UNIVERSIDAD AUTÓNOMA DE MADRID
ESCUELA POLITÉCNICA SUPERIOR

TRABAJO DE FIN DE MASTER

Desarrollo de aplicaciones móviles de clasificación y detección de objetos a partir de redes convolucionales ligeras

Máster Universitario en Ingeniería de Telecomunicación

Autor: Casa Robles, Paulo Cesar
Tutor: Carballeira López, Pablo
Ponente: Martínez Sánchez, José María

Julio, 2020

Desarrollo de aplicaciones móviles de clasificación y detección de objetos a partir de redes convolucionales ligeras

Autor: Casa Robles, Paulo Cesar

Tutor: Carballeira López, Pablo

Ponente: Martínez Sánchez, José María

Video Processing and Understanding Lab
Departamento de Tecnología Electrónica y de las Comunicaciones
Escuela Politécnica Superior
Universidad Autónoma de Madrid

Julio 2020

Trabajo parcialmente financiado por el Ministerio de Ciencia, Innovación y Universidades del Gobierno de España bajo el proyecto TEC2017-88169-R (MobiNetVideo)

Resumen

En los últimos años, la visión artifical ha evolucionado rápidamente debido al desarrollo de las redes neuronales convolucionales, y así lo demuestran las numerosas publicaciones e investigaciones de la comunidad científica en varias de las posibles aplicaciones que ésta puede tener, como por ejemplo: la clasificación, detección, segmentación, o reconocimiento de ciertos objetos. Además, por otro lado, también es conocida la alta demanda que tienen las aplicaciones móviles en estos momentos. Por lo tanto, el presente Trabajo Fin de Master (TFM) se presenta con la inteción de unir ambas materias, concretamente con el desarrollo de aplicaciones móviles en las tareas de clasificación y detección de objetos.

Sin embargo, integrar las tareas de la visión artificial en un dispositivo móvil supone un problema complejo de gran interés. Esto, se soluciona con las redes convolucionales ligeras ya que poseen ciertas características, de entre las que destacan, la eficiencia en memoria y precisión del modelo, siendo propiedades necesarias que demanda una aplicación en un dispositivo móvil para su correcto funcionamiento. Para ello, ha sido necesario explorar las diferentes alternativas que el estado del arte nos ofrece para incoroporar modelos preentrenados de redes convolucionales ligeras en dispositivos móviles. Una de las plataformas de deep learning que se encuentra actualmente en constante desarrollo y que tendrá un papel fundamental para la integración de dichos modelos será TensorFlow Lite.

Así, este TFM presenta las técnicas y configuraciones necesarias para convertir los modelos a formato TensorFlow Lite y posteriormente permitir la inserción de los modelos a dipositivos móviles con sistema operativo Android gracias a herramientas de desarrolladores. Finalmente, este documento aporta un estudio comparativo de las tareas de clasificación y detección, proporcionando conclusiones del rendimiento de las diferentes redes convolucionales integradas en términos de eficiencia y complejidad computacional.

Palabras clave

Visión Artificial, Aplicación, Dispositivo Móvil, Clasificación, Detección de Objetos, Redes Neuronales Ligeras, TensorFlow Lite.

Abstract

In recent years, the artificial vision has evolved rapidly due to the development of convolutional neural networks, and this is demonstrated by the numerous publications and research of the scientific community in several of the possible applications that it may have, such as: the classification, detection, segmentation, or recognition of certain objects. In addition, on the other hand, it is also known the high demand that mobile applications have at this moment. Therefore, this Final Master Project is presented with the intention of uniting both subjects, specifically with the development of mobile applications in the tasks of classification and object detection.

However, integrating machine vision tasks into a mobile device is a complex problem of great interest. This is solved with light convolutional networks since they have certain characteristics, among which stand out, the memory efficiency and model precision, being necessary properties that an application demands on a mobile device for its correct operation. For this, it has been necessary to explore the different alternatives that the state of the art offers us to incorporate pre-trained models of light convolutional networks into mobile devices. One of the deep learning platforms that is currently in constant development and that will have a fundamental role in the integration of these models will be TensorFlow Lite.

Thus, this TFM presents the necessary techniques and configurations to convert the models into TensorFlow Lite format and later allow the insertion of the models into mobile devices with Android operating system using developer tools. Finally, this document contributes a comparative study of the classification and detection tasks, providing conclusions of the different integrated convolutional networks performances in terms of efficiency and computational complexity.

Keywords

Artificial Vision, Aplication, Mobile Device, Classification, Object Detection, Light Convolutional Networks, TensorFlow Lite.

Agradecimientos

En primer lugar, me gustaría agradecer de corazón el apoyo incondicional que me ha brindado mi familia, siendo un pilar fundamental en mi vida. Muy especialmente, a mis padres, Paulo y Ximena, los cuales siempre han tenido palabras de aliento y de confianza. Por lo tanto, les quiero dedicar a ellos este pequeño paso en mi carrera profesional.

También, agradecer a mi tutor de TFM, Pablo, por la ayuda y apoyo recibido durante el desarrollo del proyecto, incluso en periodos diferentes e inusuales para hacer un seguimiento. Así mismo, a mi ponente Chema.

Finalmente, recordar a todas las personas que he conocido y han formado parte de mi vida, desde mi pareja y amigos, hasta mis compañeros de máster.

Gracias.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

1.1 Motivación

En la actualidad, las redes neuronales han revolucionado muchas áreas de la inteligencia artificial (IA) [13], teniendo impacto en una de las técnicas más clásicas de la IA como lo es el campo de la visión artificial. Siguiendo esta influencia, la motivacion general del trabajo parte de la aspiración en incorporar tareas de la visión artificial en aplicaciones móviles. De entre dichas tareas, se encuentran dos de los problemas más fundamentales y desafiantes dentro de la visión artificial como son la clasificación y detección de objetos [14]. Este hecho se da entre otros, por el continuo progreso de las redes neuronales convolucionales (CNNs, por sus siglas en inglés), convirtiéndose en el enfoque dominante del aprendizaje automático para el reconocimiento de objetos visuales, lo cual, ha impulsado en los últimos años a la mejora de dichas tareas. Aunque originalmente las CNNs se introdujeron hace más de 30 años, la tendencia de los modelos ha sido crear redes más profundas, grandes y complejas, añadiendo cada vez más diferentes tipos de capas (p.ej.,convolucional, pooling, fully connected) a la red de forma generalizada [15]. Como muestra, de entre las redes más conocidas, LeNet-5 [16] constaba de 5 capas, VGG presentaba hasta 19 [17], y Redes Residuales (ResNets) llegaron a superar las 100 capas [2].

Sin embargo, a medida que los modelos CNN modernos se vuelven cada vez más profundos y el número de parametros va incrementado, los resultados de su rendimiento son muy buenos en cuánto a la precisión tanto de clasificación como de detección de objetos, pero a costa de un coste computacional muy alto como son los requisitos de almacenamiento y la memoria de las redes [15] [18], haciendo que sea necesario en algunos casos, el uso de unidades de procesamiento gráfico (GPUs) con memorias de varios gigabytes para que la carga de trabajo durante el entrenamiento de las redes sea viable.

Tales aumentos en las demandas computacionales hacen que no sea posible en ciertos casos implementar dichos modelos CNN en plataformas con recursos limitados, como pueden ser los dispositivos móviles. Y es que, el diseño de redes neuronales convolucionales para dispositivos móviles es un gran desafío debido a que los modelos deben deben empezar a reducir el número de parámetros, así como los requisitos de memoria y computo para su correcta ejecución, pero aun así manteniéndose precisos [19]. Por lo tanto, una dirección de investigación durante estos últimos años, ha sido plantear modelos, los cuales, además de tener en cuenta la calidad de predicción, así mismo tengan en cuenta el costo computacional para reducirlo. Por este motivo, recientemente se han propuesto nuevos modelos con arquitecturas diferentes a las ya creadas

con el fin de obtener redes convolucionales ligeras [19] [20]. De entre dichas redes convolucionales ligeras, se ofrecen modelos con unas arquitecturas de red eficientes en memoria que pueden adaptarse a los requisitos necesarios para su implementación en dispositivos móviles o con capacidades limitadas [21] [22] [12] .

1.2 Objetivos

Este TFM se desarrolla en el contexto de las redes convolucionales de baja complejidad. En donde uno de los principales objetivos será el desarrollo de ejemplos de aplicaciones de clasificación y detección de objetos que permitan la prueba de diferentes redes ligeras en dispositivos móviles con sistemas operativos Android, mediante la integración de herramientas existentes. Además, se realizará una evaluación del rendimiento de estas redes en términos de eficiencia de clasificación/detección y tiempo de proceso/coste computacional.

Para ello, el marco general del trabajo será:

- Implementar aplicaciones de clasificación y detección permitiendo integrar diferentes modelos de redes en dispositivos móviles utilizando herramientas existentes y que se encuentran en estado inicial de desarrollo.
- Analizar el rendimiento de diferentes redes ligeras entre sí, además de frente a redes más complejas en términos de eficiencia para tareas de clasificación y detección de objetos. Separándolo en:
  - Por un lado, el rendimiento de clasificación o detección, utilizando métricas y bases de datos existentes en el estado del arte.
  - Por otro lado, el rendimiento de la carga computacional evaluando el tiempo de procesado requerido por las diferentes redes.

Todo ello se realizará con entornos de trabajo como *Android Studio* [23] para desarrollar las aplicaciones o *Bazel* [24] para la conversión de modelos al formato necesario. Así como, herramientas que se encuentran en desarrollo inicial y en constante evolución con bibliotecas de código abierto para el aprendizaje automático como *TensorFlow* [25], su versión más ligera *TensorFlow Lite* [26] la cual permite la implementación de modelos en dispositivos móviles, y APIs de redes neuronales de alto nivel escrita en python como *Keras* [27].

1.3 Estructura del documento

El presente documento está dividido por seis capítulos. A continuación, se comenta la estructura e información que se puede encontrar en cada uno de ellos.

- Capítulo 1. **Introducción**. En el primer capítulo se puede observar la motivación y los objetivos para el desarrollo del proyecto.
- Capítulo 2. **Estado del Arte**. Se introduce al lector de forma teórica sobre la visión artificial, el aprendizaje profundo y los estudios de las redes neuronales convolucionales, además de otorgar una visión general de las arquitecturas ligeras.

- Capítulo 3. **Marcos de Implementación**. En este capítulo se comenta de que manera se utilizan y aportan al proyecto los entornos de desarrollo utilizados.
- Capítulo 4. **Desarrollo**. El objetivo de este capítulo es exponer los pasos y la funcionalidad que tiene el marco del desarrollo del trabajo.
- Capítulo 5. **Evaluación**. Se aportan y exponen las pruebas de evaluación realizadas en las tareas del proyecto, además de un análisis de los mismos.
- Capítulo 6. **Conclusiones y Trabajo Futuro**. El último capítulo aporta las conclusiones del trabajo realizado. También se plantean nuevas líneas de trabajo futuras y posibilidades de mejora.

Capítulo 2

Estado del arte

El estado del arte de este trabajo expone por un lado, una descripción de la visión artificial unido al uso del campo del aprendizaje automático para comprender la función de ambos algoritmos juntos. Más adelante, se trata el aprendizaje profundo junto con las redes neuronales y cómo su evolución en redes neuronales convolucionales (CNNs) permite que su arquitectura esté directamente adaptada a la señal que se está procesando (en este caso imágenes ya que las tareas que nos ocupan en este proyecto son la de clasificación y detección de objetos). En paralelo se observa como el uso de gpus, desarrolladas para el tratamiento de imágenes o gráficos (muchas operaciones en paralelo) ayuda a que el entrenamiento y procesado con CNNs sea viable, ya que las redes tendían a arquitecturas cada vez más profundas. Finalmente, continuaremos conociendo cual es la visión general que se tiene cuando se habla de arquitecturas ligeras y cuáles son esas posibles estructuras que los modelos pueden tener.

2.1 Visión artificial

La visión artificial nace como una disciplina científica, la cual, trabaja para proporcionar modelos computacionales del sistema visual humano teniendo como objetivo construir sistemas autónomos que sean capaces de imitar algunas de las tareas que el sistema visual puede realizar e incluso tratar de superarlas [28], destacando la clasificación, detección, segmentación, o el reconocimiento de ciertos objetos. Para ello en los últimos años, el uso del aprendizaje automático o machine learning (rama de la inteligencia artificial) ha potenciado la disciplina en cuestión. Este hecho se da gracias al estudio de modelos y algoritmos de aprendizaje automáticos, los cuales otorgan a las máquinas la capacidad de mejora progresiva en el desempeño de varias tareas. De esta forma, y a partir de un conjunto de datos de entrada, es posible originar un resultado que generalmente se traducen en diversas predicciones o decisiones [29].

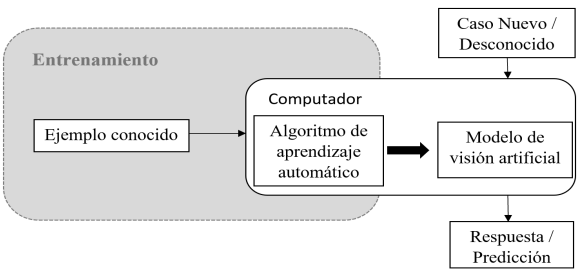


Figure 2.1: Esquema Aprendizaje Automático en Visión Artificial

El desempeño de una tarea se mide por cierto rendimiento, por lo tanto, se podría decir que un modelo aprende de una experiencia, si dicho desempeño mejora la experiencia de la tarea.

Por otro lado, en lo que respecto al proceso de aprendizaje del modelo, se conoce que los algoritmos se pueden clasificar principalmente en dos diferentes tipos basándose en la existencia o la clase de retroalimentación que se da durante el entrenamiento [29]:

- Aprendizaje supervisado: Este aprendizaje necesita realizar previamente una preparación que consiste en la etiquetación de un conjuntado de datos de entrada, con la finalidad en este caso, de que el modelo de visión artificial aprenda las características necesarias para la consecución de la tarea. Mediante este proceso, el modelo va modificando los parámetros con el fin de adaptarse y ofrecer mejor rendimiento.
- Aprendizaje no supervisado: A diferencia del aprendizaje anterior, se reciben un conjunto de datos de entrada no etiquetados, por lo tanto, el modelo trata de aprender de estos datos a partir de la exploración de patrones exitentes en ellos.

Con lo visto anteriormente y sabiendo que la visión artificial abarca diferentes temas, se especifica que en este proyecto se trabajará con el aprendizaje supervisado y en dos tareas concretas, que son la clasificación y la detección de objetos [30]. Dentro de las cuales y con la evolución en ambas tareas, también se puede hablar de una clasificación y detección de objetos en tiempo real:

- Clasificación: Es la tarea de asignar a una imagen de entrada, una etiqueta dado un conjunto fijo de categorías. Por lo tanto, se puede decir que es el proceso en el cual, la salida especifica la posible clase predicha a la que puede pertenecer el objeto de la imagen de entrada [31].
- Detección de objetos: Es el proceso de extraer los objetos relevantes de una imagen y su ubicación. Esto significa que la detección de objetos no es solo una tarea de clasificación, sino también una tarea de regresión. De hecho, el resultado final es una etiqueta más una tupla de números que identifican la posición de cada objeto en la imagen (x0, y0, ancho, alto). Pudiendo dicha detección final extenderse desde una a varias clases de objetos [30].
- Clasificación y detección de objetos a tiempo real: Es necesario que el algoritmo para desarrollar en tiempo real tenga una complejidad computacional menor, con el fin de menorar la latencia de respuesta en clasificación o detección.

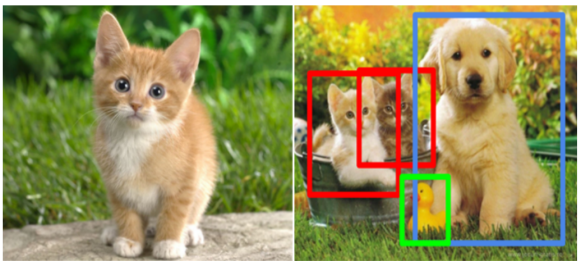


Figure 2.2: Tareas de Clasificación y Detección sobre imágenes

2.2 Aprendizaje Profundo

Los algoritmos más clásicos y anteriores de machine learning tienen un límite de rendimiento debido a que las especificaciones computacionales no permitían desarrollar modelos más complejos. Por lo tanto, con el propósito de superar este límite, se introducen técnicas más potentes, aumentando el rendimiento entre el aprendizaje profundo moderno y los algoritmos más clásicos.

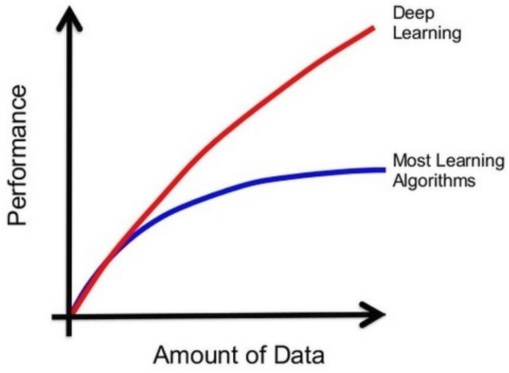


Figure 2.3: Rendimiento comparativo aprendizajes profundo y algoritmos más clásicos [1]

La idea básica de este aprendizaje profundo (subconjunto del aprendizaje automático) es la profundidad de las estructuras de las redes neuronales, permitiendo que los modelos computacionales se compongan de múltiples capas de procesamiento y aprendan representaciones de datos con varios niveles de abstracción. Por lo tanto, estos algoritmos permiten a la computadora construir conceptos complejos a partir de otros más simples [31].

En el aprendizaje profundo se encuentran las redes neuronales artificiales (ANN), y como dos subconjuntos de ésta se tienen a las redes neuronales convolucionales (CNN) y redes neuronales recurrentes (RNN). Las redes neuronales convolucionales han producido avances en el procesamiento de imágenes, video, voz y audio, mientras que las redes recurrentes han generado progreso en los datos secuenciales como el texto. Como se ha comentado, estas arquitecturas son utilizadas en varios campos, y uno de ellos, el cual nos concierne en este trabajo, es la visión artificial donde las técnicas basadas en aprendizaje profundo se han aplicado con éxito [3], es por esto que en este apartado se presentan dichos conceptos.

2.2.1 Redes Neuronales

Una red neuronal artificial es un conjunto de elementos simples o nodos, cuya funcionalidad está basada e inspirada en el funcionamiento de las neuronas y conexiones del cerebro que sirven para procesar información [32]. Por lo tanto, la finalidad de las redes neuronales artificiales, es obtener un modelo computacional fundamentado en la reproducción aproximada del comportamiento en el proceso de decisión de las neuronas del sistema nervioso central. Así, la neurona se posiciona como el elemento básico dentro de esta red, cuya estructura del modelo matemático se puede observar en la Figura 2.4.

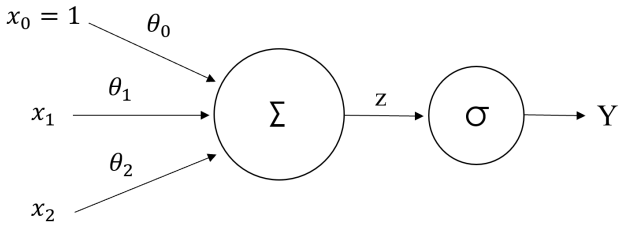


Figure 2.4: Neurona artificial

A partir de ella, podemos extraer que la neurona recibe la entrada (X_i) del exterior o de otras neuronas de la red y computa la salida (Z) de la siguiente forma $Z = \sum_{i=1}^{N}(x_i\theta_i) + \theta_0$. Por lo que, cada entrada se encuentra asociada a un peso (θ_i), los cuales asignan la importancia que toma para esta neurona cada una de las entradas. Mientras que la salida Y permiten que las redes aprendan a funcionar como sistemas no lineales, ya que depende de la función de activación (σ) que se decida emplear, obteniendo una salida acotada y quedando de la siguiente manera: $Y = \sigma(Z)$. Dentro de la definición de la neurona, cabe mencionar que en algunas situaciones el parámetro θ_0 es omitido por simplificación, pero es necesario a la hora de evitar interrupciones en el transcurso del aprendizaje cuando las entradas X_i son nulas.

Por su parte, la función de activación calcula el estado de actividad de la neurona a partir del resultado, obteniendo un estado de activación con un rango normalmente entre (-1; 1) o (0; 1). Algunos ejemplos de función de activación se pueden observar en la siguiente Figura 2.5.

