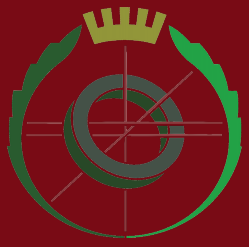


Portabilidad y optimización de una red neuronal para la detección rápida de daños en terremotos usando el toolkit OpenVINO

Portability and optimization of a neural network for rapid damage detection in earthquakes using OpenVINO toolkit

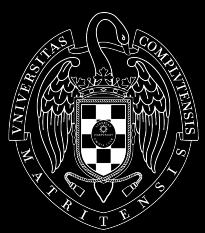
Grado en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid

Trabajo de Fin de Grado

Autor
Adrián Fernández de la Torre

Tutores
Sergio Bernabé García
Carlos González Calvo

26 de Junio de 2020



UNIVERSIDAD
COMPLUTENSE
MADRID

Agradecimientos

En primer lugar, a mis tutores por el apoyo y libertad que me han proporcionado durante todo el desarrollo de este trabajo, mostrando una amplia confianza en mi. Proponiendo ideas sin imponerlas, siempre contando con mi opinión por igual. Por otro lado, mis amigos y compañeros con los cuales he compartido grandes experiencias a lo largo del grado. Y por último y parte principal, mi familia que siempre me ha apoyado ciegamente sin saber muy bien lo que estaba haciendo. Dejando que el testarudo se tropiece una y otra vez con la misma piedra, pero siempre haciendo que se pregunte el porqué.

Índice general

Índice de figuras

Índice de tablas

Resumen

El contenido de este trabajo se centra en el análisis y mejora de todos los procesos relacionados con la generación e implementación de una red neuronal convolucional. Mediante esta técnica perteneciente al campo del aprendizaje profundo o *Deep Learning*, ha sido posible diseñar una aplicación que permite detectar si en una imagen se encuentran zonas dañadas por el impacto de un terremoto. Para ello, se emplea una topología de red específica para esta problemática. En concreto, las imágenes utilizadas provienen del terremoto ocurrido en Haití en el año 2010, obtenidas a partir de un satélite de observación terrestre llamado GeoEye-1.

Mediante el registro de diferentes métricas como el tiempo de ejecución y la precisión, se consigue mejorar el rendimiento del modelo en las fases de entrenamiento e inferencia usando diferentes dispositivos en cada una de estas fases. Empleando en esta última, el *toolkit* de OpenVINO desarrollado por Intel, como medio para aumentar la eficiencia del modelo. Para ello, siendo necesario el establecimiento de una metodología concreta para su uso.

Finalmente, se establece un aumento de rendimiento en los parámetros analizados en la fase de entrenamiento. Mientras tanto en la parte de inferencia, se constata el amplio nivel de mejora obtenido al utilizar OpenVINO en comparación a la herramienta con el mismo propósito proporcionada por Tensorflow. Suponiendo entre un 20 % - 60 % dependiendo de la tecnología y dispositivos empleados. Siendo entre ellos, un *stick* Movidius NCS2.

Palabras clave

Aprendizaje Profundo, Redes Neuronales Convolucionales, Clasificación de Imágenes, Tensorflow, Movidius NCS2, OpenVINO, Inferencia.

Abstract

The content of this work focus on the analysis and enhancement of all the processes related to the generation and implementation of a convolutional neural network. Through this technique belonging to the field of Deep Learning, it has been possible to design an application which allows us to detect whether there are damaged zones for an earthquake impact in an image. Therefore, it is used a specific topology for this problem. Specifically, the images used provided by the earthquake occurred in Haití in 2010, obtained from an earth observation satellite called GeoEye-1.

Through this registry of different metrics such as execution time and accuracy, it manages to improve the performance of the model in training and inference phases using different devices for each one. Using in the last one, the OpenVINO toolkit developed by Intel for increasing the efficiency of the model. For that, being necessary for the establishment of a methodology for its usage.

Finally, it establishes an increase in the performance of the analyzed parameters in the training phase. Whereas in the inference part, it verifies the wide level of improvement obtained by the usage of OpenVINO in comparison to the other tool with the same purpose provided by Tensorflow. Assuming between a 20 % to 60 % of improvement depending on the technology and devices used. Being one of those, a Movidius stick NCS2 (Neural Compute Stick 2).

Keywords

Deep Learning, Convolutional Neural Network, Image Classification, Tensorflow, Movidius NCS2, OpenVINO, Inference.

Capítulo 1

Introducción

En la actualidad, la inteligencia artificial es una de las principales tendencias en diversos sectores como la sanidad, seguridad, etc. Esta ofrece una amplia variedad de aplicaciones posibles, que en algunos casos permiten mejorar procesos internos y aumentar su eficiencia. Dentro de esta, se encuentra el campo *Deep Learning*, el cual, alberga todo lo relacionado con redes neuronales suponiendo una de las técnicas más aplicadas.

El conocimiento de este tipo de tecnologías se remonta varias décadas atrás, donde Frank Rosenblatt definió en 1957 el primer modelo de red neuronal, denominada perceptron[1]. Esta se sigue utilizando hoy en día para aplicaciones destinadas al reconocimiento de patrones.

Sin embargo, el descubrimiento de estas innovaciones no ha ido a la par con la capacidad de cómputo disponible para su ejecución. Principalmente, porque requerían cantidades ingentes de datos y no se disponía de esa potencia computacional para realizar su procesamiento debido a la utilización de operaciones de alta complejidad matemática y la gestión en paralelo de ese gran número de datos.

La capacidad de cómputo ha dado un vuelco estos últimos años, donde cada vez los dispositivos de menor tamaño poseen un hardware de altas prestaciones, generando el paradigma de computación en el borde (*Edge Computing*) [3]. El cual, trata de acercar lo máximo posible el procesamiento y almacenamiento de los datos a la localización donde se necesiten. De esta manera, mejoran los tiempos de respuesta y reducen el ancho de banda con otros componentes.

Por lo tanto, este paradigma requiere que la eficiencia hardware-software sea la máxima para

[1] `https://www.timetoast.com/timelines/historia-del-perceptron`

poder ofrecer el mayor soporte posible. Esto provoca que los propios fabricantes del hardware generen herramientas para optimizar estos procesos.

1.1. Motivación

En el campo de *deep learning* hay diversos *frameworks* disponibles (como Tensorflow, Pytorch, Caffe, etc.) [4] para su puesta en práctica. Sin embargo, actualmente no existe una democratización de estos, por lo que el uso de cada uno suele depender de sus características y la experiencia del desarrollador con éste.

Por lo tanto, los fabricantes hardware tienen que adaptar su tecnología en base a estos *frameworks*, generando que su optimización sea más costosa y compleja. Estas tecnologías se encuentran actualmente en desarrollo, por lo que no existen evidencias concretas de que haya una mejora real con respecto al software destinado a diseñar e implementar la red. Aunque, si la hubiese también se desconoce cuanto supone. Adicionalmente, la eficiencia de la red depende en gran medida de las características de esta y los datos suministrados, siendo este hecho particularmente notable en las que su diseño es completamente personalizado.

Al tratarse de un software tan reciente, la documentación y procesos no se encuentran muy bien definidos por lo que este trabajo tratará de abordar todas estas cuestiones, realizando la implementación de una red diseñada desde cero para una problemática concreta.

1.2. Objetivos

El principal objetivo es establecer una **metodología** para portar redes neuronales a dispositivos Intel, mediante el *toolkit* de OpenVINO [5] para la aceleración de la fase de inferencia. Con este fin, sería conveniente desglosarlo en otros propósitos de menor envergadura para poder establecer un flujo preliminar a seguir y una referencia en detalle de las posibles partes que incluyen el desarrollo de esta metodología. Los objetivos propuestos en base al principal son:

- Optimización y análisis de la red neuronal proporcionada.
- Portabilidad e implementación de los modelos generados para cualquier dispositivo de Intel,

mediante el uso del *toolkit* de OpenVINO.

- Optimización del motor de inferencia proporcionado por el *toolkit*.
- Obtención de resultados y realización de comparativas de rendimiento entre diferentes dispositivos.

1.3. Plan de Trabajo

Una vez definidos los objetivos principales dispuestos en este trabajo, se establece la planificación de su desarrollo en el marco temporal establecido para el proyecto. Para su representación, se ha utilizado un diagrama de Gantt (ver Figura 1.1), el cual, define de manera muy sencilla la duración y competencias de los objetivos establecidos.

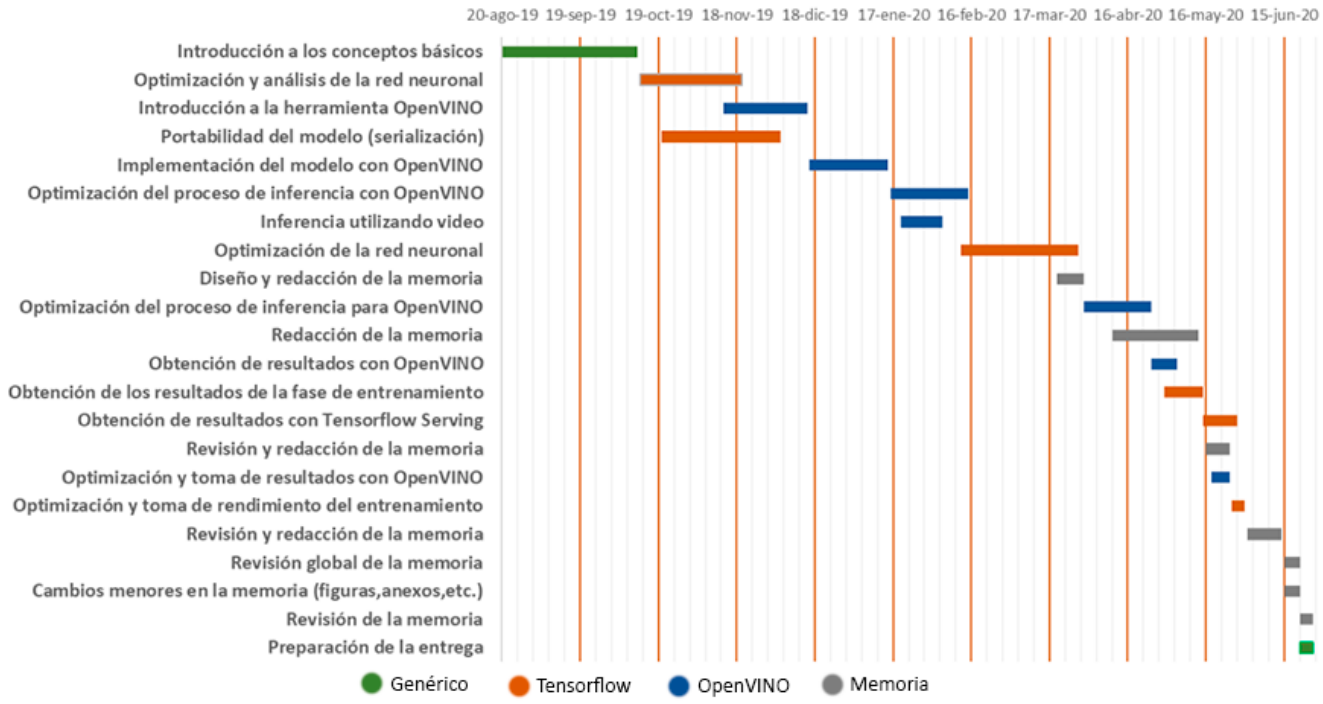


Figura 1.1: Diagrama de Gantt de la planificación realizada.

De esta manera, se recogen y estructuran todos los aspectos englobados en este proyecto. Aunque es necesario mencionar que todas las partes no recogidas en este documento como pueden ser código, artefactos, etc., se encuentran almacenados en el siguiente **repositorio de GitHub:**

`https://github.com/adri1197/DP_Image-Binary-Classification`.

1.4. Organización de la memoria

Teniendo en cuenta los objetivos especificados anteriormente, esta memoria aborda todo el proceso desde el análisis de la topología de la red neuronal hasta su implementación y obtención de resultados de calidad y rendimiento. Por lo tanto, el presente documento seguirá el orden mencionado, detallando el proceso y la teoría en los casos que se considere necesario para la comprensión de la idea o ideas posteriores.

Por tanto, la memoria contendrá (siguiendo el orden en el que están expuestos):

- **Estado del arte**: Descripción de la situación y explicación de los conceptos que representan la base teórica utilizada en la realización de este proyecto. Además, se incluirán todas las herramientas tanto de ámbito software como hardware, empleadas durante el proceso de desarrollo.

- **Descripción del modelo**: Explicación en detalle de las diferentes partes que forman una red neuronal para que pueda ser implementada, particularizando en la composición de este modelo. A través de la explicación de estos elementos se busca proporcionar unos conceptos más generales, con el propósito de dar una visión mucho más amplia, es decir, aprender el funcionamiento interno de una red mediante este caso específico.

- **Portabilidad e inferencia en dispositivos Intel**: Descripción del proceso de implementación de una red neuronal funcional usando un *toolkit* creado por la empresa Intel para poder adaptar de manera homogénea una red neuronal a cualquier dispositivo perteneciente a la compañía. Se definen los pasos que se deben realizar, además de diversas funcionalidades y aplicaciones que han sido implementadas.

- **Resultados Obtenidos**: Exposición y medición del rendimiento de la red neuronal con diferentes configuraciones (número de datos, configuración de la red, etc.) en el proceso de entrenamiento y con el *toolkit* de OpenVINO, implementando la red en distintos dispositivos

Intel. Con todo ello se busca comparar los diferentes rendimientos resultantes y obtener una serie de conclusiones en base a estos.

- **Conclusiones y trabajo futuro**: Resumen de las diferentes respuestas o ideas que aporta este proyecto sobre el tema propuesto, utilizando los datos y descubrimientos expuestos durante los anteriores apartados. Además, a partir de estos, se reflejarán una serie de ideas que se consideran factibles o buenas para ser implementadas como una continuación a este trabajo (si se considera viable en un futuro), pero que por diversos factores (falta de tiempo, recursos o que simplemente no se encontraba en el espectro planteado por el proyecto) no han sido posibles de implementar.

Capítulo 2

Estado del Arte

Tras definir la motivación y el propósito de este proyecto, se van a tratar los principales conceptos que lo componen. Estos representan la base teórica sobre la que se sustenta. La intención con este capítulo es ofrecer una vista introductoria sobre los conocimientos necesarios, procurando que esta no sea excesivamente teórica o difícil de entender sin una experiencia previa. Además de esto, se expondrá el software y el hardware utilizados mostrando sus objetivos de uso y sus funcionalidades para este caso en particular.

2.1. Deep Learning

El concepto *deep learning* o aprendizaje profundo [6] consiste en un conjunto de algoritmos basados en redes neuronales que forman parte de uno de los múltiples métodos de aprendizaje automático o *machine learning* dentro del campo de la inteligencia artificial [7] (mostrado en la Figura 2.1).

Estos algoritmos permiten inferir características abstractas de los datos de entrada. Con ellos se intenta emular la percepción humana, permitiendo asimilar diferentes representaciones de datos (por ejemplo, una imagen o un vector de *píxeles*) generando un modelo para reconocer estas e intentado definir cuál es la mejor de ellas[1].

Este tipo de técnicas han sido aplicadas a campos como visión artificial (*computer vision*), reconocimiento automático del habla o reconocimiento de señales de audio y música.

[1] `https://es.wikipedia.org/wiki/Aprendizaje_profundo`

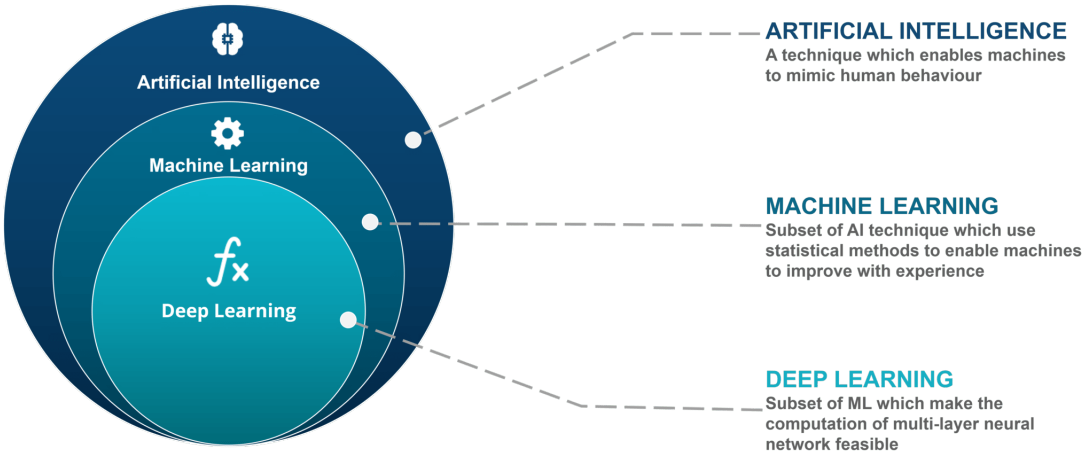


Figura 2.1: Visión general de la inteligencia artificial.

2.1.1. Redes Neuronales

Uno de los puntos clave del concepto de *deep learning* es la idea de **red neuronal**. Esta se puede definir como la simulación del funcionamiento neuronal humano de manera artificial mediante un modelo matemático. Por ello, está compuesta por un conjunto de neuronas o nodos que realizan una serie de operaciones en cada capa (*layer*) de tal forma que, a partir de unos datos de entrada obtengamos la predicción (salida).

Sin embargo, para llegar a obtener un resultado concluyente con la red, es necesario definir y realizar una serie de procedimientos. En primera instancia, hay que establecer una definición/topología que se ajuste al problema que se desee resolver. A continuación, la red desconoce totalmente el contenido y nuestro propósito de uso con los datos que se encuentren disponibles, por lo que es necesario generar un proceso de aprendizaje. De tal manera que esta pueda ofrecer resultados más cercanos a los esperados y, por tanto, considerar fidedigna la información obtenida por estos. En el anexo B, se encuentran explicados de manera detallada todos estos conceptos y procedimientos relacionados con redes neuronales.

Como se menciona anteriormente, la casuística del problema define la topología y características de la red. Aun así, existen tipos de redes neuronales que proporcionan especialmente buenos resultados en ciertos ámbitos. En el caso de las imágenes, se suelen utilizar las **redes convolucionales** (*Convolutional Neural Networks* (CNN)), que además de reducir la dimensionalidad de

la imagen, nos permiten extraer los patrones más importantes de esta.

2.1.2. Redes Neuronales Convolucionales

Las **Redes Neuronales Convolucionales** [8] son muy similares a las redes ordinarias mostradas en el apartado anterior. Están formadas por una serie de neuronas con las que aprenden, almacenando esa información en pesos y *biases*. En cada una de ellas se reciben unos datos de entrada, se aplica la función de transferencia y posteriormente, una función de activación que será no lineal en este tipo de redes. Además, posee una función de pérdida (SVM/Softmax[2], normalmente) situada en la última capa, para seguir aplicando el conjunto de consejos desarrollados en el aprendizaje de una red neuronal convencional [9].

Las redes neuronales convencionales no escalan muy bien con imágenes. Hay que tener en cuenta que cada una de ellas está formada por tres dimensiones: alto (*height*), ancho (*width*) y los canales de color (*depth*). Por lo que cada neurona tendrá: alto x ancho x color pesos. Esto puede resultar "manejable" para imágenes no muy grandes, pero seguirá suponiendo un número muy considerable de parámetros, lo que provoca que la estructura de la red sea más grande y haya una mayor posibilidad de tener que lidiar con sobreajuste en la red y altos tiempos de ejecución.

Este tipo de redes (CNN) toman ventaja suponiendo que cada dato de entrada va a ser una imagen, haciendo que cada neurona perteneciente a cada capa solo esté conectada con una pequeña región de la capa anterior, en lugar de con todas las neuronas como sucede en una red totalmente conectada (*fully-connected*). Por lo tanto, las dimensiones de la imagen se van reduciendo progresivamente en su paso por cada capa.

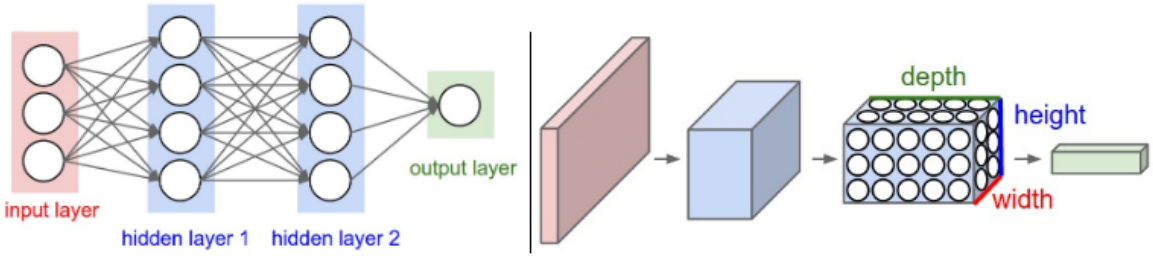


Figura 2.2: Funcionamiento general de una red neuronal convolucional.

[2] `https://cs231n.github.io/linear-classify/`

Como se muestra en la Figura 2.2 y se menciona al principio de este apartado, este tipo de redes contienen un conjunto de capas que se consideran por defecto, o al menos que suelen ser utilizadas en su implementación. En el capítulo 3 se explica más detenidamente el conjunto de capas empleadas en el modelo utilizado, explicando su funcionamiento y el propósito de su uso. De manera más general, la topología de una red de estas características posee las siguientes capas definidas como estándar:

- Capa de Entrada
- Capa de Convolución
- Capa de *Pooling*
- Capa Totalmente Conectada (*fully-connected*, FC)

Algunas de estas capas poseen parámetros adicionales (hiperparámetros) a definir. Siguiendo esta estructura "generalizada", las capas de convolución y FC aplican una serie de transformaciones en función de los parámetros (pesos y *biases*) y de los datos de entrada. Mientras que las capas de *pooling* y de activación implementan una función fija en cada capa.

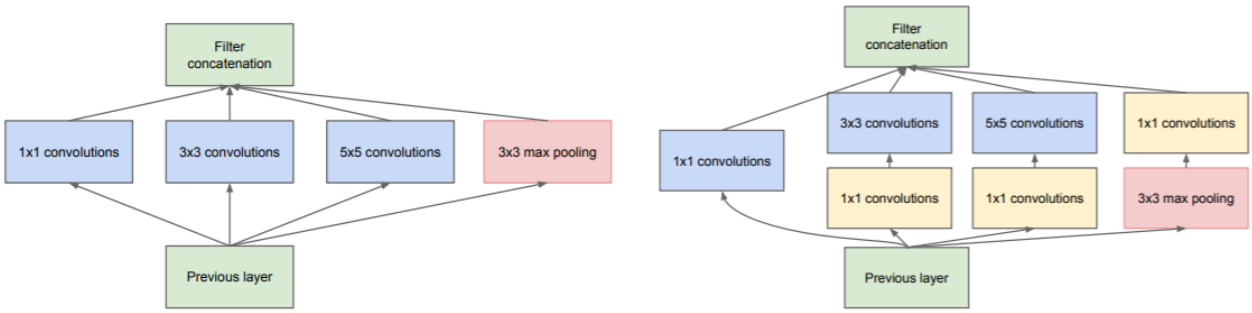
Dentro de este tipo concreto de redes existen una gran cantidad de arquitecturas que tienen un nombre específico, con una estructura muy probada en ciertos ámbitos y proporcionando muy buenos resultados con esa topología o conjunto de elementos en específico. Cada una de ellas añaden, normalmente, alguna técnica o valor con respecto a las anteriores proporcionando nuevos elementos, como por ejemplo, la omisión de conexiones en la red *ResNet*. Las más comunes son (en orden cronológico a su descubrimiento):

- **LeNet:** La primera aplicación exitosa de este tipo de redes. Se utiliza principalmente para leer códigos zip, dígitos, etc.
- **AlexNet:** Se trata de la primera aplicación de esta arquitectura en problemas de visión artificial. La red tiene una arquitectura muy similar que **LeNet**, pero es más profunda, grande y caracterizada con capas convoluciones apiladas una encima de otra (anteriormente era común tener solo una capa de este tipo seguida con una de *pooling*).

- **ZF Net:** Resulta ser una mejora a la anterior **AlexNet**, permitiendo un **ajuste en los hiperparámetros de la arquitectura**, particularmente en la expansión del tamaño de las capas de convolución intermedias. Además, permite que el tamaño de desplazamiento (*stride*) y del filtro sea inferior en la primera capa.

- **GoogleNet:** Su principal contribución es el desarrollo de un *Incepcion Module* que **reducía drásticamente el número de parámetros de la red** (4M, comparado con *AlexNet* con 60M). Asimismo, utiliza *Average Pooling* en vez de capas totalmente conectadas (*fully-connected*) al principio de la red, eliminando un gran conjunto de parámetros que no parecen tener mucha relevancia. Existen diferentes versiones relacionadas con este tipo, la más actual se denomina **Inception-v4**.

  El ***Inception Module*** permite una computación más eficiente y redes más profundas a través de la reducción dimensional. Además de resolver los problemas del gasto computacional, permiten reducir el sobreajuste de la red y otros problemas más. La solución que propone consiste en coger múltiples tamaños de filtros del *kernel* con la red neuronal convolucional. Y en vez de ser colocados secuencialmente, se establecen de tal manera que pueden operar al mismo nivel (como muestra la Figura 2.3).



(a) Inception module, naïve version (b) Inception module with dimension reductions

Figura 2.3: Representación de la idea de *Inception Module*.

- **VGGNet:** Muestra que la **profundidad de la red** es un componente **crítico** en el buen rendimiento de una red. La mejor red obtenida con esta arquitectura contiene 16 capas de convolución y de conexión total con una distribución homogénea, ejecutando convoluciones

de 3×3 y *pooling* de 2×2 desde el principio hasta el final. La versión pre-entrenada de este modelo se encuentra en el *framework* de *Caffe*.

- **ResNet:** Introduce el concepto de **omitir conexiones** y un uso de normalización por lotes (*batch normalization*). La arquitectura prescinde de capas de conexión total al final de la red. **ResNets** son actualmente el **estado del arte** de los modelos de redes convolucionales y la opción por defecto a la hora de implementar una red de este tipo.

