Escola Politécnica de Enxeñaría de Ferrol

UNIVERSIDADE DA CORUÑA

TÍTULO Paralelización y estudio de los resultados de la implementación sobre un código de entrenamiento de RN para la clasificación de galaxias

MÁSTER Informática Industrial y Robótica

ASIGNATURA Python para Ingenieros Avanzado

ESTUDIANTE Álvarez Crespo, Marta María

FECHA Mayo de 2024

Índice

Listado de figuras

Introducción

Este proyecto consiste en el análisis de los resultados de la adaptación y mejora de la eficiencia de un código de entrenamiento y validación de redes neuronales para la clasificación de galaxias mediante un conjunto de imágenes mediante su paralelización utilizando diferentes técnicas de multihilo y multiproceso.

1 Adaptación del código original

Se parte de la práctica entregada de manera conjunta en las asignaturas de Python para Ingenieros Introductorio y Visón Artificial I. En esta práctica, se realizaba un entrenamiento de diferentes redes neuronales con varias combinaciones para su estudio mediante técnicas de *Transfer Learning* y *Fine Tunning*.

El objetivo del ejercicio era encontrar la combinación de red neuronal y configuración que mejores resultados ofrecía, en cuanto a métricas, tiempo consumido y recursos computacionales, para el conjunto de imágenes dado.

Tras su corrección y revisión por parte del profesorado de ambas asignaturas, se propusieron una serie de mejoras en el código con respecto a la práctica original.

En primer lugar, se reestructuró el *main* para reducir su extensión, agrupando las tareas en funciones. De esta manera, se logra un código más pulcro y legible. Dichas funciones, se crearon en un nuevo fichero llamado *fnc*, en el que, además, se realizaron mejoras sobre el código.

En el resto de archivos, se eliminaron funciones repetidas, se dividieron funciones excesivamente largas en varias funciones individuales (en relación a las tareas que realizaba) y se reestructuró por completo el código, logrando una arquitectura más modular y escalable, adaptable a futuras implementaciones o experimentos.

Finalmente, se genera un archivo de configuración de tipo *json* (*configuracion.json*), en el que se puede decidir qué configuraciones de red neuronal probar, así como el *dataset* sobre el que realizar el experimento. Aunque la práctica se basaba en la identificación de galaxias, cualquier *dataset* de imágenes es susceptible de ser procesado, siempre que sea del formato compatible.

Este archivo permite, a la vez, elegir de qué manera ejecutar la experimentación: hilos, procesos o de manera secuencial. Este apartado es el que se desarrolló y estudió para el ejercicio propuesto en la asignatura.

Es relevante destacar que, para realizar la paralelización, se elimina del código principal la generación de gráficas por parte de *MatPlotLib* debido a que no es *threadsafe*. En este caso, no se busca la obtención de resultados por parte del código, si no buscar la implementación que mejor rendimiento muestre, por lo que se decide simplificar el código para lograr realizar el estudio con mayor facilidad.

Por otro lado, para la ejecución, se reduce el *dataset* considerablemente. Como lo que se busca es un estudio de tiempos, se considera que el aumento del número de imágenes tan solo influirá de manera proporcional en los tiempos obtenidos, aunque se comprobará que esto no es del todo cierto durante la realización del estudio.

2 Paralelización

Antes de realizar la implementación, se analiza el proceso para determinar qué sección de código paralelizar. Teniendo en cuenta la arquitectura utilizada originalmente, se decide que, para ambas aproximaciones (multihilo y sus variaciones, y multiproceso y sus variaciones), los hilos y/o procesos que se generan corresponderán a cada prueba a realizar.

Es decir, para cada red neuronal y cada configuración asociada, se generará un hilo/proceso en sí mismo. Una vez terminadas las pruebas, se continuará con la ejecución secuencial del resto del código.

2.1 Multihilo

Para la implementación de la ejecución por hilos se realizan dos aproximaciones distintas mediante el uso de dos librerías diferentes:

- *Thread*: Por un lado, se implementa desde la librería *threading* la clase *Thread*, a través de la definición de una nueva clase hija denominada ”MiHilo” en la que se definen los métodos *run()* y *get_result()* para permitir la ejecución de las funciones y la obtención de los resultados de forma paralela.
- *ThreadPoolExecutor*: Por otro lado, se implementa desde *concurrent.futures* el objeto *ThreadPoolExecutor* como *pool*, permitiendo el uso de los métodos necesarios para la ejecución del código de forma paralela utilizando hilos a través de esta librería.