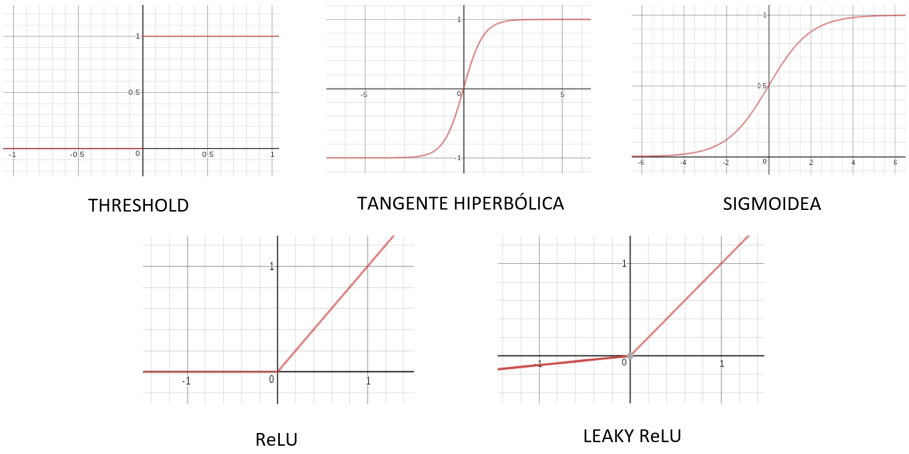


Figure 2.5: Funciones de activación

Cuando las neuronas artificiales se estratifican, crean una red de neuronas artificiales (ANN), pudiendo distribuirse dentro de la red en diferentes capas o niveles, con cierto número determinado de neuronas en cada capa. Dichas capas se puede clasificar en una capa de entrada, capas intermedias y capa de salida [33].

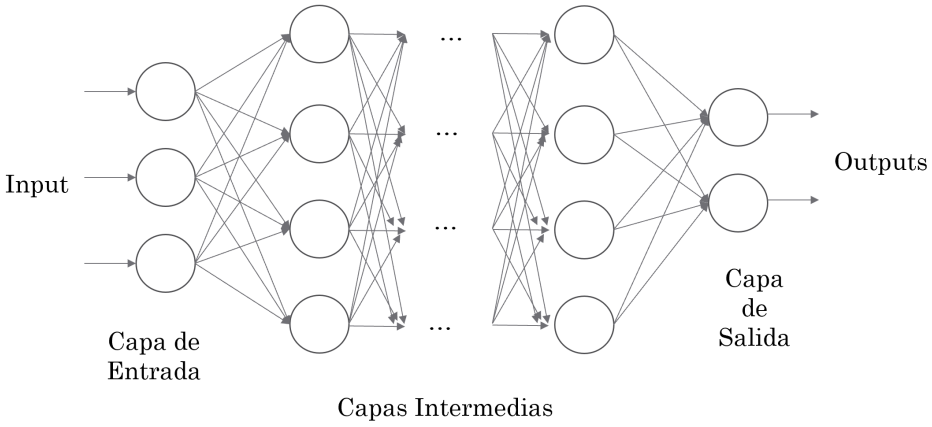


Figure 2.6: Esquema red neuronal artificial

En la anterior (Figura 2.6), vemos el esquema de una red neuronal artificial, en concreto es del tipo *feedforward* , donde cada neurona de una capa solo tiene conexión directa con las neuronas de la siguiente capa, en dirección hacia la capa de salida [32]. La capacidad de procesamiento de la red reside en la fuerzas de conexión entre unidades, o peso, obtenido por un proceso de aprendizaje o adaptación sobre un conjunto de datos etiquetados. Puede ser que el número de neuronas de entrada sea igual al número de variables del conjunto de datos, y el número de neuronas de salida igual al número de símbolos necesarios para representar la salida deseada. Por ejemplo, si se trata de un clasificador, el número de salidas será igual al número de clases existentes.

Las redes con un uso elevado de capas ocultas se denominan redes profundas. En función de la problemática a abordar, puede ser más o menos conveniente hacer uso de dichas redes más profundas. A medida que la profundidad aumenta, una red residual prodría contener ciertas ventajas debido al tipo de conexión que hay, existiendo conexiones directas entre capas no consecutivas, siempre que la relación de las conexiones sea de capa anterior a capa posterior, como se muestra en la Figura 2.7, donde se aprecia como la salida de la red "omite" capas [2].

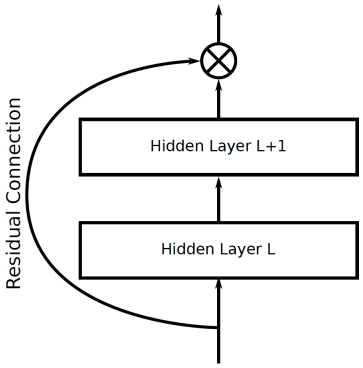


Figure 2.7: Ejemplo conexión residual [2]

2.2.2 Entrenando la red

Cuando entrenamos a una red, lo que estamos haciendo formalmente es adaptar una red a nuestro conjunto de entrenamiento (datos etiquetados), para así tratar de encontrar la mejor manera de ajustar la información a una línea de regresión u otro modelo matemático. El conjunto de capacitación es la información de muestra que creemos que es lo suficientemente representativa como para que nuestra red pueda aprender de ella. Al entrenar, queremos que la red funcione bien en una tarea dada sobre la base de información no vista. Durante este entrenamiento, la red aprende modificando los pesos de las conexiones entre las distintas neuronas utilizando las etiquetas asociadas a los datos de entrenamiento. El núcleo de los algoritmos de aprendizaje profundo consiste en encontrar los pesos correctos en los bordes de la red para minimizar la suma de los errores de las neuronas de salida. Para lograr este objetivo es necesario establecer:

- La función de activación a usar en cada neurona.
- La función de perdida a utilizar para calcular la suma de errores. Para ello, en el aprendizaje se define una función de costes, donde se calcule el error entre el valor proporcionado por la red y el valor correcto previamente etiquetado.
- El algoritmo a emplear con el fin de obtener los correctos pesos de la red. Una de las técnicas comúnmente más usada es la optimización por descenso de gradiente.

Así, el entrenamiento puede entenderse como un problema de optimización, por ello es necesario definir la función a optimizar, ya que los diferentes parámetros que existen en una red neuronal han de ser ajustados para aproximar con mayor precisión la función que se desea modelar.

El número de parámetros de la red en arquitecturas profundas es considerable, por lo que, surge una necesidad de que las bases de datos sean lo suficientemente grandes para evitar inconvenientes. Dentro de esta bases de datos, los datos de entrenamiento se usan para ajustar los pesos de la red, mientras que los de validación miden el rendimiento de la red en datos no vistos durante la fase de entrenamiento. Cuando no se está entrenando lo suficiente como para hacer esto, se llama *underfitting*, lo que significa que la red no aprendió lo suficiente del conjunto de entrenamiento. Lo opuesto a esto, se llama *overfitting*, donde la red aprende el conjunto de entrenamiento tan bien que no puede aplicarse efectivamente a datos que no ha visto antes (datos de validación). Estos conceptos se pueden entender mejor haciendo referencia a la Figura 2.8.

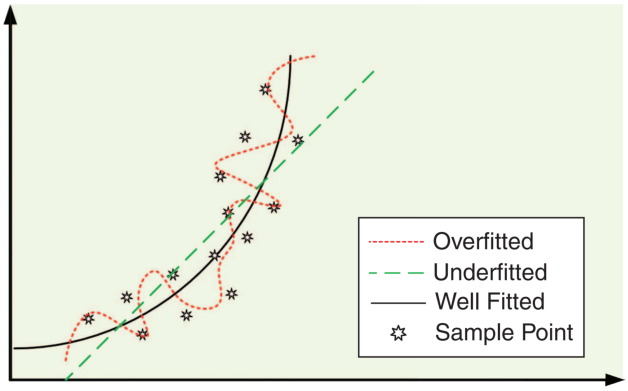


Figure 2.8: Ejemplo overfitting y underfitting sobre datos bidimensionales [3]

Si el modelo está *underfitting*, podemos aumentar el número o el tamaño de los parámetros o mejorar el modelo. Si un modelo está *overfitting*, podemos reducir el tamaño del modelo o aplicar otras técnicas. Un desafío clave es identificar con precisión si el modelo está cerca del ajuste óptimo. Esta es otra de las razones por las que se necesitan conjuntos de datos muy grandes, para lograr soluciones de trabajo en estos problemas.

2.2.3 Uso de unidades de procesamiento gráfico

La GPU es un coprocesador dedicado al procesamiento de imágenes y operaciones de coma flotante, ofreciendo la posibilidad de manejar múltiples tareas simultáneamente gracias en gran parte a su arquitectura paralela de miles de núcleos. En este sentido, tener una GPU se considera muy necesario a la hora de procesar los grandes conjuntos de datos que se manejan en el aprendizaje profundo y así disminuir de forma considerable el tiempo del entrenamiento de los modelos, permitiendo entrenar redes 10 o 20 veces más rápido [34], haciendo que sea posible experimentar y probar con diferentes diseños de modelos y parámetro en un periodo de tiempo más corto. Así, el empleo de las unidades de procesamiento gráfico o GPUs en el entrenamiento de los modelos es uno de los componentes más influyentes en el progreso del aprendizaje profundo ya que hacen viable el uso de arquitecturas profundas actuales.

2.3 Redes Neuronales Convolucionales

Las redes neuronales convolucionales (CNNs o ConvNets) son un tipo concreto de redes neuronales artificiales, las cuales toman este nombre dado que realizan una operación matemática lineal llamada convolución [31] y tratan de imitar a las neuronas en la corteza visual de un cerebro biológico, demostrando ser muy efectivas en tareas de visión artificial y han superado a otros modelos de machine learning [35]. Estas redes, son un modelo especializado de red neuronal para procesar datos ordenados en forma de cuadrícula, sobre todo en procesar imágenes. La idea subyacente de la CNN se basa en el trabajo previo en el reconocimiento de patrones visuales, donde se ha demostrado que es útil extraer y combinar características locales con características de orden superior más abstractas [36]. En la siguiente figura se muestra una CNN para la tarea de clasificación.

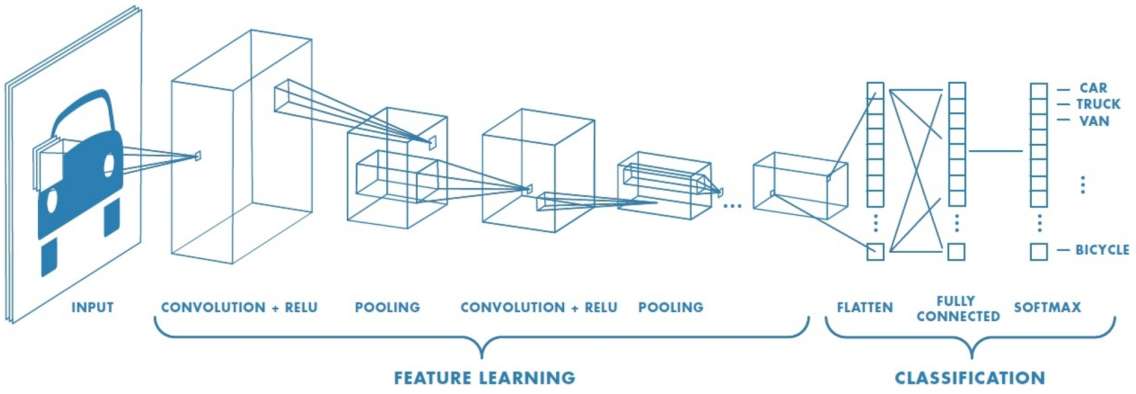


Figure 2.9: Ejemplo red neuronal convolucional [4]

Estas redes neuronales poseen la operación de convolución, la cual obteniene una reducción en el número de conexiones entre la capa de entrada y la capa oculta. Esta convolución se realiza esencialmente a partir de un filtro (también llamado kernel) que se pasa sobre cada área de la imagen de entrada de la CNN. Una imagen es una matriz multidimensional, donde para una imagen RGB estándar, tenemos una profundidad de tres: un canal para cada uno de los canales Rojo, Verde y Azul, respectivamente. Dado este conocimiento, podemos pensar en una imagen como una matriz grande y un kernel o matriz convolucional como una matriz pequeña que se utiliza para desenfocar, enfocar, detectar bordes y otras funciones de procesamiento. Esencialmente, este pequeño núcleo aplica una operación matemática (convolución) en cada coordenada (i, j) de la imagen original y obtene un único valor de salida. El valor de salida se almacena en la imagen de salida en las mismas coordenadas (i, j) que el centro del núcleo. Esta descripción se puede observar en la siguiente Figura 2.10.

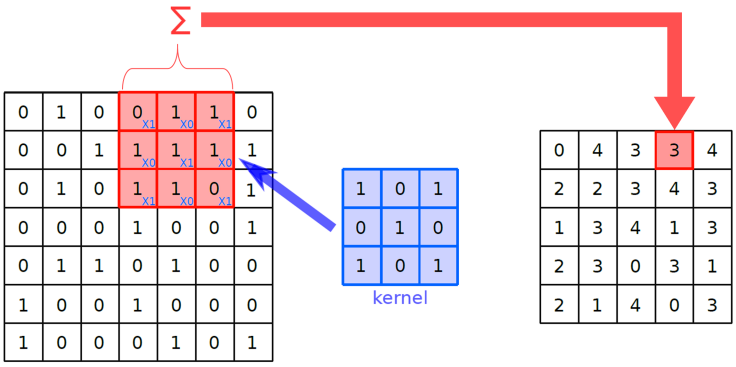


Figure 2.10: Ejemplo convolución con kernel 3 x 3

La convolución de un kernel (K) con tamaño (m x n) colocado en el píxel (i, j) sobre una imagen I se describe en la siguiente ecuación [31].

$$(I * K)(i,j) = \sum_m \sum_n (i+m, j+n) K(m,n) \tag{2.1}$$

Las CNNs están compuestas por diferentes capas (ver Figura 2.9), donde las primeras capas se ocupan de extraer ciertas características como los esquinas o bordes de una imagen. Una vez detectados, son utilizados para localizar formas en las capas posteriores. Cuando se localizan dichas formas, entonces se continua con la detección de las características de alto nivel como para el caso expuesto en la Figura 2.9 (un coche como imagen de entrada) puede ser una rueda. Las capas convolucionales sirven como extractores de las principales características [37], aprendiendo así, las representaciones de singularidades de las imágenes de entrada. Por su parte, las últimas capas están totalmente conectadas y son las encargadas de otorgar una predicción gracias a estas últimas características extraídas.

Normalmente, a la salida de una capa convolucional y con la intención de introducir la no linealidad en un sistema que tiene operaciones lineales durante la capa convolucional, se aplica la unidad lineal rectificada (ReLU), cuya función es sustituir los valores negativos (que se hayan podido dar en la convolución) en cero [37], consiguiendo así un resultado no lineal a partir de

una operación lineal como es la convolución. Las unidades lineales rectificadas se presentaron como una función de activación alternativa no saturante a las funciones tangente hiperbólica y sigmoidea.

En las redes neuronales convolucionales es habitual el uso de una capa con función de agrupación o pooling. La función principial de esta capa es la reducción de forma progresiva de las dimensiones de la salida de la capa convolucional, reduciendo la resolución espacial de los mapas de características y logrando así la invariancia espacial para las distorsiones de entrada [5]. También, menoraría la cantidad de parámetros y la carga computacional, además de intentar controlar el sobreaprendizaje.

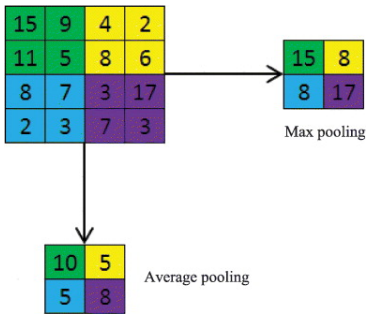


Figure 2.11: Max pooling y average pooling con filtro 2 x 2 y paso de 2 unidades de píxel [5]

Inicialmente, era una práctica común usar capas de agregación promedio para propagar el promedio de todos los valores de entrada, de una pequeña vecindad de una imagen a la siguiente capa. Sin embargo, en modelos más recientes, las capas de agregación de agrupación máxima propagan el valor máximo dentro de un campo receptivo a la siguiente capa. En ambas, se define un tamaño de filtro o ventana y se asigna el valor promedio o máximo de los que se encuentren en su interior (ver Figura 2.11). Esta operación se realiza de manera independiente para cada mapa de características, por lo que para una imagen en color que genere tres capas, obtendremos tres salidas.

Finalmente, se agregan unas capas completamente conectadas para completar la arquitectura CNN (visto como fully connected en la Figura 2.9). Esta es la misma arquitectura ANN totalmente conectada de la que se habló anteriormente. Pero, previamentes es necesario un aplanamiento ya que la salida de las capas de convolución y agrupación son volúmenes en tres dimensiones, mientras que por su parte una capa completamente conectada espera un vector de una dimensión (de números). Por lo tanto, se aplana la salida de la capa de agrupación final a un vector y eso se convierte en la entrada a la capa completamente conectada. Las capas completamente conectadas que siguen a estas capas interpretan las representaciones de características que se han obtenido y realizan la función de razonamiento de alto nivel.
Dentro del marco de las CNN, su principal función es la de realizar la clasificación en la última parte del modelo, ya que con lo contado en este capítulo se podrá obtener una decisión basada en la imagen completa de entrada. En estas últimas capas (las capas densas, menos en la capa de salida) es habitual aplicar la técnica de *dropout* que desactiva un número de neuronas para evitar overfitting [37]. El número de neuronas en la última capa ha de ser igual al número de clases para los cuales ha sido entrenada la red, activándose una única neurona cuando el fin de la CNN sea clasificar.

2.4 Desarrollos de redes en tareas de visión artificial

A partir de las CNNs han ido surgiendo diferentes arquitecturas que han sido relevantes en el contexto de la clasificación de imágenes y detección de objetos, y se enumeran a continuación:

2.4.1 Redes de clasificación

- **LeNet** [16]. Es una de las primeras redes exitosas y la cual posee una de las arquitecturas más simples. Concretamente 5 capas (2 capas convolucionales y 3 completamente conectadas). Esta arquitectura poseía alrededor de 60,000 parámetros, y se utilizó para la lectura de los dígitos de los códigos postales.

- **AlexNet** [38]. Con 60 millones de parámetros, AlexNet incluye 5 capas convolucionales, max-pooling, dropout y 3 capas totalmente conectadas. La diferencias principales con LeNet es que en este caso se presenta una red más profunda. Además, se usó ReLU para las funciones no lineales, siendo los primeros en implementarlo como funciones de activación. Esta red se diseñó para clasificar las 1000 posibles categorías de ImageNet. AlexNet es considerada como una de las arquitecturas más importantes e influyentes ya que popularizó el uso de las CNNs en la comunidad de la visión artificial.

- **GoogLeNet** [39]. Su principal contribución fue conseguir reducir de forma considerable el número de parámetros en comparación con AlexNet, pasando de 60 millones de parámetros a los 4 millones de parámetros. Uno de los aspectos que caracteriza a esta red es la creación de *Inception*, módulo que basa su funcionamiento en poseer distintos tipos y tamaños de convoluciones para una misma entrada, guardando sus diferentes valores de salida como parte del entrenamiento. También, destacar las versiones posteriores evolucionadas a partir de esta red, como Inception-v3 e Inception-v4.

- **VGGNet** [17]. Esta arquitectura posee diferentes versiones, siendo VGG-16 y VGG-19 los modelos más conocidos en el entorno del reconocimiento de imágenes. El desarrollo de la red, aportó la idea de que la profundidad de la arquitectura es un componente culminante a la hora de obtener buenos resultados. Y, es que la red posee alrededor de 140 millones de parámetros, convirtiéndolo en un modelo pesado, difícil de manejar y costoso de entrenar

- **ResNet** [2]. Con el aumento de la profundidad de la red, la precisión se satura y luego se degrada rápidamente. *Microsoft Research* abordó el problema con ResNet, ganando el desafío del *ILSVRC* 2015 y fijando nuevos récords en clasificación, detección y localización con su arquitectura. ResNet implementa conexiones de omisión (también conocidas como conexiones de acceso directo entre capas no contiguas, residuales) y la normalización por lotes, consiguiendo prevenir el sobreaprendizaje y acelerando la evaluación del modelo.

2.4.2 Redes para detección de objetos

Con el avance de las redes en el campo de la visión artificial, nacen sistemas capaces no solo de clasificar o reconocer objetos en una imagen, sino localizar cada uno de ellos a través de un cuadro trazado alrededor del objeto a detectar. Esto convierte a la detección de objetos en una tarea significativamente más compleja en comparación con la clasificación de imágenes. Todos estos modelos comparten el mismo núcleo ya que están basados en CNNs y en las extensiones de los modelos de clasificación de imágenes.

