Trabajo Fin de Máster
Máster en Ingeniería Electrónica, Robótica y Automática

Implementación de redes neuronales en Raspberry Pi 3 con Movidius Neural Compute Stick

Autor: Antonio José Toro Valderas
Tutores: Daniel Gutiérrez Reina
Sergio Toral Marín

Departamento de Ingeniería Electrónica
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2020

Trabajo Fin de Máster
Máster en Ingeniería Electrónica, Robótica y Automática

Implementación de redes neuronales en Raspberry Pi 3 con Movidius Neural Compute Stick

Autor:
Antonio José Toro Valderas

Tutores:
Daniel Gutiérrez Reina
Sergio Toral Marín

Departamento de Ingeniería Electrónica
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2020

Trabajo Fin de Máster: Implementación de redes neuronales en Raspberry Pi 3 con Movidius Neural Compute Stick

Autor: Antonio José Toro Valderas

Tutores: Daniel Gutiérrez Reina

Sergio Toral Marín

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2020

El Secretario del Tribunal

Agradecimientos

A mis padres, por haberme dado todo lo necesario para mi formación, por haberme apoyado siempre durante toda mi vida y porque sin ellos no sería la persona quién soy hoy en día. Por haberme enseñado a ser mejor persona.

A mis abuelos paternos, que siempre me han apoyado y han cuidado de mí, sobre todo cuando era pequeño. Por cuidar de mí cuando mis padres no podían hacerlo y por tener toda la paciencia del mundo conmigo cuando me hice mayor.

A mi novia, por haber estado siempre junto a mí, apoyándome y animándome tanto en momentos buenos como en momentos malos. Por hacerme reír siempre, por estar siempre tan pendiente de mí y por animarme siempre con la realización del presente proyecto.

A mis compañeros del máster, que hicieron que la etapa académica fuese mucho más divertida. Por ser unas grandes personas que te animaban a ir a clase y por estar siempre dispuestos a ayudarte con cualquier duda por absurda que fuera.

A mis amigos, por apoyarme siempre y confiar en mí, por ayudarme siempre en mis malos momentos, por hacerme reír, por ser partícipes de tan buenos momentos y ser precisamente eso, buenos amigos.

A mis profesores, que me ayudaron a aprender y fueron grandes profesionales durante la duración de máster, y sobre todo a Daniel y Sergio, mis dos tutores en este trabajo que me han guiado de forma excepcional a lo largo del mismo.

Y en especial, a mi abuela materna, que siempre ha cuidado de mí desde el día que nací de forma excepcional. Por preocuparse de mí incluso cuando es ella la que está enferma. Porque, aunque ya no sea la misma persona que conocí, siempre formará parte de mí y siempre la recordaré con su personalidad original y, sobre todo, siempre recordaré los buenos y grandes momentos vividos con ella.

Antonio José Toro Valderas

Sevilla, 2020

Resumen

En este trabajo de fin de máster se realiza un estudio comparativo de la ejecución de redes neuronales profundas en un sistema embebido como es la Raspberry Pi. Para ello se cuenta con el Neural Compute Stick (NCS), una herramienta de desarrollo para inferencia de aprendizaje profundo con un consumo mínimo.

De esta forma, se pretende crear una red neuronal desde cero para clasificar imágenes y ejecutarla en la Raspberry Pi sin el stick y con el stick, y comparar así los resultados para ver si realmente dicho stick proporciona un aumento de rendimiento en un sistema embebido. Además, esta red neuronal se ejecutará también con el portátil para poner en contexto los resultados de la Raspberry Pi y del NCS respecto a una máquina con grandes recursos.

Por otra parte, también se pretende estudiar el tiempo de respuesta de las redes neuronales según la complejidad de la red. Para ello, se usan también algunas redes ya entrenadas por defecto, cada una de ellas con un número diferente de parámetros, y se ejecutarán en la Raspberry Pi sin el stick y con él, para comparar los resultados y ver cuál es el tiempo de respuesta según su complejidad. Dichas redes se ejecutarán también en el portátil para poner en contexto los resultados y comparar éstos también según la plataforma de ejecución de las redes neuronales.

Abstract

In this master thesis, a comparative study of the execution of deep neural networks in an embedded system such as the Raspberry Pi is carried out. With this aim, the Neural Compute Stick (NCS) will be used as, a development tool for deep learning inference with minimal energy consumption.

The overall aim is to create a neural network from scratch to classify images and run it on the Raspberry Pi with and without the stick, and thus compare the results to check to what extent the stick really improves the performance of an embedded system. In addition, this neural network will also be run on the laptop to put the results of the Raspberry Pi and the NCS in context with respect to a machine with larger resources.

As an additional aim, this work will also study the response time of neural networks according to the complexity of the network. Some well-known neural networks already trained by default will be used for this purpose, each one with a different number of parameters. They will also be run on the Raspberry Pi with and without the stick in order to compare the results and analyze how the response varies with the network complexity. These experiments will also be run on the laptop to put the results in context and compare them.

Índice

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

Notación

API	Interfaz de Programación de Aplicaciones
ASIC	Circuito Integrado de Aplicación Específica
BLOB	Objeto Binario Grande
CNN	Redes Neuronales Convolucionales
CPU	Unidad Central de Procesamiento
DLL	Biblioteca de Enlace Dinámico
GB	Gigabyte
GHz	Gigahercio
GPU	Unidad de Procesamiento Gráfico
HDMI	Interfaz Multimedia de Alta Definición
IA	Inteligencia Artificial
IoT	Internet de las Cosas
NCS	Movidius Neural Compute Stick
PCIe	Interconexión rápida de Componentes Periféricos
RAM	Memoria de Acceso Aleatorio
s	Segundos (tiempo)
SHAVE	*Streaming Hybrid Architecture Vector Engine*
TPU	Unidad de Procesamiento Tensorial
URL	Localizador de Recursos Uniforme
USB	Bus Universal en Serie
VPU	Unidad de Procesamiento de Visión
vs	*Versus*
<	Menor o igual
>	Mayor o igual

1 INTRODUCCIÓN

Como introducción al presente proyecto se expone en este primer apartado una breve introducción sobre la motivación del mismo y los objetivos a cumplir, de modo que se tenga una visión general del presente trabajo.

1.1 Motivación

Hoy en día el sector de la inteligencia artificial (IA) está cada vez más implantado en el mundo actual y cada vez más empresas y dispositivos se aprovechan de ella para ganar en innovación y tecnología. Dentro de la inteligencia artificial, uno de los elementos que más están destacando en estos últimos años son las redes neuronales, ya que con ellas se pueden realizar multitud de tareas en diversas áreas de aplicación.

Se conoce como redes neuronales a aquellos modelos computacionales que intentan comportarse de manera parecida a como lo hacen las neuronas humanas del cerebro, y es que en este modelo computacional existe un número determinado de neuronas artificiales que se conectan entre sí mediante unos enlaces, transmitiendo información de esta manera. De esta forma, la información de entrada de la red neuronal va pasando por todas las capas y neuronas artificiales hasta llegar a la capa de salida, donde se envía un resultado.

A lo largo de estos años se han llegado a entrenar redes neuronales con una gran cantidad de datos y parámetros sin ningún problema, por lo que el problema reside actualmente en minimizar el tiempo de inferencia para conseguir los resultados más rápidamente, sobre todo si estas redes neuronales se aplican en tareas donde la respuesta debe ser prácticamente inmediata.

Por otro lado, aparte de minimizar de forma general el tiempo de inferencia, ejecutarla en sistemas embebidos supone una gran limitación del rendimiento, ya que este tipo de sistemas disponen de una potencia computacional muy limitada. Esta limitación no es un gran impedimento en redes neuronales sencillas sin mucha complejidad o sin muchos cálculos, pero en redes más complejas no es suficiente y se necesita introducir alguna herramienta o dispositivo que ayude a realizar la computación. Debido a esto, las grandes compañías están poniendo un gran empeño para desarrollar dispositivos que aceleren la fase de inferencia de una red neuronal y que además favorezcan el paradigma *Edge Computing*, es decir, que el procesamiento y la computación se desarrolle cerca de estos sistemas embebidos para reducir el tiempo de respuesta y la latencia.

Por todo ello, la motivación del presente proyecto es la de usar uno de estos dispositivos aceleradores y poder afrontar el desafío de implementar redes neuronales en un dispositivo embebido con pocos recursos.

1.2 Objeto

El desarrollo de este trabajo de fin de máster tiene como objetivo principal comparar el rendimiento de las redes neuronales cuando se ejecuta en un dispositivo embebido como es la Raspberry Pi. Se compara el rendimiento sin el uso de ningún dispositivo acelerador de inferencia con el rendimiento al usar uno de ellos, en este caso el Movidius Neural Compute Stick. Además, también se ejecutarán estas redes neuronales en el portátil que se está usando, tanto con la CPU del mismo, como con una GPU de Google Colab, con el objetivo de poner los resultados en contexto con las diferentes plataformas de ejecución, y ver el rendimiento que puede extraerse con el uso del stick en la Raspberry Pi.

Para ello se va a crear una red neuronal desde cero que se encargará de clasificar imágenes. Se entrenará dicha red neuronal desde cero utilizando Python y el kit de herramientas OpenVINO, del que ya se hablará más adelante en el proyecto. Dicha red se encargará de clasificar dos categorías de imágenes, perros o gatos, ampliamente utilizada como *benchmark* a partir de los datos disponibles en kaggle. Se considerarán cuatro plataformas de ejecución:

- Raspberry Pi sin el uso del Neural Compute Stick
- Raspberry Pi con el uso del Neural Compute Stick
- CPU del portátil (Intel i7 de 8º generación)
- GPU de Google Colab

Una vez desplegada se tomará una media del tiempo de respuesta de la red en cada una de estas plataformas de ejecución, con el foco puesto en la mejora que supone el uso del NCS en la Raspberry Pi en dicho dispositivo, y se verá si su mejora de rendimiento se acerca al de un dispositivo con mayores recursos como es un ordenador.

Además de esta red neuronal creada desde cero, también se tiene como objetivo desplegar en estas cuatro plataformas de ejecución otras redes neuronales ya entrenadas por defecto para ver cuál es la respuesta del NCS según la complejidad de la red. Para ello se utilizarán otras tres redes neuronales usadas para clasificación de imágenes, teniendo cada una de ellas una complejidad distinta y, por tanto, un número de parámetros distinto. Estas tres redes son capaces de clasificar hasta mil categorías distintas pertenecientes al ILSVRC (*ImageNet Large Scale Visual Recognition Challenge*), el concurso anual de software de ImageNet, una de las más grandes bases de datos de imágenes. Como ocurrió anteriormente, también se observarán los resultados y el tiempo medio de clasificación de imágenes en todas las plataformas de ejecución poniendo el foco sobre el uso del NCS en la Raspberry. Se estudiará con estos resultados y con los anteriores la diferencia del tiempo de ejecución según la complejidad de la red y también se analizará la diferencia del tiempo de ejecución entre las distintas plataformas de ejecución dentro de una misma red.

Por último, se pretende hacer un diagrama de cajas con la respuesta en cada plataforma de ejecución de cada una de las redes para visualizar las diferencias de rendimiento de un modo más gráfico y se calculará en base a esos resultados obtenidos el intervalo de confianza al 95% para cada uno de dichos métodos de ejecución de cada red neuronal.

2 ESTADO DEL ARTE

En este capítulo se analiza el estado del arte de la implementación de redes neuronales en sistemas embebidos. Se estudia la situación actual y los últimos avances de los aceleradores de inferencias que favorecen el paradigma *Edge Computing*.

2.1 Edge Computing

El incremento de los dispositivos *IoT* en el perímetro de la red está produciendo una cantidad masiva de datos en los *data centers*. A pesar de las mejoras en la tecnología de la red, estos *data centers* no pueden garantizar una tasa de transferencia de datos aceptable, lo que podría ser un requisito fundamental para muchas aplicaciones [1]. Además, los dispositivos en el perímetro consumen constantemente datos que vienen de la nube, forzando a las compañías a construir una red de distribución de contenidos para descentralizar los datos y el aprovisionamiento de servicios. De forma similar, el objetivo del *Edge Computing* es mover la computación de estos *data centers* hacia el propio borde o perímetro de la red para mejorar el tiempo de respuesta y la tasa de transferencia de datos.

De esta forma, se entiende como *Edge Computing* un paradigma en el que los datos producidos por el Internet de las cosas (*IoT*) son procesados más cerca de la ubicación donde se produjeron, en vez de enviarlos a una nube, favoreciendo de este modo la rapidez de los resultados y la disminución de latencia [2].

Figura 1: Beneficios del *Edge Computing* [3]

Se trata de un paradigma muy importante sobre todo en aplicaciones en *IoT* que requieran una respuesta casi inmediata, como puede ser el caso de los automóviles inteligentes. El hecho de que se reduzca la latencia y se procesen los datos sin necesidad de esperar a que la nube lo haga y recogerlos se trata de un aspecto fundamental con el objetivo de evitar accidentes o mejorar el tiempo de respuesta ante imprevistos. Al igual que con el resto de los dispositivos del Internet de las cosas (*Internet of Things* o *IoT*), el automóvil debe procesar los datos en el menor tiempo posible.

También, gracias a que los datos son procesados cerca del dispositivo que los origina se puede decir que hay una mayor seguridad y confianza, puesto que no existe transferencia de datos desde la nube hasta el dispositivo (o entre varios dispositivos). Esto favorece también los costes, puesto que contratar soluciones en la nube no es necesario al realizarse los cálculos de forma local [3].

Como se ha podido comprobar, este paradigma tiene grandes beneficios, aunque sin duda el mayor de ellos es el de mejorar el tiempo de respuesta, puesto que la capacidad para agregar y analizar datos masivos in situ le permite tomar decisiones casi en tiempo real. Precisamente este aspecto del *Edge Computing* es el que se investiga actualmente para los métodos de *deep learning* con el objetivo de crear dispositivos y tecnologías que puedan actuar en los sistemas generadores de datos *IoT* o junto a ellos para que la respuesta sea inmediata, acelerando así la fase de inferencia de las redes neuronales.

Tal es así que incluso grandes compañías tecnológicas como Google, NVIDIA o Intel han decidido estos años investigar sobre esta temática con el fin de poder conseguir este objetivo.

2.2 Tecnologías y dispositivos embebidos aceleradores de IA

Como se ha comentado en el apartado anterior, existen grandes compañías tecnológicas que han desarrollado dispositivos o tecnologías para acelerar la fase de inferencia de las redes neuronales en el perímetro. En este apartado se hablará, por tanto, de esos productos. Concretamente de los productos de NVIDIA, Google e Intel, que son los más importantes y afianzados en el mercado a día de hoy.

2.2.1 NVIDIA

NVIDIA por una parte ha desarrollado TensorRT, un kit de desarrollo software para realizar en *deep learning* inferencias de alto rendimiento. Incluye un optimizador de inferencias para aplicaciones de *deep learning* y un sistema en tiempo de ejecución que proporciona una baja latencia y un alto rendimiento para la fase de inferencia [4].

Las aplicaciones basadas en TensorRT rinden mucho más rápido que las plataformas que solo se basan en CPU durante la inferencia. Con TensorRT se pueden optimizar redes neuronales entrenadas en la mayoría de *frameworks*, calibrarlas para una precisión más exacta con una alta precisión y finalmente desplegarlas en sistemas entre los que se incluyen los sistemas embebidos.

Está construido en CUDA, el modelo de programación paralela de NVIDIA, y permite optimizar la inferencia para todos los *frameworks* de *deep learning* aprovechando todas sus librerías, herramientas de desarrollo y tecnologías.

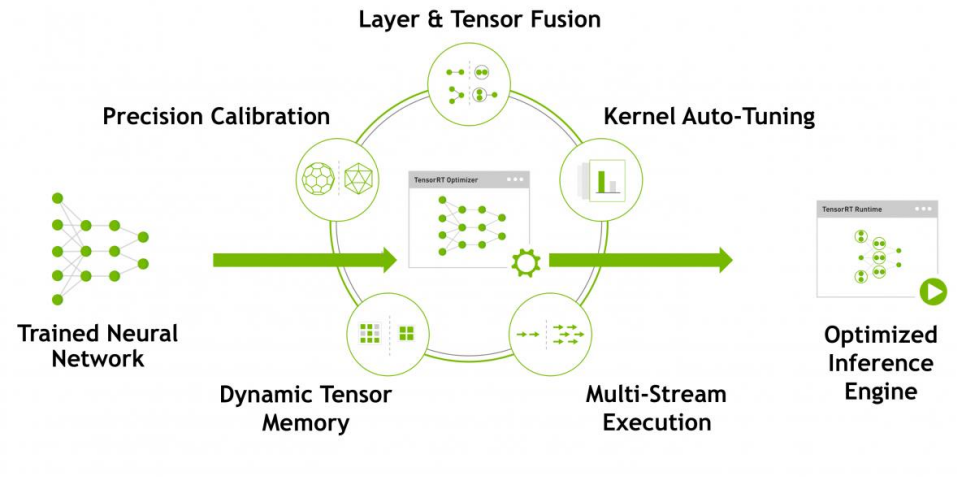

Figura 2: Esquema de optimización de inferencia de TensorRT [4]

Soporta optimizaciones de despliegue de producción para aplicaciones de inferencia de *deep learning* tales como video streaming, reconocimiento de voz, recomendaciones y procesamiento de lenguaje natural. Además, reduce de forma muy significativa la latencia de cualquier aplicación, lo cual es un requisito fundamental, como se comentó en el apartado anterior, para muchas aplicaciones en tiempo real que funcionan sobre dispositivos embebidos.

Para optimizar la fase de inferencia de un modelo, TensorRT toma la definición de la red, realiza optimizaciones (incluyendo optimizaciones a nivel de plataforma) y genera el motor de inferencia. Este proceso es conocido como la fase de construcción. Esta fase puede llevar un tiempo considerable, especialmente si se ejecuta en sistemas embebidos. Por lo tanto, una aplicación construirá dicho motor una vez, y después lo serializará para su uso posterior.

Se pueden importan modelos entrenados de TensorFlow, Caffe, PyTorch o MXNet a TensorRT. Después de aplicar las oportunas optimizaciones, TensorRT maximiza el rendimiento en las plataformas embebidas Jetson.

Y es que junto a Tensor RT, NVIDIA también ha desarrollado una familia de dispositivos embebidos llamados Jetson. Se tratan de sistemas embebidos de pequeña potencia diseñados para acelerar aplicaciones de aprendizaje automático.

De entre todas ellas la Jetson Nano es la más reciente. Se trata de un microordenador que permite ejecutar múltiples redes neuronales en paralelo para aplicaciones como clasificación de imágenes, detección de objetos, segmentación, y procesamiento de voz [5].

Figura 3: Jetson Nano [5]

La Jetson Nano cuenta con un slot para tarjeta microSD para almacenamiento principal, un puerto micro-USB, un encabezado de 40 pins, un puerto Ethernet, 4 puertos USB 3.0, salida HDMI, un jack para fuente de alimentación y un conector para módulos de cámaras.

2.2.2 Google

La alternativa de Google se llama Coral, un kit de desarrollo para construir productos con IA local similar a TensorRT. Las capacidades de inferencia de sus dispositivos permiten construir productos eficientes, privados y rápidos sin necesidad de conectarse a la red. Coral incluye de momentos dos dispositivos de prototipado.

En primer lugar, está la placa de desarrollo. Se trata de una placa para crear rápidamente prototipos de productos de aprendizaje automático en el dispositivo. Realiza inferencia de alta velocidad gracias al coprocesador Edge TPU de la placa, un circuito ASIC de Google diseñado a medida para generar inferencias en el perímetro.

Figura 4: Placa de desarrollo de Google Coral [6]

La placa se apoya en el *framework* TensorFlow Lite. Los modelos de este *framework* pueden ser compilados para que se ejecuten con el coprocesador Edge TPU [6].

La placa posee, además, una ranura para una tarjeta microSD, puerto USB 3.0, un puerto Gigabit Ethernet, terminales de 4 pin para altavoces, conector de audio analógico, salida de video HDMI 2.0 y un conector para

módulos de cámaras.

Por otra parte, Google también posee un acelerador USB. Se trata de un accesorio USB que realiza también inferencia a alta velocidad para aplicaciones de aprendizaje automático. El elemento clave es de nuevo el coprocesador Edge TPU que es el que provoca esta mejora de computación, ya que es capaz de realizar 4 billones de teraoperaciones por segundo (TOPS) [7].

Este acelerador funciona en la mayoría de plataformas. Se puede conectar por USB a cualquier sistema con Windows 10, con MacOS o incluso con Debian Linux (incluyendo en una Raspberry Pi); y como también sucede con la placa de desarrollo, funciona con TensorFlow Lite.

Figura 5: Acelerador USB de Google Coral [7]

Cabe destacar que Coral, además de estos dos dispositivos para crear prototipos, también tiene varios dispositivos PCIe para entornos de producción que simplemente permiten integrar fácilmente el procesador Edge TPU en sistemas existentes.

2.2.3 Intel

La baza de Intel se llama Movidius Neural Compute Stick (NCS), un dispositivo USB *plug-and-play* para acelerar la fase de inferencia en el perímetro de aplicaciones de inteligencia artificial y *deep learning*. Intel hasta el momento ha desarrollado este stick, y también su revisión, el Neural Compute Stick 2 (NCS2) [8].

Figura 6: Movidius Neural Compute Stick

Permite desarrollar y escalar con un rendimiento excepcional redes neuronales gracias a la Unidad de Procesamiento de Visión (VPU) que posee, un microprocesador específico para la aceleración de tareas de visión artificial (Movidius Myriad 2 para la primera revisión del NCS y Movidius Myriad X para el NCS2). La principal

diferencia entre la VPU Myriad 2 y la VPU Myriad X es el número de núcleos de procesamiento SHAVE que poseen (12 núcleos el NCS y 16 núcleos el NCS2). Estos núcleos poseen una arquitectura VLIW (*Very Long Instruction Word*), lo que favorece el paralelismo a la hora de la computación.

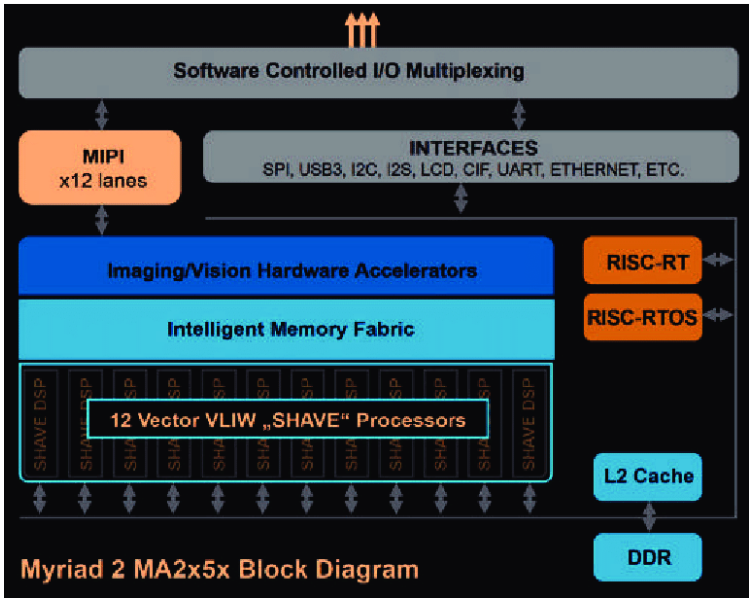

Figura 7: Diagrama de bloques del acelerador Myriad 2 (incluido en NCS)

Es capaz de operar sin necesidad de la computación en la nube, ya que realiza la inferencia en el dispositivo a tiempo real, y además permite elaborar prototipos en el perímetro con dispositivos de bajo consumo como la Raspberry Pi.

No necesita periféricos adicionales para empezar a desplegar soluciones. Además, soporta *frameworks* como TensorFlow, Caffe, Apache MXNet, ONNX, PyTorch y PaddlePaddle mediante conversión desde ONNX. Por último, destacar que posee conectividad USB 3.0 [8].

Junto con éstos sticks, Intel también ha desarrollado OpenVINO, un kit de herramientas que optimiza la inferencia de aplicaciones de *deep learning* basadas en redes neuronales convolucionales (CNN) en el perímetro. Se trata del soporte software de los dispositivos Neural Compute Stick y Neural Compute Stick 2 de los que se ha hablado anteriormente, además de otros sistemas hardware de Intel. Incluye librerías de funciones de visión artificial como OpenCV y kernels pre-optimizados, por lo que mejora el tiempo de desarrollo y acelera el rendimiento de los procesadores de Intel [9].

OpenVINO se compone principalmente de dos elementos:

- Un optimizador de modelos
- Un motor de inferencia.

El optimizador de modelos es una herramienta que facilita la transición entre los entornos de entrenamiento y despliegue de una red neuronal, realiza análisis estáticos de modelos y ajusta los modelos de *deep learning* para una ejecución óptima en los dispositivos finales de uso. Este optimizador asume que ya se tiene una red entrenada habiendo usado uno de los *frameworks* con los que es compatible (Caffe, TensorFlow, MXNet, Kaldi u ONNX).

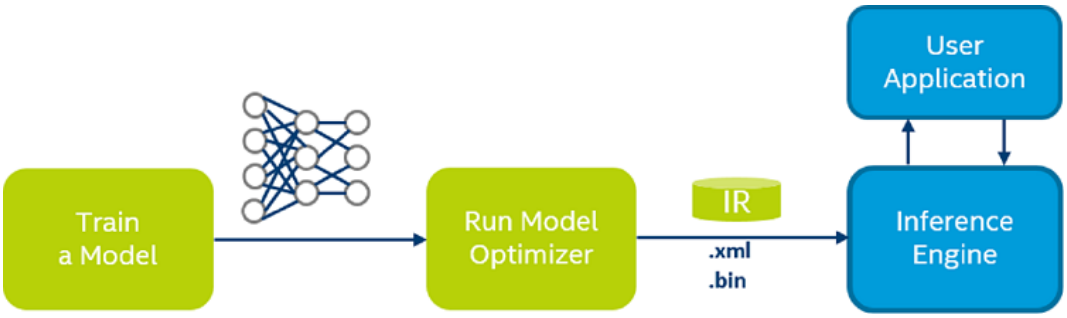

Figura 8: Flujo de trabajo para desplegar un modelo entrenado de *deep learning* en OpenVINO [9]

El optimizador de modelos crea una representación intermedia de la red ya entrenada que puede ser leída, cargada e inferida con el motor de inferencia. Esta representación intermedia consta de dos archivos que describen el modelo de la red:

- Archivo .xml: Describe la topología de la red
- Archivo .bin: Contiene los pesos y los datos binarios de la red

Después de usar el optimizador de modelos para crear estos archivos de representación intermedia, el motor de inferencia se encargará de inferir los resultados dado unos datos de entrada.

Este motor de inferencia es una librería en C++ que se encarga de inferir unos datos de entradas (imágenes) y obtener unos resultados. Posee una API para leer los archivos de representación intermedia creados, establecer los formatos de entrada y salida y ejecutar el modelo en los dispositivos finales [10].

Posee una arquitectura basada en plugins. De esta forma, para cada uno de los dispositivos soportados del hardware de Intel, el motor de inferencia despliega un plugin (una librería DLL) que contiene una implementación completa para la inferencia de ese dispositivo en particular. Así, entre otros, el motor de inferencia tiene plugins como CPU, FPGA o MYRIAD (este último para el Movidius Neural Compute Stick o el Neural Compute Stick 2) [10].

2.3 Dispositivos embebidos compatibles con aceleradores de IA

Como ya se ha visto en los apartados anteriores, hay algunas tecnologías que pueden conectarse mediante USB a otro dispositivo para acelerar la inferencia de este último. Es el caso del acelerador USB de Google Coral o los Neural Compute Stick de Intel. Estos dispositivos pueden conectarse a un sistema embebido con el que tengan compatibilidad, como es la Raspberry Pi, ayudándola con el proceso de inferencia de las redes neuronales.

2.3.1 Raspberry Pi Foundation

Dentro de este apartado se incluye la placa de desarrollo Raspberry Pi en todos sus modelos. Se tratan de las placas más conocidas a nivel mundial y no tienen como uso específico acelerar aplicaciones de *deep learning*, sino que se tratan más bien de placas de carácter general para aprender ciencias de la computación.

Se han ido lanzando varios modelos y revisiones a lo largo de los años de esta placa. Precisamente con uno de estos modelos se va a realizar parte del trabajo del que se está escribiendo. El más moderno de éstos es actualmente la Raspberry Pi 4, aunque para el proyecto se va a usar la Raspberry Pi 3 B+.

Figura 9: Raspberry Pi 3 B+

Su sistema operativo es una versión adaptada de Debian, denominada Raspbian, aunque también permite usar otros sistemas operativos. A pesar de no ser una placa con carácter específico para aplicaciones con redes neuronales, puede ejecutar aplicaciones de *deep learning* no muy exigentes gracias a la librería OpenCV.

Además, como se ha mencionado anteriormente, es compatible con dispositivos aceleradores de inferencia como

el Movidius Neural Compute Stick o el acelerador USB de Google Coral, por lo que, si se conectase alguno de ellos a la Raspberry, ésta última podría aprovechar los beneficios de cualquiera de estos dos aceleradores USB y conseguiría acelerar las aplicaciones de *deep learning*.

Se trata de una placa con salida HDMI, 4 puertos USB 2.0 para poder conectar teclado y ratón entre otros, conectividad con módulos de cámaras para Raspberry Pi, un puerto Gigabit Ethernet, ranura microSD y soporte para conectar una pantalla táctil [11].

Respecto al modelo más novedoso, la Raspberry Pi 4, se puede decir que es más potente que el modelo anterior. Destaca la adición de mejores especificaciones técnicas y de puertos USB 3.0 y micro HDMI. Dentro del mismo modelo se pueden elegir distintas variantes para la elección de la memoria RAM (2GB, 4GB o 8GB).

En base a todos estos dispositivos y avances en el tema de estudio, se puede crear una pequeña tabla comparativa de las especificaciones técnicas entre las placas de desarrollo mencionadas:

Tabla 1: Comparación de especificaciones técnicas de las placas de desarrollo

Placa	GPU	CPU	Memoria	Precio
Jetson Nano	128 core Maxwell	Quad-core ARM A57 @ 1.43 GHz	4 GB 64-bit LPDDR4 25.6 GB/s	140 €
Coral Dev Board	Integrated GC7000 Lite Graphics	NXP i.MX 8M SoC (quad Cortex-A53, Cortex-M4F)	1 GB LPDDR4	127 €
Raspberry Pi 3 B+	Broadcom Videocore-IV	Broadcom BCM2837B0, Cortex-A53 (ARMv8) 64-bit SoC	1GB LPDDR2 SDRAM	30 €
Raspberry Pi 4 B	Broadcom VideoCore VI	Broadcom BCM2711, Quad core Cortex-A72 (ARM v8) 64-bit SoC @ 1.5GHz	2GB, 4GB o 8GB LPDDR4 (depende del modelo)	31 € (2 GB) 49 € (4 GB) 66 € (8 GB)

Cabe destacar que las placas de NVIDIA y Google son placas de desarrollo específicas cuyo objetivo es potenciar aplicaciones de inteligencia artificial tal y como se ha visto en todo este apartado. La Raspberry Pi, en cambio, es una placa de bajo coste sin este objetivo específico.