2.2. Visión Artificial

Este campo de la computación se encarga de permitir que una máquina pueda obtener un entendimiento a alto nivel de una imagen o vídeo. Viéndolo desde la perspectiva de la ingeniería, se busca comprender y automatizar las tareas que la vista humana puede realizar[3]. Se considera como parte de uno de los tipos de inteligencia artificial (IA) más potentes y que, en estos últimos años, el campo del *deep learning* (DL) ha conseguido impulsar con una serie de avances e innovaciones. Otro factor que ha promovido esta situación es el aumento de los datos generados, permitiendo que se puedan entrenar mejor los modelos para poder cubrir más situaciones en base a su casuística, generando una mejora y un crecimiento exponencial.

2.2.1. Funcionamiento

A un cierto nivel, la visión artificial trata sobre el reconocimiento de patrones. Por lo que una manera de entrenar a una máquina para entender los datos visuales es proporcionando una gran cantidad de imágenes que hayan sido categorizadas y, a continuación, mediante una serie de técnicas software o algoritmos, permitir al ordenador indagar sobre esos patrones relacionados con las etiquetas previamente conocidas y definidas [10].

Estas etiquetas mencionadas con anterioridad dependen en gran medida de la información que se quiera obtener de la imagen. Puede consistir únicamente en detectar la aparición de una determinada forma o, añadiendo un cierto nivel de complejidad, reconocer la localización exacta

[3] `https://en.wikipedia.org/wiki/Computer_vision`

de esta en la imagen. Por lo tanto, será necesario conocer la localización con antelación para poder entrenar ese algoritmo, además de la etiqueta de la forma a identificar.

Una imagen analizada desde un punto de vista computacional, representa una matriz de píxeles. Como muestra la Figura 2.4, hay diversos factores que pueden proporcionar una información clave a la hora de realizar ese reconocimiento de patrones descrito con antelación: el color, las formas, las distancias entre ellas, donde los objetos se delimitan, etc.

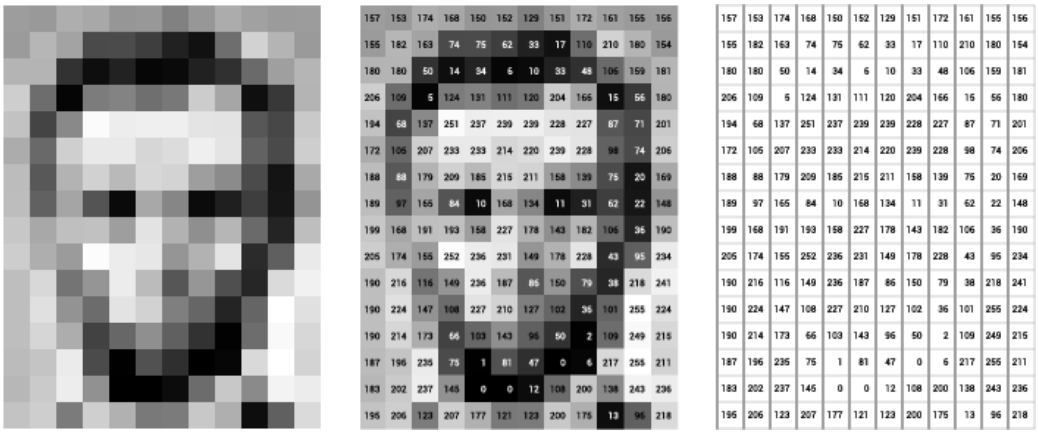


Figura 2.4: Representación de una imagen (escala de grises) en una matriz de píxeles.

2.2.2. Procesamiento de Imágenes

En primer lugar, el concepto de visión artificial difiere al de procesamiento de imágenes. Este último se define como un método para aplicar operaciones en una imagen, en orden de mejora o para extraer información útil de ella [11]. Por lo tanto, es considerado como un tipo de procesamiento de señal digital y no se preocupa por el entendimiento del contenido de la imagen. Sin embargo, este tipo de procesado es necesario en este campo para poder facilitar o destacar ciertas características de la imagen para, posteriormente, realizar ese proceso de aprendizaje por parte de la red. También en un caso donde la imagen este muy deteriorada o no sea muy nítida, este tipo de técnicas nos permiten reducir el ruido o la distorsión en la imagen produciendo una nueva que pueda resultar útil para el propósito de uso propuesto.

Típicamente en trabajos relacionados con *Deep Learning*, suele existir una fase de pre y post procesado de las imágenes o fotogramas (en el caso de un vídeo) para preparar el contenido acorde al posterior proceso de inferencia. Un ejemplo muy sencillo de este tipo de procedimientos sería

la reducción de las dimensiones de los datos de entrada. Esto es debido a que cuanto más grande sea la imagen, más capas necesitará la red para obtener una predicción acorde provocando que el coste computacional sea mayor y por consiguiente, el tiempo de ejecución. Aunque hay que tener especial cuidado con los valores a los que se reduce debido a que se puede perder información relevante de esta. Esta idea representa una técnica de preprocesado muy usual, siendo una de las múltiples que se pueden realizar.

2.2.3. Aplicaciones

En el campo de la visión artificial, el uso de imágenes como datos de entrada es parte de su definición. Hay una innumerable cantidad de problemas que se pueden satisfacer con imágenes y técnicas de *deep learning*. A continuación, se van a identificar los principales usos [12]:

- **Clasificación de Imágenes** (*Image Classification*): Requiere asignar una **etiqueta** a una imagen entera. Este problema también se puede referir a "clasificación de objetos" o quizás de manera más genérica como "reconocimiento de imágenes". Sin embargo, el objetivo de todas ellas es clasificar el contenido de las imágenes con una de las etiquetas anteriormente mencionadas. Algunos ejemplos incluyen:

  - Etiquetar si en una prueba de rayos X se observan indicios de cáncer o no (clasificación binaria).
  - Clasificar un dígito escrito a mano (clasificación con múltiples clases).
  - Asignar el nombre de la persona mediante una fotografía de su cara (clasificación con múltiples clases).

  Dentro de este tipo de aplicación se puede incluir la clasificación **con localización**, la cual consiste en mostrar la ubicación del objeto en la imagen mediante un cuadro delimitador (*bounding box*), asignando una clase por cada imagen.

- **Detección de Objetos** (*Object Detection*): Sigue el mismo principio que la clasificación de imágenes con localización, pero la imagen contiene múltiples objetos que requieren ese proceso de ubicación y clasificación.

- **Segmentación de Objetos** (*Object Segmentation*): Es una tarea de detección de objetos donde una línea es dibujada alrededor de cada objeto detectado, delimitándolo. En algunas ocasiones se refiere a este tipo como detección de objetos.

- **Transferencia de Estilo** (*Style Transfer*): Se trata de un proceso de aprendizaje que permite mediante una imagen o conjunto de estas, generar una nueva aplicando el estilo de las imágenes anteriormente especificadas (ver Figura 2.5).

Figura 2.5: Ejemplo de Transferencia de Estilo.

- **Colorización de Imágenes** (*Image Colorization*): Trata de convertir imágenes que se encuentran en blanco y negro a color. Este tipo de técnicas se utilizan sobre todo para el campo de la restauración y reconstrucción de fotos antiguas o en el campo de la cinematografía.

- **Reconstrucción de Imágenes** (*Image Reconstruction*): Consiste en rellenar las partes que faltan o que se encuentran corruptas de una imagen. El campo de uso es muy parecido al descrito en la colorización de imágenes.

2.3. Software & Hardware

En esta sección se van a tratar las principales herramientas software utilizadas durante el desarrollo de este proyecto. Además se hará mención a algunas otras que no han sido usadas, pero que pueden tener la misma utilidad que las empleadas. De esta manera se puede obtener una vista general en cuanto a las tendencias o recomendaciones con respecto al campo que cubren cada una de ellas.

Finalmente se realizará una descripción del hardware utilizado para poder realizar tanto el proyecto como las pruebas de rendimiento. Hay que tener en cuenta que para este último caso se necesitan equipos relativamente potentes, ya que este tipo de tecnologías requieren una alta capacidad computacional para ser ejecutadas.

2.3.1. Tensorflow

Tensorflow es una plataforma *end-to-end* de código libre para aprendizaje automático diseñada por la empresa Google. Está formado por un ecosistema de herramientas, librerías y otros recursos con los que abastece los flujos de trabajo de alto nivel, APIs. El *framework* ofrece diversos niveles conceptuales para que puedas elegir el que necesites para generar y desplegar modelos. Por ejemplo, si necesitas hacer tareas muy largas y costosas computacionalmente hablando, puedes usar la API *Distributed Strategy* con el fin de configurar distribuciones hardware distribuidas. Algunas otras características que alberga son:

- **Facilidad en la creación de un modelo:** Ofrece diferentes niveles de abstracción para la generación y entrenamiento de los modelos, mediante una serie de APIs (ver Figura 2.6): nivel inferior, te permite definir el modelo mediante una serie de operaciones matemáticas o un nivel superior (como tf.estimator), para especificar arquitecturas predefinidas, como redes neuronales [13].

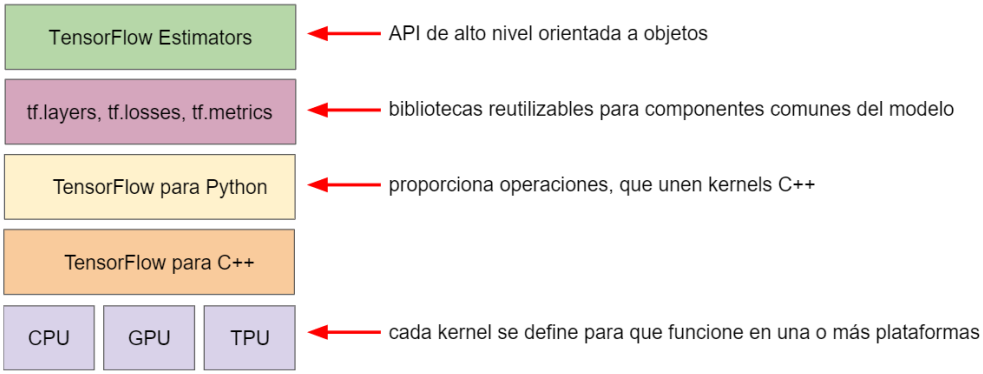


Figura 2.6: Jerarquía de los kits de herramientas disponibles en Tensorflow.

- **Producción robusta del modelo:** Es sencillo entrenar y desplegar los modelos tanto en la nube, en el navegador, como en cualquier otro dispositivo sin importar el lenguaje que uses.
- **Flexibilidad en la creación del modelo:** Mediante sus diferentes APIs se puede definir el modelo siguiendo diferentes niveles de especificación (abstracción).

Proporciona una versión estable para Python y C mediante su API. Aunque también tiene compatibilidad con otros lenguajes: C++, Golang, Java, etc.

Adicionalmente a las funcionalidades que ofrece este *framework*, posee un conjunto de recursos extra a las diferentes APIs que componen esta plataforma. Estos proporcionan una serie de funcionalidades complementarias[4]:

- **Colab:** Se trata de entorno donde se puede editar y ejecutar cuadernos *Jupyter* con hardware dedicado. Se encuentra explicado más en detalle en el apartado 2.3.3.
- **What-If-Tool:** Es un herramienta que permite sin definir código, probar modelos. Esto resulta útil para entenderlos, depurarlos y mejorarlos.
- **ML Perf:** Permite medir la actuación de la fase de entrenamiento e inferencia en el hardware, software o servicio en el que se haya implementado.
- **TensorBoard:** Permite analizar las métricas seleccionadas en el proceso de entrenamiento del modelo. Posee las siguientes funciones principales:

[4] `https://www.tensorflow.org/`

- Visualizar el grafo del modelo (operaciones y capas).
- Revisar el histograma de pesos, *biases* y demás parametros mientras van cambiando.
- Mostrar imágenes, texto o audio generado durante el proceso de entrenamiento.
- *Profiling* o perfilado de los programas generados con Tensorflow.
- Seguimiento y visualización de las métricas definidas en el proceso de entrenamiento (explicado en el apartado 3.3).

Todo ello se puede realizar de manera "local", es decir, generar unos archivos de reporte al finalizar el proceso de entrenamiento de la red y, así, poder revisar el comportamiento de este. Por otro lado, también se puede generar una monitorización en tiempo real de este proceso.

- **XLA** (*Accelerated Linear Algebra*): Es un compilador específico para álgebra lineal, el cual, optimiza la computación con **Tensorflow** haciendo que la velocidad y uso de memoria mejoren. Además, permite la portabilidad a servidores o plataformas móviles.
- **TensorFlow Playground:** Permite probar diferentes topologías de red neuronal en el navegador usando un sistema de ***drag-and-drop***.

Esta plataforma además de proporcionar un entorno de desarrollo, ofrece uno de producción donde poder implementar los modelos y realizar las inferencias en un ecosistema únicamente con este propósito.

Este sigue un modelo cliente-servidor, por lo tanto, es necesario realizar peticiones vía HTTP/gRPC[5] para obtener la predicción (ver flujo en la Figura 2.7). El modelo a utilizar en estas inferencias debe estar almacenado previamente en el servidor y debe ser previamente seleccionado al arrancar este.

[5] `https://grpc.io/`

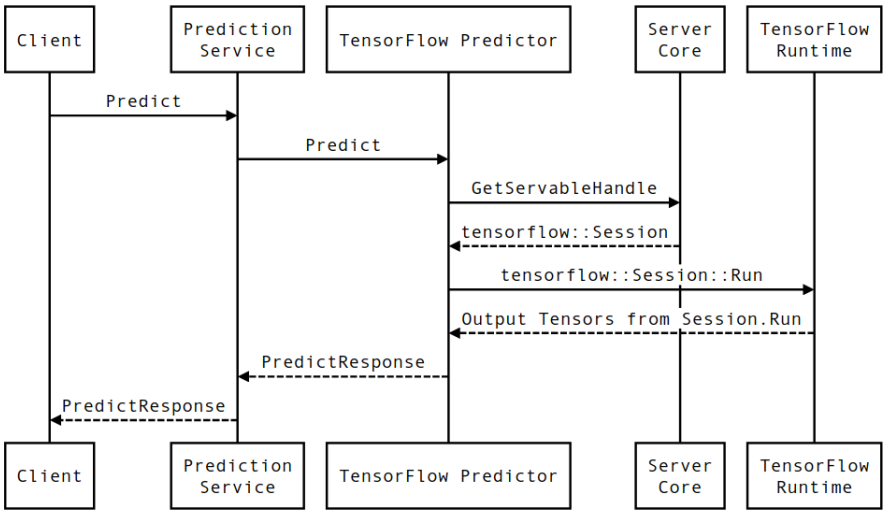


Figura 2.7: Diagrama de secuencia para obtener una predicción, *Tensorflow Serving*.

Existen diferentes formas de desplegar el servidor. Se puede instalar de manera local en nuestro equipo o, por otro lado, utilizar la aplicación contenerizada mediante su imagen de Docker[6].

Esta plataforma forma parte del *toolkit* que trata de integrar desde el desarrollo hasta la puesta de producción, lanzado por Google recientemente. Toda este ecosistema *end-to-end* se denomina *Tensorflow Extented*[7].

2.3.2. OpenVINO

OpenVINO se trata de una herramienta que facilita la optimización de modelos de *Deep Learning* desde un *framework* y despliegue de estos usando un motor de inferencias diseñado para hardware de Intel (CPUs, GPUs, VPUs y FPGAs), permitiendo además la ejecución heterogénea o múltiple de varios de ellos. Este *toolkit* posee dos versiones: una *open-source* mantenida por la comunidad (OpenVINO *toolkit*) y otra soportada por Intel (*Intel Distribution of OpenVINO toolkit*).

[6] `https://docs.docker.com/get-started/`
[7] `https://www.tensorflow.org/tfx`

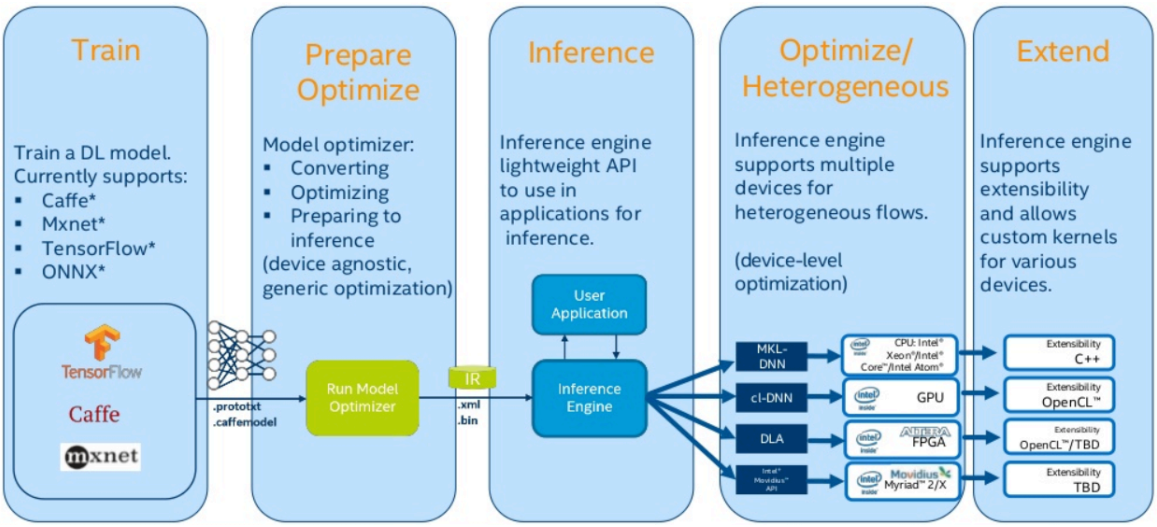


Figura 2.8: Introducción al flujo de ejecución por defecto ofrecido por Intel.

La Figura 2.8 muestra la consecución de pasos necesarios para implementar el modelo elegido y obtener los resultados mediante el motor de inferencias que este dispone. Por otro lado, también muestra el conjunto de dispositivos que soporta y la serie de extensiones disponibles para cada uno de ellos con el fin de mejorar el rendimiento de estos.

En el caso del estudio realizado, se sigue este esquema de manera más o menos fidedigna, aunque se añaden otros elementos a este flujo. Este hecho queda detallado en el capítulo 4. Por lo tanto, este apartado centrará en las principales herramientas ofrecidas por el *toolkit*. OpenVINO alberga dos herramientas principales:

- **Model Optimizer:** Permite la transición entre el entorno de entrenamiento y el de despliegue. Ajusta los modelos para su ejecución óptima en el hardware final. Para ello, carga el modelo en memoria, lo lee y genera una representación interna del modelo. La optimiza y produce lo que se denomina como la **Representación Intermedia** o *Intermediate Representation*, que se trata de la única forma en la que el motor de inferencia acepte y entienda el modelo.

  En primer lugar, hay que obtener un modelo previamente entrenado para poder realizar esta transición. Los *frameworks* soportados para la generación y entrenamiento de ese modelo son: Caffe, TensorFlow, MXNet, Kaldi y ONNX.

Por tanto, el *Model Optimizer* tiene dos propósitos:

- **Producir una representación intermedia válida:** Genera dos archivos (.xml y .bin) que forman esa representación mencionada.
- **Optimizar esa representación:** Los modelos pre-entrenados poseen capas que son importantes para la fase de entrenamiento como *Dropout*. Sin embargo, no tienen ninguna utilidad durante la inferencia y aumentan el tiempo de ejecución innecesariamente. En la mayoría de casos, estas capas pueden ser automáticamente eliminadas como resultado de la **representación intermedia**. Aunque si un grupo de capas puede simplificarse en una única capa, el *Model Optimizer* reconoce estos patrones y remplaza estas capas por una. Finalmente, la *Intermediate Representation* dispondrá de menos capas que el modelo original y esto indicará que el tiempo de inferencia será menor.

Adicionalmente a esto, permite realizar un conjunto de **operaciones** como: dimensionar los datos de entrada, cambiar el tamaño de lotes (*batch*) durante la inferencia, cambiar la estructura de la red (eliminando o intercambiando las capas de su topología) y por último, aplicar estandarización y escalado (normalización) a los datos de entrada.

Otro aspecto a destacar de esta herramienta es la **cuantificación**. La mayoría de los modelos utilizan el formato FP32 como datos de entrada, aumentando el tiempo de inferencia y consumiendo una gran cantidad de memoria. Existen otros tipos de formatos como FP16 y INT8 que pueden usarse, pero tienen ciertas desventajas como la reducción de la precisión en los resultados y en el caso de ciertos dispositivos y capas, algunos formatos no son soportados. Esto hace que la representación resultante genere un modelo con capas que tengan los datos de entrada en formatos diferentes. La encargada de esta conversión se denomina la **capa de calibración** o *Calibrate Layer*.

- **Inference Engine** o Motor de Inferencias: Se trata de una librería escrita en C++ que permite mediante una serie de datos de entrada, obtener un resultado (predicción). Esta posee una API para leer la representación intermedia anteriormente generada, establecer el

formato de entrada y salida y ejecutar el modelo en los dispositivos seleccionados. Aunque está escrita en C++, posee una API para poder realizar inferencias en Python.

Otra herramienta con un propósito similar a esta y que forma parte de una de las competencias de Intel en el campo de los aceleradores, sería NVIDIA con su SDK (*Software Development Kit*) **TensorRT**[8]. El cuál, según detalla su documentación, posee un optimizador de inferencias y permite un tiempo de ejecución de baja latencia y un alto rendimiento para aplicaciones *deep learning* que sean implementadas en dispositivos de la marca[9]. Su desarrollo está realizado en CUDA (creado por NVIDIA), un lenguaje que permite la gestión de computación en paralelo creado para tener un control total sobre el uso de los recursos hardware disponibles en los dispositivos NVIDIA.

2.3.3. Google Colab

Consiste en una plataforma alojada en la nube que permite **albergar** (mediante la integración con Google Drive) y **ejecutar** cuadernos (*notebooks*) de Jupyter[10] en un entorno remoto. Estos cuadernos permiten ejecutar celdas de código (en Python 2.7 o 3.6) y además combinarlas con fragmentos de texto en formato *Markdown*. Esta tecnología resulta ser un estándar en el ámbito de aprendizaje automático, principalmente por lo cómodo que resulta el poder ejecutar pequeños fragmentos de código y visualizar los resultados de manera rápida. Este último hecho es muy importante, por lo que un entorno dinámico como este es de una gran utilidad.

La principal particularidad de esta herramienta es el hardware sobre el que se ejecutan estos *playbooks*, ya que es ciertamente potente. Añadiendo el factor de que, a efectos prácticos, esta plataforma permite ejecutar todo este hardware de manera gratuita, aunque posee ciertas limitaciones relacionadas con la disponibilidad del entorno. El hardware disponible (de manera gratuita) es el siguiente:

- **CPU:** Intel Xeon CPU @ 2.20GHz.

[8] `https://docs.nvidia.com/deeplearning/tensorrt/developer-guide/index.html`
[9] `https://developer.nvidia.com/tensorrt`
[10] `https://jupyter-notebook.readthedocs.io/en/stable/notebook.html`

- **GPU:** NVIDIA Tesla K80.
- **Memoria RAM:** 13 GB.
- **TPU:** Se trata de un tipo de acelerador desarrollado por Google específicamente para el aprendizaje automático (ML). En comparación con las GPUs, estos dispositivos están diseñados implícitamente para un mayor volumen de cálculo de precisión reducida (por ejemplo, desde 8 bits de precisión (INT8)) y carecen de hardware para la rasterización/cartografía de la textura. Este término (TPU) ha sido acuñado para este chip especifico diseñado para el marco de Tensorflow de Google. De todas las versiones que existen de este dispositivo, la *versión 2* es la que se encuentra disponible y dispone de 180 TFLOPS con 64 GB de ancho de banda (HBM).

Existe una versión de pago denominada **Colab Pro** que proporciona una mejora tanto en el hardware que utiliza (GPU, TPU y memoria RAM) como en la parte software, por ejemplo, permitiendo unos tiempos de ejecución más largos.

El principal valor que aporta este entorno es la facilidad de ejecutar y desplegar un cuaderno de Jupyter en un hardware de altas prestaciones, independientemente del equipo disponible. Debido a esto, se ha convertido en una herramienta muy utilizada en tecnologías relacionadas con aprendizaje automático. Hay que añadir que posee ciertas herramientas de *tensorflow* integradas, haciendo su uso mucho más sencillo y con ciertas optimizaciones.

2.3.4. Hardware utilizado

Durante el progreso de este proyecto se han dispuesto de varios equipos con los que se ha podido desarrollar partes especificas de la solución presentada en este documento. Se considera necesario recordar que para el desarrollo y prueba de este tipo de tecnologías y paradigmas se necesita lo que se conoce como **HPC** (*High Performance Computing*). Por lo tanto, las máquinas empleadas disponen de un hardware de altas prestaciones. Este hecho resulta especialmente necesario en las tomas de rendimiento de cada uno de los fragmentos que forman el proyecto.

La solución presentada posee dos partes diferenciadas: **entrenamiento** e **inferencia**. Esto hace que las necesidades sean totalmente diferentes en cada una de ellas. En el primer caso, el hardware

utilizado corresponde al proporcionado por el entorno de **Google Colab** (especificaciones técnicas descritas en el apartado 2.3.3). Por otro lado, en el siguiente, es necesario para la implementación de OpenVINO disponer un hardware de Intel, por lo que ha sido necesario el disponer de otro equipo para implementar esta fase de la solución. Para ello se ha dispuesto de una máquina remota con las siguientes prestaciones:

- **CPU:** Intel Xeon E3-1225 v3 @ 3.20GHz.
- **RAM:** 32 GB.

Adicionalmente a estos dispositivos, ha sido posible disponer de un microprocesador perteneciente a la serie ***Movidius***, diseñada por Intel. Principalmente, destinada a ejecutar aplicaciones de visión artificial usando redes neuronales, proporcionando eficiencia energética, un coste muy reducido en comparación con otros aceleradores de este tipo y siendo relativamente sencillos de escalar según su documentación [14].

