UNIVERSIDAD AUTÓNOMA DE MADRID

ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería Informática (EURO-INF®)

TRABAJO FIN DE GRADO

EJECUCIÓN DE REDES NEURONALES EN MÓVILES ANDROID CON ACELERACIÓN HARDWARE MEDIANTE KERAS Y TENSORFLOW LITE

Ángel Fragua Baeza
Tutor: Miguel Ángel García García

JUNIO 2021

EJECUCIÓN DE REDES NEURONALES EN MÓVILES ANDROID CON ACELERACIÓN HARDWARE MEDIANTE KERAS Y TENSORFLOW LITE

AUTOR: Ángel Fragua Baeza
TUTOR: Miguel Ángel García García

Video Processing and Understanding Lab
Escuela Politécnica Superior
Universidad Autónoma de Madrid
Junio de 2021

Trabajo parcialmente financiado por el Ministerio de Economía y Competitividad del Gobierno de España bajo el proyecto TEC2017-88169-R (MobiNetVideo) (2018-2020)

Resumen (castellano)

Las redes neuronales han evolucionado en todos los sentidos desde su primera aparición en 1943 a manos de Warren McCulloch. Estas primeras redes eran muy simples y tenían muchas limitaciones, pero después de poco más de una década surgió otro nuevo modelo de red neuronal llamado Perceptrón.

Este modelo trajo consigo una nueva tendencia en el diseño de redes neuronales, pero poco después se encontró con ciertas limitaciones, como la imposibilidad de resolver problemas no separables linealmente como es el caso de la puerta lógica XOR. Fue en ese momento cuando surgió lo que se consideró como el primer "Invierno de la Inteligencia Artificial". Este fue solventado con la aparición del mecanismo de la propagación hacia atrás, que permitiría entrenar redes neuronales profundas.

Desde aquel momento hasta ahora, ha habido unos cuantos "Inviernos de la Inteligencia Artificial", pero con el tiempo siempre se acaba encontrando una solución. Uno de los mayores problemas con los que se han topado las redes neuronales es su alto coste computacional, que sigue siendo ciertamente alto hoy en día.

Pero, aun así, siguen surgiendo soluciones para poder ejecutar redes neuronales con recursos mínimos y alta eficiencia. Este es el motivo por el cual este trabajo tratará de hacer un estudio completo de las redes neuronales en dispositivos móviles, aprovechando las ventajas que ofrecen las herramientas de TensorFlow Lite.

En el trabajo se partirá de unas redes preentrenadas en Keras, que pueden dividirse en dos grupos, las redes más consolidadas entre las que se verán VGG16 y ResNet50, y las enfocadas a dispositivos móviles, como es el caso de MobileNet y EfficientNet.

Las redes están entrenadas bajo una base de datos de imágenes muy conocida llamada ImageNet, con el objetivo de ser capaces de clasificar dichas imágenes u otras muy similares.

Estas redes serán convertidas a los modelos específicos de TensorFlow Lite, aplicándoles optimizadores en aquellos casos que sean posibles. Una vez se tiene el modelo en TensorFlow Lite se hará un estudio exhaustivo sobre el impacto de usar diferentes aceleraciones *hardware* y *software* al realizar la inferencia de las imágenes, mediante una serie de delegados como la GPU y NNAPI.

También se introducirán estos nuevos modelos de TensorFlow Lite en una aplicación Android real, con la consiguiente introducción de los preprocesamientos de imágenes que requiera cada modelo. Esto permitirá terminar el proceso que implicaría querer introducir una red neuronal en un dispositivo móvil desde cero.

Palabras clave

Red neuronal, red neuronal convolucional, clasificación de imágenes, ImageNet, ILSVRC, CPU, GPU, NNAPI, Keras, TensorFlow Lite, VGG16, ResNet50, EfficientNet, MobileNet, Android

Abstract (English)

The Neural Networks have evolved in every way possible, since its first appearance in 1943 by Warren McCulloch. These early networks were too simple and the had plenty of limitations, but after a decade it arises a new model named Perceptron.

This new Neural Network brought a new idea of learning, but shortly after there were some discoveries about its limitations, like the impossibility of learning the pattern of the XOR gate. It was that moment when the first “AI Winter” occurred. This problem was solved by the Backpropagation mechanism.

Since that moment until now, there has been plenty “AI Winters”, but with effort and time everyone has been solved. One of the biggest problems of the Neural Networks that nowadays is still a thing, is its high computational cost.

However, there are some options that let you use this high demanding Networks with minimal resources with high efficiency. This is the reason why this project will make a full study about Neural Networks in mobile devices, taking advantage of all the tools that TensorFlow Lite provides.

This paper will make use of some pretrained Networks obtained from Keras, that can be divided in two main groups, the more consolidated models including VGG16 and ResNet50, and the models focused on mobile devices like MobileNet and EfficientNet.

The Neural Networks are trained based on a very well-known dataset named ImageNet, which objective is to be able to classify those images or some similar ones once trained.

These Networks will be converted to its corresponding TensorFlow Lite models, applying them some optimizers in those cases that they are possible. Once obtained the TensorFlow Lite model, it will be an exhaustive study about the impact of applying different hardware and software acceleration on the image inference, using its specific delegates like the GPU and NNAPI.

Also, it will be introduced these new TensorFlow Lite models into a real Android application, with the corresponding image pre-processing required for each model. This will close the complete cycle about using Neural Networks within a mobile device from the start.

Keywords

Neural Network, Convolutional Neural Network, image classification, ImageNet, ILSVRC, CPU, GPU, NNAPI, Keras, TensorFlow Lite, VGG16, ResNet50, EfficientNet, MobileNet, Android

Agradecimientos

Me gustaría comenzar dando las gracias a todos y cada uno de los profesores que me han instruido a lo largo de mi paso por la universidad, ya que en diferente medida han conseguido avivar esa curiosidad que debería caracterizar a todos los ingenieros.

En especial, me gustaría agradecerle a mi tutor académico y de trabajo de fin de grado, Miguel Ángel García García, su apoyo constante y ayuda en los momentos de mayor necesidad.

También agradecer a toda mi familia la dedicación, el tiempo, los recursos y el grandioso esfuerzo que han llevado a cabo para formarme y educarme de la mejor forma posible, siempre dándome la libertad de realizar mis propias elecciones para afrontar la vida a mi manera.

Mención especial a todos mis compañeros de universidad, los cuales me han acompañado tanto en los buenos como en los malos momentos sirviéndonos de apoyo los unos a los otros. Aunque en especial he de agradecer a mi compañero Joaquín Jiménez López de Castro, que me proporcionó de forma voluntaria su teléfono, sin el cual no se podría haber llegado a las conclusiones definitivas de este trabajo.

ÍNDICE DE CONTENIDOS

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

ÍNDICE DE CÓDIGO

1 Introducción

La Inteligencia Artificial (IA) es una de las mayores tecnologías en auge, y en los últimos años se está introduciendo en prácticamente todos los ámbitos de nuestra vida con el ánimo de mejorarla [1].

La IA plantea infinidad de posibilidades, de entre las cuales actualmente se pueden destacar la detección de fraude en sistemas bancarios o empresas privadas [2], el reconocimiento de datos semánticos permitiendo la clasificación de textos o incluso la creación de *chatbots*, máquinas capaces de mantener conversaciones normales con una persona [3], etc.

El área en el que se centrará este trabajo es el de las redes neuronales profundas, y en específico en el de las redes convolucionales o en inglés *Convolutional Neural Network* (CNN) [4], las cuales intentan replicar el comportamiento de ciertos campos receptivos controlados por las neuronas de un cerebro biológico. Las CNN se tienden a usar para tareas relacionadas con la visión artificial, en este caso el estudio se realizará sobre redes entrenadas para clasificar una imagen entre un conjunto de posibles resultados.

La tarea de la clasificación de imágenes puede tener infinidad de aplicaciones, como por ejemplo la clasificación de plantas [5] para saber si ciertas plantas son comestibles o no, o incluso la conducción automática, tan polémica en estos momentos.

1.1 Motivación

La principal motivación será la introducción en al campo de las redes neuronales convolucionales y de las infinitas posibilidades que estas plantean. Además, gracias al enfoque propuesto en el que se estudiará su funcionamiento sobre dispositivos móviles, será posible tener un acercamiento a estas tecnologías, en concreto una primera toma de contacto con las aplicaciones en Android y sobre todo con el programa AndroidStudio. También se podrán conocer nuevas librerías para la implementación de redes neuronales como es el caso de TensorFlow y Keras, que hoy en día son utilizadas en muchos entornos profesionales por su fiabilidad y versatilidad.

1.2 Objetivos

El objetivo de este TFG consistirá principalmente en el estudio de distintas redes neuronales ya entrenadas aplicadas al problema de reconocimiento o clasificación de imágenes. Las redes neuronales empleadas en el estudio serán ejecutadas en dispositivos móviles, en concreto dispositivos Android, y se evaluará la influencia de distintas herramientas de aceleración hardware y software para intentar optimizar las redes ya existentes. Entre las redes neuronales a evaluar se encuentran algunas especialmente diseñadas para dispositivos móviles como MobileNet o EfficientNet y otros modelos consolidados como es el caso de VGG16 y ResNet50.

El desarrollo del trabajo puede ser dividido en dos secciones bien diferenciadas. La primera consistirá en la puesta en marcha del entorno de desarrollo, desde la configuración de una aplicación que permite identificar los objetos captados por la cámara del dispositivo móvil en tiempo real aplicando una red neuronal a elección, junto con una serie de opciones de aceleración, hasta la puesta a punto de ciertas herramientas de medición para lograr una cuantificación de los tiempos de ejecución y de la precisión de clasificación de un mismo conjunto de imágenes usando distintas redes neuronales y configuraciones para su posterior comparación.

1.3 Organización de la memoria

La memoria consta de una primera parte donde se realiza una exposición del **estado del arte** del trabajo, donde se explican brevemente las principales herramientas y recursos imprescindibles empleados para el desarrollo de este. Además de explicar brevemente en qué consiste cada herramienta, se hace una introducción a las redes neuronales que se van a usar, la forma de obtenerlas y el tipo de preprocesamiento que estas requieren.

A continuación, se explicarán los pasos a seguir para la puesta en marcha del entorno de **desarrollo**. Esta parte se divide en varios apartados, que recorren el proceso entero que se ha de llevar a cabo para replicar los pasos seguidos para obtener los resultados finales. Este recorrido cuenta con la explicación de la instalación de ciertos programas y librerías, la configuración inicial de repositorios y la creación, modificación u obtención de los *scripts* usados para la medición. También se explica el proceso entero de transformar un modelo de Keras a TensorFlow Lite desde su conversión hasta su introducción en una aplicación real de Android, haciendo un estudio exhaustivo de todas las optimizaciones tanto en la conversión como en la inferencia a través de diferentes configuraciones de delegados.

Por último, se expondrán las **pruebas y resultados** a los que se han llegado tras realizar todas las mediciones pertinentes para cada una de las redes neuronales con sus respectivas optimizaciones, junto con la extracción de sus principales características de tamaño, tasa de aciertos y el tiempo sobre distintos dispositivos móviles.

2 Estado del arte

2.1 ADB

Android Debug Bridge (ADB) [6] se trata de una herramienta desarrollada por Google incluida en las herramientas de desarrollo de Android SDK. ADB traducido al español sería algo similar a "un puente de depuración Android", y tal y como indica su nombre, se trata de una herramienta de depuración, implementada a través de una línea de comandos. Esta herramienta permite una conexión directa entre nuestro ordenador y un dispositivo Android, lo que se traduce en la posibilidad de realizar operaciones de subida y bajada de archivos, instalaciones y desinstalaciones, en resumen, todo tipo de consultas.

ADB se trata de una especie de programa cliente servidor, compuesto por tres componentes:

- Cliente: Se encarga de enviar los comandos y peticiones. Este se ejecuta en segundo plano desde la máquina a la que se conecta el dispositivo Android.
- Daemon(adbd): Recibe los comandos del cliente y los ejecuta en segundo plano en el dispositivo en cuestión. El *daemon* en si también se ejecuta en segundo plano en el smartphone.
- Servidor: Se encarga de gestionar la comunicación entre el cliente y el *daemon*, y al igual que el cliente se ejecuta en segundo plano en la máquina emisora.

Al intentar iniciar el cliente del ADB, el mismo programa intentaría buscar el servidor en cuestión, si este se encontrara activo se tendría acceso a todos los dispositivos conectados en modo depuración. En caso contrario, inicializaría dicho servidor buscando las posibles conexiones disponibles.

2.2 Bazel

Bazel [7] se trata de una herramienta de código libre cuya finalidad es facilitar la compilación y automatización de grandes proyectos de forma similar a como lo hacen otros programas como puede ser el caso de Make o Maeven. Para que Bazel pueda funcionar, se debe de crear un archivo BUILD que contenga el conjunto de reglas a ejecutar.

Los principales motivos por los que se usa Bazel en este trabajo son: su rapidez; unido a su reproducibilidad, haciendo que un mismo conjunto de entradas siempre devuelva una misma salida; su escalabilidad, pues permite gestionar proyectos muy grandes dependientes de múltiples lenguajes de programación, o como el caso del repositorio usado de TensorFlow, cuyos scripts están destinados para multiples arquitecturas de CPU distintas.

Bazel es la herramienta que facilitará la compilación de algunas herramientas de medición las cuales se aprovecharán para realizar las diferentes pruebas de rendimiento de las diferentes redes convolucionales en nuestro dispositivo Android.

2.3 TensorFlow Lite

TensorFlow Lite [8] consiste en un conjunto de herramientas, diseñadas para la implementación y uso de aprendizaje automático en dispositivos móviles y embebidos, es decir en dispositivos de tipo IoT (*Internet of Things*).

El uso de TensorFlow Lite puede ser dividido en dos partes:

- **Conversor** de TensorFlow Lite: Permite convertir modelos entrenados en TensorFlow o en su defecto en Keras a un formato especial conocido como FlatBuffers (cuya extensión es de tipo **.tflite**). Este formato consigue entre otras cosas reducir el tamaño del modelo y optimizarlo de tal forma que mejora su tiempo de inferencia, permitiendo una ejecución eficiente bajo dispositivos con recursos de computación o memoria reducidos. Por el momento, no todas las operaciones de TensorFlow están preparadas para ser transformadas a un modelo TenrsorFlow Lite, pero existe suficiente documentación para crearlas "a mano" en caso de que sea necesario. Además, este conversor cuenta con una serie de optimizaciones que se pueden aplicar a la hora de realizar la conversión, de las cuales se hablará en el apartado de Conversión de modelos desde Keras a TensorFlow Lite.
- **Intérprete** de TensorFlow Lite: Permite ejecutar el proceso de inferencia del modelo sobre prácticamente todo tipo de dispositivos, desde móviles Android o iOS, hasta Linux o microcontroladores varios. Este proceso de inferencia se puede llevar a cabo en infinidad de lenguajes como Java, Swift, C++, Python y Objective-C. La inferencia a su vez cuenta con una serie de opciones de aceleración tanto *hardware* como *software* las cuales se pueden aplicar haciendo uso de lo que se conoce como delegados. Los principales delegados son los de GPU, NNAPI, Hexagon y Core ML, algunos de ellos son específicos de Android como NNAPI y Hezagon, y otros de iOS como Core ML. Estos delegados serán comentados más a fondo a lo largo del documento.

La aparición de TensorFlow Lite tenía como principal objetivo optimizar el aprendizaje automático sobre aquellos dispositivos considerados "en el perímetro de la red". Los aspectos que consigue optimizar son los siguientes:

- Latencia: evitando enviar información de ida y vuelta a servidores para que lleven a cabo las inferencias u operaciones similares.
- Privacidad: al llevar a cabo la inferencia en el dispositivo, no hace falta que ningún tipo de información salga del dispositivo.
- Conectividad: evita a toda costa la necesidad de estar conectado a internet, puesto que todo el proceso se puede llevar a cabo en el mismo dispositivo.
- Consumo de energía: al no necesitar establecer conexiones con servidores externos ni hacer uso de redes externas, se puede reducir enormemente la energía requerida.

2.4 Keras

Keras [9] consiste en una biblioteca de código abierto implementada en Python cuyo objetivo es la construcción de redes neuronales de infinidad de tipos. Todo el código está diseñado e implementado de forma modular, siempre enfocado a facilitar su extensibilidad. Una de las cosas más importantes con las que cuenta es un uso para el usuario final muy amigable, como ellos mismos dicen *"Keras is an API designed for human beings, not machines"* [9], que viene a significar que Keras está diseñada para seres humanos y no para máquinas, haciendo así hincapié en la facilidad del uso de la biblioteca.

Además de contar con una gran cantidad de capas diferentes, optimizadores, funciones de activación y todo tipo de elementos para diseñar tu propia red, también cuenta con una serie de redes neuronales preentrenadas y listas para ser descargadas y usadas [10].

Estas redes ya entrenadas son las que se usarán para el proyecto, puesto que realizar el entrenamiento de una red de clasificación de imágenes tan compleja y profunda como las que se tratarán a continuación podría llevar entre 24 y 48 horas, e incluso más, aunque se contase con los componentes óptimos de hardware como son las unidades de procesamiento de tensores de Google conocidas en inglés como *Tensor Processing Unit* (TPU) [11].

No todos los modelos que se estudiarán en este trabajo funcionan de la misma manera ni requieren de un mismo preprocesamiento, aunque todos ellos están entrenados con el *dataset* reducido de ImageNet, es por ello por lo que se detallarán un poco más a fondo a continuación:

- VGG16. Se trata de una red convolucional muy profunda [12], la cual se puede obtener desde **tf.keras.applications.VGG16()**. La entrada a la red por defecto ha de ser de 224x224 y en formato RGB, además requiere de un preprocesado especial implementado en **tf.keras.applications.vgg16.preprocess_input**, el cual convertirá las imágenes de RGB a BGR y a continuación centrará respecto al origen de coordenadas cada canal de colores.
- ResNet50. Se trata de una red profunda residual [13], la cual se puede obtener desde **tf.keras.applications.ResNet50()**. La entrada a la red por defecto ha de ser de 224x224 y en formato RGB, además requiere de un preprocesado especial implementado en **tf.keras.applications.resnet.preprocess_input**, el cual convertirá las imágenes de RGB a BGR y a continuación centrará respecto al origen de coordenadas cada canal de colores, de la misma forma que se hacía en el modelo VGG16. Este tipo de preprocesamiento es bastante usado en las redes más consolidadas y es conocido como preprocesamiento *caffe*.
- MobileNet. Se trata de una red convolucional eficiente diseñada específicamente para móviles y dispositivos con recursos reducidos [14], la cual se puede obtener desde **tf.keras.applications.MobileNet()**. La entrada a la red por defecto ha de ser de 224x224 y en formato RGB, además requiere de un preprocesado especial implementado en **tf.keras.applications.mobilenet.preprocess_input**, el cual transformará el valor de cada pixel al rango comprendido entre -1 y 1.
- EfficientNet. Se trata de una red convolucional la cual es capaz de reescalar para obtener mejores *accuracy* [15], la cual cuenta con múltiples arquitecturas, pero se usará la B0 que se puede obtener desde **tf.keras.applications.EfficientNetB0()**. La entrada a la red por defecto ha de ser de 224x224 y en formato RGB, y aunque tenga definida en **tf.keras.applications.efficientnet.preprocess_input** una función de preprocesamiento, esta es innecesaria, puesto que realmente no requiere de ningún tipo de preprocesamiento especial, ya que contiene en su interior una capa de reescalado que realiza esta función. Aun así, hay que asegurarse que el formato de entrada del input sea decimal de tipo *float* y que se encuentre comprendido en el rango 0 a 255.

2.5 AndroidStudio

AndroidStudio [16] es un entorno de desarrollo integrado a los cuales comúnmente se los conoce con las siglas IDE del inglés *Integrated Development Enviroment*. AndroidStudio es el IDE oficial para el desarrollo de aplicaciones de Android creado por Google y IntelliJ, cuya interfaz está basada en el resto de las aplicaciones de IntelliJ IDEA. Entre las principales ventajas que ofrece esta aplicación y serán aprovechadas para completar este proyecto son:

- Herramientas que facilitan la visualización del código, junto con autoayudas a la hora de programar.
- Un emulador de dispositivos Android para facilitar el testeo de aplicaciones, sin la necesidad de tener un dispositivo Android físico conectado.
- Soporte de las herramientas de C++ y NDK.
- Posee incorporadas gran cantidad de las aplicaciones de Android SDK.
- Integración plena con GitHub facilitando el uso de esta herramienta.
- Un sistema de compilación especialmente flexible, y con un gran rendimiento llamado Gradle.
- Contiene un conjunto de herramientas que permiten relanzar la aplicación habiendo introducido nuevos cambios en la misma sin la necesidad de reinstalarla desde cero.

Este IDE será usado para modificar una aplicación ya existente e incorporar todos los nuevos modelos generados a lo largo de este proyecto. Así, se podrá demostrar su funcionamiento sobre una aplicación real, en vez de pruebas a través de scripts especializados en la medición del rendimiento de este tipo de modelos, los cuales tienden a dar resultados menos realistas, ya que su diseño suele ser mucho más minimalista y requerir muchos menos recursos.

2.6 ImageNet

ImageNet se trata de una base de datos conformada por algo más de 14 Millones de imágenes distintas etiquetadas con el tipo de objeto u objetos que aparecen en las mismas. Este tipo de base de datos facilitan a los investigadores y desarrolladores la labor de entrenamiento y testeo de redes neuronales profundas, pero principalmente las redes neuronales convolucionales.

Esta base de datos es excesivamente grande para realizar pruebas, y por ello se suele usar una porción menor de la misma, agilizando el proceso de aprendizaje. Esta base de datos reducida se reconoce con el nombre de *ImageNet Large Scale Visual Recognition Challange* (ILSVRC) la cual consta de 1000 clases de objetos distintos etiquetados con un identificador único, 1281157 imágenes de entrenamiento, 50000 imágenes de validación y 100000 imágenes para las pruebas. Este subconjunto de imágenes surgió en 2010 como resultado de un desafío para ver quién lograba crear una red neuronal capaz de obtener los mejores resultados, y que se fue repitiendo año tras año hasta 2017.

Para proceder a la descarga de esta base de datos reducida se debe de acceder a su página oficial [17] y crearse una cuenta. Es importante que independientemente del correo que se use para el registro, se debe facilitar un correo institucional de algún centro de estudio o investigación oficial como es el caso del de la universidad, puesto que de lo contrario no se podrá descargar la base de datos en cuestión. Una vez hecho esto, junto con la solicitud de acceso a la base de datos con carácter no comercial, se puede pasar a la descarga de la base de datos ILSVRC. En el caso del *dataset* ILSVRC para la clasificación de imágenes se ha mantenido inmutable desde 2012, por lo que se podrá acceder a la descarga de los archivos de interés desde [18]. En concreto solo se necesitarán dos archivos, las *Validation Images (all tasks)* que contienen todas las imágenes de validación y el *Development kit (Task 1 & 2)* que entre otras cosas incluye un archivo con la salida esperada de cada una de las imágenes de validación, que es lo que generalmente se reconoce con el nombre de *validation ground truth*.

3 Desarrollo

3.1 Configuración de entorno

Todo el proceso llevado a cabo para la configuración de entorno se puede dividir en dos partes, la primera consiste en preparar el teléfono Android adecuadamente lo cual se desarrollar en el Anexo A de Configuración del dispositivo Android y una preparación del computador, para configurar y descargar las librerías, aplicaciones y repositorios necesarios, lo cual se expone en el Anexo B de Configuración del computador.