Esta sección presentará las principales arquitecturas que explotan el aprendizaje profundo para la detección de objetos, realizando una división de las redes para detección de objetos. Por una lado, estarán las redes de dos fases y por otro las redes de una fase. Siendo la principal diferencia que las de una fase (SSD, y YOLO la cual no se utilizará en este proyecto) son más rápidas en comparación a las de dos fases y dependiendo del caso, no se pierde demasiada eficiencia de detección.

2.4.2.1 Redes de dos fases

Nos encontraremos las redes de dos etapas, las cuales utilizan el método externo de propuesta de regiones y una red de clasificación.

- **R-CNN** [6]. El propósito es la localización de los objetos, en donde, este proceso se puede dividir en:
  - Extraer las regiones donde los objetos podrían potencialmente estar. Esta fase, llamada fase de propuesta de regiones (o ROI por sus siglas en inglés), se calcula a través de un algoritmo llamado búsqueda selectiva [40]. Este algoritmo considera la imagen a través de ventanas de diferentes tamaños, y para cada dimensión intenta agrupar píxeles adyacentes por textura, color o intensidad.
  - Posteriormente, para extraer caracteristicas de la region se usa una SVM (Support Vector Machine) para clasificar la región.

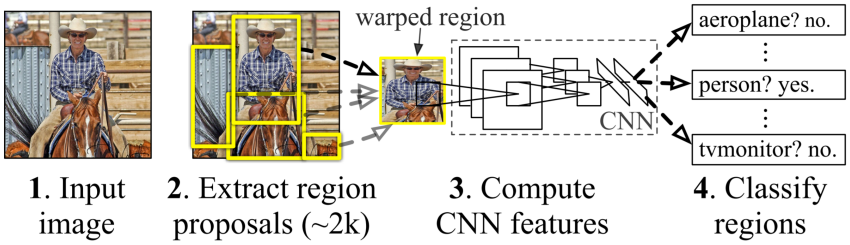


Figure 2.12: Resumen del sistema de trabajo de una R-CNN [6]

De forma general, los métodos de cálculo para hallar las propuestas de regiones tienen una alta complejidad computacional. Cada región de interés de esta arquitectura se convierte en la entrada de una CNN, por lo que, el numero veces que hay que aplicarla es igual a la cantidad de regiones de interés (generalmente unos pocos miles por imagen). Esto hace que el problema no sea adecuado para aplicaciones en tiempo real, donde el tiempo de detección debe ser casi instantáneo.

- **Fast R-CNN** [7]. Esta versión supone una mejora de la R-CNN. La diferencia sustancial consiste en colocar una capa de agrupación ROI entre las CNN y el primer nivel completamente conectado, haciendo que solo haya que aplicar las capas convolucionales una vez. Por otro lado, se cambia SVM por clasificador softmax y regresor del cuadro delimitador (*bbox regressor* en la Figura 2.13), lo cual, mejora en que se pueda entrenar la red de principio a fin, además de la precisión de localización

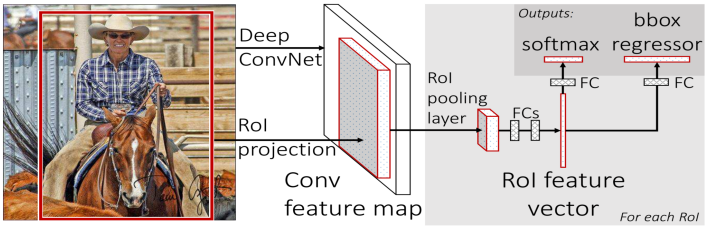


Figure 2.13: Arquitectura Fast R-CNN [7]

- **Faster R-CNN** [8]. Faster R-CNN añadió una Red de Propuesta de Región (RPN) después de la última capa convolucional, sustituyendo de esta manera el método de propuestas de regiones por una red profunda. La RPN (ver Figura 2.14) reduce la complejidad y mejora la precisión. Específicamente, el RPN usa una ventana deslizante que se desplaza en el mapa de características y clasifica lo que está debajo de la ventana deslizante como un objeto/no objeto proponiendo un cuadro delimitador. Faster R-CNN logra una mejora en velocidad y precisión.

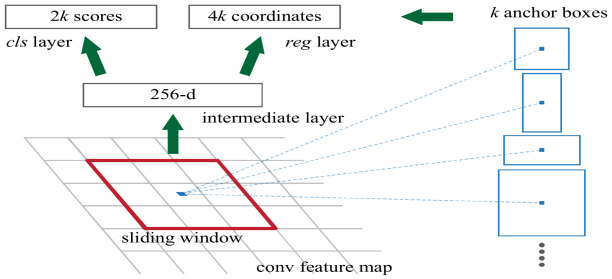


Figure 2.14: Region Proposal Network [8]

- **R-FCN** [9]. El mecanismo del uso compartido de una sola CNN para todas las regiones es el que usa esta arquitectura y hace que aumente la velocidad maximizando el número de parámetros compartidos. Todas las capas de extracción de características se mueven a la parte que aplica a la imagen completa, reduciendo el procesamiento. Además, las capas fully connected se conviertene en capas convolucionales. La arquitectura se vería de la siguiente forma:

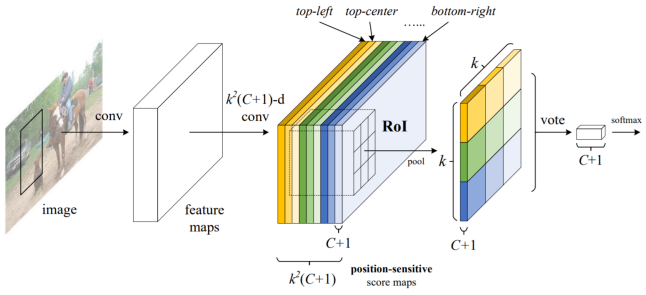


Figure 2.15: Estructura R-FCN [9]

- **Mask R-CNN** [41]. Esta red, se presenta como una evolución de Faster R-CNN y es definida generalmente para segmentación de imágenes. Por su parte, añade un incremento en el cálculo, al aumentar una salida (máscara del objeto) con respecto a Faster R-CNN, la cual posee dos salidas para cada propuesta en detección de objeto (etiqueta de clase y caja envolvente). Pese a ser una red de segmentación, también se puede utilizar para detección y será analizada en este proyecto.

2.4.2.2 Redes de una fase

- **SSD** [10]. El último modelo a analizar es el SSD o Single-Shot Detector. Esta arquitectura acelera la fase de procesamiento al eliminar el RPN. Lo nuevo es el mapa de características de múltiples escalas, en donde las múltiples escalas se utilizan para hacer la detección de objetos a dierentes escalas. A partir de los mapas de características a diferentes escalas, se hace predicción de localización y clase, utilizando bboxes (cuadros delimitadores) con relacion de aspecto predefinida.

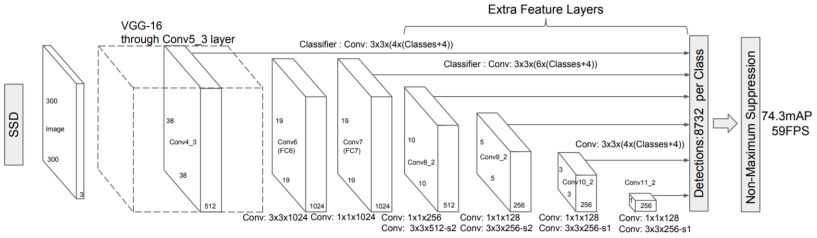


Figure 2.16: Arquitectura SSD [10]

Un conjunto de capas de convolución permite extracciones de entidades de escala múltiple. Además, las capas convolucionales adicionales ayudan a manejar objetos más grandes, mientras que el algoritmo de supresión no máxima se usa para filtrar múltiples cuadros que pueden aparecer.
Los resultados experimentales en los conjuntos de datos PASCAL VOC, COCO e ILSVRC confirman que SSD es el más rápido de los modelos anteriormente mencionados, con una precisión competitiva para la reducción de tamaño que se produce [10].

2.5 Visión general de las arquitecturas ligeras

Como se ha comentado, las redes neuronales convolucionales profundas han logrado un gran éxito en muchas tareas de reconocimiento visual, logrando rendimientos notables a un costo de capas más profundas y millones de parámetros, así como grandes requisitos de memoria. Una de las razones de esto, especialmente en los primeros años de aprendizaje profundo, puede haber sido dada por el hecho de que el progreso fue impulsado principalmente por lograr los mejores resultados.

En un entorno basado en la nube con abundantes capacidades computacionales, habilitado por múltiples unidades de procesamiento gráfico, tales requisitos de memoria masiva pueden no considerarse una restricción. Sin embargo, estos modelos de redes neuronales profundas son computacionalmente costosos e intensivos en memoria, lo que dificulta su implementación en

aplicaciones con estrictos requisitos de latencia o en dispositivos con bajos recursos de memoria a pesar del rápido desarrollo del hardware y sistemas integrados móvil [42]. En esta sección, se hablará de la importancia de este tema otorgando una visión general del mismo.

2.5.1 Estructura de los modelos

Conociendo que el tamaño del modelo es un factor clave a la hora de obtener una arquitectura ligera, la primera idea sería la compresión de redes neuronales modificando y reduciendo la precisión de los parámetros, intentando mantener la mayor parte de su rendimiento, mientras los requisitos de memoria se reducen.
Así mismo, en los últimos tiempos se han empezado a estudiar los factores individuales que contribuyen a la complejidad de las capas convolucionales, repensando en el diseño de la arquitectura CNN para llegar a modelos más afables en cuanto a memoria y logrando diseños de nuevas arquitecturas de red más eficientes en memoria. En esta sección se conocerán las nuevas arquitecturas emergentes actualmente.

2.5.2 SqueezeNet

SqueezeNet logra la misma precisión que AlexNet, pero con 50 veces menos parámetros. Además, con las técnicas de compresión que se aplican, SqueezeNet se comprime a menos de $0.5MB$ alrededor de 500 veces más pequeño en cuanto a tamaño que AlexNet [11]. Para lograr eso, SqueezeNet tiene las siguientes ideas clave:

- La mayoría de los filtros 3x3 se reemplazan con filtros 1×1: Ya que en comparación se tendrá 9 veces menos parámetros.
- Disminuir la cantidad de canales de entrada a filtros 3×3: la cantidad de parámetros de una capa convolucional depende del tamaño del filtro, la cantidad de canales de entrada y la cantidad de filtros. Por lo tanto, para disminuir el número de canales de entrada a los filtros 3x3, se hará usando capas de compresión.
- Reducir la muestra al final de la red para que las capas de convolución tengan mapas de activación grandes. El modelo debe ser pequeño, pero debemos asegurarnos de obtener la mejor precisión posible, así que cuanto más tarde se reduzcan los datos, entonces más información se podrá retener para las capas intermedias, lo que aumenta la precisión.

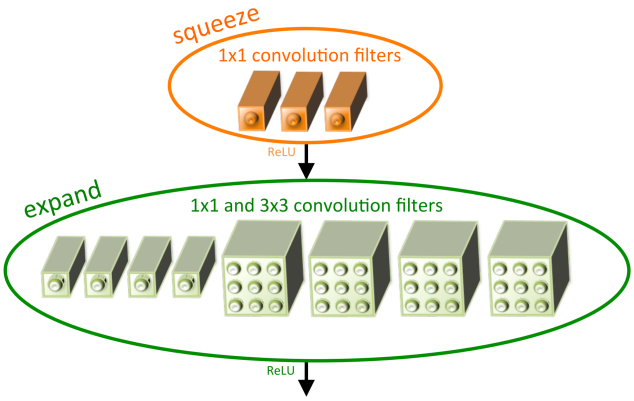


Figure 2.17: Modulo de disparo [11]

El llamado módulo de disparo, se divide en una capa de compresión y una capa de expansión (Figura 2.17). La capa de compresión consta de convoluciones 1×1, con el fin de combinar todos los canales de los datos de entrada en uno y, por lo tanto, reducir el número de canales de entrada para la siguiente capa. El siguiente paso es la capa de expansión. Aquí las convoluciones 1×1 se mezclan con las convoluciones 3×3, ya que las coonvoluciones 1×1 por sí solas no pueden detectar estructuras en la imagen. Es una forma de intentar mantener la capacidad de extracción de características de los filtors 3x3 pero reduciendo el numero de parámetros.

SqueezeNet utiliza ocho de estos módulos de disparo y una sola capa convolucional como capa de entrada y salida El relleno correcto asegura que la salida de las convoluciones tengan el mismo tamaño. Además, se usa la agrupación promedio global en lugar de capas completamente conectadas.

2.5.3 MobileNet

La red MobileNet de *Google*, está diseñada específicamente para sistemas integrados y va un paso más allá al modificar la operación convolucional como tal. De este modo, MobileNet sigue un enfoque un poco diferente, utilizando convoluciones separables en profundidad (característica clave de esta arquitectura). La red, además, posee dos hiperparámetros globales simples, capaces de intercambiar entre eficiencia y precisión, lo que permite a los desarrolladores elegir el modelo correcto en función de las limitaciones de su sistema [21].

Una convolución estándar filtra y combina las entradas en un nuevo conjunto de salidas en un solo paso (Figura 2.18 *apartado a*). Usando la convolución separable en profundidad, estas operaciones se dividen. En primer lugar, se aplica un filtro único a cada canal de entrada (la Figura 2.18 *apartado b* muestra esta técnica). Más tarde, la convolución puntual aplica una convolución 1x1 para combinar las salidas de la convolución en profundidad (se describe en la Figura 2.18 *apartado c*). Así, se tiene una capa separada para filtrar y otra capa para combinar. Este mecanismo reduce de forma considerable tanto el cálculo como el tamaño del modelo.

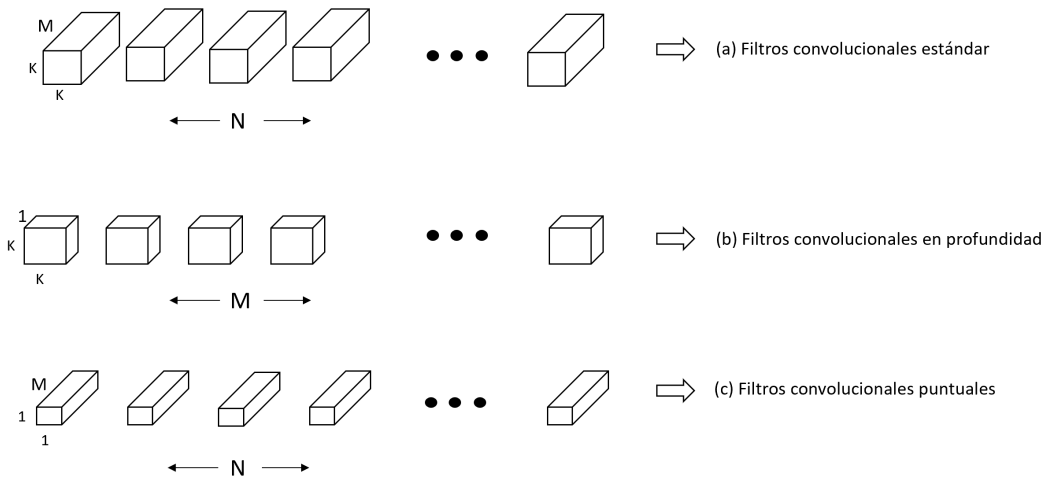


Figure 2.18: Filtros convolucionales estándar (a) se reemplazan por dos capas: convolución en profundidad (b) y convolución puntual (c) para construir un filtro separable en profundidad

En resumen, el concepto es que se tiene una convolución que se hace en el espacio igual en todas las capas, y para diferenciar que cada capa tenga una respuesta distinta, se tiene un filtro que solo coge un píxel en el dominio espacial y filtra en profundidad. Pongamos como ejemplo que se usa cinco filtros 3×3 y se tiene diez canales de entrada, lo cual significa 450 parámetros (5x3x3x10), pero si se usa la convolución separables en profundidad, los calculos se harían de la siguiente forma: 5x3x3 + 10x1x1x5 = 95 parámetros.

Como ya se explicó, los diseñadores pueden cambiar la precisión y la eficiencia gracias a dos hiperparámetros: α y ρ.

- α se llama al multiplicador de ancho y su función es la de adelgazar una red de manera uniforme en cada capa ya que define el número de canales de entrada y salida. α tiene un rango entre 0 y 1. MobileNet, en su forma básica, tiene $\alpha = 1$, que corresponde al número predeterminado de canales en las convoluciones, mientras que los modelos reducidos tienen $\alpha < 1$, en donde a parte de la reducción de la cantidad de canales, también a su vez habrá una disminución en la cantidad de pesos y los costos computacionales.
- Por su parte, ρ es el multiplicador de resolución: al aplicar ρ se establece implícitamente el cambio de tamaño de las imágenes de entrada, teniendo así influencia en el número de cálculos.

2.5.4 MNasNet

El equipo de investigación de *Google Brain* reformuló el problema del diseño de redes neuronales convolucionales en móviles como un problema de aprendizaje de refuerzo. Conceptualmente, MNasNet utiliza una búsqueda automatizada de arquitectura neuronal basada en el aprendizaje de refuerzo para diseñar CNN móviles, basándose en dos ideas fundamentales [12]. En primer lugar, la tarea de diseñar una CNN móvil se formula como un problema de optimización de objetivos múltiples que considera tanto la precisión como la latencia de inferencia del modelo. Y en segundo lugar, MNasNet utiliza la búsqueda de arquitectura con aprendizaje de refuerzo para encontrar el modelo que logre el mejor equilibrio entre precisión y latencia.

La arquitectura básica de MNasNet consta principalmente de tres componentes:

- Un controlador basado en una red neuronal recurrente (RNN) con el fin de aprender y muestrear arquitecturas de modelos.
- Un entrenador que construye y entrena modelos.
- Un motor de inferencia para medir la velocidad del modelo en teléfonos móviles reales, por ejemplo con el uso de *TensorFlow Lite*.

Como se mencionó de forma previa, MNasNet manifiesta un problema de optimización multi-objetivo que apunta a lograr alta precisión y velocidad, usando un algoritmo de aprendizaje de refuerzo con una función de recompensa personalizada para encontrar soluciones *óptimas de Pareto* (tales como, modelos que tienen la mayor precisión sin deteriorar la velocidad).

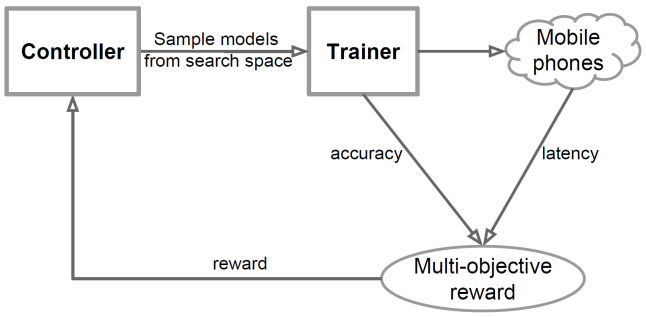


Figure 2.19: Descripción de la búsqueda de arquitectura neuronal compatible con plataformas para dispositivos móviles [12]

El diagrama de bloques del modelo MnasNet se muestra en la Figura 2.19, donde se observa que MnasNet es un enfoque inspirado en AutoML (aprendizaje automático) en el diseño de modelos móviles mediante el aprendizaje de refuerzo.

Para lograr el equilibrio correcto entre la flexibilidad de búsqueda y el tamaño del espacio de búsqueda, la arquitectura propone un nuevo espacio de búsqueda jerárquica factorizada, que factoriza una red neuronal convolucional en una secuencia de bloques, y luego utiliza un espacio de búsqueda jerárquica para determinar la arquitectura de capa para cada bloque. Así, se permite que diversas capas usen diferentes operaciones y conexiones. Mientras tanto, se fuerza que todas las capas en cada bloque, compartan la misma estructura, reduciendo así de forma significativa el tamaño del espacio.

Capítulo 3

Marcos de implementación

Hoy en día, existe una variedad de entornos para desarrolladores relacionados con trabajos exclusivos en el ámbito del aprendizaje profundo. Dentro de los entornos de deep learning se emplea el lenguaje de programación *Python* interpretado como la interfaz de programación, convirtiéndolo en el escenario más desplegado para desarrollar aplicaciones de machine learning. Las librerías de aprendizaje profundo más conocidas y populares son *TensorFlow* [43], *Keras* [44], *Pytorch* [45] ó *Caffe* [46]. En el caso específico de este trabajo y tal como se enunciará a lo largo de este capítulo se usan fundamentalmente dos: TensorFlow y Keras, debido a que son las librerías con mayor versatilidad y ambos son capaces de lograr sinergias con *TensorFlow Lite* para la incorporación de los modelos en el dispositivo móvil. Además, esto puede permitir también integraciones más directas con respecto a otras plataformas como *Bazel* o *Android Studio*.
Además, se introducirán las bases de datos que se usarán en el trabajo para tener una idea de la importancia que tienen en el ámbito de la visión artificial y del conjunto de imágenes que llegan a contener.