Dentro de esta serie, ha sido posible contar con el acelerador **Movidius Myriad X**[11]. Pero en este caso en forma de *stick* USB, este modelo se conoce como Movidus NCS2[12] (*Neural Compute Stick 2*). Incluso, es posible combinar varios de ellos con el fin de obtener un mejor rendimiento [15].

Este tipo de aceleradores y microprocesadores diseñados para ejecutar algoritmos de visión artificial como CNN, SIFT[13] (*Scale-Invariant Feature Transform*) y similares, se conocen como **VPU**(*Vision Processing Unit*[14]). Se trata de una clase de microprocesador denominado como un acelerador pensado para tener un buen flujo de datos *on-chip* con muchas unidades de procesamiento paralelas con *scratchpad memory* [16] (memoria interna de alta velocidad utilizada para el almacenamiento temporal de cálculos, datos y otros trabajos en progreso). Al igual que las unidades de procesamiento de vídeo, tienen una precisión en punto flotante muy baja (FP16) y fija. Algunos ejemplos podrían ser *Pixel Visual Core* (PVC) para dispositivos móviles, el acelerador utilizado en las *HoloLens* de Microsoft o la propia serie de *Movidius*.

[11] `https://www.intel.com/content/www/us/en/artificial-intelligence/movidius-myriad-vpus.html`
[12] `https://software.intel.com/en-us/neural-compute-stick`
[13] `https://towardsdatascience.com/sift-scale-invariant-feature-transform-c7233dc60f37`
[14] `https://en.wikipedia.org/wiki/Vision_processing_unit`

Otra característica esencial de estos dispositivos es su tamaño, debido a que, en relación con la serie de prestaciones expuestas en el párrafo anterior, es relativamente pequeño. Este hecho hace que sean especialmente útiles para problemas relacionados con el **Internet de las Cosas** (IoT).

Capítulo 3

Explicación del Modelo

En este tipo de paradigma (algoritmo de clasificación de imágenes), la definición del modelo no resulta ser un inconveniente ya que se suele escoger sobre un catálogo de modelos específicos para esa problemática concreta. Por supuesto, el rendimiento y tipología de estos modelos está muy probada y medida, pero el conocimiento de los datos disponibles y el análisis del problema a resolver resultan indispensables para la elección del modelo. Además, hay que tener en cuenta los aspectos relacionados con el apartado más técnico del modelo como el formato de los datos de entrada, el funcionamiento del modelo en sí o la salida obtenida. En algunos casos, ninguno de los modelos existentes se ajusta a los datos disponibles o la problemática planteada.

Siendo este uno de ellos, donde el modelo utilizado es específico para la resolución de este problema, el cual, consiste en la clasificación rápida de imágenes relacionadas con **catástrofes naturales (terremotos)**, correspondientes al terremoto sucedido en Haití en el año 2010. El objetivo es identificar qué imágenes representan zonas dañadas y cuáles no. Debido a que solo tienen dos categorías o etiquetas a las que pertenecer: dañado o no dañado, esto se denomina **clasificación binaria**. La principal razón por la que la red está diseñada desde cero es debido a la necesidad de tener la flexibilidad para modificar las capas al igual que los parámetros asociados con cada capa, con el fin de maximizar la precisión de la red [2].

Este modelo está pensado para poder obtener resultados en un entorno casi a tiempo real (lo que se denomina anteriormente como clasificación rápida). Por lo tanto, la relación precisión-tiempo de ejecución del modelo resulta muy importante en las decisiones de diseño de la red. Esto hace que sea importante buscar un equilibrio entre ambas, donde la precisión sea lo suficientemente

buena (mayor número de capas en la red) y que no se obtenga un tiempo de ejecución demasiado alto (menor número de capas definidas en el modelo). Por tanto, se busca maximizar la precisión, minimizando el tiempo de ejecución. Tal y como se expone en los objetivos del trabajo previo [2] de donde se obtiene el diseño de la red, uno de los principales objetivos es ajustar ("*fine-tuned*") el diseño del modelo.

En este capítulo, vamos a tratar el conjunto de elementos y técnicas utilizadas en este modelo específico desde la topología usada hasta algunos métodos utilizados en el modelo o en los datos. Como anteriormente se menciona en la sección 1.4, con esta explicación se busca tanto realizar una descripción exhaustiva de los componentes de esta red en específico, como la de redes neuronales convolucionales (CNN) en general.

El orden a seguir para esta explicación va a estar muy relacionado con el código necesario para especificar esta red. En primer lugar, se define la topología usada en la red, definiendo cada capa y usando diferentes técnicas de optimización en los datos. Y por último, se establecen los parámetros de entrenamiento y el posterior análisis de los resultados obtenidos.

El código es el siguiente:

Código 3.1: Definición de la red neuronal utilizando Keras.

```
model = Sequential()

#Same Layers 1
model.add( Conv2D(64,(3,3), input_shape = x.shape[1:]))
model.add(Activation("relu"))
model.add(MaxPooling2D(pool_size=(3,3)))
model.add(Dropout(0.50))


#Same Layers 2
model.add( Conv2D(64,(3,3)))
model.add(Activation("relu"))
model.add(MaxPooling2D(pool_size=(3,3)))
model.add(Dropout(0.50))

#Same Layers 3
model.add( Conv2D(64,(3,3)))
model.add(Activation("relu"))
model.add(MaxPooling2D(pool_size=(3,3)))
model.add(Dropout(0.50))

#Added one more layer
model.add( Conv2D(64,(3,3)))
model.add(Activation("relu"))


model.add(Flatten())
model.add(Dense(64))
model.add(Dense(1))
```

```
30
31 model.add(Activation("sigmoid"))
```

3.1. Topología

La red neuronal utilizada posee la siguiente estructura:

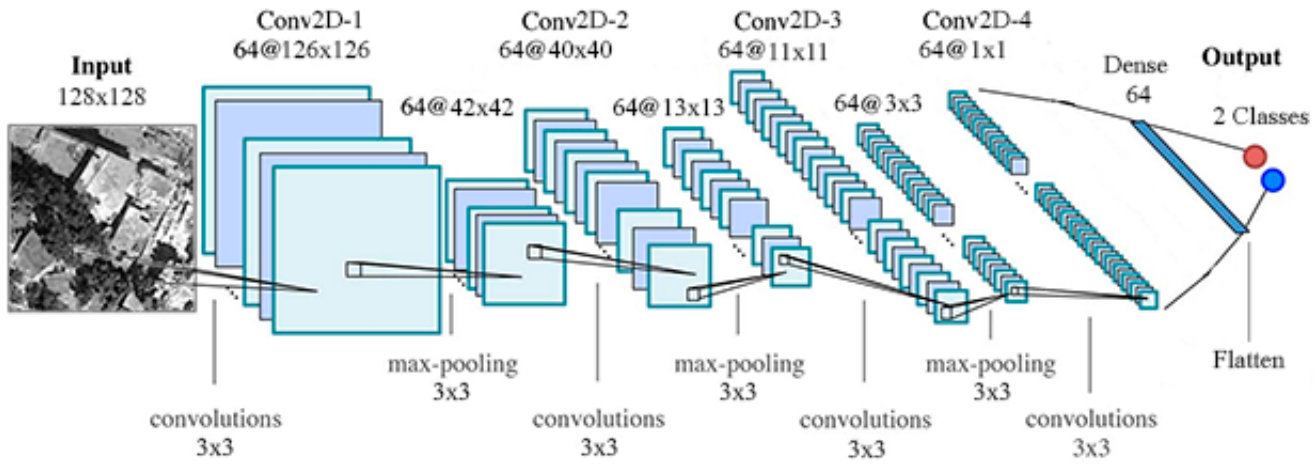


Figura 3.1: Topología del modelo utilizado.

Como muestra la representación de la red en la Figura 3.1, el modelo está formado por las siguientes capas o *layers*:

1. Capa de Entrada

Esta capa permite introducir los datos al modelo. Estos deben tener unas características idénticas en cuanto a dimensiones y forma. En este caso, se trata de imágenes de 128×128 píxeles con un único canal de color, ya que cuando se cargan se realiza una conversión de color a escala de grises (preprocesado). Como en este último caso solo se necesita un canal, reducimos la forma de la imagen. Sin embargo, esta decisión de diseño tiene otra razón mucho más importante. Además, en este caso, se reducen los valores de color entre 0-1 dividiendo entre 255 (valor máximo del canal). Todas las razones detrás de estas modificaciones se encuentran explicadas a lo largo de este capítulo.

2. Convolucion2D

Según la definición matemática, una convolución trata de transformar dos funciones f y g en una tercera función que en cierto sentido representa la magnitud en la que se superponen f y una versión trasladada e invertida de g[1].

En el ámbito de las redes neuronales y visión artificial, una convolución consiste en una imagen de entrada (matriz f) y un conjunto de matrices que sirven como filtro, cada una de ellas se define como *kernel* (matriz g) [17]. Esta técnica trata de extraer múltiples patrones dentro de la imagen, pero manteniendo la relación espacial de estos. La operación es relativamente sencilla conceptualmente hablando, consiste en la multiplicación matricial entre f y g. Normalmente, el conjunto de *kernels* a aplicar resulta ser una matriz de tamaño inferior al de la imagen para conseguir una reducción de tamaño en la imagen resultante. Por lo tanto, es necesario realizar varias operaciones para poder abarcar toda la imagen (ver Figura 3.2).

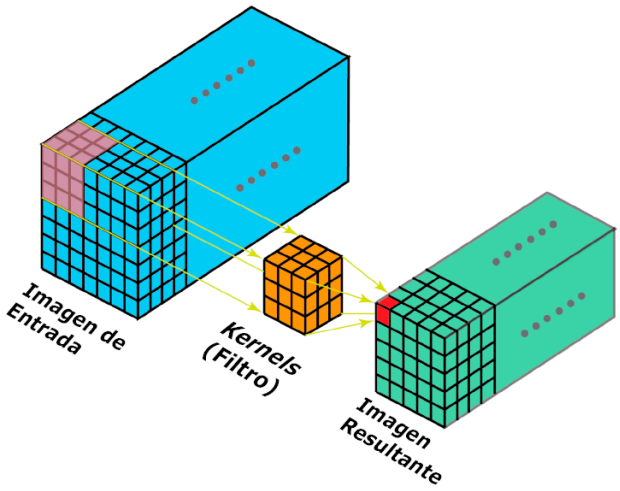


Figura 3.2: Representación de la idea de convolución de dos dimensiones.

Cada vez que se desplaza el *kernel* por el alto y ancho de la imagen de entrada, se generarán unos mapas de activación de dos dimensiones, los cuáles, proporcionan información a cada *kernel* sobre cada posición espacial. Intuitivamente, la red aprenderá patrones que se activan cuando detectan algún tipo de característica visual como la orientación del borde o una mancha de algún color en

[1] `https://en.wikipedia.org/wiki/Convolution`

la primera capa, o eventualmente patrones circulares en altas capas de la red. Normalmente, en las capas de convolución se aplican varios *kernels* en la misma, esto hace que se generen varios mapas de activación (uno por cada filtro aplicado a la imagen). Estos se apilarán a largo de la profundidad de la imagen, produciendo la imagen resultante [9].

Una de las principales características de este tipo de capas es lo que se denomina como **conectividad local**. Ya que al recibir unos datos de entrada de alta dimensionalidad (imágenes) no resulta práctico conectar los nodos de una capa con todos los de su capa anterior. Debido a que se busca conectar cada uno de ellos con solo una pequeña región de la entrada. El grado de conexión lo define un hiperparámetro denominado **campo de recepción** (representando el tamaño del filtro, es decir, el número de *kernels* a aplicar) [9].

Anteriormente se ha tratado la conectividad entre neuronas, pero no se ha mostrado el número de neuronas que hay en la salida y como se organizan (*spatial arrangement*). Mediante tres hiperparámetros se controla el tamaño de la salida:

- **Profundidad** (*depth*): Corresponde con el número de *kernels* que se van a querer usar. Por ejemplo, suponiendo que la primera capa de convolución coge como entrada una fila de la imagen, entonces diferentes neuronas a lo largo de la profundidad de esa fila se activarán en presencia de algún patrón o característica reconocible (color, formas, etc.)

- **Desplazamiento** (*stride*): Representa el número de píxeles que se deslizan al aplicar el *kernel*. A modo de ejemplo, si el desplazamiento es 1, entonces el conjunto de matrices que representan el filtro se moverá un píxel cada vez.

- ***Zero-Padding***: Alguna vez puede ser conveniente colocar ceros sobre el borde de la imagen. Este tamaño se define con este parámetro. Normalmente, su principal aplicación es preservar el alto y ancho de la imagen de entrada con respecto a la de salida.

Teniendo en cuenta la idea previa, cada neurona está asociada a un píxel, por lo que el número de neuronas necesarias será acorde al alto, ancho y profundidad de la imagen. Esto supone una gran cantidad de neuronas, haciendo que cuanto mayor sea su número, más capacidad computacional será necesaria y más tiempo se tarda en recorrer la red. Para reducir este número (excluyendo la

técnica de reducir la dimensionalidad de la imagen), se utiliza una técnica llamada ***parameter sharing***. Esta permite disminuir el número de parámetros dramáticamente haciendo una única suposición: si se encuentra una característica útil a calcular en una determinada posición espacial, entonces esta debe ser igualmente útil de calcular en otra posición diferente.

En algunos casos, esta suposición no tiene del todo sentido. Por ejemplo, cuando las imágenes no tienen unas características uniformes a identificar, es decir, que los patrones a detectar son totalmente diferentes de una zona a otra de esta. En estos casos se suele reducir el esquema de *parameter sharing* y en vez de esto, simplemente se llama a la capa **localmente conectada** (*Locally-Connected*) [9].

En la definición de esta red, se establece el filtro, es decir, el número de kernels a 64 por cada capa de convolución y la dimensión de cada uno, fijada en 3x3.

3. MaxPooling2D

La capa de *pooling* trata de reducir progresivamente la región espacial de la imagen, reducir el número de parámetros y la carga computacional de la red, y por tanto, controla el *overfitting* de la imagen.

En cuanto a su funcionamiento se debe suponer que tenemos una imagen, es decir, una matriz de 3 dimensiones (alto, ancho y los canales de color). En esta capa se define el parámetro denominado como **ventana** o *filter*, que consiste en una sub-matriz de dimensiones N*N y el desplazamiento o ***stride*** que es número de píxeles en los que se va a desplazar la ventana. Dentro de cada sub-matriz, se obtiene el valor perteneciente a esa sección de la imagen mediante diferentes técnicas: Max o Average. Cogiendo el mayor valor de la sub-matriz o haciendo una media de los valores, respectivamente. Con ello, se obtiene como salida una imagen con menores dimensiones y, por tanto, supone un menor coste computacional. Además, mantiene las características relevantes de la imagen inicial.

En el caso de este modelo, se utiliza el máximo como técnica para extraer el valor más representativo de esa celda (ventana) de 3×3, tal y como se muestra en la Figura 3.1 con el diseño del modelo. Para reforzar la explicación anterior, la Figura 3.3 muestra la idea de manera más clara.

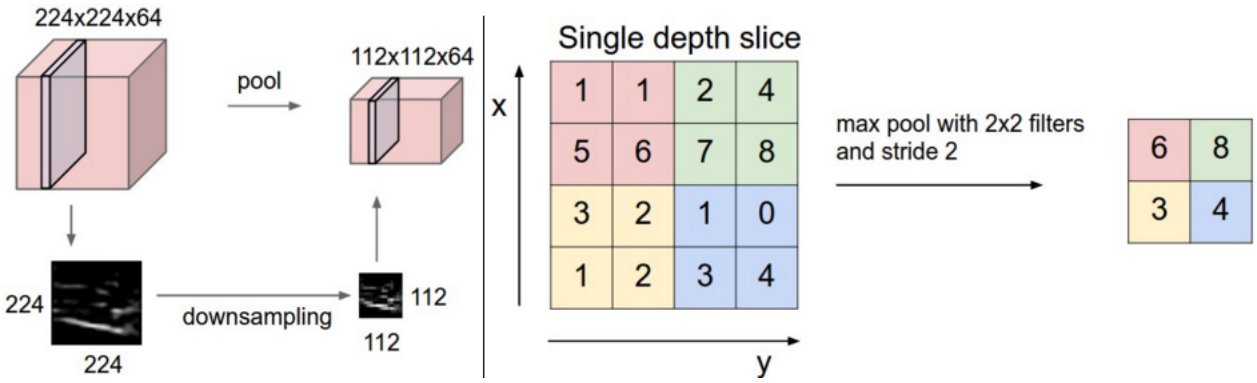


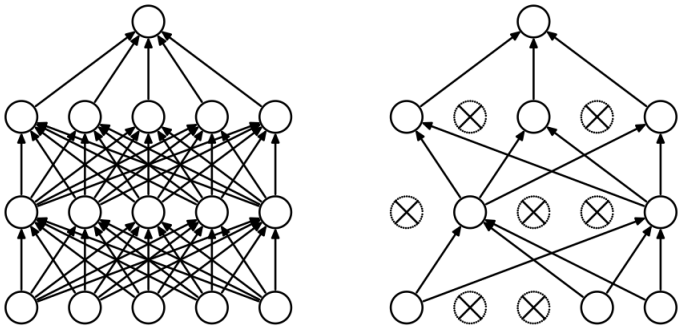
Figura 3.3: Representación del funcionamiento de la capa *MaxPooling*.

4. Dropout

Pertenece a una de las técnicas más utilizadas de regularización de datos. A modo de introducción, esta capa permite reducir el sobreajuste u *overfitting* dentro del modelo.

El mecanismo utilizado consiste en desprenderse de ciertos enlaces con la siguiente capa y así reducir la capacidad de entrenamiento para eludir ese sobreajuste del modelo (mostrado en la Figura 3.4). Debido a esto, obtener un *accuracy* demasiado alto en el entrenamiento del modelo puede no ser una buena señal, ya que es posible que el modelo esté memorizando los datos en vez de aprenderlos.

Esta técnica requiere definir el porcentaje de relaciones que van a ser desprendidas. En el caso de este modelo, este parámetro se define como el 50 % de las conexiones con la siguiente capa.



(a) Standard Neural Net (b) After applying dropout.

Figura 3.4: Representación del funcionamiento de la capa *dropout*.

5. Flatten

Consiste en transformar la forma de los datos que reciba como entrada (de cualquier dimensión) en un vector, tal y como se muestra en la Figura 3.5. Esto tiene como objetivo transformar los datos en un formato adecuado para las últimas capas del modelo (también llamada *fully-connected layers*). Esta capa no aparece en el diagrama de definición del modelo, debido a que se trata de una operación prácticamente necesaria en cualquier red neuronal de este tipo (CNN).

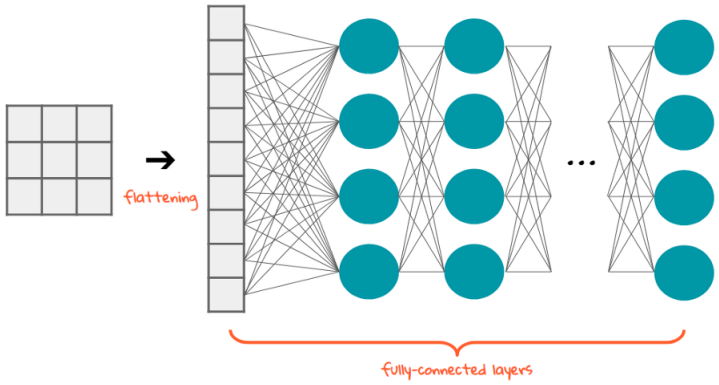


Figura 3.5: Representación del funcionamiento de la capa *flatten*.

6. Dense

Se trata de unas de las capas más simples de las descritas anteriormente junto con la de entrada. Esta se denomina como *fully-connected*, ya que todos los nodos están conectados con los siguientes. Como no realiza ninguna operación dentro de la capa, los datos resultantes de esta serán los originados por la función de transferencia. Normalmente, se aplica una función de activación en la salida de esta capa y se suele localizar como una de las últimas capas definidas en la red, principalmente debido a que permite definir el número de nodos donde se centran las conexiones de la anterior capa. De esta manera, se van reduciendo su número y, por tanto, la dimensión de la salida. Por este motivo, al final de la topología se define una capa con 64 neuronas activas y posteriormente se reduce a una para poder obtener el resultado, que va a ser un único valor comprendido entre 0 y 1.

Funciones de activación

Como se ha tratado anteriormente, las funciones de activación permiten definir la relación de los nodos en el paso entre capa y capa. En este modelo en específico se utilizan dos tipos principalmente: **ReLU** y **Sigmoid**.

La función **ReLU**[2] se utiliza como salida para la capa de convolución. De esta manera, nos permite quedarnos con las neuronas que proporcionan un cierto valor en la predicción del modelo. Sus principales características son: los valores obtenidos son siempre no negativos (comprendidos entre [0,∞]), evitando que los errores se propaguen por toda la red (**Representación positiva**); los valores de entrada y salida son idénticos, siempre que la entrada sea positiva (**linealidad**) y resulta relativamente sencilla de implementar.

Por otro lado, también es utilizada la función **Sigmoid**[3] como función de activación en las últimas capas del modelo (*Dense*). Esta permite: proporcionar una **salida comprendida** entre [0,1], siendo esencial para los modelos de clasificación con solo dos categorías e introducir **no-linealidad**, generando una salida muy significativa cuando la entrada es muy pequeña [-2,2].

En el caso de utilizar un algoritmo de multi-clasificación (con más de dos categorías), es recomendable usar la función *softmax*.[4]

3.2. Diferentes técnicas aplicadas a los datos

En el ámbito del ML se utilizan diversas técnicas para poder "adaptar" los datos disponibles con el objetivo de facilitar el aprendizaje al modelo elegido o diseñado. En algunos casos, consiste únicamente en agrupar los datos sin perder la integridad de estos con respecto al modelo. Pero por otro lado, puede surgir la necesidad de modificarlos totalmente debido a que se quiera obtener una característica concreta de ellos.

Esta modificación de los datos suele surgir a partir de las diferentes necesidades que pueden aparecer durante el desarrollo. Puede darse el caso que el conjunto de datos sea demasiado pequeño, por lo que puede ser interesante generar más datos a partir de los existentes de alguna manera

[2] https://medium.com/@danqing/a-practical-guide-to-relu-b83ca804f1f7
[3] https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6
[4] https://cs231n.github.io/linear-classify/

(sección 3.2.3, argumentación). Otra casuística relacionada con la explicada anteriormente, podría ser modificar los datos para mejorar la interpretación de estos por parte de la red (3.2.2, normalización). Por otro lado, puede ser debido a que se necesitan ajustar diversas características del modelo (sección 3.2.1, regularización). Como se puede ver hay una infinidad de razones por las que utilizar este tipo de técnicas, pero siempre están basadas en una necesidad existente o un aspecto a mejorar.

En esta sección, vamos a intentar de tratar las diversas técnicas que han sido utilizadas en el desarrollo y posterior despliegue del modelo, sin olvidar algunas de ellas que no se han usado en la solución final, pero que en algún momento del desarrollo han sido contempladas como una posible técnica a usar.

3.2.1. Regularización

Esta técnica se utiliza para evitar el *overfitting* del modelo, con el fin de que la red aprenda en vez de memorizar. Esta diferencia resulta muy importante, ya que el objetivo es conseguir que la red proporcione unos resultados en base a las relaciones que ha ido generado con su entrenamiento. Por otro lado, si la red memoriza estas relaciones, los resultados solo van a ser adecuados cuando la entrada pertenece a unos datos con los que ya ha sido entrenada. Esto hace que no se cumpla el objetivo anteriormente propuesto y, debido a esto, quita el factor que hace especialmente interesante a las redes neuronales, su capacidad de adaptación a distintos datos de entrada (siempre con una naturaleza común con respecto a los que ha sido entrenada) y el nivel de precisión en la predicción sobre estos.

En este caso, la **regularización** consiste en mejorar la generalización de la red. Para ello, debemos reducir la capacidad de aprendizaje durante el proceso de entrenamiento. Aunque el problema sea el *overfitting* o sobre ajuste, también puede ser un problema el caso inverso, *underfitting*, haciendo que la red generalice demasiado y no se ajuste a los datos introducidos. Esto hace que el porcentaje de resultados "incorrectos" aumente.

Como se puede observar en la Figura 3.6, el punto en el que esta técnica es realmente útil es cuando la red es específica para el problema y datos propuestos. Pero también debe ser lo

suficientemente general como para poder adaptarse y generar unos buenos resultados en base a unos nuevos datos.

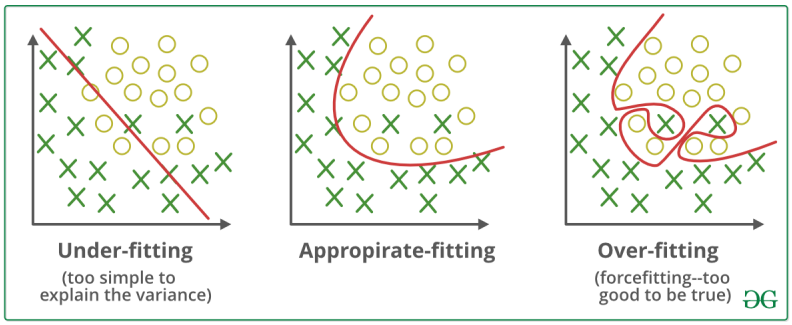


Figura 3.6: Posibles resultados obtenidos al utilizar la técnica de regularización.

Un ejemplo de la aplicación de este tipo de técnicas sería la capa *Dropout*, que forma parte de la topología del modelo utilizado.

3.2.2. Normalización

En visión artificial, la **normalización** de imágenes tiene el objetivo de simplificar y/o cambiar la representación de la imagen en algo más entendible y sencillo de analizar. Este tipo de técnicas se denominan de preprocesado, ya que se realizan antes de introducir los datos al modelo para su posterior entrenamiento con ellos, es decir, la preparación de estos. De esta forma, se busca extraer las características más importantes de los datos y reducir la complejidad con el objetivo de obtener la mejor actuación del posterior modelo acorde a la solución planteada.