3.2 Conversión de modelos desde Keras a TensorFlow Lite

Una vez se tiene el entorno configurado correctamente, se puede pasar al siguiente paso. Este, consistirá en transformar un modelo de clasificación de imagen preentrenado, en concreto con la base de datos de ImageNet, obtenido de Keras a un modelo de TensorFlow Lite. Para llevar esta transformación a cabo, existen tres posibles métodos, el primero sería desde el modelo de Keras, la segunda opción y más recomendada, sería desde un modelo de Keras previamente guardado, y la tercera y última opción sería desde una función específica.

Para los códigos que se muestren a lo largo de este apartado se hará la suposición de que siempre se cuenta con las siguientes librerías de Python importadas tal y como se muestra en el Código 3-1.

```
1.  import numpy as np
2.  import tensorflow as tf
3.  import keras as k
```

Código 3-1: Importación librerías básicas Python

La opción que se ha usado es la recomendada por las guías de TensorFlow Lite, y el resultado quedaría como se muestra en el Código 3-2. El ejemplo muestra cómo se transforma un modelo ResNet50 obtenido desde la librería de Keras en la línea 1, aunque esta línea podría cargar cualquier otro modelo de Keras. En la línea 2 y 3 se establece el nombre del directorio donde se guardará el modelo inicial sin ninguna transformación y el modelo resultante tras la transformación de forma respectiva. La línea 5 contiene el comando que guarda el modelo establecido en la línea 1 en la dirección establecida en la línea 2. Las líneas 7 y 8 convierten el modelo previamente guardado en un modelo de TensorFlow Lite, y por último las líneas 10 y 11 guardan el modelo de TensorFlow Lite como un fichero en la dirección establecida en la línea 3.

```
1.  model = k.applications.ResNet50()
2.  saved_model_dir = 'tflite_models/resnet50/'
3.  model_name = 'tflite_models/resnet50.tflite'
4.
5.  tf.saved_model.save(model, saved_model_dir)
6.
7.  converter = tf.lite.TFLiteConverter.from_saved_model(saved_model_dir)
8.  tflite_model = converter.convert()
9.
10. with open(model_name, 'wb') as f:
11.   f.write(tflite_model)
```

Código 3-2: Conversión de ResNet50 desde Keras a TensorFlow Lite

El Código 3-2 muestra la forma más simple de llevar a cabo una transformación de Keras a TensorFlow Lite, pero en este proceso también se pueden usar una serie de optimizaciones que permitirán reducir el tamaño del modelo resultante, junto con su latencia, intentando siempre minimizar las pérdidas del *accuracy*. En el momento en el que se llevó a cabo la investigación según la API oficial tan solo existían cuatro optimizadores [19]:

- DEFAULT
- OPTIMIZE_FOR_LATENCY
- OPTIMIZE_FOR_SIZE
- EXPERIMENTAL_SPARSITY

De todos estos optimizadores tan solo funcionaban los tres primeros. El optimizador llamado EXPERIMENTAL_SPARSITY no se encuentra reconocido ni siquiera dentro del código fuente. El resto de optimizadores realmente realizan las mismas optimizaciones independientemente de cual selecciones (esto se demostrará más adelante). En el Código 3-3, se muestra el mismo ejemplo que el mostrado anteriormente, pero con la introducción del código correspondiente a la optimización en la línea 8.

```
1.  model = k.applications.ResNet50()
2.  saved_model_dir = 'tflite_models/resnet50_default/'
3.  model_name = 'tflite_models/resnet50_default.tflite'
4.
5.  tf.saved_model.save(model, saved_model_dir)
6.
7.  converter = tf.lite.TFLiteConverter.from_saved_model(saved_model_dir)
8.  converter.optimizations = [tf.lite.Optimize.DEFAULT]
9.  tflite_model = converter.convert()
10.
11. with open(model_name, 'wb') as f:
12.   f.write(tflite_model)
```

Código 3-3: Conversión de ResNet50 desde Keras a TensorFlow Lite con optimizador

Los códigos discutidos hasta el momento solo permitirán transformar aquellos modelos que no tengan unas dimensiones de entrada, normalmente conocidos con el nombre de *input_shape*, de tipo estático. Esto implica que solo se podrá usar para los modelos más consolidados como es el caso de ResNet50 y VGG16. Para otros modelos con *input_shape* dinámica como es el caso de MobileNet o EfficientNet se deberá usar un planteamiento algo diferente para poder transformar dicho *input_shape* a estático, puesto que de lo contrario cuando se fuera a usar el modelo con algún delegado diferente a la CPU saldría un error similar al siguiente *"ERROR: Attempting to use a delegate that only supports static-sized tensors with a graph that has dynamic-sized tensors. Failed to apply GPU delegate."*.

Para este tipo de modelos que tienen un *input_shape* dinámico se deberán de realizar ciertas modificaciones a nuestro código original, las cuales están inspiradas en [20]. En el Código 3-4 se puede observar cómo tras guardar el modelo en la línea 3 vuelve a cargarse en la línea 5. A continuación, en la línea 9 se vuelve a cargar el modelo, pero como una función con su respectiva firma. En la siguiente línea, se establecen las dimensiones de entrada deseadas. Estas dimensiones se establecen a partir de un array donde el primer valor indica el tamaño del *batch* o dicho de otra forma el número de imágenes que se tratarán simultáneamente. El segundo y tercer valor indican las dimensiones de las imágenes en su dimensión X e Y respectivamente. Finalmente, el último valor indica el número de canales, y como por lo general las imágenes se encuentran en RGB se necesitará hacer uso de tres canales para guardar cada uno de dichos colores. En el hipotético caso de que el modelo clasificara imágenes en blanco y negro bastaría con un único canal.

Si se quisiera añadir un optimizador se puede hacer de forma idéntica a como se hizo en el Código 3-3, simplemente añadir la línea de código correspondiente entre la línea 12 y la 13 en el Código 3-4.

```
1.  model = k.applications.MobileNet()
2.  saved_model_dir = 'Salida/tflite_models/mobilenet/'
3.  model_name = 'Salida/tflite_models/mobilenet.tflite'
4.
5.  tf.saved_model.save(model, saved_model_dir)
6.
7.  model = tf.saved_model.load(saved_model_dir)
8.
9.  concrete_func = model.signatures[tf.saved_model.DEFAULT_SERVING_SIGNATURE_DEF_KEY]
10. concrete_func.inputs[0].set_shape([1, 224, 224, 3])
11.
12. converter = tf.lite.TFLiteConverter.from_concrete_functions([concrete_func])
13. tflite_model = converter.convert()
14.
15. with open(model_name, 'wb') as f:
16.   f.write(tflite_model)
```

Código 3-4: Conversión de MobileNet desde Keras a TensorFlow Lite estableciendo un *input_shape* por defecto

3.3 Medición de Accuracy

Una vez se tienen los modelos de TensorFlow Lite sería interesante conocer cuál es su *accuracy*, para comprobar si realmente la transformación se ha hecho de forma correcta. Para llevar a cabo este proceso de una forma sencilla y rápida se va a realizar desde un ordenador, ya que, si de lo contrario se hiciera desde un móvil, la clasificación sería más lenta y el *accuracy* sería el mismo. A la hora de crear un programa capaz de llevar a cabo esta tarea se va a usar como punto de partida un script de ejemplo de TensorFlow [21], que clasifica una imagen cualquiera aportando a dicho script un modelo TensroFlow Lite, la imagen en cuestión, y un archivo de etiquetado donde cada fila hace referencia a cada neurona de salida de la red pudiendo así dar el nombre de cada clase en vez de un número sin tanto significado, este último archivo es comúnmente conocido con el nombre de *label_file*.

A partir del script mencionado previamente se ha realizado una serie de modificaciones para lograr el objetivo de medición de *accuracy* de un modelo cualquiera. El primer problema que se plantea era que el preprocesamiento utilizado en el script inicial era demasiado simple y algunos de los modelos con los que se está trabajando requieren de otros preprocesamientos más específicos, como cambio de RGB a BGR. Entonces, para lograr este cambio se ha añadido un nuevo parámetro de entrada llamado **--name** que permite seleccionar entre varios tipos de preprocesamiento, uno para cada uno de los modelos que se pretende estudiar. El siguiente problema por solucionar es que, en vez de clasificar una única imagen, se desean leer todas las imágenes de un directorio para así poder hacer la media de las clasificaciones independientes de cada una de las imágenes. Para esto bastaría con cambiar el nombre del parámetro de entrada, por mantener la coherencia del código, que ahora pasará a llamarse **–directory** e iterar con un bucle por cada una de las imágenes que se encuentren dentro de dicha ruta. Ahora solo quedaría introducir un nuevo parámetro que se llamará **–ground_truth_labels** que incluya las salidas objetivo de cada una de las imágenes del directorio a clasificar, por supuesto haciendo que el índice del documento coincida con el orden de tratamiento de las imágenes dentro del directorio. Como medidas de *accuracy* se ha usado el Top-1 que indica si la clase con mayor activación coincide con la salida objetivo, y el Top-5 que indica si alguna de las 5 clases con mayor activación en la salida coincide con la salida objetivo.

Dos de los archivos necesarios para ejecutar este script aún no se ha explicado cómo conseguirlos. El primero es lo que generalmente se conoce como *label_file* que incluye las etiquetas de cada una de las neuronas de salida del modelo ordenadas decrecientemente, una por línea. En el caso de todos los modelos que se van a usar, al estar entrenados con la base de datos de ImageNet, tienen un total de 1000 clases distintas. Para conseguir este archivo lo más fácil es acceder al directorio donde está instalada la librería de Keras, que en general se puede localizar en **~/.keras/models/**, en donde se podrá localizar entre otros archivos uno llamado **imagenet_class_index.json**.

A partir de dicho archivo solamente es necesario extraer la información relevante al problema, que en este caso se encuentra en la segunda posición del array que guarda cada clave o identificador. El valor que se está rescatando del JSON es el nombre de la clase que representa la neurona de salida con dicho identificador. Para lograr esta operación se ha creado un pequeño programa en Python el cual se muestra en el Código 3-5, el cual dado un archivo de origen el cual se establece en la línea 3, y un archivo de destino que establecido en la línea 4 busca el nombre de cada clase en base a su identificador. Además, en el caso de que el nombre de la clase esté compuesto por varias palabras, se cambiará el separador por defecto que es la barra baja por un espacio en blanco.

```
1.  import json
2.
3.  file_input = 'imagenet_class_index.json'
4.  file_output = 'imagenet_labels_1000.txt'
5.
6.  output = open(file_output, 'w')
7.
8.  with open(file_input) as json_file:
9.      data = json.load(json_file)
10.     for _, values in data.items():
11.         output.write(values[1].replace('_', ' ') + '\n')
12.     output.close()
```

Código 3-5: Script de conversión del *label_file*

El segundo archivo que se necesita obtener es el comúnmente conocido como *ground truth labels* que almacena la salida esperada de cada una de las entradas. Como se van a usar las imágenes de validación de la base de datos ILSVRC también se debe de usar el *ground truth labels* que estos facilitan. Pero este archivo, en vez de tener los nombres de las clases de salida, tienen un identificador en su lugar. Para trasformar el identificador en el nombre de clase correspondiente al establecido en el archivo de *label_file* se puede usar un transformador encontrado en el repositorio de TensorFlow [22].

Partiendo del archivo oficial de *Development kit (Task 1 & 2)* y suponiendo que la ruta de acceso al mismo sea **ILSVRC2012_devkit_t12**, se podría ejecutar el Código 3-6 el cual contiene el comando a lanzar desde dentro del repositorio de TensorFlow.

```
1.  (venv) Ubuntu: ~/tensorflow$ python3
    tensorflow/lite/tools/evaluation/tasks/imagenet_image_classification/generate_vali
    dation_labels.py \
2.  --ilsvrc_devkit_dir=ILSVRC2012_devkit_t12 \
3.  --validation_labels_output=output_directory_file
```

Código 3-6: Comando de ejecución del script de generación de *ground_truth_lables*

Este transformador generaría un archivo en la ruta establecida como parámetro --**validation_labels_output** con un total de 50000 entradas que coincidirían con la clase objetivo para cada una de las 50000 imágenes de validación con las que se puede trabajar.

Como hacer pruebas con 50000 imágenes puede llevar demasiado tiempo, se ha considerado oportuno crear otro archivo a partir del anterior, pero con tan solo 5000 entradas es decir un 10% de las originales.

En este momento ya se tiene todo lo necesario para poder lanzar el script de medición de *accuracy*. Un ejemplo de ejecución se puede ver en el Anexo D de Medición de accuracy de un modelo TensorFlow Lite.

3.4 Pruebas y benchmarks en Android

Partiendo de que se tiene el repositorio de TensorFlow configurado con Bazel correctamente, tal y como se ha explicado en el apartado de configuración de entorno, se puede acceder al mismo para construir una serie de binarios que podrán ser ejecutados en cualquier dispositivo Android.

Lo primero que se debe de hacer antes de compilar ningún código es saber cuál será la arquitectura del procesador destino. Para obtener esta información se ha de conectar nuestro dispositivo móvil al ordenador y mediante ADB hacer la consulta mostrada en el Código 3-7.

```
1.  Windows: $ > adb shell getprop ro.product.cpu.abi
2.  arm64-v8a
```

Código 3-7: Comando para la obtención de la arquitectura de un móvil Android

Se puede observar que para un modelo Samsung Galaxy S10+ la arquitectura de su microprocesador es de tipo ARM de 64 bits con versión 8a, información que se puede respaldar fácilmente en alguna página web [23]. Con esta información ya se conoce qué tipo de compilación se deberá de establecer para construir correctamente los binarios objetivos.

Ahora se van a preparar dos binarios distintos ofrecidos dentro del repositorio de TensorFlow con los cuales se harán diferentes pruebas. Se comenzará con un binario preparado especialmente para la realización pruebas sobre modelos entrenados con la base de datos ILSVRC [22]. Para construir el binario se ha de lanzar el comando mostrado en el Código 3-8 que se encuentra entre las líneas 1 y 4, en el cual solo se debe establecer en la segunda línea el tipo de arquitectura sobre la que se ejecutará el binario, que como ya se ha visto se trata de un ARM de 64 bits que ejecuta Android. En caso de que el comando se ejecute de forma exitosa, las últimas tres líneas deberían mostrar algo similar, aunque posiblemente con diferentes tiempos, a lo mostrado entre las líneas 7 y 9.

```
1.  (venv) Ubuntu: ~/tensorflow$ bazel build -c opt \
2.    --config=android_arm64 \
3.    --cxxopt='--std=c++17' \
4.    //tensorflow/lite/tools/evaluation/tasks/imagenet_image_classification:run_eval
5.  …
6.  …
7.  INFO: Elapsed time: 164.899s, Critical Path: 29.17s
8.  INFO: 2465 processes: 5 internal, 2460 local.
9.  INFO: Build completed successfully, 2465 total actions
```

Código 3-8: Comando para compilar el código para la clasificación de imágenes de ILSVRC

El único problema que plantea inicialmente este binario es que no resultaría nada fácil modificarlo para introducir un preprocesamiento específico para cada modelo. Tras probar este binario en el móvil, se ha visto que para unas mismas pruebas los tiempos de inferencia varían demasiado, lo cual ha hecho necesario la búsqueda de alguna alternativa que ofrezca unos valores más consistentes.

Tras investigar por el repositorio de TensorFlow, se ha descubierto otro script capaz de realizar mediciones de tiempos sobre modelos [24], usando valores aleatorios de entrada, lo cual no es ningún problema, puesto que lo que realmente interesa es el rendimiento de los modelos con diferentes configuraciones de delegados bajo nuestro dispositivo Android. Para crear el binario específico para nuestro móvil, se debe de lanzar un comando similar al del Código 3-8, pero cambiando la ruta objetivo en la línea 3, resultando en el comando del Código 3-9.

```
1.  (venv) Ubuntu: ~/tensorflow$ bazel build -c opt \
2.    --config=android_arm64 \
3.    //tensorflow/lite/tools/benchmark:benchmark_model
4.  …
5.  …
6.  INFO: Elapsed time: 77.193s, Critical Path: 17.11s
7.  INFO: 765 processes: 2 internal, 763 local.
8.  INFO: Build completed successfully, 765 total actions
```

Código 3-9: Comando para compilar el código del benchmark de modelos TensorFlow Lite

Para cualquier compilación con Bazel el resultado de esta se encontrará dentro de un link de Linux llamado **bazel-bin**, que realmente se puede visualizar como un directorio que hace referencia a un directorio de caché de Bazel, tal y como se muestra en el Código 3-10.

```
1.  (venv) Ubuntu: ~/tensorflow$ ls -la bazel-bin
2.  lrwxrwxrwx 1 aanxel aanxel 124 May 12 14:34 bazel-bin ->
    /home/aanxel/.cache/bazel/_bazel_aanxel/a301eb60dc68372b3b932e9c041280dd/execroot/
    org_tensorflow/bazel-out/arm64-v8a-opt/bin
```

Código 3-10: Link objetivo de los resultados de compilación del repositorio de TensorFlow

Y dentro de dicho link se genera una ruta idéntica a la utilizada para la compilación en cuyo interior se localiza el binario resultante de la compilación, junto con objetos **.o** y las librerías **.a** necesarios para la compilación. En la compilación del Código 3-9 la ruta objetivo era **tensorflow/lite/tools/benchmark:benchmark_model** por lo que el binario resultante se encontrará en la ruta **bazel-bin/tensorflow/lite/tools/benchmark/benchmark_model**.

Una vez se tienen los binarios, lo recomendable sería guardarlos, para así evitar tener que recompilarlos, ya que, aunque dependa del ordenador, este proceso suele tardar un rato.

Para extraer el binario se puede usar el comando de copiar de Linux mostrado en el Código 3-11. En este comando ORIGEN sería la ruta donde se encuentra el binario que se desea copiar, y DESTINO el nuevo lugar donde se copiará el binario. El destino puede perfectamente ser un directorio montado sobre Windows para así transferir el archivo desde el WSL a Windows.

```
1.  (venv) Ubuntu: ~/tensorflow$ cp bazel-bin/ORIGEN DESTINO
```

Código 3-11: Comando para extraer el binario resultante del Link objetivo

Una vez se tiene el binario listo, basta con usar la aplicación de ADB, en concreto el comando **push** para subirlo a nuestro móvil, tal y como se muestra en el Código 3-12. El destino al que se recomienda subir todos los archivos necesarios para las pruebas sería **/data/local/tmp**. Esto se debe a que se trata de un directorio de acceso abierto dentro del teléfono, por lo que no habrá problemas de privilegios ni similares, y además se trata de un directorio cuyos archivos son temporales, por lo que el teléfono acabará eliminándolos tras no ser usados en un periodo de tiempo considerable.

```
1.  Windows: $ > adb push BINARIO /data/local/tmp
```

Código 3-12: Comando para subir cualquier archivo a un móvil Android

Es imprescindible que una vez subido el binario al móvil, sea transformado en un ejecutable, puesto que por defecto no tendrá ningún permiso. Para ello, se usará ADB junto con el comando **shell** que permite lanzar comandos de terminal Linux sobre el dispositivo Android, esto es debido a que Android usa el *kernel* de Linux. En concreto el comando que se debe usar es el mostrado en el Código 3-13.

```
1.  Windows: $ > adb shell chmod +x /data/local/tmp/BINARIO
```

Código 3-13: Comando para dar permisos de ejecución a un archivo de un móvil Android

Dependiendo del binario que se quiera ejecutar en el móvil, este posiblemente requiera de diferentes archivos para su ejecución.

En el caso del primero binario que se ha construido, el cual permitía medir el *accuracy* y el tiempo de inferencia de modelos TensorFlow Lite usando como datos de entrada las imágenes de ILSVRC, requerirá de los siguientes archivos:

- El modelo TensorFlow Lite que se desea medir. El cual se sube con el comando del Código 3-14.

```
1.  Windows: $ > adb push MODELO /data/local/tmp
```

Código 3-14: Comando para subir un *MODELO* TensorFlow Lite a un móvil Android

- Las imágenes sobre las que se realizará la clasificación. Para subir estas imágenes, primero se deberá crear una carpeta en el teléfono donde guardar todas las imágenes con el comando de la primera línea, y a continuación ya se podrán subir todas las imágenes a dicho directorio con el segundo comando. Tal y como se muestra en el Código 3-15.

```
1.  Windows: $ > adb shell mkdir /data/local/tmp/ilsvrc_images
2.  Windows: $ > adb push ILSVRC_IMAGES /data/local/tmp/ilsvrc_images
```

Código 3-15: Comando para crear una carpeta y subir contenido a la misma desde un móvil Android

- El archivo comúnmente conocido como *ground truth labels* que almacena la clase de salida objetivo para cada una de las imágenes que se van a probar. El cual se subirá con el comando del Código 3-16.

```
1.  Windows: $ > adb push GROUND_TRUTH_LABELS /data/local/tmp
```

Código 3-16: Comando para subir el *GROUND_TRUTH_LABELS* a un móvil Android

- El archivo que almacena el nombre de las etiquetas de cada una de las neuronas de salida de nuestro modelo, el cual suele ser reconocido como *label_file*. Este archivo también se ha generado en el apartado anterior, y se subirá con el comando del Código 3-17.

```
1.  Windows: $ > adb push LABEL_FILE /data/local/tmp
```

Código 3-17: Comando para subir el *LABEL_FILE* a un móvil Android

En el caso del segundo binario, el cual permite hacer *benchmarks* del tiempo de inferencia de forma más realista, requiere de un único archivo auxiliar para hacerlo funcionar. Haciendo que el proceso de testeo sea mucho más rápido y sencillo. El archivo requerido se trata del modelo TensroFlow Lite que se desea medir, y para transferirlo al móvil se debe usar el mismo comando que se usó en el Código 3-14.

