UNIVERSIDAD
DE LA RIOJA

TRABAJO FIN DE ESTUDIOS

Título

Paralelización de algoritmos de clasificación para imagen biomédica

Autor/es

DANIEL ARIAS RUIZ-ESQUIDE

Director/es

JÓNATAN HERAS VICENTE

Facultad

Facultad de Ciencia y Tecnología

Titulación

Grado en Ingeniería Informática

Departamento

MATEMÁTICAS Y COMPUTACIÓN

Curso académico

2018-19

Paralelización de algoritmos de clasificación para imagen biomédica, de
DANIEL ARIAS RUIZ-ESQUIDE
(publicada por la Universidad de La Rioja) se difunde bajo una Licencia Creative
Commons Reconocimiento-NoComercial-SinObraDerivada 3.0 Unported.
Permisos que vayan más allá de lo cubierto por esta licencia pueden solicitarse a los
titulares del copyright.

© El autor, 2019
© Universidad de La Rioja, 2019
publicaciones.unirioja.es
E-mail: publicaciones@unirioja.es

UNIVERSIDAD
DE LA RIOJA

Facultad de Ciencia y Tecnología

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Paralelización de algoritmos de clasificación
para imagen biomédica

Parallelization of classification algorithms
for biomedical imaging

Realizado por:
Daniel Arias Ruiz-Esquide

Tutelado por:
Jónathan Heras Vicente

Logroño, junio, 2019

Resumen:

El *deep learning* está a la orden del día en problemas de clasificación de imágenes en el contexto biomédico. Sin embargo diversas complicaciones como no tener suficientes imágenes, deber realizar comparaciones exhaustivas de los modelos a utilizar y los resultados conseguidos con ellos, o la diversidad de librerías, herramientas y *hardware* especializado que existen para esta tarea hacen que sea todo un reto. El *framework* FrImCla, *open-source* y gratuito, emplea varios algoritmos de extracción de características para generar un modelo de clasificación de imágenes a partir de un *dataset*, no necesariamente grande. El modelo final es el resultado de la mejor combinación probada, comparando los resultados con un análisis estadístico. FrImCla permite a usuarios casi sin experiencia en *deep learning* crear modelos de clasificación, a partir de *datasets* pequeños y sin *hardware* adicional.

En este trabajo se ha diseñado una manera de paralelizar algunas componentes de FrImCla para conseguir los mismos resultados en un mejor tiempo. Para ello, se ha realizado un análisis de FrImCla para detectar sus "cuellos de botella", es decir, aquellas partes del código que lo ralentizan y que pueden ser paralelizadas. Tras este análisis se realizó un estudio de alternativas que pueden usarse para la paralelización. Finalmente se ha concluido con la implementación de todas estas técnicas para compararlas y decidir cuál es la mejor solución a este problema. Probando *threading*, *multiprocessing* (ambos módulos de Python), Hadoop y Spark (librerías externas), se ha observado que, en las condiciones de FrImCla, la mejor solución es utilizar la librería *multiprocessing* para esta tarea.

Abstract:

Deep learning is the state of the art approach in problems of image classification in biomedicine. However, various complications, such as not having enough images, making exhaustive comparisons of the models to be used and the results achieved with them, or the diversity of libraries, tools and specialized hardware that exist for this task make it a challenge. The framework FrImCla, open-source and free, uses several feature extraction algorithms to generate a classification model of images from a dataset, not necessarily large. The final model is the result of the best combination tested, comparing the results with a statistical analysis. FrImCla allows users, almost without experience, to create classification models, from small datasets and without additional hardware.

In this work, it has been designed a way to parallelize the functionality of FrImCla to achieve the same results in less time. Therefore, an analysis of FrImCla has been carried out to detect its "bottlenecks", that being, the parts of the code that slow it down and that can be parallelized. After such an analysis, a study of the alternatives that can be used for the parallelization was conducted. Finally, we concluded with the implementation of all these techniques to compare them and decide which is the best solution to the problem at hand. Testing threading, multiprocessing (both Python modules), Hadoop and Spark (external libraries), it has been observed that, under FrImCla's conditions, the best solution is to use the multiprocessing library for this task.

1. Introducción **3**
1.1. Visión por computador 3
1.2. Proyecto CLODE 3
1.3. FrImCla 4
2. Planificación **5**
2.1. Alcance 5
2.2. Calendario de trabajo 5
2.3. Diagrama Gantt 6
2.4. EDT 7
2.5. Plan de riesgos 9
3. Análisis **10**
3.1. Creación de modelos de clasificación de imagen 10
3.2. Diagrama de FrImCla 10
3.3. Cuellos de botella 11
4. Diseño **15**
4.1. Estudio de alternativas 15
4.2. Pruebas 18
4.3. Técnicas de evaluación 20
4.4. Ampliaciones 21
5. Pruebas **22**
5.1. Threading 22
5.2. Multiprocessing 24
5.3. Hadoop 27
5.4. Spark 32
6. Comparativa **35**
7. Seguimiento y control **37**
8. Lecciones aprendidas y conclusiones **41**
9. Referencias **43**

1. Introducción

Con esta breve sección se pretende contextualizar y explicar la motivación que hay detrás de este trabajo. Primero se hablará brevemente de la visión por computador, para luego exponer el problema que esta disciplina presenta de manera global dentro del Proyecto CLODE y más concretamente cómo se da este problema dentro del *framework* FrImCla.

1.1. Visión por computador

La visión por computador, también llamada visión artificial [1, 2], es una rama de inteligencia artificial que trata de procesar imágenes, generalmente del mundo real, para extraer información en forma de números u otros símbolos que pueda ser interpretada por un ordenador y así puedan "aprender" a percibir objetos y actuar en consecuencia.

Obviamente es un reto complejo e involucra a varias ramas como informática, física, matemáticas o neurobiología, entre otras. Pero también cuenta con un amplio abanico de aplicaciones como pueden ser la detección de eventos [3], las interacciones con humanos [4], la navegación [5], la modelización [6] o la inspección automática [7].

Algunos de los problemas que presenta este campo son la falta de imágenes o la calidad de estas. Se puede decir que existen "ruidos", interferencias en la imagen a modo de píxeles aleatorios en la escala de grises; o interferencias por contexto como deformaciones, puntos de vista, escalas y más. Todos ellos hacen que la computación necesaria sea compleja y costosa en tiempo. Esto provoca la aparición de *hardware* [8] especializado para analizar imágenes, por ejemplo en 2016 se estaban desarrollando *vision processing units* (unidades de procesamiento de visión) o por sus siglas en inglés VPU [9]. En el mismo año, Google, presenta las TPU [10] (*tensor processing unit*, unidad de procesamiento tensorial) de primera generación que son específicas para el aprendizaje automático. El problema que presentan estas tecnologías es su precio, no todo el mundo puede permitírselo. Por ejemplo, las TPU de Google Cloud se pagan, en el momento de escribir esto, por horas a un mínimo de 1.35 dólares estadounidenses [11] y las mejores GPUs (*graphics processing unit*, unidad de procesamiento gráfico), también empleadas en inteligencia artificial, tienen un precio mínimo actual de 1199 dólares estadounidenses, siendo este el de la RTX 2080 Ti de Nvidia [12].

1.2. Proyecto CLODE

Este trabajo se encuentra dentro de los objetivos del proyecto CLODE [13], integrado en el grupo de investigación denominado Grupo de Informática de la Universidad de La Rioja.

El proyecto CLODE se centra en resolver tres problemas útiles de la visión por computador: la clasificación (determinar a qué categoría pertenece una imagen), la localización (determinar dónde está un objeto en una imagen) y la detección (localización de múltiples objetos dentro de la misma imagen) [14]. Para ello se pretende desarrollar un motor que aplique distintas técnicas punteras para solucionar estos problemas. Además dicho motor debe ser extensible a nuevos métodos, ser integrable en otros desarrollos *software* y permitir determinar la mejor, o mejores, técnicas a utilizar en cada situación de una manera sistemática y eficiente. En la actualidad ya se ha desarrollado un *framework* para clasificación llamado FrImCla dentro del proyecto CLODE.

1.3. FrImCla

El *framework* FrImCla [15] está desarrollado en Python [16], es *open-source* y gratuito. Este *software* busca simplificar las tareas de construcción de modelos de clasificación para personas con menos experiencia en este ámbito y sin componentes específicos de *hardware*. FrImCla se enfrenta a un problema de clasificación de imágenes en biomedicina, la falta de imágenes.

Todo esto requiere de librerías de terceros que son empleadas y revisadas por una la gran comunidad de desarrolladores. Algunas de las herramientas son: scikit-learn [17], para *machine learning* y un poco de paralelización; OpenCV [18], para extraer características; Keras [19], para extraer características con *deep learning*; y cPickle, [20] para serializar objetos. Para más detalle sobre el *framework* y ejemplos de su uso referenciamos la documentación en [15].

La creación de estos modelos de clasificación, con el proceso que sigue FrImCla, supone un estudio exhaustivo de numerosas alternativas y esto es costoso en cuanto a tiempo. Al no estar completamente paralelizado, FrImCla presenta cuellos de botella. La motivación de este trabajo es mejorar la paralelización de sus tareas. Para ello, se propone realizar un estudio de alternativas en cuanto a la paralelización de código en Python, implementar y evaluar cada una de estas y finalmente realizar una comparación de lo logrado con cada una de ellas.

2. Planificación

En esta sección se describe el trabajo a realizar para solucionar el problema planteado en este TFG. Se tratará el alcance, es decir, los requisitos que debe cumplir para considerar que el proyecto está finalizado y la metodología a seguir. También se presenta una distribución del trabajo en el tiempo: un calendario y un diagrama Gantt; y una descomposición del trabajo a realizar: EDT y su diccionario correspondiente. Finalmente se incluye un plan de riesgos para establecer una relación entre las posibles incidencias, positivas o negativas, que puedan darse durante la realización del proyecto y cómo afrontarlas.

2.1. Alcance

Esta subsección comienza con el objetivo del proyecto, los requisitos funcionales y no funcionales que derivan de este, restricciones implícitas en los términos del proyecto CLODE y la implementación de FrImCla, y para terminar se expone la metodología que se va a utilizar.

Objetivo del proyecto

- Mejora del rendimiento de FrImCla a través de técnicas de paralelización.

Requisitos funcionales

- Implementación de dichas técnicas de paralelización en los cuellos de botella.

Requisitos no funcionales

- Encontrar los cuellos de botella de FrImCla.
- Realizar un estudio de las diferentes alternativas de paralelización.
- Realizar una comparativa de las técnicas implementadas.

Restricciones

- Mantener FrImCla extensible a nuevos métodos e integrable a nuevos desarrollos (como parte del proyecto CLODE).
- Desarrollo en Python o con compatibilidad, pues FrImCla está implementado en Python.

Metodología a utilizar

Una vez completado el análisis de los cuellos de botella, el estudio de las posibles alternativas y un primer diseño de las pruebas a realizar, se llevarán a cabo pequeños desarrollos en cascada para cada alternativa de mejora. Cada una podrá empezar con un pequeño análisis y diseño, si fueran necesarios, y concluirá con una evaluación de los resultados obtenidos con la técnica de paralelización que se esté usando en ese momento.

2.2. Calendario de trabajo

En la Figura 1 se muestra un calendario con los hitos del trabajo. Al ser en su mayoría estimaciones por no haber realizado antes ningún proyecto similar, los hitos están marcados por semanas, salvo los puntos de control y el inicio del proyecto. El código de colores facilita el paso del calendario al diagrama Gantt, ver Figura 2, y así el seguimiento.

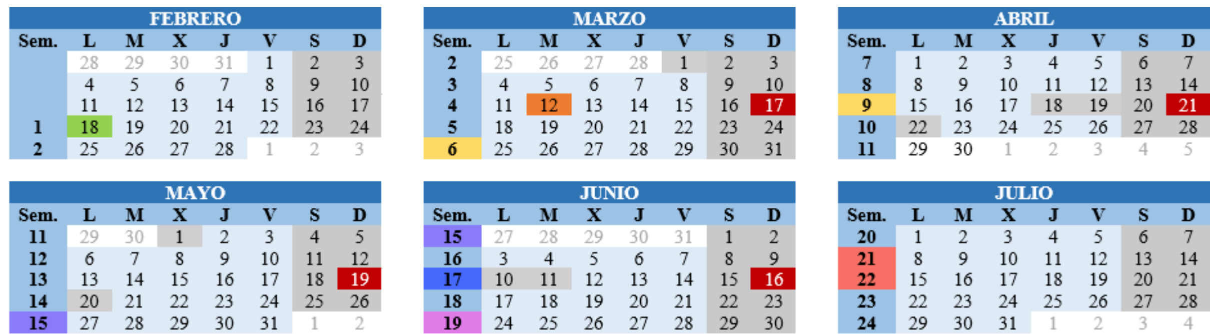

Figura 1. Calendario con hitos

Hitos marcados:

- 18 de febrero: inicio del TFG.
- 12 de marzo: final de la planificación.
- Semana 6 (del 25 al 31 de marzo): final del análisis (general).
- Semana 9 (del 15 al 21 de abril): final del diseño (general).
- Semana 15 (del 27 de mayo al 2 de junio): final de las pruebas.
- Semana 17 (del 10 al 16 de junio): final memoria.
- Semana 19 (del 24 al 26 de junio): periodo de depósito de la memoria.
- Semanas 21 y 22 (del 11 al 16 de julio): periodo de defensa del TFG.
- Tercer domingo de marzo, abril, mayo y junio: punto de control.

Aunque el final de la memoria esté estimado para la semana 17, el tiempo entre este final y su depósito puede ser dedicado a su revisión y mejora.

2.3. Diagrama Gantt

Como se ha adelantado, en la Figura 2 encontramos el diagrama de Gantt, en él se ha respetado el código de colores del calendario para relacionarlo fácilmente.

Al ser el propósito del proyecto el detectar y corregir cuellos de botella, las fases de Análisis y Diseño están muy relacionadas entre sí y se ha decidido utilizar el mismo color. Además estas dos fases son muy importantes ya que se trabajará para mejorar lo que se detecte en ellas y se utilizarán las alternativas, pruebas y mediciones que allí se especifiquen.

Puede darse el caso de que se necesite volver a analizar o diseñar si se detectan nuevos cuellos de botella o se descubren, o son necesarias, nuevas técnicas.