Centrando la atención a lo que este proyecto se refiere, se han utilizado diversos métodos relacionados con esta técnica. En primer lugar, hay que analizar las imágenes disponibles y asegurarse de que todas tienen unas ciertas características comunes: relación de aspecto, color, perspectiva de la imagen, etc. En este caso, se trata de imágenes de 128×128 píxeles a color (con 3 canales RGB por píxel) con una perspectiva cenital, es decir, vista perpendicular al suelo. Como todas ellas siguen las mismas características, esto quiere decir que ya ha habido un trabajo previo de preprocesado para unificar la forma e información de cada una de ellas. Un ejemplo de estas se encuentra en la Figura 3.7.

(a) Zona dañada. (b) Zona No Dañada.

Figura 3.7: Ejemplos extraídos del dataset adquirido por el satélite de observación terrestre de alta resolución GeoEye-1 durante el terremoto ocurrido en Haití en 2010.

Al cerciorarse de que todas las imágenes poseen unos rasgos esenciales comunes, se puede empezar a pensar las acciones a aplicar en estas para obtener información que resulte relevante para el propósito u objetivo inicial que se planteé. Un aspecto a tratar es el impacto que el color de la imagen tiene con respecto a la solución, es decir, cuanta relevancia tienen los distintos matices que los tres canales nos pueden ofrecer. En principio la respuesta es muy poco, ya que los rangos de color no determinan si lo está o lo dañada que está una zona. Además, no se busca conocer cuánto de dañada se encuentra una zona específica de la imagen, se trata de obtener si esta muestra alguna zona dañada. Por lo que, el objetivo principal es obtener la mayor distinción posible entre lo que presumiblemente sea una zona deteriorada y la que no. Para ello, se transforman las imágenes a escala de grises con el objetivo anteriormente mencionado. Este hecho tiene otras implicaciones que también resultan importantes, la principal de ellas es que los canales pasan de ser 3 a uno único. Esto repercute en que la imagen pasará de tener unas dimensiones de 128(ancho)×128(alto)×3(canales) a 128(ancho)×128(alto)×1(canal), provocando que el coste computacional sea mucho menor, debido a que la cantidad de píxeles a procesar es inferior (esta idea se ve reflejada en las Figuras 3.8a y 3.8b).

(a) Color. **(b)** Escala de grises. **(c)** Pseudocolor.

Figura 3.8: Aplicación de una técnica de normalización a los datos usados.

La reducción de estos canales supone que la cantidad de información obtenida por cada píxel es menor, pero por otro lado, esta no resultaba relevante para el problema propuesto. Hay que recordar que los canales RGB representan los tres colores primarios y cada uno de ellos posee como valor máximo 255, es decir, que el valor de cada canal está comprendido entre 0 y 255. Esto hace que al haber ahora un único canal tras las últimas modificaciones realizadas, cada píxel tendrá un único valor. Por lo que el rango está entre [0,255], provocando que el modelo a la hora de entrenar con estos datos tenga que formar relaciones con estos valores ciertamente dispersos. Con lo cual, se realiza una nueva modificación a la imagen que consiste en dividir el valor de cada píxel con el valor máximo posible (255). Esto provoca que el rango de valores posible sea entre 0 y 1, agrupando los datos y facilitando un mejor aprendizaje por parte de la NN.

3.2.3. Argumentación

Este método está más relacionado con el tratamiento de imágenes y el uso de estas en *Deep Learning*. Pertenece a un conjunto de técnicas de preprocesado de datos al igual que la descrita en el apartado anterior. En este proyecto no se ha usado esta técnica, aunque sí fue probada. No se utilizó debido a que los resultados obtenidos no fueron satisfactorios, ya que no aportaban ninguna mejora en la actuación del modelo.

Este concepto se utiliza principalmente cuando se requiere extender un conjunto de datos ya existente, en este caso, imágenes. Esto puede ser en base a que el comportamiento del modelo

no es el esperado o que simplemente se quiere mejorar la precisión del modelo añadiendo mayor conjunto de imágenes de entrada o teniendo un mayor set de imágenes de prueba. Para ello y mediante diversas librerías que implementan esta funcionalidad (como Keras, Pytorch, etc.), es posible generar réplicas de la misma imagen, pero con ciertas modificaciones en cada una de ellas. Por ejemplo de una imagen se puede obtener su réplica rotada 120°, aplicando un cierto nivel de zoom a una zona, etc. Esto hace que de imágenes prácticamente iguales en contenido (con algunas modificaciones), se extraiga cierta información que pueda resultar útil al modelo, por ejemplo, la localización espacial de un determinado objeto. Por lo que si se dispone de la misma imagen, pero rotada de diferentes maneras (ver Figura 3.9), se obtendrá posiblemente más información espacial sobre las características a identificar.

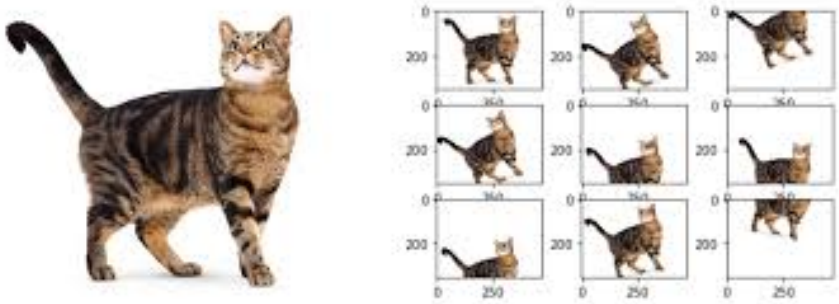


Figura 3.9: Ejemplo de argumentación de datos.

3.3. Entrenamiento

Una vez diseñado y, por tanto, definido el modelo, es el momento de entrenarlo para poder ver la actuación de este sobre los datos disponibles. Aunque conceptualmente la red neuronal sea impecable a nivel de diseño, hasta que no es probada con los datos no se puede deducir lo bien que se ajusta a estos y a la predicción que se espera obtener. Por lo tanto, se vuelve un proceso cíclico donde se realiza el entrenamiento del modelo, se analizan las métricas que definen la actuación del modelo y en base a esta información, se reajustan los parámetros de entrenamiento o el diseño del modelo para obtener unos mejores resultados.

Por consiguiente y siguiendo los pasos anteriormente mencionados, sería el momento de efectuar el inicio del proceso de aprendizaje de la red. Sin embargo, es necesario definir previamente los

parámetros que definen el modo de realización de este paso, con el objetivo de maximizar la precisión y minimizar el tiempo de entrenamiento.

3.3.1. Hiperparámetros

El concepto de **hiperparámetros** se refiere a las variables que se utilizan en el propio proceso de instanciación de un modelo. Se puede definir como el conjunto de configuraciones necesarias (o no) en cada capa para poder describir una red neuronal y su entrenamiento. En este documento se utilizarán ambos términos indistintamente, pero se refieren al mismo concepto. Para poder adentrarse en toda la parametrización relacionada con el entrenamiento del modelo, hay que conocer en primer lugar las líneas de código que lo definen:

Código 3.2: Definición de la fase de entrenamiento.

```
#Compile stage
model.compile(loss="binary_crossentropy", optimizer= "adam", metrics=['accuracy'])
#Fitting stage
history=model.fit(x,y, batch_size=32, epochs=200, validation_split=0.2)
```

Teniendo en cuenta la definición del entrenamiento, hay que distinguir dos fases:

1. **Compilación:** Antes de entrenar el modelo, se diseña el proceso posterior de aprendizaje. En este hay que definir:

   - **Optimizador:** Se usa a la hora de entrenar la red neuronal para optimizar la función de coste $\boldsymbol{J}$. El valor de esta (función $\boldsymbol{J}$) es la media de la función de pérdida entre el valor predicho $\boldsymbol{y\prime}$ y el valor actual $\boldsymbol{y}$. El valor $\boldsymbol{y\prime}$ se obtiene durante el proceso de propagación hacia adelante y hace uso de los pesos $\boldsymbol{W}$ y biases $\boldsymbol{b}$ de la red. Por tanto, con la ayuda de los algoritmos de optimización, minimizamos el valor de la función de coste $\boldsymbol{J}$ actualizando las variables de entrenamiento $\boldsymbol{W}$ y $\boldsymbol{b}$ [18].

     $$J(W,b) = \frac{1}{m}\sum_{i=1}^{M} L(y'^{i}, y^{i}) \tag{3.1}$$

     Hay diversas funciones de este tipo como *Momentum*, *Gradient Descent*, *RMS*, etc. En este caso, la función utilizada en el modelo se denomina **Adam** (*Adaptative Moment Estimation*). Esta combina dos funciones *RMSprop* y *Stochastic Gradient Descent* con

Momentum. Permite tasas de aprendizaje adaptativo por cada parámetro, esto hace que sea posible controlar como de rápido el modelo se adapta a nuestro problema. Por tanto, cuanto mayor sea este valor, se requerirá un mayor número de *epochs* (explicado a continuación) de entrenamiento proporcionando pequeños cambios en los pesos en cada actualización. Una tasa muy grande puede causar que el modelo converja demasiado rápido hacia una solución menos óptima, mientras que si es demasiado pequeña puede hacer que el proceso se estanque [19], es decir, que mantenga el mismo nivel de precisión (*accuracy*) entre los datos y la NN.

- **Función de pérdida:** Es la función que se trata de minimizar con la función de optimización (en este caso Adam). Siguiendo la notación anteriormente propuesta, esta función representaría la función de coste J, cuyo objetivo es minimizarla. En relación con este concepto de función de coste, habría que definir la idea de *loss* (L), que representa la diferencia entre el valor predicho $y\prime$ y el valor real y.

  Tal y como se puede observar en el Código 3.2, la función de pérdida usada es ***Binary Cross-Entropy***.

  Hay que recordar que al ser binaria solo hay dos posibles soluciones: 0 ó 1. El principal objetivo de esta función es comparar como de bien es la salida de la función *Sigmoid* (función de activación de la última capa) con las dos posibles categorías disponibles (0 (dañado) ó 1 (no dañado)). Esta idea queda representada en la siguiente ecuación:

$$J(w) = \frac{1}{N}\sum_{n=1}^{N} H(p_n, q_n) = -\frac{1}{N}\sum_{n=1}^{N}[y_n \log \check{y}_n + (1 - y_n)\log(1 - \check{y}_n)] \tag{3.2}$$

  Por lo que si el valor $\boldsymbol{y} = 0$, tendremos que el primer término de la suma dentro del sumatorio será cero, por lo que lo único que se tendría que calcular es la segunda parte. Además ya se sabe que (1-y) = 1, por lo que solo quedaría calcular el logaritmo. Por otra parte, si $\boldsymbol{y} = 1$, el segundo término del sumatorio es cero, por tanto, el primero solo tomaría efecto [20].

- **Lista de métricas**: Se definen los aspectos del modelo que se deben evaluar durante el proceso de entrenamiento y *testing*:

- Precisión o *Accuracy*: Representa la fracción de predicciones que el modelo realizó correctamente.

$$Accuracy = \frac{N°correctas}{N°totales} \tag{3.3}$$

En clasificación binaria, la exactitud también se puede calcular en términos de positivos o negativos de la siguiente manera [21]:

$$Accuracy = \frac{VP + VN}{VP + VN + FP + FN} \tag{3.4}$$

Donde VP = Verdaderos positivos, VN = Verdaderos negativos, FP = Falsos positivos y FN = Falsos negativos.

Una matriz de confusión representa esta idea y sirve para analizar la *accuracy* del modelo en este tipo de problemas.

- Pérdida o *Loss*: Indica cuanto de incorrecta ha sido la predicción del modelo en un solo ejemplo. Si la predicción del modelo es perfecta, la pérdida es cero, de lo contrario, el valor es mayor. El objetivo es encontrar un conjunto de ponderaciones y ordenadas al origen, que en promedio tengan pérdidas bajas en la mayoría de los ejemplos [22].

2. **Ajuste**

- **Entrada y salida (x e y):** Representan los datos de entrada $\boldsymbol{X}$, en nuestro caso serian 268 imágenes de 128 píxeles de alto y 128 de ancho, cada píxel con un canal de color. Por otro lado, $\boldsymbol{y}$ representaría la categoría asociada a cada imagen, es decir, el resultado correcto que posteriormente se coteja con la predicción.
- **Tamaño de *Batch*:** Define el número de datos que se van a utilizar cada vez que el modelo actualiza sus parámetros internos.
- ***Epochs*:** Especifica el número de veces en el que el algoritmo de aprendizaje va a trabajar con todo el dataset.
- **Separación de validación:** Permite seleccionar (aleatoriamente o no, según se defina) un porcentaje de los datos de entrada que únicamente se usarán para testear el modelo

durante su entrenamiento. Esto quiere decir que estos prueban el comportamiento del modelo y el conjunto restante de datos servirán para el entrenamiento de la red. De esta manera, evitamos el *overfitting* del modelo y nos permiten analizar la actuación de este en un entorno "real", es decir, con datos que previamente no han sido utilizados por el modelo.

3.4. Optimizaciones y Mejoras Aplicadas

En este apartado se van a tratar los diferentes cambios/mejoras que han sido realizadas sobre el modelo inicial (fragmento de código en el apartado 3.1). Algunas de ellas han sido desarrolladas con el objetivo de mejorar el rendimiento del modelo, aunque otras han sido realizadas debido a una serie de limitaciones generadas con los pasos posteriores a la obtención del modelo.

Teniendo en cuenta las modificaciones realizadas, la especificación del modelo resultante ha sido la siguiente (incluyendo la fase de definición y entrenamiento):

Código 3.3: Definición de la red neuronal y entrenamiento con las modificaciones realizadas.

```
#Fase de definición de la topología del modelo
def create_model(x,name):
  tf.keras.backend.clear_session()

  inputs = Input(x.shape[1:])

  #Layer 1
  layer = Conv2D(64,(3,3),activation='relu')(inputs)
  layer = MaxPooling2D(pool_size=(3,3))(layer)
  layer = Dropout(0.5)(layer)

  #Layer 2
  layer = Conv2D(64,(3,3),activation='relu')(layer)
  layer = MaxPooling2D(pool_size=(3,3))(layer)
  layer = Dropout(0.5)(layer)

  #Layer 3
  layer = Conv2D(64,(3,3),activation='relu')(layer)
  layer = MaxPooling2D(pool_size=(3,3))(layer)
  layer = Dropout(0.5)(layer)

  #Layer 4
  layer = Conv2D(64,(3,3),activation='relu')(layer)
  #layer = Activation('relu')(layer)

  layer = Flatten()(layer)
  layer = Dense(64)(layer)
  #layer = Activation('relu')(layer)

  output = Dense(1,activation='sigmoid')(layer)
  return Model(inputs=inputs,outputs=output,name=name)


model = create_model(x)
```

```
op = Adam(learning_rate=0.0003)
model.compile(loss=tf.keras.losses.BinaryCrossentropy(),optimizer= op, metrics=['accuracy',tf.keras.metrics.
    ↪ BinaryCrossentropy()])



#Fase de especificación de los hiperparámetros para el entrenamiento de la red
#TensorBoard
tensorboard = TensorBoard(log_dir='logs/{}'.format(name_file), write_graph=True, write_images=True,
                          histogram_freq=1,profile_batch=3)
#Model Checkpoint (callbacks)
checkpoint = ModelCheckpoint(filepath,verbose=1,monitor='accuracy',save_best_only = True, mode='max')
callbacks_list = [tensorboard,checkpoint]

history=model.fit(x=x,y=y, batch_size=32,epochs=200,validation_split=0.1,shuffle=True,callbacks=callbacks_list)
```

Como se puede apreciar en el Código 3.3, la **estructura de definición del modelo** cambia ligeramente con la inicial, descrita al principio de este capítulo (Código 3.1). Aunque, en cuanto a contenido ambas definen lo mismo, en la primera de ellas se define el modelo como *Sequential*. Esto permite especificar las capas que conforman la red como un *buffer*, donde estas se van añadiendo de manera consecutiva, resultando su especificación mucho más sencilla. Por lo tanto, la modificación entre ambas definiciones viene dada por un error a la hora de realizar la conversión del modelo a este mismo serializado (explicado en la sección 4.1.1), provocando que la definición de la topología deba ser "manual". Este paso resulta necesario para poder integrar ese modelo en OpenVINO.

La topología del modelo se ha mantenido intacta durante las modificaciones y no ha habido cambios sustanciales en esta parte de definición de la red. Aunque en **la definición del entrenamiento** es donde se encuentran la mayor parte de ellas. Comenzando en la etapa de compilación del entrenamiento, donde se establece **la capacidad de aprendizaje del optimizador** en un valor inferior (0.0003) al por defecto (0.001). El propósito de este es conseguir reducir el ruido obtenido al analizar las gráficas de precisión y pérdida, pero sin producir un descenso muy alarmante en la precisión de la red y, por tanto, obtener una curva más "suave" en las gráficas del entrenamiento.

Continuando con la fase de entrenamiento, es necesario introducir el concepto de ***callback***. Esta se define como un conjunto de funciones que se pueden establecer para ser ejecutadas con una determinada periodicidad durante el proceso de *training*. De esta manera, podemos obtener información sobre los estados internos de este proceso. En este caso específico, han sido utilizadas dos funciones con diferentes propósitos:

- *TensorBoard*: Permite obtener un reporte del proceso de entrenamiento del modelo y almacenarlo en una serie de ficheros. Por tanto, tras finalizar el proceso de entrenamiento, es posible analizar la actuación del modelo de manera más precisa.

- *ModelCheckpoint*: Almacena una captura del modelo al finalizar un determinado *epoch* durante la fase de entrenamiento. El momento en que se efectúa esa captura depende de los parámetros definidos. Ciñéndonos a los cambios realizados, se establece que cuando la métrica de precisión (*accuracy*) sea la más alta, se realice una captura de los pesos de la red. También, se podría plantear a la inversa, en el que esa captura se realice cuando se obtenga el valor de pérdida más bajo hasta el momento.

En contrapartida, la utilización de esta serie de llamadas (*callbacks*) en el proceso de entrenamiento provoca que el tiempo de ejecución sea mayor, debido a la generación y almacenamiento de ese conjunto de reportes definidos.

En añadido a esta etapa donde se establecen los hiperparámetros pertenecientes a la fase de ajuste, se incorpora una nueva funcionalidad que consiste en mezclar el orden de los datos de entrada (parámetro ***shuffle***), obteniendo un contexto más "real" durante el ajuste del modelo. Esto es debido a que si los datos se encuentran agrupados en sus respectivas categorías, pueden almacenarse en las neuronas relaciones no tan correctas, ya que durante un periodo del entrenamiento, los datos pertenecerán exclusivamente a una categoría y a continuación, a la otra. Sin embargo, si se mezclan todos los datos, el proceso se ceñirá lo más posible a lo que posteriormente se le requerirá al modelo cuando sea implementado. Haciendo que esta práctica sea comúnmente aplicada y muy recomendada a la hora de definir los parámetros de ajuste de la red.

Capítulo 4

Portabilidad e Inferencia en dispositivos Intel

Tras finalizar las fases de entrenamiento y prueba (las fases de aprendizaje de la red), el modelo obtenido se acerca lo más posible al comportamiento esperado, por lo que llega el momento de implementarlo. Esta es una de las características principales que ofrece esta herramienta, permitiendo el despliegue del modelo en un entorno de producción y así, poder realizar las inferencias obteniendo el mayor rendimiento por parte del hardware de Intel.

En esta sección, se describe la metodología y los pasos realizados para esta solución en concreto, yendo al detalle de cada una de las partes. Es recomendable revisar previamente el capítulo 2.3.2, que ofrece una vista preliminar del propósito y herramientas integradas en el *toolkit*.

4.1. Metodología

El flujo de ejecución establecido (ver Figura 4.1) posee una cierta complejidad, ya que esta herramienta esta pensada para utilizar modelos ya predefinidos e integrarlos en tu casuística. Sin embargo, no es el caso de este proyecto, por lo que se requiere una serie de procedimientos por parte de *OpenVino* para poder compatibilizar e integrar el modelo y, a continuación, generar una solución.

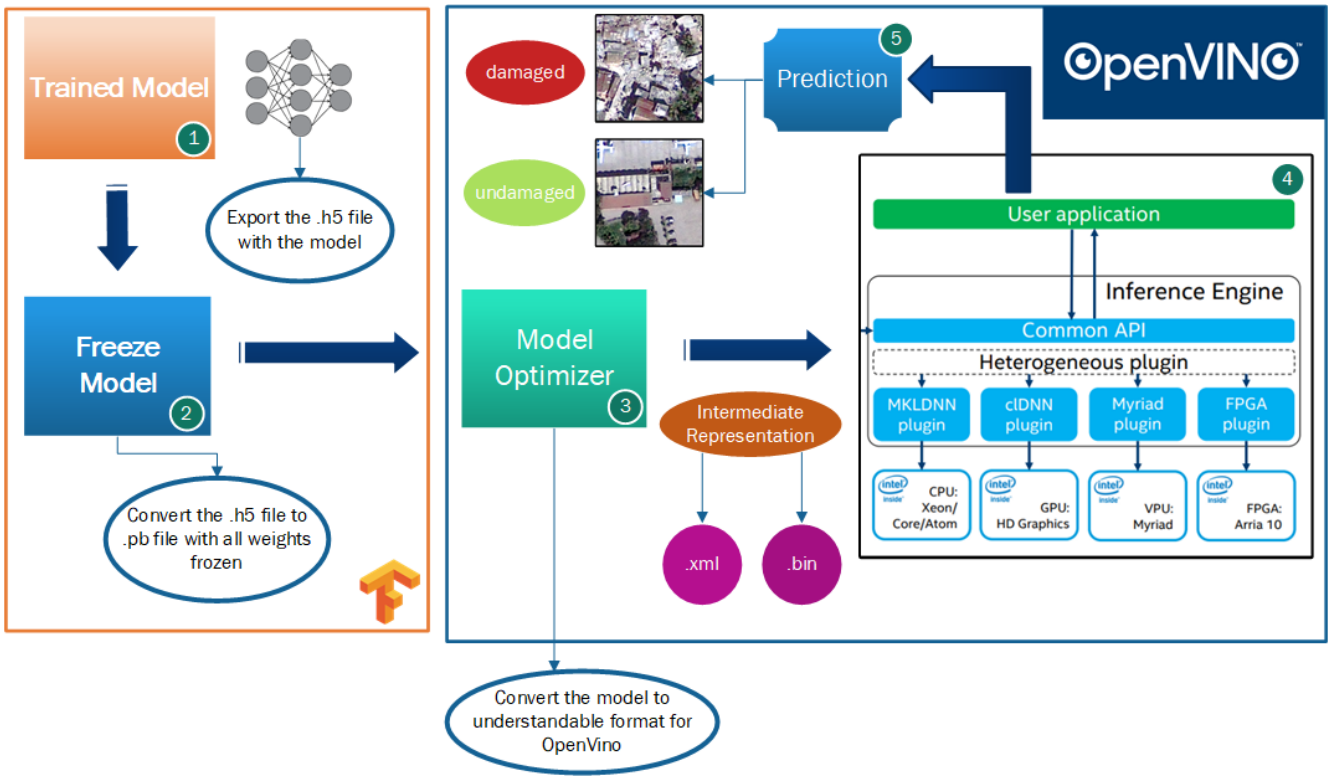


Figura 4.1: Metodología definida en la portabilidad del modelo y realización de las inferencias.

Todo el proceso queda recogido en el diagrama que aparece en la Figura 4.1, en el cuál, se recogen las siguientes fases:

1. **Obtener** la red neuronal entrenada con un rendimiento adecuado y **exportarla** en un fichero, formato .h5 (keras). Este proceso queda recogido en el capítulo 3.

2. **Serializar** el grafo que define el modelo. Esto, también, se denomina como **congelar** el modelo y se expone a continuación en el apartado 4.1.1.

3. **Transformar** la red neuronal congelada a un **formato compatible** para el *toolkit* de OpenVINO, para ello se utiliza una herramienta proporcionada por este, llamada *Model Optimizer*. De ella, obtenemos dos ficheros:

   - **.xml:** Describe la topología de la red.
   - **.bin:** Almacena los valores constantes (como los pesos, parámetros,etc.) en formato binario.

4. Se realiza la **inferencia** usando el motor proporcionado por OpenVino. Por lo tanto, es necesario especificar el modelo que se va a usar, los datos (imagen o conjunto de imágenes) y el dispositivo, para poder realizar la predicción.

5. **Obtener** la predicción en base a la imagen introducida, identificando si la imagen presenta zonas **dañadas** (*damaged*) o, por el contrario, **no dañadas** (*undamaged*).

En el anexo C, se trata de manera superficial el proceso de inferencia utilizando vídeo como dato de entrada.

4.1.1. Serialización del Modelo

El proceso de **serialización** (segundo paso definido en la metodología 4.1) consiste en aglutinar en un solo archivo la información procedente del grafo y de los parámetros guardados durante el proceso de entrenamiento, almacenando estos últimos como constantes dentro de la estructura del grafo. Esto genera que ciertos datos adicionales se pierdan, como por ejemplo, los gradientes en cada punto, los cuales, permiten que la red se pueda volver a cargar y entrenar desde el punto donde se encontraba. Aunque, cuando se plantea que una red sea implementada en un sistema de inferencias, quiere decir que se encuentra en lo que se considera como un entorno de producción, por lo que este último dato no es necesario que sea almacenado [23].