Finalmente, para ejecutar cualquier binario sobre un dispositivo Android, se deberá usar ADB con el comando **shell** seguido del BINARIO que se desee ejecutar junto con sus respectivos PARAMETROS opcionales y obligatorios, en caso de que existan, como se indica en el Código 3-18.

```
1.  Windows: $ >adb shell /data/local/tmp/BINARIO     PARAMETROS
```

Código 3-18: Comando para ejecutar cualquier BINARIO en un móvil Android

Un ejemplo de ejecución del primer binario con sus respectivos parámetros obligatorios quedaría como en el Código 3-19. Para comprender a la perfección cuales son los parámetros opcionales y como configurarlos se puede acceder a su página oficial [22].

```
1.  Windows: $ > adb shell /data/local/tmp/run_eval \
2.     --model_file=/data/local/tmp/MODELO \
3.     --ground_truth_images_path=/data/local/tmp/ilsvrc_images \
4.     --ground_truth_labels=/data/local/tmp/GROUND_TRUTH_LABELS \
5.     --model_output_labels=/data/local/tmp/ LABEL_FILE
```

Código 3-19: Comando para ejecutar el binario para la clasificación de imágenes de ILSVRC desde un móvil Android

El Segundo binario es el que realmente se usará para la realización de las pruebas finales, por lo que se explicará más a fondo su funcionamiento y especialmente sus parámetros más interesantes. El ejemplo más sencillo de ejecución sería el que se muestra en el Código 3-20. En este ejemplo solo se usa un parámetro el cual es obligatorio y permite determinar cuál es la dirección del modelo que se usará para la prueba de rendimiento.

```
1.  Windows: $ > adb shell /data/local/tmp/benchmark_model \
2.     --graph=/data/local/tmp/MODELO
```

Código 3-20: Comando para ejecutar el binario que realiza benchmarks de modelos TensorFlow Lite desde un móvil Android

Pero existen multitud de parámetros opcionales que se pueden aplicar. Entre los principales parámetros que se han encontrado interesantes para el estudio del rendimiento de los modelos sobre un dispositivo Android se encuentran los siguientes:

- --warmup_runs. Permite determinar el número de ejecuciones de prueba que se harán antes de comenzar a medir tiempos en la inferencia de la clasificación. Este parámetro es muy importante, debido a que las primeras ejecuciones tienden a tener unos tiempos de inferencia muy altos, debido al coste inicial de reserva de memoria y la construcción inicial de los tensores. Por defecto el número de ejecuciones de calentamiento sería una. Como método de limitación de tiempo, el script obliga a que la prueba de calentamiento dure más de 0.5 segundos y menos de 150 segundos, por lo que el número de ejecuciones de calentamiento pueden variar respecto a las establecidas.
- --num_runs. Determina el número de pruebas que se realizarán al modelo para obtener los tiempos medios de inferencia. Por defecto el número de ejecuciones es de 50. Como característica especial, la ejecución total de esta prueba ha de tardar más de un segundo y menos de 150 segundos, por lo que en caso de que no se cumplan estos tiempos se harán más o menos ejecuciones independientemente del número establecido hasta llegar a dichos valores.

El resto de los parámetros que se van a presentar son excluyentes entre ellos, por lo que solo se podrá usar uno de ellos a la vez:

- --num_threads. Por defecto cualquier ejecución va a usar la CPU como método de ejecución de las pruebas, y el número de *threads* que use será el establecido de forma arbitraria en base al dispositivo usado. Pero si se quiere medir el impacto de usar uno o múltiples *threads* se puede modificar manualmente gracias a este parámetro.
- --use_gpu. Permite activar el delegado de GPU.
- --use_nnapi. Permite usar el delegado de NNAPI.

3.5 Introducción de modelos en aplicación real

TensorFlow ofrece una serie de aplicaciones de ejemplo para probar el funcionamiento de modelos preentrenados de TensorFlow Lite tanto para dispositivos Android como iOS e incluso Raspberry Pi [25].

En este proyecto se partirá del ejemplo para la clasificación de imágenes [26] a partir del cual ha sido posible la incorporación de los nuevos modelos que han sido generados de cero desde Keras.

Antes de comenzar con la incorporación de los nuevos modelos, viene bien conocer mínimamente cómo está planteada la aplicación. La aplicación consta de dos modos para la construcción de esta, como se puede visualizar en la Figura 3-1, permitiendo escoger entre dos librerías diferentes, cada una con un planteamiento y diseño interno diferente, pero manteniendo la misma API y una estructura similar. Las *build variable* que se encuentran dentro de la *app* se llaman **support** y **taskApi**. Para la introducción de nuevos modelos se ha escogido la librería de **support** ya que cuenta con un formato más transparente a la hora de realizar el preprocesamiento de las imágenes, facilitando en cierto grado la introducción de nuevos preprocesamientos.

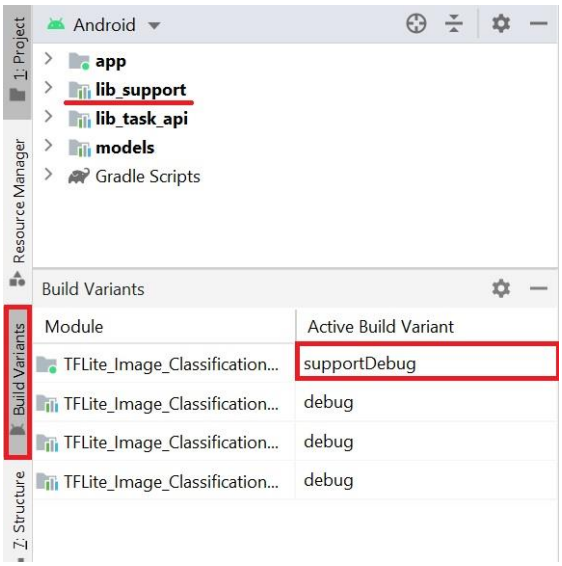

Figura 3-1: Selector de las variables de construcción de la aplicación

Una vez reconocida la librería sobre la que se va a trabajar y en la que se introducirán los cambios, se comenzará con la introducción de los cambios visuales y gráficos. Para ello, lo primero es localizar la parte de la aplicación que gestiona la lista desplegable desde la cual se ha de seleccionar el modelo deseado. La declaración de este desplegable se puede encontrar en **app/src/main/res/layout/tfe_ic_layout_bottom_sheet.xml** y si se indaga un poco más, el array de valores al que se hace referencia sería **@array/tfe_ic_models**, tal y como refleja la Figura 3-2.

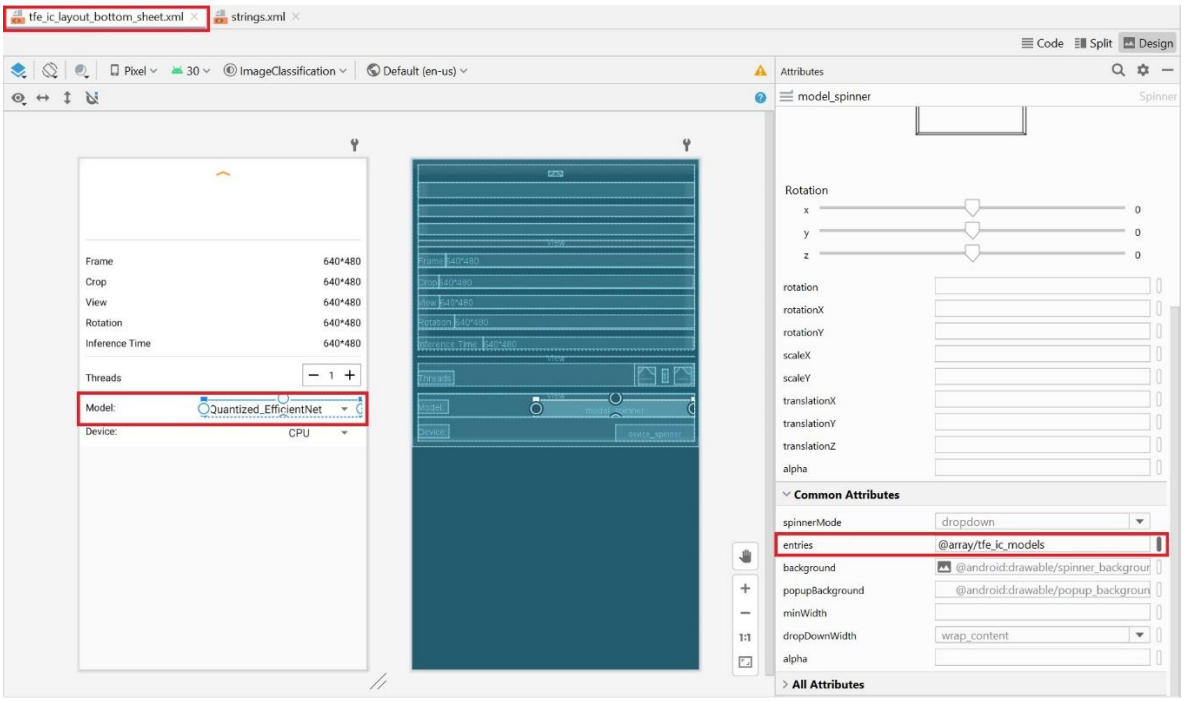

Figura 3-2: Desplegable para la selección del modelo de inferencia

Por lo general los arrays de valores y las constantes se suelen guardar dentro del directorio **app/src/main/res/values**, pero en concreto el archivo que interesa es el localizado en **app/src/main/res/values/strings.xml** en donde se puede encontrar el array de *strings* con el mismo nombre o identificador que se había localizado previamente. Ahora solo haría falta introducir en dicho array tantos nuevos valores como sean necesarios, siempre y cuando se mantenga el lenguaje de marcado establecido, en el cual se ha de preceder a la constante con la etiqueta **<item>**, y se ha de cerrar la misma con la etiqueta **</item>**. En la Figura 3-3 se puede observar cómo se han añadido cuatro nuevos modelos, de los cuales todos ellos comienzan con la palabra Keras y les sigue el nombre del modelo que representarán.

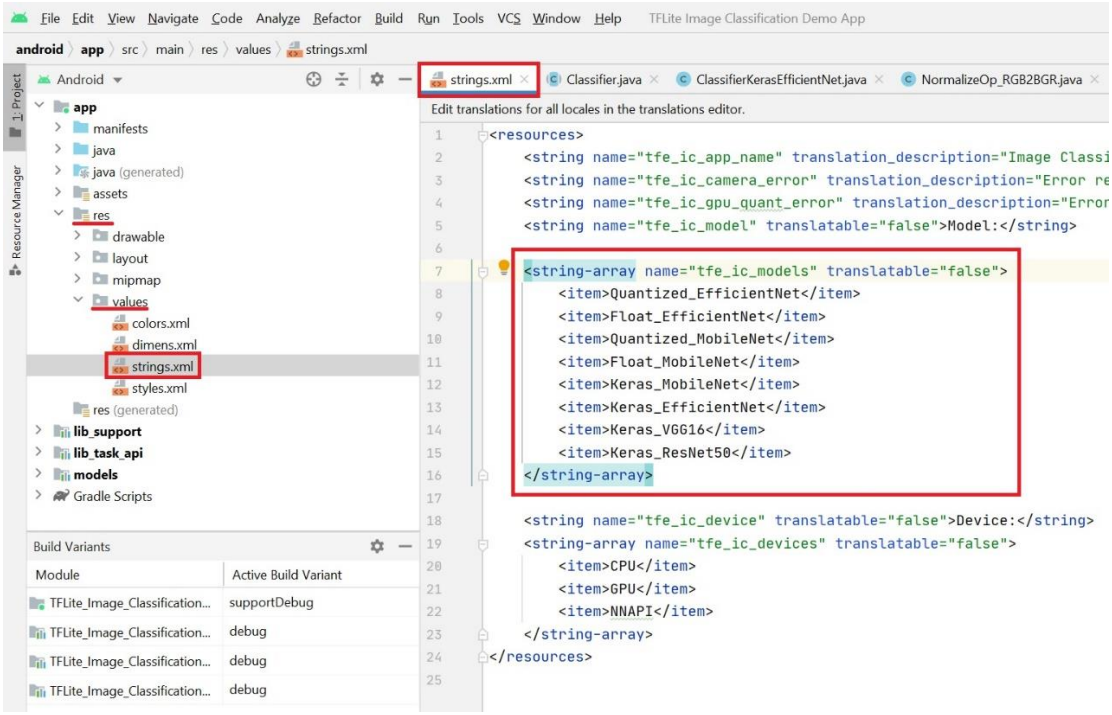

Figura 3-3: Array asociado al desplegable selector de modelos

Con este paso, ya se pueden ver en la aplicación los nuevos valores dentro del selector desplegable de modelos. Pero si se seleccionan saltará una excepción, lo cual es bastante lógico puesto que no se ha añadido ningún código de control sobre los mismos.

Para solucionar este problema se deberá crear una nueva clase que sirva como clasificador para cada uno de los nuevos modelos que se están introduciendo. Para este paso lo más sencillo sería duplicar el código de alguno ya existente y cambiarlo de tal forma que se adapte a las necesidades de cada modelo. Por poner un ejemplo, se va a partir del código de **lib_support/src/main/java/org/tensorflow/lite/examples/classification/tflite/ClassifierFloatMobileNet.java** para implementar el nuevo clasificador de EfficientNet. Los cambios a llevar a cabo en caso de que no fuera necesario añadir ningún tipo de preprocesamiento especial aparte del de normalizar los valores gracias a la media y la desviación estándar serían los siguiente, los cuales quedan reflejados en el Código 3-21:

- En la línea 12 cambiar el nombre de la clase para que coincida con el del archivo que lo contiene.
- En las líneas 15 y 17 se deberán de poner los valores que se usarán para normalizar los datos en la entrada a la red, en el caso de EfficientNet no necesita de ningún tipo de preprocesamiento, por eso en la media se pone un 0 y en la desviación un 1, ya que cualquier número menos 0 dividido entre 1 resultará en ese mismo número.
- En las líneas 23 y 25 se deberán poner los valores que se usarán para normalizar los datos a la salida de la red. Por lo general esto solo se suele hacer sobre los modelos cuantificados.
- En la línea 32 se declara el constructor de la clase, por lo que el nombre ha de coincidir con el de la clase.
- De la línea 37 a la 43 se encuentra una función que permite obtener el modelo TensorFlow Lite que se usará para la clasificación. Hasta el momento no se ha subido ningún modelo a la aplicación, por lo que para conseguirlo basta con introducirlo en el directorio **models/src/main/assets**. Una vez subido solo se ha de cambiar la línea 42 para que haga referencia al nombre del archivo en cuestión.
- De la línea 45 a la 48 se encuentra la función que permite seleccionar el archivo de etiquetado del modelo. Como las redes que se usaban por defecto se encontraban preentenadas con ImageNet al igual que todos los modelos que se están incorporando se puede aprovechar este archivo y así evitar subir el *label_file* generado en el apartado de Medición de accuracy. Hay que tener mucho cuidado con el *label_file* que se use, puesto que ciertas redes entrenadas con ImageNet tienen 1001 neuronas de salida en vez de 1000 que sería lo normal, puesto que añaden una primera neurona de clase *background* o ausencia de clasificación.
- Por supuesto, no olvidar actualizar todos los comentarios para mantener la estética y documentación del código

```
1.  package org.tensorflow.lite.examples.classification.tflite;
2.
3.  import android.app.Activity;
4.
5.  import org.tensorflow.lite.support.common.TensorOperator;
6.  import org.tensorflow.lite.support.common.ops.NormalizeOp;
7.
8.  import java.io.IOException;
9.
10.
11. /** This TensorFlowLite classifier works with the float EfficientNet model. */
12. public class ClassifierKerasEfficientNet extends Classifier {
13.
14.   /** Float EfficientNet requires additional normalization of the used input. */
15.   private static final float IMAGE_MEAN = 0.0f;
```

```
16.
17.   private static final float IMAGE_STD = 1.0f;
18.
19.   /**
20.    * Float model does not need dequantization in the post-processing. Setting mean
    and std as 0.0f
21.    * and 1.0f, repectively, to bypass the normalization.
22.    */
23.   private static final float PROBABILITY_MEAN = 0.0f;
24.
25.   private static final float PROBABILITY_STD = 1.0f;
26.
27.   /**
28.    * Initializes a {@code ClassifierKerasEfficientNet}.
29.    *
30.    * @param activity
31.    */
32.   public ClassifierKerasEfficientNet(Activity activity, Device device, int
    numThreads)
33.       throws IOException {
34.     super(activity, device, numThreads);
35.   }
36.
37.   @Override
38.   protected String getModelPath() {
39.     // you can download this file from
40.     // see build.gradle for where to obtain this file. It should be auto
41.     // downloaded into assets.
42.     return "efficientnet.tflite";
43.   }
44.
45.   @Override
46.   protected String getLabelPath() {
47.     return "labels_without_background.txt";
48.   }
49.
50.   @Override
51.   protected TensorOperator getPreprocessNormalizeOp() {
52.     return new NormalizeOp(IMAGE_MEAN, IMAGE_STD);
53.   }
54.
55.   @Override
56.   protected TensorOperator getPostprocessNormalizeOp() {
57.     return new NormalizeOp(PROBABILITY_MEAN, PROBABILITY_STD);
58.   }
59. }
```

Código 3-21: Clase controladora del modelo EfficienteNet en Java

Este mismo planteamiento se puede llevar a cabo tanto para el modelo de MobileNet como para EfficientNet, puesto que tienen un preprocesamiento sencillo o en su defecto no tienen ninguno. Para el caso de MobileNet todos los cambios serían idénticos, menos la media y desviación estándar, que se debería de establecer a 127.5, permitiendo normalizar los valores entre [-1, 1], y por supuesto el nombre del modelo deberá de coincidir con el subido al directorio de **models/src/main/assets**.

Pero esta solución no es aplicable para los modelos de VGG16 y ResNet50, ya que su preprocesamiento se basa en el establecido por el modelo caffe [27], que en resumidas cuentas consiste en realizar una transformación de RGB a BGR, o lo que es lo mismo hacer una permutación del canal rojo con el azul, seguido de un *zero-center* que consiste en centrar todos los valores respecto al origen de coordenadas.

Para añadir este nuevo preprocesamiento se ha necesitado crear una nueva clase que ha sido llamada **NormalizeOp_RGB2BGR** que extiende de **NormalizeOp**, tal y como se puede ver en el Código 3-22. El objetivo de esta nueva clase es realizar la transformación de RGB a BGR antes de la normalización típica.

Para conseguir esto, en la función de **apply**() de la línea 21, antes de llamar al método superior del padre, se transforma el input en un array unidimensional en la línea 22, sobre el cual se itera de tres en tres, intercambiando cada primer valor por el tercero tal y como se puede ver entre las líneas 24 y 27. Una vez hecho el cambio de RGB a BGR se transforma el array unidimensional de nuevo a un **TensorBuffer** de TensorFlow para finalmente llamar a la función de normalización original en la línea 38.

```
1.  package org.tensorflow.lite.examples.classification.tflite;
2.
3.  import androidx.annotation.NonNull;
4.
5.  import org.tensorflow.lite.DataType;
6.  import org.tensorflow.lite.support.common.ops.NormalizeOp;
7.  import org.tensorflow.lite.support.tensorbuffer.TensorBuffer;
8.  import org.tensorflow.lite.support.tensorbuffer.TensorBufferFloat;
9.
10. class NormalizeOp_RGB2BGR extends NormalizeOp {
11.
12.     public NormalizeOp_RGB2BGR(float mean, float stddev) {
13.         super(mean, stddev);
14.     }
15.
16.     public NormalizeOp_RGB2BGR(@NonNull float[] mean, @NonNull float[] stddev) {
17.         super(mean, stddev);
18.     }
19.
20.     @NonNull
21.     public TensorBuffer apply(@NonNull TensorBuffer input) {
22.         float[] rgb = input.getFloatArray();
23.         float[] bgr = rgb.clone();
24.         for(int i = 0; i < rgb.length; i += 3) {
25.             bgr[i] = rgb[i+2];
26.             bgr[i+2] = rgb[i];
27.         }
28.
29.         TensorBuffer output_bgr;
30.         if (input.isDynamic()) {
31.             output_bgr = TensorBufferFloat.createDynamic(DataType.FLOAT32);
32.         } else {
33.             output_bgr = TensorBufferFloat.createFixedSize(input.getShape(),
    DataType.FLOAT32);
34.         }
35.
36.         output_bgr.loadArray(bgr, input.getShape());
37.
38.         TensorBuffer output = super.apply(output_bgr);
39.
40.         return output;
41.     }
42. }
```

Código 3-22: Clase encargada del preprocesado especial de Caffe

Una vez se tiene la clase **NormalizeOp_RGB2BGR**, la cual es capaz de realizar el preprocesado de las imágenes basado en el modelo de caffe, solo quedaría introducirlo en las clases de los clasificadores deseados, que como se ha mencionado previamente serían VGG16 y ResNet50. Para mostrar un ejemplo de cómo se incorporaría este preprocesado se utilizará el clasificador de VGG16 expuesto en el Código 3-23. Los cambios a realizar serían idénticos a los del Código 3-21 pero con una diferencia, que tanto la media como la desviación de las líneas 13 y 15 respectivamente usadas para la normalización, no serán un único valor, sino un vector, esto permite que el valor almacenado en cada índice del vector se utilice únicamente para el canal con mismo índice. Es decir, el primer valor de cada vector se usará para normalizar los valores de la matriz del primer canal, y así sucesivamente. Los vectores de la media y desviación han sido obtenidos directamente del código original del

preprocesado de la API de Python [28] (línea 207). Una vez se han establecido todas las constantes que representan el modelo, solo queda modificar la forma en la que se realiza el preprocesado. De esto se encarga la función entre las líneas 48 y 51, pero bastaría con cambiar en la línea 50 la clase **NormalizeOp** por **NormalizeOp_RGB2BGR**. El resto de los parámetros no hace falta tocarlos, puesto que la nueva clase hereda de la antigua y no se han añadido nuevos parámetros.

```
1.  package org.tensorflow.lite.examples.classification.tflite;
2.
3.  import android.app.Activity;
4.  import java.io.IOException;
5.  import org.tensorflow.lite.support.common.TensorOperator;
6.  import org.tensorflow.lite.support.common.ops.NormalizeOp;
7.
8.
9.  /** This TensorFlowLite classifier works with the float VGG16 model. */
10. public class ClassifierKerasVGG16 extends Classifier {
11.
12.   /** Float VGG16 requires additional normalization of the used input. */
13.   private static final float[] IMAGE_MEAN = {103.939f, 116.779f, 123.68f};
14.
15.   private static final float[] IMAGE_STD = {1.0f, 1.0f, 1.0f};
16.
17.   /**
18.    * Float model does not need dequantization in the post-processing. Setting mean
    and std as 0.0f
19.    * and 1.0f, repectively, to bypass the normalization.
20.    */
21.   private static final float PROBABILITY_MEAN = 0.0f;
22.
23.   private static final float PROBABILITY_STD = 1.0f;
24.
25.   /**
26.    * Initializes a {@code ClassifierKerasVGG16}.
27.    *
28.    * @param activity
29.    */
30.   public ClassifierKerasVGG16(Activity activity, Device device, int numThreads)
31.       throws IOException {
32.     super(activity, device, numThreads);
33.   }
34.
35.   @Override
36.   protected String getModelPath() {
37.     // you can download this file from
38.     // see build.gradle for where to obtain this file. It should be auto
39.     // downloaded into assets.
40.     return "vgg16.tflite";
41.   }
42.
43.   @Override
44.   protected String getLabelPath() {
45.     return "labels_without_background.txt";
46.   }
47.
48.   @Override
49.   protected TensorOperator getPreprocessNormalizeOp() {
50.     return new NormalizeOp_RGB2BGR(IMAGE_MEAN, IMAGE_STD);
51.   }
52.
53.   @Override
54.   protected TensorOperator getPostprocessNormalizeOp() {
55.     return new NormalizeOp(PROBABILITY_MEAN, PROBABILITY_STD);
56.   }
57. }
```

Código 3-23: Clase controladora del modelo VGG16 con el preprocesamiento de imagen de Caffe en Java

Con esta serie de pasos se puede completar el proceso de codificar todos los clasificadores necesarios para incorporar los nuevos modelos a ser introducidos. Ahora solo quedaría vincular cada clase que representa al clasificador de cada modelo con su respectiva opción visual, que se han añadido previamente como nuevos valores dentro del selector desplegable de modelos. Para conseguir esto, se debe de acceder a la clase llamada **Classifier** ubicada en **lib_support/src/main/java/org/tensorflow/lite/examples/classification/tflite/Classifier.java**, la cual es la encargada de controlar y gestionar el proceso de selección del clasificador correcto.