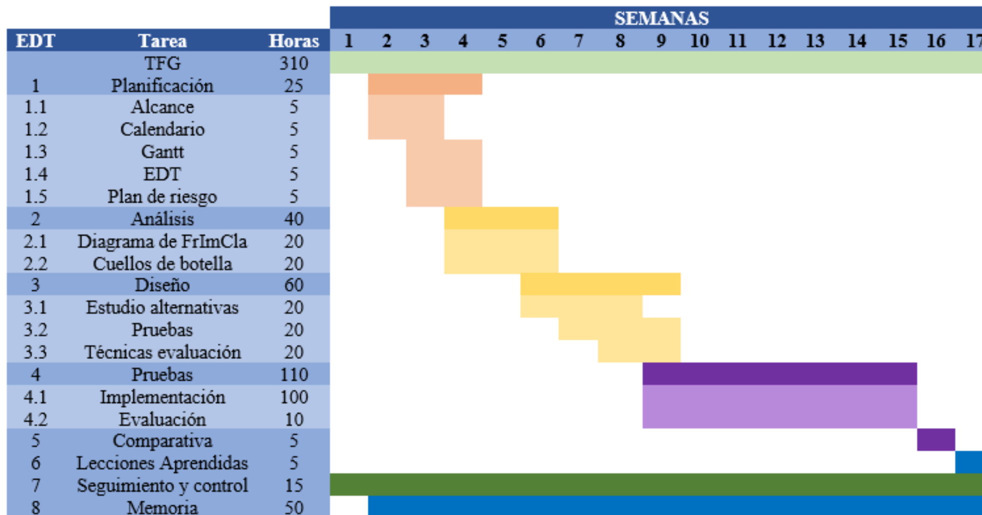

Figura 2. Diagrama Gantt del proyecto

En las estimaciones de horas no se ha tenido en cuenta el tiempo de formación empleado durante las prácticas previas a comenzar el TFG aunque está relacionado. Esta formación fue la primera toma de contacto con el tema de la visión por computador, y la lectura sobre cómo se realiza o cuáles son los principales retos a los que se enfrentan los programadores que quieren dedicarse a él.

Las prácticas consistían en el seguimiento del curso *PyImageSearch* [21], haciendo especial hincapié en los módulos *Computer Vision Basics*, *Image Classification and Machine Learning* y *Hadoop + BigData*. Como ya se ha dicho, muchos de los conceptos eran nuevos, quizás no tanto como tales pero sí en sus definiciones, problemas o maneras de trabajar con ellos utilizando un ordenador.

2.4. EDT

El EDT aparece en la Figura 3 y su diccionario en la Tabla 1.

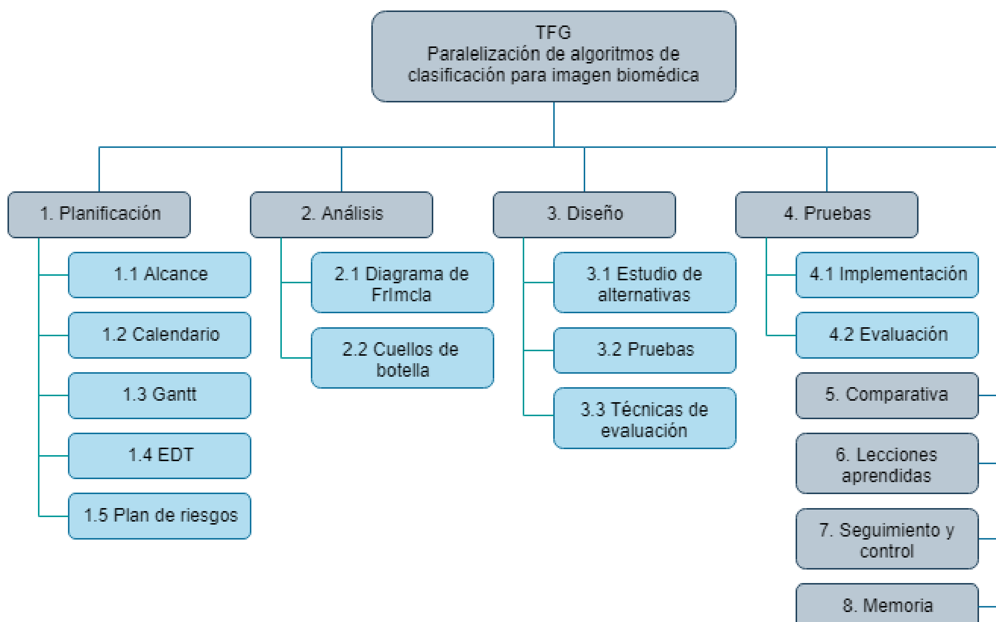

Figura 3. Esquema de descomposición del trabajo (EDT)

ID	Nombre	Descripción
1	**Planificación**	
1.1	**Alcance**	Concretar el objetivo, requisitos, restricciones y metodología del proyecto.
1.2	**Calendario**	Hitos y fechas más relevantes del proyecto en formato de calendario.
1.3	**Gantt**	Estimación en horas y semanas de las diferentes tareas del proyecto.
1.4	**EDT**	Descomposición del proyecto en tareas a realizar.
1.5	**Plan de riesgos**	Relación de incidencias que pueden suceder, su probabilidad, su impacto y cómo actuar si se dan.
2	**Análisis**	
2.1	**Diagrama de FrImCla**	Estructura del *framework*, desde una vista de alto nivel para explicar sus componentes.
2.2	**Cuellos de botella**	Explicación de los cuellos de botella encontrados en el programa.
3	**Diseño**	
3.1	**Estudio de alternativas**	Estudio de las distintas técnicas de paralelización que pueden usarse para mejorar el programa.
3.2	**Pruebas**	Pruebas a realizar, una por cada técnica. Decidir qué cuellos de botella se intentará corregir.
3.3	**Técnicas de evaluación**	Parámetros a considerar para luego poder realizar las tareas de evaluación y comparativa.
4	**Pruebas**	
4.1	**Implementación**	Cambios en el código para introducir la paralelización. Podrían ser varias técnicas o en varios lugares así que no tiene porque haber solo una implementación.
4.2	**Evaluación**	Observación de los resultados de cada implementación de acuerdo a las mediciones.
5	**Comparativa**	Organización de los resultados de las pruebas.
6	**Lecciones aprendidas**	Conocimientos adquiridos al realizar este proyecto.
7	**Seguimiento y control**	Revisión del cumplimiento de la planificación durante el TFG.
8	**Memoria**	Desarrollo de este documento. Simultáneo al trabajo.

Tabla 1. Diccionario del EDT

2.5. Plan de riesgos

A continuación, ver Tabla 2, se plantean las posibles incidencias que pueden darse durante el desarrollo del trabajo y cómo se actuará en cada caso.

Incidencia	Plan de actuación	Riesgo	Impacto
No poder implementar alguna alternativa	Pausar esta implementación, retomar al final si queda tiempo. De no ser posible la segunda vez, cancelar.	Medio	Alto
Alguna alternativa obtiene malos resultados	No debe tomarse como algo negativo, se dejará constancia de ello al evaluar la alternativa y en la comparativa de todas.	Bajo	Bajo
Se descubre una nueva alternativa	Si es posible añadirla, por motivos de tiempo, se intentará y añadirá al estudio de alternativas	Bajo	Medio
Excedente de tiempo al agotar las alternativas	Se dedicará a la revisión de esta memoria y/o búsqueda de nuevas alternativas.	Bajo	Medio
Retrasos en el proyecto	Como se plantea realizar una comparativa de técnicas de paralelización se priorizará acabar al menos dos.	Medio	Alto
Problemas técnicos	Consultas al tutor para su rápida resolución o pasar a las siguientes tareas.	Medio	Medio

Tabla 2. Plan de riesgos

3. Análisis

Esta sección comienza con una explicación de cómo se crean modelos de clasificación de imágenes, continua con un diagrama del *framework* FrImCla, explicando para qué sirven las partes del mismo; y concluye con un estudio de los cuellos de botella que presenta actualmente este *framework*, tomando una decisión sobre cuál o cuáles se puede actuar.

3.1. Creación de modelos de clasificación de imagen

El proceso clásico de creación de modelos para clasificar imágenes consiste en tener un *dataset* de imágenes del que se extraen unos datos que se llaman *features* (características), que pueden ser muy variados según la técnica de extracción que se emplee, algunos ejemplos son HOG, Haar o SIFT [22, 23, 24]. Luego estas *features* se utilizan para entrenar un algoritmo de clasificación (entre otros, KNN, redes neuronales o SVM [25, 26, 27]), obteniendo así lo que se denomina modelo [28].

Un problema de este proceso es el conocido como *no free lunch theorem* [29] que básicamente expone que si un algoritmo (que incluye las dos fases: extracción de características y entrenamiento del algoritmo de clasificación) funciona bien para un conjunto de problemas, paga este mejor rendimiento con uno peor para el resto de problemas. Esto aplicado a los algoritmos de clasificación, quiere decir que no existe uno que funcione mejor que el resto para todos los *datasets*. Aquí entra FrImCla pues automatiza la tarea de comparar los algoritmos que elige el usuario, según el criterio que este da, y le devuelve el mejor ya entrenado, además de información sobre la comparativa.

3.2. Diagrama de FrImCla

FrImCla recibe como entradas un *dataset* y un fichero de configuración en formato JSON. En dicho fichero de configuración aparecen: la ruta al *dataset*, los métodos de extracción de características a utilizar, los algoritmos de clasificación a evaluar y la medida utilizada para evaluar los modelos construidos. Finalmente produce el mejor modelo, de los probados, para el *dataset* en cuestión. Este modelo se entrena con el código de FrImCla de manera que está listo para usarse con nuevas imágenes, relacionadas al *dataset*, para predicciones. En los siguientes apartados se nombran las fases del proceso que realiza FrImCla y las técnicas empleadas en dichas fases. En la Figura 4 se puede ver un diagrama general del flujo de trabajo con estas fases.

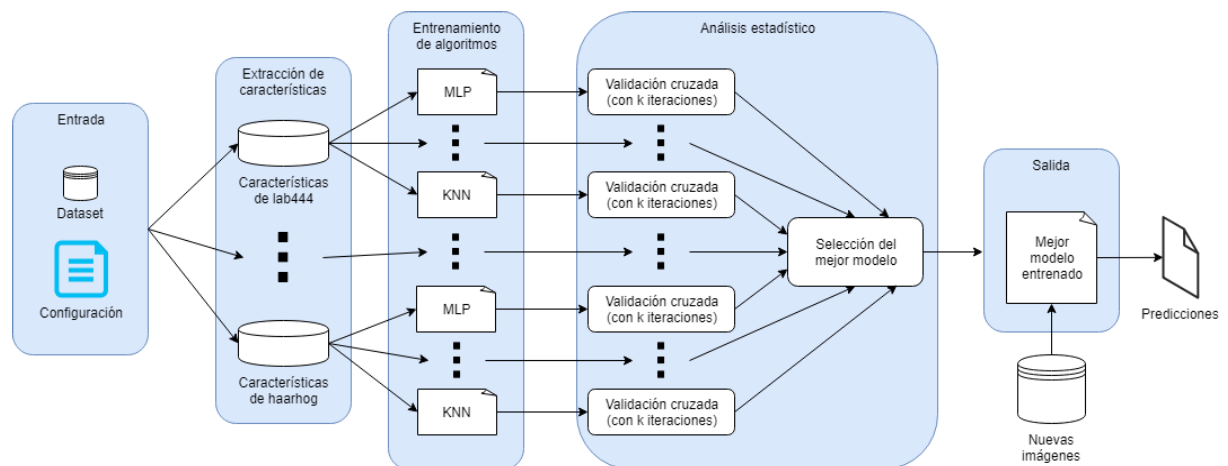

Figura 4. Diagrama de flujo de FrImCla, incluye el uso posterior de la salida generada

Extracción de características:

En un primer paso se consiguen las características del dataset, muy importantes pues se emplearán tanto para entrenar como para probar los modelos de clasificación. Para ello, se hace uso tanto de técnicas clásicas como de *transfer-learning* [30]. Las técnicas que están incluidas con FrImCla son las siguientes. Por el lado tradicional: histogramas, tanto LAB como HSV [31]; y características de Haralick [32] y de HOG [22]. Por el lado de *transfer-learning* se utiliza la salida de varias redes de *deep learning* entrenadas para el desafío ImageNet como por ejemplo: VGG19 [33], ResNet [34], GoogleNet [35] o Xception [36]; entre otras.

Entrenamiento de algoritmos:

El segundo paso consiste en entrenar los algoritmos de clasificación indicados en el fichero de configuración. Inicialmente FrImCla cuenta con los algoritmos: KNN [25], redes neuronales [26], SVM [27], *Gradient Boost* [37], regresión logística (*Logistic Regression*) [38] y *Random Forest* [39].

Análisis estadístico:

Este último paso tiene dos partes. Como primera parte cada método es entrenado con una validación cruzada de k iteraciones (*k-fold cross validation*) que se explica en el siguiente párrafo. Esta parte está a caballo entre el paso anterior y el actual ya que incluye el entrenamiento de los modelos pero por respetar el diagrama se pone en este.

Al realizar una *k-fold cross validation* el primer paso es dividir el *dataset* en k partes iguales. De estas se utilizan todas menos una para entrenar y la restante para evaluar. Este proceso se repite k veces, utilizando una parte diferente para la evaluación de cada iteración. Gracias a los resultados de la validación cruzada el *framework* conoce el desempeño de cada combinación extractor-clasificador. Obviamente esto se repite para cada técnica de extracción de características empleada en el paso correspondiente. Los resultados de las combinaciones pueden venir dados en: *accuracy*, *F1-score*, *recall*, *precision* y AUROC [40, 41].

En la segunda parte se hace un análisis estadístico basado en los resultados de las combinaciones anteriores [42, 43]. El resultado de este análisis es el mejor par extractor-clasificador para el dataset empleado.

Salidas:

A parte de la obvia, expuesta en el diagrama, que es la mejor combinación de extracción de características y modelo de clasificación una vez entrenada con todo el dataset, el *framework* produce otras salidas que no son relevantes para este trabajo. El modelo generado puede utilizarse para predecir las clases de imágenes, similares al dataset, una vez acabado el proceso

3.3. Cuellos de botella

En este subapartado se dará un vistazo de alto nivel a los ficheros de FrImCla utilizando las entradas y salidas de los métodos más relevantes de cada fichero. Para contar con un ejemplo del

estado actual se realizará un análisis de los tiempos empleados en cada una. Finalmente se estudiará cuáles pueden ser los cuellos de botella presentes en el código. Hay tres ficheros principales que merecen mención: `index_features.py`, `StatisticalComparison.py` y `train.py`.

index_features.py

Este fichero contiene el primer proceso que realiza FrImCla y se relaciona con el paso de extracción de características. Tiene dos métodos principales `generateFeatures` y `extractFeatures`. El primero realiza varias llamadas al segundo, una para cada `extractor` seleccionado por el usuario. A continuación se explica cada uno de ellos:

- `extractor`: es un objeto de la clase `Extractor`, que cuenta con el método `describe`. Este método recibe un *batch* (lote o conjunto) de imágenes y realiza el proceso de extracción de características de manera secuencial, pero independiente para cada una de ellas. Por ello, el método `describe` es un buen candidato para aplicar paralelización.
- `extractFeatures`: como entradas recibe el extractor a utilizar y una lista con rutas de ficheros. Separa el *dataset* en *batches* y aplica el método `describe` del extractor a cada uno. Su salida son las características extraídas con el extractor, almacenadas en un fichero nuevo (el nombre del fichero depende del extractor utilizado).
- `generateFeatures`: como entradas recibe los extractores a utilizar en esta ejecución y la ruta al *dataset*. Aplica el método `extractFeatures` una vez por cada `extractor`.

StatisticalComparison.py

El segundo fichero engloba los pasos de entrenamiento de algoritmos con distintos tipos de características y el posterior análisis estadístico de los resultados que consigue cada combinación. El método que se usa para todo esto se llama `statisticalComparison`.