2.2 Multiproceso

Para la implementación de la ejecución por multiproceso, se realizan dos aproximaciones distintas mediante el uso de dos librerías diferentes:

- *Process*: Por un lado, y análogamente al esquema implementado en los hilos, se importa desde la librería *multiprocessing* la clase *Process*, a través de la definición de una nueva clase hija denominada ”MiProceso” en la que se define el método *run()*, el cual se ejecuta cuando se inicia el proceso y se encarga de llamar al objetivo del proceso (*_target*) pasándole los argumentos (*_args*). Luego, coloca el resultado en la cola (*queue*) para que pueda ser recuperado por el proceso principal.
- *ProcessPoolExecutor*: Por otro lado, se implementa desde c*oncurrent.futures* el objeto *ProcessPoolExecutor* como *pool*, permitiendo el uso de los métodos necesarios para la ejecución del código de forma paralela utilizando hilos a través de esta librería.

3 Estudio del rendimiento

Una vez realizadas las implementaciones, se hace un estudio del rendimiento de ambos desarrollos y sus correspondientes variaciones.

Para la experimentación, se utiliza un ordenador con la siguientes características:

- *Procesador*: 12th Gen Intel(R) Core(TM) i7-12700F 2.10 GHz
- *RAM instalada*: 32,0 GB (31,8 GB usable)
- *Tipo de sistema*: Sistema operativo de 64 bits, procesador basado en x64

En cuanto al sistema operativo utilizado, cuenta con las siguientes características:

- *Edición*: Windows 11 Pro
- *Versión*: 23H2
- *Versión del sistema operativo*: 22631.3593

3.1 Medición de tiempos con *Timeit*

Para su estudio comparativo inicial, se utiliza la librería *Timeit*, que proporciona una forma sencilla de cronometrar pequeños fragmentos de código Python. En este caso se realiza el estudio utilizando el módulo *repeat*. Esta decisión se fundamenta en la idea de obtener el tiempo acumulado que le lleva cada ejecución, en vez de obtener el tiempo acumulado de n ejecuciones. De esta manera, se comprueba el rendimiento de cada una de las ejecuciones, permitiendo sacar métricas de medición de tiempos medios, máximos y mínimos, junto con sus desviaciones típicas, de configuraciones diferentes, pudiendo despreciar ralentizaciones debido a procesos realizados de manera simultánea por el sistema operativo o el uso del PC durante la experimentación.

Para la obtención de resultados se realiza un bucle que ejecuta cada configuración un total de 10 veces, midiendo el tiempo de cada una de ellas.

A pesar de que el código está preparado para la implementación y el estudio de paralelización por multiproceso, la incompatibilidad de *Keras* en Windows con procesado paralelo impidió el estudio de esta casuística en este experimento. A expensas de encontrar una librería compatible con dicho módulo, se realiza el estudio comparativo entre diferentes implementaciones de multihilo, así como de una ejecución de tipo secuencial.

Las configuraciones propuestas son las siguientes:

- *Número de redes*: Se realizará la ejecución con dos redes neuronales precargadas, en este caso: ”VGG-16” y ”MobileNet”.
- *Configuraciones*: Para cada red, se probarán un total de 24 configuraciones diferentes, compuestas por:
  - *Número de neuronas*: 1, 2 y 3 neuronas
  - *Número de capas*: 1 y 2 capas
  - *Dropouts*: 0.0 (sin dropout) y 0.1
  - *Activaciones*: *relu* y *linear*

Dando lugar a un total de 48 experimentos a paralelizar.

El experimento, por tanto, se ejecutará un total de:

- *Multihilo clase Thread*: 10 repeticiones del experimento, dando lugar a un total de 480 entrenamientos.
- *Multihilo ThreadPoolExecutor*: 10 repeticiones del experimento, variando la cantidad de *max_workers* de 2 a 16, dando lugar a un total de 6720 entrenamientos.
- *Secuencial*: 10 repeticiones del experimento, dando lugar a un total de 480 entrenamientos.