Dentro de esta clase lo primero que se debe de añadir es una referencia dentro de la enumeración de modelos llamada **Model** con el mismo nombre, pero en mayúsculas, que se estableció como valor para el array que se muestra en el desplegable configurado en el archivo **app/src/main/res/values/strings.xml**, como se plasma en la Figura 3-4.

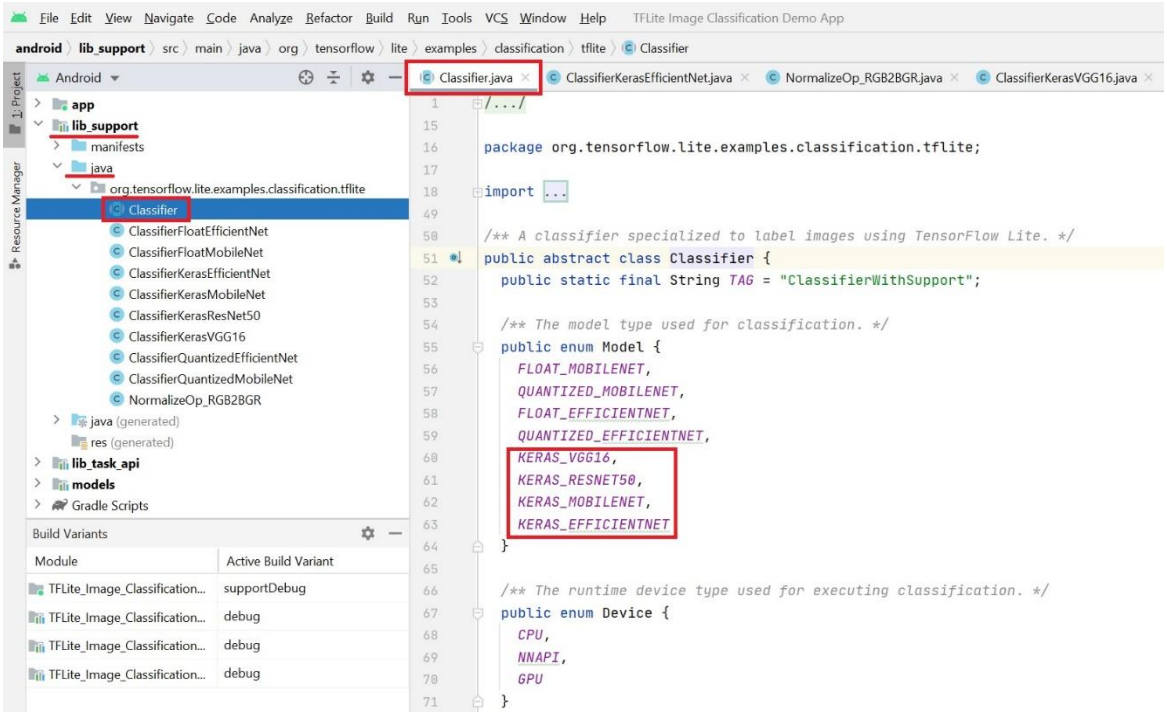

Figura 3-4: Enumeración de las clases que representan a cada modelo

Una vez añadida la referencia a los nombres de los modelos establecidos en la parte visual de la aplicación, se debe controlar que, al crear un clasificador, realmente se selecciona la clase del modelo correspondiente. Y para ello se debe localizar el método **create()** en donde se deberá añadir una nueva condición por cada uno de los nuevos modelos introducidos, en donde se comparará el nombre del modelo seleccionado en el desplegable, recibido como argumento en la función, con la lista de modelos disponibles, lanzando la clase del clasificador que corresponda o en caso de que no coincida con ninguno una excepción. El resultado de este cambio se puede visualizar en la Figura 3-5.

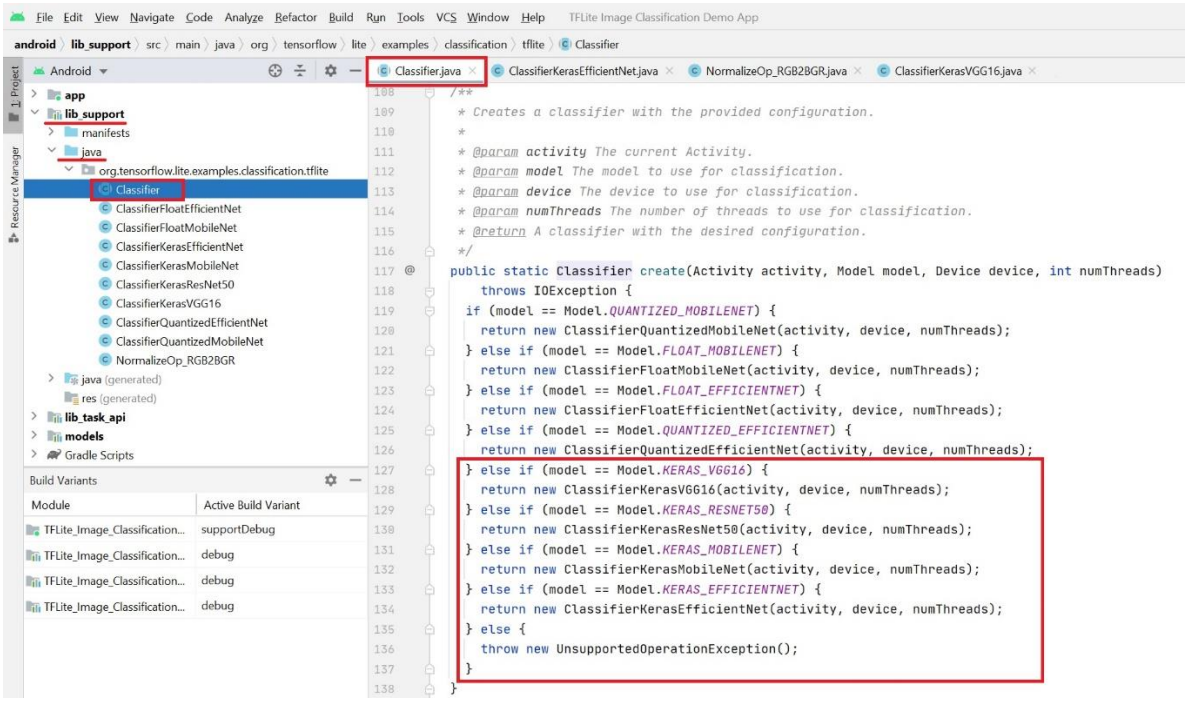

Figura 3-5: Control del modelo que clasificará las imágenes

Una vez han sido aplicados todos los cambios mencionados para cada uno de los modelos que se desee introducir solo haría falta construir el proyecto y correrlo en el dispositivo virtual, o en el móvil personal. Para correrlo en el teléfono hay dos opciones:

- La primera opción y más ágil, es tener el móvil conectado al ordenador y darle al botón de correr aplicación teniendo seleccionado el dispositivo Android en cuestión. Esto se puede hacer con los botones y desplegables marcados en la Figura 3-6.

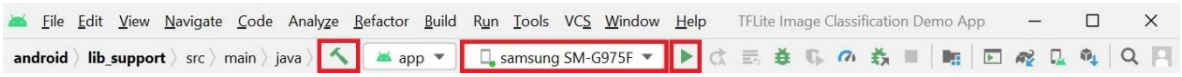

Figura 3-6: Construcción de aplicación y selección del dispositivo de lanzamiento

- La segunda opción es algo más tediosa, pero permitirá crear y almacenar un *apk* o instalador de la aplicación, de forma que será más fácil compartirlo para permitir que otros usuarios puedan tener acceso a la misma y puedan probarla sin la necesidad de tener instalado AndroidStudio ni el código fuente. Los pasos a seguir serían, acceder al apartado de *Build*, a continuación, entrar en *Build Bundle(s) / APK(s)* y finalmente seleccionar la opción de *Build APK(s)*. Gráficamente podemos ver este proceso en la Figura 3-7. Esto generará el instalador en el directorio **app\build\outputs\apk\support\debug**. Ese archivo se podrá pasar a cualquier dispositivo Android y podrá ser fácilmente instalado y usado.

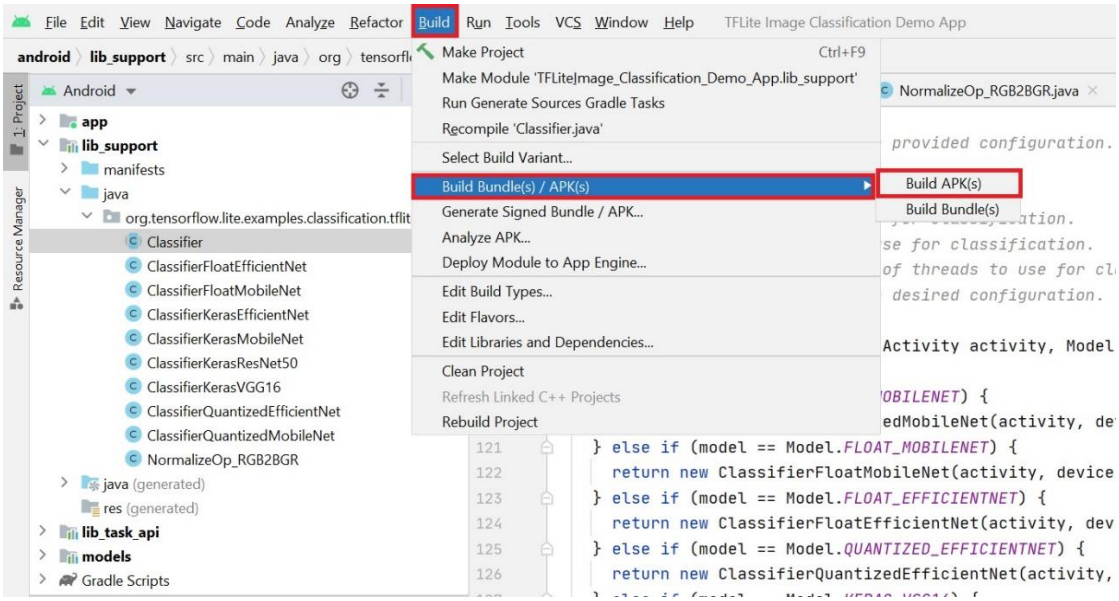

Figura 3-7: Construcción de una APK a partir del código fuente

4 Pruebas y resultados

Todas las pruebas se van a llevar a cabo sobre cuatro modelos distintos. VGG16 y Resnet50 son dos de los modelos más consolidados en el mundo de la clasificación de imágenes y a lo largo de los años han ganado múltiples competiciones tanto de identificación de imágenes, como de detección de objetos. La diferencia entre la identificación de objetos y la detección de objetos consiste en que el primer tipo dada una imagen devuelve el conjunto de probabilidades de que dicha imagen pertenezca a una de las clases memorizadas durante el entrenamiento, mientras que la detección de objeto se centra no solo en detectar el tipo de objeto, sino también en localizarlo dentro de la imagen, y en el caso de que hubiera múltiples objetos tratar de identificarlos todos.

En el caso de la detección de objetos se suelen recuadrar los objetos detectados, aunque hay alguna red más moderna capaz de marcar la forma o *shape* de los objetos detectados, lo cual es sin duda alguna absolutamente sorprendente. Para el modelo VGG hay otras variantes como la VGG19, y lo mismo sucede con ResNet, como por ejemplo ResNet101, ResNet152, ResNet50V2, etc. pero se ha considerado más interesante estudiar una mayor diversidad de modelos en vez de centrarse en las múltiples versiones con las que cuenta cada modelo.

Estás redes serán comparadas entre sí y con los modelos de MobileNet y EfficientNetB0, otras dos redes neuronales profundas, pero diseñadas por y para ser usadas en dispositivos móviles o similares, cuyos recursos son altamente limitados o reducidos. Igual que para los modelos anteriores, existen infinidad de versiones para cada modelo, por ejemplo, para MobileNet existe otra versión llamada MobileNetV2 e igual sucede con EfficientNet cuyos diseños de arquitectura van desde EfficientNetB0 a EfficientNetB7.

Gracias a las pruebas que se mostrarán a continuación se podrán graficar los resultados para sacar unas conclusiones de las comparaciones.

4.1 Tamaño de los modelos

Una de las primeras características que se pueden usar para comparar los modelos sería la del tamaño, valores que se pueden obtener de forma muy sencilla, con una simple inspección de las propiedades del archivo generado.

De partida se observa que en general los modelos más consolidados tienen un tamaño mucho mayor que el de los modelos destinados a dispositivos móviles. Esto es lo que se esperaba, puesto que uno de los principales objetivos de las redes neuronales destinadas a móviles es que su tamaño sea el menor posible, a la par que conseguir el máximo *accuracy* con la menor cantidad posible de recursos.

Cabe mencionar que VGG16 es la que posee un mayor tamaño y con mucha diferencia, de hecho, es aproximadamente cinco veces mayor que ResNet50 la otra red consolidada que se está evaluando. Y si de nuevo se enfrenta el tamaño del modelo ResNet50 con el de cualquiera de los dos modelos destinados a móviles, se aprecia que su tamaño es de nuevo aproximadamente cinco veces mayor. Lo cual significa que VGG16 ocupa unas veinticinco veces más que un modelo destinado al móvil.

Si hubiese que elegir un modelo cualquiera para un dispositivo con una gran limitación de memoria, se debería de escoger el modelo de MobileNet que, sin duda hoy en día, es posiblemente el modelo más compacto y optimizado para este tipo de situaciones.

Una vez convertidos los modelos de Keras a TensorFlow Lite tal y como se explicó en el apartado de Conversión de modelos desde Keras a TensorFlow Lite se puede ver en la Figura 4-1 que para todas y cada una de las redes se ha conseguido reducir, aunque sea mínimamente, su tamaño respecto al de partida. Esto tampoco es una sorpresa, puesto que es uno de los factores que pretendía optimizar TensorFlow Lite, y al parecer lo ha logrado en todos los casos.

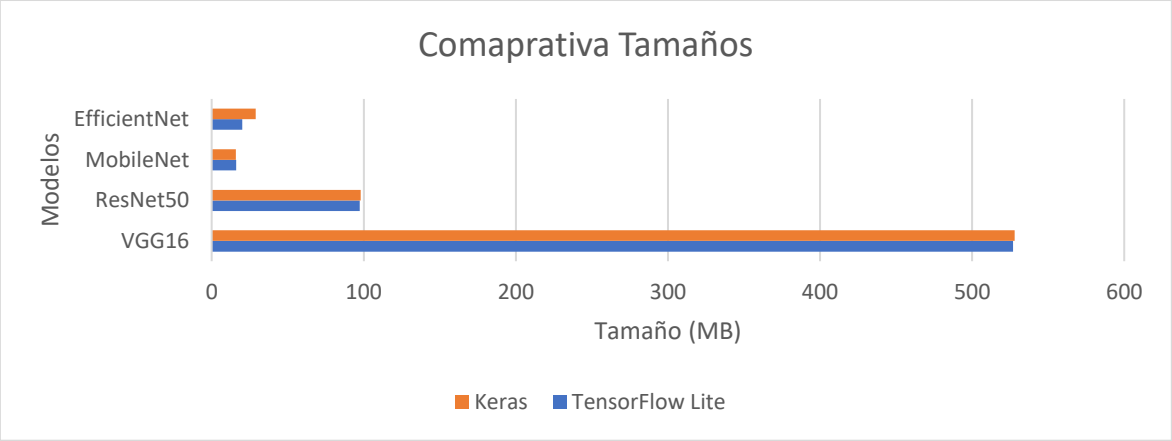

Figura 4-1: Comparativa del tamaño (MB) de los modelos

En el mismo apartado donde se explica cómo usar el Conversor de TensorFlow Lite, se comenta la existencia de una serie de opciones de optimización que podrían ser aplicadas a la hora de realizar la conversión del modelo. Estas optimizaciones no se pueden aplicar a cualquier modelo, de hecho, solo han funcionado para los modelos convencionales, y aunque se listaban cuatro posibles optimizaciones solo tres de ellas realmente se podían usar, puesto que una de ellas ni siquiera estaba declarada en el código fuente.

De las tres opciones de optimización restantes, en realidad todas realizan las mismas optimizaciones independientemente de cual se seleccione. Esto se irá demostrando a lo largo de este apartado, pero se puede ver en la Figura 4-2 como los tres modelos optimizados de VGG16, cada uno convertido con un optimizador diferente dentro de los existentes, resultan en un tamaño idéntico de 131MB, lo cual parece ofrecer claros indicios de que los tres modelos optimizados son idénticos.

Es muy importante destacar que la optimización que ofrece el conversor de TensorFlow Lite, aunque no se pueda aplicar a todos los modelos, a aquellos a los que sí, les ofrece una gran reducción de su tamaño de en torno a un 75% respecto al tamaño que tendrían en el caso de que no se hubiese aplicado ningún tipo de optimizador.

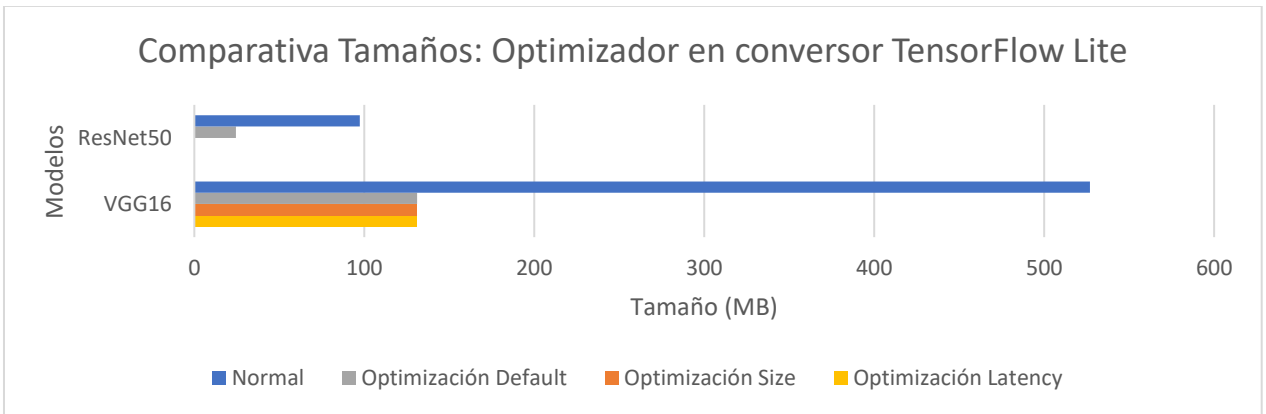

Figura 4-2: Comparativa del tamaño (MB) de modelos con optimizaciones

En la Tabla 4-1 se muestran los valores de los tamaños antes y después de hacer la conversión de los modelos de Keras a TensorFlow Lite tanto con optimizadores como sin ellos. De ahora en adelante los modelos que hagan uso de optimizadores irán acompañados de un paréntesis que indica el tipo de optimizador que se está usando.

Modelos	TensorFlow Lite Size (MB)	Keras Size (MB)
VGG16	527	528
VGG16 (Default)	131	-
VGG16 (Size)	131	-
VGG16 (Latency)	131	-
ResNet50	97,4	98
ResNet50 (Default)	24,4	-
MobileNet	16,1	16
EfficientNet	20,2	29

Tabla 4-1: Comparativa del tamaño (MB) de los modelos

4.2 Accuracy de los modelos

En este apartado se pretende comparar el rendimiento de las redes neuronales, a través del parámetro de *accuracy*, el cual mide la tasa de aciertos de la red. En concreto para la medición del *accuracy* se usarán el Top-1 que mide el número de veces que la red otorga la mayor activación a la clase objetivo, pero también se usará el Top-5 que determina si alguna de las cinco neuronas de salida con mayor activación coincide con la salida esperada.

Todas las mediciones de *accuracy* de los modelos en Keras han sido obtenidas de la API oficial de Keras [10], a excepción del modelo EfficientNet que ha sido obtenido directamente de su *paper* oficial [14].

Haciendo un primer análisis rápido, se puede observar como el modelo de EfficientNet tiene el mejor Top-1 y Top-5 *accuracy* de todos los modelos estudiados antes de transformarlos a TensorFlow Lite. Pero realmente no existe una diferencia excesivamente notable entre unos modelos y otros, ya que las tasas de acierto varían como máximo entre unos modelos y otros no más de un 5%.

Tras realizar la conversión de los modelos a TensorFlow Lite, se puede medir su *accuracy* gracias al script del Anexo D Medición de accuracy de un modelo TensorFlow Lite y al contenido explicado en el punto 3.3 de Medición de Accuracy. En el caso de este proyecto, por temas de recursos y tiempo el *accuracy* de los modelos en TensorFlow Lite se ha hecho únicamente sobre las 5000 primeras imágenes de ILSVRC, en vez de las 50000 imágenes de validación con las que cuentan, por lo que la tasa obtenida puede que no sea perfecta, pero permitirá tener una aproximación bastante acertada.

Una vez se tienen los valores de *accuracy* para los modelos de TensorFlow Lite, tal y como se observa en la Figura 4-3 en todas las redes neuronales a excepción de MobileNet desciende su Top-1 entre un 5% y un 6%, mientras que el Top-5 desciende algo menos entorno al 3% o 4%.

El caso de MobileNet es muy curioso, ya que el Top-1 en vez de disminuir aumenta mínimamente, lo cual puede haber sucedido perfectamente por lo explicado previamente acerca de que las mediciones de las tasas de acierto no se han hecho sobre el conjunto total de validación, sino sobre el primer 10% del mismo. El Top-5 de MobileNet también decrece muy poco, de hecho, es la tasa que menos se reduce entre todos los modelos, la cual ronda en torno a un 0,5%.

El hecho de que MobileNet no sufra prácticamente cambios al ser convertida a un modelo de TensorFlow Lite es probable que se deba a que ya se encuentra altamente optimizada de base, debido a que desde un inicio está pensada para dispositivos con recursos reducidos.

Tras realizar las conversiones a TensorFlow Lite se puede observar en la Figura 4-3 como el modelo que mantiene una mejor tasa de aciertos tanto en el Top-1 como en el Top-5 es el de EfficientNet, aunque de nuevo tanto ResNet50 como MobileNet tienen unos resultados muy similares, al igual que sucedía antes de la conversión.

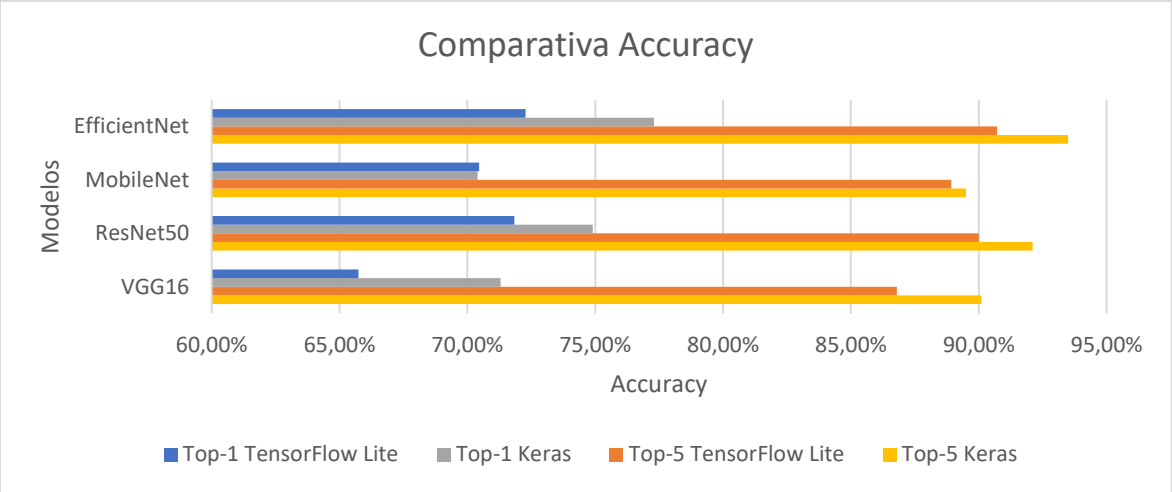

Figura 4-3: Comparativa de la tasa de aciertos de los modelos

Si se aplica una optimización a los modelos convencionales a la hora de realizar la conversión a TensorFlow Lite les sucede una cosa muy similar a lo ocurrido con MobileNet en la explicación anterior. Como se observa en la Figura 4-4 Para Top-1 ambos mejoran mínimamente alrededor de 0,05%, pero en el Top-5 empeoran entorno a un 0,3% lo cual es despreciable teniendo en cuenta que la reducción de tamaño era del 75%. El motivo por el cual es posible que el Top-1 mejore es debido a que la medición se ha hecho sobre una porción del *dataset* de validación, por lo que los resultados pueden estar mínimamente desviados.