Las entradas del método son las rutas de dónde depositar su salida y del *dataset* utilizado. También recibe la lista de extractores utilizados en el fichero anterior, la lista de algoritmos de clasificación, para entrenar y comparar los modelos generados; y la medida elegida por el usuario para la comparación. La salida es una lista de archivos con información sobre las combinaciones, de extractor y clasificador, que producen los mejores modelos para la medida seleccionada.

Un método que puede destacar durante la ejecución de `statisticalComparison` es:

- `compare_methods_h5py`, que está en el fichero `Comparing.py`, y que realiza las comparaciones de clasificadores para un extractor. El proceso está paralelizado pero se dan condiciones de carrera de las que se hablará más adelante.

train.py

El último fichero realiza el entrenamiento del mejor modelo a partir de la mejor combinación de extractor y clasificador. Los datos para esto los encuentra en el archivo `ConfModel.json`, un fichero generado automáticamente por el análisis estadístico. Allí se encuentran el extractor, los parámetros para este y el clasificador. Al ejecutar este fichero se produce un modelo y si el usuario lo desea una aplicación web para realizar predicciones con él.

En este fichero no se encuentra nada paralelizable pues simplemente entrena un algoritmo usando las características ya extraídas.

Después de ver estos tres ficheros se comprueba el tiempo que consume cada uno al ponerlos a prueba con un *dataset* de doscientas dos imágenes, la mitad de perros y la mitad de gatos. Este *dataset* ha sido creado reduciendo el *dataset* Kaggle Dogs and Cats [44].

Evaluación:

Para realizar una prueba inicial del rendimiento de FrImCla se ha utilizado el *dataset* reducido Dogs and Cats en una máquina con sistema operativo Linux Ubuntu 18.04, que cuenta con una *CPU* Intel Core i5-4210M 2.60 GHz y 8 GiB de *RAM*. La configuración elegida utiliza trece extractores de los disponibles: VGG16, VGG19, ResNet, Inception, GoogleNet, Overfeat, LAB444, LAB888, HSV444, HSV888, Haralick, HOG y *HaarHOG*; seis clasificadores: MLP, SVM, KNN, *LogisticRegression*, *GradientBoost* y *RandomForest*; y utilizando como medida la *accuracy* y cinco pasos en la validación cruzada. Con todo esto se consigue la respuesta que vemos representada en la Figura 5 y explicada después.

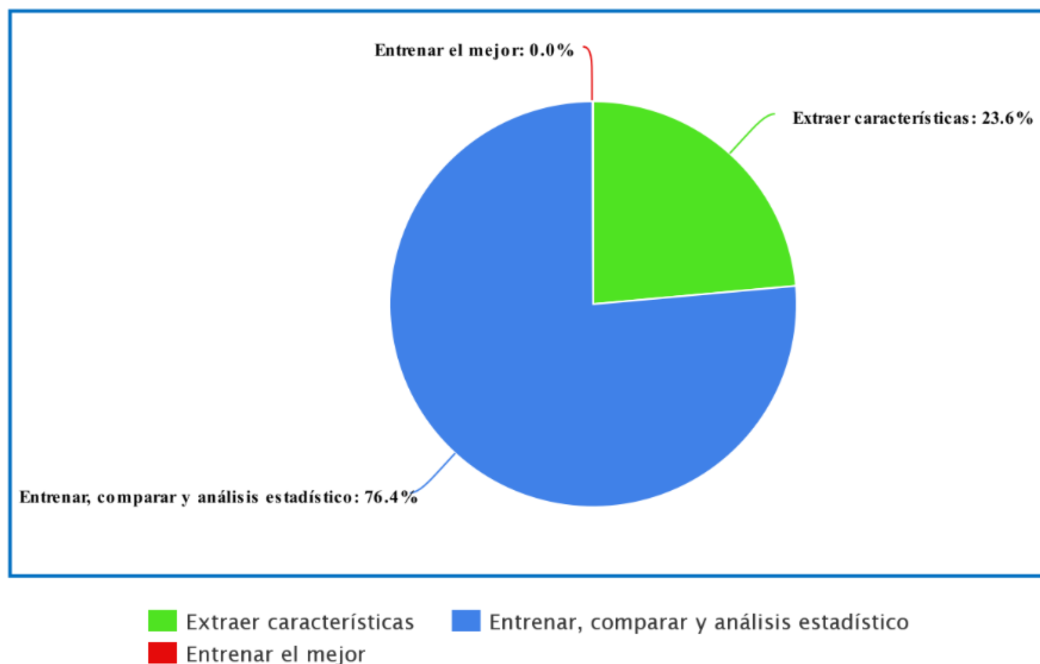

Figura 5. Porcentaje de tiempo consumido en cada paso

De un total de 2453.8631 segundos (40.8977 minutos) de ejecución, 578.3367 (9.6389 minutos) han sido dedicados a extraer características, 1874.3992 (31.2340 minutos) al segundo paso de entrenar todas las combinaciones, compararlas y realizar el análisis estadístico y finalmente 1.1272 para entrenar el mejor modelo, esta vez utilizando todo el *dataset*.

Puede verse que la mayor parte del tiempo se emplea en la comparación de modelos y el análisis estadístico, contando que en esta parte deben entrenarse varios modelos tiene sentido. También es mucho tiempo el empleado en la extracción de características, solo teníamos doscientas dos imágenes, trece métodos extractores y ha tardado un poco menos de diez minutos.

Cuellos detectados:

Tomando como base los tiempos anteriores, el código y experiencia de otras personas con el *framework* se pueden encontrar dos cuellos de botella principales.

El primero es el método `compare_methods_h5py`. Se expone como primero ya que cuenta con una paralelización implementada. La experiencia previa con este método concluye que se da un retraso por esperar al algoritmo más lento. Al estar paralelizado, los métodos más rápidos terminan y dan sus resultados pero no se puede continuar con la ejecución hasta tener todos y por eso debe esperar al último. El tiempo empleado en el paso del análisis estadístico es necesariamente el mayor ya que puede realizar hasta setenta y ocho combinaciones de extractores y clasificadores (setenta y ocho serían todas pero algunas son incompatibles) si esto lo multiplicamos por el número de pasos de la validación cruzada, en este ejemplo 5, es increíble que no tarde más tiempo en este paso. Esto es fruto de la paralelización ya implementada.

El segundo es el método `describe` del que se habló en la parte dedicada a `index_features.py` y que no cuenta con ningún tipo de paralelización. Como ya se señaló, este método realiza la extracción de características de manera secuencial sobre las imágenes pero es independiente para cada una de ellas, pues cada imagen puede ser tratada sin involucrar al resto. En la Figura 6 puede verse esta secuencialidad, en ella la última imagen es procesada en el tiempo t_{n-1} y al paralelizar pueden empezar varias simultáneamente en el t_0 o instante inicial. Un objeto de la clase `Indexer` se ocupa de escribir todas las características extraídas en un archivo en formato HDF5 (*Hierarchical Data Format*) [45].

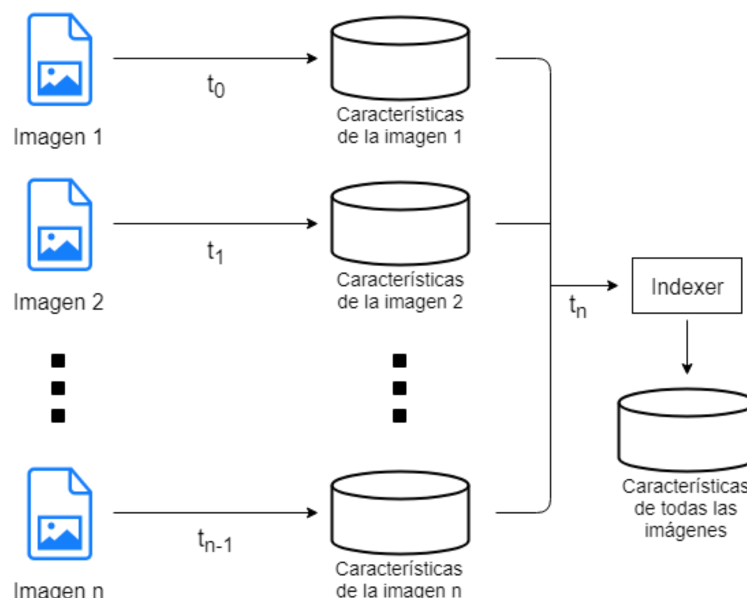

Figura 6. Trabajo secuencial sobre las imágenes al extraer características

Esta parece ser la causa del retardo que sufre la primera parte del *framework* y eso que solo se han extraído características de doscientas dos imágenes. Por todo esto, se escoge este segundo cuello de botella como objetivo a mejorar.

4. Diseño

Esta sección cuenta con tres subapartados que coinciden con los indicados en la planificación. En el primero se realiza un breve estudio de técnicas de paralelización alternativas que se intentarán implementar para producir una mejora. Luego uno dedicado a las pruebas a realizar, básicamente sobre el cuello de botella detectado, utilizando técnicas diferentes. Finalmente una aclaración sobre cómo medir la mejora de FrImCla tras realizar cada prueba y posibles ampliaciones.

4.1. Estudio de alternativas

La paralelización típica en Python son los *threads* o hilos y los *processes* o procesos, de dos módulos diferentes, *threading* [46] y *multiprocessing* [47]. A parte de estas dos alternativas se quiere explorar algunas extra.

***Threads* en Python:**

Los hilos son utilizados para efectuar varias llamadas a la misma función o tarea. Normalmente se utilizan cuando una tarea involucra una espera, por ejemplo descargar de datos de una URL. Tener varios hilos permite ejecutar otro código durante la espera, en el ejemplo, empezar la descarga de la siguiente URL.

Disponer de varios hilos no significa que se vayan a ejecutar los procesos a paralelizar en diferentes CPUs; aunque se cuente con ellas, los hilos se ejecutan en el mismo *heap* (montículo) de memoria. Tampoco que se vaya a poder mejorar la velocidad de un programa que ya usa en su totalidad una de estas unidades constantemente. No es una solución universal pero en ejemplos como el de las URL o en nuestro caso la extracción de características de cada imagen, puede probarse.

Para utilizar los hilos hay que importar el módulo *threading* y se puede crear una clase que herede de la clase `Thread` (que está en *threading*), por ejemplo:

```
class MiHilo(threading.Thread):
      def run(self):
            // Hace algo
```

Luego un objeto de esta clase se creará con `MiHilo()` y se iniciará con el método `start()` que prepara la ejecución del método `run` en un hilo distinto del principal, uno nuevo.

Otra manera de utilizar hilos es usar como base un método. Si se quiere hacer esto tiene que pasarse el método como *target* (objetivo) al constructor de la clase `Thread`, por ejemplo:

```
def HacerAlgo():
      // Hace algo

threading.Thread(target=HacerAlgo).start();
```

Este código hace que el código de `HacerAlgo()` se ejecute en un hilo separado del principal. Ambas alternativas presentan el mismo comportamiento.

***Processes* en Python:**

El módulo de *multiprocessing* consigue algo similar a *threading* pues permite lanzar varios procesos independientes los unos de los otros para conseguir el mejor uso posible de los núcleos de la CPU. Tiene muchísimas características potentes que están en la documentación oficial [47].

Las clases más básicas de esta librería son `Process` y `Pool`. La primera es similar a la invocación de `Thread()` con una función para el caso de hilos:

```
def HacerAlgo():
      // Hace algo
p = multiprocessing.Process(target=HacerAlgo);
p.start()
p.join()
```

Este código invoca a un proceso que ejecuta `HacerAlgo()` de manera independiente mientras el principal espera en la instrucción `join()`.

La segunda clase, `Pool`, posee cuatro métodos `apply`, `map` y sus versiones asíncronas (`apply_async` y `map_async`). Siendo la diferencia que las primeras bloquean el proceso principal hasta tener todos los resultados y las versiones asíncronas no, pero se necesita utilizar el método `get()` del resultado (un objeto de la clase `AsyncResult`) para conseguir los valores devueltos.

Los métodos del párrafo anterior se efectúan sobre un *pool* o piscina de procesos. En ella se tiene una cantidad de procesos igual a la usada al inicializar el *pool*. Por defecto se usa el número de CPUs disponibles. Al final el número utilizado es el máximo de procesos que se ejecutarán simultáneamente.

Se puede utilizar la clase `Pool` para realizar una versión simple de la estrategia MapReduce que utiliza la siguiente librería o *framework*, Hadoop.

Hadoop:

Hadoop [48] es un *framework* de código libre que permite repartir grandes cantidades de datos en un clúster de máquinas no tan especializadas como un servidor dedicado y superpoderoso, sin embargo, también se puede realizar lo mismo en un solo ordenador. Esto lo consigue con su sistema de archivos, *HDFS* (*Hadoop Distributed File System*), y que parte la información en bloques de 64 a 128 MB y los reparte por el clúster de ordenadores de bajo coste.

Hadoop no solo facilita este reparto sino que da mucho poder de procesamiento a estas mismas máquinas comunes con su estrategia MapReduce, que es lo que nos interesa ya que permite procesar grandes cantidades de datos en paralelo.

El término MapReduce en realidad engloba dos tareas. La primera es Map, que consiste en descomponer los datos de entrada en pares de clave-valor que serán pasados a la siguiente. Esta otra es Reduce que coge los pares y los agrega para formar un único *output*. La clave de la paralelización es

que un *reducer* coge las salidas de varios *mappers* y esto permite procesar en paralelo. En la Figura 7 se expone un ejemplo sencillo (contar letras) que aplica MapReduce. En la figura aparecen dos pasos extra: separar, que se da para paralelizar la entrada; y mezclar/ordenar, que podría considerarse parte de Reduce ya que mezcla las salidas de Map.

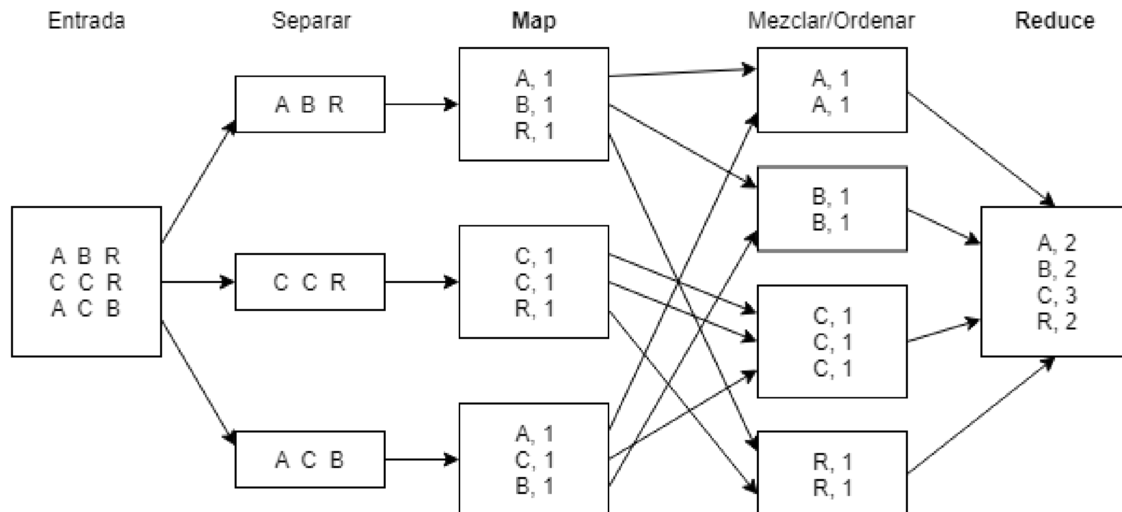

Figura 7. Diagrama ejemplo de MapReduce para contar apariciones de letras

El problema de Hadoop con Python es que Hadoop está implementado en Java así que se tiene que buscar la manera de utilizar las funciones desde el otro lenguaje. Para ello se crean las clases `mapper.py` y `reducer.py` en Python. Posteriormente utilizando Hadoop este usará la clase hadoop-streaming escrita en Java pero usando las nuestras escritas en Python como *mapper* y *reducer*.