Por otra parte, también se puede realizar otra tabla comparativa de los distintos aceleradores USB:

Tabla 2: Comparación de las especificaciones técnicas de los aceleradores USB

Acelerador USB	Procesador	Conector	Dimensiones	Precio
Google Coral	Edge TPU	USB 3.0	65mm X 30 mm	53 €
Intel Movidius Neural Compute Stick	Myriad 2	USB 3.0	72.5mm X 27mm X 14 mm	59 €
Intel Neural Compute Stick 2	Myriad X	USB 3.0	72.5mm X 27mm X 14 mm	61 €

3 REDES NEURONALES

El objetivo de este capítulo es introducir lo que se conoce como *deep learning* o aprendizaje profundo, además de conocer las capas principales que se pueden encontrar en una red neuronal. También se hablará de las principales operaciones para procesar la información en un modelo de *deep learning* o de algunos modelos de redes neuronales conocidos.

3.1 Introducción

Antes de comenzar es preciso diferenciar términos y áreas de aplicación para entender cuando se está hablando de inteligencia artificial, *machine learning* (aprendizaje automático) o *deep learning* (aprendizaje profundo).

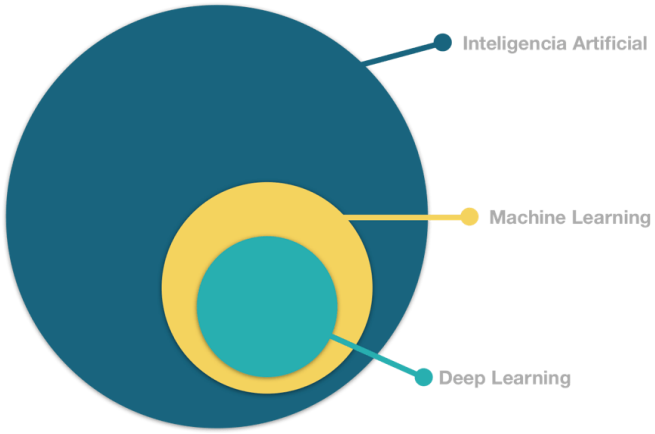

Figura 10: Relación entre IA, *machine learning* y *deep learning* [13]

Para empezar, se puede definir a la inteligencia artificial como aquella inteligencia que muestran las máquinas y que les sirve para automatizar tareas intelectuales que normalmente serían realizadas por un humano. Se trata, por tanto, de un campo muy grande que abarca muchas áreas de conocimiento, de forma que cualquier aplicación relacionada con inteligencia computacional podría conocerse como inteligencia artificial.

Por otra parte, lo que se conoce como *machine learning* o aprendizaje automático es un área específica dentro de la inteligencia artificial que proporciona a los ordenadores la capacidad de aprender por ellos mismos sin necesidad de que se les indique las reglas que debe seguir para lograr su tarea [12]. De esta forma, se desarrolla un algoritmo de predicción cuyo rendimiento va a ir aumentando a medida que es expuesto a más datos con el tiempo.

Dentro de este campo hay algunos términos que merecen conocerse y que pueden ser útiles más adelante. Así, por ejemplo, se le llama label o etiqueta a la salida, es decir, a lo que se intenta predecir con un modelo. En cambio, a las variables de entradas se le llaman features o características, es decir, a los atributos descriptivos. De esta forma, un modelo va a definir siempre la relación entre las características (entradas) y las etiquetas (salidas). Es decir, el modelo predecirá una etiqueta final en base a las características que se tenían en la entrada [12]. Posee, además, dos fases claramente diferenciadas:

- Fase de entrenamiento: En esta fase es cuando se crea y se construye el modelo. Se le pasan los ejemplos de entrada etiquetados con sus respectivas etiquetas para que el propio modelo aprenda de forma iterativa y gracias a una gran cantidad de datos, la relación existente entre las características de entrada y las etiquetas de salida.
- Fase de inferencia: En esta fase es cuando el modelo, tras ya ser entrenado, realiza las predicciones sobre ejemplos no etiquetados, de forma que predice la salida o etiqueta de estos ejemplos.

Cabe resaltar también que en la fase de entrenamiento es donde el modelo aprende los valores ideales de los parámetros del modelo, esto es, los pesos y el sesgo. Estos valores ideales minimizan la *loss* o error del modelo hasta hacerla lo más mínima posible, puesto que representa la penalización de una mala predicción. A su vez,

estos parámetros ideales también aumentan la *accuracy* o eficacia del modelo hasta hacerla lo más grande posible, ya que representa el acierto de una buena predicción [12].

Por último, se tiene lo que comúnmente se llama *deep learning* o aprendizaje profundo, un área específica dentro del aprendizaje automático que utiliza como algoritmo de computación redes neuronales artificiales con varias capas ocultas para procesar enormes cantidades de datos. Este algoritmo crea un sistema de interconexión en varias capas formadas por neuronas artificiales que colaboran entre sí para procesar los datos de entrada y al final generar una salida.

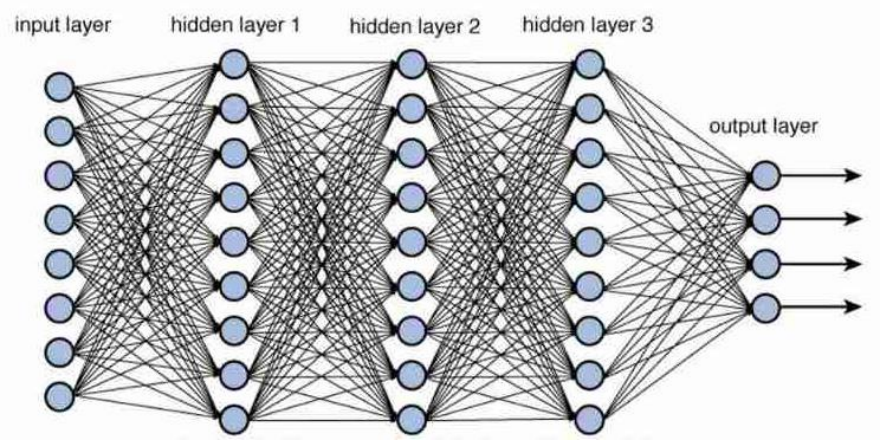

Figura 11: Red neuronal profunda [14]

En la figura 11 se tiene la típica estructura de una red neuronal profunda. Concretamente la red neuronal de la imagen tiene 5 capas: una de entrada (*input layer*), que es la que recibe los datos de entrada, una capa de salida (*output layer*) y tres capas ocultas (*hidden layers*). Los círculos representan cada una de las neuronas y como se puede apreciar, están conectadas unas con otras mediante enlaces. Esto es así para este caso en particular, pero el número de capas ocultas y de neuronas varía en cada modelo.

Cada una de las neuronas tendrá sus propios parámetros (pesos y sesgo), y realizarán una transformación simple de los datos que reciben de la capa anterior para pasarlos a su vez a la capa posterior. La unión de todas ellas sirve para detectar patrones complejos.

3.2 Desarrollo de una red neuronal

Una vez introducido los principales conceptos generales del mundo del *deep learning*, es necesario aprender cómo suele ser el proceso de desarrollo de una red neuronal, es decir, que conjuntos de datos se suelen usar, como se propaga la información en una red, que hiperparámetros establece el programador para construir un modelo, etc.

3.2.1 Entrenamiento, validación y prueba

Para construir todas estas redes neuronales y para la configuración y evaluación de un modelo, se suele utilizar un conjunto de datos. Este único conjunto de datos se suele dividir en otras tres porciones u otros tres subconjuntos de datos o *datasets* en el proceso de desarrollo de una red neuronal. Estos son los datos de entrenamiento (*training*), los datos de validación (*validation*) y los datos de prueba (*test*) [15]:

- Los datos de entrenamiento son aquellos datos que se usan para ajustar el modelo y obtener los parámetros de la red (pesos y sesgos). El modelo "ve" y "aprende" a partir de estos datos.
- Los datos de validación son aquellos que proporcionan una evaluación imparcial del modelo entrenado con los datos de entrenamiento a la vez que se afinan los hiperparámetros del modelo (ya se verán algunos más adelante).
- Los datos de prueba son aquellos que son usados para proporcionar una evaluación imparcial del modelo final entrenado con los datos de entrenamiento.

Se puede decir de forma muy resumida que los datos de entreno hacen que el modelo examine los datos, los de validación hacen que el modelo aprenda de sus errores, y los de prueba hacen que el modelo haga una conclusión de lo bien que rinde el modelo

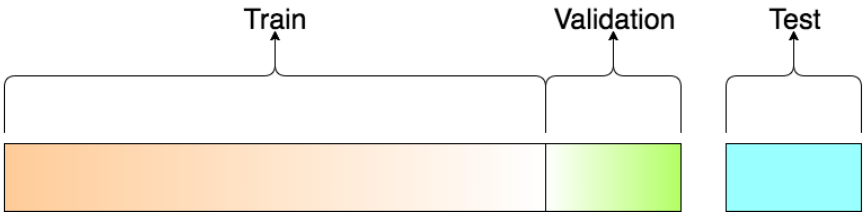

Figura 12: Repartición de los datos de entreno, validación y prueba [15]

Hay dos aspectos principales para saber cómo dividir el conjunto de datos inicial en estos tres *datasets* que se acaban de describir. El primero de ellos es el número de muestras en el conjunto de datos; y el segundo es el modelo actual que se está entrenando.

Algunos modelos necesitan grandes cantidades de datos para poder ser entrenados, así que en este caso habría que optimizar la repartición para un mayor porcentaje de datos en el conjunto de entrenamiento. Modelos con muy pocos hiperparámetros serán fáciles de validar y poner a punto, por lo que probablemente pueden tener un menor conjunto de datos de validación. En cambio, si el modelo tiene muchos hiperparámetros sucede lo contrario, es conveniente utilizar una mayor cantidad de datos para el conjunto de datos de validación. Del mismo modo, si se tiene un modelo con ningún hiperparámetro o alguno que no puede ser fácilmente configurable, no se necesitaría siquiera un conjunto de datos de validación. Por eso mismo, la proporción de división de los datos de entreno, validación y prueba es muy específico al caso de uso.

Independientemente de esto, los datos de entrenamiento siempre se llevan más de la mitad de las muestras de todos los datos, puesto que es el conjunto más importante para el proceso de aprendizaje del modelo. En la mayoría de las reparticiones el conjunto de entrenamiento tiene el 50% o más de todo el conjunto total de datos.

3.2.2 Proceso de aprendizaje de una red neuronal

Como se ha visto en esta sección, las neuronas de una red neuronal están conectadas entre ellas. Cada conexión entre dos neuronas tiene asociado un peso que determina la importancia de esa relación en la neurona al multiplicarse por el valor de entrada.

Cada neurona tiene a su vez una función de activación que define la salida de la neurona. Esta función de activación, sea del tipo que sea, se usa para introducir la no linealidad en las capacidades de modelado de la red. Entrenar la red neuronal, es decir, aprender los valores de los parámetros (pesos y sesgo) es la parte más pura del área de *deep learning*, y se puede ver este proceso de aprendizaje en una red neuronal como un proceso iterativo de ida y vuelta por las capas de neuronas.

El ir propagando hacia delante se llama *forward propagation*, mientras que el ir propagando hacia detrás se llama *back propagation*. El primero de ellos se da cuando se expone a la red los datos de entreno y éstos cruzan toda la red neuronal hacia delante para que sus predicciones o etiquetas sean calculadas. De esta forma, se traspasan los datos de entrada a través de la red neuronal de forma que todas las neuronas de la red aplican su transformación a la información que reciben de las neuronas de la anterior capa y la envíen a las neuronas de la capa siguiente. Cuando los datos crucen todas las capas de la red y todas las neuronas realicen sus cálculos, se llegará a la capa final con un resultado de predicción de la etiqueta para aquellos ejemplos de entrada.

Después de usa una función de *loss* para estimar la *loss* o error y así comparar y medir cuánto fue de bueno el resultado de la predicción en comparación con el resultado correcto. Lo ideal es que este valor fuera cero, es decir, que no hubiera diferencia entre la etiqueta estimada y la esperada en ninguno de los casos. Por eso, a medida que se va entrenando el modelo se van ajustando los pesos de las conexiones entre las neuronas de forma automática hasta que se obtienen buenas predicciones.

Una vez la *loss* es calculada, esta información se propaga hacia atrás. Esto es lo que se conoce ya como *back propagation*. Partiendo de la capa de salida, esta información acerca de la *loss* se propaga hacia las neuronas de la capa oculta anterior a la capa de salida. Sin embargo, las neuronas de esta capa oculta solo reciben una pequeña fracción de toda la señal de *loss* en base a la contribución relativa que haya aportado cada neurona a la salida. Este proceso de propagación de la información de *loss* se repite capa por capa hasta que todas las neuronas de

la red reciban una fracción de la señal de *loss* en base a su contribución relativa respecto a la *loss* total [12].

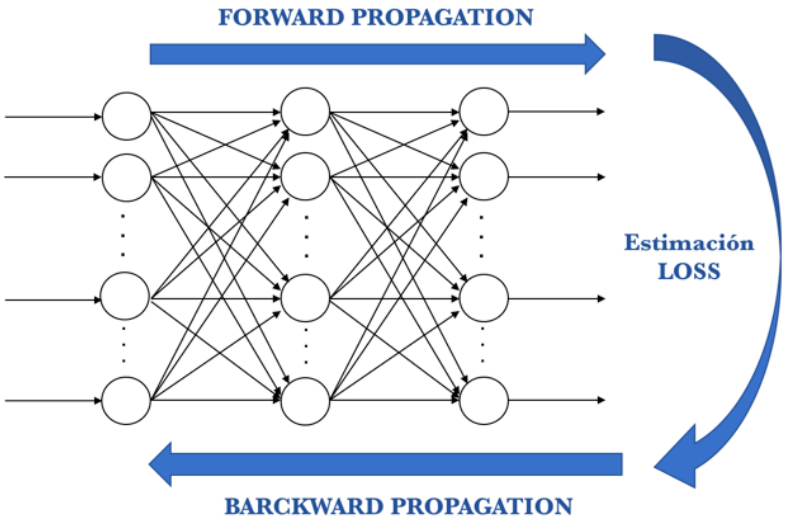

Figura 13: *Forward propagation* y *back propagation* [12]

Con esta información propagada ya hacia detrás, se pueden ajustar los pesos de las interconexiones de las neuronas. De esta forma, se busca que la *loss* se aproxime a cero la próxima vez que se use la red para una predicción. Para ello se usa un optimizador que va cambiando los valores de los pesos en pequeños incrementos de forma que al final se alcance el mínimo global. Esto lo hace en lotes de datos (*batches*) en todas las iteraciones (*epochs*) del conjunto de los datos que se pasan a la red en cada repetición.

Para concluir, y resumiendo todo lo descrito, los pasos para construir el algoritmo de aprendizaje son:

- Comenzar los parámetros de la red (pesos y sesgo) con unos valores (normalmente aleatorios).
- Pasar un conjunto de ejemplos de datos por la red para obtener la predicción.
- Comparar las predicciones obtenidas con las esperadas, y a partir de aquí calcular la *loss*.
- Realizar *back propagation* para propagar parte de la señal de *loss* a cada una de las neuronas y parámetros que forman la red.
- Utilizar la información propagada hacia atrás para actualizar con un optimizador los parámetros de la red, reduciendo la *loss* total y mejorando el modelo.
- Repetir los pasos descritos hasta considerar que ya se tiene un gran modelo.

En los siguientes apartados se introduce un poco mejor cada uno de los elementos que se han descrito en esta subsección.

3.2.2.1 Funciones de activación

Las funciones de activación son usadas para propagar hacia delante la salida de una neurona y sirven para introducir, en caso de que se quiera, no linealidad en el modelo de la red neuronal [12]. Las más importantes son:

- Función linear: La señal no cambia
- Función *sigmoid*: Reduce valores extremos, ya que convierte señales de rango enormes a rangos entre 0 y 1.
- Función *tanh*: Maneja fácilmente números negativos y normaliza los valores entre -1 y 1.
- Función *ReLU* (rectificador): Si la entrada está por encima de un umbral determinado (normalmente cero), activa el nodo. En este caso, la salida es una relación lineal con la variable de entrada.

3.2.2.2 Función de *loss* y optimizadores

Como se ha comentado anteriormente, el *back propagation* es el método en el que la señal de *loss* se propaga hacia atrás a cada una de las neuronas y en el que el optimizador actualiza los parámetros reduciendo la *loss*. Existen dos elementos fundamentales en el *back propagation*: la función de *loss* y los optimizadores.

Una función de *loss* es la función usada para estimar la *loss* de un modelo para que los parámetros de la red posteriormente puedan actualizarse y reducir la *loss* en la siguiente iteración [16]. La elección de la mejor función de *loss* depende del tipo concreto de problema al que al que se enfrenta la red:

- Un problema de regresión: Se suele usar el error cuadrático medio (*mean squared error*). Se calcula como la media de las diferencias al cuadrado entre los valores predichos y los reales.
- Un problema de clasificación binaria: Se suele usar la entropía cruzada binaria (*binary cross-entropy*), donde los valores objetivos están en el conjunto de 0 y 1. Calcula una puntuación que resume la diferencia media entre las distribuciones de probabilidad reales y pronosticadas para predecir la clase 1. La puntuación se minimiza y un valor perfecto de entropía cruzada es 0.
- Un problema de clasificación de múltiples clases: Aquí se suele utilizar la entropía cruzada multiclase (*multi-class cross-entropy*). En este caso, está diseñado para su uso con varias clases donde los valores objetivos están en el conjunto (0,1,2…n), donde cada clase tiene asignado un valor entero único. Calcula una puntuación que resume la diferencia media entre las distribuciones de probabilidad reales y pronosticadas para todas las clases del problema. La puntuación se minimiza y un valor perfecto de entropía cruzada es 0.

El optimizador, por su parte, es el que se encarga propiamente de cambiar los parámetros de la red neuronal con el objetivo de minimizar la función de *loss*. El más importante es el descenso de gradiente. Es usado en problemas de regresión y de clasificación y calcula de qué manera los parámetros de la red deben ser alterados para que la función de *loss* alcance un mínimo [17]. Hay varios tipos de optimizadores más basados en el descenso de gradiente que son también muy usados:

- Adagrad: Con este optimizador algunos de los pesos de la red tendrán diferentes *learning rates* que otros, aunque hace el entrenamiento de la red muy lento.
- RMSprop: Versión especial de Adagrad que busca corregir los problemas de ésta última. Mantiene un promedio móvil del cuadrado de los gradientes y divide el gradiente entre la raíz de este promedio.
- Adam: La clave de este optimizador es que reduce la velocidad para una búsqueda segura del mínimo de la función de *loss*, de forma que no se salte por encima de él al ir demasiado rápido.

3.2.2.3 Parámetros e hiperparámetros

Los parámetros de un modelo son variables de configuración internas del propio modelo que son estimadas con datos. Por el contrario, los hiperparámetros son variables de configuración externas al modelo que no pueden ser estimadas con datos, sino que son configurados por los programadores antes de entrenar la red para configurar y poner a punto el algoritmo de aprendizaje de un modelo, de forma que éste sea el mejor posible al final del proceso de entrenamiento [12]. Los principales hiperparámetros son los siguientes:

- Número de *epochs*: Representa con un número las veces que los datos pasan por la red neuronal durante el proceso de entrenamiento.
- *Batch size*: Los datos de entrenamiento se pueden particionar en lotes. Indica, por tanto, el número de muestras que hay en cada lote en cada iteración del entrenamiento.
- *Learning rate*: Determina el tamaño del paso en cada iteración del entrenamiento para alcanzar un mínimo de una función de *loss*. Si el valor es muy grande puede ser genial para el aprendizaje, pero quizá esos grandes pasos hagan que se salte el mínimo buscado y se produzca un rebote. En cambio, si el valor es muy pequeño el aprendizaje es muy lento, por eso es importante elegir el valor adecuado.
- *Learning rate decay*: Sirve para reducir el *learning rate* a la vez que las *epochs* van aumentando, de forma que se consiguen grandes pasos al principio del entreno y van disminuyendo los pasos a medida que se acerca el final del entrenamiento.

3.3 Redes neuronales convolucionales

Las redes neuronales convolucionales (CNN) son unas de las redes más usadas dentro del mundo del *deep learning*. Son un tipo de red neuronal diseñadas específicamente para visión artificial. Su nombre es debido a que dentro de su estructura usan capas de convolución para recibir y procesar datos de imágenes [12].

Todas las redes que se van a usar en el presente trabajo son redes neuronales convolucionales, y por lo tanto es necesario conocer un poco de su arquitectura y de las capas que más frecuentemente son usadas.

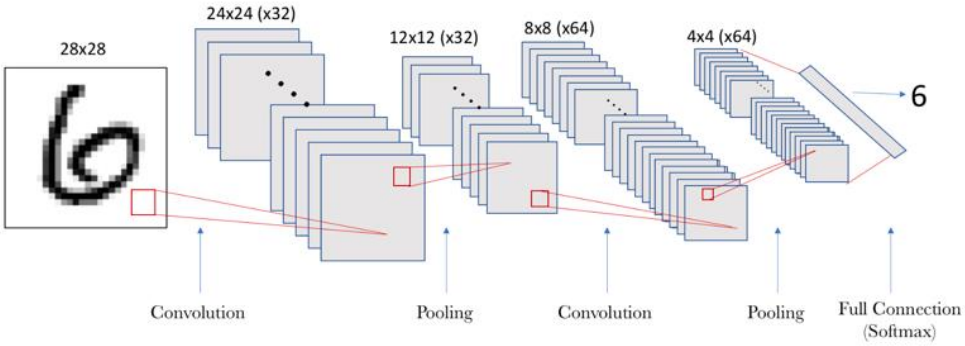

Figura 14: Estructura de una red neuronal convolucional [12]

Estas redes toman una imagen como entrada a la que se le normalizan los valores de los píxeles. Los colores de los píxeles tienen valores que van de 0 a 255, por lo que antes de alimentar la red se hace una transformación de cada píxel, dividiendo cada valor entre 255, de forma que el valor de todos los píxeles de la imagen quede normalizado entre 0 y 1.

La red neuronal convolucional irá reconociendo cada vez formas más complejas a medida que se llegue a capas más profundas dentro de su estructura, hasta que al final de ella sea capaz de clasificar y predecir la etiqueta de la imagen de entrada.

La estructura de una CNN suele tener dos partes claramente diferenciadas: en la primera de ellas se produce la extracción de características de la imagen; y en la segunda de ellas se produce la clasificación de la imagen. Se suelen usar principalmente tres tipos diferentes de capas en una CNN: capas de convolución, capas de *pooling* y capas densamente conectadas. Las dos primeras de ellas se encargan de la primera parte (extracción de características). La última, en cambio, se suele encargar de la segunda parte (clasificación). Ahora se pasa a ver un poco más en profundidad cada una de ellas.

3.3.1 Capa convolucional

Las capas convolucionales son el núcleo de las redes neuronales convolucionales. Estas capas aprenden patrones locales de la imagen de entrada en pequeñas ventanas de dos dimensiones. Su objetivo principal es, por tanto, detectar características y rasgos visuales en las imágenes tales como aristas, líneas, etc. De esta forma, una vez una característica sea detectada en un punto en concreto de la imagen, se puede reconocer después automáticamente en cualquier parte de la misma [12].

Otra de las características de estas capas es que pueden aprender jerarquías espaciales de patrones conservando relaciones especiales. Es decir, una primera capa convolucional puede aprender elementos básicos como líneas, y posteriormente, una segunda capa convolucional puede aprender patrones compuestos de los elementos más básicos aprendidos en la primera capa, y así de forma sucesiva se pueden ir aprendiendo patrones cada vez más complejos en capas convolucionales posteriores.

De forma general las capas convolucionales operan sobre tensores 3D llamados feature maps o mapas de características. Estos mapas tienen tres ejes, dos para altura y anchura, y un tercero de canal o profundidad. Para una imagen de color *RGB* la dimensión del eje de canal es 3, puesto que la imagen tiene tres canales: rojo, verde y azul. En cambio, para una imagen en blanco y negro, la dimensión de este canal es 1 (solo el nivel de gris de la imagen) [12].

Por ejemplo, en la figura 12 se puede ver como la imagen de entrada es un espacio de dos dimensiones (28 de altura, 28 de anchura y tan solo 1 de profundidad al ser una imagen no en color). Se puede pensar, por tanto, en un espacio de neuronas de 2D de 28x28 como entrada de la red neuronal. A partir de aquí, una primera capa de neuronas ocultas conectada con pequeñas zonas del espacio de esta capa de entrada se encargará de realizar las operaciones convolucionales que se han descrito anteriormente. Siguiendo con el ejemplo, cada neurona de la primera capa oculta será conectada a una pequeña región de 5x5 neuronas de la capa de entrada de 28x28. Esta ventana de 5x5, se irá deslizando hacia la derecha y de arriba abajo por toda la capa de entrada de forma sucesiva desde la esquina superior izquierda hasta la esquina inferior derecha, de forma que por cada posición de la ventana de 5x5 hay una neurona en la primera capa oculta que procesa toda esa información de la ventana en esa posición específica.

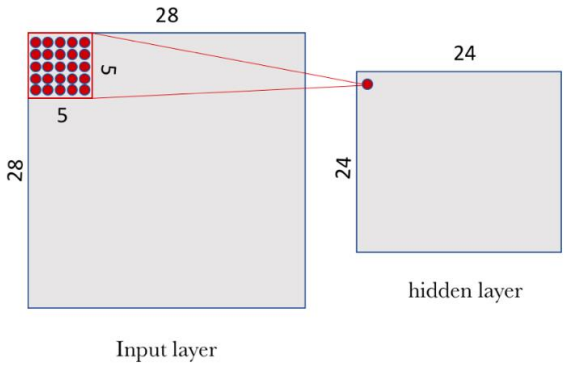

Figura 15: Ventana de conexión entre la capa oculta y la capa de entrada [12]

Para el caso concreto del ejemplo, si se aplica una ventana de 5x5 a la capa de entrada de 28x28 se tendrá en la primera capa oculta un espacio de 24x24. Esto es debido a que en un espacio de 28x28, la ventana de 5x5 solo se puede mover 23 neuronas hacia la derecha y 23 neuronas hacia abajo antes de llegar a los límites de la imagen.

Para conectar cada neurona de la capa oculta con cada una de las 25 (5x5) neuronas que le corresponden de la capa de entrada se usa un valor de sesgo y una matriz de pesos llamada filtro, con las mismas dimensiones que la ventana de la capa de entrada, es decir, 5x5 en este caso.

De esta forma, el valor de cada punto de la capa oculta se corresponde con el producto escalar entre la matriz de pesos o filtro y el puñado de 5x5 neuronas de la capa de entrada que corresponda para esa posición en particular. Cabe destacar que mientras la ventana de 5x5 de la capa de entrada respecto a la capa oculta va variando según se va desplazando, el valor de todo el filtro (matriz de pesos y sesgo) es inalterable para todas las neuronas de la primera capa oculta.

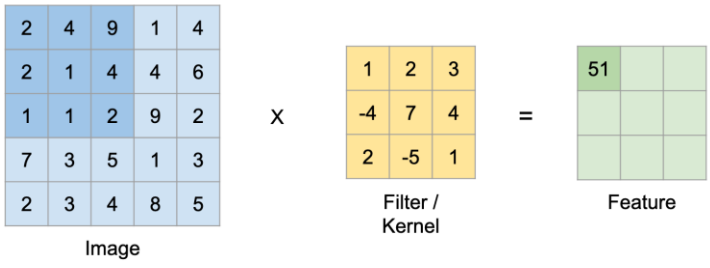

Figura 16: Producto escalar entre ventana de imagen y el filtro [18]

En la figura 14 se puede ver un ejemplo de producto escalar entre un filtro y una ventana de una imagen de otra capa. No corresponde con el ejemplo que se está explicando en este apartado puesto que las dimensiones de las ventanas son diferentes, pero sirve para captar la idea.

Volviendo al ejemplo de la figura 12, el filtro servirá para buscar características y patrones locales en pequeñas zonas de la imagen de la capa inicial. Pero un filtro definido por una matriz y un sesgo solo es capaz de detectar una característica en concreto de la imagen. Es por eso que normalmente se suelen usar varios filtros a la vez en una capa convolucional, de forma que cada uno de ellos corresponda con una característica que se quiera detectar [12].

De forma gráfica, esta capa convolucional con 32 filtros se puede simbolizar de la siguiente manera:

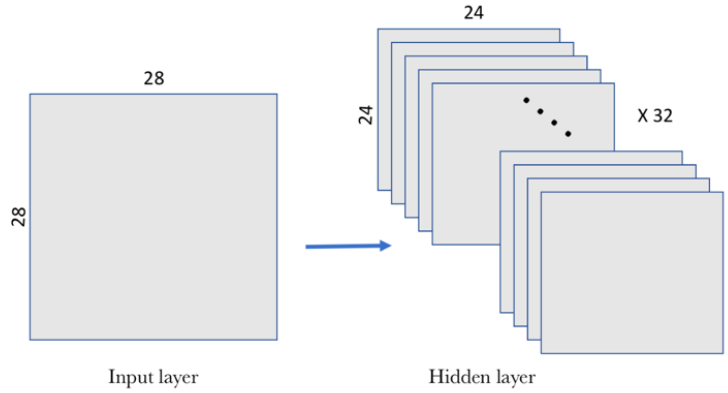

Figura 17: Capa convolucional con 32 filtros [12]

Así, cada uno de los 32 filtros tendrá su propia matriz de pesos 5x5 y su propio sesgo. En este ejemplo, la capa convolucional recibe un tensor de entrada 28x28x1 y al final de la misma genera una salida de un tensor 3D con dimensiones 24x24x32 que contiene las 32 salidas de dimensiones 24x24 como resultado de computar los 32 filtros sobre el tensor de entrada.

Por último, al final de cada capa de convolución se aplica una función de activación por si se quiere añadir no linealidad a la red.

3.3.2 Capa de *pooling*

Estas capas de *pooling* suelen acompañar a las capas de convolución inmediatamente después. Se encargan de hacer una simplificación de la información recogida por la capa convolucional, de forma que crean una versión condensada de toda la información contenida en estas últimas [12].

En el ejemplo de la figura 12 se elige una ventana de dimensiones 2x2 de la capa convolucional para sintetizar esa información en un punto de la capa de *pooling*. Quedaría algo así:

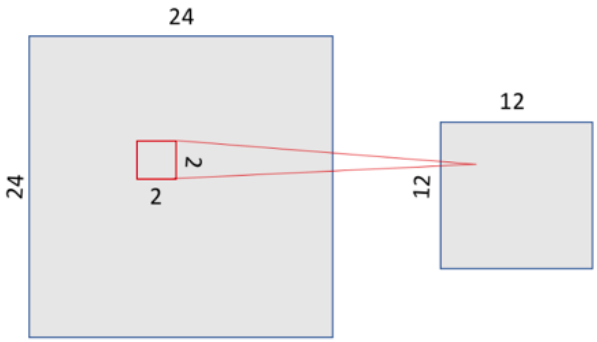

Figura 18: Ventana de 2x2 para sintetizarla en la capa de *pooling* [12]

Hay varias formas de condensar la información en esta capa, aunque la más habitual y conocida es la técnica de *max-pooling*. Esta técnica da como resultado el valor máximo de los que había en la ventana de 2x2 (en el caso del ejemplo). Al haber en este caso una ventana de 2x2, las dimensiones de salida de la capa de pooling son de 12x12 tal y como se puede ver en la figura 16.

También se puede usar otra técnica llamada *average-pooling* en lugar de *max-pooling*, donde en vez de tener como resultado el valor máximo de la ventana se tiene el promedio de los valores de la ventana, pero en general la técnica de *max-pooling* suele funcionar mejor [12].

En el caso del ejemplo, al ser la ventana de 2x2, ésta se va a desplazar desde la esquina superior izquierda hacia la derecha y hacia abajo de 2 en 2 píxeles, hasta llegar a la esquina inferior derecha del filtro. Cabe destacar que con la transformación de *pooling* se mantiene la relación espacial.