En cuanto a los modelos enfocados a los móviles, se ve por qué en el apartado anterior se comentaba que era posible aplicarles optimización en la conversión, ya que el Top-1 y el Top-5 respectivamente para MobileNet es de 11,78% y 28,28%, y para EfficientNet es de 2,68% y de 6,76%. Lo cual deja más que claro que la optimización no ha funcionado correctamente puesto que el modelo deja de clasificar tal y como lo hacía originalmente. Es por ello por lo que no se tendrán en cuenta estos dos modelos a partir de ahora para aplicarles ningún tipo de optimización durante la conversión.

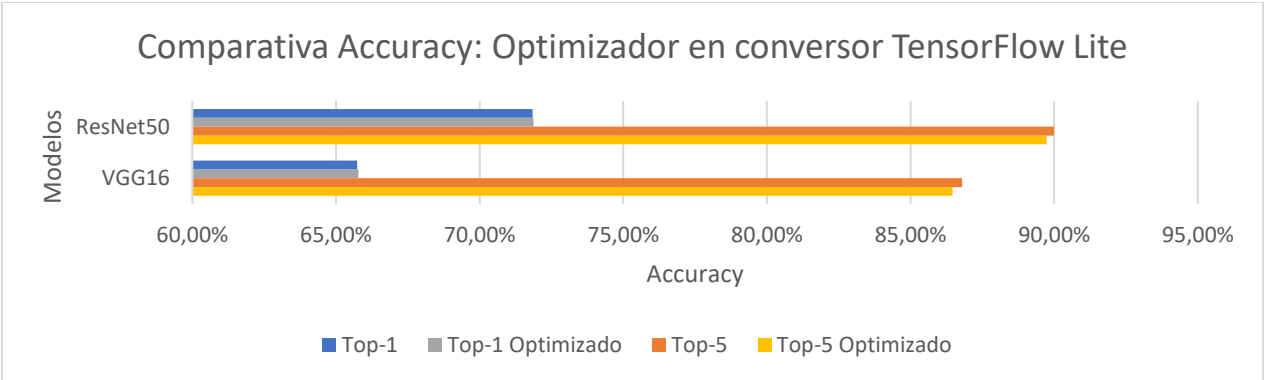

Figura 4-4: Comparativa de la tasa de aciertos de modelos con optimizadores

La Tabla 4-2 muestra los valores de *accuracy*, tanto el Top-1 como el Top-5, antes y después de hacer la conversión de los modelos de Keras a TensorFlow Lite, por supuesto haciendo uso de los optimizadores en aquellos casos que sean posible. Cuanto más se avanza con la investigación, los valores cada vez van dejando más claro que todos los optimizadores aplican las mismas acciones.

	TensorFlow Lite		**Keras**	
Modelos	**Top-1**	**Top-5**	**Top-1**	**Top-5**
VGG16	65,74%	86,80%	71,30%	90,10%
VGG16 (Default)	65,78%	86,46%	-	-
VGG16 (Size)	65,78%	86,46%	-	-
VGG16 (Latency)	65,78%	86,46%	-	-
ResNet50	71,84%	90,00%	74,90%	92,10%
ResNet50 (Default)	71,88%	89,74%	-	-
MobileNet	70,46%	88,92%	70,40%	89,50%
MobileNet (Default)	11,78%	28,18%	-	-
EfficientNet	72,28%	90,72%	77,30%	93,50%
EfficientNet (Default)	2,68%	6,76%	-	-

Tabla 4-2: Comparativa de la tasa de aciertos de los modelos

4.3 Tiempos de los modelos

Para llevar a cabo la medición de tiempos se ha llevado a cabo el proceso explicado en el apartado de Pruebas y *benchmarks* en Android, en concreto empleando el siguiente *benchmark* [24]. Este *benchmark* es una de las herramientas oficiales que se encuentran en el repositorio público de GitHub dirigido por TensorFlow.

Como el dispositivo con el que se contaba en un inicio era un Android, en concreto un Samsung Galaxy S10+, tan solo se han podido hacer pruebas de rendimiento sobre los delegados de GPU y NNAPI, y por supuesto medir el impacto de usar uno o múltiples hilos para la ejecución de las inferencias.

Debido a ciertos problemas con resultados obtenidos con el Samsung Galaxy S10+, los cuales fueron poco concluyentes y podrían haber llevado a error o una explicación incorrecta, se decidió conseguir otro dispositivo Android con un procesador distinto y repetir las pruebas. Los problemas encontrados se discutirán en el apartado específico para dicho dispositivo. El nuevo dispositivo con el que se repitieron las pruebas fue un OnePlus 8, otro teléfono de gama alta, con unas propiedades similares al usado para las pruebas iniciales.

4.3.1 Samsung Galaxy S10+

El Samsung Galaxy S10+ cuenta con un microprocesador llamado **Exynos 9820** [23], el cual está diseñado directamente por Samsung. Su arquitectura interna es un ARMv8.2 de 64 bits, gestionado mediante el sistema ARM big.LITTLE [29] el cual permite tener núcleos menos potentes pero mucho más eficientes (los cuales se considerarían LITTLE) los cuales se usarían para ahorrar energía, y se intercambiarían según la demanda y necesidad del dispositivo con otros núcleos mucho más potentes pero a cambio con un consumo energético mucho mayor (los cuales se considerarían BIG).Este procesador cuenta con 8 *cores* de los cuales 2 son **Mongoose 4** a con una frecuencia de 2,9 GHz considerados como BIG *cores*, 2 son **Cortex-A75** a una frecuencia de 2,8 GHz considerados como MIDDLE *cores* y 4 son

Cortex-A55 funcionando a una frecuencia de 1,95 GHz, los cuales tienen la menor frecuencia y con ello el menor consumo considerándose los LITTLE *cores*.

Al ser este el primer dispositivo sobre el que se hacen las pruebas, estas se han llevado a cabo de una forma muy exhaustiva. Para cada prueba se hacen un grupo de al menos 10 inferencias de *warmup* ya que las primeras suelen tardar bastante más al tener que reservar y alojar memoria para los tensores, una vez terminado el *warmup* el tiempo medio de la inferencia se obtiene de la ejecución de 5000 inferencias. A su vez, cada una de estas pruebas se repite un total de 5 veces para intentar tener unos valores lo más realistas posibles y con la menor desviación típica posible.

Cabe resaltar que todas y cada una de las pruebas se han hecho con el teléfono en modo avión, y con el wifi, datos móviles, bluetooth y el resto de los posibles módulos de conectividad desactivados, para así intentar reducir el número de procesos activos, creando un entorno lo más limpio posible para que los tiempos resultantes sean siempre lo más similares posibles.

Lo primero que se va a estudiar es la Figura 4-5 que cuenta con los resultados de las mediciones de tiempo para un mismo modelo, el cual será VGG16, pero utilizando los diferentes optimizadores existentes a la hora de realizar la conversión a TensorFlow Lite. Viendo los resultados se ve que las distancias entre los puntos son mínimas o nulas, lo cual permite llegar a la conclusión que se venía augurando desde hace un tiempo. En el momento en el que se llevó a cabo el proyecto, todos y cada uno de los optimizadores del conversor de TensorFlow Lite, resultan en el mismo modelo, y esto ha quedado demostrado ya que tanto los tamaños, la tasa de aciertos y por último los tiempos de inferencia son idénticos.

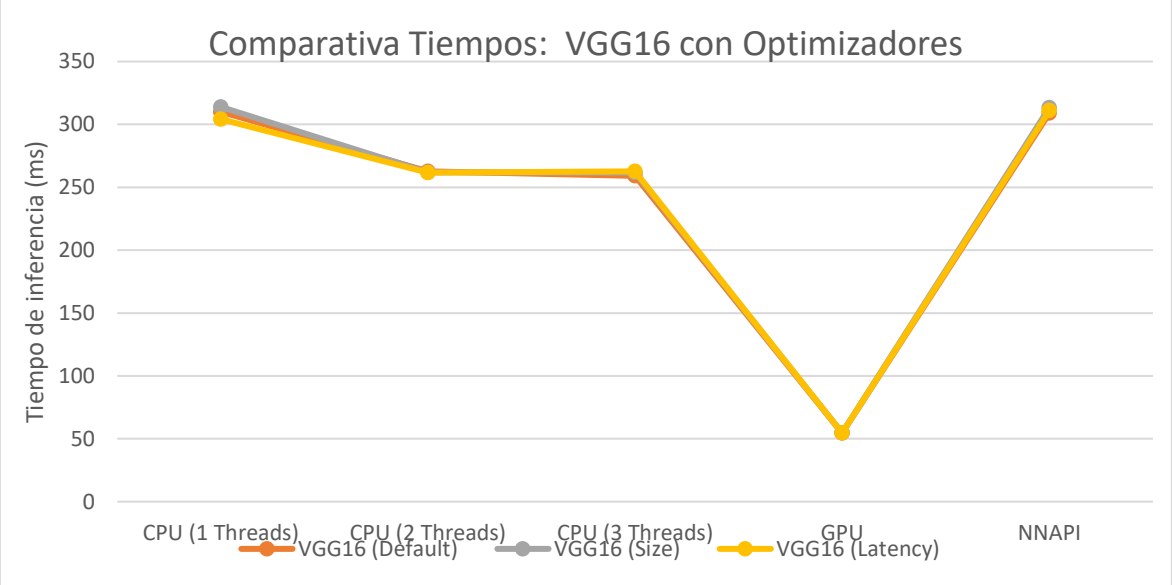

Figura 4-5: Comparativa de tiempos (ms) del modelo VGG16 con diferentes optimizadores

Solo con la Figura 4-5 se pueden observar un par de cosas extrañas. La primera es que, si previamente se ha comentado que el microprocesador contaba con 8 *cores*, por qué se han hecho pruebas únicamente hasta 3 *threads*. La respuesta es sencilla, si se ejecutaba con 4 *threads* o más podían suceder dos cosas que saliera un error parecido a "Bus Error", o que el teléfono se apagase, reiniciase o bloquease automáticamente. Aunque la respuesta fuese sencilla, quizás la explicación no lo sea tanto. Las arquitecturas que se basan en ARM big.LITTLE pueden funcionar de varios modos distintos, pero los teléfonos más modernos, y según la documentación el que se está usando también, emplean el modo reconocido como *Heterogeneous multi-processing (global task scheduling)* el cual permite hacer uso de todos

los *cores* físicos a la vez, asignando los *cores* BIG a aquellos hilos con mayor prioridad y el resto repartidos entre los disponibles. Pero lo que demuestran las pruebas es que el móvil usado no parece estar usando este modelo, sino cualquiera de los otros disponibles, en los cuales una de dos, los *cores* más potentes (BIG) se clusterizan por un lado y los menos potentes (LITTLE) por el otro, pudiéndose hacer uso únicamente de uno de ellos, este modelo se conoce como *Clustered switching*, y la otra organización que quedaría por ver es la llamada *In-kernel switcher (CPU migration)* en la cual de nuevo se hacen parejas de *cores* en las cuales se une un *core* potente (BIG) con otro de los eficientes (LITTLE), y de nuevo solo se puede hacer uso de la mitad de *cores* disponibles simultáneamente.

La segunda cosa extraña será más fácil de visualizar una vez que se empiecen a estudiar los resultados sobre el resto de las redes neuronales profundas, pero tiene que ver con el delegado de NNAPI.

Para graficar los resultados obtenidos a través del cálculo de la media de las inferencias, se ha decidido hacer una división de los datos en dos gráficas, una con los modelos más consolidados y otra de aquellos modelos enfocados a los dispositivos móviles. Esta decisión se ha llevado a cabo teniendo en cuenta que los primeros modelos tienen unos tiempos bastante más elevados que los segundos, prácticamente del doble en el mejor de los casos, haciendo que si se mostrasen todos en la misma gráfica no se podría apreciar correctamente la evolución de los modelos.

En la Figura 4-6 se observa como todos los modelos consolidados tienen una evolución muy similar en base a la configuración establecida. Para todos los modelos consolidados la peor opción es usar la CPU con un único hilo, pero la opción de la CPU va mejorando según se van introduciendo más hilos. En todos los modelos, menos en el caso de VGG16 la mejor opción para la CPU es con el máximo de hilos, que en este caso es tres, aunque con dos hilos el resultado es por lo general igual de bueno. Sin ninguna duda la opción que mejores resultados ha ofrecido es el delegado de GPU, lo cual es bastante razonable ya que la capacidad computacional que este tiene para llevar a cabo operaciones matriciales y de tensores es mucho más elevada que la de la CPU, debido a que la GPU está diseñada para esta función en específico. En cuando al delegado de NNAPI no se va a entrar a fondo en su comportamiento, ya que se hablará más en detalle en cuanto se estudien los modelos enfocados a móviles. Pero sí que es importante destacar que en el único caso donde parece que NNAPI actúa de una forma más eficaz que la CPU es para el modelo VGG16, en cuyo caso el rendimiento es prácticamente el doble de bueno que, para su mejor caso de CPU, pero sin llegar a ser tan bueno como con la configuración del delegado de GPU.

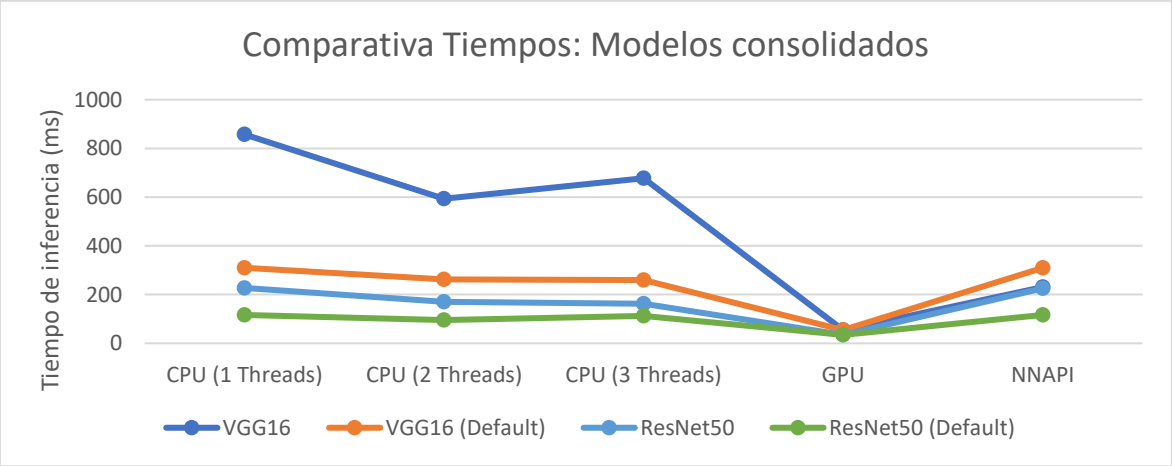

Figura 4-6: Comparativa de tiempos (ms) de los modelos consolidados – Samsung Galaxy S10+

En la Figura 4-7 se grafican los resultados de los modelos enfocados a dispositivos móviles. Se ve que en el caso de MobileNet la trayectoria seguida es prácticamente idéntica a la de los modelos consolidados, con la peculiaridad de que el mejor resultado con CPU es con una configuración de dos hilos. Como curiosidad la única red que fue capaz de soportar más de tres hilos sin problemas fue la de MobileNet, hasta incluso llegar a los ocho hilos, pero los tiempos eran idénticos o peores a los obtenidos con un único hilo. El modelo de EfficientNet es el que tiene una evolución más inesperada y opuesta al resto, ya que cuantos más hilos se introducen peor es su rendimiento tal y como se puede observar, esto posiblemente la convierta en una de las redes más eficientes en cuanto a la batería respecta, puesto que sus mejores resultados los consigue con el menor número de hilos. Al igual que sucedía con los modelos convencionales los mejores tiempos se obtienen con el delegado de GPU, aunque al ser más potente computacionalmente hablando, también implica un mayor consumo de energía.

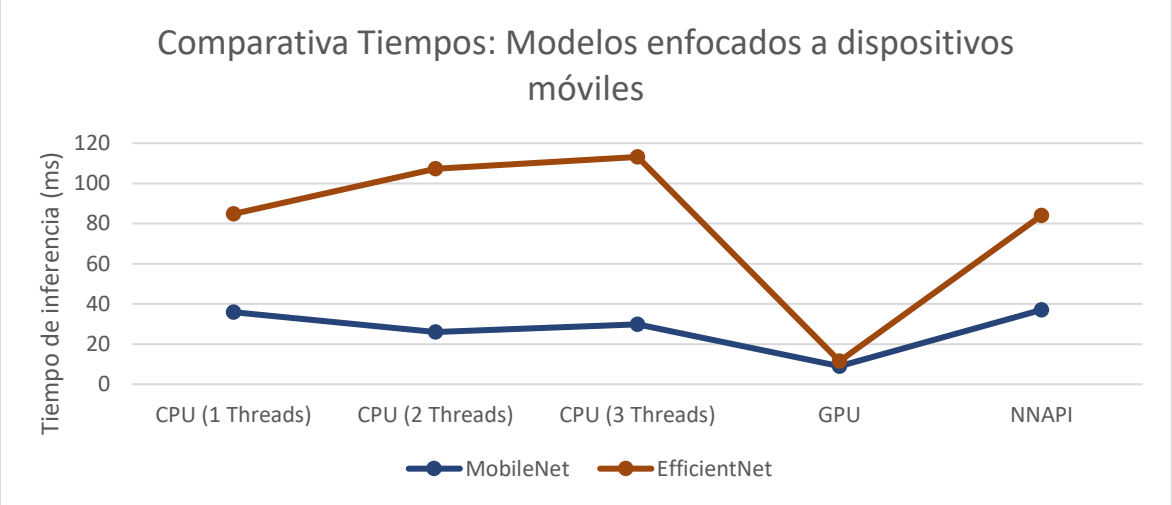

Figura 4-7: Comparativa de tiempo (ms) de los modelos enfocados a dispositivos móviles – Samsung Galaxy S10+

Una vez vistas todas las gráficas, se puede introducir la segunda cosa extraña que se detectó gracias a estas pruebas. En prácticamente todos los casos los tiempos medidos con el delegado de NNAPI tiene unos resultados idénticos a los obtenidos con la CPU con un único hilo. Solo existe una excepción, y es el caso del modelo VGG16, el cual ya se ha mencionado con anterioridad. A la conclusión a la que se llega es que por algún impedimento de software o hardware NNAPI no se puede aplicar en el dispositivo para cualquier modelo, pero cuando lo hace los resultados son muy prometedores.

La API de redes neuronales (*Neuronal Networks API* - NNAPI) [30] es una API de Android programada en C y diseñada exclusivamente para la ejecución de operaciones computacionalmente costosas de aprendizaje automático.

En caso de hacer uso del delegado NNAPI, dependiendo de los requisitos de la aplicación junto con el hardware disponible en el dispositivo, el tiempo de ejecución de la red debería distribuir la capacidad de cómputo entre todos los procesadores disponibles dentro del dispositivo, entre los que se incluyen el *hardware* dedicado para las redes neuronales como es el caso de las *Neural Processing Unit* (NPU), unidades de procesamiento gráfico (GPU) y procesadores de señales digital conocidos en inglés como *Digital Signal Processors* (DSPs).

Una vez se conoce como funciona NNAPI no se detecta un motivo claro, por el cual NNAPI no funcione correctamente para casi ningún modelo en este dispositivo. Lo único que se ha encontrado oficial es que en el caso de que el dispositivo intente usar NNAPI sin tener ningún controlador especializado, la ejecución se realizará directamente sobre la CPU, situación que parece estar sucediendo, pero esto no debería suceder para el Samsung Galaxy S10+, puesto que cuenta con una GPU y una NPU.

La Tabla 4-3 muestra los valores de los tiempos de los diferentes modelos una vez convertidos a TensorFlow Lite tanto con optimizadores como sin ellos.

Tiempo (ms)	CPU 1 Threads	CPU 2 Threads	CPU 3 Threads	GPU	NNAPI
VGG16	857,399	593,410	677,755	54,611	231,228
VGG16 (Default)	310,100	262,707	259,155	54,610	309,100
VGG16 (Size)	313,928	261,973	261,625	54,614	313,246
VGG16 (Latency)	304,141	261,793	262,740	54,623	310,985
ResNet50	227,480	170,173	162,737	34,853	225,534
ResNet50 (Default)	116,491	95,623	113,034	34,747	116,072
MobileNet	35,837	25,918	29,782	8,987	37,025
EfficientNet	84,869	107,272	113,168	11,477	84,056

Tabla 4-3: Comparativa del tiempo (ms) de los modelos – Samsung Galaxy S10+

Para hacer un estudio, un poco más matemático y menos gráfico, del impacto de usar diferentes configuraciones para cada uno de los modelos durante la inferencia, se han de conocer un par de parámetros de medición importantes.

El tiempo secuencial (T_s) es el tiempo que dura la ejecución de un proceso en un único núcleo, mientras que el tiempo en paralelo (T_p), mide la duración de ese mismo proceso bajo múltiples *cores*, aunque también se puede aplicar a hardware dedicado como GPU o incluso a librerías de software como NNAPI.

Dividiendo estas dos variables se puede calcular la **aceleración** o *speed up* (Ec. 4.1), que determina el factor de mejora que implica la introducción de más núcleos o nuevas herramientas de *hardware* y *software*:

$$Aceleración = \frac{T_s}{T_p} \qquad (4.1)$$

Como es obvio cualquier aceleración mayor a uno implica que hay una mejora respecto al valor usado para la comparación, que en este caso sería el de la ejecución con un único *core*.

En el caso de que solo se quiera medir el impacto que supone el número de núcleos que se están usando, se puede usar la medida de la **eficiencia** (Ec. 4.2) que consiste en dividir la aceleración entre el número de *cores* usados para la medición del tiempo en paralelo:

$$Eficiencia = \frac{Aceleración}{n^{o}\ cores} \qquad (4.2)$$

Por lo general, la eficiencia se puede llegar a considerar positiva si alcanza valores superiores al 80%.