Este proceso queda definido en el siguiente fragmento de código:

Código 4.1: Serialización del modelo.

```
import tensorflow as tf
import argparse
import uuid
import tensorflow.compat.v1.keras.backend as K
import tensorflow.keras as keras
import shutil
import os
import subprocess

parser = argparse.ArgumentParser(description="Convert Keras .h5 file to Tensorflow .pb frozen model.")
parser.add_argument("--h5", type=str, default="model.h5", help="Keras .h5 file (input)")
parser.add_argument("--pb", type=str, default="model.pb", help="Tensorflow frozen model (output)")
args = parser.parse_args()

#Establece la fase de aprendizaje en modo testing (0)
K.set_learning_phase(0)
#Carga el modelo que se desea serializar
model = keras.models.load_model(args.h5)
#Obtiene el nombre de la última capa del modelo cargado
model_output = model.output.op.name

```

```
tempdir = str(uuid.uuid4())
os.mkdir(tempdir)

try:

    temp_tf_file = str(os.path.join(tempdir, "tf.ckpt"))
    saver = tf.compat.v1.train.Saver()
    #Almacena la sesión actual al archivo temporal
    saver.save(K.get_session(), temp_tf_file)
    tmp_meta_file = str(temp_tf_file) + ".meta"

    #Almacena el grafo del modelo en el archivo temporal
    saver = tf.compat.v1.train.import_meta_graph(tmp_meta_file, clear_devices=True)

    #Reinicia las variables del modelo
    K.get_session().run(tf.compat.v1.global_variables_initializer())
    K.get_session().run(tf.compat.v1.local_variables_initializer())
    sess = K.get_session()

    #Importa las variables del ultimo punto de guardado del modelo a la sesión
    saver.restore(sess, tf.train.latest_checkpoint(os.path.expanduser(tempdir)))

    output_node_names = model_output

    #Cambio de nombre y eliminar algún un atributo de ciertos nodos para solventar un bug relacionado
    #con batch normalization
    gd = sess.graph.as_graph_def()
    for node in gd.node:
        if node.op == 'RefSwitch':
            node.op = 'Switch'
            for index in xrange(len(node.input)):
                if 'moving_' in node.input[index]:
                    node.input[index] = node.input[index] + '/read'
        elif node.op == 'AssignSub':
            node.op = 'Sub'
            if 'use_locking' in node.attr: del node.attr['use_locking']
        elif node.op == 'AssignAdd':
            node.op = 'Add'
            if 'use_locking' in node.attr: del node.attr['use_locking']

    #Convierte las variables anteriormente importadas en valores constantes
    converted_graph_def = tf.compat.v1.graph_util.convert_variables_to_constants(sess, gd, output_node_names.split(",")↩
        ↪ ))
    #Exporta el modelo serializado
    tf.io.write_graph(converted_graph_def, ".", args.pb, as_text=False)
    print("Done.")
finally:
    shutil.rmtree(tempdir)
```

Este fragmento de Código 4.1 recibe la ruta del archivo de definición del modelo (*–h5*) y exporta en la ruta deseada el modelo "congelado"(*–pb*). Esta diseñado únicamente para ser ejecutado con la versión **1.15** de *Tensorflow*, debido a que se utilizan ciertas funciones no soportadas por las versiones más recientes. Además, hay una parte (línea 46-60) que únicamente trata de solventar algunos problemas relacionados con los nombres de ciertas capas y atributos de estas, en base a un *bug* con la normalización de los lotes (*batch normalization*).

Con las últimas versiones, se está dejando de dar soporte a este tipo de procedimientos. Según

los últimos productos lanzados por Google, su intención es integrar toda la parte de portabilidad e inferencia en un mismo ecosistema, *Tensorflow Extended*.

4.1.2. Transformación del modelo para su uso con OpenVINO

Representa la tercera fase de la metodología definida en la sección 4.1. En ella, utilizando una de las herramientas principales del *toolkit*, es posible obtener el modelo en el formato soportado por OpenVINO. Para ello, resulta tan sencillo como ejecutar el siguiente comando con la parametrización necesaria:

Código 4.2: *Model Optimizer* para Tensorflow.

```
python3 /opt/intel/openvino/deployment_tools/model_optimizer/mo_tf.py -m <model-path> -o <output-dir> -n <name-of-the-output_model> -s 255.0 --data_type FP16 --input_shape [1,128,128,1]
```

Esta herramienta permite personalizar, mediante el paso de diferentes parámetros (ver código 4.2), el modelo resultante de la conversión (*Intermediate Representation* (IR)). Estas modificaciones pueden resultar muy útiles para la casuística en la que vaya a ser empleada la red o a modo de optimización de la inicial. En lo referido a las necesidades requeridas por este proyecto hay varios detalles al respecto a destacar:

- Permite aplicar una **operación de división** a los datos de entrada (una especie de preprocesado), añadiendo esta característica a la definición del modelo. Mediante el parámetro "-s", se puede definir ese valor, permitiendo optimizar el rendimiento del modelo. En relación con este trabajo, es necesario realizar ese proceso de normalización, por lo tanto, este valor se establece a 255.0.

- Definir la **dimensionalidad de los datos de entrada del modelo** (parámetro *–input_shape*). De esta manera, se puede definir cualquier forma que la topología de la red pueda soportar. Aunque, se puede definir este valor especificando el tamaño de lotes por defecto soportado por el modelo.

  Este *flag* resulta necesario en la mayoría de casos, ya que el modelo obtenido en la fase de entrenamiento posee un parámetro no definido, el número de datos de entrada o de imágenes

en este caso. De tal forma que la entrada de la red queda de esta manera: [?,128,128,1]. Esto hace que haya que definir un valor concreto en las dimensiones correspondientes a los datos de entrada para la generación de la IR. Se puede efectuar de dos maneras: definiendo un tamaño de lote, el cual, establecerá el número de imágenes de entrada o introducir unas dimensiones concretas.

- Por defecto, la herramienta identifica todo dato de entrada con la siguiente **capa** que representa el propósito de cada dimensión de la imagen, **NCHW**: Número, Canales, *Height* (Alto) y *Weight* (Ancho). En la fase de entrenamiento, la imagen se trata con la capa NHWC, cambiando el orden de este con respecto al *toolkit*. Esta herramienta te permite deshabilitar esa conversión de NHWC a NCHW, sin embargo, no es muy recomendable, debido a que esa convención permite un mejor rendimiento por parte del motor de inferencias.

4.1.3. Flujo de ejecución de las inferencias

Tras obtener un modelo funcional y transformado al formato que el *toolkit* requiere (IR), llega el momento de realizar la implementación de este en el ecosistema de OpenVINO, establecido como el cuarto procedimiento de la metodología (apartado 4.1). Para ello, hay una serie de pautas que definen de manera abstracta el esquema principal del código a desarrollar (ver Figura 4.2). Con el fin de ahondar en esa estructura, se van a concretar los fragmentos de la definición de código que corresponde a cada parte:

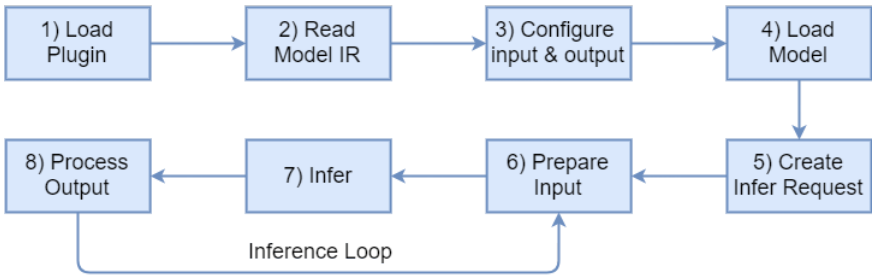


Figura 4.2: Diagrama resumen del flujo de ejecución del proceso de inferencia.

En primer término (ver *Código* 4.3), se instancia el objeto principal del motor de inferencias ***IECore***, mediante el cual, se produce la interacción con la API de OpenVINO. A continuación, se

definen (o no necesariamente) una **serie de complementos** (*plugins*) disponibles en el *toolkit* con el fin de optimizar la actuación del motor con ciertas operaciones en base a una serie de dispositivos. Esta característica inicial esta desarrollada especialmente para su uso con CPUs. Tras finalizar este paso de optimización inicial, se produce la **lectura de la red neuronal** (IR), permitiendo obtener datos de esta.

Código 4.3: Paso 1-3 del flujo de ejecución (Figura 4.2).

```
#Administración de los parametros
args = build_argparser().parse_args()
model_xml = args.model
model_bin = os.path.splitext(model_xml)[0] + '.bin'

#Inicialización de los complementos para dispositivo en especifico y cargar las extensiones en el IECore
log.info("Creating Inference Engine")
ie = IECore()
if args.cpu_extension and 'CPU' in args.device:
    ie.add_extension(args.cpu_extension,"CPU")

#Lectura de la red neuronal (IR)
log.info('Loading network files:\n\t{}\n\t{}'.format(model_xml,model_bin))
net = IENetwork(model=model_xml,weights=model_bin)


#Comprobación del soporte de las capas de la red
if "CPU" in args.device:
    supported_layers = ie.query_network(net,"CPU")
    not_supported_layers = [l for l in net.layers.keys() if l not in supported_layers]
    if len(not_supported_layers) != 0:
        log.error("Following layers are not supported by the plugin for specified device {}:\n{}".format(args.device,','.join(
            not_supported_layers)))
        log.error("Please try to specify cpu extensions library path in sample´s command line parameters using -l"
                  "or --cpu_extension command line argument")

        sys.exit(1)
```

Al poder acceder a la topología de la red, es posible obtener las dimensiones de **entrada y salida de la red**. Por lo tanto, es posible definir la estructura de datos que gestiona ese abastecimiento e ingesta de información. La estructura de datos definida se denomina *bob*, que permite definir una dimensión fija en la variable. De esta manera, la estructura de los datos tanto de entrada como de salida estará acorde a la definición de la red.

Seguidamente, se establece la **carga del modelo** que permite asociar la red leída previamente con el dispositivo donde se va a realizar el procesamiento. Esto supone la conclusión de la configuración del motor de inferencias. Por lo tanto, llega el momento de decidir el tipo de inferencias a realizar durante este proceso (ver *Código* 4.4).

Código 4.4: Paso 4-5 del flujo de ejecución (Figura 4.2).

```
#Se definen las variables de entrada y salida
```

```
log.info('Preparing for input blobs')
input_blob = next(iter(net.inputs))
out_blob = next(iter(net.outputs))

#Se obtienen las dimensiones de la variable de entrada y la variable donde se cargarán las imagenes
categories = ['damaged','undamaged']
n,c,h,w = net.inputs[input_blob].shape
images = np.ndarray(shape=(n,c,h,w))

#Carga del modelo con el IECore, especificando el dispositivo donde se van a realizar las inferencias
log.info('Loading model to the plugin')
exec_net = ie.load_network(network=net,device_name=args.device)
```

Este proceso concluye con una fase iterativa (ver *Código* 4.5). Donde, en primera instancia, se **preprocesan los datos de entrada** para poder proporcionar a la variable definida con la estructura *bob* los datos a inferir. Por consiguiente, se produce la generación de la petición de inferencia, añadiendo esta a la cola del motor de inferencias. Posteriormente, se realiza **inferencia** en base al tipo establecido. Y finalmente, se obtiene la **predicción** con la información deseada, los datos de salida.

Código 4.5: Proceso iterativo (paso 6-8) del flujo de ejecución (Figura 4.2).

```
#Gestión de las rutas de los archivos
for name in args.input:
    if os.path.isdir(name):
        #img_input = os.path.join(name,os.listdir(name))
        img_input = os.listdir(name)
    else:
        img_input = [name]

    length_images = len(img_input)
    if length_images > 1:
        img_input.sort()


    #Proceso iterativo
    for ind,img in enumerate(img_input):
        #Preprocesamiento de las imágenes
        image = cv2.imread(os.path.join(name,img),cv2.IMREAD_GRAYSCALE)

        image_input = np.array(image).reshape(-1,1,128,128)

        images[ind %n] = image_input[0]

        if ind %n == n-1 or ind==length_images-1:
        #Start Inference
            type_ie = args.inference.casefold()
            if type_ie == "sync" or type_ie == "async":
                request_id = 0
                #Abastecer al motor con una petición de inferencia
                infer_request = exec_net.requests[request_id];
                num_iter = 1
                #Mapear la petición de inferencia con el motor
                request_wrap = InferReqWrap(infer_request, request_id, num_iter)

                # Comienza la ejecución de la petición de inferencia. Esperando a que la última ejecución
                #finalice
                time = request_wrap.execute(type_inference, {input_blob: images})

```

```
38                # Procesando el blob de salida
39                res = infer_request.outputs[out_blob]
40
41                log.info("Processing output blob")
42
43            #Medir el rendimiento de la inferencia (inferencias/segundo)
44            ips=(1/time)*n
45            ips_list.append(ips)
46            log.info('Inference: {} IPS'.format(ips))
47
48            if ind==length_images-1 and length_images%n !=0:
49                number=length_images%n
50            else:
51                number=n
52
53            #Mostrar los resultados obtenidos en la inferencia
54            for j,results in enumerate(res):
55                write_result = 1 if results[0] >= 0.5 else 0
56                print('Image {}: {} -> Prediction: {} ({})'.format(ind-number+j+2,img_input[ind-number+j+1],results↩
                      ↪ [0],categories[write_result]))
```

4.2. Tipos de Inferencia

Dentro de las configuraciones disponibles en el motor de inferencias, hay que definir la manera en la que el motor interactúa con los datos y el proceso de inferencia en sí, es decir, el proceso desde que se abastece al motor hasta que se obtienen los resultados. Esta relación se puede configurar en dos vertientes: **síncrona** o **asíncrona**. La elección entre una u otra depende en gran medida de la naturaleza de los datos y su modo de obtención. Esto requiere de manera implícita, tener en cuenta el contexto donde el modelo va a ser implementado porque, en relación con lo anterior, el modo de obtención de los resultados es diferente.

La inferencia **síncrona** permite abastecer los datos al motor de inferencias de manera secuencial, permitiendo que en cada inferencia únicamente se procese una imagen o dato de entrada (mostrado en la Figura 4.3), es decir, efectuando una petición por imagen. Aunque este mecanismo pueda parecer ineficiente, en ciertos escenarios como, por ejemplo, la inferencia de datos obtenidos a tiempo real puede resultar una solución adecuada. Hay que tener en cuenta que la frecuencia de obtención de datos es muy alta y, también, la demanda de obtener resultados. Por lo tanto, el mecanismo de recibir la imagen y realizar su predicción es prácticamente instantánea y parte necesaria de este contexto. Además, en este tipo de escenarios, la información realmente útil son las predicciones, no los datos capturados u obtenidos.

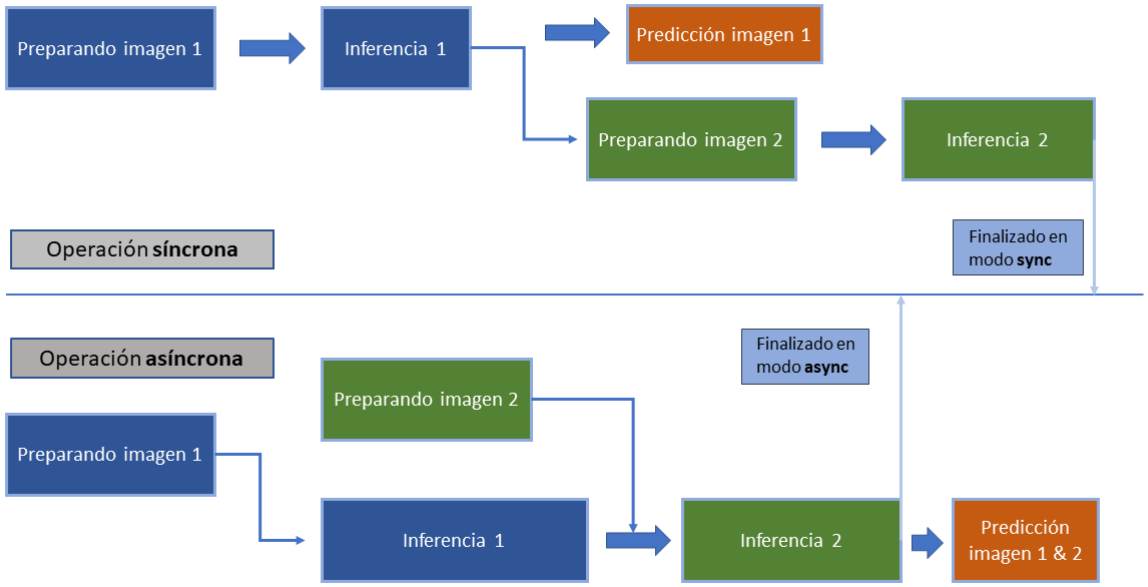


Figura 4.3: Comparativa entre inferencia **síncrona** y **asíncrona**.

A diferencia del tipo anterior, la inferencia **asíncrona** permite concatenar el procesamiento de varios datos, usando el mecanismo mostrado en la Figura 4.3. De esta manera, mientras se está realizando la inferencia de una imagen, se puede preprocesar la imagen siguiente, permitiendo agilizar este proceso. Sin embargo, esto posee ciertos inconvenientes, ya que la obtención de las predicciones se produce cuando todo este flujo finaliza. Por ello, en la anterior hipótesis, este tipo de procesamiento no tendría del todo sentido. Un escenario que puede considerarse viable es cuando se utilizan datos almacenados en lotes (*batch*), donde se conoce momentáneamente el inicio y el fin del procesamiento de los datos disponibles, teniendo en cuenta que la resolución de las predicciones no tenga que ser necesariamente instantánea a la finalización de cada inferencia.

Capítulo 5

Resultados Obtenidos

La comprobación de la adecuación del comportamiento del modelo no radica únicamente en el mejor valor obtenido en cada una de sus métricas establecidas. No obstante, la certificación de que el rendimiento extraído en la fase de entrenamiento es positivo, requiere un análisis mucho más minucioso. Para ello, uno de los principales recursos es la representación gráfica de las diversas métricas y resultados obtenidos durante esta fase. De esta manera, se pueden visualizar las tendencias generadas y ver si estas se ajustan al comportamiento esperado.

Una vez resuelta la cuestión anteriormente dispuesta por ese proceso de análisis y su posterior portabilidad a OpenVINO, se considera necesario medir la actuación de cada una de las dos fases anteriores, para mostrar el rendimiento y la eficiencia de las metodologías y procesos propuestos. Con este propósito, se probarán diferentes configuraciones con el objetivo de ver la tendencia o comportamiento en diversas situaciones, en lo que a precisión y tiempos de ejecución se refiere. Aunque, no es posible obtener esa noción de buen rendimiento en las diferentes fases si no hay un caso base del cual partir. En ambos experimentos, se establecerá un caso de referencia para poder dilucidar el nivel de adecuación y mejora con los datos obtenidos. Las razones por las que han sido elegidos estos casos y datos, quedan establecidas en el apartado correspondiente a cada fase (sección 5.2 y 5.3).

Las métricas que van a ser analizadas serán diferentes, debido principalmente a que se tratan de procesos dispares. Por lo tanto, las dos fases definidas: **entrenamiento** e **inferencia**, poseen cierta relación entre sí, manteniendo el **tiempo de ejecución** como uno de los valores base a analizar. Sin embargo, en la prueba de rendimiento planteada, establecen casos totalmente diferentes. Además,

este hecho se traduce en el hardware a utilizar, el cual, difiere por cada prueba de rendimiento (especificado en el apartado 2.3.4). Este estará principalmente relacionado con el dispuesto por cada entorno de desarrollo utilizado y el más adecuado para poder analizar el escenario base en unas condiciones equiparables.

5.1. Análisis del entrenamiento

Una vez finalizada la fase de *training* del modelo, llega el momento de analizar la actuación de este con los datos e hiperparámetros establecidos. Para ello, se analizan los valores extraídos de las métricas definidas en la fase de compilación del entrenamiento. En este caso, hay que tener en cuenta la **precisión** (*accuracy*) y la **pérdida** (*loss*) como parámetros a analizar. Con todo ello, se realiza lo que se denomina como la fase de ***tuning***, donde se ajustan los hiperparámetros del modelo para conseguir la mejor actuación posible con los datos disponibles. Como se ha mencionado con anterioridad, esto se convierte en un proceso cíclico entre la fase de entrenamiento y prueba (incluyendo en este el análisis), debido a que, en base a las métricas obtenidas, es posible dilucidar las modificaciones en los parámetros de entrenamiento, topología de la red o en los propios datos para mejorar y orientar el modelo hacia el comportamiento deseado.

Principalmente en problemas de clasificación, la matriz de confusión resulta muy útil debido a que permite comparar de manera muy sencilla los resultados obtenidos por la red neuronal y los esperados en cada caso.

Tal y como se puede observar en la Figura 5.1, la precisión del modelo es muy alta, debido a que únicamente en dos ocasiones, la predicción no concuerda con el valor correcto. De esta manera, se pueden apreciar los resultados que están asociados a cada categoría y la relación entre las categorías predichas y las correctas. Volviendo a las dos predicciones "erróneas", se muestra que dos imágenes han sido etiquetadas como dañadas cuando son zonas no dañadas. Este hecho puede indicar una serie de patrones, los cuáles, el modelo no es capaz o tiene dificultades para reconocerlos. En este caso, no resulta un dato muy destacable *a priori*.

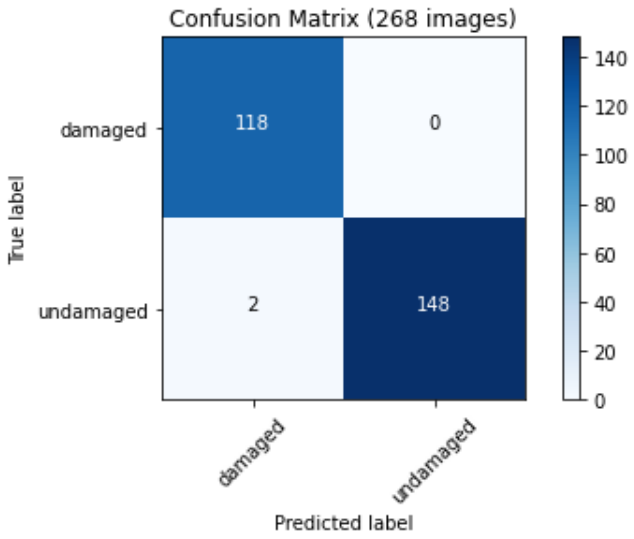


Figura 5.1: Matriz de confusión con todas las imágenes disponibles.

Otro aspecto a recalcar es el hecho de que todas las imágenes que el modelo categoriza como no dañadas son predicciones correctas con respecto a la etiqueta asociada a esa imagen. Además, resulta importante visualizar que el modelo realiza alguna predicción "incorrecta" debido a que esto es un indicativo de que la red entrenada no ha memorizado los datos, es decir, no hay *overfitting*.

Todo este análisis realizado a las métricas de entrenamiento puede suponer una mejora sustancial en la actuación del modelo, pero solo si se consiguen unos resultados ciertamente prometedores de alguna manera. Por otro lado si no se alcanza ese estado, puede suponer una absoluta pérdida de tiempo debido a que el número de combinaciones y elementos es tremendamente grande desde la topología del modelo hasta los hiperparámetros definidos y la serie de dependencias que hay entre estos. Teniendo en cuenta que cualquier mínima variación en alguna faceta del modelo puede suponer un cambio total en el comportamiento de este.

En cada fase de entrenamiento, se tiene que obtener alguna pista o indicativo para poder hallar donde el modelo puede mejorar, esto puede suponer un *downgrade* en otra faceta de este. Por ejemplo, si se quita una capa del modelo, esto seguramente suponga que la precisión del modelo baje en comparación con la anterior versión. Pero, por otro lado, su entrenamiento será mucho más rápido, ya que el número de parámetros a procesar será inferior y de esta manera, tanto el entrenamiento como la inferencia tardará mucho menos en comparación.

De las métricas utilizadas es necesario analizar su comportamiento en dos sets de datos diferentes: *training data* y *validation data*.

Figura 5.2: La **precisión** en la fase de entrenamiento del modelo.

En primer lugar, se puede observar en ambas gráficas (Figura 5.2 y Figura 5.3) la trayectoria que siguen las dos métricas y el nivel de estas, es decir, en qué valores del eje de ordenadas (y) se estabiliza la función.

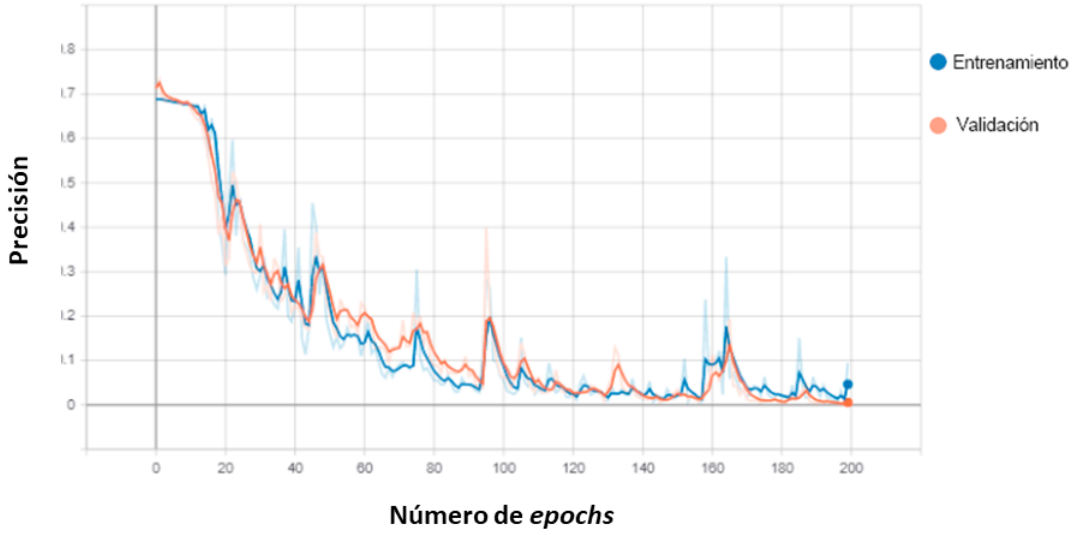


Figura 5.3: Los valores de **pérdida** en la fase de entrenamiento del modelo.

Con respecto al rendimiento obtenido, se muestran unos resultados muy buenos ya que la precisión está muy cerca del valor ideal (1) (ver la Figura 5.2) y la pérdida se encuentra muy cerca del suyo (0) (ver la Figura 5.3), siguiendo una trayectoria muy suave. Resulta especialmente importante este último hecho mencionado, la trayectoria. Ya que se trata de uno de los indicativos que se pueden examinar para dilucidar si hay un sobreajuste en la red que está afectando a la salida obtenida. Una manera de detectarlo podría ser si se aprecia demasiado **ruido** en la gráfica o si la **diferencia de rendimiento** entre los datos de validación y los de entrenamiento es significativa. Como se aprecia especialmente en la Figura 5.3, estos dos detalles se muestran pero no llegan a ser lo suficientemente concluyentes para poder entrever un mal rendimiento por parte del modelo. Por otro lado, la Figura 5.2 exhibe un mejor comportamiento en este aspecto, lo cual, constata que el ruido apreciado en los valores de pérdida no resultan alarmantes. Además, la curva que describen ambas, es un gran indicativo del proceso de aprendizaje que ha tenido la red, comenzando por unos niveles de precisión muy bajos y finalizando progresivamente con una actuación casi perfecta.