Spark:

Spark [49, 50] es otro *framework open-source* de computación en clúster que cuenta con una API para Python llamada PySpark. Al igual que Hadoop, no está limitado a computación en clúster, Spark permite lo mismo en un solo ordenador. Spark aporta librerías para manejar estructuras de datos propias, distribuirlas por un clúster o paralelizar código. Esto último es lo importante para este trabajo.

Con Spark nativo se paraleliza sobre las estructuras de datos propias de Spark y con la versión para Pandas (librería *open-source* que da estructuras y herramientas para el análisis de datos en Python) [51] se permite esta paralelización sobre estructuras de Pandas. La paralelización y la distribución son independientes en Spark, es posible conseguir paralelización sin distribución. Esto significa que toda la tarea la está efectuando el ordenador principal, sin delegar a otros denominados trabajadores.

Spark es rápido, por ejemplo, su librería MLib [52] para *machine learning* es hasta nueve veces más rápida que la implementación basada en disco que hace Apache Mahout (*framework* de álgebra lineal distribuido diseñado para que matemáticos, estadísticos y científicos de datos implementen sus propios algoritmos rápidamente) [53]. Esto la convierte en una opción interesante para mejorar los tiempos de FrImCla.

4.2. Pruebas

Tras lo expuesto en las subsecciones 3.3, sobre los cuellos de botella, y 4.1, sobre las técnicas de paralelización alternativas que se va a intentar implementar, se puede diseñar cómo van a ser las pruebas realizadas sobre FrImCla. Esta subsección lo contará de manera breve y en las subsecciones de la Sección 5 (sección también llamada Pruebas) se irán reflejando las implementaciones y resultados obtenidos por cada una.

Las pruebas consistirán en las diferentes implementaciones de las alternativas para el cuello de botella seleccionado (el del método `describe` que pertenece al paso de extracción de características de FrImCla). Además se puede ver como este cuello de botella puede tratarse a tres niveles de profundidad.

Alternativas:

Los módulos *threading* y *multiprocessing* darán una implementación más básica y más cercana a trabajar solo en Python. Por el otro lado utilizar Hadoop o Spark quizás sea mejor por la fama que tienen, sobre todo en el caso del segundo, y por ello se pondrán a prueba.

En el caso de Hadoop y PySpark es obvio que se optará por la estrategia MapReduce. Esto también se intentará en la implementación de *multiprocessing* pues permite crear un MapReduce más simple. De no lograrse esta implementación se optará por algo más sencillo y cerca de lo que hace el módulo *threading* pues este y el de procesos son similares.

Niveles:

En la extracción de características de FrImCla se aprecian tres niveles en los que se puede implementar la paralelización, pues los procesos que se realizan en cada uno parecen independientes. Se podría paralelizar a nivel de: imágenes, *batches* o extractores de características.

Lo que se hace en cada nivel es básicamente lo que se exponía en la Figura 6 (p. 14) pero aplicado en ese nivel. Para aclarar esto, a continuación están los niveles en orden ascendente (inferior significa que es utilizado por los superiores):

- **Imágenes:** es la Figura 6. Separa un *batch* en sus imágenes y realiza la extracción en cada una de ellas, ver Figura 8.
- ***Batches*:** si cambiamos en la Figura 6 “imagen” por “*batch*” pasamos a este nivel. Separa el *dataset* en *batches* y para cada uno llama al nivel inferior, imágenes. Ver Figura 9.
- **Extractores de características:** para cada extractor que esté seleccionado en la configuración llama al nivel inferior, *batches*, para que se procese todo el *dataset* con ese extractor. Puede verse en la Figura 10.

Ya que procesar las imágenes es básicamente extraer información de ellas no hay problema en paralelizar a cualquier nivel mientras Python y las librerías utilizadas lo permitan. Cómo esto puede parecer poco intuitivo se incluyen las Figuras 8, 9 y 10, ya referenciadas.

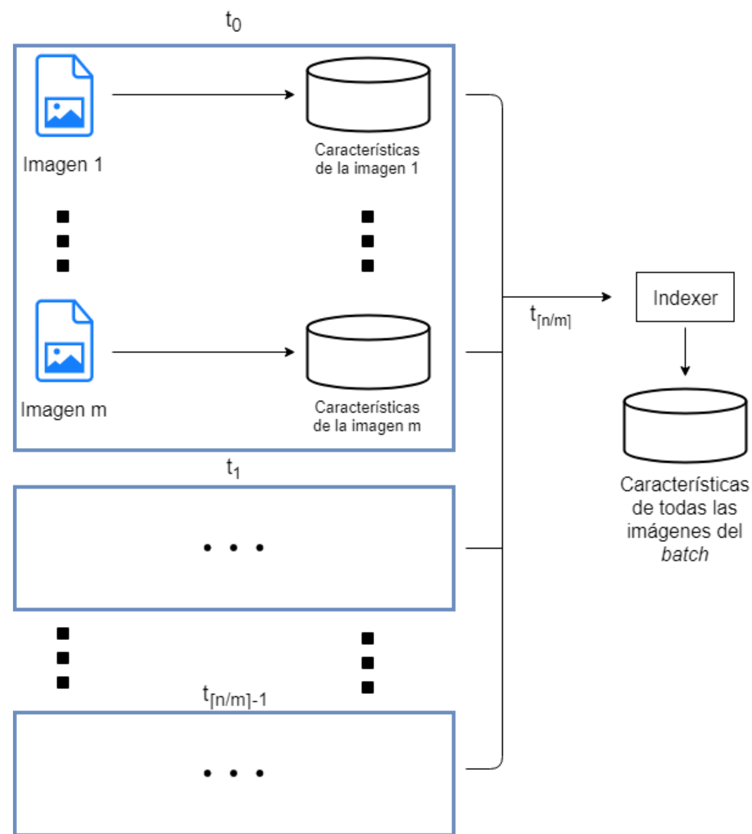

Figura 8. Nivel imágenes. Se extraen las características de m en m imágenes (en total n imágenes)

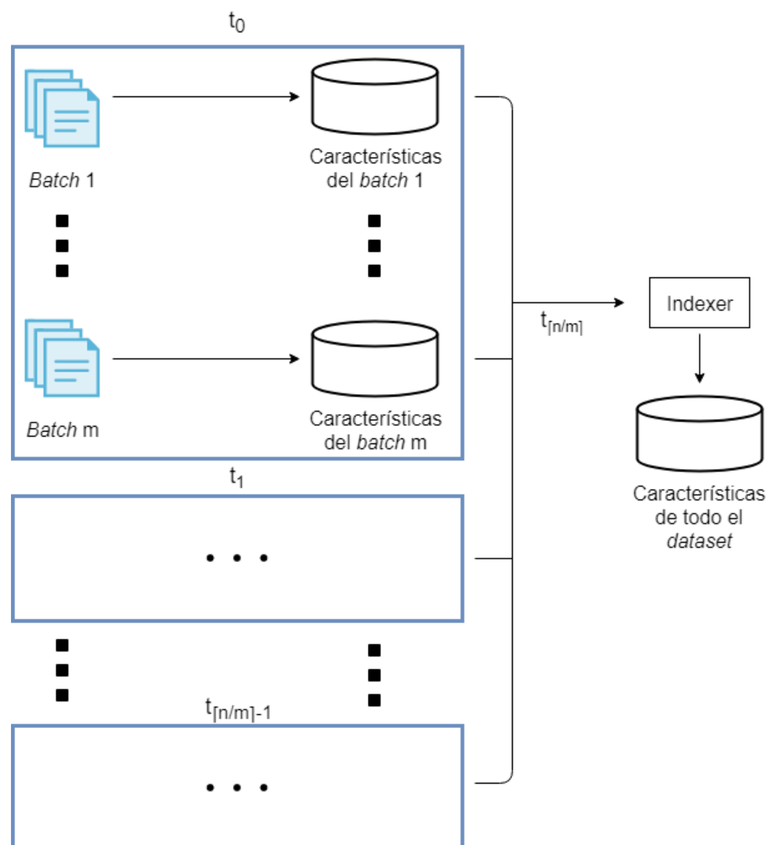

Figura 9. Nivel *batches*. Se extraen de m en m *batches*, usando solo un extractor (en total n *batches*)

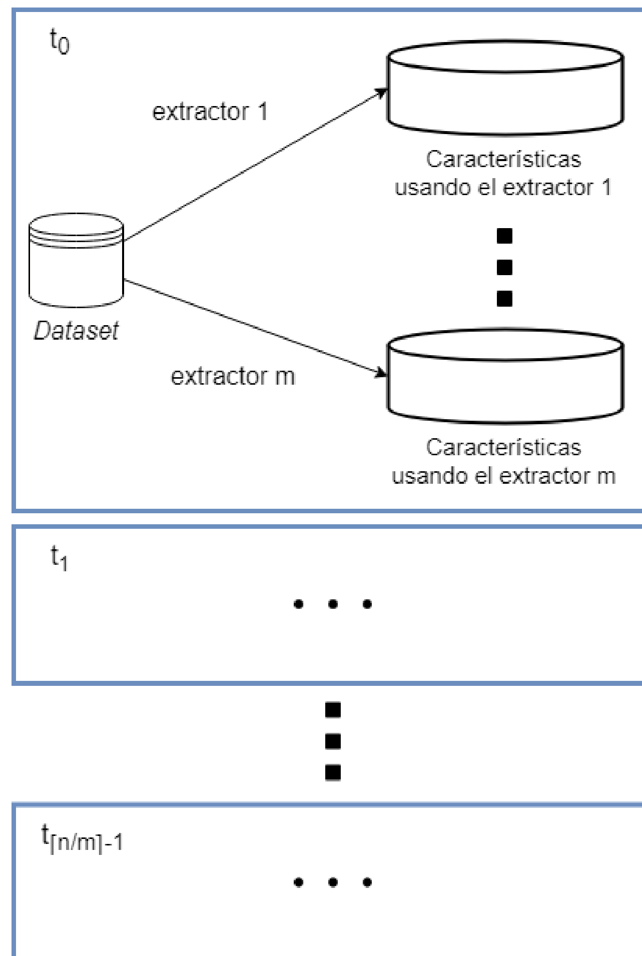

Figura 10. Nivel extractores. Se extraen usando m extractores a la vez (n total n extractores)

Es importante aclarar que los niveles, al ser una jerarquía, producen distintos resultados y se ejecutan un número distinto de veces. El nivel más alto es el de extractores, solo se ejecuta una vez y produce todas las características del dataset para todos los extractores. El siguiente es el de *batches* que se ejecuta una vez para cada extractor y produce todas las características según el extractor empleado. Finalmente el de imágenes se ejecuta una vez para cada *batch* (y los *batches* se generan para cada extractor) y produce las características de todas las imágenes del batch.

Por ello, los tiempos de las Figuras 8, 9 y 10 pueden ser diferentes pero se ha utilizado la misma notación. En las pruebas, m coincidirá con el número de núcleos del procesador utilizado.

4.3. Técnicas de evaluación

En este subapartado se hablará de cómo realizar las comparaciones entre el desempeño del código actual de FrImCla y los cambios a introducir. Al centrarse este trabajo en la paralelización del proceso se espera que se produzca una mejora en tiempo.

Cuando se pasa de una implementación secuencial a una en paralelo, la principal medida para las mejoras es el tiempo consumido. Para esto es importante contar con una salida como la de FrImCla que informa del tiempo que invierte en cada tarea. Por la elección del cuello de botella se analizará el tiempo del primer paso, la extracción de características.

Como se quiere realizar una comparación de todas las implementaciones es necesario contar con un *dataset* y una configuración inicial que vaya a utilizarse para todas las pruebas. Con esto se conseguirá el dato de tiempo original y luego los tiempos de las alternativas al realizar la prueba respectiva a cada una. Se propone pues, utilizar el mismo entorno de pruebas que en la subsección 3.3 ya que se cuenta con los tiempos conseguidos y utiliza una gran variedad de extractores de características.

4.4. Ampliaciones

Al realizar este diseño, sobre todo el estudio de alternativas Hadoop y Spark, se han detectado algunas mejoras que pueden hacerse en el futuro. Como estaba planificado si hubiese un excedente de tiempo podría realizarse una implementación que las contemplase pero por el momento quedan fuera del alcance de este trabajo y se exponen a continuación para dejar constancia de la posibilidad de hacerse.

En cuanto a la computación en clúster, distribuir el trabajo en varios ordenadores podría aumentar la capacidad de operaciones que pueden paralelizarse y ambas alternativas, Hadoop y Spark, están preparadas para este tipo de computación. Pero, al ser el propósito de FrImCla el ayudar a usuarios no tan experimentados y que quizás no cuentan con el hardware necesario, como un clúster, se deja como ampliación.

Por otro lado Spark cuenta con la librería para *machine learning* MLib de la que ya se habló. Debido a su rapidez podría mejorar el tiempo obtenido en el paso del análisis estadístico al reducir los tiempos empleados para entrenar. Esta se deja como alternativa pues aunque pueda presentar una mejora en los tiempos de entrenamiento el método que presenta el cuello de botella ya está paralelizado y tiene el problema de tener que esperar al más lento, que puede no resolverse aunque se realice esta mejora. Es decir el más lento puede acabar antes pero los demás deben esperarlo.

5. Pruebas

En esta sección se describen las implementaciones de las distintas alternativas presentadas en la sección anterior y se evalúan los resultados obtenidos ya sean mejoras en el tiempo o errores que impliquen que se descarte esta opción.

5.1. *Threading*

La primera alternativa consistía en utilizar los *threads* (hilos) del módulo *threading*.

Implementación:

Al implementar se han creado métodos que engloban las llamadas de FrImCla relacionadas con el tratamiento de una imagen o un *batch* de imágenes (pp. 18-20), por ejemplo: `predict(image)`, propio de los modelos de Keras (API de alto nivel de redes neuronales) [19] y que genera predicciones para la entrada, una imagen; y `describe(image)` del fichero `extractor.py`; o `describe(images)` que está en el fichero `index_features.py`. Lo anterior son las llamadas a los métodos, estos pertenecen a los extractores o a `extractor.py`.