Se ejecutará el código un total de 30 veces, obteniendo los datos relativos al tiempo de cada ejecución y un estudio estadístico de cada implementación.

Los datos obtenidos se registrarán en un archivo Excel y, a su vez, se generará otro archivo que agrupa cada tipo de ejecución en cuanto a sus valores máximos, mínimos, media y desviación típica. Se obtendrán gráficas comparativas del tiempo de ejecución de cada implementación.

Se estimará que la implementación más acertada para el código propuesto, será aquella con el tiempo mínimo más corto. Esto se debe a que se asume que, en condiciones óptimas, ese proceso siempre llevaría menos tiempo.

3.2 Análisis con cProfile

Una vez realizada la medición de tiempos con *timeit*, se realiza un análisis de la ejecución del programa mediante *cProfile*. De esta manera se analiza la eficiencia del código realizado y se buscan funciones en las que realizar modificaciones en su estructura para minimizar el número de llamadas u optimizar su ejecución.

Para su estudio, se partirá de la implementación que mejores resultados aporte en el estudio mediante timeit, comprobando, más en detalle, el funcionamiento de dicha implementación.

Para ello, se almacenarán los datos obtenidos mediante *cProfile* en un archivo Excel para su posterior estudio.

4 Resultados

4.1 Timeit

Los resultados obtenidos se resumen en la siguiente figura:

	Tiempo (s)			
	mean	max	min	std
Prueba				
Clase Thread	109,113	116,592	104,562	3,90328
Secuencial	188,172	194,654	183,411	3,26434
ThreadPoolExecutor (max workers=10)	195,867	199,017	190,813	2,38316
ThreadPoolExecutor (max workers=11)	197,849	201,327	195,509	2,0307
ThreadPoolExecutor (max workers=12)	114,168	200,875	99,0508	32,2706
ThreadPoolExecutor (max workers=13)	101,088	103,028	98,4744	1,54615
ThreadPoolExecutor (max workers=14)	101,861	103,427	100,436	1,06696
ThreadPoolExecutor (max workers=15)	102,486	103,266	100,852	0,74397
ThreadPoolExecutor (max workers=16)	103,383	104,645	101,063	1,1215
ThreadPoolExecutor (max workers=2)	127,764	169,754	113,487	16,0288
ThreadPoolExecutor (max workers=3)	98,7521	110,012	94,002	4,76049
ThreadPoolExecutor (max workers=4)	95,422	102,689	93,0298	2,93673
ThreadPoolExecutor (max workers=5)	191,041	194,193	187,425	2,15457
ThreadPoolExecutor (max workers=6)	191,488	198,809	186,808	3,72229
ThreadPoolExecutor (max workers=7)	182,253	195,744	121,06	23,3089
ThreadPoolExecutor (max workers=8)	188,614	193,517	168,706	7,24436
ThreadPoolExecutor (max workers=9)	195,005	199,445	191,114	2,95355

Figura 1: Estudio estadístico obtenido

Gráficamente, se pueden visualizar los resultados para obtener la implementación que mejor rendimiento aporta, agrupando los resultados según su mínimo, máximo, media y analizando su desviación típica:

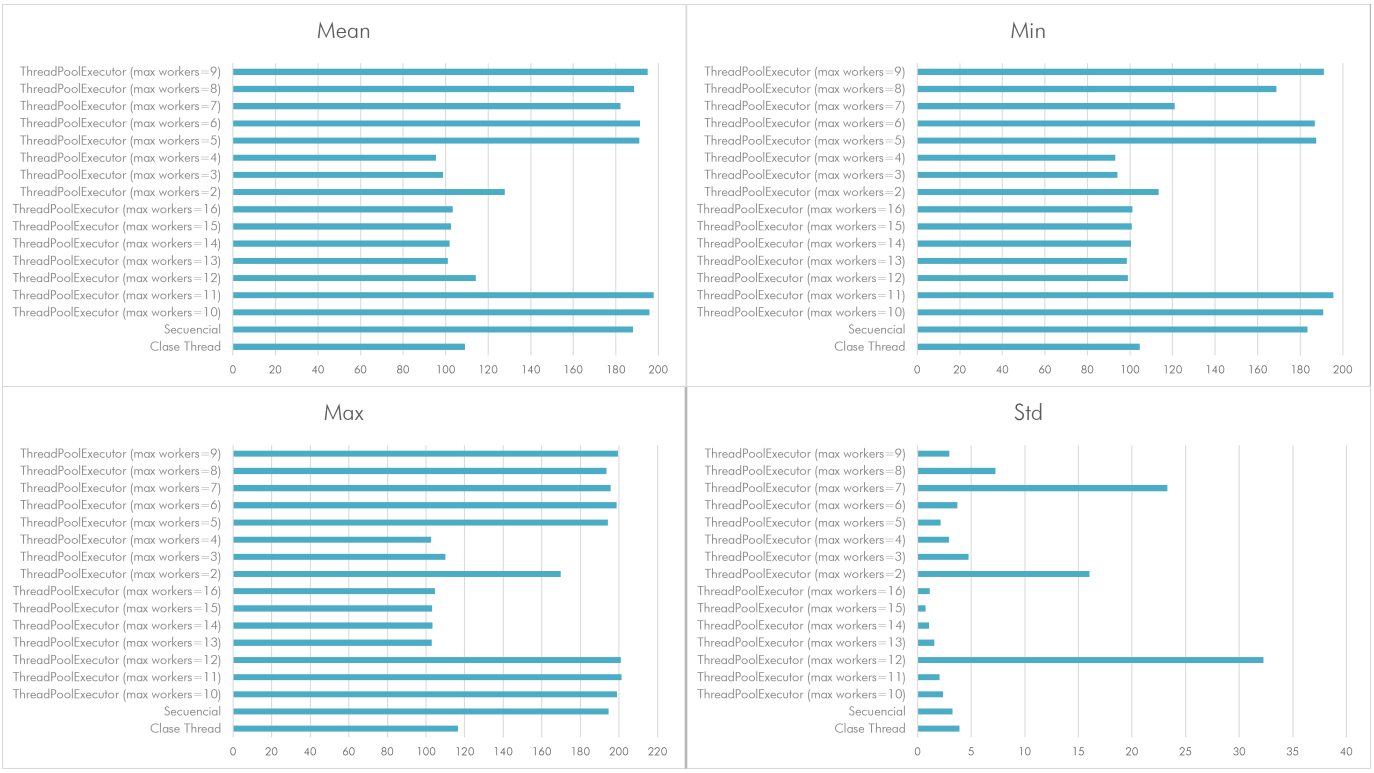

Figura 2: Visualización gráfica de los resultados

Analizando dichos resultados, se observa que, de manera general, las implementaciones que demuestran un mejor rendimiento en la ejecución que menos tiempo le llevó, se agrupan en aquellas construidas por el módulo *ThreadPoolExecutor*.

Particularmente, se observa un buen rendimiento en aquellas ejecuciones donde el número máximo de *workers* se encuentra en el intervalo [3,4] y en el intervalo [12,16], donde el rendimiento comienza a parecerse al de la clase Thread en su ejecución más rápida.

Observando el resto de métricas, se comprueba que la media de *ThreadPoolExecutor* (*max_workers*=4) es considerablemente menor que la del resto de pruebas, arrojando pruebas de que, en todas sus ejecuciones, su rendimiento fue generalmente mayor que el del resto de implementaciones. Sin ir más lejos, incluso su valor máximo, es decir, su ejecución más lenta, sigue siendo la menor de entre todas las peores ejecuciones del resto de implementaciones.

Su desviación estándar, por su parte, es mayor que la de otras ejecuciones más lentas. Esto implica que sus ejecuciones fueron menos regulares que las de las implementaciones compuestas por 15 *workers*, por ejemplo. Sin embargo, aun teniendo en cuenta este parámetro, su rendimiento sigue siendo superior a este, donde su media es sustancialmente superior a la anterior, al igual que sus máximos y mínimos.

Los motivos por los que *ThreadPoolExecutor* funciona mejor con la configuración de 4 *workers*máximos puede deberse a una gran variedad de motivos.

Por un lado, el uso de más hilos de núcleos con los que cuenta la CPU, puede afectar negativamente al rendimiento, ya que varios hilos pueden estar compitiendo por los mismos recursos de manera simultánea. En este caso, el procesador con el que se ejecutó el experimento cuenta con un total de 12 núcleos principales, 8 con un rendimiento de 2.1Ghz (hasta 4.9GHz) y 4 con un rendimiento de 1.6GHz (hasta un máximo de 3.6GHz). Esto quiere decir que, para valores más altos de 12, no debería haber diferencias significativas, tal y como se puede observar en la Figura (2).