Gracias a la Tabla 4-4 se puede ver de forma numérica cómo el incremento de velocidad usando múltiples hilos no es excesivamente notable, y como ya se vio con las gráficas la diferencia entre utilizar dos o tres hilos es prácticamente nula. Por el contrario, se ve como al hacer uso de la GPU las velocidades se multiplican por 5 o 6, lo que significa un incremento del 500% o 600%. El caso en el que más destaca la aceleración sobre la GPU es en el de VGG16 el cual incrementa su velocidad casi 16 veces lo cual es lógico, puesto que este modelo está pensado para unidades de procesamiento más potentes, y es por ello por lo que al usarlas se vean mejoras muy significativas. Con la aceleración se puede ver más claramente el problema ya tratado acerca de la falta de operatividad de NNAPI para todos los modelos excepto VGG16, ya que todos los valores menos este, se encuentran rondando el uno, lo cual significa que la aceleración es nula.

Aceleración	CPU 2 Threads	CPU 3 Threads	GPU	NNAPI
VGG16	1,445	1,265	15,700	3,708
VGG16 (Default)	1,180	1,197	5,678	1,003
VGG16 (Size)	1,198	1,200	5,748	1,002
VGG16 (Latency)	1,162	1,158	5,568	0,978
ResNet50	1,337	1,398	6,527	1,009
ResNet50 (Default)	1,218	1,031	3,353	1,004
MobileNet	1,383	1,203	3,988	0,968
EfficientNet	0,791	0,750	7,394	1,010

Tabla 4-4: Comparativa de la aceleración de los modelos – Samsung Galaxy S10+

Como se puede observar en la Tabla 4-5, no hay ningún modelo en el cual se consiga una eficiencia suficientemente alta como para que merezca la pena emplear múltiples hilos. Todos los modelos menos el de EfficientNet rondan una eficiencia para dos hilos de 60% o 70%, mientras que para tres hilos tiende a disminuir hasta un 40%. Como ya se habló previamente el modelo de EfficientNet no parece seguir una tendencia similar a la del resto de modelos, teniendo una eficiencia menor para todas las configuraciones, la cual ronda el 15%.

Eficiencia	CPU (2 Threads)	CPU (3 Threads)
VGG16	72,24%	42,17%
VGG16 (Default)	59,02%	39,89%
VGG16 (Size)	59,92%	40,00%
VGG16 (Latency)	58,09%	38,59%
ResNet50	66,84%	46,59%
ResNet50 (Default)	60,91%	34,35%
MobileNet	69,14%	40,11%
EfficientNet	39,56%	25,00%

Tabla 4-5: Comparativa de la eficiencia de los modelos – Samsung Galaxy S10+

4.3.2 OnePlus 8

El OnePlus 8 cuenta con un microprocesador llamado **Snapdragon 865** [31]. Su arquitectura interna es un ARMv8 de 64 bits, gestionado mediante el sistema ARM big.LITTLE el cual si parece estar funcionando con la gestión de *cores* de tipo *Heterogeneous multi-processing (global task scheduling)*, la cual aprovecha la totalidad de los mismos repartiendo las tareas entre todos ellos, otorgando los hilos con mayor prioridad a los *cores* más potentes (BIG). Este procesador cuenta con 8 *cores* de los cuales 4 son **Kryo 585 Silver** con una frecuencia de 1,8 GHz, 3 son **Kryo 585 Gold** a una frecuencia de 2,42 GHz y otro **Kryo 585 Gold** pero funcionando a una frecuencia de 2,84 GHz.

El primer cambio notable entre el OnePlus 8 y el Samsung Galaxy S10+ es que este sí que permite llevar a cabo inferencias con más de tres hilos, de hecho, se han hecho las pruebas hasta con ocho hilos. El segundo gran cambio es que el delegado de NNAPI parece funcionar correctamente cosa que en el Samsung Galaxy S10+ no ocurría, a excepción del modelo VGG16 sin optimización alguna.

En la Figura 4-8, se puede observar como todos los modelos cuando usan la CPU con un único hilo prestan sus peores tiempos, pero según se van añadiendo más hilos hasta un total de tres o cuatro estos mejoran y alcanzan sus mejores tiempos. Pero a partir de cuatro hilos, todos los tiempos se mantienen o incluso empeoran. Como ya se ha visto previamente el uso de GPU es la opción más rápida para todos los modelos, ya que su capacidad computacional está diseñada para este tipo de operaciones, pero a cambio tiene un coste energético mayor. Por fin se puede evaluar el delegado de NNAPI, el cual parece que obtiene unos tiempos ligeramente mejores que los obtenidos con la mejor configuración de hilos para la CPU. Esto significa que, si se quieren unos buenos tiempos, pero con un coste energético razonable, la opción más sencilla es la de emplear el delegado de NNAPI.

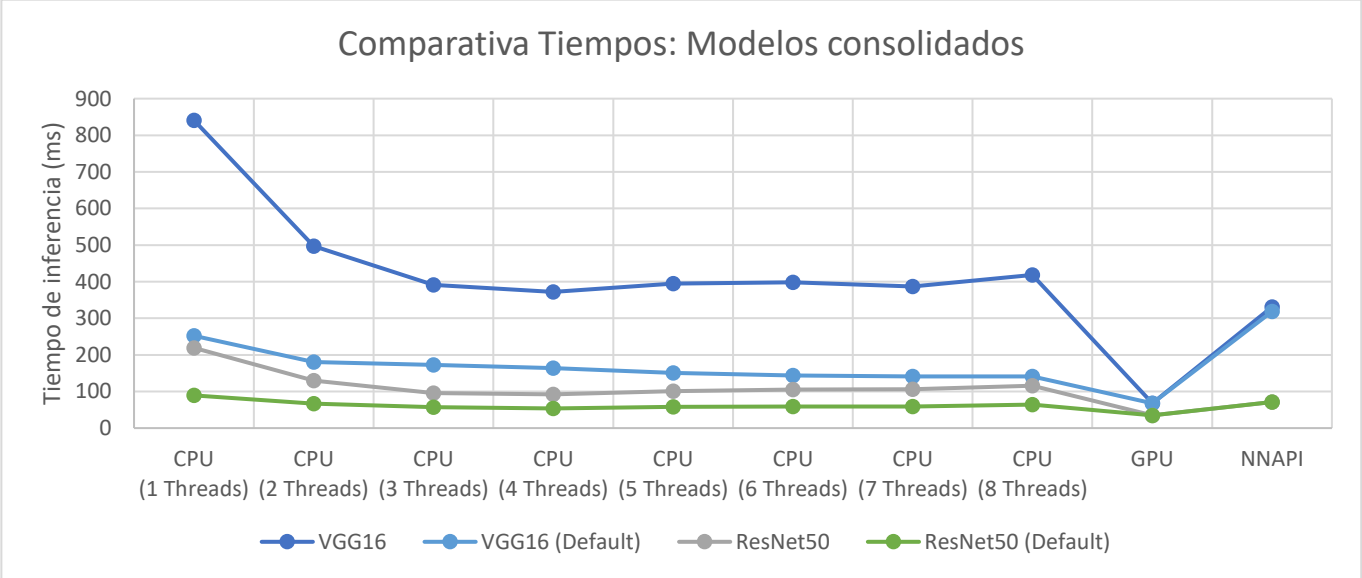

Figura 4-8: Comparativa de tiempos (ms) de los modelos consolidados – OnePlus 8

Observando la Figura 4-9 que contiene los modelos enfocados a dispositivos móviles, al ser estos bastante más rápidos por defecto, se hace más notable que la mejor configuración para la CPU es de cuatro hilos. Y de una forma similar pero más acentuada a como sucedía con los modelos consolidados, a partir de cinco hilos en adelante los tiempos empeoran.

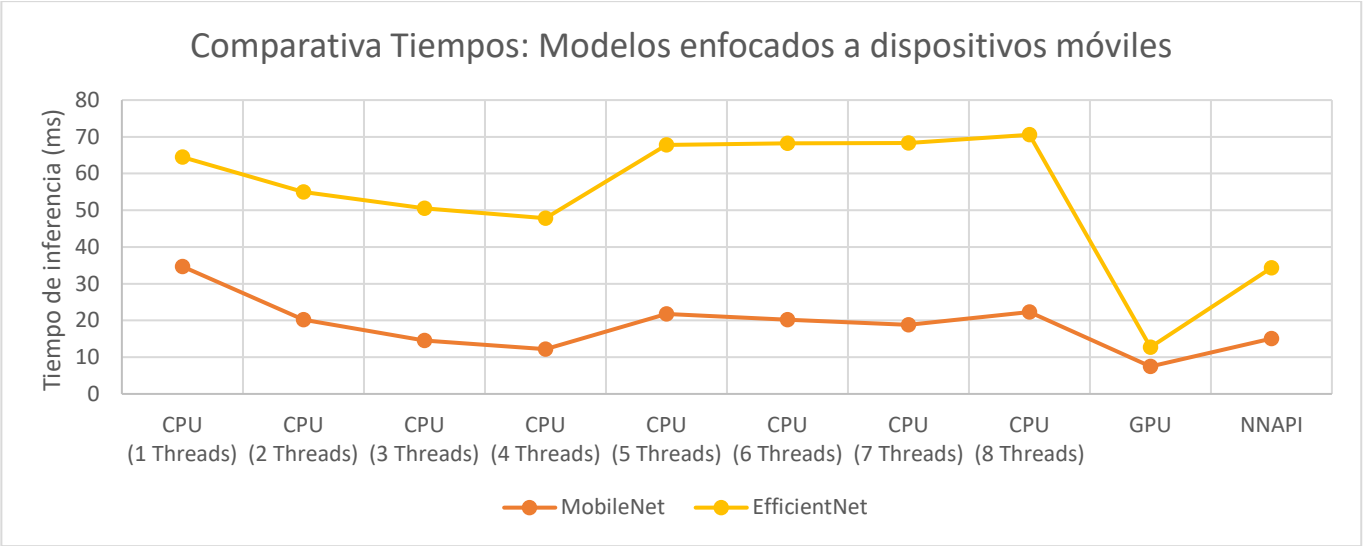

Figura 4-9: Comparativa de tiempos (ms) de los modelos enfocados a dispositivos móviles – OnePlus 8

En la Tabla 4-6, se muestran los valores de los tiempos de los diferentes modelos una vez convertidos a TensorFlow Lite, usados para graficar los diagramas anteriores.

Tiempo (ms)	CPU 1 Threads	CPU 2 Threads	CPU 3 Threads	CPU 4 Threads	CPU 5 Threads	CPU 6 Threads	CPU 7 Threads	CPU 8 Threads
VGG16	840,981	496,810	390,847	371,681	394,894	397,970	387,109	418,098
MobileNet	34,706	20,213	14,536	12,212	21,751	20,171	18,818	22,260
ResNet50	219,218	129,824	95,578	92,134	100,761	105,543	106,251	116,012
EfficientNet	64,459	54,948	50,568	47,821	67,761	68,240	68,317	70,553
VGG16 (Default)	252,221	180,252	172,939	164,229	150,812	143,907	141,223	140,792
ResNet50 (Default)	89,357	66,825	57,658	53,355	58,057	59,287	59,403	63,947

Tabla 4-6: Comparativa del tiempo (ms) de los modelos con CPU – OnePlus 8

Tiempo (ms)	GPU	NNAPI
VGG16	67,887	330,676
MobileNet	7,475	15,080
ResNet50	34,909	70,807
EfficientNet	12,729	34,297
VGG16 (Default)	67,824	318,466
ResNet50 (Default)	34,888	71,007

Tabla 4-7: Comparativa del tiempo (ms) de los modelos con GPU y NNAPI – OnePlus 8

Gracias a la Tabla 4-8, se puede observar cómo el *speed up* tiene un mayor impacto en los tiempos referentes a las pruebas con cuatro hilos o menos. A veces, el parámetro de la aceleración puede engañar un poco, como en el caso del modelo VGG16 en el que, aunque es siempre cercano a un multiplicador por dos, realmente esta no incrementa ni decrece notablemente, lo que implica que hay una mejora de velocidad con respecto a un único hilo, pero realmente da igual el número de hilos que se use siempre que sea mayor a uno. También merece la pena mencionar el caso de EfficientNet en el que a partir de cinco hilos su aceleración es inferior a uno, lo que significa que los tiempos son peores que con un único hilo. Como ya se vio en las gráficas, la aceleración obtenida con el delegado de GPU es la mejor para todos los casos.

Aceleración	CPU 2 Threads	CPU 3 Threads	CPU 4 Threads	CPU 5 Threads	CPU 6 Threads	CPU 7 Threads	CPU 8 Threads
VGG16	1,693	2,152	2,263	2,130	2,113	2,172	2,011
MobileNet	1,717	2,388	2,842	1,596	1,721	1,844	1,559
ResNet50	1,689	2,294	2,379	2,176	2,077	2,063	1,890
EfficientNet	1,173	1,275	1,348	0,951	0,945	0,944	0,914
VGG16 (Default)	1,399	1,458	1,536	1,672	1,753	1,786	1,791
ResNet50 (Default)	1,337	1,550	1,675	1,539	1,507	1,504	1,397

Tabla 4-8: Comparativa de la aceleración de los modelos con CPU – OnePlus 8

Aceleración	GPU	NNAPI
VGG16	12,388	2,543
MobileNet	4,643	2,301
ResNet50	6,280	3,096
EfficientNet	5,064	1,879
VGG16 (Default)	3,719	0,792
ResNet50 (Default)	2,561	1,258

Tabla 4-9: Comparativa de la aceleración de los modelos con GPU y NNAPI – OnePlus 8

Como se puede observar en la Tabla 4-10, parece que tan solo VGG16, ResNet50 y MobileNet parecen presentar una eficiencia igual o superior al 80% para dos hilos, valor a partir de la cual se puede considerar que la mejora de tiempo es realmente rentable en relación con el coste de recursos y energía que implica hacer uso de tal cantidad de hilos. Gracias a esta gráfica también se puede ver como cuantos más hilos se usen más decae la eficiencia, quedando claro que, aunque los tiempos sean mejores con más hilos, realmente dicha mejora no es suficientemente sustancial, ya que la eficiencia resultante sigue siendo muy baja.

Eficiencia	CPU 2 Threads	CPU 3 Threads	CPU 4 Threads	CPU 5 Threads	CPU 6 Threads	CPU 7 Threads	CPU 8 Threads
VGG16	84,64%	56,43%	42,32%	33,86%	28,21%	24,18%	21,16%
MobileNet	85,85%	57,23%	42,92%	34,34%	28,62%	24,53%	21,46%
ResNet50	84,43%	56,29%	42,21%	33,77%	28,14%	24,12%	21,11%
EfficientNet	58,65%	39,10%	29,33%	23,46%	19,55%	16,76%	14,66%
VGG16 (Default)	69,96%	46,64%	34,98%	27,99%	23,32%	19,99%	17,49%
ResNet50 (Default)	66,86%	44,57%	33,43%	26,74%	22,29%	19,10%	16,71%

Tabla 4-10: Comparativa de la eficiencia de los modelos – OnePlus 8

5 Conclusiones y trabajo futuro

5.1 Conclusiones

Este trabajo partió del objetivo inicial de la puesta en marcha y estudio de redes neuronales en dispositivos móviles gracias a las herramientas de TensorFlow Lite. Para la puesta en marcha se ha necesitado entender el funcionamiento del conversor de TensorFlow Lite y sus diferentes optimizadores. Para el estudio de las redes, se han adaptado una serie de scripts en C++ y Python que han permitido monitorizar el impacto del uso de diferentes configuraciones *hardware* y *software* a través de los delegados establecidos en TensorFlow Lite, entre los que se han podido probar se encuentran GPU y NNAPI, además del uso de múltiples hilos.

Por último, para demostrar la verdadera potencia de este tipo de redes, se han incorporado a una aplicación de clasificación de objetos en tiempo real a partir del video captado por la cámara del teléfono. Para conseguir esto, se han necesitado introducir los preprocesamientos adecuados para cada una de las redes, incluyendo la generación del código necesario para poder transformar las imágenes de RGB a BGR, junto con un proceso de *zero-centered* que consiste en centrar los valores de todos los píxeles respecto al eje de coordenadas.

En cuanto a los resultados obtenidos se podría considerar que cualquier modelo capaz de conseguir un tiempo de inferencia inferior a los 40 ms se le reconoce de tiempo real, puesto que con esos tiempos una cámara podría captar y analizar un mínimo de 25 imágenes por segundo, imágenes suficientes para hacer una evaluación del video captado por la cámara a tiempo real.

En cuanto a las redes estudiadas, el único modelo capaz de clasificar imágenes en tiempo real con cualquier configuración sería MobileNet, aunque si se usase el delegado de GPU también se podrían usar los modelos de EfficientNet y el de ResNet50 con y sin optimización en la conversión. Estos resultadas se han obtenido con dos móviles Android de gama alta, por lo que, con otros dispositivos, es posible que solo llegara a clasificar imágenes en tiempo real el modelo de MobileNet, el cual en sus mejores tiempos infería al menos tres veces más rápido que el resto.

Se puede concluir con la mejora sustancial de tiempo que implica el uso del delegado de GPU, pero sobre todo aplicado a los modelos enfocados a dispositivos móviles, los cuales cuentan con unos tiempos de inferencia mucho menores de alrededor de un tercio con respecto al mejor tiempo obtenido con cualquiera de los modelos consolidados.

En cuanto al delegado de NNAPI se ha visto que no funciona adecuadamente en todos los dispositivos, y esto puede ser por el tipo de *hardware* con los que estos cuenten, en el caso del Samsung Galaxy S10+ contaba con un NPU mientras que el OnePlus 8 con un DSP. Esto es una hipótesis, por lo que no es del todo concluyente, para llegar a un resultado más decisivo se deberían de hacer muchas más pruebas sobre distintos dispositivos.

Este trabajo no solo ha permitido ampliar el conocimiento acerca de redes neuronales, desde su diseño hasta las librerías que las implementan, sino que ha facilitado alcanzar el conocimiento necesario para iniciarse en el mundo de Android y la infinidad de posibilidades que lo rodean, pero sobre todo en el desarrollo de aplicaciones que fue una de las motivaciones iniciales.

5.2 Trabajo futuro

Como ya se ha comentado, este tipo de estudios tienen infinidad de posibilidades, pero a continuación se plantearán algunas de las ideas que han ido surgiendo durante la realización del trabajo.

Sería muy curioso estudiar el impacto que tiene la introducción y uso de estas redes dentro de aplicaciones reales, ya que, debido a los recursos implícitos de la aplicación, la competencia por los mismos dentro del dispositivo es mayor, haciendo que los tiempos de inferencia aumenten, debido al que las unidades de procesamiento se encuentran con mayor actividad. También hay que tener en cuenta que la aplicación usada está montada al completo sobre Java, lenguaje que no es puntero por su velocidad, por lo que podría ser curioso introducir ciertas operaciones costosas como las de inferencia y preprocesamiento de las imágenes en archivos embebidos en C++ y comprobar el impacto que esto tendría sobre los tiempos.

Por supuesto que sería muy interesante hacer más pruebas sobre otras redes neuronales distintas a las planteadas en este proyecto como AlexNet, DenseNet, NASNetMobile, etc. las cuales pueden arrojar mejoras significativas. También sería posible hacer pruebas sobre distintos dispositivos no solo de gama alta para encontrar las mejores redes profundas y mejores configuraciones de inferencia para que una aplicación de este tipo pudiera llegar a funcionar en cualquier dispositivo sin problemas de rendimiento.

Finalmente, una duda que surgió durante el desarrollo del trabajo era si existía la posibilidad de transformar a TensorFlow Lite una red que no hubiera sido entrenada en Keras. Esto plantea el inicio de un nuevo trabajo de investigación, aunque aparentemente parece ser posible, ya que se han encontrado ciertas formas de transformar modelos de PyTorch a TensorFlow [32], y ya desde esta se podría convertir a un modelo de TensorFlow Lite.

Referencias

[1] J. J. Bryson, «La última década y el futuro del impacto de la IA en la sociedad,» 2018. [En línea]. Available: https://www.bbvaopenmind.com/articulos/la-ultima-decada-y-el-futuro-del-impacto-de-la-ia-en-la-sociedad/. [Último acceso: 09 06 2021].

[2] S. Plata, «En el 2021 se triplicará el uso de IA para combatir el fraude,» 27 06 2019. [En línea]. Available: https://cio.com.mx/en-el-2021-se-triplicara-el-uso-de-ia-para-combatir-el-fraude/. [Último acceso: 09 06 2021].

[3] A. Torres, «Estado del arte de la Inteligencia Artificial en 2021,» 23 02 2021. [En línea]. Available: https://www.sage.com/es-es/blog/estado-del-arte-de-la-inteligencia-artificial-en-2021/. [Último acceso: 06 06 2021].

[4] Wikipedia, «Convolutional neural network - Wikipedia,» [En línea]. Available: https://en.wikipedia.org/wiki/Convolutional_neural_network. [Último acceso: 09 06 2021].

[5] B. K. Varghese, A. Augustine, J. M. Babu, D. Sunny y S. Cherian, «INFOPLANT: Plant Recognition using Convolutional Neural Networks,» de *INFOPLANT: Plant Recognition using Convolutional Neural Networks*, 2020, pp. 800-807.

[6] Developers, Google, «Android Debug Bridge (adb) | Desarrolladores de Android,» [En línea]. Available: https://developer.android.com/studio/command-line/adb. [Último acceso: 02 06 2021].

[7] Google, «Bazel - a fast, scalable, multi-language and extensible build system" - Bazel,» [En línea]. Available: https://bazel.build/. [Último acceso: 02 06 2021].

[8] G. Brain, «TensorFlow Lite | ML for Mobile and Edge Devices,» [En línea]. Available: https://www.tensorflow.org/lite. [Último acceso: 02 06 2021].

[9] G. François Chollet, «Keras: the Python deep learning API,» [En línea]. Available: https://keras.io/. [Último acceso: 02 06 2021].

[10] Keras, «Keras Applications,» [En línea]. Available: https://keras.io/api/applications/. [Último acceso: 02 06 2021].

[11] G. Cloud, «Cloud TPU | Google Cloud,» [En línea]. Available: https://cloud.google.com/tpu. [Último acceso: 02 06 2021].

[12] K. Simonyan y A. Zisserman, «Very Deep Convolutional Networks for Large-Scale Image Recognition,» *arXiv [cs.CV],* vol. 1409.1556, 2014.

[13] K. He, X. Zhang, S. Ren y J. Sun, «Deep Residual Learning for Image Recognition,» *arXiv [cs.CV],* vol. 1512.03385, 2015.

[14] A. G. H. a. M. Z. a. B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto y H. Adam, «MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications,» *arXiv [cs.CV],* vol. 1704.04861, 2017.

[15] M. Tan y Q. V. Le, «EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks,» *arXiv [cs.CV],* vol. 1905.11946, 2020.

[16] G. Android Developers, «Download Android Studio and SDK tools | Android Studio,» [En línea]. Available: https://developer.android.com/studio. [Último acceso: 02 06 2021].

[17] S. V. Lab, S. Universit y P. U. , «ImageNet,» 11 03 2021. [En línea]. Available: https://www.image-net.org/. [Último acceso: 08 05 2021].

[18] S. V. Lab, S. University y P. University, «ImageNet Large Scale Visual Recognition Challenge 2012 (ILSVRC2012),» [En línea]. Available: https://image-net.org/challenges/LSVRC/2012/2012-downloads.php#images. [Último acceso: 20 05 2021].

[19] TensorFlow, «tf.lite.Optimize | TensorFlow Core v2.5.0,» 14 05 2021. [En línea]. Available: https://www.tensorflow.org/api_docs/python/tf/lite/Optimize. [Último acceso: 20 05 2021].