Luego estos métodos se pasan a un *thread* como `target` y como argumentos (`args`) se añade todo lo necesario para efectuar las llamadas y recibir los resultados. Por ejemplo en `extractor.py` los datos que necesitan los métodos para los hilos son: la imagen a tratar en el hilo, el extractor y una lista para guardar las características extraídas de la imagen. En el caso de tratar con un modelo de TensorFlow (librería *open-source* para *machine learning*) [54, 55] también necesita el grafo de este, en el que cada nodo es una unidad de computación, una operación del modelo.

Durante la implementación se ha detectado que dos extractores de características, GoogleNet y Overfeat utilizan su método `transform`. Este funciona directamente a nivel de *batches* ya que el método recibe un conjunto de imágenes y devuelve las características de todas. Por ello, una paralelización más exhaustiva podría estudiar las operaciones que se realizan dentro de este y paralelizarlas de ser necesario.

Problemas:

Antes de presentar los resultados en los casos que ha funcionado esta implementación se exponen los errores, complicaciones y soluciones que han acarreado.

El primer error tiene que ver con la manera de interpretar los argumentos de un *thread* por parte de Python. Si se pasan argumentos a un *thread*, Python espera que el último sea algo iterable, es decir, que se puede aplicar un proceso iterativo (iterar) sobre sus componentes. Al proporcionar la imagen (que es un objeto iterable por ser una lista de listas) o una lista de resultados hay que tener cuidado pues puede producir errores al iterar sobre aquello que no queremos que se itere, por ejemplo las dimensiones en las imágenes. Para solucionarlo basta con poner una coma extra al final de la lista de argumentos.

Tras solucionar el paso de parámetros se producen errores de Keras [19] o TensorFlow, probablemente por la utilización del mismo recurso por varios hilos y esto produce condiciones de carrera. Uno de estos errores que tiene solución tiene que ver con el grafo de TensorFlow. El grafo debe devolverse a su estado por defecto (con el método `as_default()`) en cada hilo. Al hacerlo se reinicia la secuencia de operaciones a realizar y se puede evitar errores con algún modelo (en el ejemplo usado para las pruebas solo inception pero se asume que la situación se da con otros).

El resto de errores de Keras/TensorFlow se producen con los extractores VGG16, VGG19 y ResNet. Al lanzar los hilos, cada uno con una imagen o con un *batch* se puede producir en algunos una excepción que acaba con `AssertionError`. Buscar sobre este error ha llevado a intentar conseguir una `session` de TensorFlow para cada hilo pero esto recupera los errores del grafo y es contraproducente.

La excepción no detiene la ejecución del extractor actual pero si se produce (la excepción no sucede en todas las ejecuciones, aunque no se haya modificado el código) el siguiente extractor lanzará un `InvalidArgumentError` o dejará el *framework* en un estado inconsistente que previene que cargue el próximo extractor. Al producirse estos errores, al menos por lo observado, de manera aleatoria, se especula que en efecto se trate de condiciones de carrera por los recursos de Keras o TensorFlow. Si los métodos creados para los hilos se utilizan de manera secuencial, por ejemplo en un bucle *for*, los errores no suceden. Subir de nivel a extractores (Figura 10) produce errores similares.

Evaluación:

A continuación se presenta la Tabla 3 con los resultados de tiempo obtenidos utilizando distintas configuraciones. En esta ocasión se han usado varias configuraciones por los conflictos expuestos. Por facilidad se denomina **A** al conjunto de los extractores VGG16, VGG19 y ResNet, que son los conflictivos; **B** a GoogleNet y Overfeat, que son los que podrían paralelizarse a nivel de imagen modificando su método `transform` (no se ha hecho), se distinguen por tardar bastante más que el resto de modelos; y **C** el formado por los demás menos inception que pese a ser de Keras/TensorFlow no presenta los mismos errores. Serán los grupos a utilizar en el resto de evaluaciones.

Los extractores DenseNet y Xception (dos que podían haberse incluido) se han dejado fuera pues eran más lentos al cargar y ralentizaban las pruebas, además producen errores similares al grupo **A**. Estos errores están expuestos en el apartado anterior.

Extractores	Sin paralelizar (s)	Paralelizado (s)
A, **B**, **C** e inception	578.3367	564.8111*
A, **C** e inception	345.4532	337.3995*
C e inception	76.9848	76.3356
C	19.5357	18.2814

Tabla 3. Comparativa de tiempos entre código original y paralelización con *threading*

En la tabla se puede observar que se obtienen tiempos muy parecidos. Además en los casos donde se obtiene la mayor diferencia se ha producido alguno de los errores mencionados así que estos tiempos (marcados con un asterisco) han sido calculados sumando tiempos de usar los extractores problemáticos uno a uno y luego todos los no problemáticos.

Una última observación antes de rechazar esta técnica es la existencia del GIL (*Global Interpreter Lock*) [56] en Python, un bloqueo que impide a varios *threads* la utilización del mismo objeto Python al mismo tiempo. Esto es necesario porque la gestión de memoria no es *thread-safe* (segura cuando se usan hilos). Para esquivar este GIL es necesario cambiar de intérprete (por ejemplo a Jython o a IronPython [57, 58]) y se entiende que el usuario menos experimentado no tiene por qué saber hacer esto así que iría en contra de los propósitos de FrImCla realizar este cambio.

5.2. *Multiprocessing*

La segunda alternativa es utilizar los *processes* (procesos) del módulo *multiprocessing*.

Implementación:

Para la incorporación de esta técnica se ha decidido utilizar algo similar al MapReduce de Hadoop. En realidad solo necesitamos el trabajo de un *mapper* que se encarga de asignar las tareas a los procesos que las tienen que llevar a cabo. Un *mapper* sencillo en Python puede ser el siguiente:

```
from multiprocessing import Pool

class SimpleMapper(object):

      def __init__(self, map_func, n_works=None):
            self.map_func = map_func
            self.pool = Pool(n_works)

      def __call__(self, inputs, chunksize=1):
            self.pool.map(self.map_func, inputs, chunksize=chunksize)
```

Primero se importa el método constructor de objetos *pool* de procesos, `Pool`. A partir de ahí se declara la clase `SimpleMapper` que tiene dos métodos `__init__` y `__call__`, con sus respectivos parámetros. El primero se utiliza para crear un *mapper* y el segundo para llamarlo con los datos proporcionados en el parámetro `inputs` que debe ser iterable.

La función a paralelizar es `map_func` y el parámetro `chunksize` informa de cuántos elementos del iterable dar a cada trabajador. Notar que al constructor del *pool*, `Pool(n_works)`, se le puede dar el valor `None`. En ese caso crea tantos procesos como núcleos tenga el procesador.

En el caso de necesitar que el proceso devuelva una lista (esto sucede al trabajar al nivel de imágenes, pues se debe devolver una lista de las características extraídas) se puede emplear `return` en `__call__` ya que `map` devuelve una lista. Sin un `return`, `__call__` devuelve `None`.

Para utilizar este *mapper* se necesita crear un objeto de la clase `SimpleMapper` en dónde se quiere paralelizar una función que se pasa como `map_func`. Si solo tiene una entrada se pasa como la función de manera normal, como parámetro.

Si tiene varios parámetros de entrada se utiliza una función *wrapper* (envoltorio) que procese las entradas antes de llamar a la función objetivo con esos parámetros. Para que esto sea posible hay que incluir las entradas en una estructura de datos que sepa procesar la función *wrapper* que se utilice.

Por ejemplo se expone esta preparación (de los datos y la función *wrapper*) para el caso de paralelizar utilizando un *pool* de procesos a nivel de extractores. Este ejemplo busca entonces paralelizar el método `extractFeatures` de `index_features.py` que se llama una vez para cada extractor dentro del método `generateFeatures` del mismo fichero.

```
# En generateFeatures se sustituye el bucle for
# que iteraba sobre los extractores por
allData = [[fE, batchSize, imagePaths, outputPath,
            datasetPath, le, verbose]
            for fE in featureExtractors]
mapper = SimpleMapper(mapGenerate)
mapper(allData)
```

En este fragmento creamos la estructura `allData` que tiene un vector con los parámetros necesarios para cada llamada a `generateFeatures`, `fE` es el extractor y es el único parámetro que cambia pero los otros son necesarios para hacer la llamada y son fijos. Luego se crea el *mapper* pasándole como función `mapGenerate` que es una función *wrapper* sencilla para procesar cada vector de entradas (es decir cada componente de `allData`) a `generateFeatures`. El código de esta función es una simple llamada:

```
def mapGenerate(data):
  generateFeatures(data[0],data[1],data[2],data[3],data[4],data[5],data[6])
```

El *mapper* se ha encargado de trocear `allData` y le pasa a cada llamada de `mapGenerate` un *chunk*. El mantener el orden de parámetros de la cabecera de `generateFeatures` cuando se crea `allData` y que no se necesite procesar nada más hace que la función *wrapper* sea así de sencilla.

Para los otros niveles, imágenes y *batches,* se utiliza algo similar salvo que se cambia la función *wrapper* y se llama al *mapper* en las partes respectivas del código, sustituyendo los bucles que lo hacían secuencialmente.

Problemas:

Una vez más se comienza con los fallos detectados al realizar esta implementación para luego exponer los tiempos conseguidos en los casos en los que ha funcionado.

El principal fallo vuelve a ser la utilización del mismo modelo de Keras y TensorFlow en distintos procesos de forma paralela. Esto provoca las mismas situaciones que en el caso de *threading*.

Para las condiciones de carrera se puede devolver el grafo de TensorFlow a su estado por defecto (método `as_default`) o utilizar sesiones diferentes. Pero se produce el error:

```
can’t pickle _thread.RLock objects # Del modelo de Keras
```

Y es que los modelos como VGG16 o VGG19 (entre otros), poseen un RLock [59] o *lock* reentrante, es decir, que un mismo proceso o hilo puede pasar el bloqueo varias veces pero hasta que no acabe no puede pasar otro hilo. Y los procesos intentan usar el recurso, es decir, estos modelos aunque están bloqueados, por lo que se produce el error anterior.

Una solución a esto sería cargar los modelos que presentan este problema una vez por cada proceso y ejecutar todos los batches de ese extractor, pero intentarlo, al menos en el equipo utilizado para estas pruebas es demasiado y provoca el bloqueo del ordenador. Por esto, se considera algo fuera del objetivo de FrImCla ya que usuarios sin equipos dedicados deberían poder tener acceso a estas funcionalidades.

Como es la segunda vez que falla la implementación a nivel imágenes y *batches* por motivos de bloqueos, para las siguientes técnicas se investigará si es posible la implementación a estos niveles y de no serlo se pasará directamente al que sí ha funcionado en este caso, al nivel de extractores.

Evaluación:

La implementación a nivel de extractores expuesta en el apartado anterior si ha conseguido funcionar sin errores y extraer las características de las imágenes así como almacenarlas en los ficheros correspondientes y pueden ser utilizadas por el resto de FrImCla. En la Tabla 4 se realiza el mismo estudio que para *threading* creando los grupos denotados por: **A** al conjunto de los extractores VGG16, VGG19 y ResNet, que fueron los conflictivos en *threading*; **B** a GoogleNet y Overfeat; y **C** el formado por el resto menos inception.

Extractores	Sin paralelizar (s)	Paralelizado (s)
A, **B**, **C** e inception	578.3367	423.6310
A, **C** e inception	345.4532	302.3859
C e inception	76.9848	65.0971
C	19.5357	12.2939

Tabla 4. Comparativa de tiempos entre código original y paralelización con *multiprocessing*

Esta técnica ha conseguido mejorar los tiempos al utilizar los 4 núcleos disponibles para la extracción de características. Se puede notar un beneficio de casi un 27% al usar todos los extractores e incluso uno mayor del 37% al usar solo el grupo C. Obviamente al paralelizar solo a nivel de extractores el número de estos importa. Si solo se utiliza uno el resultado sería el mismo que no haber paralelizado, pero en el caso de usar muchos las mejoras van en aumento.

5.3. Hadoop

La tercera alternativa es la primera no nativa de Python, Apache Hadoop, programada en Java pero que cuenta con mecanismos para comunicarse con Python.

Instalación:

Al no ser nativa, se necesita instalar Java, Hadoop y realizar algunos ajustes en la configuración del último. Java tiene una instalación sencilla así que se obvia. Para este trabajo se ha usado la versión 1.8 de Java.

Las versiones de Hadoop se encuentran disponibles para descargar en su página oficial de lanzamientos [60]. Se ha utilizado la versión 3.1.1. Tras descargar el archivo tar se descomprime en la ruta deseada.

Lo primero a configurar es la conexión entre Hadoop y Java, para ello se localiza el fichero `hadoop-env.sh` en la ruta `etc/hadoop` dentro de la carpeta en la que se depositó Hadoop. En este fichero buscamos la línea (cerca del inicio del fichero):

```
export JAVA_HOME=${JAVA_HOME}
```

Y se cambia por la siguiente:

```
export JAVA_HOME=/ruta/hasta/bin/java
 # Por ejemplo /usr/bin/java
 # Puede mirarse con el comando “which java”
```

Para probar la instalación y configuración en cualquier momento se puede usar el comando `bin/hadoop` (en realidad es ejecutar `hadoop` dentro de la carpeta bin) desde la carpeta que lo contiene. Este comando muestra una lista de otros comandos y confirma la correcta instalación y configuración.

Para simular el trabajo con varios nodos utilizando un solo ordenador se necesita pasar al modo *pseudo-distributed* (pseudo distribuido) de Hadoop. Esto permite que el ordenador utilice los cuatro núcleos. Si no se realiza esta configuración Hadoop viene por defecto en *single-node* o *non-distributed* (un solo nodo o no distribuido). Para ello se necesita editar los archivos `core-site.xml` y `hdfs-site.xml` en la misma ruta que el fichero anterior.

En `core-site.xml` se añade:

```
<configuration>
      <property>
            <name>fs.defaultFS</name>
            <value>hdfs://localhost:9000</value>
      </property>
</configuration>
```

Esto establece el sistema de ficheros por defecto al local en el puerto 9000. Antes de modificar `hdfs-site.xml` se deben crear dos carpetas, `namenode` y `datanode`, en la ruta `hadoop_store/hdfs` dentro de la carpeta de Hadoop. Con las carpetas creadas se añade al fichero xml lo siguiente:

```
<configuration>
      <property>
            <name>dfs.replication</name>
            <value>1</value>
      </property>
      <property>
            <name>dfs.namenode.name.dir</name>
            <value>
            file:/ruta/hasta/hadoop_store/hdfs/namenode
            </value>
      </property>
      <property>
            <name>dfs.datanode.data.dir</name>
            <value>
            file:/ruta/hasta/hadoop_store/hdfs/datanode
            </value>
      </property>
</configuration>
```

Tras esto queda crear un par de claves RSA para permitir conexiones SSH y dar formato de HDFS (*Hadoop Distributed File System*) a la carpeta `namenode`. Para ello basta con usar los siguientes comandos:

```
# Para generar las claves RSA para SSH
ssh-keygen -t rsa -P '' -f ~/.ssh/id_rsa
cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
# Para dar el formato HDFS
bin/hdfs namenode -format
```

Con todo realizado Hadoop debería iniciarse al ejecutar `sbin/start-dfs.sh` y pararse con `sbin/stop-dfs.sh`, ambos se encuentran en la carpeta de Hadoop (dentro de `sbin`).