Por otro lado, las ejecuciones que cuentan con un gran número de hilos (entre 5 y 8), aportan resultados

muy desfavorables. Esto puede deberse a que probabilísticamente se les asignen núcleos de bajo rendimiento a algunos de los hilos generados, reduciendo la eficiencia de la implementación. Además, la propia generación de hilos afecta al rendimiento, por lo que un gran número de hilos no siempre mejora la eficiencia del programa. Sin ir más lejos, también es probable que se estén distribuyendo incorrectamente los recursos debido a la finalización de unos hilos antes que otros, de forma que se asignan excesivos recursos a la generación de hilos, cuando, en la realidad, nunca hay más de n hilos ejecutándose simultáneamente, disminuyendo el rendimiento.

El resultado del experimento, teniendo en cuenta esta situación, podría variar con datasets de mayor tamaño, donde fuese posible que se ejecutasen de manera simultánea más hilos durante más tiempo de forma que, ejecuciones con un mayor número de *workers*, aportasen un rendimiento mejor.

Se observa, por otro lado, que las ejecuciones con un número de *workers*en el rango de [5,11] aportan un valor medio mayor que la ejecución secuencial, debido a una combinación de las anteriores razones expuestas.

Sin embargo, también hay que tener en cuenta que, en ocasiones, se inician subprocesos por parte del sistema operativo no controlables, que pueden afectar e interferir con los resultados de los experimentos. No obstante, esta variabilidad está siempre presente, por lo que no es despreciable. Es posible, por tanto, que alguno de estos experimentos se haya visto afectado por la aparición de subprocesos dentro del sistema, aunque el estudio de los máximos y mínimos arrojan información que alejan los resultados de esta hipótesis.

4.2 cProfile

Se realiza la ejecución del experimento mediante *cProfile* de ThreadPoolExecutor con *max_workers*=4, que es la implementación que mejores resutados arroja.

Los resultados obtenidos por *cProfile* se resumen en la Figura (3).

Nombre archivo	Línea	Nombre función	Número llamadas	Número llamadas no recursivas	Tiempo total	Tiempo acumulado
c:\Users\Marta\Documents\GitHub\MIIR2324-Python-Avanzado-Trabajo-Final\scripts\preprocesado.py	120	preprocesado_mn	23634	23634	0,006073	0,0919959
c:\Users\Marta\Documents\GitHub\MIIR2324-Python-Avanzado-Trabajo-Final\scripts\preprocesado.py	67	normalizacion_mn	23634	23634	0,032753	0,1534224
c:\Users\Marta\Documents\GitHub\MIIR2324-Python-Avanzado-Trabajo-Final\scripts\preprocesado.py	138	preprocesado_vgg	23630	23630	0,060418	0,1717234
c:\Users\Marta\Documents\GitHub\MIIR2324-Python-Avanzado-Trabajo-Final\scripts\preprocesado.py	55	originales	23489	23489	0,005373	0,0053731
c:\Users\Marta\Documents\GitHub\MIIR2324-Python-Avanzado-Trabajo-Final\scripts\deep_learning.py	408	reconstruccion_mejor_modelo_df	11459	11459	0,001724	0,0017242
c:\Users\Marta\Documents\GitHub\MIIR2324-Python-Avanzado-Trabajo-Final\scripts\fnc.py	25	division_preparacion_datos_entrada	10296	10296	0,004005	0,0056441
c:\Users\Marta\Documents\GitHub\MIIR2324-Python-Avanzado-Trabajo-Final\scripts\deep_learning.py	81	entrenar_modelo	9460	9460	0,001607	0,0016073
c:\Users\Marta\Documents\GitHub\MIIR2324-Python-Avanzado-Trabajo-Final\scripts\deep_learning.py	29	cargar_vgg	9449	9449	0,010452	0,1148791
c:\Users\Marta\Documents\GitHub\MIIR2324-Python-Avanzado-Trabajo-Final\scripts\fnc.py	403	selecciona_mejor_cnn	9159	19647	0,016173	0,0236806
c:\Users\Marta\Documents\GitHub\MIIR2324-Python-Avanzado-Trabajo-Final\scripts\deep_learning.py	253	crear_dataframe	8295	8295	0,005309	0,0064196
c:\Users\Marta\Documents\GitHub\MIIR2324-Python-Avanzado-Trabajo-Final\scripts\fnc.py	215	resultados_hilo	8100	8100	0,003493	0,0040138
c:\Users\Marta\Documents\GitHub\MIIR2324-Python-Avanzado-Trabajo-Final\scripts\fnc.py	261	hilo_tl	8075	8075	0,026308	0,0593075
c:\Users\Marta\Documents\GitHub\MIIR2324-Python-Avanzado-Trabajo-Final\scripts\deep_learning.py	211	evaluar_modelo	7698	7698	0,004422	0,0089605
c:\Users\Marta\Documents\GitHub\MIIR2324-Python-Avanzado-Trabajo-Final\scripts\deep_learning.py	298	transfer_learning	7419	7419	0,00545	0,0070106
c:\Users\Marta\Documents\GitHub\MIIR2324-Python-Avanzado-Trabajo-Final\scripts\deep_learning.py	43	crear_clasificador	7269	7269	0,001954	0,0019542
c:\Users\Marta\Documents\GitHub\MIIR2324-Python-Avanzado-Trabajo-Final\scripts\funciones_datos.py	76	cnn_predict	2178	2178	0,002984	0,0029837
c:\Users\Marta\Documents\GitHub\MIIR2324-Python-Avanzado-Trabajo-Final\scripts\funciones_datos.py	60	data_augmentation	894	894	0,003255	0,0038423
c:\Users\Marta\Documents\GitHub\MIIR2324-Python-Avanzado-Trabajo-Final\scripts\funciones_datos.py	18	cargar_dataset	870	870	0,004187	0,0117229