5.2. Rendimiento en la fase de entrenamiento

Como punto de referencia para poder comparar los datos obtenidos en cuanto a precisión y tiempo de entrenamiento, se tendrán en cuenta los valores experimentales expuestos en el documento que ha servido como base para la realización de este trabajo [2]. En el experimento propuesto se demuestra que los tiempos de entrenamiento se pueden reducir muy considerablemente al usar una GPU como dispositivo y por lo tanto, se realiza una comparativa entre los tiempos de ejecución sobre diferentes aspectos relacionados con la fase de *training*. Estos son los valores extraídos del experimento:

- **Total de imágenes utilizadas**: 280 (28 de ellas destinadas al proceso de validación y 40 a la fase de prueba tras finalizar el proceso de entrenamiento).
- **Número de epochs:** 200.
- **Tamaño de los lotes (*batch size*):** 32.
- **Pico de precisión en el entrenamiento:** 90 %.

- **Pico de precisión en la fase de *testing*:** 85 %.

Operación	**CPU**	**GPU**(Tesla K80)
Cada *epoch*	3s 38ms	1ms
Proceso de entrenamiento	11min 46s	33s

Tabla 5.1: Tiempos de ejecución obtenidos en el trabajo [2] que sirve como base.

La idea planteada consiste en establecer dos escenarios, teniendo en cuenta diferentes métricas de análisis. Cada escenario tendrá un valor distinto de *epoch* (100 y 200, respectivamente), manteniendo el tamaño de los lotes (*batch size*) a 32 (al igual que en el caso de referencia, Tabla 5.1). Dentro de cada uno de ellos se dispondrá de 4 distribuciones distintas de datos comunes entre ambos escenarios, permitiendo observar la adecuación del modelo ante diferentes proporciones. En cuanto al hardware utilizado, se sigue la convención mencionada anteriormente con los dispositivos que nos ofrece la herramienta *Google Colab*, con el añadido del uso de la TPU a diferencia del caso base. La repartición de los datos se encuentra definida en esta tabla 5.2:

Distribución	**Proporción de Datos**		
	Training	*Testing*	*Validation*
1	192 (71.64 %)	44 (16.41 %)	32 (11.94 %)
2	128 (47.76 %)	81 (30.22 %)	59 (22.01 %)
3	96 (35.82 %)	108 (40.29 %)	64 (23.88 %)
4	224 (83.58 %)	22 (8.21 %)	22 (8.21 %)

Tabla 5.2: Distribuciones de datos establecidas para la toma de rendimiento.

De esta manera, siguiendo las proporciones de datos definidas, se establecen los escenarios expuestos con anterioridad, centrando especialmente la atención en dos métricas principales: tiempo de duración del proceso de entrenamiento (por cada *epoch* y duración total) y la precisión máxima obtenida con el modelo resultante, comparando el comportamiento de este durante el entrenamiento y la prueba posterior con los datos de *testing*. Con el principal propósito de mantener el menor tiempo de entrenamiento posible, pero sin afectar en gran medida a la precisión, principalmente la obtenida en la fase de prueba.

En primera instancia, hay que señalar que el escenario 2 (ver Tabla 5.4) posee una configuración

similar a la establecida en el caso de referencia. En cuanto a la distribución de los datos, la primera proporción (ver Tabla 5.2) sería la que más cerca se encuentra al caso base propuesto, ya que el número de imágenes disponibles en este proyecto es inferior (268) a las disponibles en el trabajo base referenciado.

El escenario 1 (ver Tabla 5.3) trata la idea de reducir el número de iteraciones (*epochs*) durante la fase de entrenamiento con el fin de poder llegar a obtener un rendimiento satisfactorio o, al menos, unas características similares al caso de referencia expuesto en la Tabla 5.1. Por tanto, como se puede apreciar en la tabla anteriormente mencionada y la perteneciente al escenario 1 (ver Tabla 5.3), los valores extraídos resultan ser ligeramente inferiores a lo que a tiempos se refiere. Hecho que es realmente sorprendente, debido a que el número de iteraciones establecidas es inferior. Sin embargo, en lo que a precisión se refiere, los resultados son superiores tanto en entrenamiento como en *testing*. Esto sería siguiendo únicamente la distribución 1 que es la que comparten ambas, aunque en las demás se sigue la misma tónica.

Distribución	**Dispositivo**	**Tiempo** (*seg*)		**Entrenamiento**		**Testing**
		Epochs	Proceso	acc	loss	acc
1	CPU	2.361	236.634	0.964	0.096	90.70 %
	CPU+GPU	0.396	40.572	0.948	0.177	86.05 %
	CPU+TPU	0.240	24.032	0.995	0.026	81.40 %
2	CPU	1.678	167.976	0.984	0.066	88.89 %
	CPU+GPU	0.238	23.975	0.992	0.062	86.42 %
	CPU+TPU	0.206	20.638	0.992	0.040	90.12 %
3	CPU	1.323	132.480	1.000	0.016	90.74 %
	CPU+GPU	0.192	19.388	0.990	0.079	86.11 %
	CPU+TPU	0.177	17.697	0.958	0.141	83.33 %
4	CPU	2.734	273.965	1.000	0.017	95.45 %
	CPU+GPU	0.357	35.954	0.991	0.042	86.36 %
	CPU+TPU	0.231	23.079	0.996	0.044	100.00 %

Tabla 5.3: Escenario 1 establecido a 100 *epochs* y tamaño de lotes (*batch*) igual a 32.

Realizando una comparación directa entre ambos escenarios resulta muy llamativo el descenso de precisión por parte del segundo (ver Tabla 5.4), en la sección de *testing* especialmente en la

distribución 1. Esto es un indicativo muy claro del sobre-entrenamiento y, por tanto, el sobre-ajuste que dispone la red obtenida. A esto hay que añadir el alto valor de pérdida que se obtiene en esos casos concretos. En relación a este hecho, en la mayor parte de estos, la precisión del entrenamiento es muy superior a la de *testing*, indicando que el valor de esta última fase del proceso de *training* permite verificar la veracidad del comportamiento del modelo. Por lo tanto, el resultado obtenido por las métricas de entrenamiento no certifica ni asegura el rendimiento obtenido por el modelo, siempre y cuando se mantenga en unos valores razonables.

Distribución	**Dispositivo**	**Tiempo** (*seg*)		**Entrenamiento**		**Testing**
		Epochs	Proceso	acc	loss	acc
1	CPU	2.386	477.432	0.974	0.091	69.77 %
	CPU+GPU	0.303	60.966	0.995	0.027	79.07 %
	CPU+TPU	0.196	39.164	1.000	0.002	95.35 %
2	CPU	1.722	345.057	1.000	0.004	90.12 %
	CPU+GPU	0.228	45.947	1.000	0.002	95.06 %
	CPU+TPU	0.168	33.637	1.000	0.001	90.12 %
3	CPU	1.322	264.736	1.000	0.002	93.52 %
	CPU+GPU	0.181	36.661	1.000	0.003	92.59 %
	CPU+TPU	0.144	28.734	1.000	0.003	92.59 %
4	CPU	2.668	533.908	1.000	0.001	81.82 %
	CPU+GPU	0.379	76.576	1.000	0.002	100.00 %
	CPU+TPU	0.194	38.849	1.000	0.001	90.91 %

Tabla 5.4: Escenario 2 establecido a 200 *epochs* y tamaño de lotes (*batch*) igual a 32.

Esto cuestiona la hipótesis sobre el número de iteraciones necesarias para que la red proporcione un buen rendimiento. Como hemos visto en los dos anteriores escenarios, los valores extraídos muestran que la diferencia de tiempo es la esperada, más o menos la mitad. Sin embargo, estos datos no siguen la misma tendencia en la parte de precisión, por lo que establecer el valor de *epochs* a 200 puede resultar no ser la mejor configuración en lo que relación tiempo-precisión se refiere.

No obstante, este no ha sido el único método planteado para optimizar este proceso de aprendizaje, debido a que, principalmente, los tiempos de ejecución extraídos no satisfacían del todo los valores esperados. Sobre todo, teniendo en cuenta los reflejados en el caso base. Finalmente,

se encontraron dos técnicas que propiciaron esta mejora: eliminar los *callbacks* de reporte en cada *epoch* de la ejecución y una optimización diseñada por NVIDIA denominada *Mixed Precision*.

Este concepto de *Mixed Precision*[1] ofrece una mejora en la velocidad de entrenamiento, realizando operaciones con formato de precisión media (*half-precision*, FP16). Comúnmente, se selecciona el formato de entrada y salida, tanto de las capas como los datos a introducir en estas. Sin embargo, las operaciones internas dentro del proceso dependen enteramente de la operación a realizar. Por lo que, esta optimización permite establecer un formato común para todos los cálculos necesarios para la fase de entrenamiento, generando dos ventajas: disminuir la cantidad de memoria requerida y el tiempo de entrenamiento, aunque la precisión también se reduce. Esta técnica solo es aplicable en dispositivos de NVIDIA y dentro de estos, GPUs. En los casos en los que se hayan aplicado estas dos técnicas se denominarán en este documento como optimizados.

Una vez obtenidos todos los resultados para cada uno de los escenarios y configuraciones planteadas, llega el momento de seleccionar las mejores distribuciones de datos para cada uno de ellos, buscando la mejor relación tiempo-precisión. De esta manera, se puede establecer lo que se podría considerar como el mejor modelo en base a los parámetros analizados.

Siguiendo esta convención, los casos seleccionados son los siguientes: escenario 1 con la distribución 2 (ver Tabla 5.3), escenario 2 con la distribución 3 (ver Tabla 5.4) y de los datos extraídos por parte de la optimización realizada, el escenario 2 con la distribución 3.

Realizando una comparativa entre el mejor rendimiento obtenido por cada configuración (ver Figura 5.4 y 5.5), se puede apreciar la amplia diferencia entre el tiempo de ejecución con la CPU y el resto de casos donde los valores son más parejos. Por otro lado, hay que recordar que tanto los escenarios como la distribuciones son distintas, esto hace que los datos no puedan compararse directamente, hay que realizar un compendio entre todas las métricas.

En la Figura 5.5 se puede observar que en el mejor caso del escenario 1, los valores de precisión alcanzados son inferiores a los demás casos extraídos (sobre todo en la parte de *testing*). Esto es debido a que el número de iteraciones es menor que en el resto. Aunque, en contrapartida, el tiempo se reduce en todos los casos comparados con el escenario 2 sin optimizar. Aun así, la precisión de

[1] `https://docs.nvidia.com/deeplearning/performance/mixed-precision-training/index.html`

(*testing*) obtenida en el primer caso se puede considerar como muy buena, con un **88.46 %** de media entre todos los dispositivos.

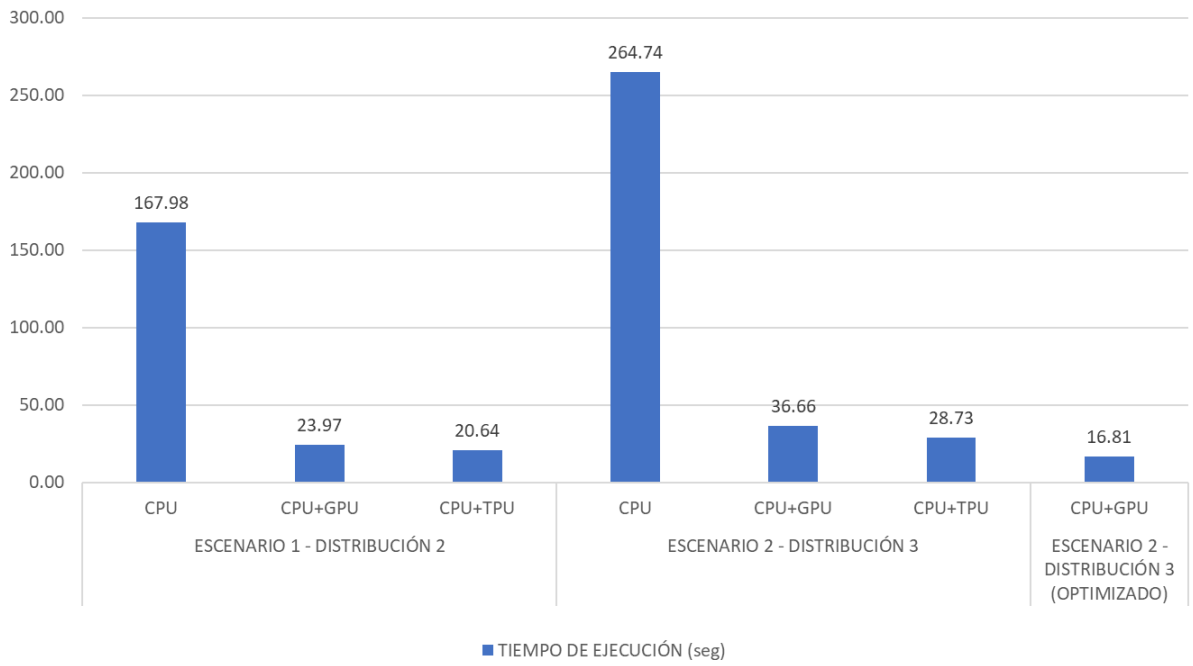


Figura 5.4: Tiempo de ejecución de los mejores casos obtenidos.

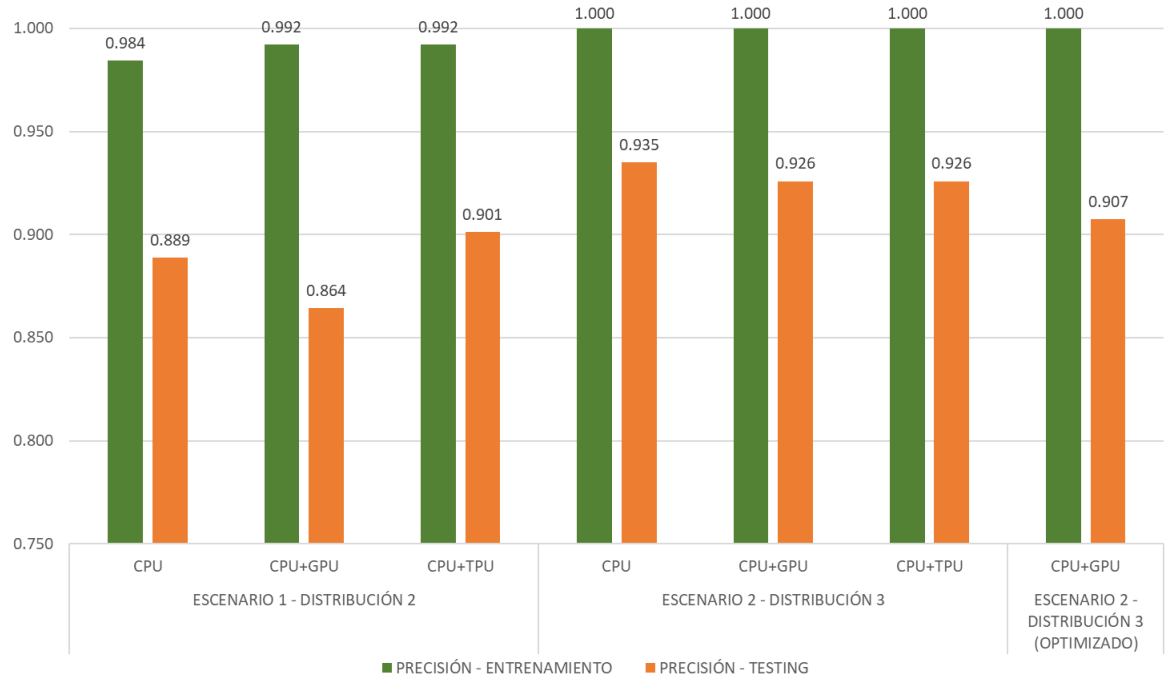


Figura 5.5: Precisión de los mejores casos obtenidos.

Otro aspecto a destacar, es la regularidad en los niveles de precisión (ver Figura 5.5) por parte del escenario 2 sin optimizar. Donde, además, se obtienen los valores pico más altos por cada dispositivo, albergando un menor número de imágenes para entrenar. Sin embargo, tiene un número similar o superior de iteraciones que el resto de casos, por lo tanto, se puede presuponer (y, finalmente, se constata) que es la configuración con un tiempo de ejecución más alto, siendo con la CPU donde se atisba esta diferencia de manera más abultada.

De esta manera, se puede apreciar que la relación tiempo-precisión no resulta sencilla de definir, especialmente cuando se trata de establecer cuál es la mejor combinación de las establecidas. Finalmente y teniendo en cuenta todas las observaciones realizadas, los resultados extraídos de la configuración **optimizada** (Escenario 2 - Distribución 3) serían los que poseen un mayor balance en esa relación. Mostrando una bajada de precisión (ver Figura 5.5) apreciable con respecto al caso del escenario 2. Sin embargo, en cuanto al tiempo de ejecución (ver Figura 5.4), es el mínimo obtenido, suponiendo una reducción del **54.14 %** con respecto a la GPU y un **41.50 %** en relación a la TPU de ambos escenarios 2. En relación al escenario 1 esta diferencia se reduce, pero los niveles de precisión son bastante más bajos.

5.3. Rendimiento en la fase de inferencia

Tras medir la eficiencia del modelo en cuanto a tiempo y precisión con diferentes configuraciones y dispositivos, continúa el flujo de la metodología propuesta (ver apartado 4.1) implementando este en un entorno de producción. A modo de poder dilucidar si hay mejora o no en el uso del *toolkit* y el grado de esta con respecto a otros entornos de las mismas características, se van a comparar los resultados obtenidos con la herramienta proporcionada por *Tensorflow*, *Tensorflow Serving*. Esto quiere decir que la información extraída por parte de este *framework*, representará el escenario base para poder comparar el rendimiento alcanzado por parte de OpenVINO.

La métricas a enfocar en este apartado van a estar principalmente relacionadas con valores temporales o el rendimiento de las ejecuciones. En cuanto a la precisión del modelo queda recogido en el apartado anterior, debido a que la portabilidad del modelo a este tipo de entornos no afecta a su comportamiento y, por tanto, los resultados obtenidos al respecto son también fidedignos en

el caso de su puesta en producción.

A modo de que las ejecuciones se realicen en las condiciones más similares posibles en cada *toolkit*, el hardware utilizado va a ser el mismo en ambos casos. El cuál, consiste en la CPU y la VPU de Movidius en los casos que sea posible, proporcionadas por la máquina remota.

5.3.1. Rendimiento en *Tensorflow Serving*

En lo que concierne a este caso, hay que tener en cuenta que el funcionamiento entre ambos *toolkits* es diferente en lo que a efectos prácticos se refiere. Por lo que la única métrica que ha sido posible capturar de cada inferencia es el tiempo de procesamiento de esta, eliminando el tiempo de transferencia de la petición, obteniendo el tiempo íntegro de ejecución.

Para el procesamiento de estas peticiones solo se admiten dispositivos como CPUs y GPUs, esto indica que estos datos pertenecen a la ejecución con la CPU de la máquina remota. Estos son los valores obtenidos (ver Tabla 5.5) en la ejecución usando diferentes cantidades de datos:

Número de imágenes	Tiempo de Ejecución
128	256.27 ms
160	318.50 ms
192	382.20 ms
224	462.25 ms
256	507.34 ms
268	533.24 ms

Tabla 5.5: Tiempos de inferencia obtenidos con *Tensorflow Serving*.

5.3.2. Rendimiento en OpenVINO

El rendimiento obtenido por este *toolkit* representa una parte muy importante del proyecto desarrollado, debido a que establece la mayoría de las aportaciones proporcionadas por este. Permitiendo mostrar el impacto y las mejoras que proporciona esta red. De esta manera, se puede establecer como una alternativa viable en las herramientas relacionadas con este tipo de optimizaciones e implementaciones.

Los principales parámetros de análisis definidos en esta serie de experimentos tratan de despejar ciertas incógnitas con respecto al *toolkit* y el comportamiento de este con el modelo. Primeramente,

la **duración** muestra de manera simple la eficiencia del proceso de inferencias desde un punto de vista temporal. Posteriormente, la **latencia** trata de mostrar las principales ventajas proporcionadas por cada tipo de inferencia, proporcionando el tiempo que se tarda en realizar cada una de estas. Para finalizar, el ***throughput*** que proporciona el rendimiento en base al volumen de trabajo realizado, indicando la velocidad de procesamiento de todo el proceso.

Siguiendo todas las directrices anteriormente mencionadas, en la Tabla 5.6 se puede observar el rendimiento obtenido con CPU. En este se pueden destacar varias tendencias que se van a ir repitiendo en el resto de casos. Siendo la principal, la mejora de los datos en el uso de inferencia asíncrona con respecto a la síncrona. Aunque, pueda no resultar tan aparente en lo que a duración se refiere, hay que observar la amplia diferencia entre las métricas de latencia y *throughput*. Donde en el caso de la primera, los valores pueden llegar a reducirse considerablemente hasta un **50.72 %** y de media un **30.68 %**. Y en el último caso, esta resulta más discreta con un aumento máximo del **8.98 %** y un **4.24 %** de media.

Distribución			**Métricas**		
Lotes	Peticiones	Datos	Duración	Latencia	Throughput
SYNC					
128	–	128	134.90 ms	134.61 ms	950.91 FPS
160	–	160	167.27 ms	167.00 ms	958.06 FPS
192	–	192	197.57 ms	197.30 ms	973.13 FPS
224	–	224	233.35 ms	233.07 ms	961.08 FPS
256	–	256	268.99 ms	268.72 ms	952.65 FPS
268	–	268	281.44 ms	281.18 ms	953.14 FPS
ASYNC					
16	8	128	122.52 ms	77.20 ms	1044.74 FPS
16	10	160	166.55 ms	110.62 ms	960.68 FPS
16	12	192	185.10 ms	125.91 ms	1037.27 FPS
16	14	224	228.86 ms	129.04 ms	978.77 FPS
16	16	256	248.25 ms	132.40 ms	1031.20 FPS
32	4	128	129.67 ms	128.24 ms	987.13 FPS
32	6	192	203.39 ms	129.05 ms	943.99 FPS
32	8	256	245.92 ms	159.47 ms	1040.98 FPS

Tabla 5.6: Métricas obtenidas de las inferencias usando la CPU.

Otro aspecto a reseñar es la obtención de una mejor latencia cuanto el valor de peticiones es mayor. Esto únicamente se puede apreciar dentro de la inferencia asíncrona, donde se expone que la tendencia es más positiva en comparación. En el caso del *throughput*, esta no queda tan clara, pero mantiene la apreciación mostrada al principio. Esto puede ser, debido a que el *toolkit* es capaz de realizar una mejor gestión del flujo de ejecución al tener un mayor número de tareas que una gran cantidad de imágenes en cada lote. La Tabla 5.7, muestra las apreciaciones anteriores de una manera más clara. Indicando, además, que la capacidad de cómputo y procesamiento de ambos dispositivos es bastante dispar, siendo la CPU mucho más potente en este aspecto que el *stick*.

Distribución			**Métricas**		
Lotes	Peticiones	Datos	Duración	Latencia	Throughput
SYNC					
128	–	128	328.37 ms	328.01 ms	390.23 FPS
160	–	160	409.96 ms	409.32 ms	390.89 FPS
192	–	192	491.91 ms	491.38 ms	390.73 FPS
224	–	224	574.65 ms	573.98 ms	390.26 FPS
256	–	256	654.91 ms	654.27 ms	391.28 FPS
268	–	268	685.37 ms	684.67 ms	391.43 FPS
ASYNC					
16	8	128	191.74 ms	121.26 ms	667.57 FPS
16	10	160	237.60 ms	144.28 ms	673.39 FPS
16	12	192	285.76 ms	168.45 ms	671.89 FPS
16	14	224	331.53 ms	191.44 ms	675.66 FPS
16	16	256	376.61 ms	214.03 ms	679.74 FPS
32	4	128	192.68 ms	144.74 ms	664.31 FPS
32	6	192	283.39 ms	188.96 ms	677.51 FPS
32	8	256	375.24 ms	235.89 ms	682.23 FPS

Tabla 5.7: Métricas obtenidas de las inferencias usando Movidius NCS2.

Por otro lado, hay que destacar la amplia diferencia entre los tipos de inferencia, mucho más evidente que el caso anterior (ver Tabla 5.6). Otra diferencia observable, es la homogeneidad de resultados dentro de cada uno de los tipos. Donde, la latencia sigue siendo una de las principales mejoras con una reducción del **63.50 %** y a esta se le une un *throughput* con un aumento del

42.09 %.

Adicionalmente a la implementación y prueba de rendimiento con cada uno de los dispositivos, OpenVINO permite el uso de varios dispositivos a la vez dentro de un proceso de inferencia. Es lo que se denomina como **MULTI**[2]. De tal manera que se especifican los dispositivos por orden de prioridad a su uso, provocando que el primero funcione como procesador principal de la tarea y el resto de ellos como coprocesadores a este.