Implementación:

Antes de empezar a utilizar Hadoop se realizó una evaluación de la posibilidad de paralelizar a niveles (pp. 18-20) inferiores al de extractores, como *batches* o imágenes. La implementación en Hadoop va a pasar por crear un *job* o tarea que tendrá un *mapper*. Este paralelizará las llamadas al método que se le indique con los parámetros que se le pasen para cada llamada. Sin embargo los niveles de *batches* o de imágenes, aunque realizan varias llamadas a métodos, también son llamados varias veces por el nivel inmediatamente superior.

Por esto se decide paralelizar únicamente a nivel de extractores. De lo contrario habría que crear más archivos para simular todas las ejecuciones de los niveles inferiores y organizar la

información devuelta en archivos. Como última nota cabe la posibilidad de que si se intenta ejecutar algún modelo de Keras/TensorFlow en varios hilos o procesos se produzcan los errores anteriores.

Lo primero a desarrollar para la implementación en Hadoop es un fichero Python que nos permita pasar los parámetros de las distintas llamadas a un fichero de texto, utilizando una línea para cada parámetro. Para esto se ha creado el fichero `prepLines.py` que lee las configuraciones necesarias para la extracción de características, siendo la más importante que extractores se quieren utilizar, y las escribe en un fichero de texto separando los parámetros con tabuladores y las llamadas con saltos de línea.

Una vez generado este fichero hay que almacenarlo en el HDFS (sistema de ficheros de Hadoop) que se creó durante la instalación. Para ello primero se una estructura de carpetas (puede ser diferente) en HDFS utilizando la opción `-mkdir`:

```
bin/hdfs dfs -mkdir /features/in
bin/hdfs dfs -mkdir /features/out
```

A continuación se copia el fichero de texto en la carpeta de *inputs*, la otra carpeta contendrá los resultados de la ejecución mostrados por consola (no los ficheros de características, estos estarán en otra ruta que se dirá más adelante).

```
# Para copiar a HDFS
bin/hdfs dfs -copyFromLocal /ruta/del/fichero.txt /features/in
# Para copiar desde HDFS (útil para sacar el output)
bin/hdfs dfs -copyToLocal /features/out /ruta/en/local
```

Tras esto es necesario crear varios archivos y directorios para lograr una estructura similar a esta:

```
|--- deploy
|    |--- frimcla.zip
|--- jobs
|    |--- features_extraction_run.sh
|--- output
|--- frimcla
|    |--- __init__.py
|    |--- hadoop
|    |    |--- __init__.py
|    |    |--- mapper
|    |    |    |--- __int__.py
|    |    |    |--- mapper.py
|    |    |--- reducer
|    |    |    |--- __int__.py
|    |    |    |--- reducer.py
|    |--- otras carpetas y archivos
|--- sbin
|    |--- deploy.sh
|    |--- start.sh
|    |--- stop.sh
|--- config.sh
|--- extraction_mapper.py
|--- prepLines.py
```

La primera carpeta, `deploy`, contiene el código desarrollado, comprimido en formato zip para pasarlo a HDFS en la ejecución del *job* que se encuentra en la carpeta `jobs`. En esta última estarán todos los *jobs* de MapReduce que queremos lanzar. Los ficheros `.sh` a excepción de `config.sh` deben hacerse ejecutables con `chmod +x fichero.sh`.

La estructura básica de uno de los *job* es la siguiente:

```
#!/bin/sh

# Directorio actual (el que tiene la estructura anterior)
BASE=$(pwd)
# Crear frimcla.zip
sbin/deploy.sh
# Cambiar a donde se puso Hadoop
cd $HADOOP_HOME
# Quitar modo seguro (a veces opcional)
bin/hdfs dfsadmin -safemode leave
# Eliminar el output anterior
bin/hdfs dfs -rm -r /features/out
# Ficheros locales para el job, separados con “,”
FILES=”${BASE}/extraction_mapper.py,\
${BASE}/deploy/frimcla.zip”

# Ejecutar el job en Hadoop
bin/hadoop jar share/hadoop/tools/lib/hadoop-streaming-*.jar \
    -D mapreduce.job.reduces=0 \
    -files  ${FILES} \
    -mapper ${BASE}/face_detector_mapper.py \
    -input /features/in/lines.txt \
    -output /features/out
```

Básicamente hace lo que indican los comentarios. Tras la configuración inicial lo importante es la ejecución del *job* que recibe, entre otros, el *mapper* a utilizar. El *mapper* utilizado será el último fichero que se explicará en la implementación antes de pasar a la evaluación. Antes de ello se explican por encima el resto de ficheros.

La carpeta `output` se crea como destino para la opción `-copyToLocal` de HDFS, pero reescribiendo las rutas usadas por FrImCla se puede hacer que lo producido por el *framework* acabe en ella. De lo contrario las rutas originales provocan que los ficheros de características se creen en `$HADOOP_HOME`. También puede usarse para ambos propósitos la carpeta `frimcla` que contiene el código del FrImCla y es donde originalmente se crean la carpeta `output` (automáticamente por FrImCla) y dentro los archivos de características.

Dentro de `frimcla` también se ha añadido una carpeta `hadoop` que se usa para contener clases base para cualquier *mapper* o *reducer*. En el caso de este proyecto al no necesitar *reducer* (porque los ficheros de características son la salida del método a paralelizar) y ser un *mapper* sencillo no se han creado estos ficheros pero se incluyen en el esquema por ser la estructura típica al trabajar con MapReduce.

La carpeta `sbin` contiene tres ficheros ejecutables que sirven para: crear `frimcla.zip` a partir de la carpeta `frimcla`, `deploy.sh`; iniciar hadoop, `start.sh`; y pararlo, `stop.sh`. Finalmente solo queda hablar del fichero `extraction_mapper.py`.

Este fichero tiene un import especial para usar el fichero zip de FrImCla:

```
# El resto de import se obvian
import sys
sys.path.insert(0, "frimcla.zip")
from frimcla.index_features import extractFeatures
```

Tras esto presenta tres funciones típicas de un *mapper*, controlar el flujo de entrada, pasando línea a línea al *job*; procesar la entrada para conseguir los datos, cada línea tiene los parámetros de una llamada separados por tabuladores (además algunos como los booleanos deben convertirse de cadena a booleano); y pasar la salida del *mapper* al *reducer*. Al no tener paso de Reduce este último es irrelevante.

El método importante de `extraction_mapper.py` es `job`. El método tiene un bucle que se encarga de la paralelización pues separa la ejecución por cada línea de la entrada (el fichero de texto). Dentro, en la primera iteración inicia variables comunes para todas y luego pasa estas variables y datos extraídos de la línea que toca al método `extractFeatures` de FrImCla. Al empezar y al acabar el método `job` se toman tiempos para conseguir los datos del apartado de evaluación.

Evaluación

Al haber realizado un estudio previo de los problemas que podían suceder en los niveles inferiores no ha ocurrido ningún inconveniente con la implementación propuesta y la versión con Hadoop se ejecuta sin fallos. En la Tabla 5 se encuentran los tiempos obtenidos para cada grupo siendo: **A** el conjunto de los extractores VGG16, VGG19 y ResNet, que fueron los conflictivos en *threading*; **B** el formado por GoogleNet y Overfeat; y **C** el resto de extractores salvo inception.

Extractores	Sin paralelizar (s)	Paralelizado (s)
A, **B**, **C** e inception	578.3367	575.5641
A, **C** e inception	345.4532	346.0862
C e inception	76.9848	66.7988
C	19.5357	19.4837

Tabla 5. Comparativa de tiempos entre código original y paralelización con Hadoop

Esta vez los resultados son muy cercanos a los de no haber utilizado ninguna paralelización. Tras investigar más sobre estos resultados se puede deducir que Hadoop sin un clúster real de ordenadores dónde realizar la paralelización (o muchos núcleos para el modo *pseudo-distributed*) no consigue la suficiente mejora. Pero un clúster escapa a los objetivos de FrImCla.

Además estos resultados no solo reflejan el tiempo empleado en la extracción sino también el que consume el *job* en iniciar Hadoop, borrar el antiguo *output*, cambiar a modo no seguro, conectarse con Java y organizar las tareas. No se han considerado el tiempo de generar las líneas del fichero de texto ni el de pasar este a HDFS pero son comandos extra a realizar cada vez que se quiera cambiar la configuración (por ejemplo los extractores a utilizar, el *dataset* o el tamaño de un *batch*).

5.4. Spark

La última alternativa tampoco es nativa a Python. Spark, basada en Hadoop MapReduce.

Instalación:

Al no ser nativa también se debe instalar, pero al contrario que Hadoop es muy sencillo, pues queremos trabajar con la versión compatible con Python, PySpark [61]. Para ello basta con utilizar el comando `pip3` (pues en este trabajo se está usando Python 3.6) ya que PySpark se encuentra en PyPI (*Python Package Index*) [62]. El comando completo es:

```
pip3 install pyspark
```

Para hacer algunas pruebas con el código de PySpark durante la investigación también se ha usado Jupyter Notebook (una aplicación web *open-source* que soporta más de cuarenta lenguajes de programación y permite compartir código e integra herramientas como Spark o TensorFlow)[63]. La instalación es similar, basta con sustituir `pyspark` por `jupyter`. Es necesario disponer de Java pero como se instaló la versión 1.8 para Hadoop ya se tiene.

Implementación

Antes de comenzar la implementación hubo una etapa de familiarización con el API de PySpark a través de ejemplos y sobre todo la documentación [61]. Dentro de esta, el paquete `pyspark` y el módulo `pyspark.sql` son los interesantes pues son los que se utilizarán para la implementación. Aún así los otros paquetes `pyspark.ml` y `pyspark.mllib`. Pueden ser útiles para la ampliación propuesta en el diseño, para mejorar los tiempos de la fase de análisis estadístico.

Volviendo a realizar un estudio previo sobre la posibilidad de paralelizar a nivel de *batches* o imágenes (pp. 18-20) se ha llegado a la conclusión de que con esta técnica tampoco se puede (y probablemente con ninguna). Para realizar el estudio, debido a lo simple que es implementar PySpark una vez comprendido, se ha preparado la implementación para el nivel de *batches* y se han vuelto a producir los mismos errores de `_thread.RLock objects`. Conseguir paralelización utilizando el mismo modelo de Keras/TensorFlow en distintos procesos o hilos se ve difícil sin programar los modelos para que permitan esto desde un principio, pero entonces la paralelización estaría en el lado de Keras o TensorFlow, no en el de FrImCla.

Dicho esto la implementación de Spark (PySpark) en FrImCla a nivel de extractores es sencilla, cuando se entiende lo que se hace. En sí no es nada más que tres líneas de código utilizando varios métodos de Spark (obviando la importación de paquetes o módulos necesarios). En concreto, consiste en añadir las líneas en el fichero `index_features.py`, dentro del método `generateFeatures` sustituyendo el bucle que llamaba a `extractFeatures` secuencialmente.

```
spark = SparkSession.builder.master("local[4]")\
                     .appName("FrImClaExtract")\
                     .getOrCreate()
sc = spark.sparkContext
sc.parallelize(featureExtractors)\
     .map(lambda fE: extractFeatures(fE, batchSize,
          imagePaths, outputPath, datasetPath, le,
          verbose))\
     .count()
```

Con la primera línea creamos una sesión de Spark que va a ejecutarse en local con cuatro núcleos, se puede cambiar esto cambiando el número entre corchetes. El resto de métodos le dan el nombre a la aplicación (opcional) y finalmente acaba la creación de la sesión con `getOrCreate` que, como su nombre indica, recupera una sesión existente o crea una nueva a partir de las opciones del `builder`.

La segunda línea simplemente consigue el objeto que representa al contexto de Spark de la sesión y lo almacena en la variable `sc` que va a ser sobre la que se usen los métodos más interesantes. Los de la tercera línea.

Un objeto `SparkContext` es la puerta de entrada a toda la funcionalidad de Spark. Sobre este objeto se aplica el método `parallelize` que toma como parámetro una colección de los extractores de características (esta está sacada de la configuración y es una variable que ya tenía FrImCla). Con una colección de Python `parallelize` crea un RDD [50] (*Resilient Distributed Dataset*) que es la abstracción de datos básica para Spark. Los RDD están basados en el HDFS de Hadoop, son tolerantes a fallos y permiten operaciones en paralelo.

Una vez se tiene un RDD se pueden realizar distintas transformaciones sobre él (recogidas en la documentación referenciada) pero la que se usa en la tercera línea es `map`. Esta recibe una función por la que pasará todos los elementos del RDD, en este caso todos los extractores, y creará un nuevo RDD con los resultados de la función.

La función es una expresión lambda para hacer una función *wrapper* anónima que recoge el extractor (`fE`) que toca y lo pasa a una llamada del método `extractFeatures`. El resto de parámetros de `extractFeatures` son del método `generateFeatures` (en el que se está implementando el código) o han sido declarados dentro de este último.

Las transformaciones de Spark son *lazy* (perezosas), mientras se van utilizando se acumulan en un grafo DAG (*directed acyclic graph*, un grafo dirigido sin ciclos que sirve para mantener el orden de la secuencia de transformaciones a realizar) [64, 65] pero no se ejecutan hasta que se detecta una acción. Este proceso perezoso tiene como ventajas la optimización por parte de Spark, facilidad de recuperación tras fallo porque mantiene la linealidad de las operaciones y si se produce un fallo este se muestra tras procesar todo el grafo de transformaciones. La acción escogida no tiene mucha importancia pero se ha implementado con `count` por ser una de las más simples y que cuenta el número de elementos del último RDD.

Evaluación:

Los únicos fallos que se han producido han vuelto a ser los de Keras/TensorFlow pero al ser considerados dentro de una prueba para ver si podía darse paralelización con Spark a nivel de *batches* o imágenes no se tienen en cuenta. No se puede hacer a esos niveles, al menos no con todos los modelos.

Para el nivel de extractores la implementación propuesta se ha ejecutado correctamente y ha producido los resultados recogidos en la Tabla 6. Se recuerda la notación de los grupos de extractores: **A** el conjunto de los extractores VGG16, VGG19 y ResNet, que fueron los conflictivos en *threading*; **B** el formado por GoogleNet y Overfeat; y **C** el resto de extractores salvo inception.

Extractores	Sin paralelizar (s)	Paralelizado (s)
A, **B**, **C** e inception	578.3367	493.2846
A, **C** e inception	345.4532	322.2236
C e inception	76.9848	72.6127
C	19.5357	21.2930

Tabla 6. Comparativa de tiempos entre código original y paralelización con PySpark

Spark ha conseguido mejorar los tiempos en tres de los cuatro escenarios en los que se ha probado, pero solo en dos de ellos la diferencia es algo significativa. En el caso de C e inception solo ha recortado cuatro segundos.

Podemos razonar de forma similar a Hadoop en los casos en los que no ha conseguido casi mejora, no es una técnica nativa y debe establecer conexiones con Java, sistemas de almacenamiento de los datos (los RDD) y sistemas de almacenamiento de las transformaciones sobre los RDD (un grafo DAG).

En definitiva, hace más que simplemente repartir las tareas y ejecutarlas como hacen los módulos nativos de Python, *threading* y *multiprocessing*, pero el resto se deja para la siguiente sección en la que se comparan los resultados.