3.1 TensorFlow

TensorFlow es una de las biblioteca más popular para Machine Learning (ML) con un software de uso común para estas tareas, proporcionando una interfaz para expresar algoritmos ML comunes y código ejecutable de los modelos. Está desarrollada para la computación numérica de alto rendimiento con la cabida de ejecutarse en múltiples CPUs, GPUs y unidades de procesamiento tensoriales (TPUs). En TensorFlow, los modelos se representan como un gráfico de flujo de datos que contiene un conjunto de nodos descritos como operaciones, donde cada operación toma como entrada un tensor y proporciona un nuevo tensor como salida. Los tensores son matrices multidimensionales de números que fluyen entre operaciones y son la forma en que los datos se representan en TensorFlow [43].

Así, se considera que TensorFlow es un sistema de aprendizaje automático que funciona a gran escala y en entornos heterogéneos. La idea que subyace de este proyecto de código abierto es el desarrollado del sistema TensorFlow para experimentar con nuevos modelos, entrenarse en grandes conjuntos de datos y poder llevarse a producción. Por lo tanto y gracias en gran medida a la gran comunidad de usuarios se ha adquirido experiencia en muchas aplicaciones diferentes de aprendizaje automático. Esta experiencia se ha querido trasladar y utilizar en el proyecto, más específicamente el de los modelos de detección preentrenados ya que es la plataforma con mayor accesibilidad que se encontró para esta tarea. Además de encontrar una conversión directa con

su plataforma más ligera, TensorFlow Lite, lo cual se considera importante para no tener que añadir mayor número de pasos durante el desarrollo.

3.2 Keras

Se trata de una Interfaz de Programación de Aplicaciones (API) de alto nivel en redes neuronales, escrita en *python* y que corre sobre diferentes motores de deep learning (TensorFlow entre ellos). Por lo tanto Keras es capaz de ejecutarse sobre TensorFlow, lo cual permite cierta extensibilidad. En este sentido, Keras proporciona múltiples herramientas para desarrollar y entrenar modelos de aprendizaje profundo [27].

Keras, se caracteriza por ofrecer una API consistente y simple que ayuda en la minimización del número de acciones del usuario, mientras que los modelos Keras se fabrican conectando bloques de construcción configurables, con pocas restricciones. Más aún, en lo que concierne al reconocimiento visual, facilita la disposición de un conjunto de modelos preentrenados en bases de datos importantes y características de este campo. En el marco del trabajo, se usarán los modelos de clasificación preentrenados al ser la plataforma que se encontró una mayor comodidad a la hora de tratar estos modelos en dicha tarea. Además como se comentró previamente se puede utilizar sobre TensorFlow y por lo tanto la conversión a TensorFlow Lite será similar al del propio TensorFlow.

3.3 TensorFlow Lite

A medida que el Machine Learning ha crecido en popularidad, se ha convertido necesario el despliegue de éste en dispositivos móviles e integrados. Por lo que una plataforma nueva se hace necesaria, la cual, esté diseñada de forma específica para ser liviana, rápida, y adecuada para agregrar ML en dispositivos. Así, TensorFlow Lite [26] se presenta como la versión más ligera de TensorFlow (su paquete IA), diseñada especialmente, para facilitar a los desarrolladores la implementación de modelos de aprendizaje automático en dispositivos compactos y móviles, siendo compatible con una variedad de plataformas, incluyendo Android e iOS.

Algunas de las optimizaciones incluidas en TensorFlow Lite son la aceleración de hardware, marcos como la API de red neuronal de Android y redes optimizadas para dispositivos móviles. Esta plataforma en el proyecto servirá de núcleo como el formato de los modelos para optimizarlos e incorporarlos para un uso en la plataforma móvil.

3.4 Android Studio

Android Studio es el entorno oficial de desarrollo integrado (IDE) para el desarrollo de aplicaciones de Android patentado por Google y basado en IntelliJ IDEA, un entorno de desarrollo integrado de Java para software. Con el objetivo de admitir el desarrollo de aplicaciones dentro del sistema operativo, Android Studio utiliza un sistema de compilación basado en Gradle, emulador. Cuenta además con un diseño en el que incorpora, herramientas de edición, desarrollo y la posibilidad de integración de diferentes kits de desarrollo de software (SDKs). Las aplicaciones creadas en Android Studio se compilan en el formato APK (paquetes de aplicaciones de Android) para su posterior ejecución en dispositivos móviles [23].

Por lo tanto, se considera la herramienta original para desarrollar aplicaciones en plataformas Android. En el proyecto se utilizará como herramienta para integrar las aplicaciones de clasificación y detección de objetos en el móvil.

3.5 Bazel

Bazel es una herramienta creada por *Google* para la construcción de software de prueba con una variedad de plataformas, que permite rapidez y escalabilidad. Más aún, está concebido con la idea de la creación de monorepos (que el código se encuentra en un único repositorio), diversidad de lenguajes, extensos procesos de pruebas y herramientas para la unificación del código con la posibilidad de simplificar multiples tareas en el desarrollo. Bazel está pensado para realizar tareas de forma paralela, con automatización y minimizando los trabajos. Una prueba de ello es que se puede utilizar para generar una salida a partir de un código fuente con la construcción de proyectos en C ++, Java, Python y más plataformas [24]. Se usará esta herramienta ya que se encuentrá relacionado con TensorFlow, específicamente con la compilación y conversión de los modelos.

3.6 Bases de Datos

Con el auge del aprendizaje profundo de la visión artificial en los últimos años, los conjuntos de datos de imágenes se han vuelto transcendentes ya que proporcionan información ayudando a los modelos a aprender en un tema específico. En esta sección se expondrá las bases de datos que se han usado y que han sido necesarias en la realización del proyecto, su importancia en las diferentes tareas de la visión artificial haciendo énfasis en sus ventajas competitivas, a la par que se da una idea de las principlaes diferencias entre ellas.

3.6.1 ImageNet y COCO

ImageNet y COCO son unas bases de datos visuales enfocadas en la investigación del reconocimiento y detección de imágenes u objetos visuales. Así que, proporcionan un recurso para promover el desarrollo de métodos mejorados para la visión artificial. Se componen de varios conjuntos de datos con diferentes número de imágenes para cada uno de ellos.

Bases de datos			
Nombre	Imágenes	Nº de Clases	Uso en el proyecto
ImageNet	>14M	1000	Clasificación
COCO	>200K	80	Detección

Table 3.1: Bases de datos

ImageNet es el conjunto de datos de imágenes más famoso. Este conjunto de datos meticulosamente etiquetado a mano tiene 1.000 categorías de objetos de ámbitos muy diversos [47] que están distribuidas de forma general en alrededor de 1.2 millones de imágenes de entrenamiento, 100.000 para evaluación y 50.000 imágenes en el caso de validación. Esta alta escala de los datos hizo a ImageNet realmente desafiante [48]. Además, parte de la popularidad se debe a su desafío y competencia anual de reconocimiento visual (ILSVRC) a gran escala en el que las

tareas utilizan subconjuntos de datos de ImageNet.

Todo ello, ha convertido a ImageNet en un conjuto de bases de datos de referencia para los algoritmos de clasificación de imágenes desde 2012 y más importante, entrenados sobre esa base de datos. Por lo tanto, muchos de los modelos que mejor rendimiento han obtenido con este conjunto de datos han sido incluidos en la biblioteca de Keras.

Por su parte, la base de datos COCO es un conjunto que contiene más de 200.000 imágenes, las cuales se encuentran distribuidas en 80 clases de objetos con características de segmentación, y representan escenas cotidianas del mundo real [49]. Además, las etiquetas de COCO también incluyen puntos clave (*keypoints*), en donde muchos algoritmos de detección de objetos se pueden beneficiar de estas anotaciones adicionales.

A diferencia del popular conjunto de datos de ImageNet, COCO tiene menos categorías pero más instancias por categoría, lo que puede ayudar a aprender modelos de objetos detallados capaces de una mejora en la precisión de la localización [50]. Este conjunto de datos es lo suficientemente grande para que un modelo entrenado a partir de esta base de datos sea capaz de aprender características visuales de calidad. Se utilizan para las principales tareas de clasificación, detección y segmentación.

Capítulo 4

Desarrollo

En este capítulo se empezará comentando la estrategia con los pasos que se tendrán que realizar para implementar las aplicaciones. Luego se describen las aplicaciones y su funcionamiento tanto en clasificación como en detección, junto con la arquitectura de las mismas, las cuales tienen como punto de referencia diferentes repositorios para cada tarea, pero se explicarán las modificaciones e integraciones de los modelos. Se continuará con la descripción de cómo hacer que los diferentes modelos funcionen con Android ya que tiene que haber una conversión al formato *tflite*, es decir, los modelos tienen que tener un procesado y será diferente para cada tarea.

4.1 Descripción de la implementación práctica en móviles

La idea general es el desarrollo de aplicaciones móviles Android con el uso de la herramienta base, *Tensorflow Lite*, ya que está diseñada para integrar modelos de aprendizaje automático en plataformas móviles.

En este proyecto, usaremos distintos modelos preentrenados, los cuales mediante el uso del convertidor de TensorFlow Lite, serán transformados tanto de Keras (para clasificación) como de TensorFlow (para detección) a formato *.tflite*. Este formato del modelo permite tener una configuración apta para el uso del intérprete de TensorFlow Lite, el cual, ejecuta modelos especialmente optimizados en muchos tipos de hardware diferentes, incluidos los teléfonos móviles Android que es el caso que nos ocupa. La optimización será un paso extra que se añadirá con el fin de mejorar el rendimiento de las diferentes redes. Finalmente estos modelos se cargarán a partir de las aplicaciones en el dispositivo móvil.

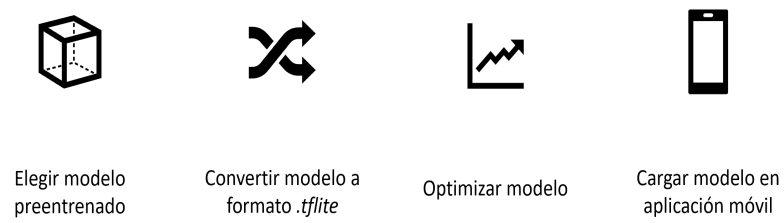


Figure 4.1: Estrategia

4.2 Funcionamiento de las aplicaciones

Una vez se construye y se ejecuta, se crea un apk de cada aplicación que se instalará como app en el móvil, teniendo así, por un lado un clasificador y por otro un detector, ambos en tiempo real a partir de redes convolucionales ligeras.

4.2.1 Aplicación de clasificación

El funcionamiento de la aplicación de clasificación será el siguiente:

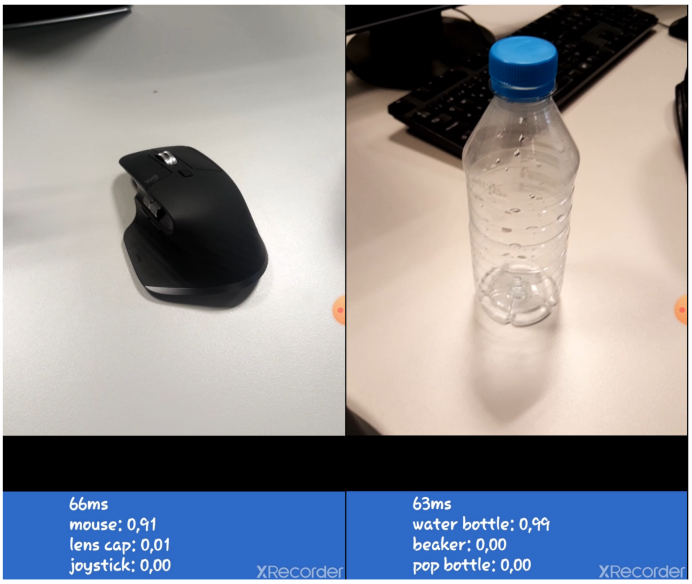


Figure 4.2: APP Clasificación

Lo que está sucediendo con el funcionamiento de la app (Figura 4.2) es que se está leyendo frames de la cámara y convirtiéndolos en imágenes, para luego usar esas imágenes como entradas del modelo que a su vez y teniendo un modelo que clasifique, generarán valores de salida. Estas salidas son indexadas a la etiqueta apropiada junto con el valor de esa etiqueta o lo que es lo mismo: la probabilidad que la imagen coincida con la etiqueta. Se cogerán las tres mejores probabilidades y serán mostradas en la interfaz, añadiendo el tiempo que tarda de inferencia. Esto, se muestra en la parte inferior de la imagen (Figura 4.2) en donde por orden se empieza enseñando el tiempo de inferencia y a continuación las tres clases más probables del objeto que la cámara del móvil esté enfocando.

4.2.2 Aplicación de detección

El funcionamiento es similar al comentado anteriormente en la Sección 4.2.1, sólo que en este caso en la interfaz aparte de añadir la etiqueta, también se están añadiendo los cuadros delimitadores para poder localizar la posición de cada objeto (Figura 4.3). Junto a estos cuadros delimitadores que detectan los objetos se muestra la clase más probable de cada detección añadiendo la probabilidad de acierto.
En la paste inferior de la Figura 4.3 nos encontramos con el tamaño de frame que tiene las imágenes que la cámara está capturando, el tamaño de recorte y dimensionado de las imágenes

que se necesitan para el procesado dependiendo de las especificaciones de cada modelo (para el caso del ejemplo de funcionamiento es 300x300). También se muestra el tiempo de inferencia y el número de hilos que se deseen utilizar. Aunque esto veamos que se puede modificar, dependerá de las características de cada móvil si se permite este uso y aumento del número de hilos. En nuestro caso en todo momento mantendremos el número de hilos igual a 1.

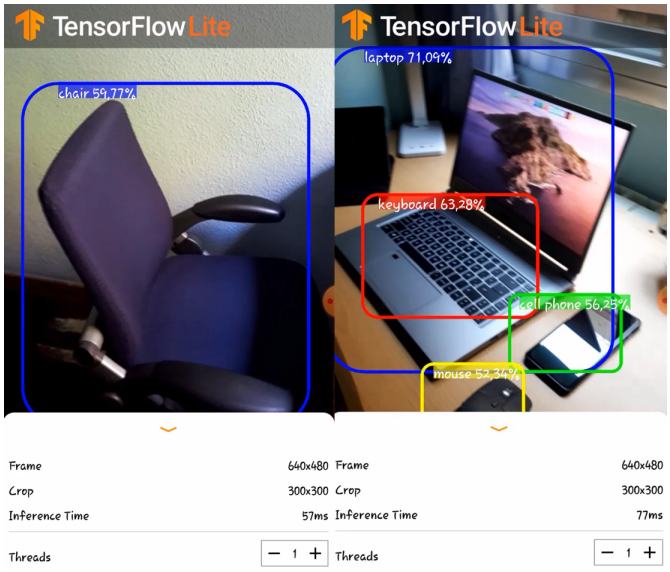


Figure 4.3: APP Detección

Esta aplicación cuenta con la limitación de que sólo se pueden ejecutar los modelos con arquitectura SSD [51]. Al momento de realizar el desarrollo, y uniendo esta limitación, se puede decir que la integración que se ha hecho es la máxima respecto a las posibilidades que nos ofrece la herramienta TensorFlow actualmente. Este factor se comentará en secciones posteriores.

4.3 Arquitectura de las aplicaciones

A continuación, se muestra el diagrama de la arquitectura que tienen las aplicaciones en las tareas de clasificación y detección de objetos.

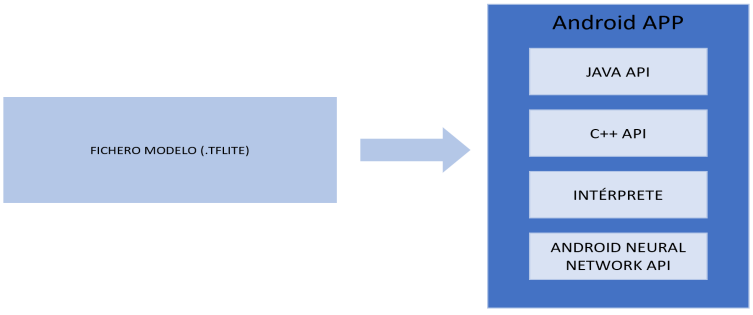


Figure 4.4: Diagrama arquitectura aplicaciones

Entonces, a la vista del diagrama (Figura 4.4), se observa que podemos usar cualquier archivo de fichero de un modelo convertido de la forma oportuna a *.tflite* e integrarlo en la aplicación móvil. Para implementarlo es necesario del uso de diferentes aspectos:

- API de Java: una clase envoltorio alrededor de la API de C ++ en Android.
- API C ++: carga el modelo Lite y llama al intérprete.
- Intérprete: Se encarga de ejecutar el modelo. Utiliza la carga selectiva del núcleo, que es una característica única de Lite en Tensorflow.
- Android Neural Network API: Podrían hacer uso de esta API los dispositivos con aceleradores de hardaware integrados. Sin embargo, en la actualidad la ANN API está disponibles para modelos con sistema operativo Android igual a 8.1 o superiores [52]. Lo cual en nuestro caso no se implementará al utilizar como se verá en el Capítulo 5 versiones anteriores de Android. De esta forma, estará predeterminado a utilizar la CPU del dispositivo

A continuación, se conocerán los repositorios en los que se ha basado cada tarea junto con las diferentes configuraciones necesarias con el fin de agregar nuevas características a las aplicaciones.

4.3.1 Integración aplicación clasificación

La página oficial de *Tensorflow Lite* emplea un ejemplo de aplicación de clasificación de imágenes con el que empezar a iniciarte en el mundo de la visión artificial y el aprendizaje automático, pero para este tarea dicha aplicación no se utilizó. Esto se debe a la falta de coherencia en su uso con los diferentes modelos como por ejemplo y entre otros de la visualización por pantalla de objetos sin clasificar. Este hecho se daba al poseer un conjunto distinto de clases. Por lo tanto y debido a este impedimento, se decidió obtener e investigar otra apliación similar que realice la misma función. Así, en vez de utilizar la aplicación de ejemplo de la página oficial de Tensorflow, se decidió extraer del repositorio de *Google Codelabs* [53] en concreto de *Tensorflow for poets*, la aplicación de clasificación de imágenes, ya que *Google Codelabs* proporciona esta opción de desarrollo necesario en este trabajo.

4.3.2 Integración aplicación detección

Esta vez se integrará la aplicación que se provee en la misma página oficial de Tensorflow (a diferencia de lo ocurrido en la tarea de clasificación). El diagrama de la arquitectura que tendrá la aplicación una vez se pueda usar el modelo será el mismo en ambas tareas (Figura 4.4). Como dato a tener en cuenta, en la actualidad existe una limitación ya que la ejecución en dispositivos móviles con TensorFlow Lite en la tarea de detección de objetos sólo está admitido para modelos SSD [51].

4.3.3 Uso de la API de TensorFlow Lite para Java

Dentro de esta API se encuentran varios ficheros utilizados para las aplicaciones finales. En el siguiente diagrama se recogen las entradas y salidas de la aplicaciones y donde interviene cada clase que posteriormente se describirán.

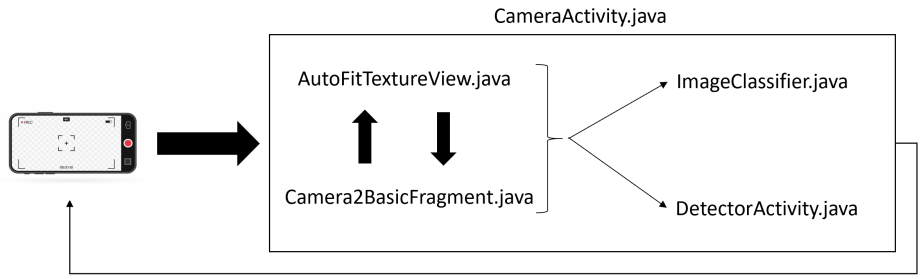


Figure 4.5: Diagrama API de TensorFlow Lite para Java

En conjunto y de forma resumida se coge como entrada los datos de la imagen que desde la cámara se reciben como un mapa de bits, conviertiendolo en un buffer de bytes para que el modelo pueda leerlo. Cuando ya se tiene esto, se cargará el contenido de la imagen en el intérprete de TFLite obtiendo dos casos. Por una lado para **clasificación (ImageClassifier.java)** se obtiene un array de las probabilidades que tendrá cada una de las etiquetas, mostrando como salida en la aplicación las tres clases más probables junto con su tiempo de inferencia. Por otro lado para **detección (DetectorActivity.java)** además se añaden las cajas delimitadoras que incorporan la localización de los objetos.