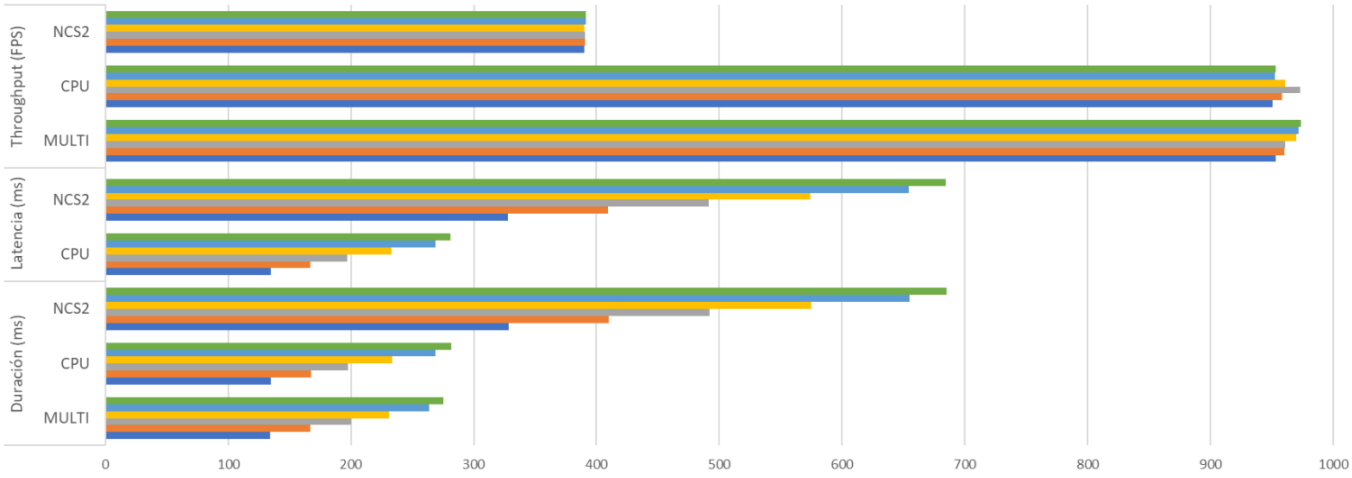
La integración realizada con ambos dispositivos se ha efectuado especificando como principal la CPU, debido a que se trata del más potente y el *stick* como auxiliar. Los resultados adquiridos mejoran ligeramente con respecto a los obtenidos con CPU (ver Tabla 5.6). Por otro lado, si el orden hubiera sido el inverso, los datos alcanzados no mejorarían en ningún caso a los registrados con la CPU. De hecho, se parecerían a los extraídos con el NCS2 y según las pruebas realizadas no hay una mejora muy apreciable.

Para poder observar de manera comparativa los resultados recabados en las tres integraciones, la Figura 5.6 muestra las métricas generadas en la inferencia **síncrona** en cada uno de ellos. Tal como se puede atisbar en esta, las mejoras de la integración multi-dispositivo con respecto a la CPU, no resultan muy destacables, aunque sí se pueden apreciar.

Sin embargo, en la Figura 5.7 que sigue el mismo formato que la anterior, sí se puede detectar de manera más clara esta diferencia prácticamente en todas las métricas que comparten. Suponiendo esta un **14.50 %** de mejora en el *throughput* en comparación con la CPU y con un valor máximo de **1490.27 FPS** siguiendo una distribución de 16 como tamaño de lote y 14 peticiones, formando un total de 224 imágenes.

Continuando esa comparativa entre la integración con CPU y MULTI, se puede extraer la media de FPS proporcionada en ambos casos, siendo 1002.39 y 1272.98, respectivamente. Aunque pueda parecer mayor en la Figura 5.7, la realidad es que en un único caso (4 peticiones y 32 de *batch*) no hay ninguna diferencia entre ambas, generando que la diferencia se reduzca.

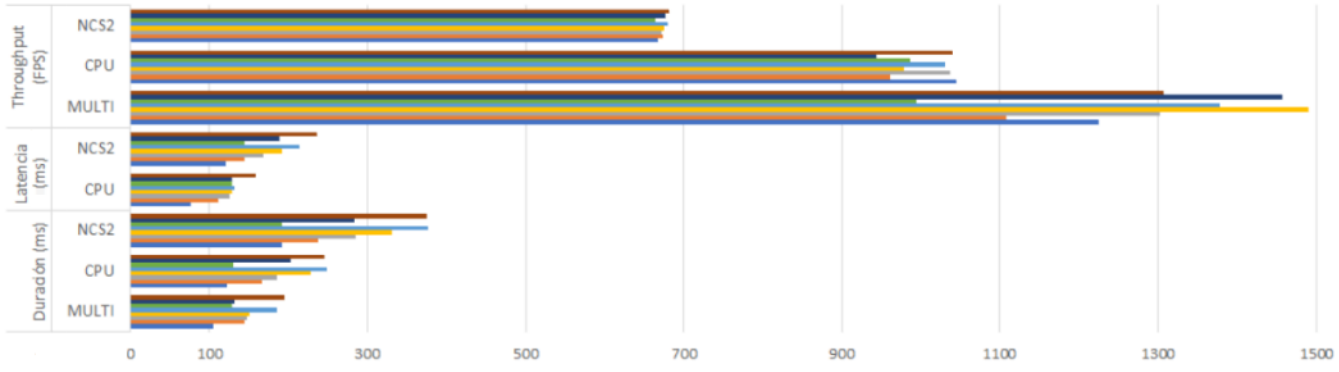
[2] `https://docs.openvinotoolkit.org/latest/_docs_IE_DG_supported_plugins_MULTI.html`



	Duración (ms)			Latencia (ms)		Throughput (FPS)		
	MULTI	CPU	NCS2	CPU	NCS2	MULTI	CPU	NCS2
268	275.30	281.44	685.37	281.18	684.67	973.99	953.14	391.43
256	263.61	268.99	654.91	268.72	654.27	971.65	952.65	391.28
224	231.03	233.35	574.65	233.07	573.98	970.17	961.08	390.26
192	199.97	197.57	491.91	197.30	491.38	960.78	973.13	390.73
160	166.76	167.27	409.96	167.00	409.32	960.22	958.06	390.89
128	134.40	134.90	328.37	134.61	328.01	953.34	950.91	390.23

268 256 224 192 160 128

Figura 5.6: Comparativa de la inferencia síncrona entre las tres integraciones: CPU, Movidius NCS2 y MULTI.



	Duración (ms)			Latencia (ms)		Throughput (FPS)		
	MULTI	CPU	NCS2	CPU	NCS2	MULTI	CPU	NCS2
32 x 8 - 256	195.84	245.92	375.24	159.47	235.89	1307.22	1040.98	682.23
32 x 6 - 192	131.68	203.39	283.39	129.05	188.96	1458.11	943.99	677.51
32 x 4 - 128	128.66	129.67	192.68	128.24	144.74	994.87	987.13	664.31
16 x 16 - 256	185.68	248.25	376.61	132.40	214.03	1378.71	1031.20	679.74
16 x 14 - 224	150.28	228.86	331.53	129.04	191.44	1490.57	978.77	675.66
16 x 12 - 192	147.42	185.10	285.76	125.91	168.45	1302.38	1037.27	671.89
16 x 10 - 160	144.32	166.55	237.60	110.62	144.28	1108.64	960.68	673.39
16 x 8 - 128	104.45	122.52	191.74	77.20	121.26	1225.50	1044.74	667.57

32 x 8 - 256 32 x 6 - 192 32 x 4 - 128 16 x 16 - 256 16 x 14 - 224 16 x 12 - 192 16 x 10 - 160 16 x 8 - 128

Figura 5.7: Comparativa de la inferencia asíncrona entre las tres integraciones: CPU, Movidius NCS2 y MULTI.

Una de las apreciaciones que se pueden extraer de esta funcionalidad multi-dispositivo proporcionada por el *toolkit*, es la rentabilidad en base al **coste** a realizar en hardware y el **rendimiento** obtenido. Resulta evidente que este planteamiento solo llegaría a tener sentido en un entorno asíncrono, pero aun así no queda del todo claro. Además, el motor de inferencias no permite extraer los valores de latencia utilizando esta funcionalidad, por lo que no es posible apreciar la mejora proporcionada en este ámbito de la inferencia.

Aun así, los resultados obtenidos suponen una mejora muy significativa en todos los aspectos, especialmente en la inferencia **asincrona**. Esta tendencia mostrada por el tipo de inferencia también se traduce en los demás dispositivos donde esta diferencia resulta claramente más abultada en el *stick* NCS2.

Adicionalmente, hay que recalcar el número de datos de entrada disponibles, debido a que resulta ciertamente pequeño en comparación a otros escenarios de similares características. No obstante, siguiendo la distribución y tendencia de los resultados extraídos, la escalabilidad de este sistema resulta uniforme. Indicando que el aumento del número de datos no provocará una bajada de rendimiento. Por lo tanto, la mejora resaltada en este experimento se mantendrá o aumentará realizando un mayor número de inferencias.

5.3.3. Comparativa

Una de las principales razones de la elección de OpenVINO era conseguir averiguar el grado de mejora que este puede proporcionar, en base a otra herramienta de similares características. En este caso, se ha considerado oportuno usar una que forma parte del mismo *framework* que ha servido para la definición y entrenamiento de la red, provocando que la comparatoria no pueda ser más idónea.

A continuación, comparando los resultados obtenidos por ambos *toolkits*, hay que reseñar los únicos casos comparables son los generados con la **CPU** y con modo de inferencia **síncrono**, debido a que se trata del único dispositivo que permite inferir *Serving* y el tipo de inferencia con el que ha sido posible extraer de este unos resultados coherentes.

Tal y como se muestra en la Figura 5.8, los datos adquiridos muestran claramente que la mejora

proporcionada por OpenVINO es muy sustancial, suponiendo casi la mitad de tiempo en la mayoría de casos. Adicionalmente, se puede determinar la tendencia que siguen ambos, identificando que la diferencia aumenta cuanto mayor sea el número de imágenes a inferir.

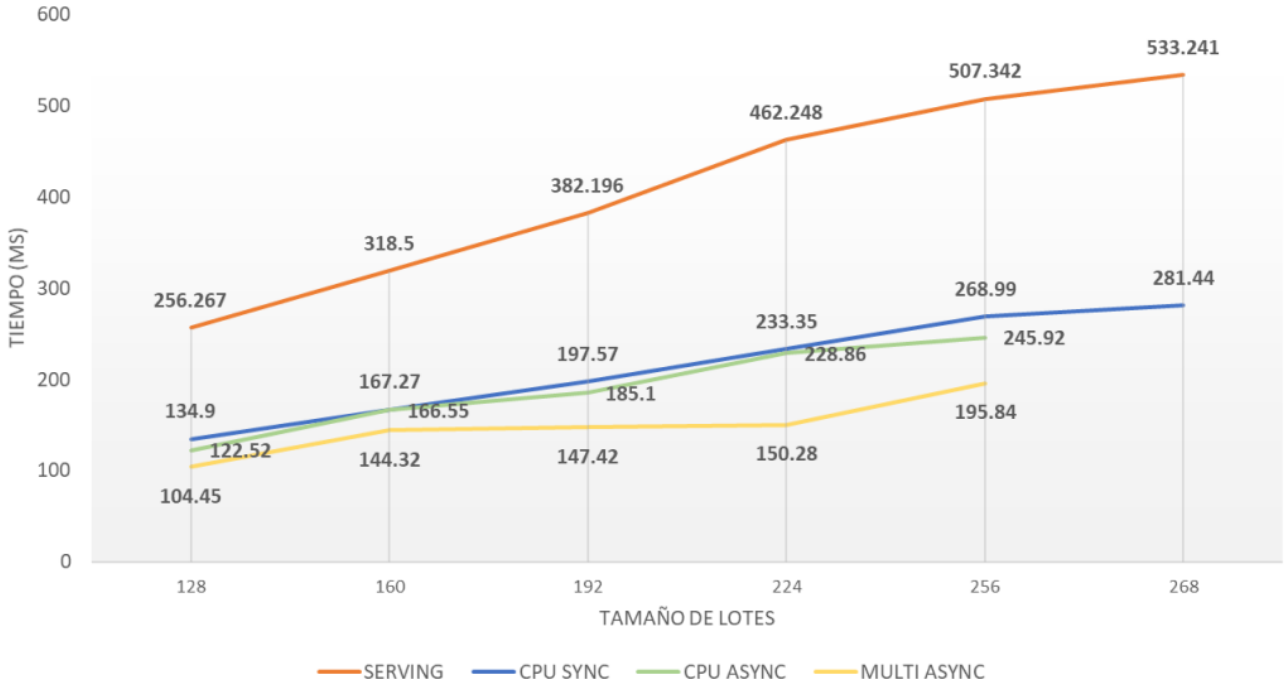


Figura 5.8: Comparativa del tiempo de ejecución entre OpenVINO y *Tensorflow Serving*.

Por tanto, se constata la subida de rendimiento proporcionado por el *toolkit* de Intel para la fase de inferencias, con un **47.80 %** de media sobre el tiempo recibido por *Serving* y una diferencia máxima de **251.80 milisegundos** en la ejecución realizada con la mayor cantidad de imágenes disponibles (268).

El resto de datos mostrados en la Figura 5.8, reflejan la mejora ofrecida tanto por la ejecución en modo asíncrono de la inferencia como el uso de la funcionalidad multi-dispositivo. Como se puede atisbar, los tiempos no difieren de manera muy abultada al cambio del tipo de inferencia, suponiendo una reducción del **10.99 %** de media.

Sin embargo, al combinar ambas, se permite apreciar una mejora muy significativa en todas las distribuciones de datos. Estableciendo esta última como la **mejor configuración** hallada con el hardware disponible. Traduciendo este hecho en datos, implica de media un **62.34 %** de mejora

con respecto a *Serving*, un **27.67 %** con CPU SYNC y por último, un **23.58 %** en relación a CPU ASYNC. Suponiendo, un aumento de rendimiento muy notable con cada uno de los casos planteados.

Capítulo 6

Conclusiones y Trabajo Futuro

Al comienzo de este proyecto se establecieron una serie de objetivos a resolver siendo el principal el establecimiento de una metodología para la portabilidad de redes neuronales a OpenVINO y por consiguiente el análisis de la mejora ofrecida por este.

Tras finalizar este proceso y efectuando un análisis de todo el trabajo realizado, se pueden extraer las siguientes deducciones:

- La metodología planteada permite realizar la portabilidad de prácticamente de cualquier red neuronal a OpenVINO y obtener resultados sobre este *toolkit*. Sin embargo, para obtener el máximo rendimiento por parte de este depende en gran medida del hardware disponible y las optimizaciones ofrecidas por Intel para ese dispositivo.

- La mejora ofrecida por el *toolkit* es **muy significativa** en cualquiera de los casos planteados. Dentro de estos, se puede establecer que, en cuanto a la selección disponible de dispositivos, la configuración **MULTI** es con la que se alcanza un mayor rendimiento y por otra parte, con el *stick* NCS2 se registran tiempos que se encuentran muy por debajo del resto. Aunque, hay que resaltar la potencia proporcionada por ambos dispositivos no es para nada la misma y además, la accesibilidad y coste son puntos a favor de este último.

  En relación al tipo de inferencia, se puede afirmar que la configuración **ASYNC**, ofrece un mayor rendimiento con los datos disponibles, independiente de la configuración de dispositivos establecida. Aunque, hay que resaltar que la diferencia es mucho más abultada en el caso del NCS2.

- Los resultados obtenidos en la fase de entrenamiento avalan que el rendimiento obtenido es **superior** al registrado en el *paper*. En lo que a precisión y perdida se refiere, todos los casos cumplen o mejoran los datos extraídos en este. El único factor que no se ha llegado a obtener de manera tan clara es el tiempo de entrenamiento con GPU, el cuál, no ha llegado a ser el resultado esperado. Sin embargo, con las optimizaciones realizadas, las métricas obtenidas identifican **una mejora en todos los aspectos**, formando la mejor configuración de entrenamiento obtenida.

- El planteamiento de los escenarios en el análisis de rendimiento del entrenamiento muestran que puede ser **excesivo** el número de iteraciones definidas. Como se puede atisbar en la tabla correspondiente al escenario 1, los valores de precisión se encuentran por encima tanto con los datos de *training* como con los de *testing*. Por lo tanto, una de las posibles valoraciones a realizar es que disminuyendo el número de iteraciones los resultados pueden llegar a ser muy satisfactorios, generando que el tiempo de ejecución se disminuya muy considerablemente.

- El análisis gráfico del modelo muestra algunas carencias albergadas en este que no podrían ser atisbadas por los valores pico de su precisión o el valor mínimo de pérdida. Esa indagación del modelo resultante ofrece algunos indicios sobre su posible mal comportamiento, como puede ser el sobreajuste o una extraña trayectoria de aprendizaje. Por lo tanto, resulta indispensable realizar un análisis gráfico de las métricas obtenidas tras revisar que los valores pico son adecuados.

Estas ideas planteadas corresponden a los datos y valoraciones realizadas durante todo el proyecto, teniendo en cuenta los datos y procedimientos realizados en este. Una vez desentrañadas las conclusiones obtenidas en base a los objetivos propuestos, se pueden abordar una serie de líneas futuras a este o, al menos, algún contexto en el que el contenido ofrecido pueda llegar a ser útil. Las razones por las que no hayan sido implementadas pueden ser varias: falta de tiempo, no se encontraba en el marco planteado del trabajo, no llego a contemplarse, etc. Algunas de estas ideas son:

- Prueba de rendimiento de la red en un hardware de bajo consumo (como puede ser una

Raspberry Pi) usando el *toolkit* para su posterior implementación en un escenario real. El cual, podría tratarse de una configuración IoT con una serie de *routers* que abastecen de datos al modelo o, por otro lado, siguiendo el paradigma de computación en el borde (*edge computing*).

- Medir el rendimiento de la red con el uso de varios *sticks* NCS2 en paralelo, extrayendo el nivel de mejora con el mejor caso propuesto en este trabajo y la diferencia de tiempo con respecto a un solo dispositivo de este tipo.

- Comparativas entre la API en Python y C/C++, debido a que la primera de ambas tiene ciertas limitaciones y no ofrece un buen soporte para diseños multiproceso o multihilo. Por tanto, la API de C/C++ ofrecerá un mayor rendimiento permitiendo este tipo de implementaciones. Además, el *toolkit* se encuentra escrito integramente en ese lenguaje.

- Rediseño de la topología del modelo para medir su precisión al usar un número inferior de capas. Los datos obtenidos por este estudio servirían como referencia para decidir si es mejor cambiar la topología o reducir el número iteraciones en la fase de entrenamiento o una combinación de ambas. De tal forma, establecer la mejor relación precisión-tiempo de esos escenarios.

- Implementación de esta topología con un paradigma de abastecimiento de datos en *streaming* y/o el uso de vídeo como dato de entrada, volviendo a entrenar el modelo y viendo su adecuación a este nuevo escenario.

Por lo tanto, estas suponen como algunas de las posibles lineas de investigación que se pueden extraer de este proyecto. No obstante, hay que remarcar que estas tecnologías se encuentran actualmente en desarrollo y son relativamente novedosas, por lo que es posible que surjan diferentes alternativas a las expuestas. De tal manera que este trabajo pueda resultar una manera para contextualizar este tipo de herramientas o, por otro lado, constituir la base de otros proyectos. Pero como sucede en este campo de inteligencia artificial, los paradigmas y tecnologías se encuentran en continuo cambio.

Bibliografía

[1] Assaad MOAWAD. Neural networks and back-propagation explained, 2018. URL https://medium.com/datathings/neural-networks-and-backpropagation-explained-in-a-simple-way-f540a3611f5e.

[2] Ujwala Bhangale, Surya Durbha, Abhishek Potnis, and Rajat Shinde. Rapid earthquake damage detection using deep learning form vhr remote sensing images. *IGARSS 2019*, pages 2654–2657, 2019.

[3] Rajkumar Buyya and Satish Narayana Srirama. *Fog and Edge Computing: Principles and Paradigms*. John Wiley and Sons, Inc., 2019.

[4] Ali Shatnawi, Ghadeer Al-Bdour, Raffi Al-Qurran, and Mahmoud Al-Ayyoub. A comparative study of open source deep learning frameworks. 04 2018. doi: 10.1109/IACS.2018.8355444.

[5] Intel AI. Documentación de la herramienta de openvino. URL http://docs.openvinotoolkit.org/latest/index.html.

[6] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[7] Peter Norvig and Stuart J.Russell. *Artificial Intelligence: A Modern Approach*. Pearson, 1995.

[8] Sakshi Indolia, Anil Kumar Goswami, S.P.Mishra, and Pooja Asopa. Conceptual understanding of convolutional neural network - a deep learning approach, 2018. URL https://www.sciencedirect.com/science/article/pii/S1877050918308019?via%3Dihub.

[9] Standford University. Cs231n convolutional neural networks for visual recognition. URL http://cs231n.github.io/convolutional-networks/#overview.

[10] Ilija Mihajlovic. Everything you ever wanter to know about computer vision, 2019. URL https://towardsdatascience.com/everything-you-ever-wanted-to-know-about-computer-vision-heres-a-look-why-it-s-so-awesome-e8a58dfb641e.

[11] University of Tartu. Digital image processing, 2014. URL https://sisu.ut.ee/imageprocessing/book/1.

[12] Jason Brownlee. 9 applications of deep learning for computer vision, 2019. URL https://machinelearningmastery.com/applications-of-deep-learning-for-computer-vision/.

[13] Google Developers. Tensorflow: Kit de herramientas, 2020. URL https://developers.google.com/machine-learning/crash-course/first-steps-with-tensorflow/toolkit.

[14] Intel Developer Zone. Intel movidius vpu, 2020. URL https://software.intel.com/en-us/iot/hardware/vision-accelerator-movidius-vpu#specifications.

[15] Juan Mas Aguilar. Tfm:aceleración de la fase de inferencia en redes neuronales profundas con dispositivos de bajo coste y consumo. pages 25–32, 2020.

[16] Jeshwari Banakar, Stefan Steinke, Bo-Sik Lee, M.Balakrishnan, and Peter Marwedel. Scratchpad memory: A design alternative for cache on-chop memory in embeedded systems, 1999. URL http://www.ann.ece.ufl.edu/courses/eel6935_10spr/papers/scratchpad_memories.pdf.

[17] Prakhar Ganesh. Types of convolution kernels: Simplified, 2019. URL https://towardsdatascience.com/types-of-convolution-kernels-simplified-f040cb307c37.

[18] Rochak Agrawal. Optimization algorithms for deep learning, 2019. URL https://medium.com/analytics-vidhya/optimization-algorithms-for-deep-learning-1f1a2bd4c46b.

[19] Jason Brownlee. Understand the impact of learning rate on neural network performance, 2019. URL https://machinelearningmastery.com/understand-the-dynamics-of-learning-rate-on-deep-learning-neural-networks/.

[20] Uniqtech. Understand cross entropy loss, 2019. URL https://medium.com/data-science-bootcamp/understand-cross-entropy-loss-in-minutes-9fb263caee9a.

[21] Google Developers. Curso de aprendizaje automatico, 2019. URL https://developers.google.com/machine-learning/crash-course/classification/accuracy.

[22] Google Developers. Estudio detallado del aa: Entrenamiento y perdida, 2019. URL https://developers.google.com/machine-learning/crash-course/descending-into-ml/training-and-loss.

[23] Joseph Bullock. Freezing a keras model, 2018. URL https://towardsdatascience.com/freezing-a-keras-model-c2e26cb84a38.

[24] Suhyun Kim. A beginners guide to cnn. URL https://towardsdatascience.com/a-beginners-guide-to-convolutional-neural-networks-cnns-14649dbddce81.

[25] Aurélien Géron. *Hands-on Machine Learning with Scikit-Learn, Keras and Tensorflow*. O´Reilly, 2 edition, 2019.

[26] François Chollet. *Deep Learning with Python*. Manning Publications, 2018.

[27] Sagar Sharma. Activation functions in neural networks. URL https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6.

[28] Song Yuheng and Yan Hao. Image segmentation algorithms overview. URL https://arxiv.org/ftp/arxiv/papers/1707/1707.02051.pdf.

[29] Rohit Gandikota. Image processing tecniques for indoor and outdoor self driving cars, 2018. URL https://arxiv.org/pdf/1812.02542.pdf.

[30] Jason Brownlee. Difference between a batch and an epoch in a neural network, 2018. URL https://machinelearningmastery.com/difference-between-a-batch-and-an-epoch/.

[31] Anukrati Mehta. Types of neural networks, 2019. URL https://www.digitalvidya.com/blog/types-of-neural-networks/.

[32] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vicent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions, 2014. URL https://arxiv.org/pdf/1409.4842v1.pdf.

[33] Dhairya Kumar. Introduction to openvino, 2019. URL https://towardsdatascience.com/introduction-to-openvino-897e705a1f0a.

[34] Ambika Choudhury. Tensorflow vs keras: Which one should you choose, 2019. URL https://analyticsindiamag.com/tensorflow-vs-keras-which-one-should-you-choose/.

Apéndice A

Introduction

Currently, the artificial intelligence is one of the main trends in various sector as health, security, and so on. This one offers a wide variety of possible applications, which in some cases allow us to enhance the internal processes and increase efficiency. Inside this one, it finds the field of Deep Learning, which has everything related to neural networks considered as one of the most applied techniques.

The knowledge of this kind of technologies goes back some decades ago, where Frank Rosenblatt defined in 1957, the first model of a neural network, called perceptron[1]. This one is still used for building pattern recognition applications.

However, the discovery of these innovations has not gone hand in hand with the computing capacity available for its execution. Mainly, because it would require a vast amount of data and it was not available this computing power for processing in that time, due to the utilization of high mathematical complexity operations and management in parallel with its quantity of data.

The computing capacity has changed a lot these last years, where each time, the devices have a lower size, but with high-performance hardware. Generating the paradigm of Edge Computing [3] which tries to bring closer as much as possible the processing and storage of the data to the specific location where it is needed. Indeed, it enhances the response times and reduces the bandwidth with other components.

Hence, this paradigm requires increasing the efficiency between hardware and software for being able to offer the best support. This means that the hardware manufacturers have to generate their

[1] `https://www.pearsonhighered.com/assets/samplechapter/0/1/3/1/0131471392.pdf`

own tools for optimizing these processes.

A.1. Motivation

In the field of deep learning exist diverse frameworks available (such as Tensorflow, Pytorch, Caffe, and among others) [4] for putting into practice. However, the democratization of these do not currently exist. Therefore, the use of each one tends to depend on its features and the developer experience with it.

Thus, the hardware manufactures have to adapt its technology based on these frameworks, making this optimization more complex and difficult. These technologies are still on development, so that there are no strong evidences of having a real enhancement related to the software designed to design and implement the network. Even though, if there was one, it would be unknown how much does it mean. Additionally, the network efficiency depends largely on the features of this and the data provided. Being this fact particularly notable in networks created from scratch.

As it is a software so recent, the documentation and process are not well-defined, therefore this project will try to cover all these questions, implementing a neural network from scratch for a particular problem.