6. Comparativa

Al haber acabado las pruebas se puede hacer una comparación de los resultados obtenidos con cada técnica, API o módulo empleado. El objetivo es aclarar cuál es la mejor opción de las que se han probado, aunque se puede justificar que otras funcionarían mejor en un contexto distinto al de FrImCla.

En la Tabla 7 se pueden ver todos los resultados de manera conjunta. Por última vez, la notación utilizada para los grupos de extractores es: **A** para el conjunto de los extractores VGG16, VGG19 y ResNet, que fueron los conflictivos en *threading*; **B** para el formado por GoogleNet y Overfeat; y **C** para el resto de extractores salvo inception.

Extractores	Sin paralelizar	*Threading*	*Multiprocessing*	Hadoop	Spark
A, **B**, **C** e inception	<u>578.3367</u>	564.8111*	**423.6310**	575.5641	493.2846
A, **C** e inception	345.4532	337.3995*	**302.3859**	<u>346.0862</u>	322.2236
C e inception	<u>76.9848</u>	76.3356	**65.0971**	66.7988	72.6127
C	19.5357	18.2814	**12.2939**	19.4837	<u>21.2930</u>

Tabla 7. Comparativa de tiempos entre código original y todas las pruebas

Para hacer más fácil el análisis de estos resultados se han marcado en negrita los mejores tiempos de cada categoría, subrayado los peores y puesto un asterisco en aquellos que no podían acabar su ejecución sin alguna excepción o error (pero que se han medido poco a poco para realizar esta tabla). Para medir los tiempos en general se han realizado varias ejecuciones y reflejado la media de todas.

Como peor técnica de paralelización se tiene al módulo de *threading* que pese a ser nativo de Python presenta algunos problemas a la hora de paralelizar los modelos de Keras/TensorFlow que las demás no han tenido. Es interesante ver que en los niveles de *batches* o imágenes ninguno ha podido paralelizarlos. En parte esto ayuda a que la tabla de comparación sea más fiel pues todas las paralelizaciones son al mismo nivel, el de extractores. Pero incluso en este nivel, *threading* ha presentado las mismas excepciones que en los inferiores y si no puede funcionar con una gran variedad de extractores no está siguiendo el propósito de FrImCla. Quizás pueda ser debido al GIL del que se habló en la evaluación de sus pruebas pero por todo ello se queda con el último puesto.

En tercera y segunda posición se encuentran las técnicas no nativas de Python, Hadoop y Spark, respectivamente. Al ser similares los comentarios a realizar sobre ambas se juntan en este párrafo. Hadoop consigue mejoras muy sutiles (incluso en uno empeora), pero funciona en todos los escenarios, al contrario que *threading*. Spark por el otro lado consigue, salvo en el caso del grupo C (pero es que este grupo es demasiado rápido sin paralelizar, solo con la estructura de Spark se empeora el tiempo), mejoras más notables. Lo común a ambas es que están orientadas a la computación en clúster y en caso de disponer de uno quizás Spark podría conseguir los mejores tiempos de toda la tabla, pero una vez más, tener un clúster de ordenadores (o alquilarlo), no es algo que se considere cuando se piensa en el usuario poco familiarizado y sin recursos dedicados.

Finalmente y como mejor opción se encuentra el módulo de *multiprocessing*. Este ha conseguido mejorar todos los tiempos del código original y además los mejores tiempos en todos los escenarios. La justificación podría ser que si cabe en memoria (como debe ser la información que maneja FrImCla, de lo contrario se dispondrá algún *hardware* dedicado) esta es la mejor opción, por ser nativa a Python, por no necesitar organizar la paralelización (o la organización necesaria estar optimizada en Python) y otros motivos que lo hacen mejor a *threading* como carecer de un GIL o dedicar realmente un núcleo a cada proceso.

La mayor pega de estos resultados es no haber podido conseguir paralelización a niveles inferiores. De haberlo hecho se podrían haber combinado niveles o conseguido mejores resultados en el caso general pues con un número de extractores constantes y un tamaño de *dataset* variable se podría continuar paralelizando.

El problema vuelve a ser las limitaciones, de *software* y *hardware*. Por el lado del *software* los modelos de Keras/TensorFlow no pueden utilizarse en dos hilos o procesos, ya que tienen un bloqueo que lo impide. Por el lado del *hardware* se podría solucionar esto cargando un modelo, del mismo tipo pero diferente instancia, en cada hilo o proceso que lo necesite, pero esto sobrepasa la memoria o capacidad de computo de un ordenador normal sin *hardware* dedicado.

7. Seguimiento y control

En esta sección se deja constancia de las semanas y los tiempos reales invertidos en la elaboración de este proyecto en comparación con los planificados. También se dan las causas de las desviaciones más significativas. Pero lo primero a señalar es cómo se ha llevado a cabo este seguimiento y la comunicación con el tutor del TFG.

Seguimiento del tutor académico:

Al tutor se le ha ido informando de las partes acabadas y redactadas. Ciertamente al principio hubo un problema de coordinación pues se pensó que, por usar una herramienta como Google Docs para la redacción, el tutor siempre tenía constancia de qué estaba redactado y qué no. Pero obviamente el tutor no puede estar pendiente al TFG en todo momento.

Tras las primeras incidencias se hizo habitual el sistema de: informar al tutor, este revisaba lo redactado y hacía sugerencias de mejora, estas se toman en cuenta por el alumno y se llevan a cabo. En algunas ocasiones se organiza una reunión para revisión de algunas sugerencias dudosas y se decide entre alumno y tutor si estas son necesarias o no. En estas también se podían solucionar dudas con aspectos teóricos o prácticos (en el caso de las implementaciones).

Seguimiento del alumno:

El alumno, a parte de la elaboración de este trabajo y los puntos de control marcados en la planificación, ha llevado un documento en paralelo a modo de diario de los hitos para hacer más fácil el recuento de horas y semanas final y a la hora de los hitos alcanzados en un punto de control. El resultado de este seguimiento es lo que se ve en los siguientes apartados.

Puntos de control:

1. **17 de marzo de 2019:** Se ha finalizado la redacción de la planificación a día 12, tras informar al tutor se sugieren los primeros cambios. Se ha empezado a trabajar, en sucio, en análisis y diseño pero se considera parte de la semana 5. Los cambios propuestos son lo suficientemente grandes, por la inexperiencia en el tema de este trabajo, como para suponer un retraso.

2. **21 de abril de 2019:** La fase de planificación termina en la semana 8 por las desviaciones expuestas luego. Por otro lado la fase de análisis está a punto de ser acabada y la de diseño acaba de empezar a pero como va muy vinculada al análisis está bastante pensada y será fácil terminarla. Con el retraso producido hasta ahora de dos semanas se prevé terminar la redacción de ambas fases para la semana 11.

3. **19 de mayo de 2019:** Las fase de diseño fue terminada en la semana 11, sus correcciones rápidamente solventadas entre el fin de semana correspondiente y el lunes de la semana 12. Desde entonces se ha trabajado en las implementaciones de *threading* y *multiprocessing*. En el caso de *threading* hubo problemas que ocasionaron un retraso pero está redactada y se utilizará de plantilla para el resto. La de *multiprocessing* está casi terminada, se prevé tenerla acabada para la semana 14 acabar con las pruebas y revisar Hadoop y Spark para las semanas de después de la 15.

4. **16 de junio de 2019:** Tras acabar las pruebas de *multiprocessing* en la semana 14 se hizo un pequeño parón de una semana por la preparación y realización de los exámenes finales y la entrega de proyectos finales. Sin embargo durante la semana de exámenes, la 15, se aprovechó para revisar errores cometidos en la redacción y refrescar las lecciones de Hadoop.

   Al llegar la semana 16 se puede empezar a trabajar casi todo el día (5 horas de mañana y 4 de tarde, más lo empleado en investigar que no se ha contabilizado con exactitud) en el proyecto por lo que se avanza mucho más rápido. En concreto la semana 16 se logra realizar toda la instalación, implementación y pruebas de Hadoop y en la semana 17, tras una pequeña investigación (recordatorio) sobre qué se necesita de Spark, se realiza lo mismo para la última librería. El tiempo que queda se dedicará a pequeñas pruebas, para sacar mejores datos que son los que aparecen en este trabajo y la redacción o corrección de toda la memoria.

Diagrama Gantt real:

Tras ver los puntos de control se expone el diagrama Gantt original (Figura 11) seguido del diagrama Gantt que refleja las semanas ocupadas realmente por cada parte (Figura 12). Se ha respetado el diagrama Gantt marcando desde el inicio de una tarea hasta la semana en la que se concluyó. La explicación de las desviaciones en semanas y en tiempos (siguiente subapartado) aparecen en el último subapartado de esta sección.

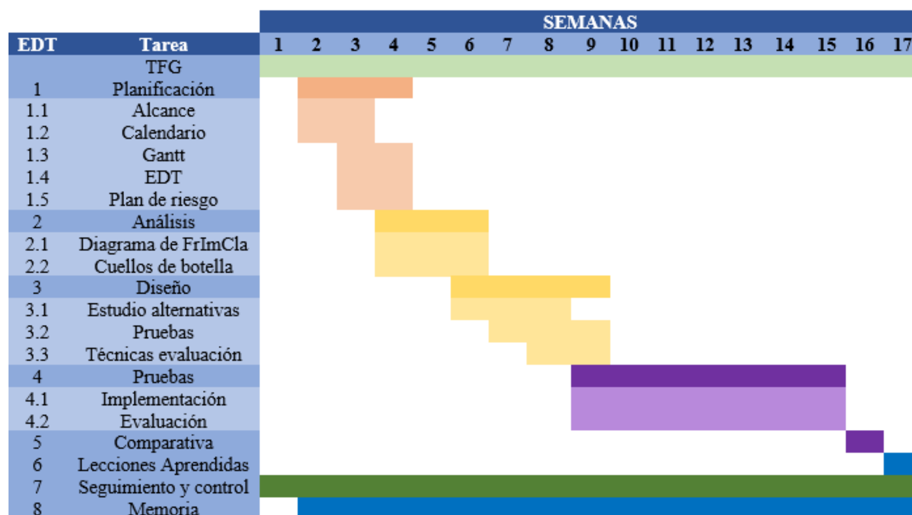

Figura 11. Diagrama Gantt original

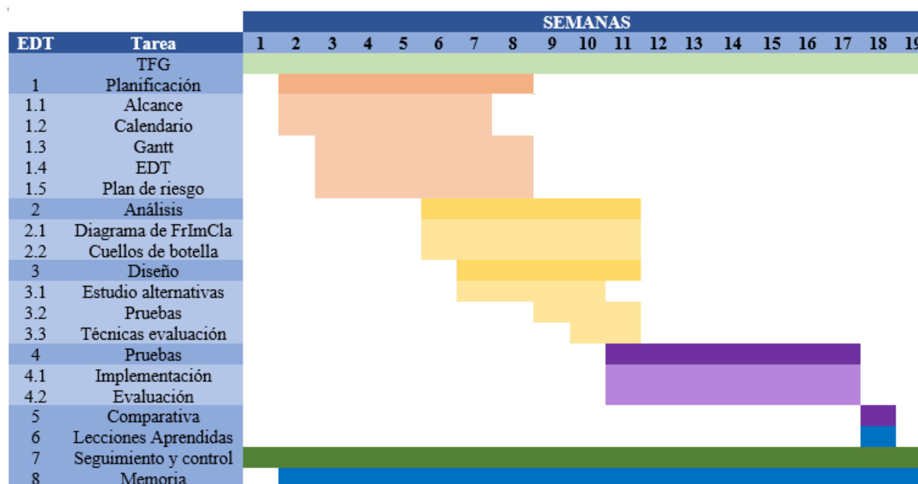

Figura 12. Diagrama Gantt final

Tiempos reales:

Para ver las desviaciones antes de explicar las causas de cada una es necesaria la Tabla 8, en la que se comparan los tiempos planificados, los empleados y las desviaciones en cada uno de ellos.

EDT	Tarea	Horas planificadas	Horas empleadas	Desviación
1	**Planificación**	**25**	**17**	**-8**
1.1	Alcance	5	7	+2
1.2	Calendario	5	3	-2
1.3	Gantt	5	3	-2
1.4	EDT	5	2	-3
1.5	Plan de riesgo	5	2	-3
2	**Análisis**	**40**	**37**	**-3**
2.1	Diagrama de FrImCla	20	15	-5
2.2	Cuellos de botella	20	22	+2
3	**Diseño**	**60**	**50**	**-10**
3.1	Estudio alternativas	20	20	0
3.2	Pruebas	20	15	-5
3.3	Técnicas evaluación	20	15	-5
4	**Pruebas**	**110**	**110**	**+4**
4.1	Implementación	100	110	+10
4.2	Evaluación	10	4	-6
5	**Comparativa**	**5**	**3**	**-2**
6	**Lecciones Aprendidas**	**5**	**2**	**-3**
7	**Seguimiento y control**	**15**	**10**	**-5**
8	**Memoria**	**50**	**75**	**+25**
	TOTAL	**310**	**308**	**-2**

Tabla 8. Comparativa y cálculo de las desviaciones en los tiempos

Desviaciones:

La mayor desviación que se puede observar en la tabla es la redacción de la memoria y es que esta ha sido la principal causa del retraso en el trabajo. Por ejemplo, si se considera la planificación esta llevó bastante menos tiempo en realizarse del que se propuso en un primer momento, pero cambios, correcciones, o incluso añadir nuevos apartados a la memoria en las semanas que se hacía la planificación provoca el retraso en considerar acabada esta fase en el diagrama Gantt. No se había podido declarar completamente acabada hasta entonces.

Otras causas de las desviaciones son la inexperiencia con las tecnologías o terminologías utilizadas en este trabajo que requerían de un tiempo extra de preparación e investigación. Este no se cuenta en las horas dedicadas pero sí ha causado sobre todo el retraso reflejado en el diagrama Gantt. Además de este tiempo el dedicado a clases, trabajos o entregas del curso académico que sucedía en paralelo y otras tareas de investigación han consumido horas que no pueden ser consideradas como del trabajo. Todas estas causas, a excepción de dejar una parada para la última semana y media del curso (la semana 15, de exámenes), fueron desapareciendo al ganar algo de experiencia y tener una visión clara de qué tareas efectuar cada semana. En definitiva, poco a poco se fue consiguiendo una mejor coordinación.

Las fases de análisis y diseño fueron, como se suponía, muy a la par. Comienzan con el retraso que viene de la planificación (y extras, pues como se ha dicho no fue solo la planificación) pero una vez empezadas incluso se tarda menos tiempo del planificado en terminarlas. Una vez más el tiempo ahorrado en estas fases, casi se traduce directamente a tiempo invertido en la redacción de las mismas. Se empezó con una estructura pero unos ligeros cambios en la planificación (correcciones) llevaron a cambiarla y redactar de nuevo. También se tuvo que elaborar ejemplos para un mejor comprensión del apartado de diseño. Pero el mayor problema era la inexperiencia a la hora de reflejar datos de un programa como FrImCla desde el punto de vista de este trabajo que busca mejorarlo. Durante la carrera se han visto muchas herramientas de análisis o diseño, por ejemplo UML, pero aquí no se usa ninguna y esto hace que la forma de presentar lo estudiado tenga que llevarse con más cuidado.