Como se mencionó anteriormente, la capa convolucional tiene varios filtros y, por tanto, al aplicar la técnica de *max-pooling* a cada uno de esos filtros, la capa de *pooling* tendrá tantos filtros de *pooling* como filtros convolucionales había en la capa anterior.

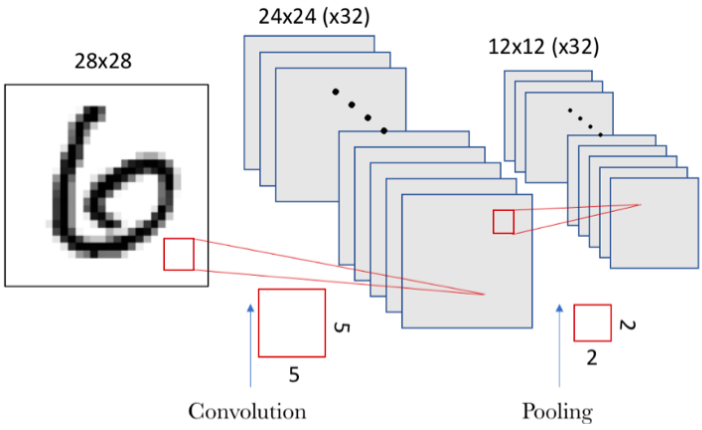

Figura 19: Número de filtros en la capa de *pooling* [12]

Como a la entrada a esta capa de *pooling* había un espacio de 24x24 neuronas en cada uno de los 32 filtros, el resultado al final de la misma tras aplicar *max-pooling* es el de un espacio con dimensiones 12x12 para cada uno de los 32 filtros tras aplicar ventanas de tamaño 2x2.

3.3.3 Capa densamente conectada

Esta capa se encargará de realizar la parte de clasificación de la imagen de entrada y proporcionarle una etiqueta u otra. Primero, hace falta transformar el tensor de 3D proveniente de la capa de *pooling* para que se convierta en un tensor de una sola dimensión y poder usar las redes densamente conectadas. Esto se hace de forma muy fácil con una capa flatten que aplana este tensor.

Este vector de valores una sola dimensión representa con cada valor la probabilidad de que ciertas características pertenezcan o no a cierta etiqueta. Por ejemplo, si la imagen es de un gato, características como un bigote deben tener altas probabilidades de pertenecer a la etiqueta 'gato'.

Las capas densamente conectadas de una red neuronal convolucional realizan su propia retropropagación para determinar los pesos más precisos. Todas las neuronas de una capa están conectadas a todas las demás de la capa densamente conectada anterior. Cada neurona recibe los pesos que priorizan la etiqueta más apropiada, los valores pasan por una función de activación y en la capa de salida cada neurona representa una etiqueta de clasificación.

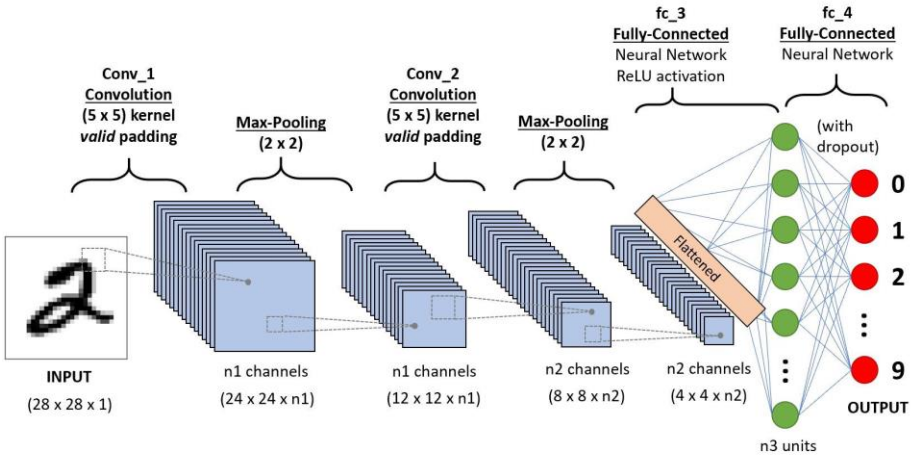

Figura 20: Red neuronal convolucional con dos capas densamente conectadas al final [19]

4 METODOLOGÍA

En este capítulo se describe cómo va a ser el proceso de despliegue de las redes neuronales en cada una de las plataformas de ejecución y cuáles son los elementos usados. También se verá como es el proceso de desarrollo de la red neuronal convolucional creada desde cero y su arquitectura. Por último, también se describirán las otras redes convolucionales utilizadas y ya entrenadas para clasificación de imágenes y se verán sus estructuras.

4.1 Plataformas de ejecución

Como se mencionó en el capítulo introductorio del proyecto, se van a ejecutar redes neuronales convolucionales en cuatro distintas plataformas de ejecución: CPU del portátil, GPU de Google Colab, Raspberry Pi sin Movidius Neural Compute Stick y Raspberry Pi con Movidius Neural Compute Stick.

Para poder ejecutar las redes neuronales en todas estas plataformas son necesarios dos elementos fundamentales. Por un lado, está OpenCV, una biblioteca de visión artificial desarrollada por Intel que cuenta con el módulo *dnn*, con el que se puede tratar y manejar redes neuronales profundas típicas de aplicaciones de *deep learning*. Con este módulo se pueden realizar cosas como cargar imágenes, leer modelos de redes de distintos *frameworks* como TensorFlow o Caffe, establecer determinados backends para realizar los cálculos de la red, calcular la salida de una red neuronal, etc.

Por el otro lado está OpenVINO, kit de desarrollo de software que ya se introdujo en el capítulo dos, compatible con el Movidius Neural Compute Stick, que optimiza modelos de *deep learning* entrenados en algún *framework* como TensorFlow y Caffe para desplegarlos usando un motor de inferencia en hardware de Intel.

OpenCV es una biblioteca autónoma que permite operar con redes neuronales profundas; sin embargo, OpenVINO tiene una gran variedad de herramientas aparte del optimizador de modelos y el motor de inferencia introducidos en el capítulo dos del proyecto, entre la que se encuentra la propia biblioteca OpenCV, tal y como se puede ver en la siguiente figura:

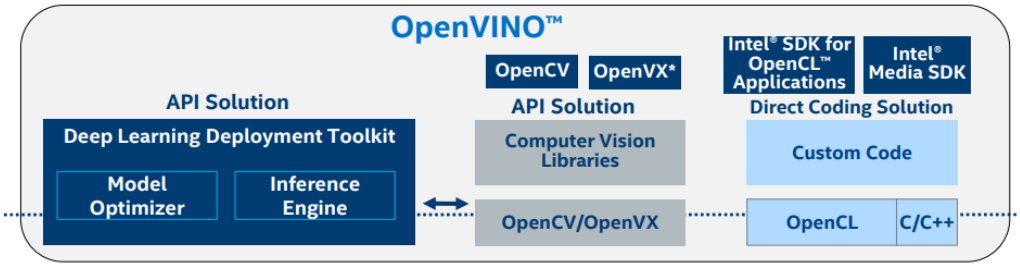

Figura 21: Componentes de OpenVINO [20]

Como se puede ver en la figura, la biblioteca OpenCV está integrado dentro de OpenVINO. Pues bien, dependiendo de la plataforma de ejecución en la que se quiera ejecutar la red neuronal se utilizará solamente la biblioteca OpenCV, utilizando su *backend*, o el kit de desarrollo completo de OpenVINO (incluyendo OpenCV) mediante el *backend* de su motor de inferencia.

De esta forma, se pueden tomar dos caminos distintos para desplegar la red neuronal:

- En uno de ellos, simplemente se cargan los archivos del modelo entrenado en cualquier *framework* como Caffe (.caffemodel y .prototxt) o TensorFlow (.pb y .pbtxt) y se realiza la inferencia utilizando el *backend* de OpenCV.
- En el otro, se pasa el modelo entrenado en cualquier *framework* por el optimizador de modelos de OpenVINO, que carga el modelo y lo optimiza para crear los archivos de representación intermedia. Con este optimizador de modelos se producen los dos archivos (.xml y .bin) que constituyen la

representación intermedia del modelo. Para ello elimina operaciones y capas innecesarias para la inferencia, además de combinar varios grupos de operaciones en una sola operación matemática, reduciendo el número de nodos respecto al modelo original. Todo esto disminuye el tiempo de inferencia en estos archivos de representación intermedia, que se utilizan con el *backend* de OpenVINO para realizarla, acelerando el proceso [21].

Todo esto se puede resumir con la siguiente imagen:

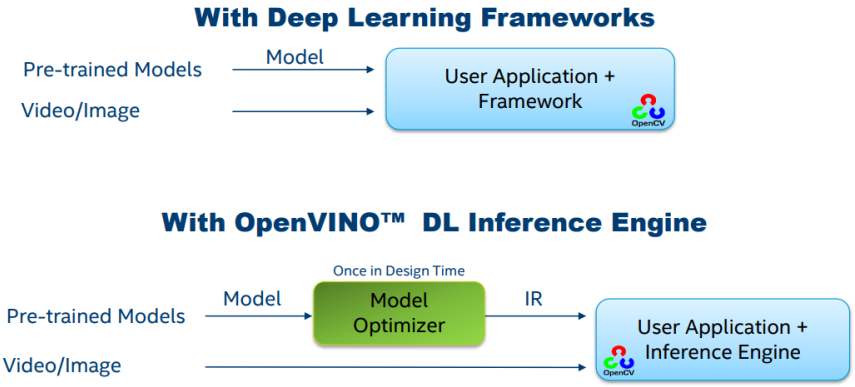

Figura 22: Despliegue de modelos con *backends* de OpenCV y OpenVINO [22]

El *backend* del motor de inferencia de OpenVINO puede ser usado con el Movidius Neural Compute Stick utilizado en el proyecto, al ser compatibles. Esto provoca la aceleración de la fase de inferencia de redes neuronales cuando el NCS se conecte a la Raspberry Pi en una de las plataformas de ejecución definidas para el proyecto. En las otras tres plataformas de ejecución, en cambio, se ejecutará el *backend* de OpenCV.

A continuación, se va a describir en las siguientes subsecciones las distintas arquitecturas seguidas para el despliegue de las redes neuronales en cada una de las plataformas definidas.

4.1.1 CPU

En este caso se va ejecutar las distintas redes neuronales para el proyecto con la CPU del portátil utilizado por el usuario. El modelo de la CPU es un Intel i7 de octava generación. En concreto se trata de un Intel i7-8550U, un procesador que cuenta con un total de 4 núcleos y 8 subprocesos a una frecuencia básica de 1.8 GHz [23].

Como sistema operativo se usa una máquina virtual de Ubuntu, una distribución de Linux basada en Debian que funciona en computadoras y que está orientada al usuario medio.

En este caso en particular se usa el *backend* de OpenCV descrito anteriormente.

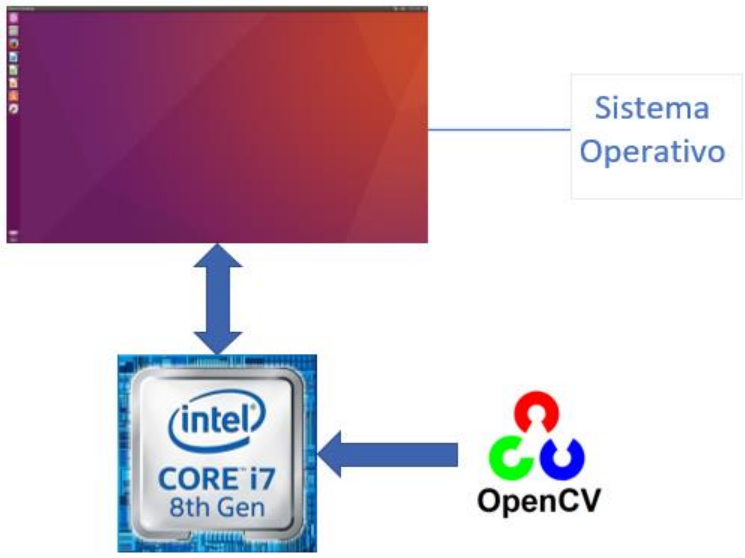

Figura 23: Esquema de ejecución con la CPU

4.1.2 GPU

Para este caso en particular se va a usar Google Colab, un entorno gratuito de Jupyter Notebook que no requiere configuración, que se ejecuta completamente en la nube y que permite crear *notebooks* (cuadernos): documentos que contienen tanto código como elementos de texto como figuras o ecuaciones.

Dentro de Colab hay que cambiar el entorno de ejecución para usar una GPU. Para ello hay que ir dentro de un cuaderno al menú de entorno de ejecución y cambiarlo:

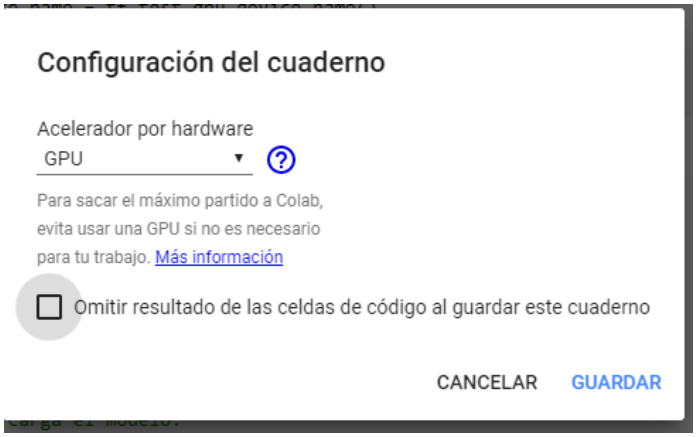

Figura 24: GPU establecida en Colab

Los tipos de GPU disponibles en Colab varían con el tiempo. Esto es necesario para que Colab pueda ofrecer un acceso gratuito a los recursos. Las GPU disponibles en Colab a menudo incluyen K80, T4, P4 y P100 de NVIDIA, pero no hay una forma de elegir el tipo de GPU a la que te puedes conectar en Colab [24].

En este caso también se usa el *backend* de OpenCV.

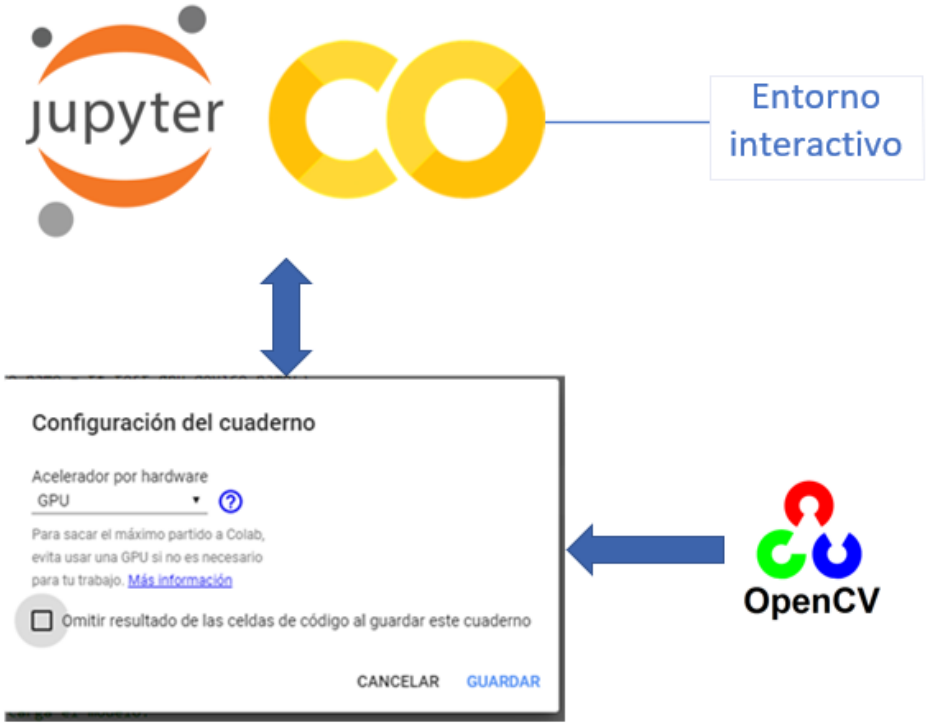

Figura 25: Esquema de ejecución con una GPU de Colab

4.1.3 Raspberry Pi

En este caso se va a utilizar un dispositivo embebido como es la Raspberry Pi. Como ya se ha podido comentar anteriormente en la memoria, se va a utilizar concretamente una Raspberry Pi 3 modelo B+, que cuenta con un procesador de cuatro núcleos ARM Cortex A53 de 64-bits a 1.4 GHz.

El sistema operativo utilizado por la placa de desarrollo es Raspberry Pi OS (anteriormente conocido como Raspbian). Se trata de un sistema operativo basado en Debian optimizado para Raspberry Pi. Desde el 2015 ha sido proporcionado por la Raspberry Pi Foundation como el sistema operativo primario para toda la familia de placas.

Se usa el *backend* de OpenCV para ejecutar las redes neuronales.

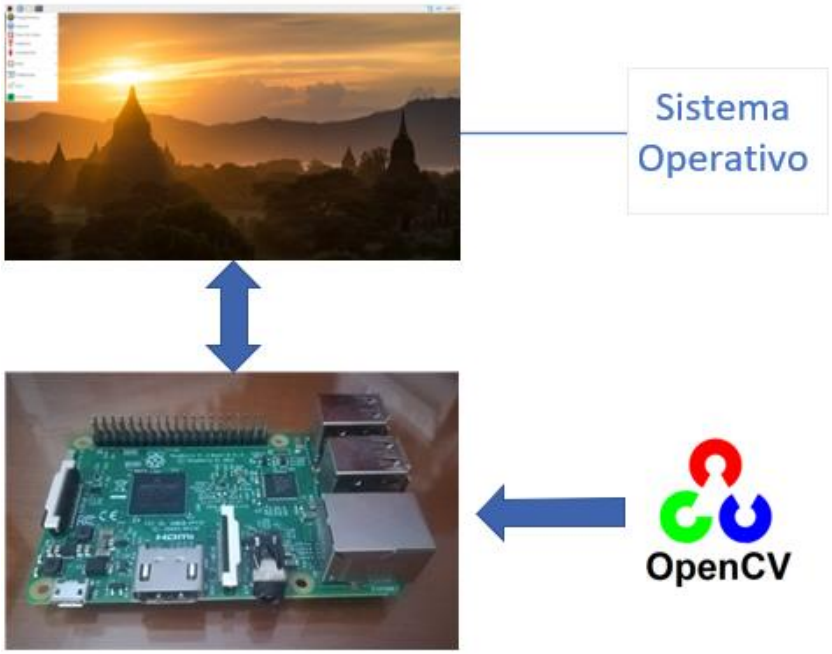

Figura 26: Esquema de ejecución en la Raspberry Pi 3 B+

4.1.4 Raspberry Pi con NCS

Este caso es el mismo que el anterior, salvo que esta vez se conecta el Movidius Neural Compute Stick a uno de los puertos USB 2.0 de la Raspberry Pi 3 B+. NCS utiliza en su interior la VPU Myriad 2, específica para acelerar aplicaciones de inteligencia artificial. Esta VPU posee aceleradores hardware de escaneo computacional, una memoria de fábrica de alto rendimiento sostenida a través de la localidad de los datos y sobre todo 12 núcleos SHAVE, unos procesadores vectoriales diseñados para tratar algoritmos de visión e imagen a alto rendimiento y a baja potencia [25].

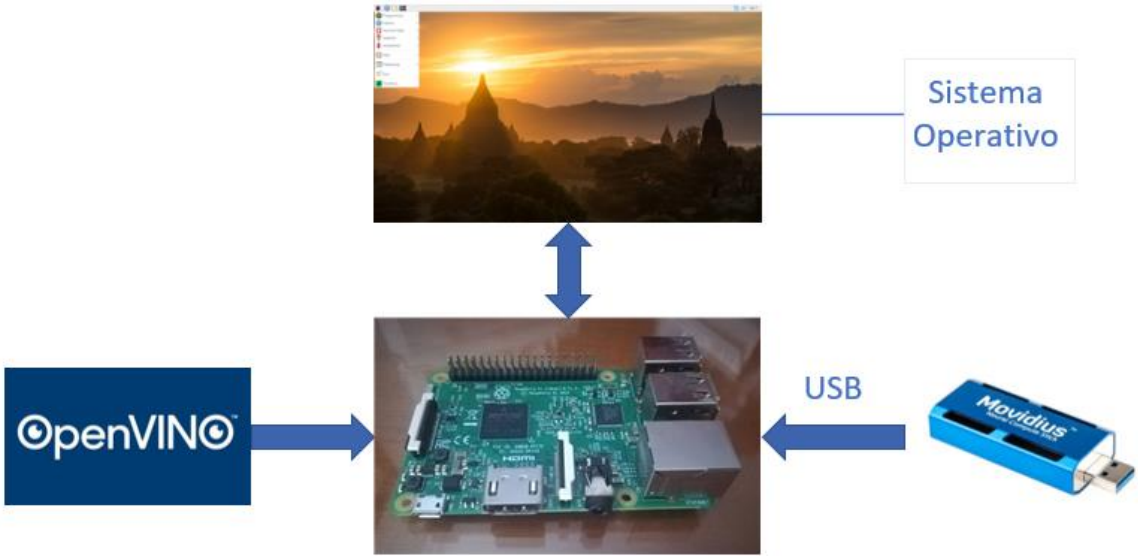

Figura 27: Esquema de ejecución en la Raspberry Pi 3 B+ con el NCS

Movidius Neural Compute Stick es compatible con OpenVINO, por lo que en este caso se puede usar el *backend*

del motor de inferencia para aprovechar la aceleración en fase de inferencia que provoca el Movidius Neural Compute Stick y el propio OpenVINO.

4.2 Redes neuronales convolucionales del proyecto

Ni OpenCV ni OpenVINO proveen herramientas para entrenar una red neuronal. Para ello hace falta usar algún *framework* o descargar un modelo ya entrenado.

En este proyecto se van a usar cuatros redes convolucionales que se ejecutarán en las distintas plataformas descritas en la sección anterior. Una de ellas será creada en Ubuntu desde cero con Keras, una API de alto nivel del *framework* TensorFlow que permite construir y entrenar modelos de *deep learning*. Las otras tres redes no serán entrenadas desde cero, sino que serán descargadas en Ubuntu con una utilidad de OpenVINO que descarga modelos ya entrenados.

4.2.1 Red neuronal creada

En esta sección se va a explicar el desarrollo de la red convolucional creada desde cero con Keras. Como se mencionó en el capítulo introductorio del proyecto, se trata de un clasificador de imágenes binario de perros y gatos.

4.2.1.1 Entrenamiento de la red

El primer paso será entrenar la red neuronal. Para ello se va a utiliza el *dataset* disponible en kaggle, cuyo enlace se encuentra en el capítulo introductorio del proyecto.

Se ha optado por entrenar la red neuronal con una GPU de Google Colab, que como se mencionó en la sección anterior, provee de *notebooks* o cuadernos, documentos que contienen tanto código como elementos de texto como figuras o ecuaciones.

Para entrenar la red es necesario subir el *dataset* de entreno a Google Drive para tenerlo almacenado permanentemente en la nube, de forma que al abrir después un *notebook* en Google Colab se pueda enlazar con la cuenta de Google Drive para tener así acceso al *dataset*.

Para montar Google Drive con Google Colab hay que ejecutar el siguiente código en cualquier *notebook* de Google Colab:

```
from google.colab import drive
drive.mount('/content/drive/')
```

Una vez introducido se proporciona un enlace a una URL que a su vez posee un código de autorización, el cual hay que pegar en el *notebook* de Google Colab. Una vez hecho esto ya se puede montar Google Drive en Google Colab pulsando sobre 'Activar Drive':

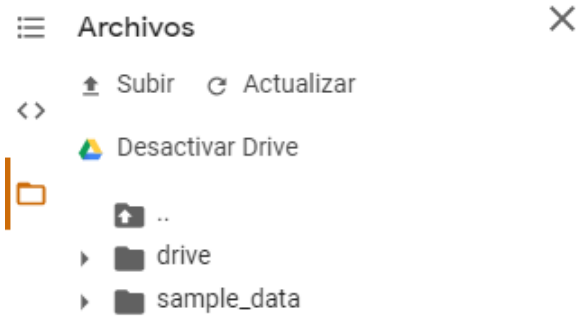

Figura 28: Google Drive activado en Google Colab

Después de esto, y una vez se tiene accesible el conjunto de datos, hay que recordar cambiar el tipo de entorno de ejecución del cuaderno de Google Colab para poder usar una GPU y entrenar la red con ella. Tal y como se dijo en la sección anterior, hay que cambiarlo dentro del menú entorno de ejecución.

A partir de este momento ya se puede escribir el código en Python para entrenar la red neuronal.

Lo primero de todo será importar las librerías necesarias para desarrollar el programa:

```
  # Se importan librerías
  import cv2 # OpenCV
  import tensorflow as tf
  import numpy as np # Arrays y matrices
  import matplotlib.pyplot as plt # Gráficas
  import os # Sistema de archivos
  from sklearn.model_selection import train_test_split #         Entreno-
Validación
  import random # Números aleatorios
  import gc # Limpiar y borrar variables
```

Se van a utilizar en total 4000 imágenes (2000 de cada clase) para la red. De estas 4000, el 80% será parte del *dataset* de entrenamiento, mientras que el 20% restante será para el *dataset* de validación:

```
  # Se corta el dataset de entreno y se cogen 2000 imágenes de cada clase:
  entreno_imagenes = entreno_perros[:2000] + entreno_gatos[:2000]
  # Se barajan estas 4000 imágenes de forma aleatoria:
  random.shuffle(entreno_imagenes)
  # SALTO DE CÓDIGO #
  # Repartir los datos entre entrenamiento y validación
  # 80% entreno, 20% validación:
  x_entreno,x_valid,y_entreno,y_valid = train_test_split(x,y,test_siz
e=0.20,random_state=2)
```

La red estará conformada por tres capas de convolución con funciones de activación *ReLU* y otras tres capas de *max pooling*, alternándose unas con otras. Las capas de convolución usarán ventanas de 5x5, mientras que las capas de *max pooling* serán de 3x3. Inmediatamente después se usará una capa para transformar el tensor 3D a una sola dimensión. Por último, se usarán dos capas densamente conectadas. La última de ellas tendrá una función de activación *sigmoid*, la más adecuada para problemas de clasificación binaria como este caso.

La arquitectura del modelo se puede ver en la siguiente imagen:

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 176, 176, 32)      2432
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 58, 58, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 54, 54, 64)        51264
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 18, 18, 64)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 14, 14, 128)       204928
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 4, 4, 128)         0
_________________________________________________________________
flatten_1 (Flatten)          (None, 2048)              0
_________________________________________________________________
dense_1 (Dense)              (None, 512)               1049088
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 513
=================================================================
Total params: 1,308,225
Trainable params: 1,308,225
Non-trainable params: 0
```

Figura 29: Modelo de la red neuronal creada

Como se puede apreciar, el clasificador cuenta con 1.3 millones de parámetros. Una vez definido el modelo de

red se utiliza una técnica llamada *data augmentation*, que sirve para aumentar el *dataset* de entrenamiento haciendo pequeñas variaciones en las imágenes, evitando también el *overfitting* o sobreajuste del modelo, para que posteriormente al entreno la red neuronal responda bien ante muestras de imágenes nuevas que no se usen en el conjunto de datos de entrenamiento.

