Antonio Plaza, Jon Atli Benediktsson, Zhongyi Sun, Jinsheng Shen, and Yangyong Zhu. Big Data for Remote Sensing: Challenges and Opportunities. *Proceedings of the IEEE*, 104(11):2207–2219, 2016.

[21] Antonio Plaza, Javier Plaza, and David Valencia. Impact of platform heterogeneity on the design of parallel algorithms for morphological processing of high-dimensional image data. *Journal of Supercomputing*, 40(1):81–107, 2007.

[22] Antonio Plaza, David Valencia, Javier Plaza, and Pablo Martinez. Commodity cluster-based parallel processing of hyperspectral imagery. *Journal of Parallel and Distributed Computing*, 2006.

[23] D. Gorgan, V. Bacu, T. Stefanut, D. Rodila, and D. Mihon. Grid Based Satellite Image Processing Platform for Earth Observation Application Development. In *2009 IEEE International Workshop on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications*, pages 247–252, 2009.

[24] I. Foster, Y. Zhao, I. Raicu, and S. Lu. Cloud Computing and Grid Computing 360-Degree Compared. In *2008 Grid Computing Environments Workshop*, pages 1–10, Nov 2008.

[25] Zeqiang Chen, Nengcheng Chen, Chao Yang, and Liping Di. Cloud Computing Enabled Web Processing Service for Earth Observation Data Processing. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 2012.

[26] J Dean and S Ghemawat. Mapreduce: Simplified data processing on large clusters. *Journal of Communication Systems*, 2004.

[27] D Borthakur. The Hadoop Distributed File System : Architecture and Design. *Access*, 2007.

[28] Apache Spark. Apache Spark™ - Unified Analytics Engine for Big Data, 2018.

[29] Juan Mario Haut, Mercedes Paoletti, Javier Plaza, and Antonio Plaza. Cloud implementation of the k-means algorithm for hyperspectral image analysis. *J. Supercomput.*, 73(1):514–529, January 2017.

[30] Victor Andres Ayma Quirita, Gilson Alexandre Ostwald Pedro da Costa, Patrick Nigri Happ, Raul Queiroz Feitosa, Rodrigo da Silva Ferreira, Dário Augusto Borges Oliveira, and Antonio Plaza. A new cloud computing architecture for the classification of remote sensing data. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 10(2):409–416, 2017.

[31] Yi Zhang, Zebin Wu, Jin Sun, Yan Zhang, Yaoqin Zhu, Jun Liu, Qitao Zang, and Antonio Plaza. A distributed parallel algorithm based on low-rank and sparse representation for anomaly detection in hyperspectral images. *Sensors*, 18(11), 2018.

[32] Javier Setoain, Manuel Prieto, Christian Tenllado, and Francisco Tirado. GPU for Parallel On-Board Hyperspectral Image Processing. *International Journal of High Performance Computing Applications*, 2008.

[33] A. J. Plaza and C. I. Chang. *High Performance Computing in Remote Sensing Book Review Book Review*. Chapman & Hall/CRC Press, Computer & Information Science Series, Boca Raton, Florida, 2008.

[34] Carlos González, Sergio Sánchez, Abel Paz, Javier Resano, Daniel Mozos, and Antonio Plaza. Use of FPGA or GPU-based architectures for remotely sensed hyperspectral image processing. *Integration, the VLSI Journal*, 46(2):89 – 103, 2013.

[35] Antonio Plaza, Pablo Martínez, Javier Plaza, and Rosa Pérez. Dimensionality reduction and classification of hyperspectral image data using sequences of extended morphological transformations. In *IEEE Transactions on Geoscience and Remote Sensing*, 2005.

[36] Devis Tuia, Sebastian Lopez, Michael Schaepman, and Jocelyn Chanussot. Foreword to the special issue on hyperspectral image and signal processing. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 2015.

[37] J.M. Haut, M. Paolleti, J. Plaza, and A. Plaza. Cloud implementation of the k-means algorithm for hyperspectral image analysis. In *International Conference on Computational and Mathematical Methods in Science and Engineering CMMSE*, volume 2, pages 630–641. J. Vigo-Aguiar, P. Schwerdtfeger (New Zealand),W. Sprößig (Germany), N.Stollenwerk (Portugal), Pino Caballero (Spain), J. Cioslowski(Poland), J. Medina (Spain),I. P. Hamilton (Canada), J.A.Alvarez-Bermejo (Spain), 2016.