- **AutoFitTextureView.java**, ejecuta la función de dimensionar de manera automática tras la captura desde la cámara de las imágenes que se van mostrando por pantalla de forma constante durante la ejecución de la aplicación para el ajuste de la proporción en el terminal.

- **ImageClassifier.java**, cuya función es la participación con la plataforma TensorFlow para el uso de los modelos en formato *tflite*. Será necesario importar un intérprete, el cual, cargue el modelo y permita la integración proporcionando un conjunto de entradas. Así, el intéprete TFLite cargará el modelo y lo ejecutará, escribiendo las salidas. Para ello, se inicia una sesión en el entorno de Tensorflow y se ejecuta la inferencia en el modelo. Una vez se procesa el resultado se mostrará en la parte inferior de la pantalla los resultados tanto de las tres mejores clasificaciones como del tiempo de coste que cada modelo tarda en el tratamiento de toda la tarea de clasificación.

  Además, esta clase necesita añadir un paso para tratar el uso del modelo de clasificación convertido a TFLite y para ello se precisa añadir dicho modelo junto con sus respectivas etiquetas (lista de los objetos que el modelo puede identificar) a una carpeta denominada *assets* donde se gestionan los ficheros externos. Además de un cambio en esta clase (ImageClassifier.java) para que se conozcan la ruta del modelo (y sus correspondientes labels) con el que se quiere integrar en la aplicación.

- **DetectorActivity.java**, el cual su función será parecida que para *ImageClassifier.java* importando un interprete, cargar el modelo y permitir la integración, pero para el caso de detección, adicionalmente, los resultados de confianza que se muestran por pantalla serán aquellas probabilidades mayores a un umbral de 0.5, umbral que se piensa que es suficiente para mostrar con una confianza alta sin temor a grandes errores ni a colapsar la pantalla del móvil con varias detecciones de objetos con probabilidades de acierto poco representativas. En conjunto, su funcionamiento será similar al de clasificación pero adaptado a la detección

para mostrar la posición del objeto junto con su etiqueta de las clases de COCO con probabilidades mayores a 0.5, además del tiempo de inferencia.

Si el modelo que se está tratando es float o cuantizado se diferenciará en que el modelo cuantizado actuará dando a cada valor un solo byte que represente un valor entre 0 y 255. Esta diferenciación se tendrá que especificar en esta clase. Además, al igual que en el caso anterior, esta clase necesita añadir un paso para tratar el uso del modelo de detección convertido a TFLite y para ello se precisa añadir dicho modelo junto con sus respectivas etiquetas a la carpeta denominada *assets* donde se gestionan los ficheros externos. Además de un cambio en esta clase (DetectorActivity.java) para que se conozcan la ruta del modelo (y sus correspondientes labels) que se quiere integrar en la aplicación.

- **Camera2BasicFragment.java**, clase que gestiona la cámara del móvil, haciendo uso tanto de la dimensión automática como de presentar los resultados.
- **CameraActivity.java**, la actividad que agrupa las funciones de las tres clases previas (*AutoFitTextureView.java, Camera2BasicFragment.java, ImageClassifier.java* ó en su defecto *DetectorActivity.java*), además de crear la interfaz gráfica añadida en la app.

4.4 Adaptación de modelos

Como se ha comentado (Sección 4.1), la estrategia de implementación parte por la elección de modelos preentrenados para posteriormente trabajar con ellos en una conversión del modelo al formato necesario que en este caso es *.tflite*, además de un tratamiento de dichos modelos a través de una optimización que como se comentará más adelante tendrá que ver con la cuantización. El siguiente diagrama enseña el procesado de los modelos que se tendrá que seguir en TensorFlow Lite para la adaptación de los diferentes modelos con los que se trabajarán para los casos de clasificación y detección. Como nota, la cuantización puede ser preentrenamiento (el entrenamiento del modelo se hace con pesos cuantificados) o posentrenamiento (la que se seguirá en este proyecto con los modelos preentrenados).

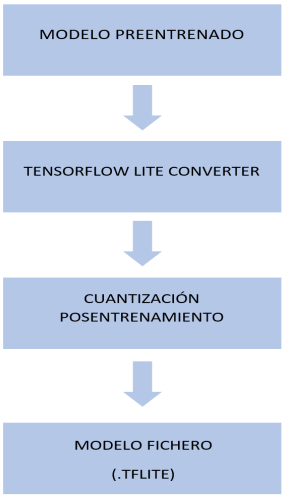


Figure 4.6: Diagrama adaptación de modelos

La cuantización o cuantificación vendrá limitada por la arquitectura de la red, es decir, si existe alguna limitación interna (por ejemplo, cómo han sido definidos los modelos durante el entrenamiento de los mismos) no se podrá realizar algún tipo de conversión. En clasificación no se han encontrado ninguna limitanción, sin embargo sí hubo limitaciones en la tarea detección, las cuales se comentarán más adelante (Sección 4.4.2.2). Además, algunos de de los diferentes hardwares con los que TensorFlow Lite es capaz de interactuar también pueden tener algunas limitaciones, dependiendo de sus características. Para el caso de este proyecto, asumiremos que nuestro harware del teléfono no posee ningún límite.

4.4.1 Clasificador de imágenes

En esta subsección se presenta el desarrollo de la adaptación de los modelos en clasificación con sus respectivos pasos.

4.4.1.1 Elección modelos preentrenados

A partir de la API de alto nivel de Keras (*tensorflow.keras*) se incluirán los modelos preentrenados, ya que permite la descargar de varios modelos de aprendizaje profundo preentrenados usando el módulo de aplicación de keras (*tensorflow.keras.applications*). Así, dentro de este módulo, hay varias clases, cada una responsable de trabajar con un modelo, donde como se mencionó en el Capítulo 3 para la tarea de clasificación se usará la base de datos de Imagenet, por lo que se tendrá que especificar también que los pesos de los modelos que se buscan son los que han sido entrenados en tal base de datos.

En la siguiente Tabla se comprueba los modelos tanto ligeros como estándar elegidos para la tarea. Todos ellos tienen un nivel de cuantificación inicial de punto flotante de precisión de 32 bits.

Modelos disponibles usados			
Modelo	Tipo de Modelo	Parámetros	Profundidad
MobileNetV2	Ligera	3,538,984	88
NASNetMobile	Ligera	5,326,716	-
ResNet50	Común	25,636,712	-
VGG16	Común	138,357,544	23

Table 4.1: Modelos disponibles usados en clasificación

La profundidad se refiere a la profundidad topológica de la red. Esto incluye capas de activación, capas de normalización por lotes, etc [54]. Además, comentar que todos los modelos tendrán una entrada general de *224 x 224* con tres canales RGB (rojo, verde, azul) por píxel.

4.4.1.2 Conversión modelos

Una vez se haya elegido el modelo preentrenado, Keras da la posibilidad de guardar los pesos del modelo en formato HDF5 (extensión *.h5*). Este es un formato de cuadrícula ideal para almacenar matrices multidimensionales, que en este caso se trata de los pesos de las redes neuronales. En el paso siguiente para esta conversión, se usará la API de Python (*tf.lite.TFLiteConverter*) para convertir modelos TensorFlow a TensorFlow Lite.

4.4.1.3 Optimización

Para la optimización de los modelos se tomará la cuantización como función principal con la que mejorar el rendimiento de los mismos. Concretamente, optimización se referire a una cuantificacion de dos elementos: los pesos y las operaciones. Así es como se tendrá que realizar una cuantización posentrenamiento al ser una técnica de conversión que puede reducir el tamaño del modelo más una perdida del rendimiento limitada. Esta optimización permitirá conocer si existe una relación entre reducción del tamaño y mejora de latencia al implementarse en la aplicación, así como evaluar si existe una degradación importante en la precisión del modelo.

Se seguirán varias opciones de optimización basándonos en el siguiente árbol de decisión:

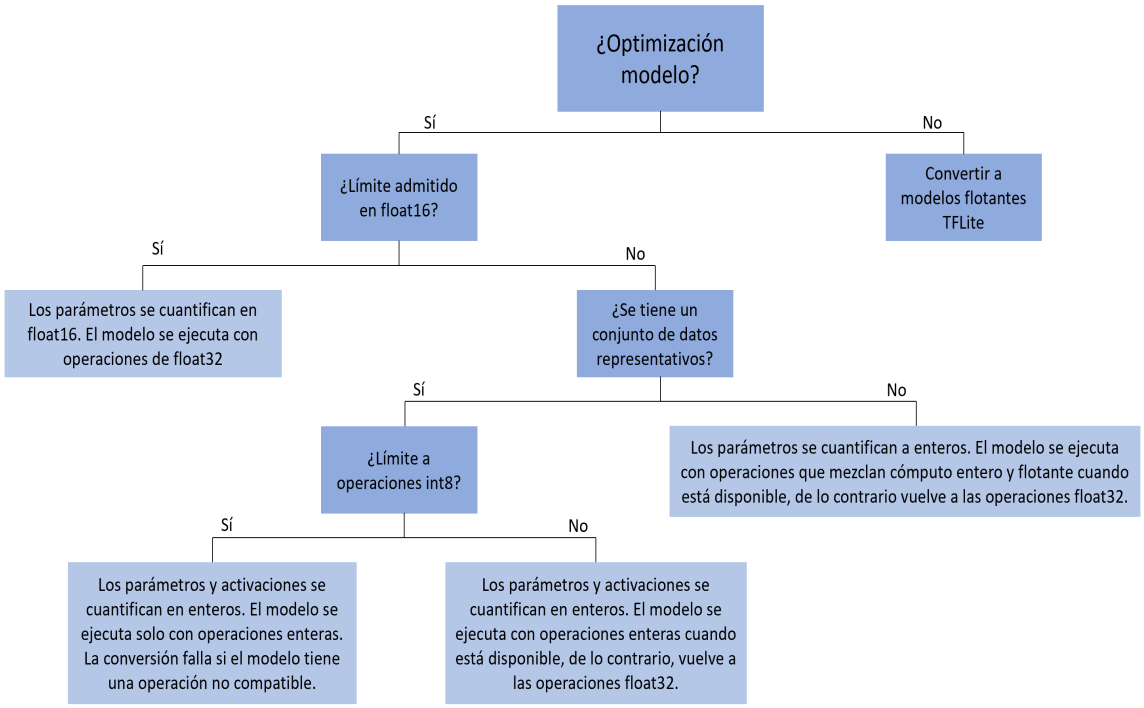


Figure 4.7: Árbol de decisión en clasificación

El árbol de decisión puede ayudar a determinar qué método de cuantificación posterior al entrenamiento es mejor para el caso que se quiera emplear, conociendo que hay una dependencia de las limitaciones que se puedan encontrar en el propio modelo con precisión de coma flotante preentrenado de tensorflow. Así, y dependiento del grado de cuantifiación que se llegue a realizar, el modelo se comportará de una forma u otra (ver Figura 4.7), obteniendo por lo tanto diversos esquemas de cuantificación.

También asumiremos que nuestro harware del teléfono no posee ningún límite, de hecho lo ideal teóricamente sería una optimización cuanto más profunda mejor ya que el tamaño del archivo tenderá a disminuir aunque ya se verá como puede afectar en el posterior rendimiento. Entonces, será necesario tomar diferentes técnicas de cuantización con el fin de efectuar sobre los modelos de clasificación las diferentes opciones de optimización. Dichas técnicas se pasarán a comentar a continuación:

- El primero se trataría de una **no optimización** del modelo, es decir, convertir el modelo al formato *.tflite* quedando pesos y activaciones en su tamaño original de float32 (números de coma flotante de 32 bits).
- El siguiente sería una **cuantización de pesos a float16** (números de coma flotante de 16 bits) reduciendo así el tamaño del modelo a la mitad.
- El tercero se trataría de una **cuantización de pesos a enteros de 8 bits** cuantificando estáticamente solo los pesos desde punto flotante hasta 8 bits de precisión, disminuyendo el tamaño del modelo en casi 4 veces el original.
- Finalmente, la última técnica con la que se trabajará será la **cuantización completa de pesos y activaciones** con el objetivo fijado de obtener mejoras en latencia cuantizando también las operaciones del modelo original. Para realizar esta técnica se tendrá que medir el rango dinámico de activaciones y entradas mediante un conjunto de datos representativo. Este conjunto de datos representativos que se le proporcionará será un puñado pequeño del conjunto de imágenes de validación de ImageNet, concretamente el usado en la competición de ImageNet de reconocimiento visual a gran escala del 2012 (ILSVRC 2012). Esto se hace con el foco puesto en que las imágenes representativas tengan el mismo formato que el conjunto de datos con los que el modelo está familiarizado (usa el mismo rango de datos) y posea un estilo similar (no necesita contener todas las clases). Este conjunto de datos representativo permite que el proceso de cuantificación mida el rango dinámico de activaciones y entradas, lo cual es crítico para encontrar una representación precisa de 8 bits de cada peso y valor de activación.

Como comentario general, especificar que TensorFlow dejará automáticamente en dimensión de coma flotante aquellas operaciones dentro de la red que no tengan implementaciones cuantificadas, permitiendo así que que la conversión se realice sin problemas. Además decir que la cuantización completa de pesos y activaciones se ejecutará sobre las tres primeras técnicas explicadas anterioremente, para obtener una visión más general de todas las posibilidades, aunque sin perder de vista el árbol de decisión que tendrá el proyecto en clasifiación.

4.4.2 Detección de objetos

En esta subsección se presenta el desarrollo de la adaptación de los modelos en detección con sus respectivos pasos

4.4.2.1 Elección de modelos preentrenados

El repositorio de TensorFlow [55] proporciona una colección de modelos de detección previamente entrenados en el conjunto de datos COCO. Sin embargo, actualmente y como ya se ha ido comentando (Secciones 4.2.2 y 4.3.2) se tiene una limitación ya que sólo serán viables el uso de los modelos con arquitectura de red SSD para su integración en la aplicación.

Por otro lado, dependiendo cómo hayan sido definidos los niveles de cuantificación de los pesos de los modelos durante el entrenamiento (float o cuantizado), se encontrarán en la descarga diferentes archivos y directorios, por lo que los pasos para su conversión al formato *tflite* serán diferente dependiendo del nivel de cuantificación de los pesos.

En la siguiente Tabla 4.2 se muestran los modelos usados, con una entrada general de *300x300* con tres canales RGB (rojo, verde, azul) por píxel.

Modelos disponibles válidos SSD		
Modelo	Tipo de Modelo	Nivel de cuantificación de los pesos
MobileNetV2	Ligera	Entero con signo de 8 bits
MobileNetV2	Ligera	Punto flotante de precisión de 32 bits
MobileNetV1	Ligera	Entero con signo de 8 bits
MobileNetV1	Ligera	Punto flotante de precisión de 32 bits
MobileNetV1-0.75	Ligera	Entero con signo de 8 bits
InceptionV2	Común	Punto flotante de precisión de 32 bits

Table 4.2: Modelos disponibles válidos en detección

4.4.2.2 Conversión modelos y optimización

Dependiendo de:

- Si el modelo está cuantizado (nivel de cuantificación de los pesos de entero con signo de 8 bits), en las descarga del modelo se encontrará un fichero llamado *tflite_ graph.pb* (con pesos congelados del modelo compatibles para convertir a tflite).
- Si el modelo por el contrario no está cuantizado (modelos con pesos floatantes de 32 bits), en la descarga del modelo no aparecerá el fichero *tflite_ graph.pb*, por lo que es necesario su transformación a éste a través de los ficheros *pipeline.config* (configuración fichero) y *model.ckpt* (checkpoint del modelo) que se nos proporcionan.

Una vez se tiene todos los modelos en su formato compatible con TFLite Converter para la conversión, se hará uso de *Bazel* por ser una herramienta de compilación que puede utilizar los materiales que proporciona TFLite Converter (*TOCO* que optimiza los modelos para permitir que se ejecuten eficientemente en TensorFlow Lite, además en este caso sin inconvenientes por falta de madurez) y mediante líneas de comando se permitirá la conversión desde *protocol buffer* (.pb) como un modelo guardado hasta llegar al formato *.tflite*. Ahora, se tiene que hacer una diferenciación entre los modelos en float y los cuantizados.

- Por un lado, un modelo donde su proceso previo sea la cuantización, sólo podrá tener un tipo de inferencia de *int8*, lo que en clasificación llamabamos **cuantización de pesos a enteros de 8 bits**.
- Por otro lado cuando se trate de un modelo de coma flotante, el tipo de inferencia sólo puede ser float32 lo que en clasificación llamabamos **no optimización** y se tendrá que especificar en la conversión.

En ambos casos, la conversión tendrá una normalización del tensor de la imagen de entrada después de cambiar el tamaño de cada cuadro de imagen de la cámara a *300x300* píxeles.
En este caso, y en comparación a lo visto en clasificación, (árbol de decisión, Figura 4.7) no nos haremos la pregunta de si se tienen un conjunto de datos representativos, ya que *TOCO* de momento no incorpora está función para poder introducir imágenes representativas durante la conversión de los modelos en detección.

Capítulo 5

Evaluación

5.1 Introducción

Este capítulo presenta una serie de pruebas de evaluación en base a analizar el rendimiento de diferentes modelos de redes convolucionales ligeras entre sí, además de enfrentarlos a redes más complejas. Para ello, se evaluarán en términos de eficiencia para las tareas de estudio haciendo una separación en:

- El rendimiento propio tanto de clasificación o detección, con el uso de las métricas y bases de datos existentes en cada tarea.
- Rendimiento de la carga computacional evaluando el tiempo de procesado requerido por las diferentes redes en el uso de las aplicaciones móviles desarrolladas.

La elección de dichos modelos para las tareas de clasificación y detección de objetos fueron mencionados en el capítulo anterior (Capítulo 4) y serán mediante los cuales se realizarán los diferentes análisis. Sin embargo, los modelos de análisis se aumentarán para la tarea de detección en el estudio del rendimiento propio (Sección 5.4.1). Esto se hará para enfrentar los modelos SSD con diferentes arquitecturas, con el fin de dar una visión más global del rendimiento de las arquitecturas de los modelos en esta tarea.

5.2 Entornos de experimentación

Para la evaluación de los experimentos se hará una diferenciación en cuanto al estudio que se realizará. En primer lugar, para el caso del rendimiento de cada tarea con el uso de las métricas de clasificación y detección, el empleo de un ordenador u otro para hacer las pruebas no es relevante para este caso ya que el resultado será el mismo.

Sin embargo, cuando se trate del rendimiento computacional del tiempo de procesado de los diferentes modelos, si que se necesitará conocer las características técnicas de cada móvil en el que se implementarán las aplicaciones para así evaluar la influencia de éstas en la ejecución de cada aplicación. En la siguiente tabla (Tabla 5.1) se da a conocer dichas especificaciones de los dos móviles con los que se ha tenido acceso para la ejecución de las pruebas.

Modelo	Procesador	Memoria RAM	Sistema Operativo
Samsung Galaxy S6 edge+	Exynos 7420 2.1GHz	4GB	Android 7.0
Huawei P8 Lite	Kirin 625 1.3GHz	2GB	Android 6.0

Table 5.1: Especificaciones móviles

5.2.1 Diseño de pruebas para evaluación

La explicación del diseño de las pruebas experimentales tendrá diferenciación según la tarea de evaluación y el término de eficiencia a evaluar.

5.2.1.1 Diseño según el rendimiento de las métricas de clasificación

En un primer momento, el modelo se cargará, siendo aquí necesario obtener sus correspondientes tensores del formato *.tflite*. Esto se realizará con la API de Python.

El número de imágenes en evaluación que se tendrán serán 1000 imágenes, siendo todos los modelos evaluados con las mismas 1000 imágenes. Dichas imágenes fueron elegidas de forma aleatoria de las imágenes de validación de ImageNet y concretamente del Large Scale Visual Recognition Challenge 2012 (ILSVRC2012). Se cree que son suficientes imágenes con los que tener una visión muy aceptable del correspondiente rendimiento ya que el margen de error encontrado es mínimo, con muy poca diferencia cuando se enfrentan a los resultados utilizando todo el dataset de validación. Posteriormente, se hará un tratamiento con dichas imágenes para incorporarlos como datos de entrada en la evaluación de los modelos.

Los modelos devolverán como respuesta las etiquetas de las clases más probables de cada una de las imágenes de validación que se le pase. Así, para relacionar la salida del modelo de clasificación será necesario la comparación con el *ground truth* de validación, ya que allí es donde se podrá comprobar si la clasificación que cada modelo otorga es válida o no.

A continuación, el fin será el cálculo de la exactitud de un problema multiclase de clasificación. En términos de redes neuronales, esta exactitud representa la relación entre el número de predicciones correctas frente al número total de predicciones hechas.