La última gran fase es la de implementación (o Pruebas), en este se dedicó más tiempo del esperado a las implementaciones debido a errores no previstos y búsqueda de soluciones para ellos. Esta búsqueda llevó a la conclusión de que los fallos eran inevitables, en el marco de FrImCla, pero se gastó tiempo en ella. Aún así la evaluación de cada método pudo ser más rápida de lo esperado y no supuso un retraso tan grande.

Por la manera que se realizaron los apartados anteriores la preparación de la comparativa de las alternativas fue bastante sencilla. Las lecciones aprendidas son conclusiones o reflexiones que han surgido durante la redacción o implementación de este trabajo así que es difícil de contabilizar pero ha llevado menos tiempo del previsto por poderlo hacer en paralelo. El seguimiento y control ha llevado menos tiempo del esperado por haber automatizado el proceso de llevar el diario (aunque era más un registro de hitos personal).

El total de horas es cercano al esperado aunque se haya redistribuido, sobre todo a favor de la redacción de la memoria. Si se considera la redacción de la memoria dentro de cada parte los tiempos cambiarían menos, pero el Gantt seguirá con el retraso de dos semanas inicial.

8. Lecciones aprendidas y conclusiones

Esta sección trata sobre pequeños apuntes sobre los problemas encontrados en este trabajo que han servido para aprender algo en caso de volver a trabajar en un proyecto similar. Quizás no son lecciones complejas para alguien más familiarizado con la paralelización de Python o modelos de clasificación como los de Keras o TensorFlow, pero para alguien que no lo esté pueden ser una ayuda. También se pueden considerar conclusiones de lo aprendido.

Trabajar con modelos de Keras y TensorFlow a nivel de *batches* o imágenes:

Al ser inexperto en el campo de las redes neuronales empecé la paralelización para todos los modelos como si fueran similares y esto me llevó a fallos con los modelos de Keras y TensorFlow, por ejemplo con VGG16 y VGG19.

En las evaluaciones de cada implementación dejo constancia de los errores que se producen al utilizarlos pero aquí aprovecho para juntarlo a modo de conclusión. Los modelos no se pueden utilizar en paralelo. Quizás se pueda hacer una versión de los mismos que trabaje en paralelo al pasarle dos imágenes, pero la utilizada por FrImCla no es así.

Las soluciones a esto que se me ocurren van desde las versiones mencionadas a tener la potencia necesaria para cargar un modelo (del mismo tipo pero distinta instancia) en cada hilo o proceso. Pasando por tener un clúster de ordenadores y claro, cada nodo del clúster tendrá su modelo propio (una vez más del mismo tipo que el resto de nodos, pero otro modelo) y podrá usarlo para sus imágenes.

Estas soluciones permitirían la paralelización que no he conseguido en este trabajo, pero las considero fuera del alcance del mismo por estar fuera del alcance de FrImCla. Recuerdo que se supone que los usuarios del *framework* no tienen los componentes *hardware* específicos para la creación de modelos de clasificación.

Los problemas de *threading* y de Hadoop *pseudo-distributed*:

El módulo de *threading* fue el que más problemas ha dado de todas las implementaciones y está relacionado con el GIL (*Global Interpreter Lock*). Básicamente los hilos utilizan la misma zona de memoria para ejecutarse y los procesos utilizan una zona dedicada para cada uno. Para proteger esta zona de memoria existe el GIL y ha sido la causa oculta de los problemas de esta parte.

Por otro lado tenemos el modo *pseudo-distributed* de Hadoop. La conclusión sobre Hadoop o Spark es la reflejada en la comparativa (sección anterior) es mejor solo usarlos con un clúster. El modo *pseudo-distributed* prometía pero la realidad es que los modos de Hadoop que no sean sobre clúster (*single node* y el pseudo distribuido) solo deben utilizarse para hacer pruebas de MapReduce, antes de llevarlo al clúster.

Otras conclusiones más personales:

El trabajo ha sido todo un reto pero me ha gustado tener la oportunidad de realizarlo. En un principio rompió mis esquemas (por el tipo de proyecto en el que estaba englobado) y no tenía mucha idea de inteligencia artificial, modelos de clasificación o las redes neuronales que están detrás de todo lo expuesto en estas páginas.

Pero con el tiempo se van pillando los conceptos. No me considero un experto en el tema pero sí ha mejorado mi visión del mismo, porque aunque no he trabajado directamente con él he tenido que paralelizar el proceso de extracción de características que es el primero para la visión por computador.

El trabajo además presenta varias tecnologías que no pensaba utilizar al empezarlo y que han requerido de dedicación, investigación, lectura de las documentaciones oficiales y ejemplos para comprenderlas y poder utilizarlas. Eso me ha gustado mucho. Con un vistazo de alto nivel tenemos a Hadoop y a Spark pero si se profundiza he tratado con el sistema de ficheros de Hadoop, HDFS; con el streaming a Java, con las RDD de Spark y más.

Al irse terminando el trabajo podía predecir los resultados porque iba enterándome mejor de qué podía hacer y qué no, como queda reflejado en las otras lecciones aprendidas. En definitiva, los resultados finales en la comparativa me resultan normales y los que podía esperar teniendo en cuenta todo lo aprendido. Pero no podría haberlo predicho al principio, personalmente pensaba que Spark iba a ser la mejor indiscutiblemente.

Para finalizar aprovecho la última frase del anterior párrafo y es que Spark puede que sea la mejor fuera de las limitaciones de este trabajo y FrImCla. Me hubiera gustado poder dedicar más tiempo a esta solución pero comprendo tres cosas: que el objetivo de mi trabajo era comparar técnicas de paralelización, si lo utilizara Spark podría creer que es la mejor sin criterio alguno; que los requisitos de mi trabajo estaban claros desde la planificación y uno de ellos era mantenerme dentro de la idea de FrImCla de tener datasets pequeños (relativamente), no necesariamente *hardware* dedicado y aún así producir buenos modelos de clasificación; y finalmente entiendo que esto no acaba aquí, con un trabajo, sino que en el futuro puede tocarme utilizar todo lo que he aprendido con él y si toca paralelizar procesos en un clúster de ordenadores sé qué alternativa voy a evaluar la primera.

9. Referencias

[1] Reinhard Klette (2014). *Concise Computer Vision*. Springer. ISBN 978-1-4471-6320-6

[2] Linda G. Shapiro; George C. Stockman (2001). *Computer Vision*. Prentice Hall.
ISBN 978-0-13-030796-5

[3] Dufour, Jean-Yves, *Intelligent Video Surveillance Systems*, John Wiley Publisher (2012)

[4] SINHA, Gaurav; SHAHI, Rahul; SHANKAR, Mani. *Human Computer Interaction.* (2010) In: Emerging Trends in Engineering and Technology (ICETET), 2010 3rd International Conference on. IEEE. pp. 1-4

[5] *Autonomous Shuttle*, Texas A&M University:
https://unmanned.tamu.edu/projects/autonomous-shuttle/

[6] Soltani, A. A., Huang, H., Wu, J., Kulkarni, T. D., & Tenenbaum, J. B. *Synthesizing 3D Shapes via Modeling Multi-View Depth Maps and Silhouettes With Deep Generative Networks*. (2019-01-25) In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition pp. 1511-1519.
https://github.com/Amir-Arsalan/Synthesize3DviaDepthOrSil

[7] Turek, Fred. *Machine Vision Fundamentals, How to Make Robots See*. (June 2011) NASA Tech Briefs Magazine. 35 (6). pp. 60–62

[8] *Hardware for Machine Learning: Challenges and Opportunities* V. Sze, Y. Chen, J. Emer, A. Suleiman, Z. Zhang, Massachusetts Institute of Technology Cambridge, MA 02139

[9] Seth Colaner. *A Third Type Of Processor For VR/AR: Movidius' Myriad 2 VPU*. (2016-01-03)
www.tomshardware.com

[10] Jouppi, Norman P.; Young, Cliff; Patil, Nishant; et al. *In-Datacenter Performance Analysis of a Tensor Processing Unit*™ (PDF) (2017-06-26) Toronto, Canada.

[11] Precios Google Cloud *TPU*: https://cloud.google.com/tpu/docs/pricing

[12] Especificaciones NvidiaGeforce GTX 2080 Ti
https://www.nvidia.com/es-es/geforce/graphics-cards/rtx-2080-ti/

[13] Proyecto CLODE: https://www.unirioja.es/servicios/sgib/investigacion/clode.shtml

[14] R. Szeliski. *Computer Vision: Algorithms and Applications*. Springer, 2010

[15] Repositorio GitHub, FrImCla: https://github.com/ManuGar/FrImCla

[16] Página oficial de Python: https://www.python.org/
https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7993626

[17] D. Cournapeau. (2018) Scikit-learn: http://scikit-learn.org/stable/

[18] Intel. (2016) OpenCV: https://opencv.org/

[19] F. Chollet. (2017) Keras: https://keras.io/

[20] Fundación Python (2018) Cpickle library: https://docs.python.org/2/library/pickle.html

[21] PyImageSearch Gurus Course, Adrian Rosebrock: https://gurus.pyimagesearch.com

[22] N. Dalal and B. Triggs, *Histograms of Oriented Gradients for Human Detection*, in Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR '05) - Volume 1 - Volume 01, ser. CVPR '05. IEEE Computer Society, 2005, pp. 886–893.

[23] Lienhart, R. and Maydt, J., *An extended set of Haar-like features for rapid object detection*, (2002) ICIP02, pp. I: 900–903

[24] Página de Scholarpedia sobre SIFT (*Scale Invariant Feature Transform*) por Prof. Tony Lindeberg, KTH Royal Institute of Technology, Stockholm, Sweden http://www.scholarpedia.org/article/SIFT

[25] T. Cover and P. Hart, *Nearest Neighbor Pattern Classification*, (2006) IEEE Trans. Inf. Theory., vol. 13, no. 1, pp. 21–27

[26] C. M. Bishop, *Neural Networks for Pattern Recognition*. (1995) Oxford University Press.

[27] C. Cortes and V. Vapnik, *Support-Vector Network*s, (1995) Machine Learning, vol. 20, no. 3, pp. 273–297.

[28] C. M. Bishop, *Pattern recognition and Machine Learning.* (2006) Springer.

[29] Wolpert, David, *The Lack of A Priori Distinctions between Learning Algorithms*, (1996) Neural Computation, pp. 1341-1390.

[30] West, Jeremy; Ventura, Dan; Warnick, Sean. *Spring Research Presentation: A Theoretical Foundation for Inductive Transfer*. (2007) Brigham Young University, College of Physical and Mathematical Sciences

[31] R. S. Hunter, *Photoelectric Color-Difference Meter*, (1948) Journal of the Optical Society of America, vol. 38, no. 7, p. 661.

[32] R. M. Haralick, K. Shanmugam, and I. Dinstein, *Textural features for image classification*, (1973) IEEE Transactions on Systems, Man and Cybernetics, vol. SMC-3, no. 6, pp. 610–621.

[33] K. Simonyan and A. Zisserman, *Very Deep Convolutional Networks for LargeScale Image Recognition*, (2014) CoRR, vol. abs/1409.1556. http://arxiv.org/abs/1409.1556

[34] K. He et al., *Deep Residual Learning for Image Recognition*, (2016) in Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR'16), ser. IEEE Computer Society. IEEE, pp. 770–778.

[35] C. Szegedy et al., *Going deeper with convolutions*, (2015) in Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR'15), ser. IEEE Computer Society. IEEE, pp. 1–9

[36] F. Chollet, *Xception: Deep learning with depthwise separable convolutions*, (July 2017) in 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 1800–1807

[37] J. H. Friedman, *Greedy function approximation: A gradient boosting machine*. Ann. Statist., vol. 29, no. 5, pp. 1189–1232, 10 2001. https://doi.org/10.1214/aos/1013203451

[38] P. McCullagh and J. A. Nelder, *Generalized Linear Models*. (1989) Chapman & Hall.

[39] L. Breiman, *Random Forests*, (2001) Machine Learning, vol. 45, no. 1, pp. 5–32.

[40] Powers, David M W (2011). *Evaluation: From Precision, Recall and F-Measure to ROC, Informedness, Markedness & Correlation*, Journal of Machine Learning Technologies. **2** (1): *37–63.* http://www.flinders.edu.au/science_engineering/fms/School-CSEM/publications/tech_reps-research_artfcts/TRRA_2007.pdf

[41] Powers, David M. W. *The Problem with Kappa*. (2012) Conference of the European Chapter of the Association for Computational Linguistics (EACL2012) Joint ROBUS-UNSUP Workshop.

[42] D. Sheskin, *Handbook of Parametric and Nonparametric Statistical Procedures*.(2011) CRC Press.

[43] I. Guyon, K. Bennett, G. Cawley, H. J. Escalante, S. Escalera, T. K. Ho, N. Macià, B. Ray, M. Saeed, A. Statnikov, and E. Viegas, *AutoML challenge 2015: Design and first results*, (2015) in Proceedings of AutoML 2015@ICML.

[44] *Dataset* Kaggle Dogs and Cats: https://www.kaggle.com/c/dogs-vs-cats/data

[45] Página oficial de HDF, *What is HDF?* https://www.hdfgroup.org/solutions/hdf5/

[46] Documentación oficial de *Threading*: https://docs.python.org/dev/library/threading.html

[47] Documentación oficial de *Multiprocessing*: https://docs.python.org/dev/library/multiprocessing.html

[48] Página oficial de Apache Hadoop: https://hadoop.apache.org/

[49] Página oficial de Apache Spark: https://spark.apache.org/

[50] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica et al. *Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing* https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf

[51] Página oficial de Pandas: https://pandas.pydata.org/

[52] Página oficial de Spark MLib: https://spark.apache.org/mllib/

[53] Página oficial de Apache Mahout: https://mahout.apache.org/

[54] Página oficial de TensorFlow: https://www.tensorflow.org/

[55] Dean, Jeff. *TensorFlow: Large-scale machine learning on heterogeneous systems* (2015-11-9) http://download.tensorflow.org/paper/whitepaper2015.pdf

[56] Página sobre GIL en la wiki de Python: https://wiki.python.org/moin/GlobalInterpreterLock

[57] Página oficial de Jython: https://www.jython.org/

[58] Página sobre IronPython en la wiki de Python: https://wiki.python.org/moin/IronPython

[59] Documentación oficial sobre RLock, Python: https://docs.python.org/2.0/lib/rlock-objects.html

[60] Página oficial de lanzamientos, Hadoop: https://hadoop.apache.org/releases.html

[61] Documentación oficial de PySpark: https://spark.apache.org/docs/2.1.1/api/python/index.html

[62] Página oficial de PyPI: https://pypi.org/

[63] Página oficial de Jupyter Notebook: https://jupyter.org/

[64] Thulasiraman, K.; Swamy, M. N. S., *5.7 Acyclic Directed Graphs, Graphs: Theory and Algorithms*, (1992) John Wiley and Son, p. 118, ISBN 978-0-471-51356-8.

[65] Tutorial para comprender DAG (*Directed Acyclic Graphs*) en Apache Spark: https://data-flair.training/blogs/dag-in-apache-spark/