[20] Google, «Converter Python API guide,» [En línea]. Available: https://android.googlesource.com/platform/external/tensorflow/+/33965c1ca30600824f1bc17d5dee30b0c80ce1b6/tensorflow/lite/g3doc/convert/python_api.md. [Último acceso: 10 02 2021].

[21] TensorFlow, «tensorflow/tensorflow/lite/examples/python at master · tensorflow/tensorflow,» 05 05 2021. [En línea]. Available: https://github.com/tensorflow/tensorflow/tree/master/tensorflow/lite/examples/python. [Último acceso: 05 06 2021].

[22] TensorFlow, «tensorflow/tensorflow/lite/tools/evaluation/tasks/imagenet_image_classification at master · tensorflow/tensorflow,» 27 05 2021. [En línea]. Available: https://github.com/tensorflow/tensorflow/tree/master/tensorflow/lite/tools/evaluation/tasks/imagenet_image_classification. [Último acceso: 27 05 2021].

[23] WikiChip, «Exynos 9820 - Samsung - WikiChip,» [En línea]. Available: https://en.wikichip.org/wiki/samsung/exynos/9820. [Último acceso: 05 06 2021].

[24] TensorFlow, «tensorflow/tensorflow/lite/tools/benchmark at master · tensorflow/tensorflow,» 21 05 2021. [En línea]. Available: https://github.com/tensorflow/tensorflow/tree/master/tensorflow/lite/tools/benchmark. [Último acceso: 05 06 2021].

[25] TensorFlow, «TensorFlow Lite Examples | Machine Learning Mobile Apps,» [En línea]. Available: https://www.tensorflow.org/lite/examples. [Último acceso: 10 05 2021].

[26] TensorFlow, «examples/lite/examples/image_classification/android at master · tensorflow/examples,» 03 05 2021. [En línea]. Available: https://github.com/tensorflow/examples/tree/master/lite/examples/image_classification/android. [Último acceso: 10 05 2021].

[27] Facebook, «Image Pre-Processing | Caffe2,» [En línea]. Available: https://caffe2.ai/docs/tutorial-image-pre-processing.html. [Último acceso: 10 05 2021].

[28] Keras, «tensorflow/imagenet_utils.py at a4dfb8d1a71385bd6d122e4f27f86dcebb96712d · tensorflow/tensorflow,» 24 03 2021. [En línea]. Available: https://github.com/tensorflow/tensorflow/blob/a4dfb8d1a71385bd6d122e4f27f86dcebb96712d/tensorflow/python/keras/applications/imagenet_utils.py#L166. [Último acceso: 05 06 2021].

[29] Wikipedia, «ARM big.LITTLE - Wikipedia,» [En línea]. Available: https://en.wikipedia.org/wiki/ARM_big.LITTLE. [Último acceso: 05 06 2021].

[30] G. Android Developers, «Neural Networks API | Android NDK | Android Developers,» 18 05 2021. [En línea]. Available:

https://developer.android.com/ndk/guides/neuralnetworks. [Último acceso: 05 06 2021].

[31] WikiChip, «Snapdragon 865 - Qualcomm - WikiChip,» [En línea]. Available: https://en.wikichip.org/wiki/qualcomm/snapdragon_800/865. [Último acceso: 05 06 2021].

[32] A. Singh, «Converting A Model From Pytorch To Tensorflow: Guide To ONNX,» 08 03 2021. [En línea]. Available: https://analyticsindiamag.com/converting-a-model-from-pytorch-to-tensorflow-guide-to-onnx/. [Último acceso: 11 06 2021].

[33] C. Loewen y olprod, «Instalación de WSL en Windows 10,» 07 04 2021. [En línea]. Available: https://docs.microsoft.com/es-es/windows/wsl/install-win10. [Último acceso: 05 06 2021].

[34] G. Android Developers, «SDK Platform Tools release notes | Android Developers,» 18 05 2021. [En línea]. Available: https://developer.android.com/studio/releases/platform-tools. [Último acceso: 05 06 2021].

[35] Bazel, «Installing Bazel on Ubuntu - Bazel main,» [En línea]. Available: https://docs.bazel.build/versions/master/install-ubuntu.html. [Último acceso: 21 02 2021].

[36] TensorFlow y G. Contributors, «tensorflow/tensorflow: An Open Source Machine Learning Framework for Everyone,» 11 06 2021. [En línea]. Available: https://github.com/tensorflow/tensorflow. [Último acceso: 10 02 2021].

[37] TensorFlow, «Build from source | TensorFlow,» 25 05 2021. [En línea]. Available: https://www.tensorflow.org/install/source#configure_the_build. [Último acceso: 10 02 2021].

[38] G. Android Developers, 09 06 2021. [En línea]. Available: https://developer.android.com/ndk/downloads. [Último acceso: 11 06 2021].

[39] S. Sistemas, «▷ Instalar Android SDK Manager Ubuntu 20.04 - Solvetic,» 17 09 2020. [En línea]. Available: https://www.solvetic.com/tutoriales/article/8865-instalar-android-sdk-manager-ubuntu-20-04/. [Último acceso: 12 02 2021].

Glosario

IA	Inteligencia Artificial
API	Application Programming Interface
ADB	Android Debug Bridge
ILSVRC	ImageNet Large Scale Visual Recognition Challenge
ROM	Read Only Memory
GUI	Graphical User Interface
WSL	Windows Subsystem for Linux
BIOS	Basic Input Output System
IoT	Internet Of Things
TPU	Tensor Processing Unit
IDE	Integrated Development Enviroment

Anexos

A Configuración del dispositivo Android

Para poder usar el dispositivo móvil con total libertad y poder tener un mayor acceso y control de los recursos de este se ha de activar el modo de desarrollador y la depuración por USB. Este proceso no dará control total sobre el dispositivo, ya que para conseguir esto se debería rootear el móvil, pero para el estudio que se va a llevar a cabo no será necesario.

El proceso para activar el modo desarrollador puede variar de un dispositivo a otro dependiendo de su versión de Android y las modificaciones que haya hecho la marca sobre la ROM (*Read Only Memory*) en cuestión. Las ROM contienen el conjunto de instrucciones de arranque del dispositivo, junto con toda la gestión de la interfaz gráfica o GUI (*Graphical User Interface*) y el *firmware* necesario para el correcto funcionamiento del dispositivo.

Este proceso se llevará a cabo desde un Samsung Galaxy S10+, el cual cuenta con la versión de Android 11. Para activar el modo desarrollador se ha necesitado acceder a la aplicación de **Ajustes**, y dentro de la misma acceder al apartado de **Acerca del teléfono**. Este paso es común para casi todas las versiones de Android, con la excepción de Android 8 que la ruta a seguir sería **Ajustes**, en donde dentro se encuentra una pestaña de **Sistema**, y finalmente en su interior se puede localizar la pestaña de **Acerca del teléfono**. Una vez dentro del apartado de **Acerca del teléfono** se debe hallar un campo llamado **Número de compilación**, en mi caso se encontraba dentro de la pestaña de **Información de Software**. Para activar el modo de desarrollador se ha de hacer click 7 veces sobre el apartado de **Numero de compilación**. Este proceso se encuentra representado visualmente en la Figura A-1.

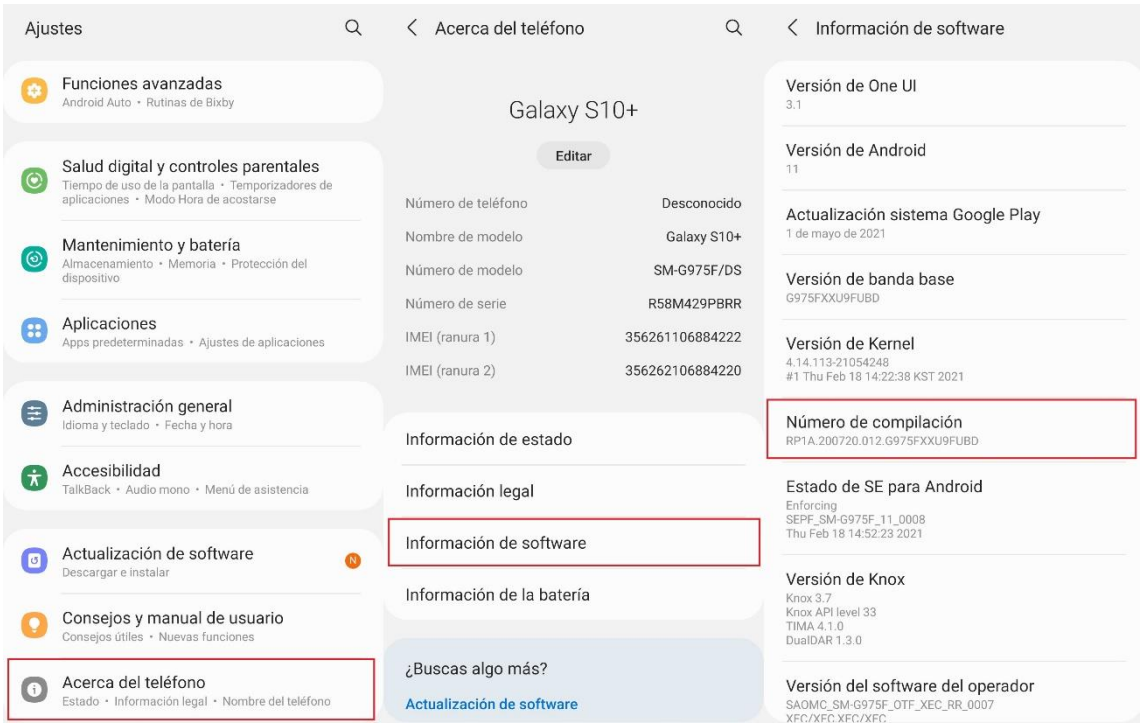

Figura A-1: Activación de Opciones de Desarrollador

Como dato a destacar, en las versiones de Android 4 e inferiores la opción de desarrollador venía activada por defecto.

Una vez activado el modo de desarrollador saldrá una nueva pestaña dentro de la aplicación de **Ajustes**, la cual, posiblemente se localice al final del todo y se ha de llamar **Opciones de desarrollador**. Dentro de esta nueva pestaña se ha de localizar la sección de Depuración para así poder activar la opción de **Depuración por USB**, esto se puede visualizar en la Figura A-2. Esta opción permitirá en un futuro usar una terminal a través de la aplicación llamada ADB desde nuestro PC para la ejecución de operaciones en nuestro dispositivo Android. Gracias a esta conexión entre el ordenador y el móvil se podrá, entre muchas posibles operaciones, debuguear aplicaciones, leer el log de actividad y errores del teléfono, etc...

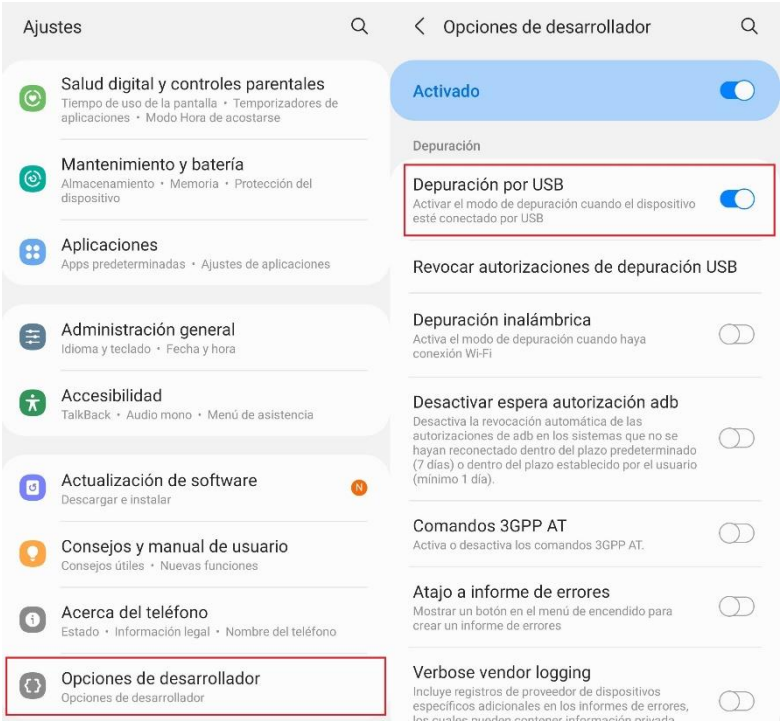

Figura A-2: Activación de la Depuración por USB

B Configuración del computador

Para poder llevar a cabo este proyecto, ha sido imprescindible la configuración e instalación de ciertas librerías, aplicaciones y programas. Por facilidad de uso e instalación se ha preferido llevar a cabo este proceso desde un sistema operativo de Windows 10. El motivo por el que se eligió esta distribución es que, debido a que desde hace ya unos años incorpora una herramienta de trabajo muy útil llamada *Windows Subsystem for Linux* (WSL) que permite tener un subsistema de Linux perfectamente funcional instalado en Windows, el cual permite hacer uso de todas las herramientas de línea de comando sin la necesidad de crear máquinas virtuales ni configurar ningún tipo de inicio dual en nuestra BIOS, lo cual sin duda es bastante más engorroso y demandante de recursos de procesamiento y almacenamiento.

Para instalar la herramienta de WSL en nuestro ordenador lo más recomendable es seguir la guía oficial de Microsoft [33]. El último paso de esta guía es entrar en la aplicación de **Microsoft Store** y desde allí descargar la distribución de Linux deseada. Para todas las pruebas de este proyecto, se hará uso de Ubuntu 20.04 LTS.

Una vez aclaradas las herramientas principales que se usarán para llevar a cabo el proyecto, se comenzará con la guía de instalación de los principales programas y aplicaciones que serán necesarios a lo largo de este.

La instalación de ADB se puede llevar a cabo de dos formas distintas. La primera consistiría en la descarga de las herramientas de trabajo de Android SDK [34] y una vez descargado el paquete, se pueden visualizar dentro de la carpeta en cuestión una serie de programas, y entre ellos se encuentra el de ADB. Por el momento la aplicación de ADB solo se podría usar desde una terminal dentro de ese directorio, lo cual es un poco incómodo, pero este problema tiene solución, y se mostrará después de la explicación de la segunda opción de instalación.

La otra forma de instalar la aplicación de ADB, y es la que se recomienda, es mediante la instalación de otra aplicación llamada AndoridStudio, ya que esta incluye de forma indirecta la instalación de los paquetes de Android SDK. Además, AndroidStudio se usará al final del proyecto, por lo que de esta forma se estarán instalando dos programas simultáneamente y de una forma más sencilla. Para instalar esta aplicación se debe acceder a su página [16] y una vez dentro seguir los pasos que se nos indiquen. Una vez terminada la instalación del programa y de todos los paquetes que se vayan necesitando, se deberá abrir la aplicación recién instalada y desde la pantalla de inicio se accederá a la opción de **Configuración** y dentro de esta, a la opción de **SDK Manager**, tal y como muestra la Figura B-1.

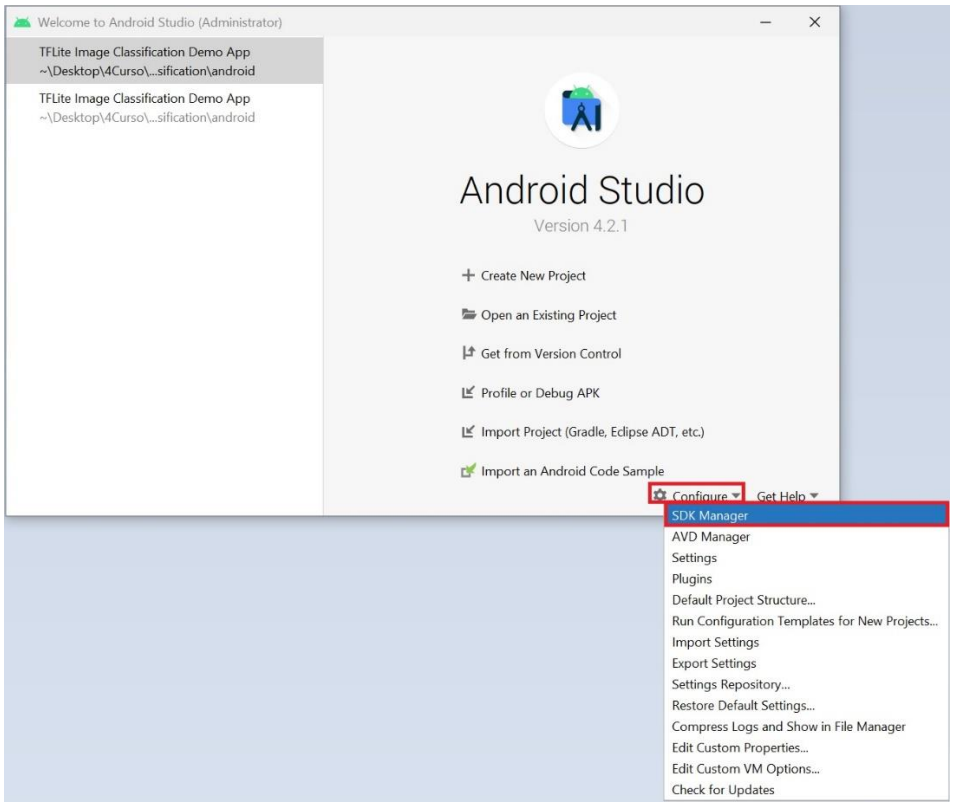

Figura B-1: Acceso a la opción del controlador *SDK Manager*

Dentro de la pestaña de **SDK Platforms** se podrán ver los paquetes con las diferentes versiones de Android que se tengan descargados. En un principio solo haría falta descargarse la versión de Android con la que vaya a trabajar, pero es altamente recomendable instalarse algunas versiones anteriores para evitar futuros problemas de incompatibilidad entre versiones u otros problemas similares. El resultado debería ser más o menos como el mostrado en la Figura B-2.

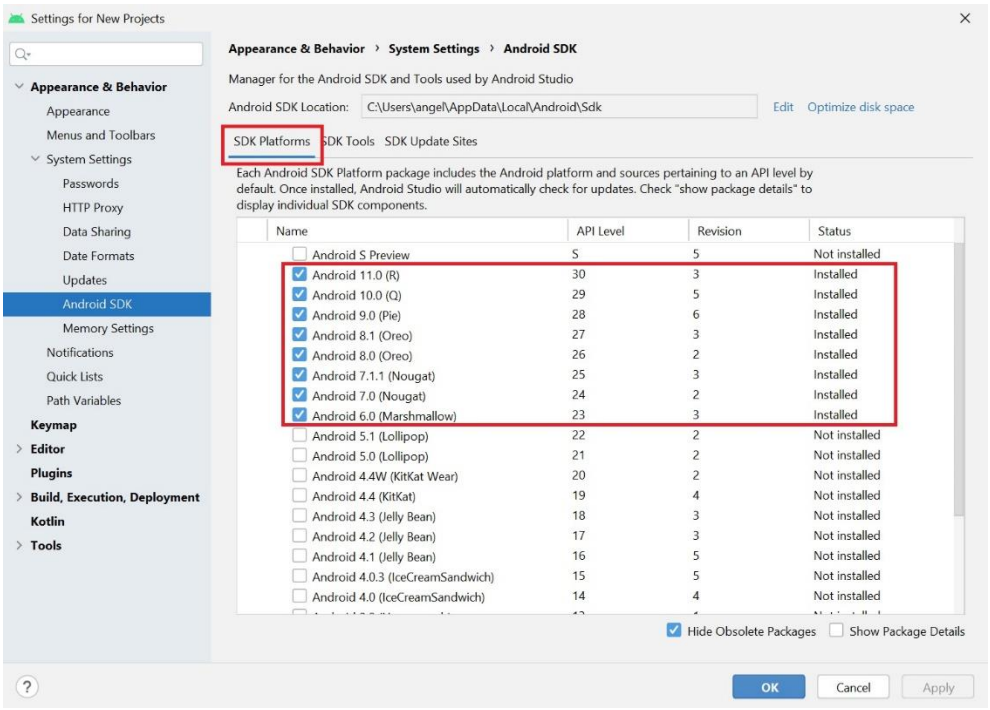

Figura B-2: Control de versiones de Android desde *SDK Platforms*

Para cerciorarse de que se han instalado todos los paquetes imprescindibles de SDK, se deberá de acceder a la segunda pestaña llamada **SDK Tools** en la cual se deberán encontrar marcadas al menos las líneas **Android SKD Command-Line Tools** y **Android SDK Platform-Tools**, tal y como muestra la Figura B-3. En caso de que esto no sea así se han de seleccionar y seguir los pasos indicados para su correcta instalación.

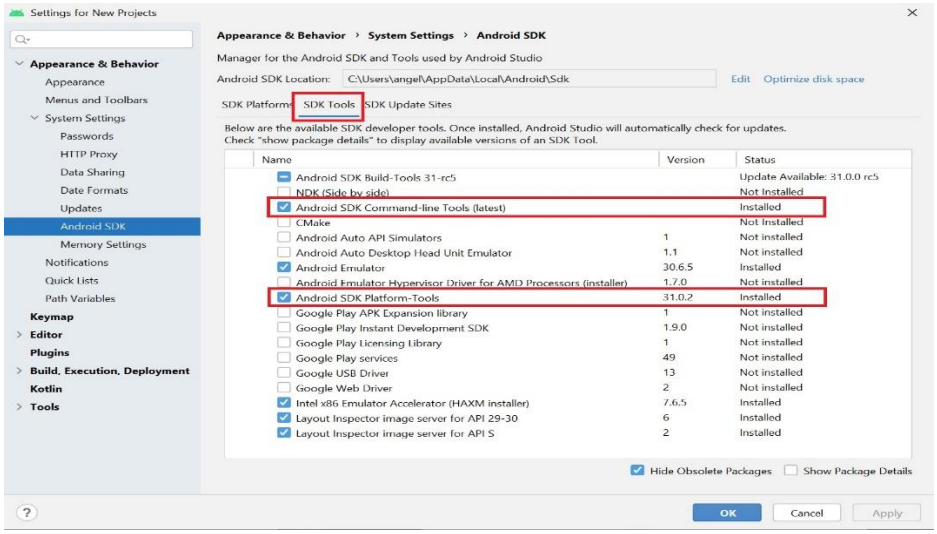

Figura B-3: Control de herramientas desde *SDK Tools*

Una vez se ha comprobado que se encuentran instalados todos los paquetes y herramientas mencionados, es posible que al intentar abrir un proyecto existente o crear uno nuevo surja un error el cual indica que no se tienen las licencias de SDK activadas.