ImageNet en clasificación se guía por la **exactitud top-1 y top-5** y serán las métricas que se utilizarán a lo largo de la evaluación. La exactitud top-1 representa el acierto de la respuesta del modelo (clasificación con mayor probabilidad) en comparación con la respuesta esperada. Mientras que, por su parte la exactitud top-5 tiene en cuenta que alguna de de las cinco respuestas con mayores probabilidades coincida con la respuesta esperada.

5.2.1.2 Diseño según el rendimiento de las metricas de detección

El modelo se cargará y será testeado sobre 1000 imágenes (mismo número que en clasificación), siendo todos los modelos evaluados con las mismas 1000 imágenes. Dichas imágenes fueron elegidas de forma aleatoria del dataset de validación de COCO, concretamente las correspondientes al año 2014. Al igual que para clasificación, el margen de error encontrado es mínimo cuando se enfrentan a los resultados utilizando todo el dataset de validación.

Se podrán cargar todos los modelos, excepto aquellos modelos que estén cuantizados, ya que en la descarga de éstos, no se encuentra un fichero específico en formato *.pb*, el cual resulta necesario para la ejecución de las pruebas.

El fichero resultante de la evaluación estará en formato *json* y es donde se guardan los resultados de las evaluaciones de los modelos siguiendo el formato de COCO:

["image_id": int, "category_id": int, "bbox": [x, y, width, height], "score": float]

Así por cada detección, se tendrá el id de la imagen, la categoría a la que pertenece, los bounding boxes de la posición del objeto y el score que posee. A continuación, para obtener los resultados del rendimiento de las métricas, se compararán por un lado el fichero de resultados generado por cada modelo frente al *ground truth* de validación de la API de COCO. Finalmente, la respuesta de esta comparación serán las diferentes métricas de COCO.

Para entender las métricas que se van a utilizar es necesario definir previamente:

- **IoU (Intersección sobre Unión):** Se utiliza para decidir si una predicción es correcta o no para un objeto. Se define como la intersección entre el área predicha y la real dividido por el área de unión. Así sabiendo que B_p hace referencia a los cuadros delimitadores predichos y B_{gt} a los cuadros delimitadores correctos del *ground truth*, la ecuación será:

$$IoU = \frac{Area(B_p \cap B_{gt})}{Area(B_p \cup B_{gt})} \tag{5.1}$$

- **Precisión y Recall:** La precisión es la probabilidad de que los cuadros delimitadores predichos coincidan con los cuadros delimitadores de verdad, también conocido como el valor predictivo positivo. Por su parte, recall o exhaustividad hace referencia a la tasa positiva verdadera, también conocida como sensibilidad, que mide la probabilidad de que los objetos del *ground truth* se detecten correctamente. Por lo que sus ecuaciones son:

$$Precision = \frac{TP}{TP + FP} = \frac{\#Verdaderos\_Positivos}{\#Predicciones\_modelo} \tag{5.2}$$

$$Recall = \frac{TP}{TP + FN} = \frac{\#Verdaderos\_Positivos}{\#Ground\_Truths\_Positivos} \tag{5.3}$$

Fijando diferentes umbrales puede construirse una curva de precisión frente exhaustividad. En donde el área que se encuentra por debajo de esta curva se considera como precisión media. También se puede construir una curva de exhaustividad para estos diferentes umbrales.

En COCO se calculan las métricas **AP (precisión media)** y **AR (exhaustividad media)**, promediadas en las 80 categorías, por lo que COCO no hace distinción entre AP y mAP (precisión media promedio) e igualmente ocurre con AR y mAR. También y a menos que se especifique lo contrario COCO promedia sobre múltiples valores de IoU. Esto se hace para recompensar a los detectores con una mejor localización. Por ejemplo, IoU=0.5:0.95 significa que habrá diez umbrales (0.5 a 0.95, en pasos de 0.05), mientras que en IoU=0.5 sólo habrá un umbral. Además, todas las métricas se calculan permitiendo como máximo 100 detecciones de puntaje máximo (salvo que se indique lo contrario) por imagen.

Si una detección con una clase determinada tiene un IoU mayor a un *umbral* con un *ground-truth* de esa clase, entonces se considerará como una correcta detección.

Finalmente las métricas que se utilizarán a lo largo de la evaluación serán:

- Average Precision (AP) @[IoU=0.50:0.95 | area= all | maxDets=100] (%)
- Average Precision (AP) @[IoU=0.50 | area= all | maxDets=100] (%)
- Average Recall (AR) @[IoU=0.50:0.95 | area= all | maxDets=100] (%)

De las dos métricas de precisión media comentadas, en las gráficas se mostrarán las correspondientes a un IoU=0.5 ya que es común tomarla como la representativa en desafíos de deteccion como por ejemplo VOC.

5.2.1.3 Diseño según el rendimiento de la carga computacional

En este caso, se añadirá un historial del *log* en las aplicaciones tanto de clasificación como en detección de objetos. Es decir, una vez se tenga el modelo cargado en las aplicaciones y éstas instaladas en el móvil, se podrá monitorizar el tiempo de inferencia que consiguen los modelos en los diferentes dispositivos móviles (Tabla 5.1) para las diferentes tareas.

5.3 Evaluación comparativa del rendimiento en clasificación

En este apartado, se exponen los resultados obtenidos junto con sus correspondientes interpretaciones para la tarea de clasificación.

5.3.1 Métricas estado del arte

En la siguiente gráfica se muestran las métricas de exactitud top-1 y top-5 junto con el tamaño de los correspondientes modelos para tener una visión general de los resultados obtenidos.

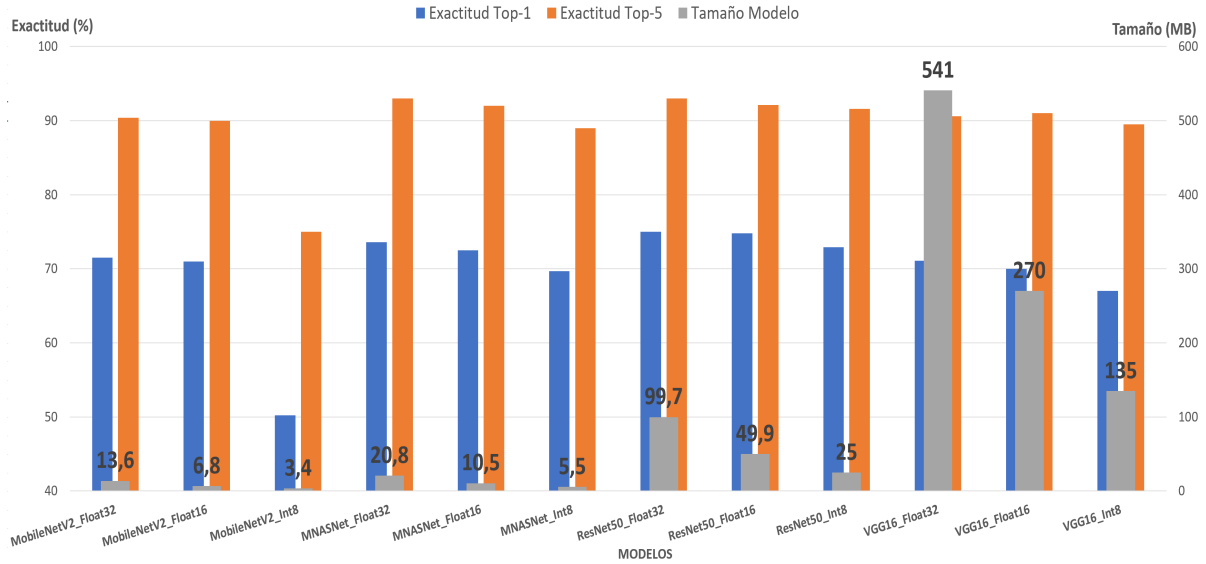


Figure 5.1: Evaluación General Clasificación

Como se puede observar en la gráfica (Figura 5.1), se muestra los cuatro modelos con los que se han realizado las pruebas (MobileNetV2, MNasNet, ResNet50, VGG16) y cada uno de ellos en sus tres niveles de cuantificación de los pesos (float32, float16 e int8).

A estos modelos de la Figura 5.1 no se les ha añadido ninguna imagen representativa a la hora de hacer la conversión a tflite. Además, se añade el tamaño de los modelos impreso en la misma gráfica. Esto se hace con el fin de tener una mejor visión sobre todo para el caso de los modelos ligeros ya que su diferencia de tamaño frente a los modelos comunes es considerable. Esta diferencia llega a ser de un 97,5% en su caso más extremo si comparamos a MobileNetV2 (modelo ligero con el menor tamaño) con VGG16 (modelo común con el mayor tamaño) y ambos con en mismo nivel de cuantificación de float32.

Si analizamos los dos modelos comunes de la Figura 5.1, vemos que MNasNet obtiene mejores resultados en sus métricas y con un tamaño alrededor de 441MB menor en comparación con VGG16 y ambos con el mismo nivel de cuantificación de float32. Por lo tanto y siguiendo este análisis, en la siguientes gráficas (Figuras 5.2 y 5.3) se pasará a comparar las exactitudes con respecto al tamaño de los modelos, pero sin tener en cuenta VGG16 por tres motivos. El primero porque VGG16 no representa una mejora en las métricas con respecto a los modelos ligeros, obteniendo unos resultados muy similares por ejemplo con respecto al modelo MNasNet. El segundo porque ya tenemos un modelo común que es el que obtiene los mejores resultados y con el que comparar respecto a los modelos ligeros. Y finalmente para no saturar la gráfica y favorecer la visualización de las diferencias en los modelos.

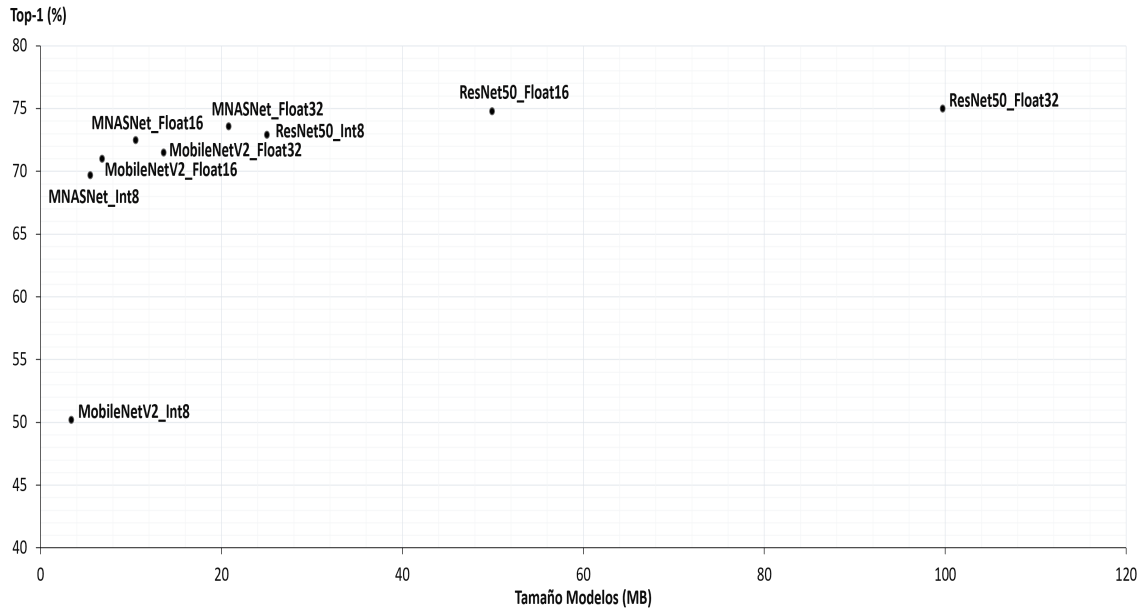


Figure 5.2: Exactitud Top-1 frente al tamaño de los modelos

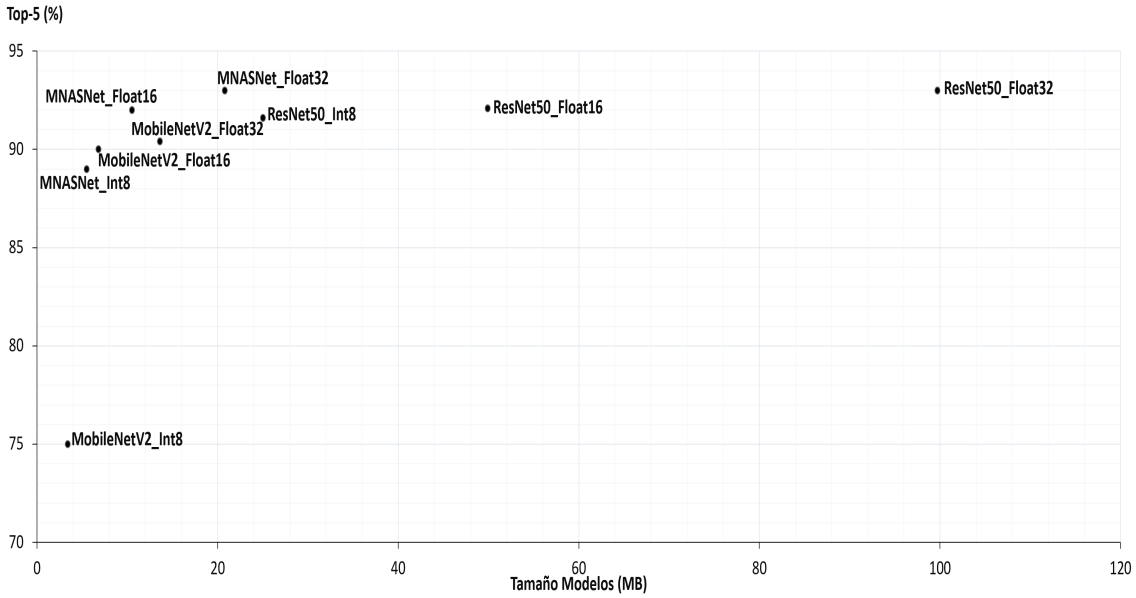


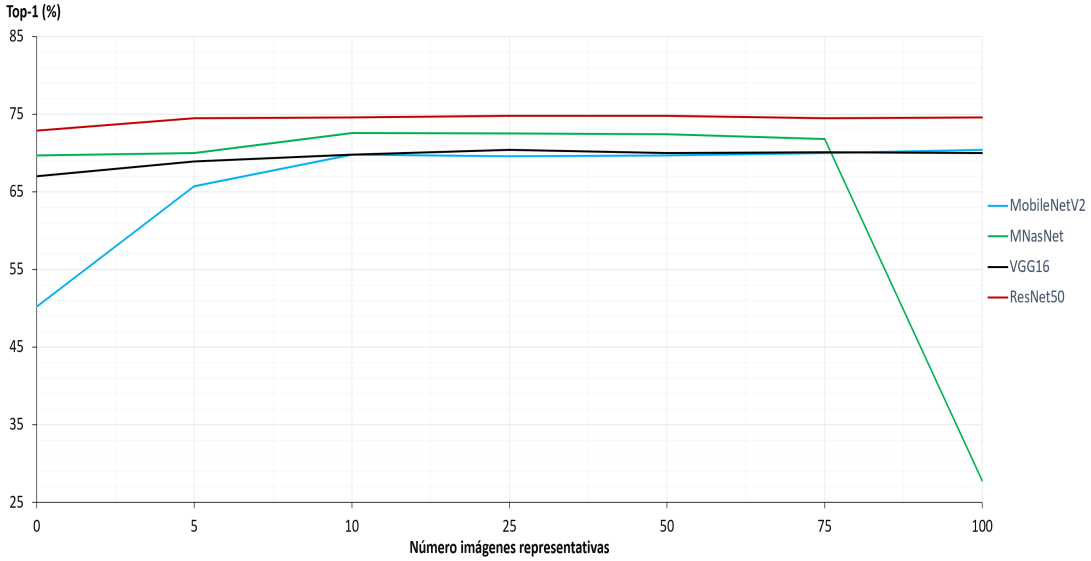
Figure 5.3: Exactitud Top-5 frente al tamaño de los modelos

A la vista de las Figuras 5.2 y 5.3:

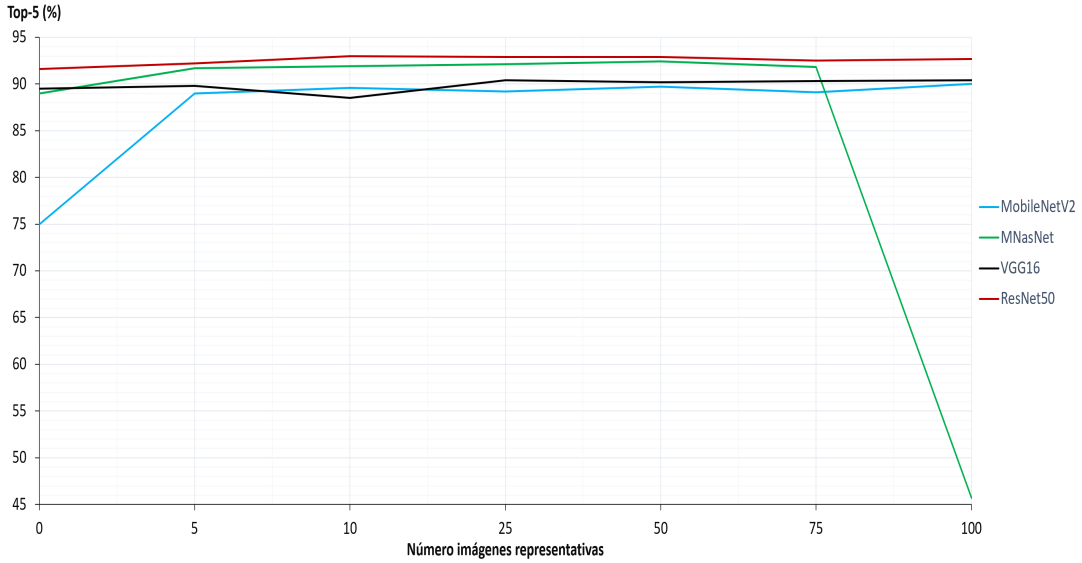
- El tamaño de los modelos disminuye a la mitad en cada paso de cuantificación.
- A medida que se disminuye el nivel de cuantificación en los pesos, el porcentaje de exactitud en las métricas disminuye paulatinamente, pero es una disminución ligera para el paso de float32 a float16. Sin embargo, el cambio de los modelos a int8 hace que la disminución del porcentaje de precisión sea más notorio esta vez, sobre todo en el caso de MobileNetV2, en donde se observa una disminución en torno al 21% en top-1 y de un 15% en top-5.
- Observamos que los modelos ligeros en general tienen una exactitudes similares al modelo común. Sin embargo, es cierto que ResNet50 obtiene los mejores resultados en top-1, pero a costa de unos tamaños en la red altos, Esto se hace perceptible cuando el modelo ResNet50 llega a 99,7 MB (con el valor máximo de los pesos) cuando por ejemplo MobileNetV2 en un nivel de cuantificación de enteros de 8 bits posee un tamaño de 3,4 MB, manifestando una diferencia grande de los tamaños.
- Con este primer análisis podríamos concluir que MNasNet en su versión de pesos de enteros de 8 bits es el modelo que aun siendo un modelo ligero tiene unas precisiones más cercanas a los modelos comunes.

Por otra parte, ahora se comprueba si la incorporación de imágenes representativas ayuda de cara a la mejora de los resultados. El número de imágenes representativas irá variando para comprobar la evolución de las métricas de precisión a medida que el número de imágenes aumentan. Dichas imágenes, serán elegidas de forma aleatoria de las mismas imágenes de validación de ImageNet (ILSVRC2012), pero los modelos serán evaluados con las mismas imágenes en cada momento.

En primer lugar, se hará la comprobación en los modelos con cuantificación de enteros de 8 bits ya que han sido donde la disminución de la exactitud ha sido más pronunciada, sobre todo en el caso de MobileNetV2. Así, obtenemos las Figura 5.4.



(a) Top-1



(b) Top-5

Figure 5.4: Evolución exactitud de los modelos en cuantificación de enteros de 8 bits con la inclusión de imágenes representativas durante la conversión

A partir de las Figuras 5.4 se puede deducir que:

- En los modelos existe una ligera mejora en las métricas de exactitud. Dichas mejoras se dan con pocas imágenes representativas y luego tiende a estabilizarse en torno a 10 imágenes. Además, en el caso de MobileNetV2 la mejora es muy pronunciada tanto para top-1 como para top-5.
- Se observa una descenso inusual en MNasNet cuando al modelo se le aumenta en 100 imágenes representativas en ambas métricas.