Figura 3: Resultados de las llamadas a las funciones custom de la implementación

- *Archivo preprocesado.py*:
  - La función *preprocesado_mn* se llama 23634 veces y tiene un tiempo acumulado de 0,0919959 segundos.
  - La función *normalizacion_mn* también se llama 23634 veces y tiene un tiempo acumulado de 0,1534224 segundos.
  - La función *preprocesado_vgg* se llama 23630 veces y tiene un tiempo acumulado de 0,1717234 segundos.
  - La función *originales* se llama 23489 veces y tiene un tiempo acumulado de 0,0053731 segundos.

- *Archivo deep_learning.py*:
    - La función *reconstruccion_mejor_modelo_df* se llama 11459 veces y tiene un tiempo acumulado de 0,0017242 segundos.
    - La función *entrenar_modelo* se llama 9460 veces y tiene un tiempo acumulado de 0,0016073 segundos.
    - La función *cargar_vgg* se llama 9449 veces y tiene un tiempo acumulado de 0,0104518 segundos.
    - La función *crear_dataframe* se llama 8295 veces y tiene un tiempo acumulado de 0,0053087 segundos.
    - La función *evaluar_modelo* se llama 7698 veces y tiene un tiempo acumulado de 0,0044223 segundos.
- *Archivo fnc.py*:
    - La función *division_preparacion_datos_entrada* se llama 10296 veces y tiene un tiempo acumulado de 0,0056441 segundos.
    - La función *selecciona_mejor_cnn* se llama 9159 veces y tiene un tiempo acumulado de 0,0236806 segundos.
    - La función *obtener_resultados* se llama 8100 veces y tiene un tiempo acumulado de 0,0040138 segundos.
    - La función *entrenar_red* se llama 8075 veces y tiene un tiempo acumulado de 0,0040138 segundos.
- *Archivo deep_learning.py*:
    - La función *evaluar_modelo* se llama 7698 veces y tiene un tiempo acumulado de 0,0089605 segundos.
    - La función *transfer_learning* se llama 7419 veces y tiene un tiempo acumulado de 0,0070106 segundos.
    - La función *crear_clasificador* se llama 7269 veces y tiene un tiempo acumulado de 0,0019542 segundos.
- *Archivo funciones_datos.py*:
    - La función *cnn_predict* se llama 2178 veces y tiene un tiempo acumulado de 0,0029837 segundos.
    - La función *data_augmentation* se llama 894 veces y tiene un tiempo acumulado de 0,0038423 segundos.
    - La función *cargar_dataset* se llama 870 veces y tiene un tiempo acumulado de 0,0117229 segundos.