```
  """ Data augmentation. Previene el sobreajuste aumentando el dataset
haciendo pequeñas variaciones en las imágenes de entreno (no utiliza las
originales): """
  entreno_pixel = ImageDataGenerator(rescale=1.0/255, # Px (0-1)
                                     rotation_range=40, # Rango rotación
                                  width_shift_range=0.2, # Rango traslación
                                 height_shift_range=0.2, # Rango traslación
                                      shear_range=0.2, # Rango cortes
                                       zoom_range=0.2, # Rango zoom
                                    horizontal_flip=True) # Voltea imágenes
```

Después se utiliza *binary cross-entropy* como función de *loss* y RMSprop como optimizador para entrenar la red, utilizando un *learning rate* de 0.0001. Además, se establece un tamaño de lote de 32 y 150 *epochs* para entrenar la red:

```
  # Tamaño de lotes de paquetes. Normalmente 32 (producto de 2):
  tamano_lote = 32
  # Solo hay dos clases, por eso se utiliza esta función de coste
  # Se utiliza un learning rate pequeño y la precisión como métrica:
  modelo.compile(loss='binary_crossentropy',
                 optimizer=optimizers.RMSprop(lr = 0.0001),
                 metrics=['acc'])
```

Una vez definido todas estas funciones e hiperparámetros se entrena la red, guardando después del proceso el modelo en un único archivo en formato de Keras (.h5). En este archivo están descritos tanto la topología como los pesos de la red.

Mediante la siguiente gráfica se puede apreciar la evolución y el resultado final de la *accuracy* (eficacia) del modelo creado tras entrenar la red:

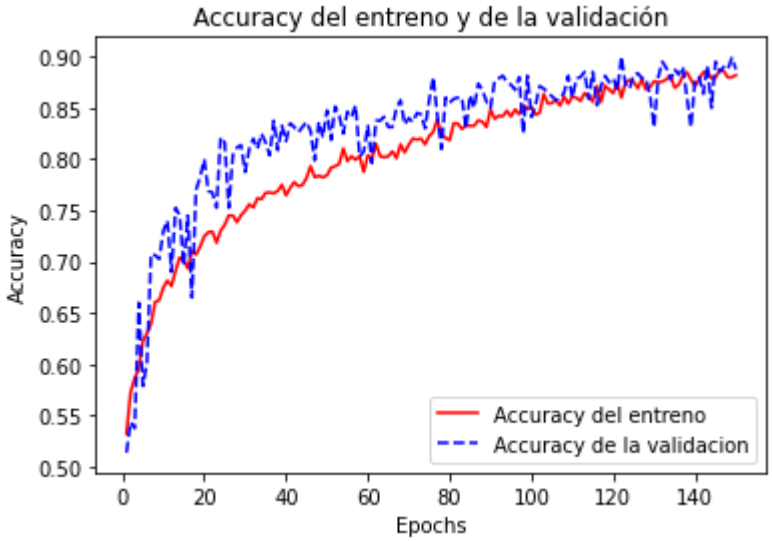

Figura 30: *Accuracy* del clasificador de imágenes binario

Como se puede apreciar, tras entrenar la red neuronal, la *accuracy* del entrenamiento y de validación va aumentando progresivamente a medida que van pasando los *epochs*. Tras 150 *epochs*, el valor final de la *accuracy* de entrenamiento está en torno a un gran valor del 90%. Se puede ver también como la *accuracy* del conjunto de validación es aproximadamente cercana a este valor, por lo que no hay mucha diferencia de valores.

Por otra parte, se puede ver también el resultado final de la *loss* de la red en esta otra imagen:

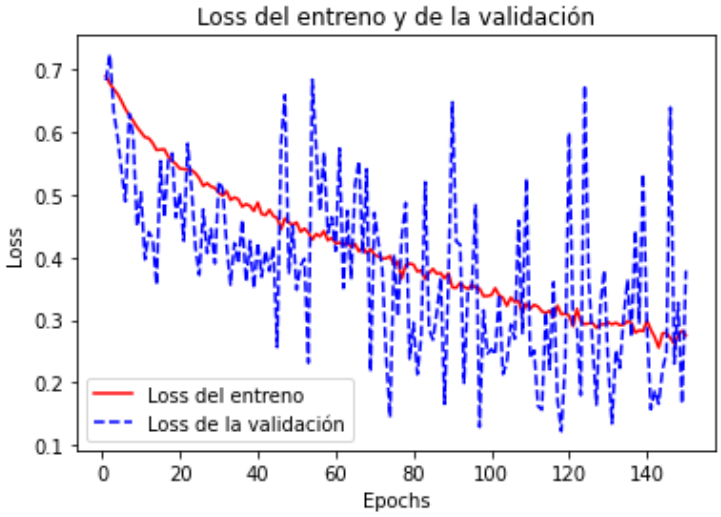

Figura 31: *Loss* del clasificador de imágenes binario

En esta ocasión, cuanto más baja sea la *loss*, mejor será el modelo. Se puede apreciar cómo tanto la *loss* de entreno y la *loss* de validación van bajando a medida que van pasando las *epochs*, con variaciones en la *loss* de validación. Al final, la *loss* del entreno y la *loss* de validación están en torno a 0.3.

Por lo tanto, y tras analizar tanto las gráficas de *accuracy* como de *loss* del modelo creado se puede concluir que no hay *overfitting*, ya que en ambas gráficas los valores de entrenamiento y validación son muy parejos, y, por tanto, se puede concluir que se trata de un buen modelo.

4.2.1.2 Conversión a TensorFlow

Una vez ya se tiene un modelo de red del clasificador de imágenes binario de perros y gatos creado en formato compatible con Keras (.h5) hay que transformarlo, ya que este formato no es compatible con el Movidius Neural Compute Stick ni tampoco con OpenCV, por lo que es necesario transformar dicho modelo a formato de TensorFlow, *framework* en el que se apoya Keras y que está diseñado para tareas de *machine learning*. Y lo más importante, compatible con OpenCV y OpenVINO.

Para ello se necesita tener el modelo de red compatible con formato de TensorFlow en dos formatos distintos: en formato binario (.pb) y en formato de texto (.pbtxt).

Para obtener estos dos archivos hace falta realizar un programa en Python que pueda transformar el archivo en formato de Keras (.h5) a formato binario (.pb) de TensorFlow. Para ello es necesario que la versión de TensorFlow sea menor que la versión 2.0, pues hay varios métodos en el código que dan problemas con versiones iguales o superiores a esta versión.

Con este programa lo que se hace es cargar el modelo de Keras, escoger la salida de la red y eliminar toda la información no necesaria para la inferencia, convirtiendo el modelo en formato binario de TensorFlow (.pb):

```
  # Se pone la fase de aprendizaje a cero y se carga el modelo completo:
  K.set_learning_phase(0)
  modelo = load_model('/home/user/T-F-
M/Valderas/modelos/modelo.h5', compile = False)
  # SALTO DE CÓDIGO #
  # Se congela la sesión y se elige la salida del modelo, eliminando
información innecesaria:
  grafo_exportado = freeze_session(K.get_session(),
                   output_names=[out.op.name for out in modelo.outputs])

  # Se escribe el modelo '.pb', guardándolo en el sistema del PC:
  tf.train.write_graph(grafo_exportado, "modelo", "modelo.pb", as_text=F
alse)
```

Una vez se obtenga este archivo, se usa de nuevo otro programa en Python que lea este archivo binario y lo transforme para poder ser guardado como archivo en formato de texto (.pbtxt):

```
  # Lee el modelo binario de TF:
  with tf.gfile.FastGFile('/home/user/T-F-
M/Valderas/modelos/modelo.pb', 'rb') as f:
      graph_def = tf.GraphDef()
      graph_def.ParseFromString(f.read())
  # SALTO DE CÓDIGO #
  # Guarda como archivo de texto en el PC:
  tf.train.write_graph(graph_def, "modelo", "modelo.pbtxt", as_text=True)
```

Una vez se tienen estos dos archivos hay que hacer varios ajustes. Como se comentó en la subsección anterior, el modelo creado tiene una capa *flatten*, que es la encargada de transformar el tensor de 3D a una sola dimensión. Esta capa tiene problemas de nomenclatura con OpenCV, por lo que hay que editar el archivo de texto creado (.pbtxt) para solucionarlos.

Dentro del archivo de texto hay que eliminar los nodos con los nombres "flatten/Shape", "flatten/strided_slice", "flatten/Prod" y "flatten/stack". Además, también hay que sustituir la operación del nodo llamado “flatten/Reshape”:

```
  node {
    name: "flatten_1/Reshape"
    op: "Reshape"
    input: "max_pooling2d_3/MaxPool"
  }
```

Por:

```
  node {
    name: "flatten_1/Reshape"
    op: "Flatten"
    input: "max_pooling2d_3/MaxPool"
  }
```

Una vez hecho esto, ya se tienen los dos archivos preparados y listos para cargarlos y realizar el proceso de inferencia con ellos utilizando el *backend* de OpenCV. Es decir, con estos dos archivos ya se puede realizar la fase de inferencia en tres de las cuatro plataformas de ejecución descritas en la sección anterior.

Si por el contrario se quiere realizar el proceso de inferencia con el *backend* de OpenVINO hay que realizar un paso adicional que se verá a continuación.

4.2.1.3 Representación intermedia

Para ejecutar cualquier red neuronal con OpenVINO hay que transformar el modelo ya entrenado en archivos de representación intermedia. Para ello hay que ejecutar el optimizador de modelos de OpenVINO que se encarga de hacer esta transformación.

Así, con el terminal de Ubuntu se ejecuta el programa *mo.py* que viene con OpenVINO y que se encarga de crear los archivos .xml y .bin de representación intermedia:

```
  sudo /opt/intel/openvino/deployment_tools/model_optimizer/mo.py --
input_model "/home/user/ruta-archivo.pb" --input_shape [1,180,180,3]
```

Como se puede ver, se le pasa como entrada el directorio donde se encuentra el modelo binario de TensorFlow y la forma de las imágenes de entrada (1 como tamaño de lote, 180 como dimensión de alto, 180 como dimensión de anchura y 3 canales de color).

Tras ejecutarlo y esperar un poco se crearán los archivos de representación intermedia:

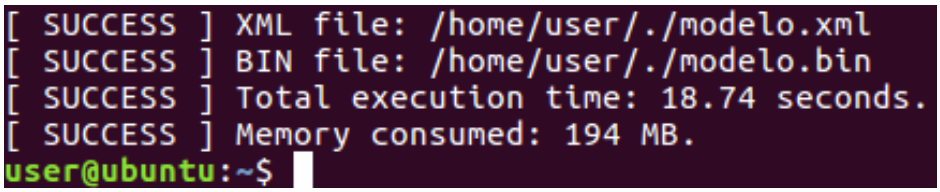

Figura 32: Operación satisfactoria del optimizador de modelos de OpenVINO

Con los archivos .bin y .xml ya creados ya se puede realizar la inferencia con el *backend* de OpenVINO. Es decir, con estos dos archivos ya se puede realizar la fase de inferencia en la única plataforma de ejecución del proyecto que usa OpenVINO como *backend* con el Movidius Neural Compute Stick.

A partir de aquí ya no hace falta hacer más conversiones. Solo falta cargar los archivos correspondientes dependiendo del *backend* usado y realizar la inferencia en cada una de las plataformas propuestas en el proyecto, algo que se verá en el siguiente episodio de la memoria.

4.2.2 Redes neuronales ya entrenadas

En esta sección se van a describir las otras tres redes neuronales convolucionales utilizadas en el proyecto. Se tratan de modelos ya entrenados que son descargados con OpenVINO. Para ello se ejecuta el programa *downloader.py* que se encargará de hacerlo automáticamente y que se encuentra en la siguiente ruta en la máquina virtual de Ubuntu:

```
/opt/intel/openvino/deployment_tools/open_model_zoo/tools/downloader
```

Esto descargará modelos ya entrenados, entre los que se encuentran redes neuronales convolucionales para clasificación de imágenes como DenseNet121 [26], DenseNet169 [26] y ResNet50 [27], que son las tres redes que van a ser usadas y que son capaces de clasificar hasta 1000 categorías distintas pertenecientes al ILSVRC, el concurso anual de software de ImageNet.

Los modelos descargados de estas tres redes están en formato compatible con Caffe, otro de los *frameworks* compatibles con OpenVINO, por lo que son descargados en formato .caffemodel y .prototxt, que son los archivos que contienen los pesos y la arquitectura de la red, respectivamente.

Como ocurrió anteriormente, estos dos archivos están preparados y listos para cargarlos y realizar el proceso de inferencia con ellos utilizando el *backend* de OpenCV. Si por el contrario se quiere realizar el proceso de inferencia con el *backend* de OpenVINO hay que utilizar nuevamente el optimizador de modelos para crear los archivos de representación intermedia.

Para ello se utiliza de nuevo el programa *mo.py*. En esta ocasión, se utiliza como modelo de entrada el archivo .caffemodel descargado:

```
python3 /opt/intel/openvino/deployment_tools/model_optimizer/mo.py --input_model "/home/user/ruta-archivo.caffemodel"
```

Tras ejecutar este comando en el terminal de Ubuntu se crean los archivos de representación intermedia, y, por tanto, ya se puede realizar la inferencia también con el *backend* de OpenVINO y usar el Movidius Neural Compute Stick con estas tres redes ya entrenadas. A partir de aquí solo hace falta cargar los archivos correspondientes dependiendo del *backend* usado y realizar la inferencia en cada una de las plataformas propuestas en el proyecto, algo que se verá en el siguiente episodio del proyecto.

A continuación, se presentará por orden creciente de número de parámetros los tres distintos modelos usados ya entrenados para conocer un poco más acerca de ellos.

4.2.2.1 DenseNet121

DenseNet121 es una red neuronal convolucional usada normalmente para tareas de visión artificial perteneciente a la familia de redes DenseNet (*Densely Connected Convolutional Network*).

Posee una estructura compleja, ya que tiene al principio una capa de convolución de 7x7 e inmediatamente después una capa de *max-pooling* de 3x3 antes de comenzar con lo que se conoce como el primer bloque denso. Y es que DenseNet121 es una red que, tras las dos primeras capas, posee un total de cuatro bloques densos y un total de tres capas de transición:

- Cada bloque denso a su vez está conformado por bloques de convolución, cada uno de los cuales contiene una capa de convolución de 1x1 para reducir características y otra capa de convolución de 3x3. En estos bloques de convolución los *feature maps* se van concatenando unos con otros [28]. Dependiendo del punto de la arquitectura de la red donde se encuentre, estos bloques de convolución serán repetidos seis veces la primera vez, doce veces la segunda vez, veinticuatro veces la tercera vez, y dieciséis veces la cuarta y última vez.
- También se tienen las capas de transición. Estas capas sirven también para reducir el número de *features* y contienen a su vez una capa de convolución de 1x1 y otra capa de *average-pool* de 2x2. Son capas situadas inmediatamente después de cada uno de los bloques densos (exceptuando el último y cuarto bloque denso).

Tras el último bloque denso viene una fase de clasificación con una capa de *average-pool* y una capa densamente conectada con capa de activación *softmax,* perfecta para aplicaciones de clasificación de imágenes multiclase como en este caso.

Todo esto se puede ver resumido en la siguiente imagen:

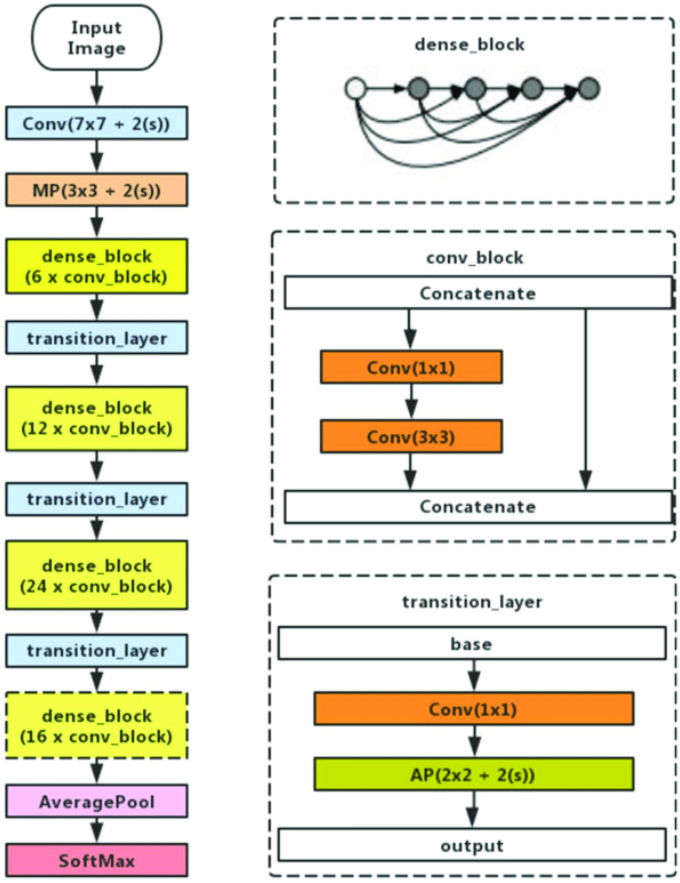

Figura 33: Arquitectura DenseNet121 [29]

La red tiene en total 121 capas (la capa inicial de convolución, 12 capas del primer bloque denso, 24 capas del segundo bloque denso, 48 capas del tercer bloque denso, 32 capas del último bloque denso, 3 de todas las capas de transición y la capa densamente conectada del final), de ahí su nombre.

Por último, destacar que DenseNet121 tiene un total de 8 millones de parámetros.

4.2.2.2 DenseNet169

DenseNet169 es otra red de la familia DenseNet para aplicaciones de visión artificial. Su arquitectura es la siguiente:

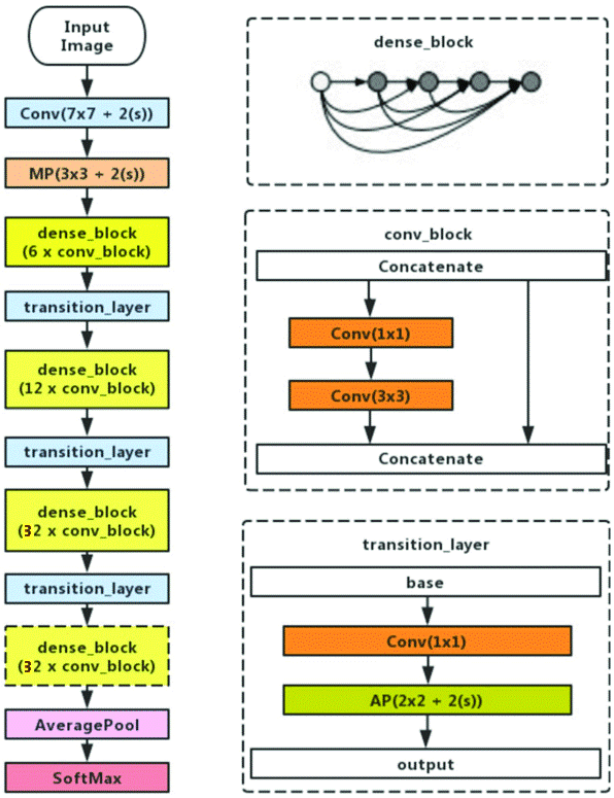

Figura 34: Arquitectura DenseNet169

Como se puede apreciar en la imagen, posee la misma estructura de DenseNet121, puesto que se tratan de modelos de la misma familia de redes. Solo existe una gran diferencia en DenseNet169: los bloques de convolución se repiten 32 veces en los dos últimos bloques densos.

La red tiene en total 169 capas (la capa inicial de convolución, 12 capas del primer bloque denso, 24 capas del segundo bloque denso, 64 capas del tercer bloque denso, 64 capas del último bloque denso, 3 de todas las capas de transición y la capa densamente conectada del final), de ahí su nombre.

Por último, destacar que DenseNet169 tiene un total de 14.3 millones de parámetros.

4.2.2.3 ResNet50

ResNet50 es una red usada normalmente para aplicaciones de visión artificial perteneciente a la familia de redes ResNet (Residual Networks). Su estructura es la siguiente:

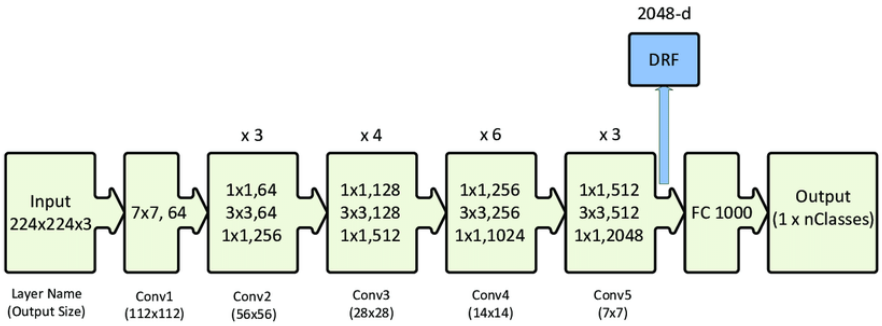

Figura 35: Arquitectura ResNet50 [30]

Como se puede ver en la imagen de arriba, se empieza con una primera capa de convolución con 64 diferentes

filtros de tamaño 7x7 seguido de una capa de *max-pooling* con ventana de 3x3 (no está incluida en la imagen).

Posteriormente le siguen una serie de bloques de convolución que se repiten un número determinado de veces [31]:

- En primer lugar, hay un bloque de convolución compuesto por tres capas: una de 64 filtros de tamaño 1x1, otra de 64 filtros de tamaño 3x3 y una última de 256 filtros de tamaño 1x1. Estas tres capas se repiten un total de tres veces cada una, por lo que al final se tienen un total de 9 capas en este paso.
- Inmediatamente después se tiene otro bloque de convolución compuesto de nuevo por otras tres capas: una de 128 filtros de tamaño 1x1, otra de 128 filtros de tamaño 3x3 y la última de 512 filtros de tamaño 1x1. Estas tres capas son repetidas un total de cuatro veces cada una, por lo que esta vez se tienen un total de 12 capas en este bloque.
- El siguiente bloque de convolución posee otras tres capas: la primera de 256 filtros de tamaño 1x1, la segunda de 256 filtros de tamaño 2x2 y la tercera de 1024 filtros de tamaño 1x1. Todas estas capas son repetidas un total de seis veces cada una, lo que hace que haya un total de 18 capas en este bloque.
- El último bloque de convolución posee nuevamente tres capas: una de 512 filtros de tamaño 1x1, otra posterior de 512 filtros de tamaño 3x3 y una última de 2048 filtros de tamaño 1x1. Como estas tres capas son repetidas tres veces, el total de capas en este último bloque de convolución es de 9 capas.

Después de estos bloques de convolución se hace una operación de *average-pool* que se termina con una capa densamente conectada que contiene 1000 nodos. Al final se utiliza la función de activación *softmax*, ideales para aplicaciones de clasificación de imágenes multiclase. Así, se tiene una sola capa en este tramo de la red (no se cuenta las funciones de activación ni las capas de *pooling*) [31].

Si se suman todas las capas de la red se tienen un total de 50 (1+9+12+18+9+1), motivo por el cual ResNet50 se llama así.

Por último, hay que destacar que ResNet50 cuenta con un total de 25.6 millones de parámetros.

5 RESULTADOS

En este capítulo se muestran los resultados obtenidos al realizar la fase de inferencia de las redes neuronales descritas en el capítulo anterior en las distintas plataformas del proyecto. Se van a medir los tiempos de clasificación para realizar la media, normalizar los resultados y así calcular el intervalo de confianza al 95%. Por último, se muestra una comparativa del rendimiento de las redes en todas las plataformas con medidas más robustas gracias a un documento en Excel (que se encuentra en la parte de anexos) y a un diagrama de cajas.

Para realizar la inferencia, se sigue un proceso similar en todas las redes y plataformas. Se presenta a continuación un diagrama de flujo para comprender el desarrollo general del proceso de inferencia:

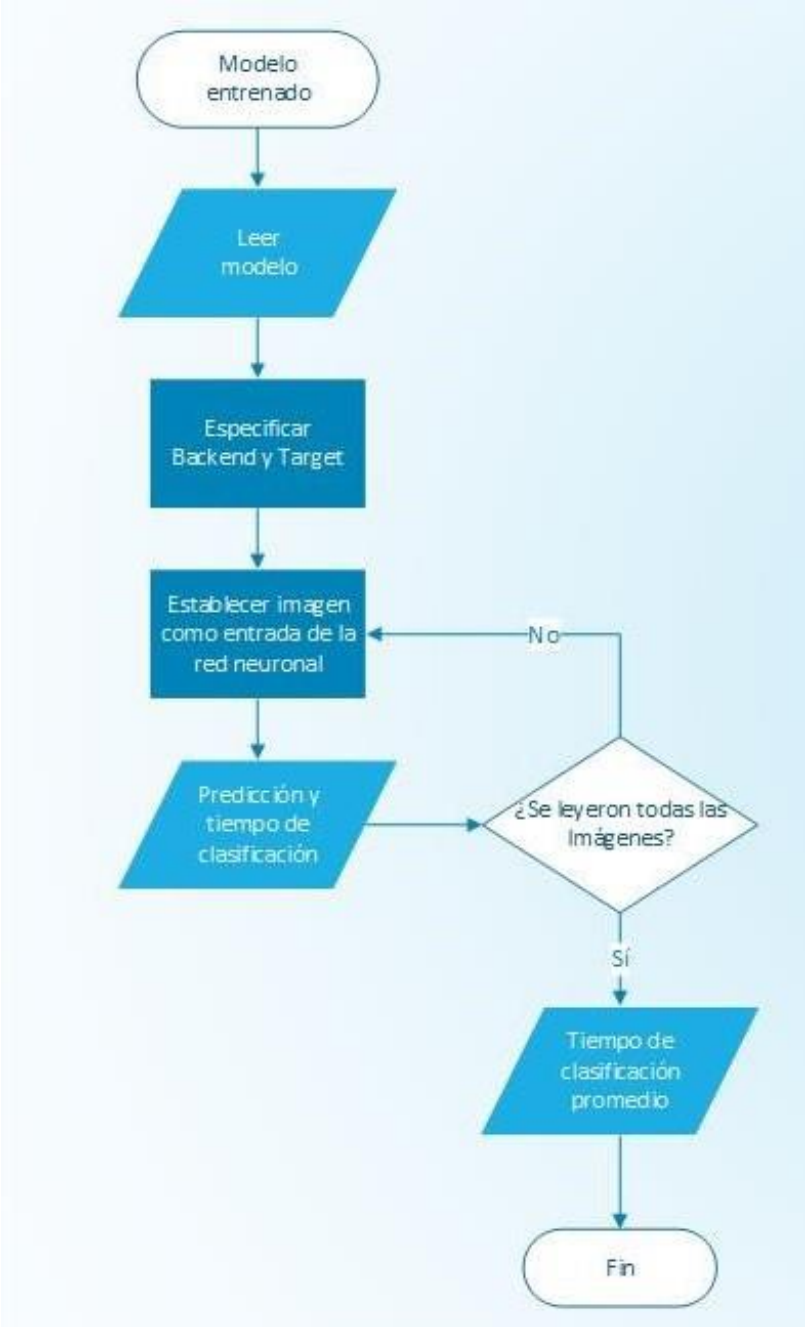

Figura 36: Diagrama de flujo del proceso de inferencia

Es proceso se hará mediante un programa en Python, que básicamente es el mismo en todas las redes y

plataformas salvo algunas pequeñas diferencias. Lo primero que se realiza es cargar las librerías y el modelo de la red ya entrenado:

```
  # Se importan librerías:
  import os # Listado archivos
  import cv2 # OpenCV
  import numpy as np # Manipular arrays
  import time # Medir el tiempo de ejecución
  # Se carga el modelo:
  red = cv2.dnn.readNetFromTensorflow('/home/pi/T-F-
M/Valderas/modelos/modelo.pb', '/home/pi/T-F-
M/Valderas/modelos/modelo.pbtxt'
```

En este caso en particular se está cargando el modelo del clasificador binario en formato compatible con TensorFlow. Si en cambio se quisiera cargar otro modelo, simplemente se cargarían los archivos correspondientes a ese modelo teniendo siempre en cuenta la plataforma de ejecución:

- Formato del *framework* TensorFlow: archivos .pb y .pbtxt. Usar con CPU, GPU y Raspberry Pi.
- Formato del *framework* Caffe: archivos .caffemodel y .prototxt. Usar con CPU, GPU y Raspberry Pi.
- Formato del motor de inferencia de OpenVINO: archivos .bin y .xml. Usar con Raspberry Pi + NCS.

Después de esto, y tal y como aparece en la figura anterior, es importante establecer en todos los programas de Python el *backend* y el dispositivo objetivo (*target*) donde se ejecutará el modelo, dependiendo de la plataforma de ejecución:

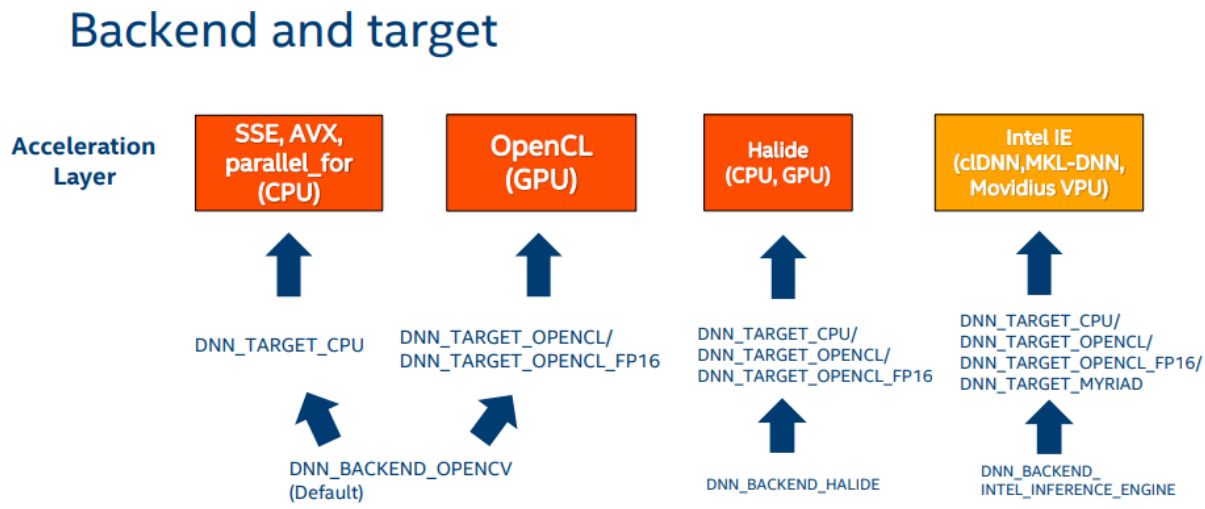

Figura 37: *Backends* y *targets* de OpenCV [32]

De esta forma, si se tienen en cuenta las cuatro plataformas establecidas para el presente proyecto, se tiene lo siguiente:

- Si se quiere ejecutar una red con la CPU del portátil se utilizará el *backend* de OpenCV y CPU como *target*.
- Si se quiere ejecutar una red con una GPU de Colab se utilizará el *backend* de OpenCV y OPENCL como *target*.
- Si se quiere ejecutar una red con la Raspberry Pi 3 B+ se utilizará el *backend* de OpenCV y CPU como *target*.
- Si se quiere ejecutar una red con la Raspberry Pi 3 B+ y el Movidius Neural Compute Stick se utilizará el *backend* del motor de inferencia de OpenVINO (*INFERENCE ENGINE*) y MYRIAD como *target*.

De esta manera, si por ejemplo se quisiera cargar el modelo de red ya entrenado en la Raspberry Pi, se escribiría

lo siguiente:

```
  # Se especifica el dispositivo objetivo (CPU):
  red.setPreferableBackend(cv2.dnn.DNN_BACKEND_OPENCV)
  red.setPreferableTarget(cv2.dnn.DNN_TARGET_CPU)
```

Cuando ya se ha cargado el modelo de red y establecido el *backend* y el *target* se especifica el directorio donde están las imágenes a las que se le van a realizar la inferencia. Además, se crean dos *arrays* para ir recopilando las imágenes del directorio y los tiempos de clasificación de cada una de ellas:

```
  # Se especifica el directorio principal y se crea un array para recopilar
las imágenes y los tiempos:
  directorio = "/ruta/al/directorio/"
  imagenes = []
  tiempos = []
```

A partir de este momento, se leen todas las imágenes del directorio y se van añadiendo al array de imágenes creado.

Una vez todas las imágenes hayan sido leídas y añadidas al array, se crea un bucle para realizar la inferencia a cada una de ellas. Para ello se establece cada una de las imágenes como entrada de la red neuronal cargada, obteniendo la clasificación de salida para mostrarla posteriormente. Además, se mide el tiempo de clasificación de cada imagen, añadiéndolo al *array* de tiempos para poder realizar después la media de clasificación (se excluye el primer tiempo debido a retardos para establecer conexión con el NCS o con el servidor de Colab):

```
  # Para todas las imágenes del array:
  for i in range(len(imagenes)):
      # Se prepara el objeto binario grande (la imagen) como entrada de la
red neuronal:
      blob = cv2.dnn.blobFromImage(imagenes[i], size=(180, 180), ddepth
=cv2.CV_32F) # Se crea BLOB desde la imagen
      red.setInput(blob) # Se establece el BLOB creado como entrada de la
red neuronal

       # Se obtiene la clasificación de salida y se mide el tiempo de
ejecución:
      inicio = time.time() # Inicio del contador para cada BLOB
      out = red.forward() # Se calcula la salida de la red neuronal
      fin = time.time() # Final del contador para cada BLOB
     print("[Info] El tiempo de ejecución fue de {:.3} segundos".format(fin
- inicio)) # 3 cifras significativas
      if i>0: # Dentro del array se excluye la 1ra clasificación (solo del
2do elemento del array al final)
        tiempos.append(fin - inicio)
```

Por último, cuando ya se haya realizado la inferencia de todas las imágenes se crean las variables para calcular el tiempo promedio de clasificación de todas las imágenes, además de los tiempos de clasificación más rápidos y lentos para imprimirlos por pantalla. Para ello se calcula el valor medio, el valor mínimo y el valor máximo del *array* de tiempos donde están almacenados todos los tiempos de clasificación:

```
  # Se crean variables para crear las estadísticas finales:
  media = np.mean(tiempos) # Media
  minimo = np.min(tiempos) # Tiempo más rápido
  maximo = np.max(tiempos) # Tiempo más lento
```

A continuación, se presentará por orden creciente de número de parámetros los resultados de clasificación de todas las redes en todas las distintas plataformas.

5.1 Clasificador de perros vs gatos

En esta sección se muestran los resultados obtenidos con el clasificador binario de perros y gatos en todas las distintas plataformas.

5.1.1 CPU

Los resultados al ejecutar esta red con la CPU del portátil son los siguientes:

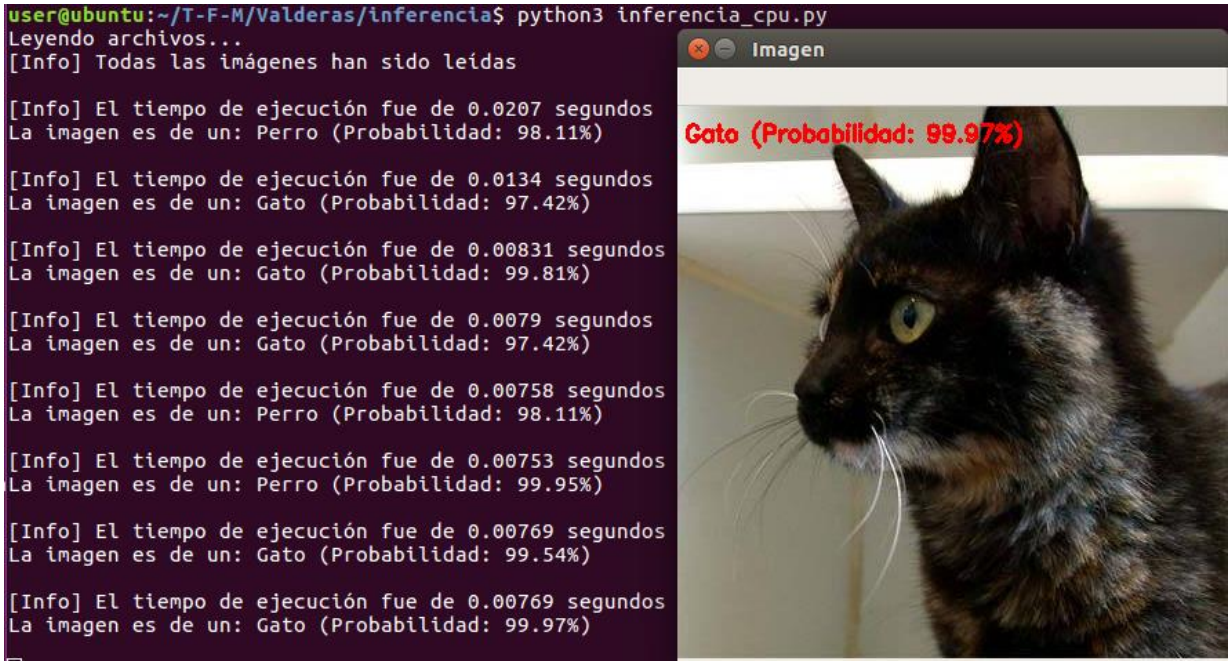

Figura 38: Resultados con la CPU del clasificador de perros vs gatos

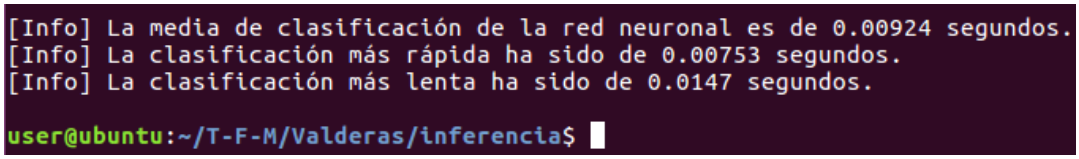

Figura 39: Tiempos de respuesta con la CPU del clasificador de perros vs gatos

En la primera imagen se puede ver un fragmento de los tiempos de clasificación de las imágenes. Como se puede apreciar, los tiempos de clasificación están en torno a las nueve milésimas.

Concretamente el tiempo de clasificación medio del clasificador es de 0.00924 segundos, tal y como se puede ver en la segunda imagen.

Respecto a la desviación estándar, su valor es el segundo mayor de todas las plataformas (0.00249). Esto quiere decir que los tiempos de clasificación se encuentran dispersos respecto al tiempo medio de clasificación si se ponen en contexto con las demás plataformas.

Una vez hecho todos los cálculos se puede decir que el intervalo de confianza al 95% es (0.00833;0.0101).

5.1.2 GPU

Los resultados del clasificador binario al usar una GPU de Colab son estos:

```
Leyendo archivos...
[Info] Todas las imágenes han sido leídas

[Info] El tiempo de ejecución fue de 0.00595 segundos
La imagen es de un: Gato (Probabilidad: 97.42%)

[Info] El tiempo de ejecución fue de 0.00181 segundos
La imagen es de un: Perro (Probabilidad: 99.95%)

[Info] El tiempo de ejecución fue de 0.00177 segundos
La imagen es de un: Gato (Probabilidad: 99.54%)

[Info] El tiempo de ejecución fue de 0.0018 segundos
La imagen es de un: Gato (Probabilidad: 99.54%)

[Info] El tiempo de ejecución fue de 0.00179 segundos
La imagen es de un: Gato (Probabilidad: 97.42%)

[Info] El tiempo de ejecución fue de 0.0018 segundos
La imagen es de un: Gato (Probabilidad: 99.81%)

[Info] El tiempo de ejecución fue de 0.0018 segundos
La imagen es de un: Gato (Probabilidad: 99.81%)

[Info] El tiempo de ejecución fue de 0.00178 segundos
La imagen es de un: Perro (Probabilidad: 98.11%)

[Info] El tiempo de ejecución fue de 0.0018 segundos
La imagen es de un: Gato (Probabilidad: 99.81%)

[Info] El tiempo de ejecución fue de 0.00184 segundos
La imagen es de un: Gato (Probabilidad: 99.97%)
```

Figura 40: Resultados con una GPU de Colab del clasificador de perros vs gatos

```
[Info] La media de clasificación de la red neuronal es de 0.0018 segundos.
[Info] La clasificación más rápida ha sido de 0.00177 segundos.
[Info] La clasificación más lenta ha sido de 0.00186 segundos.
```

Figura 41: Tiempos de respuesta con una GPU de Colab del clasificador de perros vs gatos

En la primera imagen se ve un fragmento de los tiempos de clasificación de las imágenes con la GPU. Como se puede apreciar, los tiempos de clasificación están en torno a las dos milésimas. Son tiempos parecidos al caso anterior, aunque esta vez son más rápidos pues las cifras son menores que antes.

Concretamente el tiempo de clasificación medio del clasificador es de 0.0018 segundos, tal y como se puede ver en la segunda imagen.

También se puede apreciar que los tiempos de clasificación están menos dispersos respecto el caso anterior, y, por lo tanto, más próximos a la media de clasificación. Algo que se corrobora en el documento de Excel, donde se aprecia que el valor de la desviación estándar es el menor de las cuatro plataformas (0.00002).

Una vez hecho todos los cálculos se puede afirmar que el intervalo de confianza al 95% es (0.00179;0.00181).

5.1.3 Raspberry Pi

Al usar la Raspberry Pi 3 B+, los resultados son los siguientes:

Figura 42: Resultados con la Raspberry Pi del clasificador de perros vs gatos

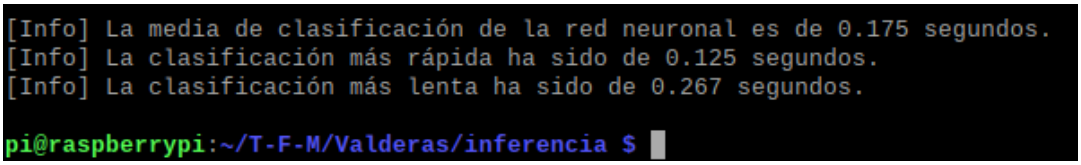

Figura 43: Tiempos de respuesta con la Raspberry Pi del clasificador de perros vs gatos

En la primera imagen se ve un fragmento de los tiempos de clasificación de las imágenes con la Raspberry Pi. Como se puede ver, los tiempos de clasificación están en torno a las décimas, siendo la plataforma de ejecución donde más lentamente se ejecuta el modelo.

El tiempo de clasificación medio del clasificador es de 0.175 segundos, tal y como se puede ver en la segunda imagen.

Una vez hecho todos los cálculos en Excel se puede decir que la desviación estándar es la mayor de las cuatro plataformas del proyecto (0.0249). Esto quiere decir que los tiempos de clasificación se extienden sobre un rango de valores más amplio respecto a la media que en las tres plataformas restantes.

Por último, destacar que el intervalo de confianza al 95% es (0.166;0.184).

5.1.4 Raspberry Pi con NCS

Cuando se usa el Neural Compute Stick en la Raspberry Pi, los resultados son los siguientes:

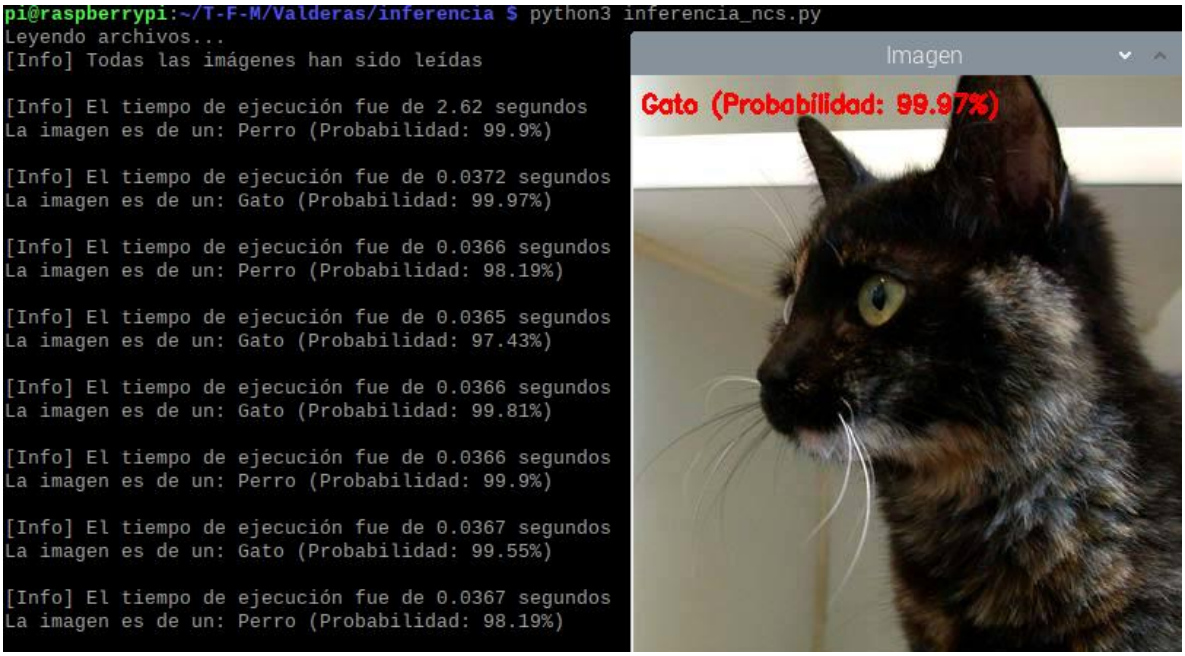

Figura 44: Resultados con la Raspberry Pi y el NCS del clasificador de perros vs gatos

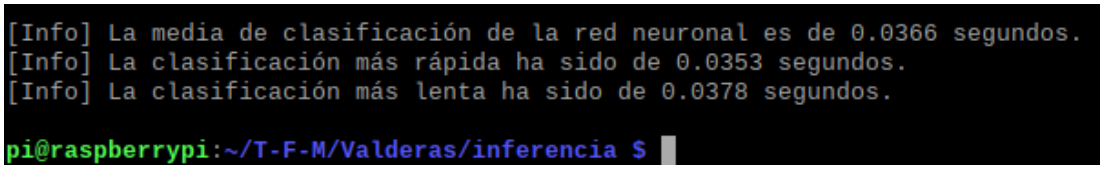

Figura 45: Tiempos de respuesta con la Raspberry Pi y el NCS del clasificador de perros vs gatos

En la primera imagen se ve un fragmento de los tiempos de clasificación de las imágenes con la Raspberry Pi usando el NCS. Como se puede ver, los tiempos de clasificación están en torno a las centésimas, mejorando mucho el rendimiento en la Raspberry respecto al caso anterior donde no se usaba el Neural Compute Stick.

El tiempo de clasificación medio del clasificador es de 0.0366 segundos, tal y como se puede ver en la segunda imagen.

Otro de los beneficios del uso del NCS en la Raspberry Pi es que la desviación estándar disminuye de forma sustancial (0.000395). Esto quiere decir que los tiempos de clasificación están agrupados cerca de la media y no se extienden sobre un rango de valores amplio respecto a la media.

Para terminar, destacar que el intervalo de confianza al 95% es (0.0365;0.0368).

5.1.5 Comparativa

Ya se ha hablado anteriormente de las medias y desviaciones estándar para calcular el intervalo de confianza al 95%. En esta ocasión, gracias al diagrama de cajas que se verá a continuación y al documento de Excel, se van a ver medidas más robustas como la mediana o el rango intercuartílico (la diferencia entre el tercer cuartil y el primer cuartil), es decir, medidas que no sean afectadas por variaciones pequeñas o valores atípicos.

En este diagrama de cajas se puede ver de forma gráfica la diferencia entre las cuatro plataformas:

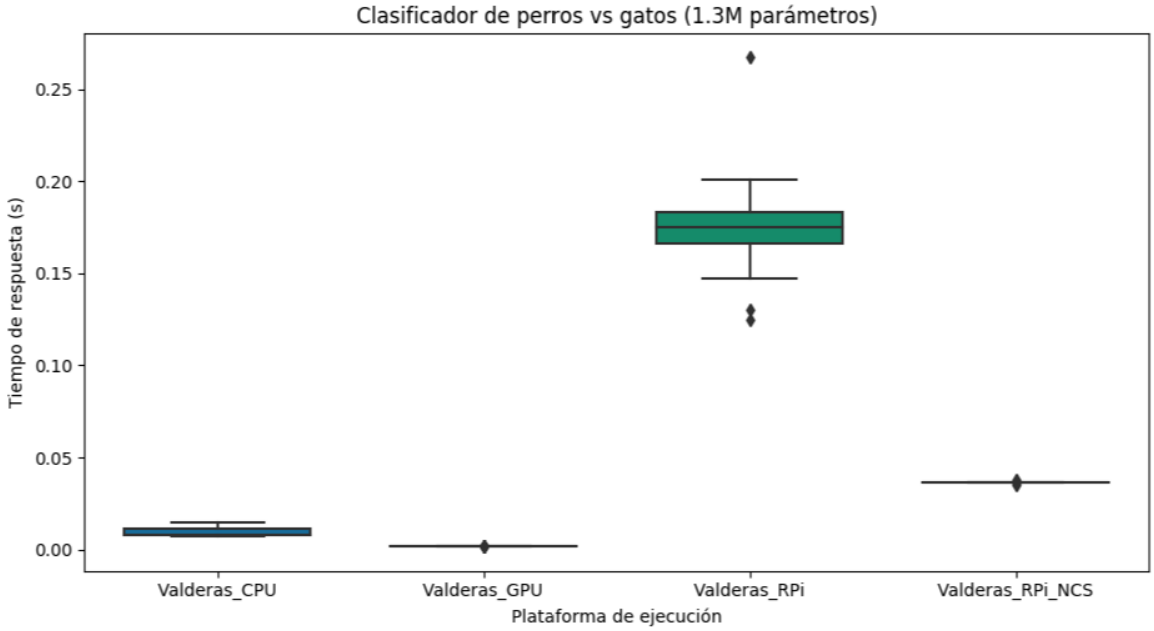

Figura 46: Diagrama de cajas del clasificador de perros vs gatos

Este diagrama de cajas muestra de una forma gráfica algunas medidas como la mediana y el rango intercuartílico. Sin embargo, no aparecen sus valores, es por ello que se calculan en el documento del Excel.

Tabla 3: Comparación del clasificador de perros vs gatos en las distintas plataformas

Plataforma	Mediana	Rango Intercuartílico
CPU	0.00782 s	0.00352
GPU	0.00180 s	0.0000100
Raspberry Pi 3 B+	0.175 s	0.0170
Raspberry Pi 3 B+ con NCS	0.0366 s	0.000100

Como se puede apreciar en el gráfico, el tiempo de respuesta tanto con la CPU como con la GPU son muy parecidos, siendo más rápidos en este último.

Los tiempos de respuesta con la Raspberry Pi son los más lentos, y a la vez los más dispersos, pues se ve como en la tabla, el valor del rango intercuartílico es el más grande de todos. Además, posee varios valores atípicos tanto por encima del máximo como por debajo del mínimo, tal y como se puede ver en la figura.

En cambio, cuando el Neural Compute Stick es introducido en la Raspberry, se puede ver como los tiempos de respuesta mejoran considerablemente. El rango intercuartílico también disminuye de forma sustancial. Esto es visible al ver que en la tabla su valor es prácticamente nulo.

Si se tiene en cuenta la mediana, el uso del Neural Compute Stick en la Raspberry para esta red proporciona una aceleración del rendimiento 4.78 veces mayor (378.14%) que sin él.

5.2 DenseNet121

En esta sección se muestran los resultados obtenidos con DenseNet121 en todas las distintas plataformas.

5.2.1 CPU

Los resultados al ejecutar esta red con la CPU del portátil son los siguientes:

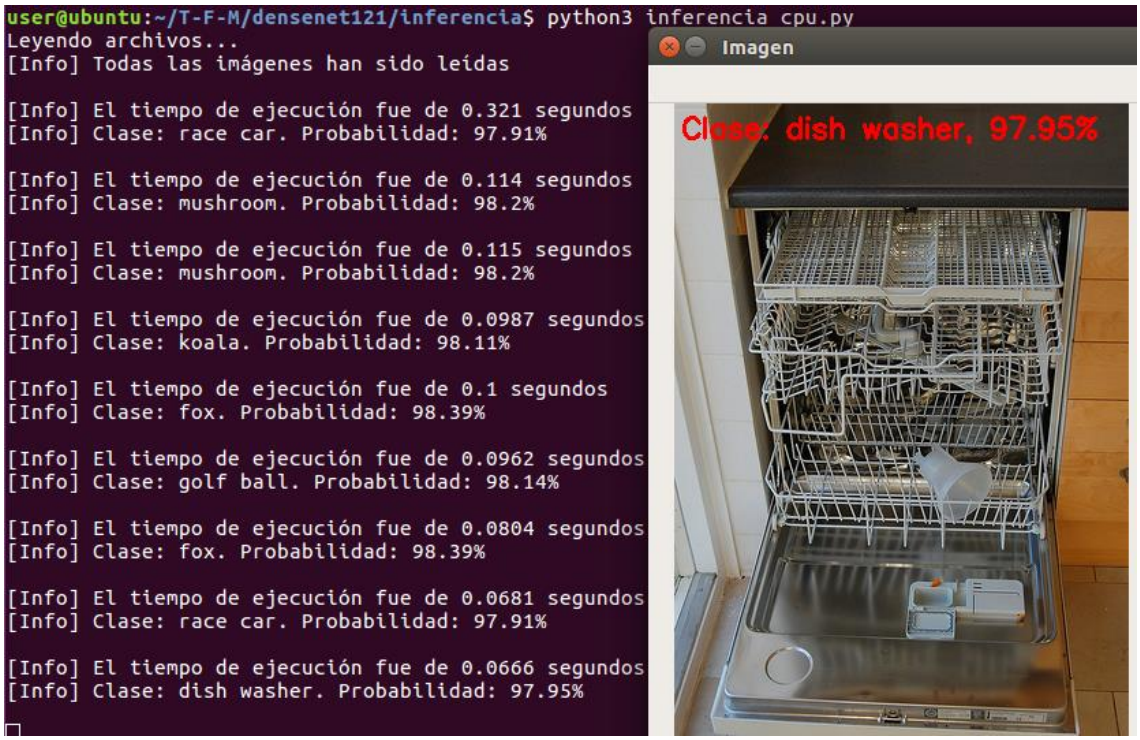

Figura 47: Resultados con la CPU de la red DenseNet121

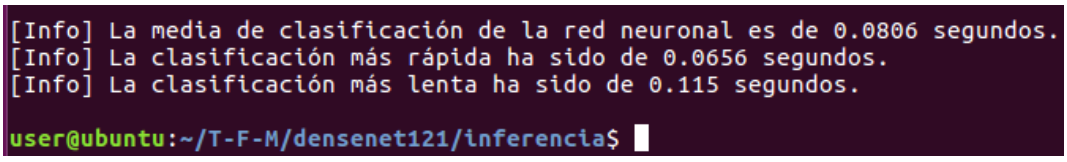

Figura 48: Tiempos de respuesta con la CPU de la red DenseNet121

En la primera imagen se puede ver un fragmento de los tiempos de clasificación de las imágenes. Como se puede apreciar, los tiempos de clasificación están alrededor de la décima. Concretamente el tiempo de clasificación medio es de 0.0806 segundos, tal y como se puede ver en la segunda imagen.

El valor de la desviación estándar es el segundo más alto de todas las plataformas con esta red, con un valor de 0.0170. De esta forma, los tiempos son los segundos más dispersos respecto a la media si se compara con otras plataformas.

Una vez hecho todos los cálculos se puede decir que el intervalo de confianza al 95% con la CPU es (0.0744;0.0868).

5.2.2 GPU

Los resultados de DenseNet121 al usar una GPU de Colab son estos:

```
Leyendo archivos...
[Info] Todas las imágenes han sido leídas

[Info] El tiempo de ejecución fue de 0.113 segundos
[Info] Clase: fox. Probabilidad: 98.39%

[Info] El tiempo de ejecución fue de 0.0374 segundos
[Info] Clase: fox. Probabilidad: 98.39%

[Info] El tiempo de ejecución fue de 0.0374 segundos
[Info] Clase: fox. Probabilidad: 98.39%

[Info] El tiempo de ejecución fue de 0.0373 segundos
[Info] Clase: fox. Probabilidad: 98.39%

[Info] El tiempo de ejecución fue de 0.022 segundos
[Info] Clase: golf ball. Probabilidad: 98.14%

[Info] El tiempo de ejecución fue de 0.0216 segundos
[Info] Clase: golf ball. Probabilidad: 98.14%

[Info] El tiempo de ejecución fue de 0.0217 segundos
[Info] Clase: golf ball. Probabilidad: 98.14%

[Info] El tiempo de ejecución fue de 0.0216 segundos
[Info] Clase: golf ball. Probabilidad: 98.14%

[Info] El tiempo de ejecución fue de 0.0213 segundos
[Info] Clase: mushroom. Probabilidad: 98.2%

[Info] El tiempo de ejecución fue de 0.021 segundos
[Info] Clase: mushroom. Probabilidad: 98.2%
```

Figura 49: Resultados con una GPU de Colab de la red DenseNet121

```
[Info] La media de clasificación de la red neuronal es de 0.0213 segundos.
[Info] La clasificación más rápida ha sido de 0.0179 segundos.
[Info] La clasificación más lenta ha sido de 0.0374 segundos.
```

Figura 50: Tiempos de respuesta con una GPU de Colab de la red DenseNet121

En la primera imagen se ve un fragmento de los tiempos de clasificación de las imágenes con la GPU. Como se puede apreciar, los tiempos de clasificación están en torno a las dos centésimas, siendo unos tiempos más rápidos que los del caso anterior con la CPU. Concretamente el tiempo de clasificación medio de la red DenseNet121 es de 0.0213 segundos, tal y como se puede ver en la segunda imagen.

También se puede apreciar que los tiempos de clasificación están menos dispersos respecto el caso anterior, y, por lo tanto, más próximos a la media de clasificación. Esto es debido a que el valor de la desviación estándar con la GPU es menor que con la CPU (0.00573< 0.0170).

Una vez hecho todos los cálculos se puede afirmar que el intervalo de confianza al 95% es (0.0192;0.0234).

5.2.3 Raspberry Pi

Al usar la Raspberry Pi 3 B+, los resultados son los siguientes:

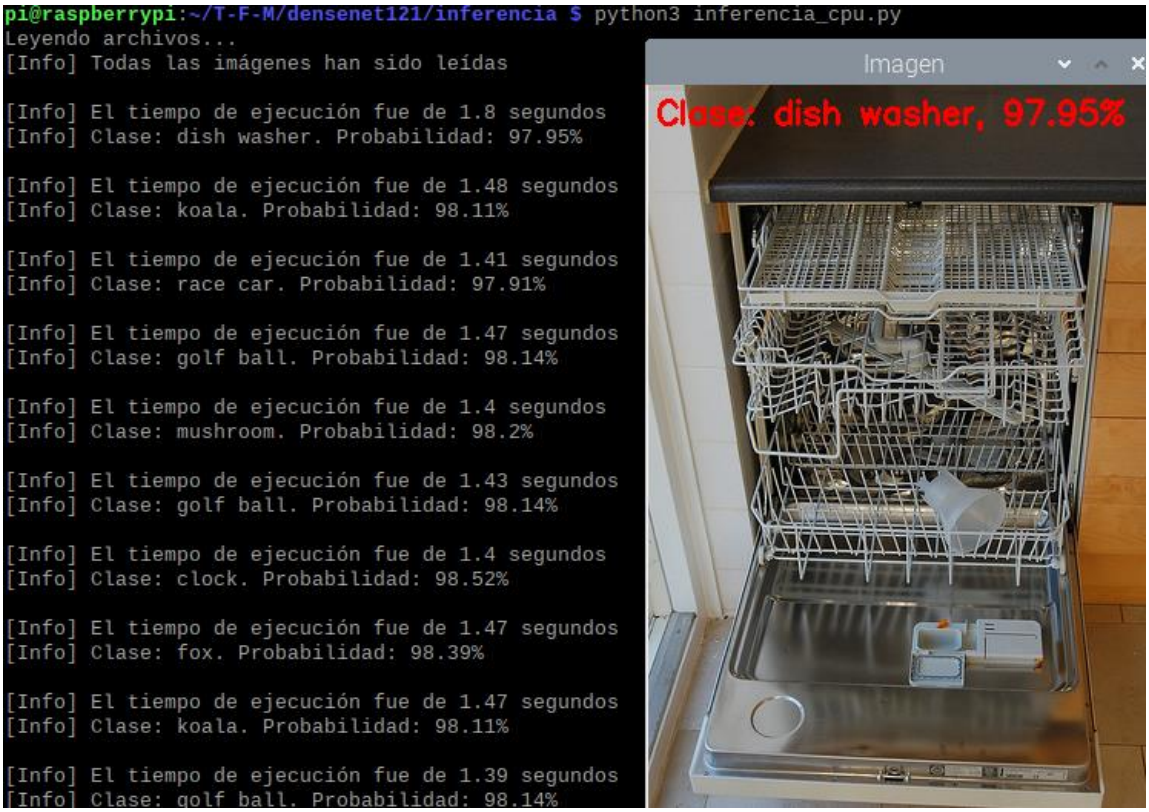

Figura 51: Resultados con la Raspberry Pi de la red DenseNet121

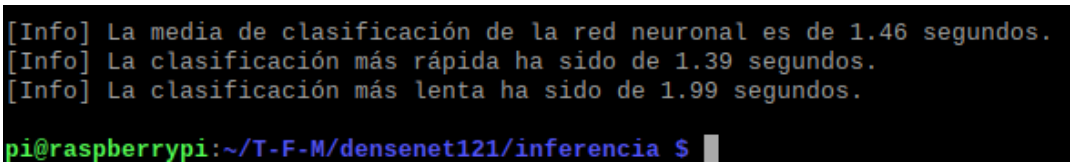

Figura 52: Tiempos de respuesta con la Raspberry Pi de la red DenseNet121

En la primera imagen se ve un fragmento de los tiempos de clasificación de las imágenes con la Raspberry Pi. Como se puede ver, los tiempos de clasificación están en torno al segundo y medio, siendo la plataforma de ejecución donde más lentamente se ejecuta el modelo. Concretamente el tiempo de clasificación medio de DenseNet121 es de 1.46 segundos, tal y como se puede ver en la segunda imagen.

Una vez hecho todos los cálculos, se puede decir que la desviación estándar es la mayor de las cuatro plataformas del proyecto (0.107). Esto quiere decir que los tiempos de clasificación se extienden sobre un rango de valores más amplio respecto a la media que en las tres plataformas restantes.

Por último, destacar que el intervalo de confianza al 95% es (1.42;1.49).

5.2.4 Raspberry Pi con NCS

Cuando se usa el Neural Compute Stick en la Raspberry Pi, los resultados son los siguientes:

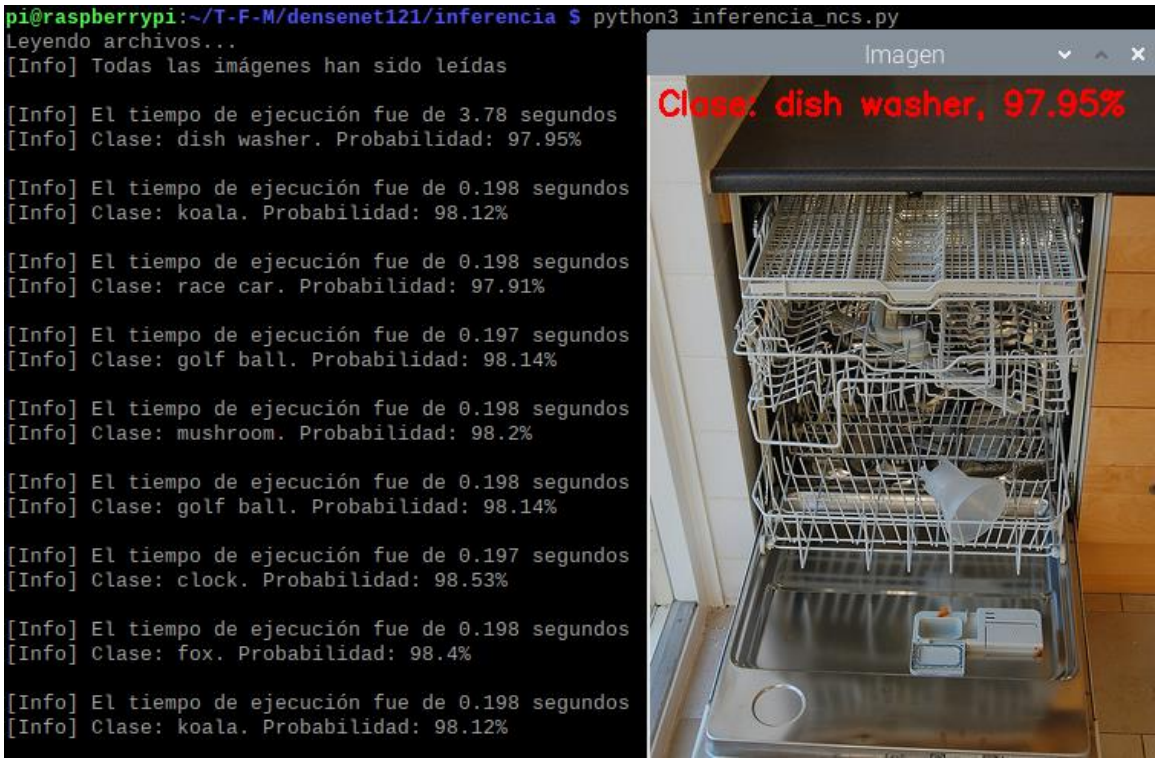

Figura 53: Resultados con la Raspberry Pi y el NCS de la red DenseNet121

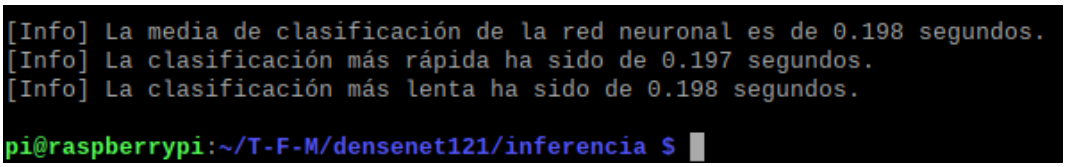

Figura 54: Tiempos de respuesta con la Raspberry Pi y el NCS de la red DenseNet121

En la primera imagen se ve un fragmento de los tiempos de clasificación de las imágenes con la Raspberry Pi usando el NCS. Como se puede ver, los tiempos de clasificación están en torno a las dos centésimas. Concretamente el tiempo de clasificación medio de la red DenseNet121 es de 0.198 segundos, tal y como se puede ver en la segunda imagen. Esto mejora mucho el rendimiento en la Raspberry respecto al caso anterior donde no se usaba el Neural Compute Stick.

Otro de los beneficios del uso del NCS es que la desviación estándar en este modelo disminuye de forma significativa. Tal es así que de hecho el valor es el menor de las cuatro plataformas (0.000351). Esto quiere decir que los tiempos de clasificación son los que están más agrupados cerca de la media y no se extienden sobre un rango de valores amplio respecto a esta.

Para terminar, destacar que el intervalo de confianza al 95% es (0.198;0.198).

5.2.5 Comparativa

Como ya ocurrió con la red anterior, se presentan en esta subsección el diagrama de cajas y la tabla comparativa para comparar la mediana, el rango intercuartílico y calcular la aceleración provocada por el uso del Neural Compute Stick en la Raspberry Pi.

En este diagrama de cajas se ve de forma gráfica la diferencia entre las cuatro plataformas:

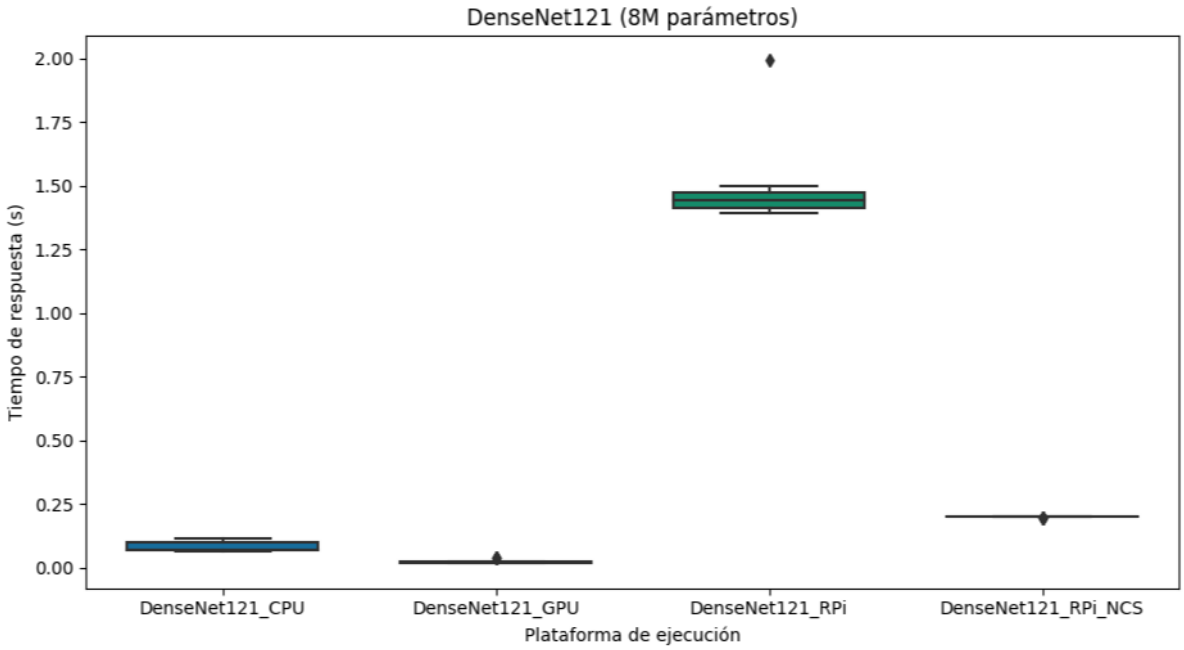

Figura 55: Diagrama de cajas de la red DenseNet121

También se tiene la tabla comparativa donde se calculan las medidas:

Tabla 4: Comparación de la red DenseNet121 en las distintas plataformas

Plataforma	Mediana	Rango Intercuartílico
CPU	0.0693 s	0.0317
GPU	0.0188 s	0.00280
Raspberry Pi 3 B+	1.44 s	0.0600
Raspberry Pi 3 B+ con NCS	0.198 s	0

Como se puede apreciar en el gráfico y en la tabla, los tiempos de respuesta y la mediana tanto con la CPU como con la GPU son muy parecidos, pero son más rápidos en este último.

Los tiempos de respuesta con la Raspberry Pi son los más lentos. De hecho, su mediana es la mayor de las cuatro plataformas tal y como se puede ver en la tabla. Por otra parte, los tiempos con las Raspberry también son los más dispersos respecto a la mediana, pues como se ve en la tabla, el valor del rango intercuartílico es el más grande de todos.

En cambio, cuando el Neural Compute Stick es introducido en la Raspberry Pi se puede ver como los tiempos de respuesta y la mediana disminuyen considerablemente. Por otra parte, el rango intercuartílico también disminuye de forma absoluta, ya que su valor es cero, tal y como se aprecia en la tabla.

Para terminar, si se tiene en cuenta la mediana, el uso del Neural Compute Stick en la Raspberry Pi para esta red proporciona una aceleración del rendimiento 7.27 veces mayor (627.27%) que sin él.

5.3 DenseNet169

En esta sección se muestran los resultados obtenidos con DenseNet169 en todas las distintas plataformas.

5.3.1 CPU

Los resultados al ejecutar esta red con la CPU del portátil son los siguientes:

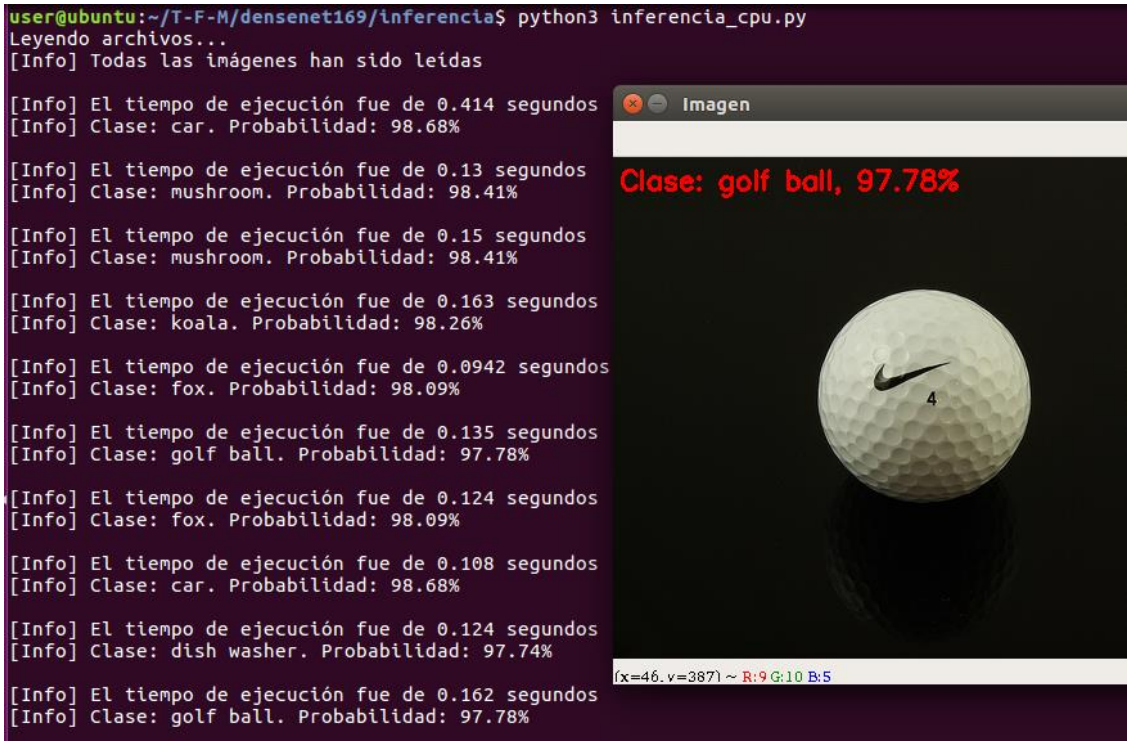

Figura 56: Resultados con la CPU de la red DenseNet169

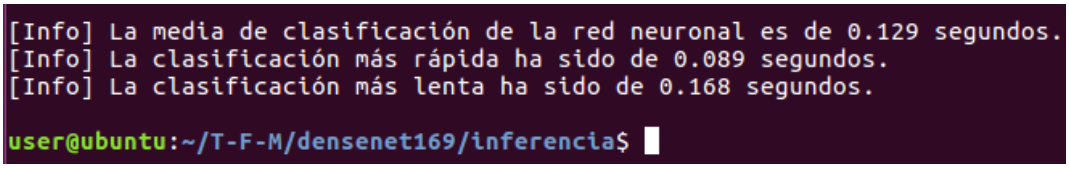

Figura 57: Tiempos de respuesta con la CPU de la red DenseNet169

En la primera imagen se puede ver un fragmento de los tiempos de clasificación de las imágenes. Como se puede ver, los tiempos de clasificación están alrededor de la décima.

Concretamente el tiempo de clasificación medio es de 0.129 segundos, tal y como se puede ver en la segunda imagen.

El valor de la desviación estándar es de nuevo el segundo más alto de todas las plataformas con esta red, con un valor de 0.0210. Por tanto, los tiempos de clasificación en esta plataforma están descentralizados respecto al tiempo medio de clasificación si se pone el resultado en contexto con las demás plataformas.

Una vez hecho todos los cálculos se puede decir que el intervalo de confianza al 95% con la CPU es (0.122;0.137).

5.3.2 GPU

Los resultados de DenseNet121 al usar una GPU de Colab son estos:

```
Leyendo archivos...
[Info] Todas las imágenes han sido leídas

[Info] El tiempo de ejecución fue de 9.69 segundos
[Info] Clase: fox. Probabilidad: 98.09%

[Info] El tiempo de ejecución fue de 0.0767 segundos
[Info] Clase: fox. Probabilidad: 98.09%

[Info] El tiempo de ejecución fue de 0.0767 segundos
[Info] Clase: fox. Probabilidad: 98.09%

[Info] El tiempo de ejecución fue de 0.0719 segundos
[Info] Clase: fox. Probabilidad: 98.09%

[Info] El tiempo de ejecución fue de 0.07 segundos
[Info] Clase: golf ball. Probabilidad: 97.78%

[Info] El tiempo de ejecución fue de 0.0677 segundos
[Info] Clase: golf ball. Probabilidad: 97.78%

[Info] El tiempo de ejecución fue de 0.0648 segundos
[Info] Clase: golf ball. Probabilidad: 97.78%

[Info] El tiempo de ejecución fue de 0.0644 segundos
[Info] Clase: golf ball. Probabilidad: 97.78%

[Info] El tiempo de ejecución fue de 0.0616 segundos
[Info] Clase: mushroom. Probabilidad: 98.41%

[Info] El tiempo de ejecución fue de 0.0613 segundos
[Info] Clase: mushroom. Probabilidad: 98.41%
```

Figura 58: Resultados con una GPU de Colab de la red DenseNet169

```
[Info] La media de clasificación de la red neuronal es de 0.0634 segundos.
[Info] La clasificación más rápida ha sido de 0.0606 segundos.
[Info] La clasificación más lenta ha sido de 0.0767 segundos.
```

Figura 59: Tiempos de respuesta con una GPU de Colab de la red DenseNet169

En la primera imagen se ve un fragmento de los tiempos de clasificación de las imágenes con la GPU. Como se puede apreciar, los tiempos de clasificación están en torno a las seis centésimas, siendo más rápidos que los del caso anterior con la CPU.

Concretamente, como se puede ver en la segunda imagen, el tiempo de clasificación medio de la red DenseNet169 es de 0.0634 segundos.

Con la GPU los tiempos de clasificación están menos dispersos respecto el caso anterior, y, por lo tanto, están más cercanos respecto a la media. Esto es debido a que el valor de la desviación estándar con la GPU es menor que con la CPU (0.00465< 0.0210).

Una vez definida la media y la desviación estándar se puede afirmar que el intervalo de confianza al 95% es (0.0617;0.0651).

5.3.3 Raspberry Pi

Al usar la Raspberry Pi 3 B+, los resultados son los siguientes:

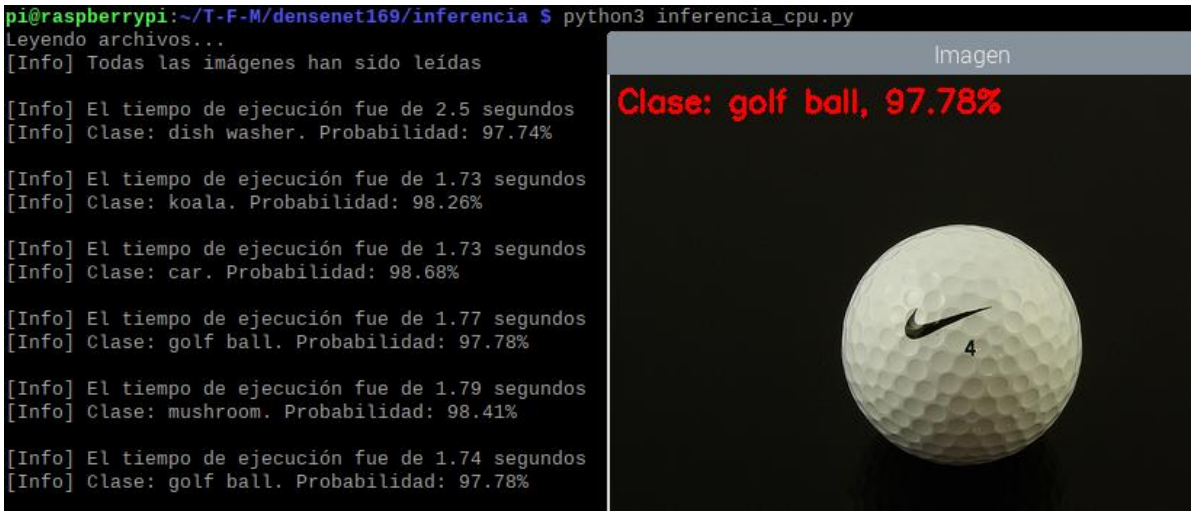

Figura 60: Resultados con la Raspberry Pi de la red DenseNet169

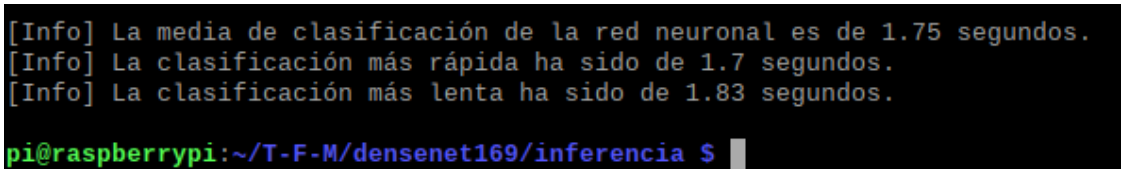

Figura 61: Tiempos de respuesta con la Raspberry Pi de la red DenseNet169

En la primera imagen se puede ver un fragmento de los tiempos de clasificación de las imágenes con la Raspberry Pi. Como se puede apreciar, los tiempos de clasificación están alrededor de los 1.7-1.8 segundos, siendo de nuevo la plataforma donde más lentamente se ejecuta el modelo de red.

Concretamente, y tal y como se ve en la segunda imagen, el tiempo de clasificación medio de DenseNet169 es de 1.75 segundos.

Una vez hecho todos los cálculos, se puede decir que la desviación estándar es de nuevo la mayor de las cuatro plataformas del proyecto (0.0347). De esta forma, los tiempos de clasificación se extienden sobre un rango de valores más amplio respecto a la media que en las tres plataformas restantes, es decir, que sus valores son los más dispersos respecto a ella.

Por último, destacar que con el uso de la Raspberry Pi el intervalo de confianza al 95% es (1.74;1.77).

5.3.4 Raspberry Pi con NCS

Cuando se usa el Neural Compute Stick en la Raspberry Pi, los resultados son los siguientes:

Figura 62: Resultados con la Raspberry Pi y el NCS de la red DenseNet169

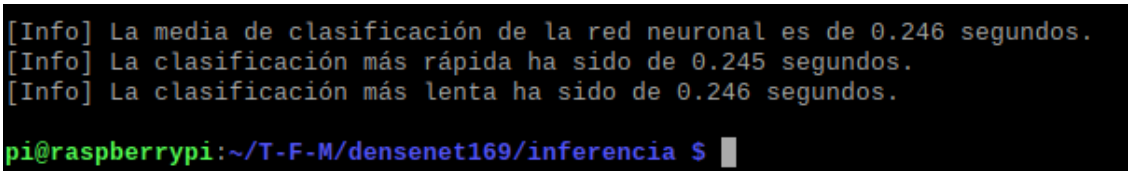

Figura 63: Tiempos de respuesta con la Raspberry Pi y el NCS de la red DenseNet169

En la primera imagen se ve un fragmento de los tiempos de clasificación de las imágenes con la Raspberry Pi usando el Neural Compute Stick. Como se puede apreciar, los tiempos de clasificación están en torno a las dos centésimas y media.

Concretamente el tiempo de clasificación medio de la red DenseNet121 es de 0.246 segundos, tal y como se ve en la segunda imagen. Como se puede apreciar, esto mejora de forma considerable la media respecto al caso anterior donde no se usaba el Neural Compute Stick en la Raspberry Pi.

Otro de los beneficios del uso del Neural Compute Stick es que la desviación estándar con este modelo también disminuye. Tal es así que de hecho el valor es el menor de las cuatro plataformas (0.000412). Por lo tanto, los tiempos de clasificación en esta plataforma son los que están más agrupados cerca de la media y no se extienden sobre un rango de valores amplio respecto a esta. Esto se puede apreciar también viendo los tiempos máximos y mínimos, ya que solo se diferencian en una milésima.

Para terminar, destacar que con el uso del Neural Compute Stick el intervalo de confianza al 95% es (0.246;0.246).

5.3.5 Comparativa

Ahora se presentan el diagrama de cajas y la tabla comparativa para comparar la mediana, el rango intercuartílico y calcular la aceleración provocada por el uso del Neural Compute Stick en la Raspberry Pi.

Aquí se presenta el diagrama de cajas:

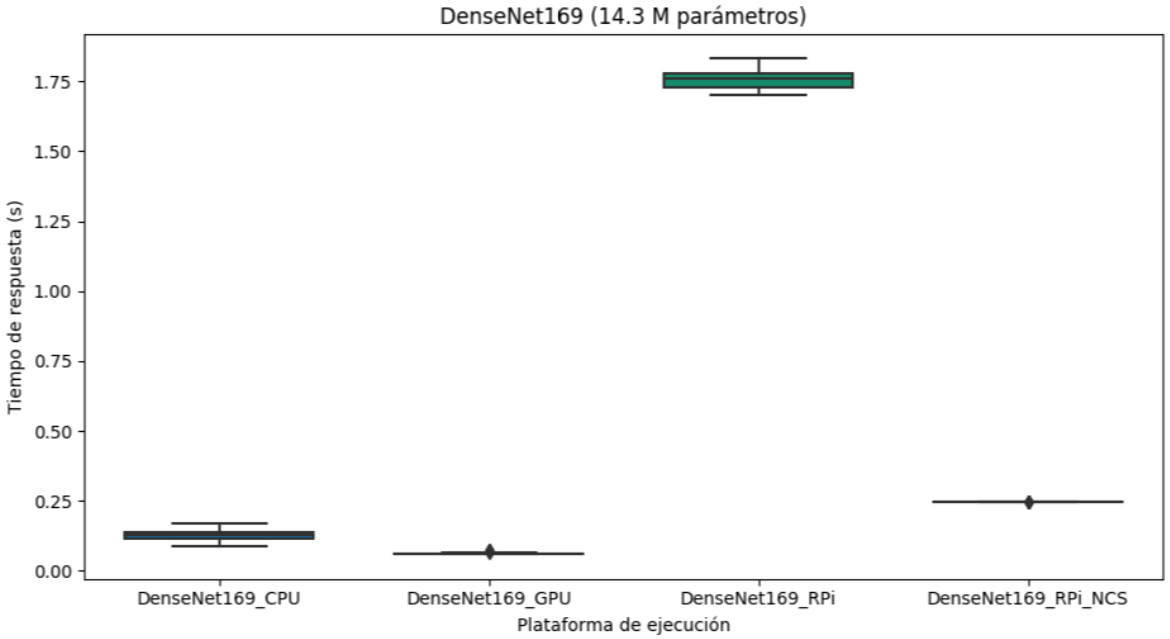

Figura 64: Diagrama de cajas de la red DenseNet169

También se tiene la tabla comparativa donde se calculan las medidas:

Tabla 5: Comparación de la red DenseNet169 en las distintas plataformas

Plataforma	Mediana	Rango Intercuartílico
CPU	0.127 s	0.0240
GPU	0.0612 s	0.00160
Raspberry Pi 3 B+	1.76 s	0.0500
Raspberry Pi 3 B+ con NCS	0.246 s	0

Como se puede apreciar en el gráfico y en la tabla, los tiempos de respuesta y la mediana de la GPU son los más rápidos de todos. Los tiempos conseguidos con la CPU son los siguientes más rápidos, no habiendo una gran diferencia de tiempos entre los datos de estas dos plataformas como se aprecia en el diagrama de cajas.

Los tiempos de respuesta con la Raspberry Pi son de nuevo, con una gran diferencia, los más lentos de todos. Como se puede ver, su mediana es la mayor de las cuatro plataformas. Por otra parte, los tiempos de clasificación conseguidos con la Raspberry también son los más descentralizados respecto a la mediana, ya que el valor del rango intercuartílico es el más grande de todos.

En cambio, cuando el Neural Compute Stick es introducido en la Raspberry Pi se puede ver como los tiempos de respuesta y la mediana disminuyen considerablemente, llegándose a poner casi al mismo nivel que los tiempos de respuestas conseguidos con la CPU, tal y como se aprecia en el diagrama de cajas. Por otra parte, el rango intercuartílico también disminuye de forma absoluta, ya que su valor es cero.

Para terminar, si se tiene en cuenta la mediana, el uso del Neural Compute Stick en la Raspberry Pi para esta red proporciona una aceleración del rendimiento 7.15 veces mayor (615.45%) que sin él.

5.4 ResNet50

En esta sección se muestran los resultados obtenidos con ResNet50 en todas las distintas plataformas.

5.4.1 CPU

Los resultados al ejecutar esta red con la CPU del portátil son los siguientes:

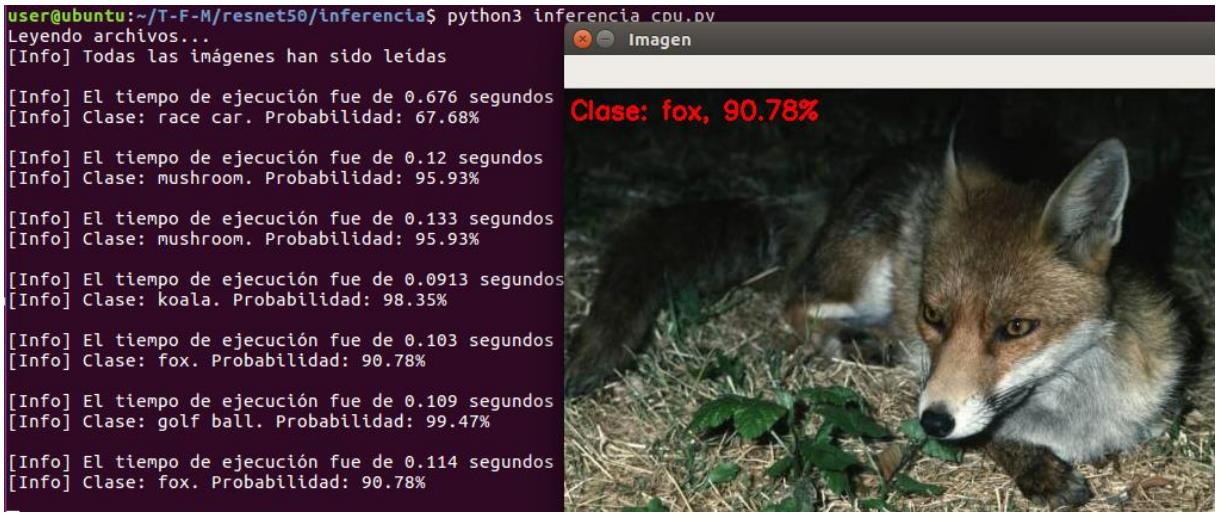

Figura 65: Resultados con la CPU de la red ResNet50

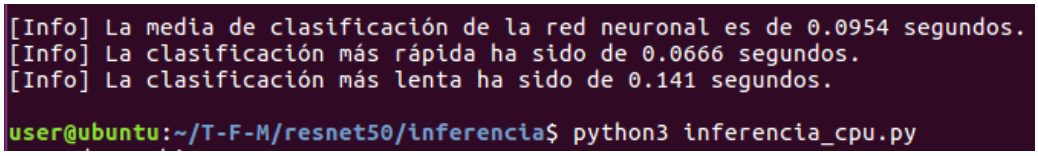

Figura 66: Tiempos de respuesta con la CPU de la red ResNet50

En la primera imagen se puede ver un fragmento de los tiempos de clasificación de las imágenes. Como se puede apreciar, los tiempos de clasificación están alrededor de la décima. Concretamente el tiempo de clasificación medio es de 0.0954 segundos, tal y como se puede ver en la segunda imagen.

El valor de la desviación estándar es nuevamente el segundo más alto de todas las plataformas con esta red, con un valor de 0.0199. Por tanto, los tiempos de clasificación en esta plataforma son los segundos más dispersos respecto al tiempo medio de clasificación si se pone el resultado en contexto con las demás plataformas.

Una vez hecho todos los cálculos se puede decir que el intervalo de confianza al 95% con la CPU es (0.0882;0.103).

5.4.2 GPU

Los resultados de ResNet50 al usar una GPU de Colab son estos:

```
Leyendo archivos...
[Info] Todas las imágenes han sido leídas

[Info] El tiempo de ejecución fue de 3.57 segundos
[Info] Clase: fox. Probabilidad: 90.78%

[Info] El tiempo de ejecución fue de 0.0799 segundos
[Info] Clase: fox. Probabilidad: 90.78%

[Info] El tiempo de ejecución fue de 0.079 segundos
[Info] Clase: fox. Probabilidad: 90.78%

[Info] El tiempo de ejecución fue de 0.074 segundos
[Info] Clase: fox. Probabilidad: 90.78%

[Info] El tiempo de ejecución fue de 0.0735 segundos
[Info] Clase: golf ball. Probabilidad: 99.47%

[Info] El tiempo de ejecución fue de 0.0669 segundos
[Info] Clase: golf ball. Probabilidad: 99.47%

[Info] El tiempo de ejecución fue de 0.0656 segundos
[Info] Clase: golf ball. Probabilidad: 99.47%

[Info] El tiempo de ejecución fue de 0.0648 segundos
[Info] Clase: golf ball. Probabilidad: 99.47%

[Info] El tiempo de ejecución fue de 0.0648 segundos
[Info] Clase: mushroom. Probabilidad: 95.93%

[Info] El tiempo de ejecución fue de 0.0643 segundos
[Info] Clase: mushroom. Probabilidad: 95.93%
```

Figura 67: Resultados con una GPU de Colab de la red ResNet50