En la siguiente tabla (Tabla 5.2) se hace patente el caso inusual que sufre el modelo MNasNet en el nivel de cuantificación de 8 bits cuando se le añaden más de 100 imágenes de calibración. Dicha tabla muestra los tres niveles de cuantificación del modelo MNasNet añadiendo más imágenes representativas en la evaluación.

Modelo	Nivel cuantificación	Imágenes representativas	Top-1	Top-5
MNasNet	Int8	25	72,6%	91,9%
		75	72,7%	92%
		100	**27,8%**	**45,7%**
		150	**27,2%**	**44,8%**
		250	**27,2%**	**43,3%**
	Float16	25	72,8%	92%
		75	72,8%	92,2%
		100	72,9%	92,1%
		150	72,8%	92%
		250	72,9%	92,3%
	Float32	25	73,6%	93,1%
		75	73,6%	93%
		100	73,7%	93%
		150	73,6%	93,1%
		250	73,6%	93,1%

Table 5.2: Evaluación MNasNet con imágenes representativas en la conversión

Tras lo observado en la Tabla 5.2 se trató de realizar una segunda prueba con el mismo número de imágenes pero esta vez utilizando imágenes capturadas con el propio móvil. En estas pruebas se obtuvieron peores resultados top-1 y top-5, y además volvió a suceder lo mismo que en las primeras pruebas, es decir, cuando se le añaden más de 100 imágenes representativas al modelo MNasNet con nivel de cuantificación de enteros de 8 bits, dichos valores de exactitud sufren una repentina e inesperada gran disminución en los resultados.

Junto con una investigación, se decidió concluir que dicho descenso inusual de MNasNet puede deberse a problemas de juventud en la incorporación del modelo a la plataforma de Keras. De hecho, *Coral* [56], la propia plataforma de Google, que proporciona herramientas para desarrollar y prototipar productos de inteligencia artificial encontró resultados inexactos después de cuantificar algunos modelos Keras, de entre ellos y a día de la evaluación de MNasNet, ésta no constaba dentro de la lista en la que se pueda verificar una correcta conversión con el uso de la cuantificación posentrenamiento.

Por otro lado, se comprobó también la evolución que tienen los modelos con niveles de cuantificación de float32 y float16 a medida que se añaden imágenes representativas en la conversión (al igual que lo visto en la Figura 5.4 para la cuantificación de 8 bits). Sin embargo, apenas se han encontrado variaciones en los valores de las métricas, las cuales se han mantenido constantes en todo momento.

La siguiente tabla resume los más destacado que se ha sido visto y comentado anteriormente, mostrando los valores exactos en cada caso.

Modelo	Nivel cuantificación	Imágenes representativas	Top-1	Top-5	Tamaño
MobileNetV2	Int8	0	50,2%	75%	**3,4MB**
		10	**69,8%**	**89,6%**	
	Float16	0	71%	90%	6,8MB
		10	71,2%	90,1%	
	Float32	0	71,5%	90,4%	13,6MB
		10	71,5%	90,5%	
MNasNet	Int8	0	69,7%	89%	5,5MB
		10	72,6%	91,9%	
	Float16	0	72,5%	92%	10,5MB
		10	72,7%	92%	
	Float32	0	73,6%	93%	20,8MB
		10	73,6%	93,1%	
ResNet50	Int8	0	72,9%	91,6%	25MB
		10	74,6%	93%	
	Float16	0	74,8%	92,1%	49,9MB
		10	74,9%	92,6%	
	Float32	0	75%	93%	99,7MB
		10	75%	93,3%	
VGG16	Int8	0	67%	89,5%	135MB
		10	69,8%	89%	
	Float16	0	70%	91%	270MB
		10	70%	91,2%	
	Float32	0	71,1%	90,8%	541MB
		10	71,2%	91,2%	

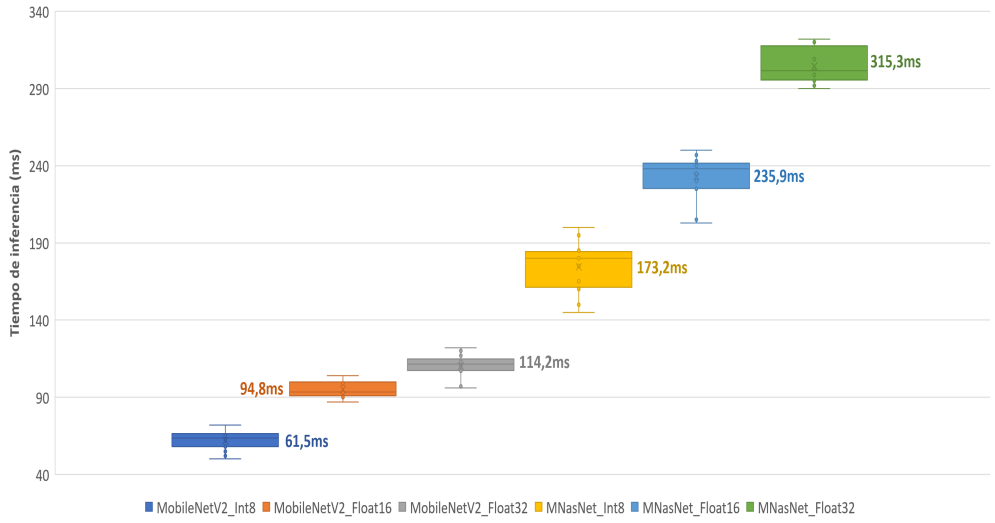
Table 5.3: Resumen evaluación de las métricas en clasificación

En la Tabla 5.3, y como resumen de lo visto en clasificación, se destaca a MobileNetV2 como modelo más ligero y con unas métricas de precisión similares a los demás. Además, los modelos enteros totalmente cuantificados tienen una precisión comparable con sus versiones flotantes de 32 bits (por ejemplo MobileNetV2 pierde en torno a 2% en top-1 y un 1% en top-5), aunque su ventaja de cara a las cargas computacionales en la aplicación es que las versiones de enteros de 8 bits cuentan con un tamaño del modelo que llega a ser 4 veces menor que el tamaño original.

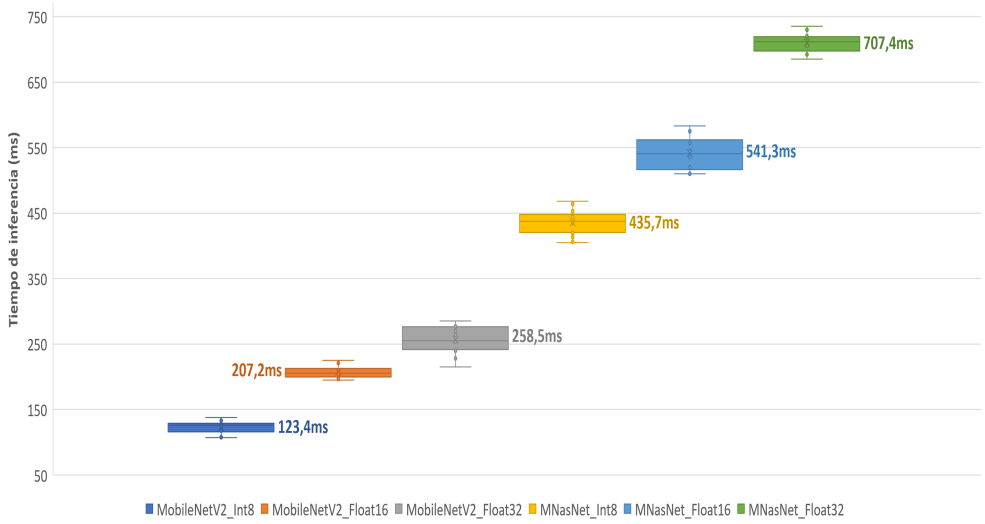
Este modelo, a costa de poseer el menor tamaño, pierde 4,8% en top-1 y un 3,4% en top-5. Estas diferencias fueron calculadas con respecto al modelo común que mejores resultados de métricas obtiene (ResNet50) frente a MobileNetV2 y ambos en las versiones de enteros de 8 bits con la inclusión de 10 imágenes representativas. Por lo tanto, y en consonancia con el anterior análisis, se puede decir que los modelos ligeros en clasificación conservan gran parte de los resultados de las métricas a pesar de poseer tamaños reducidos con respecto a los modelos comunes.

5.3.2 Carga computacional en app

En esta prueba, se han empleado todos los modelos en sus tres diferentes niveles de cuantificación, escogiendo aquellos que utilizan diez imágenes representativas durante la conversión a *.tflite*. Esto se debe a que con el uso de imágenes representativas se tienden a tener los mejores resultados teóricos (Tabla 5.3). También, la evaluación se realizará sobre las mismas condiciones de entorno y con la misma duración de uso de la aplicación para la captura de los datos. Con todo esto, nos aseguramos obtener una evaluación más rigurosa y comparable al tener las mismas condiciones. Finalmente, los modelos han sido evaluados en los dos móviles disponibles (Tabla 5.1) para poder realizar una comparación de los datos según las especificaciones del dispositivo de integración. Así, tras la evaluación del log de salida de la clasificación de los modelos en los dispositivos, se han obtenido las siguientes gráficas:



(a) Samsung



(b) Huawei

Figure 5.5: Tiempos de inferencia de los modelos en clasificación

En la Figura 5.5 se muestran todos los modelos con los que ha sido posible instalar la aplicación en el móvil y donde no se ha tenido ningún problema a la hora de testearlos. En cada uno de ellos, se visualizan los intervalos con sus valores mínimos y máximos, además de los tiempos con mayor frecuecia de inferencia (zona coloreada de cada modelo). También, se añade el valor de la media de los tiempos en cada modelo para así tener una mejor visualización de los resultados.

Observando los tiempos de inferencia de los modelos se comprueba que en ambos terminales móviles, el modelo que obtiene tiempos más bajos es el modelo MobileNetV2 y por lo tanto, es el modelo más rápido, eficiente y que mejor funciona en la aplicación de clasificación. Además, lo consigue en su versión más ligera de cuantización de pesos en enteros de 8 bits, lo cual tiene sentido ya que al poseer un menor tamaño, en el móvil interactúa de mejor manera y eso se refleja en los datos.
Continuando con el análisis, podemos deducir que las especificaciones del dispositivo en el que se integran los modelos es importante a la hora de obtener los resultados ya que los tiempos de inferencia son considerablemente más altos en todos los modelos para el móvil Huawei que es el que peores especificaciones técnicas posee en comparación con el móvil Samsung (como se pudo comprobar en la Tabla 5.1, donde el dispositivo Samsung tiene mejor procesador, el doble de memoria RAM y una versión del sistema operativo Android mayor con respecto al móvil Huawei). Así, los datos reflejan que pueden llegar a ser 2 o 2.5 veces mejores los resultados del tiempo de inferencia en el móvil Samsung dependiendo del modelo que se escoja.

Posteriormente, los modelos no ligeros (ResNet50 y VGG16) no han sido posible evaluarlos en esta sección. Recordemos que según lo visto en el Capítulo 2 las arquitecturas de los modelos no ligeros no están pensadas para la integración posterior en un móvil debido a sus tamaños, necesitando requisitos computacionales y de almacenamiento mayores. La tabla 5.4 muestra los inconvenientes que han tenido estos modelos a la hora de incorporarlos en ambos móviles.

Nivel cuantificación	Tiempos de inferencia	Inconvenientes surgidos
Enteros	>2000ms	A pesar de poder instalarlo en los móviles, a la hora de observar las probabilidades, éstas poseían probabilidades de 1% - 2%.
Flotantes	-	Imposibilidad de instalarlo en los móviles. La memoria de los dispositivos no resulta suficiente para correr estos modelos.

Table 5.4: Modelos comunes en carga computacional tarea de clasificación

A la vista de esta tabla queda claro que los modelos no ligeros no están en disposición de ser añadidos a un termina móvil para la tarea de clasificación.

5.4 Evaluación comparativa del rendimiento en detección

En este apartado, se exponen los resultados obtenidos junto con sus correspondientes interpretaciones para la tarea de detección.

5.4.1 Métricas estado del arte

En la siguiente gráfica se muestran las métricas de precisión media y exhaustividad media junto con el tamaño de los correspondientes modelos para tener una visión general de los resultados obtenidos.

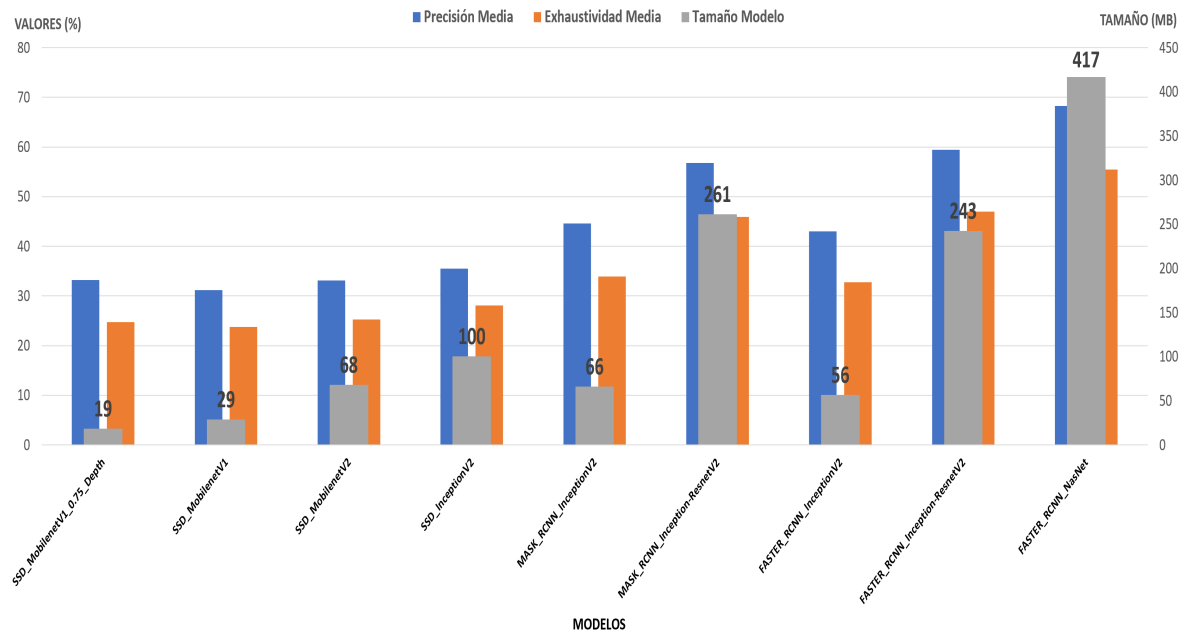


Figure 5.6: Evaluación General Detección

TensorFlow ofrece un abanico extenso de modelos que se pueden descargar con varias arquitecturas diferentes, de las cuáles en la Figura 5.6 se han escogido las más características con cuantifiación flotante de 32 bits ya que como se comentó en la Sección 5.2.1.2 no se podrán evaluar en las métricas de detección los modelos que estén cuantizados en enteros de 8 bits.

Es cierto que dependiendo del modelo, habrá un tamaño diferente, pero en general se puede observar que las redes SSD poseen un tamaño considerablemente menor en comparación con Mask_RCNN y Faster_RCNN por las diferencias en arquitectura contadas en el estado del arte (Sección 2.4.2). Esta diferencia llega a ser de un 95,4% en su caso más extremo si comparamos a SSD_MobilenetV1_0.75_Depth con FASTER_RCNN_NasNet. También se comprueba que SSD son los modelos que ofrecen los resultados más bajos, en comparación a las redes de dos fases.

En las siguientes gráficas se hará una comparación tanto de la precisión media como de la exhaustividad media con respecto al tamaño de cada modelo.

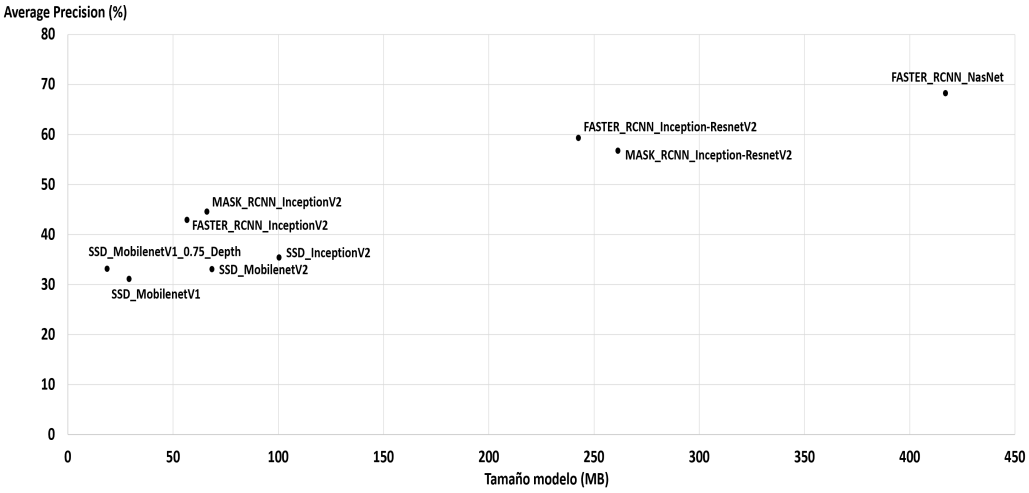


Figure 5.7: Precisión media de los modelos de detección

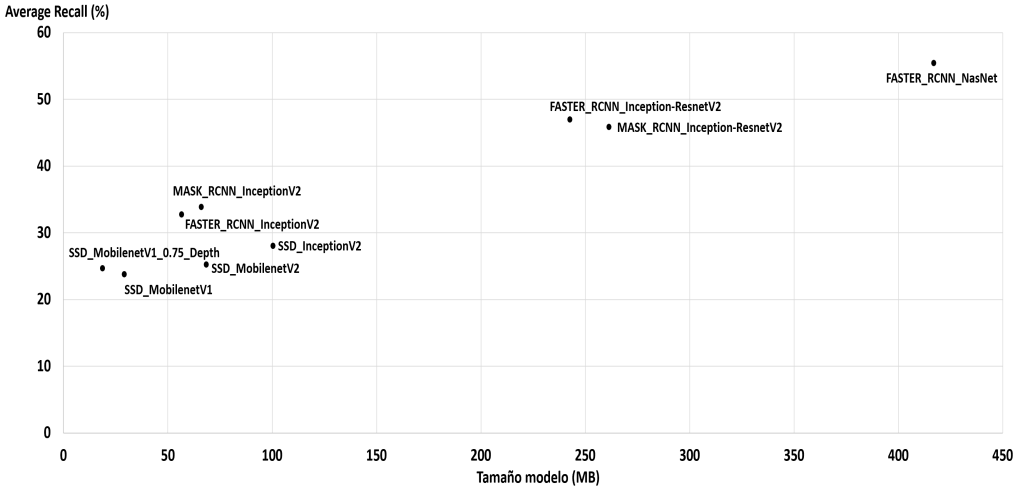


Figure 5.8: Exhaustividad media de los modelos de detección

En las gráficas anteriores (Figuras 5.7 y 5.8) se observa un comportamiento similar en los datos de los modelos en ambas métricas, pudiendo observar que:

- FASTER_RCNN_NasNet es el modelo que obitene mejores resultados, pero con una tamaño del modelo que supera los 400MB.
- A medida que un modelo ahorro en tamaño, inversamente se está produciendo una perdida considerable tanto de la precisión como de la exhaustividad. Por ejemplo en precisión existe una diferencia de un 35% y en exhaustividad de un 30,8% entre el modelo con mayor tamaño frente al de menor tamaño. Por lo tanto, las diferencias de rendimiento con respecto a los modelos grandes es muy considerable, algo que no pasaba en clasificación.
- Los modelos con arquitecturas SSD obtienen valores similares entre sí. De los que destaca, SSD_MobilenetV1 en su versión de 0.75 de profundidad, siendo el modelo con un tamaño menor de todos los evaluados y con unos resultados interesantes.

Esto último se podrá comprobar en la siguiente tabla resumen de los modelos con arquitecturas SSD (Tabla 5.5) que son los que se podrán evaluar más adelante en la Sección 5.4.2.

Estos modelos se examinarán con más detalle al añadir la métrica AP(IoU=0.50:0.95) de evaluación que aún no se ha observado en las gráficas de detección. Dicha métrica es común tomarla como la representativa de COCO [57] y con ella, nos haremos una idea más general de los diferentes valores de precisión de cada modelo. Además esta métrica recompensará a los modelos con una mejor localización al promediar entre diferentes IoU (tal y como se explicó en la Sección 5.2.1.2).