En general, da la sensación de que las funciones en *preprocesado.py* y *deep_learning.py* son las más costosas en términos de tiempo de ejecución. Para optimizar el código, sería una buena aproximación comenzar analizando estas funciones y aplicar cambios y mejoras en su arquitectura, aunque no parece que haya ningún tipo de anomalía en el código per se.

Aparecen, por otro lado, una gran cantidad de llamadas a las funciones de preprocesado y normalización. Sin embargo, estas funciones son las que se encargan de procesar los *inputs* para la red. En esta implementación, los *inputs* son conjuntos grandes de imágenes. En este caso, se trabaja con un total de 450 imágenes que, multiplicadas por el número de ejecuciones, da el número de llamadas obtenido. Por tanto, la cantidad de llamadas a estas funciones depende de la complejidad de los datos de entrada, y la cantidad de modelos y configuraciones que se están utilizando.

En caso de que el procesado se realizase íntegramente de cada vez, la implementación tendría un rendimiento muy bajo. Sin embargo, estas funciones ya se encuentran optimizadas en sí mismas, ya que los conjuntos procesados se almacenan en diversos ficheros, por lo que cada llamada a la función no implica el procesado reiterativo de un conjunto de imágenes raw, si no que, en primer lugar, busca si ese conjunto de imágenes procesado ya existe en el disco duro, agilizando la ejecución.

El resto de funciones no parecen tener un rendimiento significativamente costoso ni llama la atención ningún número excesivo de llamadas.

5 Conclusiones

El uso de librerías como *Timeit* y *cProfile* aportan herramientas útiles para el análisis comparativo de diversas formas de ejecutar un programa en función del hardware/software del PC a utilizar y de la implementación propia del programa (secuencial, *threading* o procesado paralelo).

Mientras que *timeit* permite medir tiempos de ejecución de secciones de código en un programa, *cProfile* arroja información muy útil sobre el rendimiento y eficiencia de los códigos, pudiendo determinar, a simple vista, la existencia de anomalías en las ejecuciones y llamadas a funciones.

En este trabajo, se observa cómo la ejecución de varios hilos mejora significativamente el rendimiento de la ejecución del programa, a la vez que se analizan las diferentes razones por las que la generación de excesivos hilos puede, a su vez, perjudicar su rendimiento, empeorándolo con respecto a una ejecución secuencial.

Una vez realizado el estudio e implementación de la ejecución mediante threading, sería interesante comprobar la eficacia del multiproceso. Sin embargo, la librería *Keras* no es compatible con las librerías de multiproceso probadas (*multiprocess*, *multiprocessing* y *pathos.multiprocessing*), por lo que no ha sido posible su implementación real.

En caso de lograr la compatibilidad, se propone la repetición de la experimentación con los mismos parámetros para comparar tiempos de ejecución con respecto a la técnica de *threading* para, finalmente, concluir qué técnica de procesado es más eficiente para esta implementación concreta.

6 Anexo

6.1 Códigos de Programación

Todos los códigos de programación, así como las diferentes ramas de trabajo, commits de desarrollo y documentación se puede encontrar en el siguiente enlace: Github - Python Avanzado Trabajo Final (Curso 2023-2024).

Dentro de este repositorio se pueden observar varias ramas.

- *main-medicion-tiempos*: Esta rama constituye el código utilizado para la realización del estudio estadístico mostrado en este trabajo. Se agregan bucles y funciones específicas para la automatización de la fase experimental.
- *main-una-ejecucion*: Esta rama muestra un código preparado para la realización de la experimentación y medición de tiempos para un único tipo de implementación. No es la rama seguida para la realización del estudio estadístico descrito en este trabajo, sino que es la adaptación del código a un *script* funcional.
- *resto*: Existen otras ramas dentro del repositorio con diferentes pruebas de implementación de multiproceso sobre el código.

6.2 Bibliografía

[1] LIBRERÍA TIMEIT *https://docs.python.org/es/3/library/timeit.html*

[2] LIBRERÍA CPROFILE *https://docs.python.org/3/library/profile.html*