[38] 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*, 9(6):2270–2278, June 2016.

[39] Alberto Fernández, Sara del Río, Victoria López, Abdullah Bawakid, María J. del Jesus, José M. Benítez, and Francisco Herrera. Big Data with Cloud Computing: An insight on the computing environment, MapReduce, and programming frameworks. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 4, 2014.

[40] Y. Li, Z. Wu, J. Wei, A. Plaza, J. Li, and Z. Wei. Fast principal component analysis for hyperspectral imaging based on cloud computing. In *2015 IEEE International Geoscience and Remote Sensing Symposium (IGARSS)*, pages 513–516, July 2015.

[41] Pragyansmita Paul. Seti @ home project and its website. *XRDS*, 8(3):3–5, April 2002.

[42] Georg Hager and Gerhard Wellein. *Introduction to High Performance Computing for Scientists and Engineers*. CRC Press., 2010.

[43] Katarina Stanoevska-Slabeva, Thomas Wozniak, and Santi Ristol. *Grid and Cloud Computing: A Business Perspective on Technology and Applications*. Springer, 2010.

[44] Amazon Web Services. Overview of Amazon Web Services. *Amazon Web Services*, 2017.

[45] Microsoft. Microsoft Azure Cloud Computing Platform; Services, 2017.

[46] Charles Severance. *Using Google App Engine: Start Building and Running Web Apps on Google's Infrastructure*. O'Reilly, 2009.

[47] J Haut, M Paoletti, Abel Paz-Gallardo, Javier Plaza, and Antonio Plaza. Cloud implementation of logistic regression for hyperspectral image classification. In *Proceedings of the 17th International Conference on Computational and Mathematical Methods in Science and Engineering, CMMSE*, pages 1063–2321, 2017.

[48] X. Jia, B. C. Kuo, and M. M. Crawford. Feature mining for hyperspectral image classification. *Proceedings of the IEEE*, 101(3):676–697, March 2013.

[49] R.E. Bellman. *Adaptive Control Processes: A Guided Tour*. Princeton Legacy Library. Princeton University Press, 2015.

[50] G. Hughes. On the mean accuracy of statistical pattern recognizers. *IEEE Transactions on Information Theory*, 14(1):55–63, January 1968.

[51] Y. Chen, Z. Lin, X. Zhao, G. Wang, and Y. Gu. Deep Learning-Based Classification of Hyperspectral Data. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 7(6):2094–2107, 2014.

[52] Vinod Nair and Geoffrey E Hinton. Rectified Linear Units Improve Restricted Boltzmann Machines. In Johannes Fürnkranz and Thorsten Joachims, editor, *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 807–814. Omnipress, 2010.

[53] Marianna S Apostolopoulou, Dimitris G Sotiropoulos, Ioannis E Livieris, and Panagiotis Pintelas. A memoryless bfgs neural network training algorithm. In *Industrial Informatics, 2009. INDIN 2009. 7th IEEE International Conference on*, pages 216–221. IEEE, 2009.

[54] Quoc V Le, Jiquan Ngiam, Adam Coates, Abhik Lahiri, Bobby Prochnow, and Andrew Y Ng. On optimization methods for deep learning. In *Proceedings of the 28th International Conference on International Conference on Machine Learning*, pages 265–272. Omnipress, 2011.

[55] Guangming Tan, Linchuan Li, Sean Triechle, Everett Phillips, Yungang Bao, and Ninghui Sun. Fast implementation of dgemm on fermi gpu. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 35. ACM, 2011.

[56] Martin Odersky, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, Matthias Zenger, and et al. An overview of the scala programming language. Technical report, ., 2004.

[57] J. Pearlman, S. Carman, C. Segal, P. Jarecke, P. Clancy, and W. Browne. Overview of the hyperion imaging spectrometer for the nasa eo-1 mission. In *IGARSS 2001. Scanning the Present and Resolving the Future. Proceedings. IEEE 2001 International Geoscience and Remote Sensing Symposium (Cat. No.01CH37217)*, volume 7, pages 3036–3038 vol.7, July 2001.