Modelo	AP(IoU=0.50:0.95)	AP(IoU=0.50)	AR(IoU=0.50:0.95)	Tamaño
MobileNetV1_0.75	20,9%	33,2%	24,7%	19MB
MobileNetV1	22%	31,2%	23,8%	29MB
MobileNetV2	23,5%	33,1%	25,3%	68MB
InceptionV2	26,5%	35,5%	28,1%	100MB

Table 5.5: Resumen evaluación arquitectura SSD de las métricas en detección

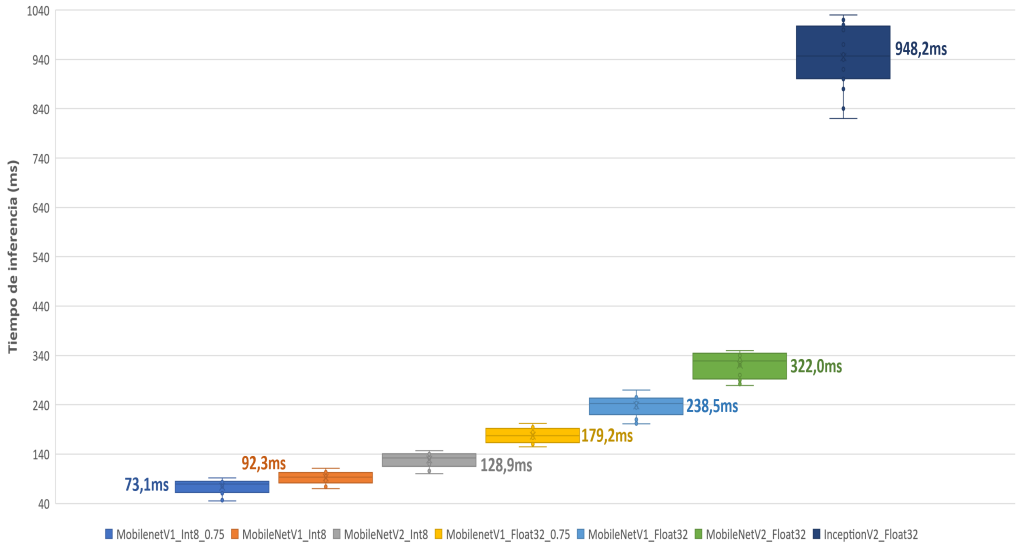
Destacar que de entre las diferentes versiones de MobileNet, aquella con una profundidad menor, su tamaño se ve reducido, que es lo que realmente hace el hiperparámetro α tal y como se explicó en la Sección 2.5.3. Esto puede dejar entrever una posibilidad de conseguir buenos resultados en la carga computacional. Además, se observa que para MobileNetV2 se tienen los mejores datos de las tres versiones de dicho modelo, sin embargo su tamaño aumenta de forma notable (39 MB con respecto a MobileNetV1), lo cual no justifica unos resultados en torno a un 2% de mejora en las métricas. Finalmente, el modelo no ligero, es decir, InceptionV2 logra los mejores resultados, pero es el modelo con mayor tamaño de los cuatro, superando en 81MB al modelo MobileNetV1 en su versión de 0.75 de profundidad.

La métrica AP(IoU=0.50:0.95) comprueba que a medida que el IoU aumenta (de 0.5 a 0.95 en pasos de 0.05), la precisión en la detección se vuelve más desafiante y es por eso que en dicha métrica se encuentra la mayor diferencia de valores (exactamente de un 5,6%) entre el modelo común (InceptionV2) y el modelo más ligero (MobileNetV1 en su versión de 0.75 de profundidad).
Por su parte, en las métrica AP(IoU=0.5) y AR(IoU=0.5:0.95) se encuentran unas diferencias de 2,3% y 3,4% respectivamente, las cuales son competitivas si tenemos en cuenta que hay una reducción de 81MB en los tamaños. Por lo tanto, se podría concluir que con la misma red SSD, en el modelo más ligero no se observa una disminución tan grande como en las comparativas donde se añadían diferentes tipos de redes y modelos con tamaños muy superiores (visto en las Figuras 5.6, 5.7 y 5.8).

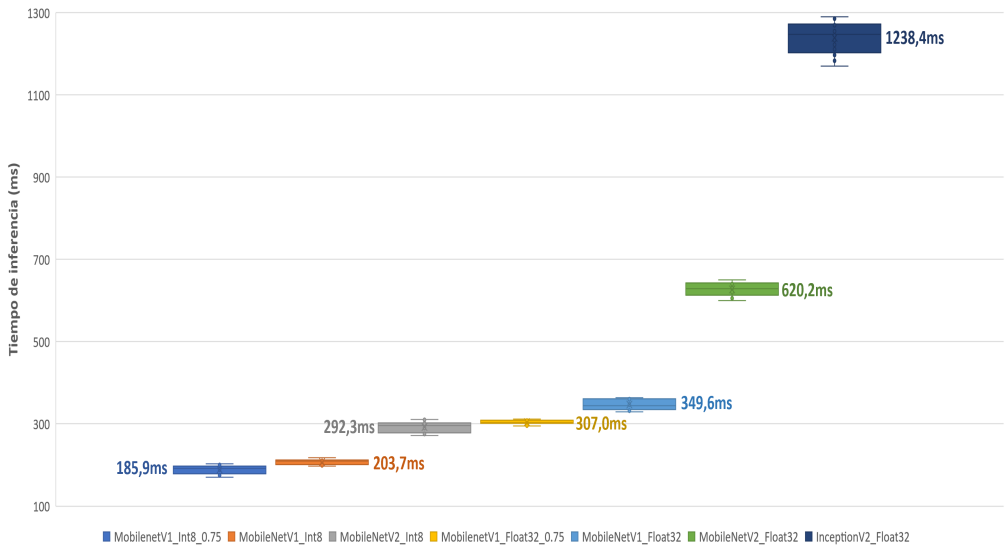
5.4.2 Carga computacional en app

Para esta prueba, se han utilizado los modelos disponibles en detección (comentado en Tabla 4.2) con arquitectura SSD y sus dos niveles de cuantificación disponibles (flotante de 32 bits y entero de 8 bits). Tal y como se ha ido comentando a lo largo de la memoria del proyecto, este hecho conlleva una reducción del tamaño en una relación de aproximadamente 4 veces cuando se compara el mismo modelo en estos dos niveles de cuantificación.

La evaluación se realizará sobre las mismas condiciones de entorno y con la misma duración de uso de la aplicación para la captura de los datos. Los modelos han sido evaluados en los dos móviles disponibles para poder realizar una posterior comparación entre las especificaciones de los terminales móviles. Así, tras la evaluación del log de salida de la detección de los modelos en los dispositivos, se han obtenido las siguientes gráficas:



(a) Samsung



(b) Huawei

Figure 5.9: Tiempos de inferencia de los modelos en detección

En la Figura 5.9 se muestran los modelos junto con la media de los tiempos para cada uno de estos modelos con el fin de tener una mejor visualización de los resultados. Observando los tiempos de inferencia de los modelos se comprueba que en ambos terminales móviles, el modelo que obtiene tiempos más bajos es el modelo MobileNetV1 en su versión cuantizada en enteros de 8 bits y con una profundidad de 0.75. Por lo tanto, es el modelo más rápido y que mejor

funciona en el dispositivo móvil en términos de eficiencia computacional.

A pesar de que en MobileNet, la versión 2 consigue mejores resultados en las métricas de detección (Tabla 5.5 para modelos con niveles de cuantificación flotantes de 32 bits), en la anterior figura se observa que sus tiempos de inferencia son mayores, debido en gran medida por poseer un tamaño del modelo mayor en comparación a sus otra versiones.

También, es claro la gran diferencia en términos de tiempos de inferencia que hay entre el modelo común (InceptionV2) en comparación con los modelos ligeros evaluados. Por ejemplo en el caso más extremo, la diferencia llega a ser de un 92% para el dispositivo Samsung y de un 85% para Huawei. Esto evidencia que la respuesta para el modelo común no es en tiempo real (entendiendo este tiempo como la respuesta inmediata según la percepción del usuario cuando se interactúa con la aplicación).

Finalmente, las especificaciones del dispositivo en el que se integran los modelos son importantes a la hora de obtener los resultados al igual que lo fue en clasificación. Los tiempos de inferencia son más altos en todos los modelos para el móvil Huawei, señalando que pueden llegar a ser entre 1.5 a 2.5 veces mejores los resultados en el móvil Samsung dependiendo del modelo evaluado.

Capítulo 6

Conclusiones y Trabajo Futuro

6.1 Conclusiones

En este trabajo, se ha presentado el desarrollo de aplicaciones móviles de clasificación y detección de objetos, siendo una primera aproximación al desarrollo de tareas de visión artificial en plataformas móviles con redes convolucionales. Además, se ha empezado a crear un marco de trabajo para la conversión de modelos con el fin de que se puedan utilizar en estas aplicaciones móviles. Dicho marco, posee también un enfoque analítico, en el cual se puede hacer una primera valoración del rendimiento de eficiencia y de complejidad computacional para las tareas concretas de clasificación y de detección de objetos.

Para este cometido, se han estudiado las redes neuronales desde sus aspectos más fundamentales hasta las investigaciones más recientes sobre redes neuronales convolucionales ligeras, las cuales se encuentran destinadas al uso en aplicaciones con recursos limitados como son los dispositivos móviles. Así mismo, el entorno de trabajo utilizado se ha basado en la integración de diversas herramientas que se encuentran en constante actualización. De entre ellas, destacar las limitaciones principales que se han ido encontrando en el desarrollo, como en el caso de detección, donde la plataforma actual de TensorFlow solo acepta modelos SSD.

Las aplicaciones de clasificación y detección tienen la capacidad de incluir distintos modelos siempre que se encuentren correctamente convertidos al formato *tflite*. Sin embargo, los resultados concluyen que, son las redes convolucionales ligeras las que destacan en su desempeño en el móvil al ser modelos pequeños y rápidos en ambas tareas (sobre todo cuánto mayor sea el nivel de cuantificación de estos modelos).
Además, concretamente en la tarea de clasificación estos modelos ligeros mantienen unas métricas de exactitudes muy competitivas, perdiendo menos de un 5% en top-1 y top-5 pero llegando a reducir su tamaño un 97,5%. Una parte de que se tengan estas métricas de clasificación se debe a añadir imágenes representativas durante la conversión ya que ayuda en la mejora de los resultados.
Mientras que en detección hay dos escenarios en los modelos con niveles de cuantificación flotante de 32 bits. El primero, la evaluación general cuando se comparan los modelos con diferentes arquitecturas de red, en donde los resultados en las métricas de detección llegan a ser más de un 30% peores para los modelos ligeros, manifestando unas diferencias importantes del rendimiento con respecto a los modelos grandes aunque el modelo más ligero llegue a tener un tamaño un 95,4% menor. Por otro lado, si sólo se comparan las redes de una fase (SSD) que son las que

se pueden incorporar en el móvil, las diferencias decrecen al volver a enfrentar al modelo más ligero con el modelo común, concretame en las métricas AP(IoU=0.50:0.95), AP(IoU=0.5) y AR(IoU=0.5:0.95) un 5,6%, 2,3% y 3,4% respectivamente, las cuales son competitivas si tenemos en cuenta que hay una reducción de un 81% en el tamaño del modelo.

De igual manera, hay que tener en cuenta la importancia de las especificaciones del dispositivo a la hora de obtener los resultados ya que se concluye que hay una relación directa entre los tiempos de inferencia y las especificaciones técnicas que posea el terminal, obteniendo de media resultados de los tiempos de inferencia dos veces mejores en las tareas para el dispositivo móvil Samsung.

Se concluye que los modelos no ligeros, en clasificación no están en disposición de ser añadidos a un terminal móvil, mientras que en la tarea de detección los tiempos rondan los 1000ms, llegando a no obtener una respuesta inmediata según la percepción del usuario. Como conclusión final, destacar MobileNet como el modelo eficiente que manifiesta una buena relación entre el rendimiento de las métricas y los tiempos de inferencia en ambas tareas, concretamente MobileNetV2 en clasificación y MobileNetV1 con una profundidad de 0.75 para detección y ambos con niveles de cuantificación de enteros de 8 bits.

6.2 Trabajo Futuro

Siguiendo con la tendencia de trabajo del proyecto, se pueden plantear varias líneas de trabajos futuras. En primer lugar, en las aplicaciones se podría integrar modelos propios que no sean preentrenados, aprovechando el marco de trabajo de conversión que se ha realizado. Como posible idea, en vez de estar los modelos preentrenados con las clases de Imagenet o COCO, se podría hacer *fine-tuning* de estos modelos para otras aplicaciones concretas dependiendo de las necesidades, es decir, si te interesa clasificar o detectar otras categorías de objetos, pudiendo así modificar la aplicación y el modelo de acorde a las necesidades concretas del usuario.

Además, otra vía de estudio sería la extensión de las aplicaciones a otras tareas del campo de la visión artificial como por ejemplo la segmentación de imágenes o la estimación de poses. Paralelamente a esto, y a medida que los modelos vayan evolucionando, se podría hacer un análisis más exhaustivo con más redes y obtener conclusiones más generales y específicas.

Problemas como el obtenido en clasificación, que imposibilitaba la implementación oficicial de la aplicación de TensorFlow debido a diferentes incoherencias en su uso o el impedimento de evaluación de los modelos cuantificados en cuanto a las métricas de detección por la falta de archivos en la descarga, son inconvenientes que se podría intentar hacer una investigación totalmente profunda para tratar de resolver estos problemas de adaptación.

Bibliografía

[1] A. Ng and K. Katanforoosh, "Cs229 lecture notes deep learning," 2018. xiii, 6

[2] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016. xiii, 1, 8, 13

[3] J. Lemley, S. Bazrafkan, and P. Corcoran, "Deep learning for consumer devices and services: Pushing the limits for machine learning, artificial intelligence, and computer vision.," *IEEE Consumer Electronics Magazine*, vol. 6, no. 2, pp. 48–56, 2017. xiii, 6, 9

[4] "Redes neuronales convolucionales: *Tres cosas que es necesario saber*." https://la.mathworks.com/solutions/deep-learning/convolutional-neural-network.html. Accessed: 2020-04-22. xiii, 10

[5] W. Rawat and Z. Wang, "Deep convolutional neural networks for image classification: A comprehensive review," *Neural computation*, vol. 29, no. 9, pp. 2352–2449, 2017. xiii, 12

[6] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 580–587, 2014. xiii, 14

[7] R. Girshick, "Fast r-cnn," in *Proceedings of the IEEE international conference on computer vision*, pp. 1440–1448, 2015. xiii, 14, 15

[8] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," in *Advances in neural information processing systems*, pp. 91–99, 2015. xiii, 15

[9] J. Dai, Y. Li, K. He, and J. Sun, "R-fcn: Object detection via region-based fully convolutional networks," in *Advances in neural information processing systems*, pp. 379–387, 2016. xiii, 15

[10] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "Ssd: Single shot multibox detector," in *European conference on computer vision*, pp. 21–37, Springer, 2016. xiii, 16

[11] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and< 0.5 mb model size," *arXiv preprint arXiv:1602.07360*, 2016. xiii, 17

[12] M. Tan, B. Chen, R. Pang, V. Vasudevan, M. Sandler, A. Howard, and Q. V. Le, "Mnasnet: Platform-aware neural architecture search for mobile," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2820–2828, 2019. xiii, 2, 19, 20

[13] T. J. Sejnowski, *The deep learning revolution*. Mit Press, 2018. 1

[14] L. Liu, W. Ouyang, X. Wang, P. Fieguth, J. Chen, X. Liu, and M. Pietikäinen, "Deep learning for generic object detection: A survey," *International journal of computer vision*, vol. 128, no. 2, pp. 261–318, 2020. 1

[15] A. Sengupta, Y. Ye, R. Wang, C. Liu, and K. Roy, "Going deeper in spiking neural networks: Vgg and residual architectures," *Frontiers in neuroscience*, vol. 13, 2019. 1

[16] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998. 1, 13

[17] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014. 1, 13

[18] W. Chen, J. T. Wilson, S. Tyree, K. Q. Weinberger, and Y. Chen, "Compressing convolutional neural networks," *arXiv preprint arXiv:1506.04449*, 2015. 1

[19] Y.-D. Kim, E. Park, S. Yoo, T. Choi, L. Yang, and D. Shin, "Compression of deep convolutional neural networks for fast and low power mobile applications," *arXiv preprint arXiv:1511.06530*, 2015. 1, 2

[20] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 4700–4708, 2017. 2

[21] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017. 2, 18

[22] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 4510–4520, 2018. 2

[23] "Android studio: *Tools for building apps*." https://developer.android.com/studio. Accessed: 2020-03-12. 2, 22

[24] "Bazel: *Build and test software*." https://bazel.build/. Accessed: 2020-05-25. 2, 23

[25] "Tensorflow: *An end-to-end open source machine learning platform*." https://www.tensorflow.org/. Accessed: 2020-04-07. 2

[26] "Tensorflow lite: *Deploy machine learning models on mobile and IoT devices*." https://www.tensorflow.org/lite/. Accessed: 2020-05-19. 2, 22

[27] "Keras: *The python deep learning library*." https://keras.io/. Accessed: 2020-04-07. 2, 22

[28] T. Huang, "Computer vision: Evolution and promise," 1996. 4

[29] M. I. Jordan and T. M. Mitchell, "Machine learning: Trends, perspectives, and prospects," *Science*, vol. 349, no. 6245, pp. 255–260, 2015. 4, 5

[30] R. Szeliski, *Computer vision: algorithms and applications*. Springer Science & Business Media, 2010. 5

[31] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016. 5, 6, 10, 11

[32] D. Kriesel, "A brief introduction on neural networks," 2007. 6, 8

[33] A. P. Engelbrecht, *Computational intelligence: an introduction*. John Wiley & Sons, 2007. 7

[34] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, pp. 436–444, 2015. 10

[35] M. D. Zeiler and R. Fergus, "Visualizing and understanding convolutional networks," in *European conference on computer vision*, pp. 818–833, Springer, 2014. 10

[36] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Backpropagation applied to handwritten zip code recognition," *Neural computation*, vol. 1, no. 4, pp. 541–551, 1989. 10

[37] A. Rosebrock, *Deep Learning for Computer Vision with Python: ImageNet Bundle*. PyImageSearch, 2017. 11, 12

[38] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, pp. 1097–1105, 2012. 13

[39] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1–9, 2015. 13

[40] J. R. Uijlings, K. E. Van De Sande, T. Gevers, and A. W. Smeulders, "Selective search for object recognition," *International journal of computer vision*, vol. 104, no. 2, pp. 154–171, 2013. 14

[41] K. He, G. Gkioxari, P. Dollár, and R. Girshick, "Mask r-cnn," in *Proceedings of the IEEE international conference on computer vision*, pp. 2961–2969, 2017. 16

[42] K. Nan, S. Liu, J. Du, and H. Liu, "Deep model compression for mobile platforms: A survey," *Tsinghua Science and Technology*, vol. 24, no. 6, pp. 677–693, 2019. 17

[43] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pp. 265–283, 2016. 21

[44] N. Ketkar, "Introduction to keras," in *Deep learning with Python*, pp. 97–111, Springer, 2017. 21

[45] N. Ketkar, "Introduction to pytorch," in *Deep learning with python*, pp. 195–208, Springer, 2017. 21

[46] J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, T. Darrell, Y. Jia, and E. Shelhamer, "Caffe: Convolutional architecture for fast feature embedding," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Citeseer, 2014. 21

[47] "Imagenet is an image database." http://www.image-net.org/. Accessed: 2020-04-11. 23

[48] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*, pp. 248–255, Ieee, 2009. 23

[49] "Coco: *Common Objects in Context*." http://cocodataset.org/. Accessed: 2020-04-14. 24

[50] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft coco: Common objects in context," in *European conference on computer vision*, pp. 740–755, Springer, 2014. 24

[51] "*Tensorflow Lite object detection task on mobile*." https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/running_on_mobile_tensorflowlite.md. Accessed: 2020-04-07. 27, 28

[52] "Tensorflow: *Android Neural Network API*." https://developer.android.com/ndk/guides/neuralnetworks. Accessed: 2020-04-07. 28

[53] "Repository for the *TensorFlow for poets 2* series of google codelabs." https://github.com/googlecodelabs/tensorflow-for-poets-2. Accessed: 2020-02-03. 28

[54] "Keras: *The python deep learning library applications*." https://keras.io/applications/. Accessed: 2020-05-18. 31

[55] "Tensorflow detection model zoo: *Collection of detection models*." https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/detection_model_zoo.md/. Accessed: 2020-05-11. 33

[56] "Coral: *Coral with Tensorflow Lite and keras models*." https://coral.ai/docs/edgetpu/models-intro. Accessed: 2020-05-11. 42

[57] "Coco: *Metrics*." http://cocodataset.org/#detection-eval. Accessed: 2020-06-10. 48