Para solucionar este problema, se deberá acceder a la ruta donde se haya instalado Android SDK. Si no se está seguro donde se encuentra esta, se puede localizar en la ventana de la Figura B-3 en el apartado superior de **Android SKD Location**. Una vez dentro de este directorio se debe de acceder a la carpeta de **tools** y una vez dentro, a la carpeta de **bin**. Finalmente, dentro de este directorio se deberá abrir una terminal, en caso de ser posible en modo administrador, y ejecutar el script llamado **sdkmanager.bat** con parámetro **–licenses** tal y como se muestra en el Código B-1.

```
1.  Windows: Android\Sdk\tools\bin > sdkmanager.bat --licenses
```

Código B-1: Comando de activación de las licencias de Android SDK en Windows

Una vez se termine la ejecución del script, AndroidStudio y las herramientas de SDK estarían instaladas y listas para su uso. Aun así, si se accede a una terminal cualquiera y se intenta lanzar el comando adb, lo más posible es que se muestre un error, el del Código B-2 en el cual se indica que el nombre del comando no se ha reconocido como ningún programa existente, lo cual significa que aún no se ha establecido la variable de entorno necesaria.

```
1.  Windows: $ > adb
2.  adb : El término 'adb' no se reconoce como nombre de un cmdlet, función, archivo
    de script o programa ejecutable. Compruebe si escribió correctamente el nombre o,
    si incluyó una ruta de acceso, compruebe que dicha ruta es correcta e inténtelo de
    nuevo.
3.  En línea: 1 Carácter: 1
4.  + adb
5.  + ~~~
6.      + CategoryInfo          : ObjectNotFound: (adb:String) [],
    CommandNotFoundException
7.      + FullyQualifiedErrorId : CommandNotFoundException
```

Código B-2: Resultado esperado cuando ADB no esta establecido como variable de sesión

Para establecer la variable de entorno se deberá dirigir al buscador de Windows y buscar la opción llamada **Editar las variables de entorno del sistema**. Dentro de la ventana emergente se podrá ver un botón llamado **Variables de entorno…** Una vez dentro de esta nueva pestaña se deberá de buscar una variable de sistema, cuidado no confundirla con una variable de usuario, la cual se ha de llamar **Path**. Una vez encontrada se puede hacer doble click sobre ella o seleccionar la opción de **Editar…** Dentro de esta nueva ventana se verán todas las direcciones asociadas a dicha variable de sesión, pero el objetivo es añadir una nueva, por lo que se seleccionará la opción de **Nuevo** y se añadirá la misma dirección donde se tenga instalado **Android SDK** añadiéndole al final de la ruta **/platform-tools** puesto que es dentro de este directorio donde se encuentran todas las aplicaciones de línea de comando de Android SDK. Los pasos a seguir se muestran de forma visual en la Figura B-4.

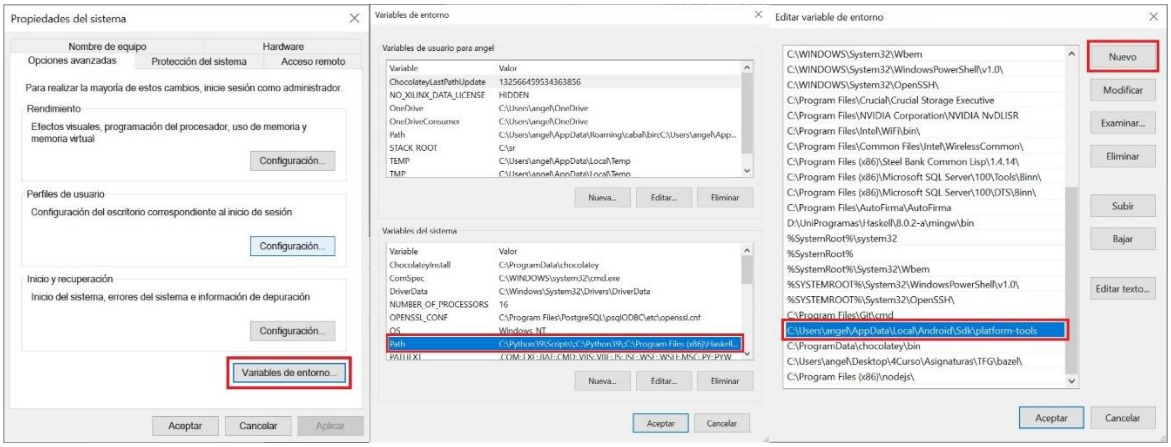

Figura B-4: Activación de la variable de entorno de ADB

Una vez se han terminado estas modificaciones se podrá ejecutar el comando de ADB desde una terminal cualquiera sin tener en cuenta el directorio desde el que se desee lanzar el comando.

Ahora toca centrarse en la realización de toda la configuración de programas y librerías en el subsistema de Linux WSL.

Se comenzará por la apertura de una terminal en la distro de elección. El primer programa que se instalará será el de Bazel. En el caso de usar una distribución de Ubuntu se pueden seguir los pasos explicados en esta página [35]. El primer paso consta de una serie de comandos para añadir la URI de la distribución de Bazel como un paquete en el sistema. El segundo paso consiste en ejecutar el comando de actualización de paquetes seguido del instalador de la aplicación de Bazel. En este paso hay que tener mucho cuidado e instalar la versión adecuada para el repositorio de TensorFlow [36], ya que, si por el contrario se usa una versión superior o inferior, no se podrá hacer uso de sus herramientas. En concreto, a la fecha en de realización del proyecto el repositorio requiere de la versión 3.7.2, por lo que el comando a lanzar debería de ser el del Código B-3. Para estar seguros de cuál es la versión se debería de estar usando se puede consultar el archivo llamado **.bazelversion** que se encuentra dentro del repositorio.

```
1.  Ubuntu:~$ sudo apt update && sudo apt install bazel-3.7.2
```

Código B-3: Comando para la instalación de la versión 3.7.2 de Bazel en Ubuntu

Una vez se tiene Bazel instalado, se ha de clonar el repositorio de TensorFlow que posee muchas de las herramientas que se usarán para las mediciones de tiempo y rendimiento de modelos en Android y otros muchos scripts útiles. Para llevar esto a cabo basta con clonar el repositorio desde cualquier terminal con el comando del Código B-4.

```
1.  Ubuntu:~$ git clone https://github.com/tensorflow/tensorflow.git
```

Código B-4: Comando para clonar el repositorio oficial de TensorFlow

Dentro de la carpeta del repositorio lo primero que se debe de hacer es configurar el funcionamiento de Bazel estableciéndole cuáles son las rutas de instalación donde se encuentran ciertos módulos imprescindibles de los que se hablarán a continuación. Para llevar a cabo este proceso se puede lanzar el script mediante la llamada **./configure** o también se puede llamar directamente al archivo de Python con el comando **python3 configure.py** [37].

La parte más importante de este proceso es cuando se pregunta si se desea configurar el entorno para construcciones sobre arquitectura Android, es ahí donde se debe de responder si ir estableciendo cada ruta según se vayan solicitando. Para la descarga de Android NDK se puede llevar a cabo desde [38] y solamente habría que extraerlo del archivo comprimido y guardarlo en el directorio deseado. Por el contrario, Android SDK se puede descargar desde [16] aunque este segundo paquete es algo más complicado y se recomienda visitar alguna otra fuente como [39] ya que quizás se deba de configurar alguna variable de sesión y el entorno de JDK Java. Una ejecución de ejemplo de esta configuración es la mostrada en el Código B-5.

```
1.  Ubuntu:~/tensorflow$ ./configure
2.  You have bazel 3.7.2 installed.
3.  Please specify the location of python. [Default is /usr/bin/python3]:
4.
5.
6.  Found possible Python library paths:
7.    /usr/lib/python3/dist-packages
8.    /usr/local/lib/python3.8/dist-packages
9.  Please input the desired Python library path to use.  Default is
    [/usr/lib/python3/dist-packages]
10.
11. Do you wish to build TensorFlow with ROCm support? [y/N]: N
12. No ROCm support will be enabled for TensorFlow.
13.
14. Do you wish to build TensorFlow with CUDA support? [y/N]: N
15. No CUDA support will be enabled for TensorFlow.
16.
17. Do you wish to download a fresh release of clang? (Experimental) [y/N]: N
18. Clang will not be downloaded.
19.
20. Please specify optimization flags to use during compilation when bazel option "--
    config=opt" is specified [Default is -Wno-sign-compare]:
21.
22. Would you like to interactively configure ./WORKSPACE for Android builds? [y/N]: y
23. Searching for NDK and SDK installations.
24.
25. Please specify the home path of the Android NDK to use. [Default is
    /home/aanxel/Android/Sdk/ndk-bundle]: /home/aanxel/Android/android-ndk-r21e
26.
27. Please specify the (min) Android NDK API level to use. [Available levels: ['16',
    '17', '18', '19', '21', '22', '23', '24', '26', '27', '28', '29', '30']] [Default
    is 21]:
28.
29. Please specify the home path of the Android SDK to use. [Default is
    /home/aanxel/Android/Sdk]: /home/aanxel/Android/commandlinetools-linux-
    7302050_latest
30.
```

```
31. Either /home/aanxel/Android/commandlinetools-linux-7302050_latest does not exist,
    or it does not contain the subdirectories "platforms" and "build-tools".
32. Please specify the home path of the Android SDK to use. [Default is
    /home/aanxel/Android/Sdk]: /home/aanxel/DevTools/Android
33.
34. Please specify the Android SDK API level to use. [Available levels: ['26']]
    [Default is 26]:
35.
36. Please specify an Android build tools version to use. [Available versions:
    ['26.0.1']] [Default is 26.0.1]:
```

Código B-5: Comando y pasos para configurar el repositorio oficial de TensorFlow

Una vez terminada la ejecución del script de configuración de Bazel, en caso de que se hayan puesto todos los directorios de forma correcta y no haya surgido ningún problema en el proceso, se tendrá Bazel correctamente configurado para la construcción de cualquier aplicación del repositorio sobre la arquitectura de procesadores que se haya establecido en la configuración.

Por un motivo de organización y limpieza, se ha decidido crear un entorno virtual en donde se descargarán las librerías pertinentes para el desarrollo del proyecto. Para crear dicho entorno virtual, basta con ejecutar la primera línea del Código B-6. Una vez creado, se podrá activar dicho entorno gracias al comando de la segunda línea de dicho código. Y cuando se haya terminado y se desee cerrar el entorno virtual, bastaría con pulsar **Ctrl+D** o escribir el comando de la tercera línea.

```
1.  Ubuntu:~$ python3 -m venv –system-site-packages ./venv
2.  Ubuntu:~$ source ./venv/bin/activate
3.  Ubuntu:~$ deactivate
```

Código B-6: Comando para relacionados con el entorno virtual de Python

Tres de las librerías imprescindibles de Python3 que se usarán en este proyecto son las de **Numpy** para el manejo de arrays, **Keras** para la obtención de modelos de clasificación de imágenes ya preentrenados y **TensorFlow** para poder hacer uso de la API de Lite que permitirá transformar los modelos de Keras a TensorFlow Lite. Para instalar estas librerías se debe de tener el entorno virtual activo para poder aprovechar las ventajas mencionadas previamente y lanzar el conjunto de comandos del Código B-7.

```
1.  (venv) Ubuntu:~$ pip3 isntall numpy
2.  (venv) Ubuntu:~$ pip3 install --upgrade tensorflow
3.  (venv) Ubuntu:~$ pip3 install keras
```

Código B-7: Comando para instalar las librerías necesarias en el entorno virtual de Python

C Clasificar una imagen con un modelo TensorFlow Lite

El script de Python del Código C-1 permite obtener la salida de un modelo de TensorFlow Lite sobre una imagen. La diferencia con el script original obtenido de [21] es que se ha añadido un control más exhaustivo sobre el preprocesamiento previo de la imagen antes de que sea clasificada por el modelo.

```
1.  import argparse
2.  import time
3.
4.  import numpy as np
5.  from PIL import Image
6.  import tensorflow as tf
7.
8.
9.  def load_labels(filename):
10.   with open(filename, 'r') as f:
11.     return [line.strip() for line in f.readlines()]
12.
13.
14. if __name__ == '__main__':
15.   parser = argparse.ArgumentParser()
16.   parser.add_argument(
17.       '-n',
18.       '--name',
19.       default='',
20.       help='Model name used to preprocess info')
21.   parser.add_argument(
22.       '-i',
23.       '--image',
24.       default='/tmp/grace_hopper.bmp',
25.       help='image to be classified')
26.   parser.add_argument(
27.       '-m',
28.       '--model_file',
29.       default='/tmp/mobilenet_v1_1.0_224_quant.tflite',
30.       help='.tflite model to be executed')
31.   parser.add_argument(
32.       '-l',
33.       '--label_file',
34.       default='/tmp/labels.txt',
35.       help='name of file containing labels')
36.   parser.add_argument(
37.       '--input_mean',
38.       default=127.5, type=float,
39.       help='input_mean')
40.   parser.add_argument(
41.       '--input_std',
42.       default=127.5, type=float,
43.       help='input standard deviation')
44.   parser.add_argument(
45.       '--num_threads', default=None, type=int, help='number of threads')
46.   args = parser.parse_args()
47.
48.   interpreter = tf.lite.Interpreter(
49.       model_path=args.model_file, num_threads=args.num_threads)
50.   interpreter.allocate_tensors()
51.
52.   input_details = interpreter.get_input_details()
53.   output_details = interpreter.get_output_details()
54.
55.   # check the type of the input tensor
56.   floating_model = input_details[0]['dtype'] == np.float32
57.
58.   # NxHxWxC, H:1, W:2
59.   height = input_details[0]['shape'][1]
60.   width = input_details[0]['shape'][2]
61.   img = Image.open(args.image).resize((width, height))
```

```
62.
63.   # add N dim
64.   input_data = np.expand_dims(img, axis=0)
65.
66.    # Get preprocess function
67.   preprocess_input = lambda x : x
68.   if args.name == 'vgg16':
69.     preprocess_input = lambda x : tf.keras.applications.vgg16.preprocess_input(x)
70.   elif args.name == 'resnet50':
71.     preprocess_input = lambda x :
    tf.keras.applications.resnet50.preprocess_input(x)
72.   elif args.name == 'mobilenet':
73.     preprocess_input = lambda x :
    tf.keras.applications.mobilenet.preprocess_input(x)
74.   elif args.name == 'efficientnet':
75.     # Por defecto la funcion de preprocesado no hace nada
76.     # preprocess_input = lambda x :
    tf.keras.applications.efficientnet.preprocess_input(x)
77.     if input_data.dtype != input_details[0]['dtype']:
78.       preprocess_input = lambda x : x.astype(input_details[0]['dtype'])
79.   else:
80.     if floating_model:
81.       input_data = (np.float32(input_data) - args.input_mean) / args.input_std
82.
83.   input_data = preprocess_input(input_data)
84.
85.   interpreter.set_tensor(input_details[0]['index'], input_data)
86.
87.   start_time = time.time()
88.   interpreter.invoke()
89.   stop_time = time.time()
90.
91.   output_data = interpreter.get_tensor(output_details[0]['index'])
92.   results = np.squeeze(output_data)
93.
94.   top_k = results.argsort()[-5:][::-1]
95.   labels = load_labels(args.label_file)
96.   for i in top_k:
97.     print('{:08.6f}: {}'.format(float(results[i]), labels[i]))
98.
99.   print('time: {:.3f}ms'.format((stop_time - start_time) * 1000))
```

Código C-1: Programa en Python para clasificar una imagen con un modelo TensorFlow Lite

Suponiendo que el Código C-1 estuviera guardado sobre un archivo nombrado **label_image.py** para usarlo bastaría con lanzar el comando del Código C-2. El cual cuenta con los siguientes parámetros:

- **--name**. Tipo de preprocesamiento a usar. Hay implementados 4 tipos, los cuales se pueden llamar mediante **vgg16**, **resnet50**, **mobilenet**, **efficientnet**. Si por defecto no se usa este parámetro, en caso de que el modelo soporte entrada de tipo flotante, se hará el preprocesamiento por defecto que consiste en que a cada valor se le reste la media establecida en **–input_mean** y se le divida entre la desviación estándar establecida en **–input_std**.
- **--input_mean**. El valor de media a restar en caso de usar la normalización más sencilla.
- **--input_std**. El valor de la desviación por la cual se dividirán los valores en caso de usar la normalización más sencilla.
- **--image**. Dirección de la imagen a clasificar.
- **--model_file**. Dirección del modelo usado para la clasificación.
- **--label_file**. Dirección del archivo de etiquetado del modelo, que contiene en cada fila el nombre de la clase que le correspondería a la neurona de salida con mismo índice.

```
1.  (venv) Ubuntu:~$ python3 label_image.py --model_file tflite_models/resnet50.tflite
    --label_file imagenet_labels_1000.txt --image images/ILSVRC2012_val_00000001.JPEG
    --name resnet50
2.  0.507477: sea snake
3.  0.219672: rock python
4.  0.124925: hognose snake
5.  0.073309: water snake
6.  0.026637: night snake
7.  time: 647.621ms
```

Código C-2: Ejemplo de ejecución del programa para clasificar una imagen con un modelo TensorFlow Lite

D Medición de accuracy de un modelo TensorFlow Lite

Gracias al script del Código D-1 se pude calcular el Top-1 y Top5 *accuracy* de cualquier modelo seleccionando el tipo de preprocesamiento deseado, el directorio con el conjunto de imágenes a clasificar y un listado con las salidas objetivo de las imágenes en cuestión.

Por el momento solo se puede usar como preprocesamiento, el establecido por defecto por Keras para VGG16, ResNet50, EfficientNet y MobileNet. Pero al estar programado en Python es muy fácil añadir nuevos tipos de preprocesamiento, lo cual se haría entre las líneas 60 y 72.

Como detalle a destacar, es que se ha añadido un array que guarda el índice de aquellas imágenes que no han entrado ni en el Top-1 ni en el Top-5, por si se desea hacer algo con ellas, como usarlas de nuevo para ampliar el conocimiento de la red, o simplemente tenerlas en cuenta para aplicarlas cualquier tipo de filtro.

```
1.  import argparse
2.  import time
3.  import os
4.
5.  import numpy as np
6.  from PIL import Image
7.  import tensorflow as tf
8.
9.
10. def load_labels(filename):
11.   with open(filename, 'r') as f:
12.     return [line.strip() for line in f.readlines()]
13.
14.
15. if __name__ == '__main__':
16.   parser = argparse.ArgumentParser()
17.   parser.add_argument(
18.       '-n',
19.       '--name',
20.       default='',
21.       help='Model name used to preprocess info: vgg16, resnet50, mobilenet,
    efficientnet')
22.   parser.add_argument(
23.       '-g',
24.       '--ground_truth_labels',
25.       default='/tmp/ground_truth_labels.txt',
26.       help='file with ground truth labels of images to be classified')
27.   parser.add_argument(
28.       '-d',
29.       '--directory',
30.       default='/tmp/grace_hopper.bmp',
31.       help='directory with images to be classified')
32.   parser.add_argument(
33.       '-m',
34.       '--model_file',
35.       default='/tmp/mobilenet_v1_1.0_224_quant.tflite',
36.       help='.tflite model to be executed')
37.   parser.add_argument(
38.       '-l',
39.       '--label_file',
40.       default='/tmp/labels.txt',
41.       help='name of file containing labels')
42.   parser.add_argument(
43.       '--num_threads', default=None, type=int, help='number of threads')
44.   args = parser.parse_args()
45.
46.   interpreter = tf.lite.Interpreter(
47.       model_path=args.model_file, num_threads=args.num_threads)
48.   interpreter.allocate_tensors()
49.
```

```
  input_details = interpreter.get_input_details()
  output_details = interpreter.get_output_details()

  # check the type of the input tensor
  floating_model = input_details[0]['dtype'] == np.float32

  # NxHxWxC, H:1, W:2
  height = input_details[0]['shape'][1]
  width = input_details[0]['shape'][2]

  # Get preprocess function
  preprocess_input = lambda x : x
  if args.name == 'vgg16':
    preprocess_input = lambda x : tf.keras.applications.vgg16.preprocess_input(x)
  elif args.name == 'resnet50':
    preprocess_input = lambda x :
tf.keras.applications.resnet50.preprocess_input(x)
  elif args.name == 'mobilenet':
    preprocess_input = lambda x :
tf.keras.applications.mobilenet.preprocess_input(x)
  elif args.name == 'efficientnet':
    # Por defecto la funcion de preprocesado no hace nada
    # preprocess_input = lambda x :
tf.keras.applications.efficientnet.preprocess_input(x)
    preprocess_input = lambda x : x.astype(input_details[0]['dtype'])

  # Get labels
  labels = load_labels(args.label_file)
  ground_truth_labels = load_labels(args.ground_truth_labels)

  # Get number of images inside folder
  n_images = len(ground_truth_labels)

  # Some values to measure
  top_1 = 0
  top_5 = 0
  fail = []
  for image, i in zip(os.listdir(args.directory), range(n_images)):
    img = Image.open(args.directory + image).resize((width, height))
    if img.mode != 'RGB':
      img = img.convert('RGB')

    # add N dim
    input_data = np.expand_dims(img, axis=0)

    # Preprocess input with the specify function
    input_data = preprocess_input(input_data)

    # Check if input_data has same dtype as supposed. Transfromed if needed
    # if input_data.dtype != input_details[0]['dtype']:
    #   input_data = input_data.astype(input_details[0]['dtype'])

    interpreter.set_tensor(input_details[0]['index'], input_data)

      interpreter.invoke()

      output_data = interpreter.get_tensor(output_details[0]['index'])
      results = np.squeeze(output_data)

      top_k = results.argsort()[-5:][::-1]
      token = False
      for j, k in zip(top_k, range(5)):
        if labels[j] == ground_truth_labels[i]:
          token = True
          top_5 += 1
          if k == 0:
            top_1 += 1
          break
      if token == False:
        fail.append(i)
```

```
117.
118.     print('Top-1 Accuracy: {:.4f}'.format(top_1 / n_images))
119.     print('Top-5 Accuracy: {:.4f}'.format(top_5 / n_images))
120.     print(fail)
```

Código D-1: Programa en Python para clasificar el conjunto de imágenes contenidas en una carpeta con un modelo TensorFlow Lite y medir su *accuracy*

Suponiendo que el Código D-1 estuviera guardado sobre un archivo llamado **label_directory.py** para usarlo bastaría con lanzar el comando de la primera línea del Código D-2. El cual cuenta con los siguientes parámetros:

- **--name**. Tipo de preprocesamiento a usar. Hay implementados 4 tipos, los cuales se pueden llamar mediante **vgg16**, **resnet50**, **mobilenet**, **efficientnet**. Si no se usa este parámetro o el nombre no coincide con los preestablecidos, no se usará ningún tipo de preprocesamiento.
- **--directory**. Dirección del directorio con las imágenes a clasificar.
- **--model_file**. Dirección del modelo usado para la clasificación.
- **--label_file**. Dirección del archivo de etiquetado del modelo, que contiene en cada fila el nombre de la clase que le correspondería a la neurona de salida con mismo índice.
- **--ground_truth_labels**. Dirección del archivo que contiene las salidas objetivo de las imágenes a clasificar. Cada línea contiene la salida objetivo de la imagen con el mismo índice.

```
1.  python3 label_directory.py --name mobilenet --model_file
    tflite_models/mobilenet.tflite --label_file imagenet_labels_1000.txt --directory
    images/ --ground_truth_labels validation_labels_500.txt
2.  Top-1 Accuracy: 0.7040
3.  Top-5 Accuracy: 0.8940
4.  [5, 7, 16, 21, 36, 49, 61, 69, 70, 118, 120, 123, 127, 135, 136, 139, 149, 155,
    160, 161, 175, 191, 203, 204, 227, 231, 235, 238, 243, 254, 259, 268, 269, 278,
    285, 291, 295, 319, 330, 333, 338, 344, 359, 379, 384, 397, 399, 431, 432, 436,
    441, 468, 491]
```

Código D-2: Ejemplo de ejecución del programa para clasificar el conjunto de imágenes contenidas en una carpeta con un modelo TensorFlow Lite y medir su *accuracy*