```
[Info] La media de clasificación de la red neuronal es de 0.0662 segundos.
[Info] La clasificación más rápida ha sido de 0.064 segundos.
[Info] La clasificación más lenta ha sido de 0.0799 segundos.
```

Figura 68: Tiempos de respuesta con una GPU de Colab de la red ResNet50

En la primera imagen se ve un fragmento de los tiempos de clasificación de las imágenes con una GPU de Colab. Como se puede apreciar, los tiempos de clasificación están en torno a las seis centésimas, siendo de nuevo más rápidos que con la CPU. Concretamente, el tiempo de clasificación medio de la red ResNet50 es de 0.0662 segundos.

Con la GPU los tiempos de clasificación están nuevamente más centralizados respecto a la media que con la CPU. Esto es debido a que el valor de la desviación estándar con la GPU es menor que con la CPU (0.00442< 0.0199).

Una vez definida la media y la desviación estándar se puede afirmar que el intervalo de confianza al 95% es (0.0646;0.0678).

5.4.3 Raspberry Pi

Al usar la Raspberry Pi 3 B+, los resultados son los siguientes:

Figura 69: Resultados con la Raspberry Pi de la red ResNet50

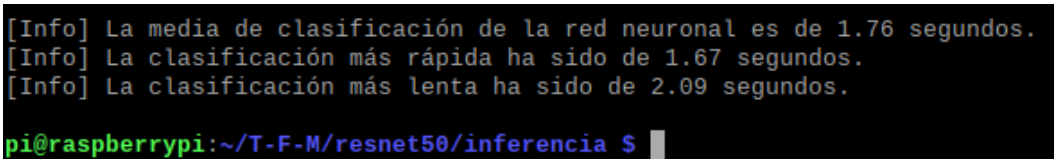

Figura 70: Tiempos de respuesta con la Raspberry Pi de la red ResNet50

En la primera imagen se puede ver un fragmento de los tiempos de clasificación de las imágenes con la Raspberry Pi. Como se puede apreciar, los tiempos de clasificación están alrededor de los 1.7-1.8 segundos, siendo de nuevo la plataforma donde más lentamente se ejecuta el modelo de red.

Concretamente, y tal y como se ve en la segunda imagen, el tiempo de clasificación medio de DenseNet169 es de 1.76 segundos, siendo un tiempo muy superior al de las demás plataformas.

Respecto a la desviación estándar, esta es de nuevo la mayor de las cuatro plataformas del proyecto (0.0720). De esta forma, los tiempos de clasificación están descentralizados respecto a la media, siendo los tiempos más dispersos respectos a las tres plataformas restantes.

Por último, destacar que con el uso de la Raspberry Pi el intervalo de confianza al 95% es (1.73;1.78).

5.4.4 Raspberry Pi con NCS

Cuando se usa el Neural Compute Stick en la Raspberry Pi, los resultados son los siguientes:

Figura 71: Resultados con la Raspberry Pi y el NCS de la red ResNet50

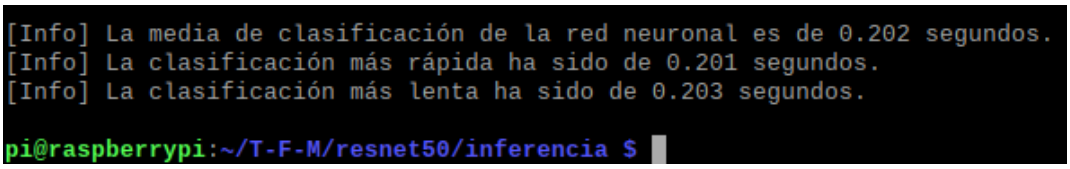

Figura 72: Tiempos de respuesta con la Raspberry Pi y el NCS de la red ResNet50

En la primera imagen se ve un fragmento de los tiempos de clasificación de las imágenes con la Raspberry Pi usando el Neural Compute Stick. Como se puede apreciar, los tiempos de clasificación están en torno a las dos centésimas.

Concretamente el tiempo de clasificación medio de la red DenseNet121 es de 0.202 segundos, tal y como se ve en la segunda imagen. Como se puede apreciar, esto mejora nuevamente de forma considerable la media respecto al caso anterior donde no se usaba el Neural Compute Stick en la Raspberry Pi.

Además, la desviación estándar con este modelo también disminuye. Tal es así que de hecho el valor es el menor de las cuatro plataformas (0.000491). Por lo tanto, los tiempos de clasificación en esta plataforma son los que están más agrupados cerca de la media y no se extienden sobre un rango de valores amplio. Esto se puede apreciar también viendo los tiempos máximos y mínimos, ya que solo se diferencian en dos milésimas.

Para terminar, destacar que con el uso del Neural Compute Stick el intervalo de confianza al 95% es (0.202;0.202).

5.4.5 Comparativa

Ahora se presenta de nuevo el diagrama de cajas y la tabla comparativa para comparar la mediana, el rango intercuartílico y calcular la aceleración provocada por el uso del Neural Compute Stick en la Raspberry Pi.

El diagrama de cajas es el siguiente:

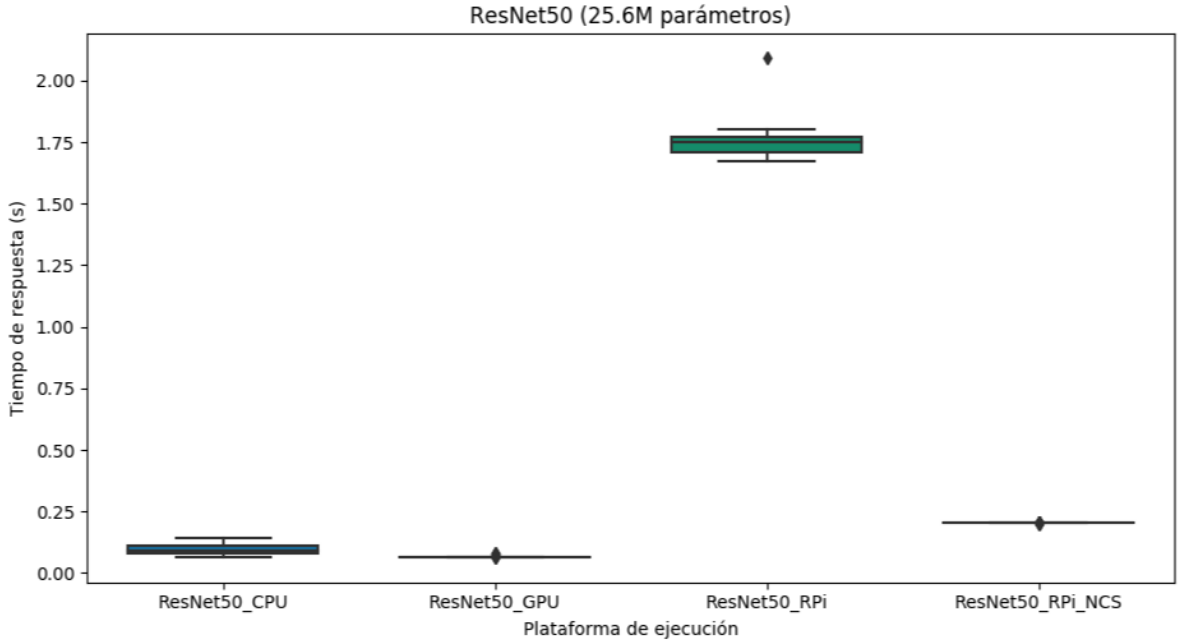

Figura 73: Diagrama de cajas de la red ResNet50

La tabla comparativa es la siguiente:

Tabla 6: Comparación de la red ResNet50 en las distintas plataformas

Plataforma	Mediana	Rango Intercuartílico
CPU	0.0912 s	0.0302
GPU	0.0644 s	0.0005
Raspberry Pi 3 B+	1.75 s	0.0600
Raspberry Pi 3 B+ con NCS	0.202 s	0

Como se puede apreciar en el gráfico y en la tabla, los tiempos de respuesta y la mediana de la GPU son los más rápidos de todos nuevamente. Los tiempos conseguidos con la CPU son los segundos más rápidos, no habiendo una gran diferencia de tiempos entre los datos de estas dos plataformas como se aprecia en el diagrama de cajas.

Los tiempos de respuesta con la Raspberry Pi son de nuevo, con una gran diferencia, los más lentos de todos. Como se puede ver, su mediana es la mayor de las cuatro plataformas. Por otra parte, los tiempos de clasificación conseguidos con la Raspberry también son los más descentralizados respecto a la mediana, ya que el valor del rango intercuartílico es el más grande de todos.

Con el Neural Compute Stick y la Raspberry Pi se puede ver como los tiempos de respuesta y la mediana disminuyen considerablemente, llegando a establecerse casi al mismo nivel que los tiempos de respuestas conseguidos con la CPU, tal y como se aprecia en el diagrama de cajas. Por otra parte, el rango intercuartílico también disminuye de forma absoluta, ya que su valor es cero.

Para terminar, si se tiene en cuenta la mediana, el uso del Neural Compute Stick en la Raspberry Pi para esta red proporciona una aceleración del rendimiento 8.66 veces mayor (766.34%) que sin él.

5.5 Discusión de resultados

Una vez hecho todos los experimentos se pueden extraer algunas ideas generales de todos ellos:

- El tiempo de clasificación medio más rápido es el conseguido con la GPU de Colab, seguido del tiempo conseguido con la CPU del portátil, pues son las dos plataformas más potentes del proyecto. Después irían los tiempos con la Raspberry Pi y el NCS y, por último, los tiempos conseguidos solamente con la Raspberry Pi.
- La dispersión de los tiempos de clasificación es prácticamente inexistente con la Raspberry Pi y el NCS. Esto probablemente sea debido a los núcleos de procesamiento SHAVE del NCS, los cuales implementan paralelismo a nivel de instrucción. Le sigue en este aspecto la GPU, muy de cerca. Los terceros tiempos menos dispersos son los conseguidos con la CPU del portátil (aunque ya con una gran diferencia con las dos plataformas anteriores) y, por último, se tienen los resultados conseguidos solamente con la Raspberry Pi, donde la dispersión es la mayor de todas.

Respecto a los resultados específicos conseguidos con el uso del Movidius Neural Compute Stick en la Raspberry Pi, se tiene lo siguiente:

- El tiempo de clasificación medio en la Raspberry Pi es más rápido con el Movidius Neural Compute Stick que sin él, llegando a alcanzar porcentajes de mejora de hasta el 700% y estableciendo estos tiempos casi a la altura de los conseguidos con la CPU del portátil.
- La dispersión de resultados es muy grande al usar al Raspberry Pi. Algo completamente distinto ocurre cuando se conecta el Movidius Neural Compute Stick, donde los resultados solo difieren en una o dos milésimas.
- El tiempo de respuesta con la Raspberry Pi y el NCS es de forma general más lento a medida que aumenta el número de parámetros de la red (al igual que en las demás plataformas), tal y como se puede ver en la siguiente imagen:

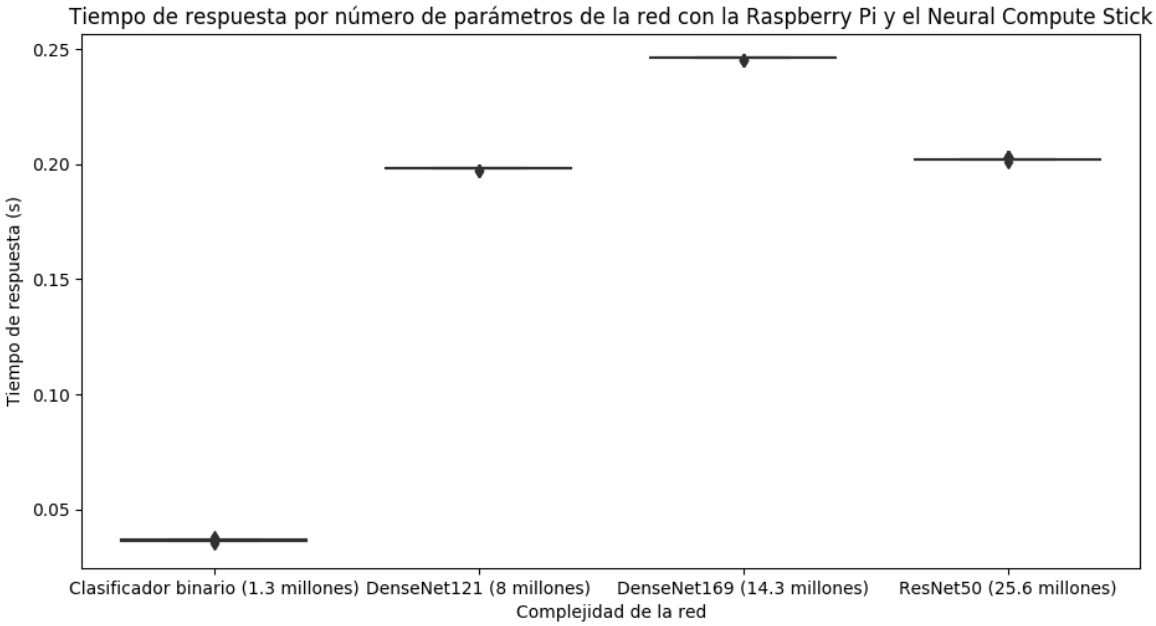

Figura 74: Tiempo de respuesta según el número de parámetros de la red

Como se puede ver en la imagen, se han ordenado por orden ascendente de parámetros las redes neuronales utilizadas en el proyecto. Se puede apreciar de forma general como el tiempo de respuesta es mayor cuánto más parámetros tiene la red. Sin embargo, hay una excepción: la red ResNet50 tiene una ejecución más rápida que la red DenseNet169. Esto probablemente sea debido a la propia estructura de las redes de la familia DenseNet, ya que conectan todas las capas directamente entre sí mediante los *feature maps* y, por lo tanto, presentan un flujo total de información entre capas, aumentando el coste computacional y el tiempo de inferencia, pero condensando toda la información en menos parámetros y en una menor profundidad de capas (50 > 169) [26].

6 CONCLUSIONES Y FUTUROS TRABAJOS

Una vez se realizan los experimentos pertinentes y se concluye el proyecto, es hora de realizar unas conclusiones acerca del trabajo realizado y de los resultados obtenidos. Además, y en base a esto último, también se especifica una serie de trabajos futuros que se podrían realizar para incrementar los resultados y las conclusiones obtenidas en este momento.

6.1 Conclusiones

Al inicio de este proyecto se declaró una serie de objetivos con el fin de utilizar y demostrar la viabilidad de dispositivos aceleradores de la fase de inferencia en sistemas embebidos que cuentan con pocos recursos como la Raspberry Pi.

El primero de ellos era comparar el funcionamiento en aplicaciones de clasificación de imágenes el rendimiento al ejecutar redes neuronales en la Raspberry Pi con el uso de un dispositivo acelerador de fases de inferencia con el rendimiento sin usar uno de ellos. En este sentido, los resultados han sido muy satisfactorios. Como ya se ha podido ver en el proyecto, se ha creado un clasificador binario de perros y gatos desde cero para poder realizar posteriormente esta comparativa. Así, el uso de un dispositivo acelerador de inteligencia artificial como el Neural Compute Stick en la Raspberry Pi 3 B+ del proyecto ha supuesto una mejora 5x del tiempo de respuesta del clasificador binario de perros y gatos. Además, el uso del Neural Compute Stick ha mejorado también de forma excelente la dispersión de los tiempos de clasificación entre imágenes distintas, disminuyéndola hasta hacerla prácticamente nula. Tal es así que la diferencia entre los tiempos de clasificación entre las imágenes es de apenas una o dos milésimas. Esto es culpa de los núcleos procesadores SHAVE de los que se compone el Movidius Neural Compute Stick, los cuales implementan simultaneidad a nivel de instrucción.

El segundo de los objetivos era saber la respuesta del Neural Compute Stick según la complejidad y el número de parámetros de la red neuronal. Para ello, además de la red que ha sido creada, se han usado otras tres redes distintas y ya entrenadas para clasificar imágenes, teniendo cada una de ellas una complejidad y un número de parámetros distinto. En este sentido, se puede decir que de forma general y acorde lo esperado, los tiempos de respuesta han sido más lentos a medida que se ha ido aumentando el número de parámetros de la red. Se trata de un comportamiento normal puesto que, al haber mayores parámetros, mayores cálculos son necesarios. No obstante, cabe destacar que, aunque una red tenga un mayor número de parámetros que otra, es posible que tenga un mejor tiempo de respuesta si tiene un coste computacional mucho menor y los cálculos no están condensados en un menor número de parámetros. Por otra parte, es de suponer que, ante nuevas redes no estudiadas en este proyecto, el NCS se comportaría de la misma forma, es decir, con unos tiempos más lentos acorde avanza el número de parámetros de la red. Eso sí, el comportamiento no sería totalmente lineal, pues como se ha podido comprobar, los tiempos de respuesta también dependen de la propia estructura y filosofía de las redes neuronales y del flujo de información que hay entre sus capas.

El tercer objetivo era poner los resultados del Neural Compute Stick en contexto con plataformas con más recursos de computación como el uso de una CPU o una GPU de un ordenador. En este sentido se puede decir que el uso del NCS ha mejorado de forma considerable los tiempos de ejecución en todas las redes usadas en el proyecto, estableciendo estos tiempos cerca de los conseguidos con la CPU (Intel i7 de 8º generación), que son los tiempos más rápidos de todas las plataformas justo después de los conseguidos con la GPU establecida en Google Colab. De esta forma, se consiguen aproximadamente unos tiempos de unas ciento cincuenta milésimas más lentas que con la CPU, por lo que son muy próximos a plataformas con mayores recursos de computación como un ordenador portátil.

En conclusión, este proyecto evidencia claramente la utilidad de los dispositivos aceleradores de inferencia en sistemas embebidos, ya que aceleran el rendimiento para clasificación de imágenes de 5-9 veces dependiendo de la red neuronal, consiguiendo unos tiempos prácticamente a la altura de sistemas con mayores recursos. Se trata, por tanto, de herramientas que permiten desplegar aplicaciones de *deep learning* en escenarios IoT con una mayor capacidad de cómputo.

6.2 Trabajo futuro

En esta sección se describen las posibles líneas abiertas que deja el presente proyecto y que podrían ser reanudadas en un futuro para seguir indagando sobre el tema en cuestión:

- Comparación con otras tecnologías de la fase de inferencia. Como se ha visto en el capítulo dos, hay otras soluciones aparte de la de Intel para acelerar la fase de inferencia. En este sentido sería interesante comparar los resultados obtenidos con el Neural Compute Stick con el acelerador USB de Google Coral. También se podría comparar el resultado con la placa de desarrollo de Google Coral o alguna de la familia Jetson de NVIDIA, específicas para acelerar tareas de *machine learning*.
- Realizar el mismo proyecto con el hardware más moderno posible. En este sentido, sería interesante realizar el proyecto con la nueva Raspberry Pi 4, un modelo con un hardware más potente que el usado en el proyecto y que cuenta además con puertos USB 3.0, mejorando la velocidad de conexión con el Movidius Neural Compute Stick. Por otra parte, también sería interesante realizar el proyecto con el Neural Compute Stick 2, ya que se trata de un dispositivo más potente que su predecesor, por lo que los tiempos de respuesta también podrían ser incluso mejores.
- Realizar el mismo proyecto con aplicaciones para detección de objetos. En este sentido, se podría comparar y analizar el rendimiento con el uso del Movidius Neural Compute Stick en la Raspberry Pi. De esta forma, se podría estudiar como mejora la cantidad de fotogramas por segundo (FPS) en vídeos donde se realicen aplicaciones de detección de objetos.

El mundo del *deep learning* y las aplicaciones IoT está continuamente en desarrollo y avanza muy deprisa, por lo que no es de extrañar que con el tiempo puedan aparecer otras líneas abiertas que deja el proyecto realizado además de las ya comentadas.

De todas formas, los resultados mostrados en el presente proyecto sirven como punto de partida para enmarcar y establecer futuras mejoras y comparaciones con nuevos sistemas y dispositivos.

ANEXOS

Anexo A: Tablas de resultados

En este anexo se recopilan las tablas de todos los resultados obtenidos, los cuales han sido usados para apoyar la información estadística usada en la memoria.

Tabla 7: Resultados del clasificador binario

	CPU	GPU	Raspberry Pi	Raspberry Pi + NCS
1.º resultado	0,0207	0,00595	0,148	2,6200
2.º resultado	0,0134	0,00181	0,193	0,0372
3.º resultado	0,0083	0,00177	0,154	0,0366
4.º resultado	0,0079	0,00180	0,267	0,0365
5.º resultado	0,0076	0,00179	0,125	0,0366
6.º resultado	0,0075	0,00180	0,181	0,0366
7.º resultado	0,0077	0,00180	0,179	0,0367
8.º resultado	0,0077	0,00178	0,189	0,0367
9.º resultado	0,0147	0,00180	0,170	0,0367
10.º resultado	0,0078	0,00184	0,183	0,0368
11.º resultado	0,0078	0,00180	0,182	0,0367
12.º resultado	0,0076	0,00180	0,182	0,0368
13.º resultado	0,0077	0,00179	0,181	0,0366
14.º resultado	0,0139	0,00180	0,166	0,0367
15.º resultado	0,0077	0,00182	0,164	0,0366
16.º resultado	0,0078	0,00180	0,188	0,0366
17.º resultado	0,0133	0,00180	0,168	0,0368
18.º resultado	0,0079	0,00179	0,169	0,0366
19.º resultado	0,0076	0,00179	0,151	0,0378

20.º resultado	0,0076	0,00179	0,158	0,0367
21.º resultado	0,0078	0,00179	0,177	0,0366
22.º resultado	0,0078	0,00184	0,191	0,0366
23.º resultado	0,0075	0,00180	0,130	0,0364
24.º resultado	0,0082	0,00178	0,147	0,0366
25.º resultado	0,0079	0,00186	0,184	0,0363
26.º resultado	0,0142	0,00181	0,173	0,0364
27.º resultado	0,0082	0,00179	0,201	0,0367
28.º resultado	0,0120	0,00178	0,170	0,0365
29.º resultado	0,0112	0,00181	0,168	0,0353
30.º resultado	0,0117	0,00177	0,175	0,0359
Media	0,00924	0,00180	0,175	0,0366
Mediana	0,00782	0,00180	0,175	0,0366
Desv. Estándar	0,00249	0,00002	0,0249	0,000395
Interv. Confianza 95%	0,000904	0,00000728	0,00908	0,000144
Límite inferior	0,00833	0,00179	0,166	0,0365
Límite superior	0,0101	0,00181	0,184	0,0368
1.º cuartil	0,00768	0,00179	0,166	0,0366
3.º cuartil	0,0112	0,00180	0,183	0,0367
Rango intercuartílico	0,00352	0,0000100	0,0170	0,000100

Tabla 8: Resultados de la red DenseNet121

	CPU	GPU	Raspberry Pi	Raspberry Pi + NCS
1.º resultado	0,3210	0,113	1,80	3,780
2.º resultado	0,1140	0,0374	1,48	0,198
3.º resultado	0,1150	0,0374	1,41	0,198
4.º resultado	0,0987	0,0373	1,47	0,197
5.º resultado	0,1000	0,0220	1,40	0,198
6.º resultado	0,0962	0,0216	1,43	0,198
7.º resultado	0,0804	0,0217	1,40	0,197
8.º resultado	0,0681	0,0216	1,47	0,198
9.º resultado	0,0666	0,0213	1,47	0,198
10.º resultado	0,0998	0,0210	1,39	0,197
11.º resultado	0,0782	0,0210	1,41	0,198
12.º resultado	0,0694	0,0210	1,50	0,198
13.º resultado	0,0668	0,0206	1,45	0,198
14.º resultado	0,0772	0,0185	1,44	0,198
15.º resultado	0,0693	0,0181	1,48	0,197
16.º resultado	0,0661	0,0179	1,46	0,198
17.º resultado	0,0656	0,0180	1,44	0,198
18.º resultado	0,0664	0,0180	1,40	0,198
19.º resultado	0,0675	0,0181	1,42	0,198
20.º resultado	0,0664	0,0182	1,47	0,198
21.º resultado	0,1040	0,0185	1,42	0,198
22.º resultado	0,1040	0,0186	1,41	0,198
23.º resultado	0,0983	0,0188	1,99	0,198
24.º resultado	0,0662	0,0187	1,41	0,198
25.º resultado	0,0661	0,0188	1,47	0,198

26.º resultado	0,0666	0,0188	1,45	0,198
27.º resultado	0,0689	0,0186	1,44	0,198
28.º resultado	0,0680	0,0185	1,43	0,198
29.º resultado	0,0953	0,0186	1,41	0,198
30.º resultado	0,0688	0,0185	1,39	0,198
Media	0,0806	0,0213	1,46	0,198
Mediana	0,0693	0,0188	1,44	0,198
Desv. Estándar	0,0170	0,00573	0,107	0,000351
Interv. Confianza 95%	0,00619	0,00208	0,0391	0,000128
Límite inferior	0,0744	0,0192	1,42	0,198
Límite superior	0,0868	0,0234	1,49	0,198
1.º cuartil	0,0666	0,0185	1,41	0,198
3.º cuartil	0,0983	0,0213	1,47	0,198
Rango intercuartílico	0,0317	0,00280	0,0600	0,0000

Tabla 9: Resultados de la red DenseNet169

	CPU	GPU	Raspberry Pi	Raspberry Pi + NCS
1.º resultado	0,4140	9,69	2,50	4,400
2.º resultado	0,1300	0,0767	1,73	0,245
3.º resultado	0,1500	0,0767	1,73	0,245
4.º resultado	0,1630	0,0719	1,77	0,246
5.º resultado	0,0942	0,0700	1,79	0,245
6.º resultado	0,1350	0,0677	1,74	0,245
7.º resultado	0,1240	0,0648	1,78	0,246
8.º resultado	0,1080	0,0644	1,74	0,246
9.º resultado	0,1240	0,0616	1,76	0,246
10.º resultado	0,1620	0,0613	1,71	0,246
11.º resultado	0,1050	0,0615	1,79	0,246
12.º resultado	0,1210	0,0608	1,79	0,246
13.º resultado	0,1390	0,0611	1,71	0,246
14.º resultado	0,1150	0,0608	1,81	0,246
15.º resultado	0,1380	0,0607	1,70	0,245
16.º resultado	0,1280	0,0611	1,71	0,246
17.º resultado	0,1570	0,0606	1,72	0,246
18.º resultado	0,1170	0,0607	1,70	0,246
19.º resultado	0,1160	0,0624	1,78	0,246
20.º resultado	0,1060	0,0613	1,77	0,246
21.º resultado	0,1190	0,0611	1,75	0,245
22.º resultado	0,1680	0,0622	1,76	0,246
23.º resultado	0,1660	0,0612	1,83	0,246
24.º resultado	0,1290	0,0610	1,76	0,246
25.º resultado	0,1200	0,0611	1,71	0,246

26.º resultado	0,1160	0,0608	1,73	0,246
27.º resultado	0,1480	0,0623	1,78	0,246
28.º resultado	0,0890	0,0608	1,75	0,246
29.º resultado	0,1400	0,0607	1,79	0,246
30.º resultado	0,1270	0,0608	1,76	0,246
Media	0,129	0,0634	1,75	0,246
Mediana	0,127	0,0612	1,76	0,246
Desv. Estándar	0,0210	0,00465	0,0347	0,000412
Interv. Confianza 95%	0,00766	0,00169	0,0126	0,000150
Límite inferior	0,122	0,0617	1,74	0,246
Límite superior	0,137	0,0651	1,77	0,246
1.º cuartil	0,116	0,0608	1,73	0,246
3.º cuartil	0,140	0,0624	1,78	0,246
Rango intercuartílico	0,0240	0,00160	0,0500	0,000

Tabla 10: Resultados de la red ResNet50

	CPU	GPU	Raspberry Pi	Raspberry Pi + NCS
1.º resultado	0,6760	3,57	4,77	6,960
2.º resultado	0,1200	0,0799	1,77	0,202
3.º resultado	0,1330	0,0790	1,75	0,202
4.º resultado	0,0913	0,0740	1,76	0,202
5.º resultado	0,1030	0,0735	1,75	0,202
6.º resultado	0,1090	0,0669	1,71	0,202
7.º resultado	0,1140	0,0656	1,67	0,202
8.º resultado	0,1410	0,0648	1,78	0,202
9.º resultado	0,0666	0,0648	1,71	0,202
10.º resultado	0,0912	0,0643	1,72	0,202
11.º resultado	0,0788	0,0641	1,76	0,202
12.º resultado	0,1170	0,0643	1,71	0,203
13.º resultado	0,1070	0,0640	1,72	0,202
14.º resultado	0,0888	0,0643	1,76	0,203
15.º resultado	0,0819	0,0644	1,78	0,202
16.º resultado	0,1000	0,0645	1,74	0,203
17.º resultado	0,1170	0,0644	1,71	0,202
18.º resultado	0,1040	0,0644	2,09	0,203
19.º resultado	0,0668	0,0644	1,77	0,202
20.º resultado	0,0777	0,0644	1,74	0,203
21.º resultado	0,0790	0,0641	1,71	0,202
22.º resultado	0,0786	0,0642	1,80	0,203
23.º resultado	0,0722	0,0642	1,71	0,202
24.º resultado	0,0713	0,0642	1,79	0,202
25.º resultado	0,1150	0,0642	1,71	0,201

26.º resultado	0,0904	0,0646	1,73	0,202
27.º resultado	0,1060	0,0644	1,77	0,202
28.º resultado	0,0879	0,0643	1,76	0,203
29.º resultado	0,0779	0,0643	1,73	0,202
30.º resultado	0,0818	0,0645	1,80	0,202
Media	0,0955	0,0662	1,76	0,202
Mediana	0,0912	0,0644	1,75	0,202
Desv. Estándar	0,0199	0,00442	0,0720	0,000491
Interv. Confianza 95%	0,00724	0,00161	0,0262	0,000179
Límite inferior	0,0882	0,0646	1,73	0,202
Límite superior	0,103	0,0678	1,78	0,202
1.º cuartil	0,0788	0,0643	1,71	0,202
3.º cuartil	0,109	0,0648	1,77	0,202
Rango intercuartílico	0,0302	0,000500	0,0600	0,000

Anexo B: Códigos utilizados

En este anexo se muestran los códigos utilizados en Python durante la realización del proyecto.

Entrenamiento del clasificador binario:

```
  %tensorflow_version 2.x
  import tensorflow as tf
  device_name = tf.test.gpu_device_name()
  if device_name != '/device:GPU:0':
    raise SystemError('GPU device not found')
  print('Encontrada GPU en: {}'.format(device_name))

  #!/usr/bin/env python
  # coding: utf-8
  # Se importan librerías
  import cv2 # OpenCV
  import tensorflow as tf
  import numpy as np # Arrays y matrices
  import matplotlib.pyplot as plt # Gráficas
  import os # Sistema de archivos
  from sklearn.model_selection import train_test_split
  # Entreno-Validación
  import random # Números aleatorios
  import gc # Limpiar y borrar variables

  # Directorios de entrenamiento y testeo:
  entreno_dir = '/content/drive/My Drive/train/'

  # Se extraen imágenes de entreno de perros y gatos por separado:
  entreno_perros = ['/content/drive/My
Drive/train/{}'.format(i) for i in os.listdir(entreno_dir) if 'dog'
in i]
  entreno_gatos = ['/content/drive/My
Drive/train/{}'.format(i) for i in os.listdir(entreno_dir) if 'cat'
in i]
  # os.listdir coge todas las imágenes

  # Se corta el dataset de entreno y se cogen 2000 imágenes de cada
clase:
  entreno_imagenes = entreno_perros[:2000] + entreno_gatos[:2000]
  # Se barajan estas 4000 imágenes de forma aleatoria:
  random.shuffle(entreno_imagenes)

  # Se borran variables que ya no hacen falta para ahorrar memoria:
  del entreno_perros
  del entreno_gatos
  gc.collect()

  # Todas las imágenes a la misma dimensión:
  alto = 180
  ancho = 180
  canales = 3 # Imágenes a color (RGB) = 3
```

```
  # Se redimensiona las imágenes y se crea las etiquetas:
  x = [] # Imágenes redimensionadas
  y = [] # Etiquetas
  for imagen in entreno_imagenes: # Bucle for
      # Leer y redimensionar imagen:
       x.append(cv2.resize(cv2.imread(imagen,cv2.IMREAD_COLOR),(alto,
ancho),interpolation=cv2.INTER_CUBIC))
      # Obtener las etiquetas:
      if 'dog' in imagen:
          y.append(1) # No valen strings
      elif 'cat' in imagen:
          y.append(0)
  # 'x' es ahora un array de valores de pixeles de cada imagen.
  # 'y' es una lista de dos etiquetas (0 o 1)

  # Convertir a arrays numpy para manipular los datos:
  x = np.array(x)
  y = np.array(y)

  # Repartir los datos entre entrenamiento y validación
  # 80% entreno, 20% validación:
  x_entreno,x_valid,y_entreno,y_valid = train_test_split(x,y,
test_size=0.20,random_state=2)

  # Se borran variables ya inservibles:
  del entreno_imagenes
  del x
  del y
  gc.collect()

  # Construir la red neuronal. Paquetes importados:
  from keras import layers # Capas
  from keras import models # Modelo secuencial

  """Entrada de 180x180x3, como las imágenes redimensionadas. Hay 3
capas de convolución (ventanas 5x5) y 3 capas de max pooling (ventanas
3x3), una capa para aplanar el tensor a 1D y poder utilizar 2 capas
densamente conectadas. Capa final con activación sigmoid al tratar
solamente con dos clases: """
  modelo = models.Sequential()
  modelo.add(layers.Conv2D(32,(5,5),activation='relu',
input_shape=(alto,ancho,canales)))
  modelo.add(layers.MaxPooling2D((3,3)))
  modelo.add(layers.Conv2D(64,(5,5),activation='relu'))
  modelo.add(layers.MaxPooling2D((3,3)))
  modelo.add(layers.Conv2D(128,(5,5),activation='relu'))
  modelo.add(layers.MaxPooling2D((3,3)))
  modelo.add(layers.Flatten()),
  modelo.add(layers.Dense(512,activation='relu')),
  modelo.add(layers.Dense(1,activation='sigmoid'))
  modelo.summary()
```

```
  from keras import optimizers # Optimizadores

  # Solo hay dos clases, por eso se utiliza esta función de coste
  # Se utiliza un learning rate pequeño y la precisión como métrica:
  modelo.compile(loss='binary_crossentropy',
                 optimizer=optimizers.RMSprop(lr = 0.0001),
                 metrics=['acc'])

  # Genera tensores procesados a partir de archivos de imágenes
  #  Escala los pixeles de imágenes entre (0,1) y hace data
augmentation:
  from keras.preprocessing.image import ImageDataGenerator

  """ Data augmentation. Previene el sobreajuste aumentando el dataset
haciendo pequeñas variaciones en las imágenes de entreno (no utiliza
las originales) como rotaciones de ángulos, traslaciones, cortes,
zooms, etc. Aparte se escalan los píxeles entre 0 y 1: """
  entreno_pixel = ImageDataGenerator(rescale=1.0/255,
rotation_range=40,width_shift_range=0.2,height_shift_range=0.2,
shear_range=0.2,zoom_range=0.2,horizontal_flip=True)

  # Solo escalado de píxeles. Sin data augmentation:
  valid_pixel = ImageDataGenerator(rescale=1.0/255)

  # Tamaño de un lotes. Normalmente 32 (producto de 2's):
  tamano_lote = 32

  """Se instancian generadores de lotes de las imágenes modificadas
de los datasets con los pixeles (0-1) y el data augmentation con el
método flow: """
  entreno_generador = entreno_pixel.flow(x_entreno,y_entreno,
batch_size = tamano_lote)
  valid_generador = valid_pixel.flow(x_valid,y_valid,
batch_size = tamano_lote)

  # Longitud de las imágenes de entreno y validación:
  longitud_entreno = len(x_entreno)
  longitud_valid = len(x_valid)
```

```
  # Entreno de la red neuronal:
  # Primer parámetro: set de entrenamiento de ImageDataGenerator.
  # Segundo parámetro: lotes por epochs = hay 3200 imágenes de
entrenamiento, y el tamaño de un lote es 32, luego los lotes son 100
(100x32 = 3200).
  # Tercer parámetro: número de epochs.
  # Cuarto parámetro: datos de validación.
  # Quinto parámetro: pasos (25 pasos)
  entrenamiento = modelo.fit_generator(entreno_generador,
steps_per_epoch = longitud_entreno // tamano_lote,
epochs = 150,
validation_data = valid_generador,
validation_steps = longitud_valid // tamano_lote,
verbose = 2)

  # Gráfica para comprobar si hay o no sobreajuste:
  # Las métricas del entreno se guardan dentro del método 'history'
  acc = entrenamiento.history['acc']
  val_acc = entrenamiento.history['val_acc']
  loss = entrenamiento.history['loss']
  val_loss = entrenamiento.history['val_loss']
  epochs = range(1,len(acc)+1,1)

  # Accuracy del entreno y de la validación
  plt.plot(epochs,acc,'r',label='Accuracy del entreno')
  plt.plot(epochs, val_acc,'b--',label='Accuracy de la validación')
  plt.title('Accuracy del entreno y de la validación')
  plt.ylabel('Accuracy')
  plt.xlabel('Epochs')
  plt.legend()
  plt.figure()

  # Loss del entreno y de la validación
  plt.plot(epochs,loss,'r',label='Loss del entreno')
  plt.plot(epochs,val_loss,'b--',label='Loss de la validación')
  plt.title('Loss del entreno y de la validación')
  plt.ylabel('Loss')
  plt.xlabel('Epochs')
  plt.legend()

  plt.show() # Se muestran las gráficas

  modelo.save('modelo.h5') # Se guarda el modelo
```

Conversión de archivo .h5 de Keras a archivo .pb de TensorFlow:

```
  # Se importan los paquetes necesarios:
  from keras import backend as K
  from keras.models import load_model

  # Se pone la fase de aprendizaje a cero para quitar el modo de
entrenamiento y se carga el modelo completo:
  K.set_learning_phase(0)
  modelo = load_model('/home/user/T-F-
M/Valderas/modelos/modelo.h5', compile = False)

  # Se importan los paquetes necesarios:
  from keras import backend as K
  import tensorflow as tf

  # Se define una función para conseguir un grafo computacional:
  def freeze_session(session, keep_var_names=None,
output_names=None, clear_devices=True):
      """ Congela el estado de una sesión y crea un nuevo modelo donde
los nodos variables se reemplazan por constantes con su valor actual
en la sesión. La información no necesaria del modelo será eliminada.
      @param session: La sesión de TensorFlow que será exportada.
      @param keep_var_names: Lista de nombres de variables que no serán
exportadas.
      @param output_names: Nombres de las salidas del modelo.
      @param clear_devices: Elimina directrices del modelo para una
mejor portabilidad.
      @return: La definición del grafo exportado."""

      from tensorflow.python.framework.graph_util import
convert_variables_to_constants
      grafo = session.graph
      with grafo.as_default():
          freeze_var_names = list(set(v.op.name for v in
tf.global_variables()).difference(keep_var_names or []))
          output_names = output_names or []
          output_names += [v.op.name for v in tf.global_variables()]
          input_graph_def = grafo.as_graph_def()
          if clear_devices:
              for node in input_graph_def.node:
                  node.device = ""
          grafo_exportado = convert_variables_to_constants(session,
input_graph_def,output_names, freeze_var_names)
          return grafo_exportado

  # Se congela la sesión y se elige la salida del modelo creado,
eliminando información innecesaria:
  grafo_exportado = freeze_session(K.get_session(),
output_names=[out.op.name for out in modelo.outputs])

  # Se escribe el modelo de salida '.pb', guardándolo en el sistema
de archivos del PC:
  tf.train.write_graph(grafo_exportado, "modelo", "modelo.pb",
as_text=False)
```

Conversión de archivo .pb a archivo .pbtxt de TensorFlow:

```
  import tensorflow as tf

  # Lee el modelo binario de TF:
  with tf.gfile.FastGFile('/home/user/T-F-
M/Valderas/modelos/modelo.pb', 'rb') as f:
      graph_def = tf.GraphDef()
      graph_def.ParseFromString(f.read())

  # Elimina nodos 'Const'.
  for i in reversed(range(len(graph_def.node))):
      if graph_def.node[i].op == 'Const':
          del graph_def.node[i]
      for attr in ['T', 'data_format', 'Tshape', 'N', 'Tidx',
'Tdim','use_cudnn_on_gpu', 'Index', 'Tperm', 'is_training',
'Tpaddings']:
          if attr in graph_def.node[i].attr:
              del graph_def.node[i].attr[attr]

  # Guarda como archivo de texto en el PC:
  tf.train.write_graph(graph_def, "modelo", "modelo.pbtxt",
as_text=True)
```

Código de inferencia del clasificador binario:

```
  #!/usr/bin/env python
  # coding: utf-8
  # Se importan librerías:
  import os # Listado archivos
  import cv2 # OpenCV
  import numpy as np # Manipular arrays
  import time # Medir el tiempo de ejecución

  # Se carga el modelo:
  red = cv2.dnn.readNetFromTensorflow('/home/user/T-F-
M/Valderas/modelos/modelo.pb', '/home/user/T-F-
M/Valderas/modelos/modelo.pbtxt')

  # Se especifica el dispositivo objetivo (CPU):
  red.setPreferableBackend(cv2.dnn.DNN_BACKEND_OPENCV)
  red.setPreferableTarget(cv2.dnn.DNN_TARGET_CPU)

  # Se especifica el directorio principal y se crea un array para
recopilar las imágenes y los tiempos:
  directorio = "/home/user/T-F-M/Valderas/test"
  imagenes = []
  tiempos = []

  # Se leen todas las imágenes y se van añadiendo a la lista:
  for dirPath, dirNames, fileNames in os.walk(directorio):
  # Genera los nombres de los archivos
      print('Leyendo archivos...')
      for f in fileNames:
          img = cv2.imread(os.path.join(dirPath, f))
  # Se lee una imagen dentro del directorio establecido
          img = img.astype('float32') / 255
  # Transformar imagen a formato float32 y se escalan los píxeles con
valores entre (0-1)
          imagenes.append(img) # La imagen se añade al array
  print('[Info] Todas las imágenes han sido leídas\n')

  # Para todas las imágenes del array:
  for i in range(len(imagenes)):
      # Se prepara el objeto binario grande (la imagen) como entrada
de la red neuronal:
      blob = cv2.dnn.blobFromImage(imagenes[i], size=(180, 180),
ddepth=cv2.CV_32F) # Se crea BLOB desde la imagen
      red.setInput(blob)
  # Se establece el BLOB creado como entrada de la red neuronal
```

```
      # Se obtiene la clasificación de salida y se mide el tiempo de
ejecución:
      inicio = time.time() # Inicio del contador para cada BLOB
      out = red.forward() # Se calcula la salida de la red neuronal
      fin = time.time() # Final del contador para cada BLOB
        print("[Info]  El  tiempo  de  ejecución  fue  de  {:.3}
segundos".format(fin - inicio)) # 3 cifras significativas
      if i>0: # Dentro del array se excluye la 1ra clasificación)
        tiempos.append(fin - inicio)

      # Se imprime por pantalla salida y la probabilidad. La salida
de la red da un valor dentro del intervalo (0-1).
      if out[0] > 0.5: # Cuanto más cerca del '1', más probable que
la imagen sea de un perro.
            print('La  imagen  es  de  un:  Perro  (Probabilidad:
{:.4}%)\n'.format(out[0][np.argmax(out)]*100))
                              texto = "Perro            (Probabilidad:
{:.4}%)".format(out[0][np.argmax(out)]*100)
  # Mostrar imagen en pantalla
        cv2.putText(imagenes[i], texto, (5, 25),
cv2.FONT_HERSHEY_SIMPLEX, 0.55, (0, 0, 255), 2)
        cv2.imshow("Imagen", imagenes[i])
        cv2.waitKey(0)
      else: # Cuanto más cerca del '0', más probable que la imagen sea
de un gato.
            print('La  imagen  es  de  un:  Gato  (Probabilidad:
{:.4}%)\n'.format(100-out[0][np.argmax(out)]*100))
                texto = "Gato   (Probabilidad:   {:.4}%)".format(100-
out[0][np.argmax(out)]*100) # Mostrar imagen en pantalla
        cv2.putText(imagenes[i], texto, (5, 25),
cv2.FONT_HERSHEY_SIMPLEX, 0.55, (0, 0, 255), 2)
        cv2.imshow("Imagen", imagenes[i])
        cv2.waitKey(0)
  cv2.destroyAllWindows()

  # Se crean variables para crear las estadísticas finales:
  media = np.mean(tiempos) # Media
  minimo = np.min(tiempos) # Tiempo más rápido
  maximo = np.max(tiempos) # Tiempo más lento

  # Se imprime por pantalla la media, y la clasificación más rápida y
más lenta:
  print('[Info] La media de clasificación de la red neuronal es de
{:.3} segundos.'.format(media))
  print('[Info]  La  clasificación  más  rápida  ha  sido  de  {:.3}
segundos.'.format(minimo))
  print('[Info]  La  clasificación  más  lenta  ha  sido  de  {:.3}
segundos.\n'.format(maximo))
```

Este es el código de inferencia del clasificador binario para usar con la CPU del portátil. Para realizar la inferencia en las otras plataformas, basta con cambiar el *backend* y el *target* y los formatos de archivos a cargar según corresponda, tal y como se mencionó en el capítulo cinco de la memoria.

Código de inferencia de DenseNet121:

```
#!/usr/bin/env python
# coding: utf-8
# cv2.imshow está desactivado en Colab
# Se importan librerías:
import os # Listado archivos
import cv2 # OpenCV
import numpy as np # Manipular arrays
import time # Medir el tiempo de ejecución

# Se carga el documento de clases de Imagenet:
filas = open('/content/drive/My                    Drive/T-F-
M/densenet121/modelos/labels.txt').read().strip().split("\n")
# Divide en filas el documento
clases = [r[r.find(" ") + 1:].split(",")[0] for r in filas]
# Recoge las clases por cada fila

# Se especifica el directorio principal y se crea un array para
recopilar las imágenes:
directorio = "/content/drive/My Drive/T-F-M/densenet121/test"
imagenes = []
tiempos = []

# Se carga la red preentrenada:
red = cv2.dnn.readNetFromCaffe('/content/drive/My        Drive/T-F-
M/densenet121/modelos/densenet-121.prototxt', '/content/drive/My
Drive/T-F-M/densenet121/modelos/densenet-121.caffemodel')

# Se especifica el dispositivo objetivo (GPU):
red.setPreferableBackend(cv2.dnn.DNN_BACKEND_OPENCV)
red.setPreferableTarget(cv2.dnn.DNN_TARGET_OPENCL)

# Se leen todas las imágenes y se van añadiendo al array:
for dirPath, dirNames, fileNames in os.walk(directorio):
# Genera los nombres de los archivos
    print('Leyendo archivos...')
    for f in fileNames:
        imagen = cv2.imread(os.path.join(dirPath, f))
# Se lee una imagen dentro del directorio establecido
        imagenes.append(imagen)
# La imagen se añade al array
print('[Info] Todas las imágenes han sido leídas\n')

# La red requiere dimensiones fijadas para la imagen de entrada.
#  Se realiza "mean subtraction" (103.94,116.78,123.68) para
normalizar las imágenes de entrada
# Después de esto el BLOB de entrada tiene forma (1, 3, dim, dim)
donde la dimensión tiene que ser de 224.

# Para todas las imágenes del array:
for j in range(len(imagenes)):
    # Se especifica el BLOB como entrada de la red y se escala los
valores BGR multiplicando por 0.017:
    blob = cv2.dnn.blobFromImage(imagenes[j], 1, (224, 224),
(103.94,116.78,123.68))
```

```
      blob = blob * 0.017
      red.setInput(blob)

      # Se obtiene la clasificación de salida y se mide el tiempo de
ejecución:
      inicio = time.time()
      resultado = red.forward()
      final = time.time()
        print("[Info]  El  tiempo  de  ejecución  fue  de  {:.3}
segundos".format(final - inicio))
      if j>0: # Dentro del array se excluye la 1ra clasificación
        tiempos.append(final - inicio)

      # Top1 predicción:
      resultado = resultado.reshape((1, 1000))
      indice = np.argsort(resultado[0])[::-1][:1]

      # Bucle sobre las predicciones:
      for (i, indice) in enumerate(indice):
          # Dibuja la predicción principal en la imagen:
          if i == 0:
              # Muestra la clase y la probabilidad en la consola:
                      print("[Info]  Clase:  {}.  Probabilidad:
{:.4}%\n".format(clases[indice+1], 100-(resultado[0][indice])/10))

  # Se crean variables para crear las estadísticas finales:
  media = np.mean(tiempos) # Media
  minimo = np.min(tiempos) # Tiempo más rápido
  maximo = np.max(tiempos) # Tiempo más lento

  # Se imprime por pantalla la media, y la clasificación más rápida y
más lenta:
  print('[Info] La media de clasificación de la red neuronal es de
{:.3} segundos.'.format(media))
  print('[Info]  La  clasificación  más  rápida  ha  sido  de  {:.3}
segundos.'.format(minimo))
  print('[Info]  La  clasificación  más  lenta  ha  sido  de  {:.3}
segundos.\n'.format(maximo))
```

Este es el código de inferencia de la red DenseNet121 para usar con la GPU proporcionada por Colab. Para realizar la inferencia en las otras plataformas, basta con cambiar el *backend* y el *target* y los formatos de archivos a cargar según corresponda, tal y como se mencionó en el capítulo cinco de la memoria.

Código de inferencia de DenseNet169:

```
#!/usr/bin/env python
# coding: utf-8
# Se importan librerías:
import os # Listado archivos
import cv2 # OpenCV
import numpy as np # Manipular arrays
import time # Medir el tiempo de ejecución

# Se carga el documento de clases de Imagenet:
filas = open('/home/pi/T-F-
M/densenet169/modelos/labels.txt').read().strip().split("\n")
# Divide en filas el documento
clases = [r[r.find(" ") + 1:].split(",")[0] for r in filas]
# Recoge las clases por cada fila

# Se especifica el directorio principal y se crea un array para
recopilar las imágenes:
directorio = "/home/pi/T-F-M/densenet169/test"
imagenes = []
tiempos = []

# Se carga la red preentrenada:
red = cv2.dnn.readNetFromCaffe('/home/pi/T-F-
M/densenet169/modelos/densenet-169.prototxt', '/home/pi/T-F-
M/densenet169/modelos/densenet-169.caffemodel')

# Se especifica el dispositivo objetivo (RPi):
red.setPreferableBackend(cv2.dnn.DNN_BACKEND_OPENCV)
red.setPreferableTarget(cv2.dnn.DNN_TARGET_CPU)

# Se leen todas las imágenes y se van añadiendo al array:
for dirPath, dirNames, fileNames in os.walk(directorio):
# Genera los nombres de los archivos
    print('Leyendo archivos...')
    for f in fileNames:
        imagen = cv2.imread(os.path.join(dirPath, f))
# Se lee una imagen dentro del directorio establecido
        imagenes.append(imagen)
# La imagen se añade al array
print('[Info] Todas las imágenes han sido leídas\n')

# La red requiere dimensiones fijadas para la imagen de entrada.
#  Se realiza "mean subtraction" (103.94,116.78,123.68) para
normalizar las imágenes de entrada
# Después de esto el BLOB de entrada tiene forma (1, 3, dim, dim)
donde la dimensión tiene que ser de 224.

# Para todas las imágenes del array:
for j in range(len(imagenes)):
    # Se especifica el BLOB como entrada de la red y se escala los
valores BGR multiplicando por 0.017:
    blob = cv2.dnn.blobFromImage(imagenes[j], 1, (224, 224),
(103.94,116.78,123.68))
```

```
      blob = blob * 0.017
      red.setInput(blob)

      # Se obtiene la clasificación de salida y se mide el tiempo de
ejecución:
      inicio = time.time()
      resultado = red.forward()
      final = time.time()
        print("[Info]  El  tiempo  de  ejecución  fue  de  {:.3}
segundos".format(final - inicio))
      if j>0:
  # Dentro del array se excluye la 1ra clasificación (solo del 2do
elemento del array al final)
        tiempos.append(final - inicio)

      # Top1 predicción:
      resultado = resultado.reshape((1, 1000))
      indice = np.argsort(resultado[0])[::-1][:1]

      # Bucle sobre las predicciones:
      for (i, indice) in enumerate(indice):
          # Dibuja la predicción principal en la imagen:
          if i == 0:
              # Muestra la clase y la probabilidad en la consola:
                        print("[Info]  Clase:  {}.  Probabilidad:
{:.4}%\n".format(clases[indice+1], 100-(resultado[0][indice])/10))

             # Muestra la imagen que se le ha realizado la inferencia:
                                              texto = "Clase:      {},
{:.2f}%".format(clases[indice+1],100-(resultado[0][indice])/10)
              cv2.putText(imagenes[j], texto, (5, 25),
cv2.FONT_HERSHEY_SIMPLEX, 0.7, (0, 0, 255), 2)
              cv2.imshow("Imagen", imagenes[j])
              cv2.waitKey(0)

  # Se crean variables para crear las estadísticas finales:
  media = np.mean(tiempos) # Media
  minimo = np.min(tiempos) # Tiempo más rápido
  maximo = np.max(tiempos) # Tiempo más lento

  # Se imprime por pantalla la media, y la clasificación más rápida y
más lenta:
  print('[Info] La media de clasificación de la red neuronal es de
{:.3} segundos.'.format(media))
  print('[Info]  La  clasificación  más  rápida  ha  sido  de  {:.3}
segundos.'.format(minimo))
  print('[Info]  La  clasificación  más  lenta  ha  sido  de  {:.3}
segundos.\n'.format(maximo))
```

Este es el código de inferencia de la red DenseNet169 para usar con la Raspberry Pi. Para realizar la inferencia en las otras plataformas, basta con cambiar el *backend* y el *target* y los formatos de archivos a cargar según corresponda, tal y como se mencionó en el capítulo cinco de la memoria.

Código de inferencia de ResNet50:

```
  #!/usr/bin/env python
  # coding: utf-8
  # Se importan librerías:
  import os # Listado archivos
  import cv2 # OpenCV
  import numpy as np # Manipular arrays
  import time # Medir el tiempo de ejecución

  # Se carga el documento de clases de Imagenet:
  filas = open('/home/pi/T-F-
M/resnet50/modelos/labels.txt').read().strip().split("\n")
  # Divide en filas el documento
  clases = [r[r.find(" ") + 1:].split(",")[0] for r in filas]
  # Recoge las clases por cada fila

  # Se especifica el directorio principal y se crea un array para
recopilar las imágenes y los tiempos:
  directorio = "/home/pi/T-F-M/resnet50/test"
  imagenes = []
  tiempos = []

  # Se carga la red:
  red = cv2.dnn.readNet('/home/pi/T-F-M/resnet50/modelos/resnet-
50.xml', '/home/pi/T-F-M/resnet50/modelos/resnet-50.bin')

  # Se especifica el dispositivo objetivo (MYRIAD):
  red.setPreferableBackend(cv2.dnn.DNN_BACKEND_INFERENCE_ENGINE)
  red.setPreferableTarget(cv2.dnn.DNN_TARGET_MYRIAD)

  # Se leen todas las imágenes y se van añadiendo al array:
  for dirPath, dirNames, fileNames in os.walk(directorio):
  # Genera los nombres de los archivos
      print('Leyendo archivos...')
      for f in fileNames:
          imagen = cv2.imread(os.path.join(dirPath, f))
  # Se lee una imagen dentro del directorio establecido
          imagenes.append(imagen)
  # La imagen se añade al array
  print('[Info] Todas las imágenes han sido leídas\n')

  # La red requiere dimensiones fijadas para la imagen de entrada.
  # Se realiza "mean subtraction" (104, 117, 123) para normalizar las
imágenes de entrada
  # Después de esto el BLOB de entrada tiene forma (1, 3, dim, dim)
donde la dimensión tiene que ser de 224.

  # Para todas las imágenes del array:
  for j in range(len(imagenes)):
      # Se especifica el BLOB como entrada de la red y se obtiene la
clasificación en la salida de la red:
       blob = cv2.dnn.blobFromImage(imagenes[j], 1, (224, 224), (104,
 117, 123))
      red.setInput(blob)
```

```
      # Se obtiene la clasificación de salida y se mide el tiempo de
ejecución:
      inicio = time.time()
      resultado = red.forward()
      final = time.time()
        print("[Info]  El  tiempo  de  ejecución  fue  de  {:.3}
segundos".format(final - inicio))
      if j>0:
  # Dentro del array se excluye la 1ra clasificación (solo del 2do
elemento del array al final)
        tiempos.append(final - inicio)

      # Top1 predicción:
      resultado = resultado.reshape((1, 1000))
      indice = np.argsort(resultado[0])[::-1][:1]

      # Bucle sobre las predicciones:
      for (i, indice) in enumerate(indice):
          # Dibuja la predicción principal en la imagen:
          if i == 0:
              # Muestra la clase y la probabilidad en la consola:
                        print("[Info]  Clase:  {}.  Probabilidad:
{:.4}%\n".format(clases[indice+1], resultado[0][indice]*100))

             # Muestra la imagen que se le ha realizado la inferencia:
                                          texto = "Clase:      {},
{:.2f}%".format(clases[indice+1],resultado[0][indice] * 100)
              cv2.putText(imagenes[j], texto, (5, 25),
cv2.FONT_HERSHEY_SIMPLEX, 0.7, (0, 0, 255), 2)
              cv2.imshow("Imagen", imagenes[j])
              cv2.waitKey(0)

  # Se crean variables para crear las estadísticas finales:
  media = np.mean(tiempos) # Media
  minimo = np.min(tiempos) # Tiempo más rápido
  maximo = np.max(tiempos) # Tiempo más lento

  # Se imprime por pantalla la media, y la clasificación más rápida y
más lenta:
  print('[Info] La media de clasificación de la red neuronal es de
{:.3} segundos.'.format(media))
  print('[Info]  La  clasificación  más  rápida  ha  sido  de  {:.3}
segundos.'.format(minimo))
  print('[Info]  La  clasificación  más  lenta  ha  sido  de  {:.3}
segundos.\n'.format(maximo))
```

Este es el código de inferencia de la red ResNet50 para usar con la Raspberry Pi y el Movidius Neural Compute Stick. Para realizar la inferencia en las otras plataformas, basta con cambiar el *backend* y el *target* y los formatos de archivos a cargar según corresponda, tal y como se mencionó en el capítulo cinco de la memoria.

BIBLIOGRAFÍA

[1] Edge Computing: Vision and Challenges [Online]. Available: https://cse.buffalo.edu/faculty/tkosar/cse710_spring20/shi-iot16.pdf [Último acceso: 28-05-2020]

[2] ¿Qué es Edge Computing? [Online]. Available: https://www.business-solutions.telefonica.com/es/cloud-hub/knowledge-center/what-is-edge-computing/ [Último acceso: 21-05-2020]

[3] Edge Computing - Key drivers and benefits [Online]. Available: http://iiot-world.com/smart-manufacturing/edge-computing-key-drivers-and-benefits-for-smart-manufacturing/ [Último acceso: 28-05-2020]

[4] NVIDIA TensorRT: Programmable Inference Acelerator [Online]. Available: https://developer.nvidia.com/tensorrt [Último acceso: 29-05-2020]

[5] Jetson Nano Developer Kit [Online]. Available: https://developer.nvidia.com/embedded/jetson-nano-developer-kit [Último acceso: 29-05-2020]

[6] Dev Board Coral [Online]. Available: https://coral.ai/products/dev-board [Último acceso: 30-05-2020]

[7] USB Accelerator Coral [Online]. Available: https://coral.ai/products/accelerator [Último acceso: 30-05-2020]

[8] Intel Developer Zone [Online]. Available: https://software.intel.com/content/www/us/en/develop/hardware/neural-compute-stick.html [Último acceso: 31-05-2020]

[9] OpenVINO Toolkit [Online]. Available: https://docs.openvinotoolkit.org/latest/index.html#toolkit_components [Último acceso: 31-05-2020]

[10] Introduction to Inference Engine [Online]. Available: https://docs.openvinotoolkit.org/latest/_docs_IE_DG_inference_engine_intro.html [Último acceso: 31-05-2020]

[11] Raspberry Pi 3 Model B+ [Online]. Available: https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus/ [Último acceso: 01-06-2020]

[12] Deep Learning: Introducción práctica con Keras (Jordi Torres) [Online]. Available: https://torres.ai/deep-learning-inteligencia-artificial-keras/ [Último acceso: 01-06-2020]

[13] Diferencia entre Inteligencia Artificial - Machine Learning - Deep Learning (ligdieli) [Online]. Available: https://ligdigonzalez.com/diferencia-entre-inteligencia-artificial-machine-learning-deep-learning/ [Último acceso: 07-06-2020]

[14] Redes neuronales profundas - Tipos y Características [Online]. Available: https://www.codigofuente.org/redes-neuronales-profundas-tipos-caracteristicas/ [Último acceso: 07-06-2020]

[15] About Train, Validation and Test Sets in Machine Learning (Tarang Shah) [Online]. Available: https://towardsdatascience.com/train-validation-and-test-sets-72cb40cba9e7 [Último acceso: 03-06-2020]

[16] How to Choose Loss Functions When Training Deep Learning Neural Networks (Jason Brownlee) [Online]. Available: https://machinelearningmastery.com/how-to-choose-loss-functions-when-training-deep-learning-neural-networks/ [Último acceso: 04-06-2020]

[17] Overview of different Optimizers for neural network (Renu Khandelwal) [Online]. Available: https://medium.com/datadriveninvestor/overview-of-different-optimizers-for-neural-networks-e0ed119440c3 [Último acceso: 04-06-2020]

[18] Convolutional Neural Network -- A Begginer's Guide (Krut Patel) [Online]. Available: https://towardsdatascience.com/convolution-neural-networks-a-beginners-guide-implementing-a-mnist-hand-written-digit-8aa60330d022 [Último acceso: 01-06-2020]

[19] A Comprehensive Guide to Convolutional Neural Networks (Sumit Saha) [Online]. Available: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53 [Último acceso: 01-06-2020]

[20] Accelerate Computer Vision & Deep Learning with OpenVINO toolkit [Online]. Available: https://software.intel.com/content/www/us/en/develop/blogs/accelerate-computer-vision-from-edge-to-cloud-with-openvino-toolkit.html [Último acceso: 10-06-2020]

[21] How the Model Optimizer Works [Online]. Available: https://docs.openvinotoolkit.org/latest/_docs_MO_DG_prepare_model_Prepare_Trained_Model.html [Último acceso: 13-06-2020]

[22] Using OpenVINO with OpenCV (Vishwesh Shrimali) [Online]. Available: https://www.learnopencv.com/using-openvino-with-opencv/ [Último acceso: 10-06-2020]

[23] Procesador Intel Core i7-8550U [Online]. Available: https://ark.intel.com/content/www/es/es/ark/products/122589/intel-core-i7-8550u-processor-8m-cache-up-to-4-00-ghz.html [Último acceso: 10-06-2020]

[24] FAQ Colaboratory [Online]. Available: https://research.google.com/colaboratory/faq.html [Último acceso: 10-06-2020]

[25] Myriad 2: Eye of the Computational Vision Storm (David Moloney) [Online]. Available: https://www.hotchips.org/wp-content/uploads/hc_archives/hc26/HC26-12-day2-epub/HC26.12-6-HP-ASICs-epub/HC26.12.620-Myriad2-Eye-Moloney-Movidius-provided.pdf [Último acceso: 14-06-2020]

[26] Densely Connected Convolutional Networks paper [Online]. Available: https://arxiv.org/pdf/1608.06993.pdf [Último acceso: 13-06-2020]

[27] Deep Residual Learning for Image Recognition paper [Online]. Available: https://arxiv.org/pdf/1512.03385.pdf [Último acceso: 13-06-2020]

[28] Image Classification Architectures review (Prakash Jay) [Online]. Available: https://medium.com/@14prakash/image-classification-architectures-review-d8b95075998f [Último acceso: 12-06-2020]

[29] DenseNet121 architecture [Online]. Available: https://www.researchgate.net/figure/Left-DenseNet121-architecture-Right-Dense-block-conv-block-and-transition-layer_fig4_331364877 [Último acceso: 12-06-2020]

[30] ResNet50 architecture [Online]. Available: https://www.researchgate.net/figure/ResNet-50-architecture-26-shown-with-the-residual-units-the-size-of-the-filters-and_fig1_338603223 [Último acceso: 12-06-2020]

[31] Understanding ResNet50 architecture (Aakash Kaushik) [Online]. Available: https://iq.opengenus.org/resnet50-architecture/ [Último acceso: 12-06-2020]

[32] Deep Learning In OpenCV (Wu Zhiwen) [Online]. Available: https://elinux.org/images/9/9e/Deep-Learning-in-OpenCV-Wu-Zhiwen-Intel.pdf [Último acceso: 16-06-2020]

GLOSARIO

Benchmark: Prueba de rendimiento 1

Kaggle: Comunidad online mundial de ciencia de datos donde se publican *datasets* o se realizan competiciones para superar unos desafíos de ciencia de datos 1

Data Center: Centro de procesamiento de datos. 3

Nube: Modelo de almacenamiento de datos basado en redes de computadoras, donde los datos están alojados en espacios de almacenamiento virtualizados. 3

Red de distribución de contenidos: Conjunto de servidores ubicados en distintos puntos de la red que contienen copias locales de contenidos que están almacenados en otros servidores. 3

Sistema en tiempo de ejecución: Software que provee servicios para un programa en ejecución pero que no es considerado en sí mismo como parte del sistema operativo. 4

Framework: Estructura tecnológica de soporte definido que puede servir de base para la organización y desarrollo de software. 4

Kernel: Núcleo del sistema que gestiona recursos y comunica hardware y software. 4

Plug-and-play: Característica de aquellos dispositivos que pueden conectarse a algún dispositivo informático sin necesidad de configuración. 6

Pesos: Parámetros numéricos de una red neuronal que determinan cuán fuertemente cada una de las neuronas afecta a las otras. 12

Sesgo: Parámetro de una red neuronal que se utiliza para ajustar la salida junto con la suma ponderada de las entradas a la neurona. 12

Regresión: Problema de *machine learning* que predice valores de una variable dependiente 'y' respecto otras variables 'x'. 15

Clasificación: Problema de *machine learning* que tiene el objetivo de predecir a que clase pertenece un conjunto de datos. 15

RGB: Modelo de colores en el que el rojo, el verde y el azul se unen para poder representar cualquier color. 16

Tensor: Objetos matemáticos que almacenan valores numéricos y que pueden tener distintas dimensiones. Así, por ejemplo, un tensor de 1D es un vector, de 2D una matriz, de 3D un cubo, etc. 16

Backend: Parte de un sistema o aplicación que no es accesible para el usuario y que normalmente se encarga de almacenar y manipular datos. 21

Debian: Comunidad conformada por desarrolladores y usuarios, que mantiene un sistema operativo GNU basado en software libre. 22

Jupyter: Entorno de trabajo interactivo que permite desarrollar software integrando en un mismo documento llamado *notebook* (cuaderno) tanto código como elementos de texto. 23

Procesador vectorial: Diseño de CPU capaz de ejecutar operaciones matemáticas sobre múltiples datos de forma simultánea, en contraste con los procesadores escalares, capaces de manejar sólo un dato cada vez. 24

Montar: Acción de integrar un sistema de archivos alojado en un determinado dispositivo dentro del árbol de directorios de un sistema operativo. 25

Programa: Secuencia de instrucciones que pueden ser ejecutadas por una computadora para realizar una tarea específica. 26