A.2. Milestones

The main goal is to establish a methodology for porting neural networks to Intel devices, through the OpenVINO toolkit for speeding-up the inference phase. To this end, it would be convenient to break down in other low-level purposes to be able to establish a preliminary flow to follow and a reference in detail of the possible parts which including the development of this methodology. The milestones proposed based on the main one are:

- Optimization and analysis of the provided neural network
- Portability and implementation of the generated models for any Intel device, through the usage of the OpenVINO toolkit
- Optimization of the inference engine provided by the toolkit

- Getting the results and make comparatives of the performance between the different available devices

A.3. Work Plan

Once it is defined as the main goals disposed in this project, it will establish the planning of its development in the timeslot defined in this project. For its representation, it has been used a Gantt´s diagram (see Figure A.1), which defines in an easy way the duration and competences of the milestones established:

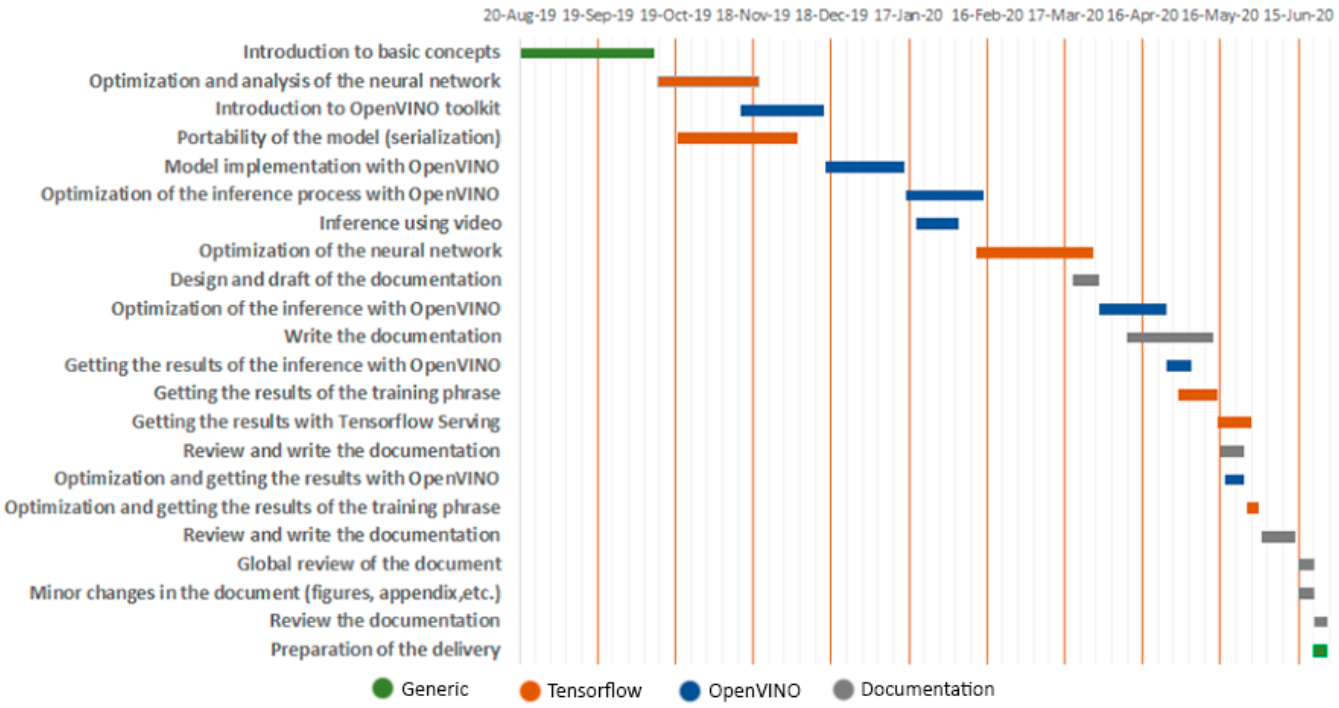


Figura A.1: Gantt´s diagram of the project planning.

In this way, it will be covered and structured all the aspects involved in this project. Although it is needed to mention that all the parts which are not included in this document such as code, artifacts, and among others. They are placed in this repository[2].

[2] `https://github.com/adri1197/DP_Image-Binary-Classification`

A.4. Structure of the Document

Taking into account the milestones previously specified, this document boards the whole process from the design of the neural network to its implementation and getting results of quality and performance. Therefore, this document will follow the order mentioned, pointing out the process and the theory behind just in cases where it is considered necessary for the comprehension of the ideas or the following ones.

Thus, this report will contain (following the order which they are displayed):

- **State of art:** Description of the situation and explanation of the main concepts which represent the basic theory used in the process of this project. Moreover, it will be included all the tools hardware and software used during the development.

- **Description of the model:** Explanation in detail of the different parts in which a neural network is formed by to be able to be implemented. It is about the different pieces in which this model is composed. Through this explanation, it is looking for providing more general concepts to give a wide view, that is to say, learning the internal way of working of the network through this specific case.

- **Portability and inference in Intel devices**: Process description of the implementation of a functional neural network using the toolkit created by Intel for being able to adapt in a homogeneous way a neural network in any device belonging to the company. The steps to be followed, in addition to diverse functionalities and applications which have been implemented.

- **Results obtained:** Exposition and measure of the performance of the neural network with different configurations (number of data, network configuration, and so on) in the training phase and with the OpenVINO toolkit, implementing the network in different Intel devices. With all that, looking for comparing different performance results and get a series of conclusions based on these.

- **Conclussions and future work:** Summary of the different responses or ideas contributed by this project about the topic proposed, using data and discoveries exposed during these

previous sections. Furthermore, from these, it will reflect a series of ideas that are considered as feasible or good to be implemented as a continuation of this work (whether it is considered viable in the future), but for diverse factors (lack of time, resources or it is not simply a part of the spectrum planned for the project) have not been considered as a possibility to be implemented.

Apéndice B

Redes Neuronales en Detalle

En este apéndice del documento se trata de explicar en detalle y de manera relativamente sencilla los conceptos básicos relacionados con redes neuronales, que han formado parte de la base teórica para el entendimiento de su funcionamiento y, por tanto, han resultado necesarias en el desarrollo del trabajo. Esta información puede resultar ciertamente útil para aportar un cierto bagaje al lector que no esté familiarizado con este campo.

Por definición, una red neuronal trata de imitar el comportamiento de una red biológica, mediante su especificación en un modelo matemático.

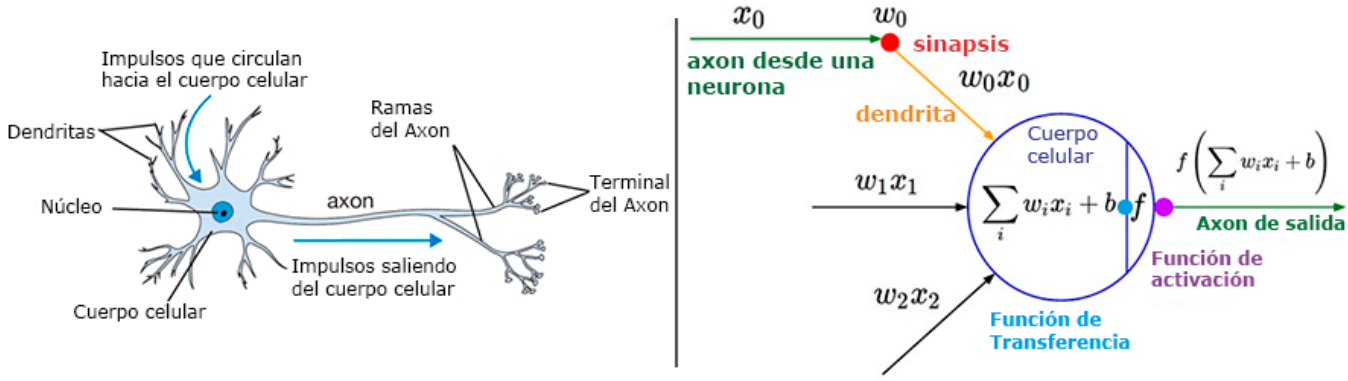


Figura B.1: Comparativa entre una neurona biológica y su modelo matemático.

La Figura B.1 representa ese modelo que sigue la red en cada capa o *layer*, generando una serie de elementos que forman parte, por defecto, de cada una:

- **Datos de entrada o *Inputs*:** Genera una capa específica para adaptar e introducir los

datos a la red.

- **Pesos o *Weights*** (W_i)**:** Es el número que almacena el aprendizaje realizado durante el entrenamiento. Este valor va cambiando durante dicho proceso para conseguir adecuar la entrada con la salida esperada.

- **Función de transferencia:** Sirve como entrada para la función de activación y permite agrupar el contenido de los anteriores valores de entrada x_i y pesos w_i. Consiste en la siguiente operación (siguiendo la notación de la Figura B.1):

$$\sum_{i=0}^{N}(x_i + w_{ij}) * b \tag{B.1}$$

- **Función de activación:** En algunas ocasiones resulta interesante definir una función en base a la *función de transferencia*. Por ejemplo: aplicar una función no-lineal.

  Esta define la salida de un nodo en base a los datos recibidos de la entrada o del nodo de una capa anterior. Por tanto, se puede determinar la cantidad de información en la que ese nodo puede ser relevante o no para la predicción del modelo.

  A modo de ejemplo, supongamos que en una determinada capa la información resultante es negativa. Esto puede generar un error en la predicción ya que el error se irá propagando por todas las capas siguientes. Para ello, se puede aplicar una función que permita eliminar los valores negativos. Por ejemplo: la función *ReLU*, especificada en la topología del modelo usado.

En algunos libros no se realiza una distinción entre *Función de transferencia* y *Función de activación*, se considera que ambas representan la función de activación. En este caso, se ha preferido hacer esa distinción para que resulte más clara la explicación matemática de la red neuronal más básica.

B.1. Proceso de Aprendizaje de la Red

La configuración y disposición de los elementos anteriormente mencionados genera una **topología**, produciendo un comportamiento y **características** específicas en esa NN (*Neural Network*).

La buena definición de este proceso establece el comportamiento de la red, teniendo como objetivo alcanzar el comportamiento deseado, es decir, la salida esperada en cada caso. Sin embargo, son a su vez los datos y la problemática a resolver con ellos los que establecen lo anterior. Provocando que posteriormente y mediante una serie de iteraciones definidas previamente, se tratará de ajustar los datos obtenidos por la red con el resultado esperado. Este proceso definido con anterioridad se divide en las siguientes fases:

- **Fase de Entrenamiento:** Usando unos datos de entrada o patrones de entrenamiento, se determinan los pesos (parámetros) que definen el modelo. Se calculan de manera iterativa de acuerdo con los valores de entrenamiento, con el objeto de minimizar el error cometido entre la salida obtenida por la red neuronal y la deseada (mostrado en la Figura B.2).

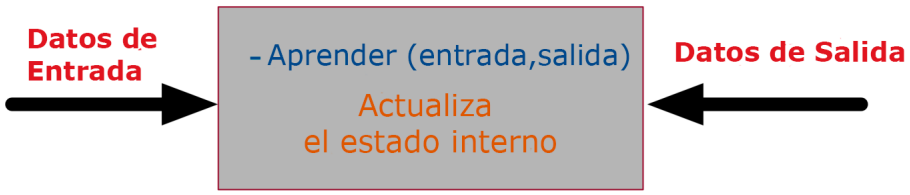


Figura B.2: Concepto de entrenamiento en redes neuronales.

En el campo de *Deep Learning* suele realizarse la mecánica de aprendizaje supervisado, aunque existen otros tipos que no son tan usuales en ese ámbito. La técnica de aprendizaje a utilizar la definen los datos disponibles y la problemática a resolver, por lo que, es conveniente conocer los distintos tipos de paradigmas existentes en la fase de aprendizaje:

- **Supervisado:** Es la forma más popular de entrenar un modelo. Los datos para el entrenamiento están constituidos por varios patrones de entrada y de salida, es decir, tanto la entrada como la salida son conocidas previamente a la predicción. Este tipo de aprendizaje utiliza una técnica de entrenamiento denominada **propagación hacia atrás** (ver apartado B.1.1).
- **No supervisado:** El conjunto de datos de entrenamiento consiste solo en los patrones de entrada. Por tanto, la red aprende en base a la información recogida por los patrones de entrenamiento anteriores.

- **Por refuerzo:** Trata de descubrir las acciones que se deben tomar para maximizar la señal de recompensa. Al agente no se le dice que acciones tomar, sino que debe experimentar para encontrar las acciones que conllevan una mayor recompensa. Este tipo de aprendizaje se trata sobre todo en el ámbito del aprendizaje automático (*Machine Learning*) y una de sus aplicaciones es en el campo de la robótica. Este proceso lo representa la Figura B.3.

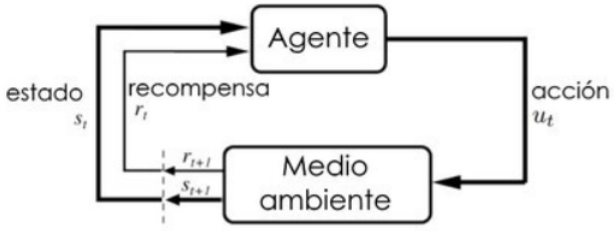


Figura B.3: Concepto de aprendizaje por refuerzo o *reinforcement learning*.

- **Fase de Prueba:** Se realiza una comparación entre la predicción obtenida y los resultados esperados. Es posible que en la fase anterior, el modelo se ajuste demasiado a las particularidades presentes en los datos de entrada, perdiendo su habilidad para aprender (*overfitting*). Para evitar este tipo de problemas es aconsejable utilizar un segundo grupo de datos diferentes a los de entrenamiento, datos de *validación*, que permita controlar el proceso de aprendizaje y además, probar el comportamiento de la NN con unos datos con los que no ha sido entrenado. De esta forma, podremos visualizar de una mejor manera la actuación del modelo en una simulación más cercana a la realidad cuando se implemente.

B.1.1. Propagación hacia atrás

Este modo de propagación se utiliza como método de cálculo del gradiente en los algoritmos de aprendizaje supervisado utilizados para **entrenar** redes neuronales[1]. Como ha sido mostrado en la Figura B.2, el proceso de entrenamiento coge los datos de entrenamiento con las salidas deseadas y actualiza su estado interno (pesos) para obtener una salida lo más cercana posible a la deseada.

El desarrollo de este método se divide en 7 pasos (recogidos de manera resumida en la figura

[1] `https://en.wikipedia.org/wiki/Backpropagation`

B.5):

1. **Inicialización del modelo:** El primer paso del aprendizaje comienza con la primera hipótesis, la cual, siempre resulta ser desconocida. Por lo que una práctica muy común es realizar una **inicialización aleatoria** del modelo y tras un proceso iterativo de aprendizaje, llegar a obtener nuestro pseudo-modelo ideal.

2. **Propagación hacia adelante:** El paso posterior consistiría en **comprobar** la actuación del modelo. Para ello, se introducen los datos en el modelo y se obtienen los resultados aleatorios de este.

3. **Función de pérdida:** En este momento estarían disponibles tanto los datos de salida deseados como los datos aleatorios obtenidos del modelo. Con la denominada **función de pérdida**, es posible medir como la red está manejando su objetivo de acercar lo más posible los datos deseados con las predicciones realizadas. La manera más sencilla de medirlo sería con la diferencia en valor absoluto entre el valor actual y el deseado.

   Sin embargo, varias situaciones pueden lidiar con el mismo número total de errores: por ejemplo, muchos pequeños errores o pocos pero muy grandes, pueden resultar ser el mismo error. Como el objetivo es que la predicción funcione correctamente en cualquier situación, es preferible tener una distribución con muchos errores de pequeña factura.

   Por tanto, el objetivo de esta fase es minimizar esta métrica con el fin de obtener una predicción lo más cercana posible al valor objetivo.

4. **Diferenciación:** Se puede utilizar cualquier técnica de optimización que modifique los pesos internos de las redes, con el propósito de minimizar la función de pérdida total que hemos definido antes. Estas técnicas pueden incluir tanto algoritmos genéticos como una búsqueda usando fuerza bruta. Esta última podría funcionar si hay un número relativamente pequeño de parámetros y la precisión no resulta muy importante. Aunque en el caso de procesamiento de imágenes esta idea es totalmente inviable, debido a la potencia de computo necesaria.

   Por ello, usando diversas técnicas matemáticas podemos hallar la tendencia de la función

de pérdida. Una de ellas, es mediante la derivada de la función que nos permite conocer la velocidad en que los valores de la función cambian en un punto específico. De esta manera, obtenemos una técnica más rápida y precisa a la anterior propuesta.

Entonces, el proceso de aprendizaje sería el siguiente:

- Comprobar la derivada.
- Si es positiva, significa que el error aumenta si se acrecenta el peso. Por tanto, se debe reducir.
- Si es negativa, el error decrece si se aumenta el peso. Por lo que se debe acrecentar el peso.
- Si es 0, se ha alcanzado el punto de estabilización. No habría que hacer nada en ese caso.

5. **Propagación hacia atrás:** Obtener una función que agrupe toda la composición generada con todas las capas de la red resulta muy complicado. Pero, hay que recordar que la derivada se puede descomponer y esto hace que pueda ser propagada hacia atrás (explicado de manera simplificada en la Figura B.4).

   Si se genera una librería de funciones o capas **diferenciables** donde cada función se sabe cómo se propaga hacia adelante (directamente utilizando esa función) o hacia atrás (calculando la derivada de la función de pérdida), se puede componer cualquier red neuronal compleja. Lo único que se necesitaría es apilar un conjunto de llamadas durante el paso hacia adelante y sus parámetros. De esta manera, se puede saber la manera de propagar los errores hacia atrás usando las derivadas de esas funciones. Esta técnica se denomina **auto-diferenciación**.

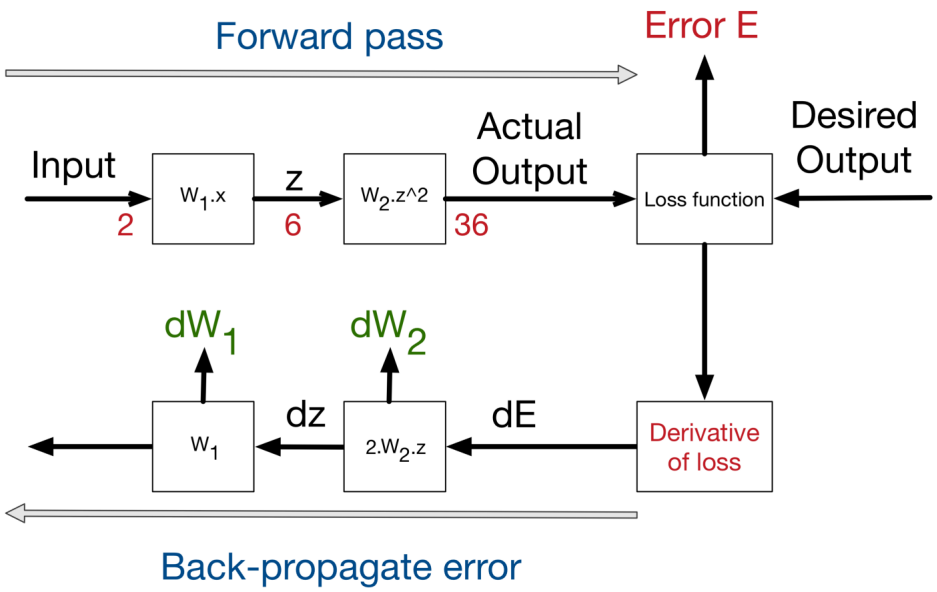


Figura B.4: Diagrama del proceso paso hacia adelante y propagación hacia atrás.

6. **Actualización de los pesos:** Con anterioridad se ha mostrado que las derivadas solo miden la tasa en la que el error cambia con respecto a los cambios en los pesos. Este cambio se realiza con modificaciones muy pequeñas, debido a que, por ejemplo, en distribuciones no lineales, un cambio muy grande en los pesos puede generar un comportamiento caótico en el modelo.

   La idea de la **tasa de aprendizaje** es introducir una constante que permita forzar que los pesos se actualicen de manera suave. Aunque este parámetro suele estar relacionado con una función que realiza esta tarea, denominados *optimizadores*. Un ejemplo de ellos sería: *Adam* (utilizada en la fase de entrenamiento del modelo), *Stochastic Gradient Descent*[2], etc.

7. **Iterar hasta que converja:** Desde que se actualizan los pesos hasta que la red aprende se necesitan una serie de iteraciones. Por tanto, se puede cuestionar cuantas son necesarias para conseguir la convergencia del modelo. Este hecho depende de muchos factores:

   - La **tasa de aprendizaje** establecida. Si es muy alta, el modelo aprenderá muy rápido, pero hay una mayor probabilidad de inestabilidad y resultados no óptimos.

[2] `http://cs231n.github.io/optimization-1/`

- La **definición de la red** (número de capas, como de complejas son las funciones lineales, etc.). Cuanto mayor sea el número de variables más tiempo tardará en converger, pero la precisión será mucho mayor.
- El método de **optimización** usado, algunas reglas de actualización de la red no están probadas que sean más rápidas que otras.
- La **inicialización** aleatoria de la red. Puede que genere un valor cercano a la solución óptima.
- La **calidad de los datos de entrenamiento**. Si no hay ninguna correlación entre los datos de entrada y los de salida, la red será capaz de aprender una correlación aleatoria, por lo que será muy complicado obtener buenos resultados.

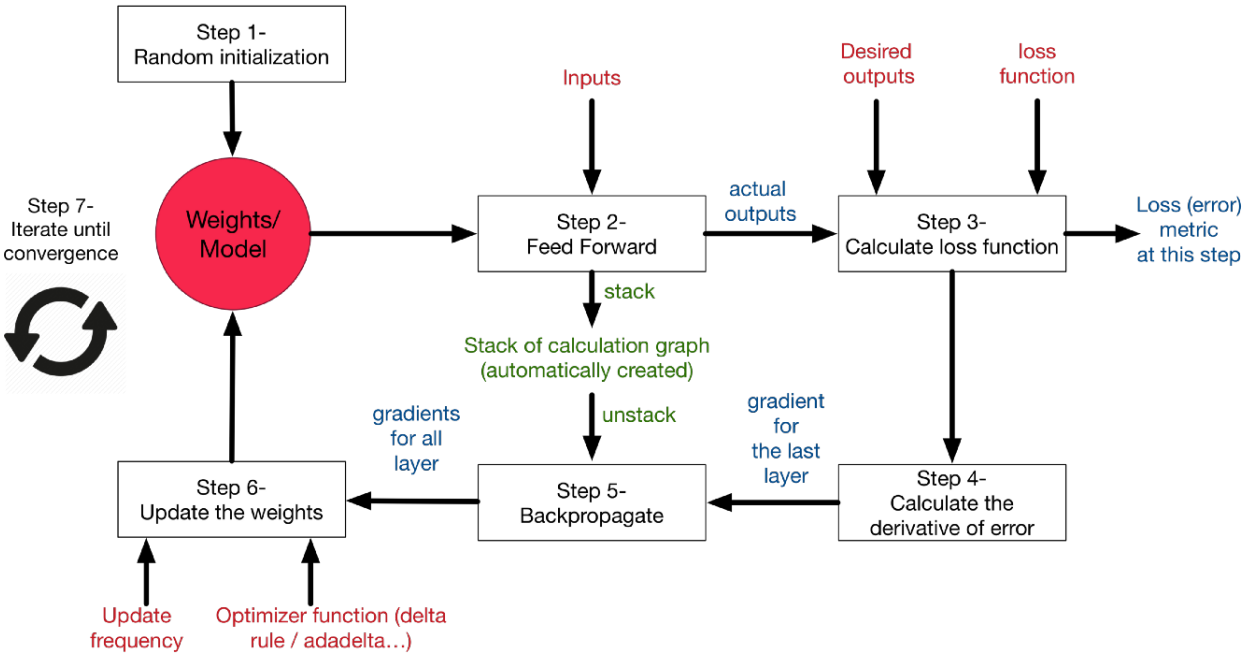


Figura B.5: Resumen del método de propagación hacia atrás o *backpropagation* [1].

Una vez establecido el proceso básico para poder obtener una red "funcional", entendiendo este término como un modelo que nos ofrece unos resultados que se ajustan en cuanto a datos y tiempo de ejecución a nuestra problemática y salida esperadas. Por ello, hay que tener en cuenta que la definición de la topología de la red neuronal resulta extremadamente importante para su posterior

éxito.

B.2. Tipos de Redes Neuronales

Durante el desarrollo de este concepto de red neuronal como modelo matemático, se han generado una serie de estructuras y procedimientos internos que se utilizan como estándar para ciertas finalidades de uso. Esto hace que se realice una clasificación de los diferentes tipos de redes en base a su topología y funcionamiento internos:

- **Feedfoward Neural Network - Artificial Neuron**: Representa el tipo más simple de red. Su modo de propagación es hacia adelante, esto quiere decir que los datos pasan a través de los diferentes nodos hasta que es alcanzado el de salida, es decir, que los datos se mueven únicamente hacia una dirección, a diferencia de otras redes más complejas. Se utilizan en tecnologías de reconocimiento facial y visión artificial [31]. Un ejemplo de esta sería la red mostrada en la Figura B.1.

- **Multilayer Perceptron** (MLP): Posee 3 o más capas. Se usa para clasificar datos que no pueden ser separados linealmente, utilizando funciones de activación no lineales (por ejemplo: tangente hiperbólica o logística). Este tipo se aplica en problemas relacionados con reconocimiento de voz y traducción automática. [31]

- **Convolutional Neural Network** (CNN): Utilizada sobre todo en problemas relacionados con el uso de imágenes, resulta un estandar en este tipo de problemas.

- **Recurrent Neural Network** (RNN): La salida de una capa específica es almacenada, nutriendo a la capa de entrada y ayudando en la predicción de las siguientes capas o *layers*. De esta manera, cada nodo recordará algún tipo de información relacionada con uno de la capa anterior. Si la predicción es incorrecta, el sistema autoaprende y trabaja para obtener la correcta predicción durante la propagación hacia atrás (mostrado el proceso en la Figura B.6). Su principal aplicación es en tecnología de autogeneración de texto o conversión de texto a voz [31].

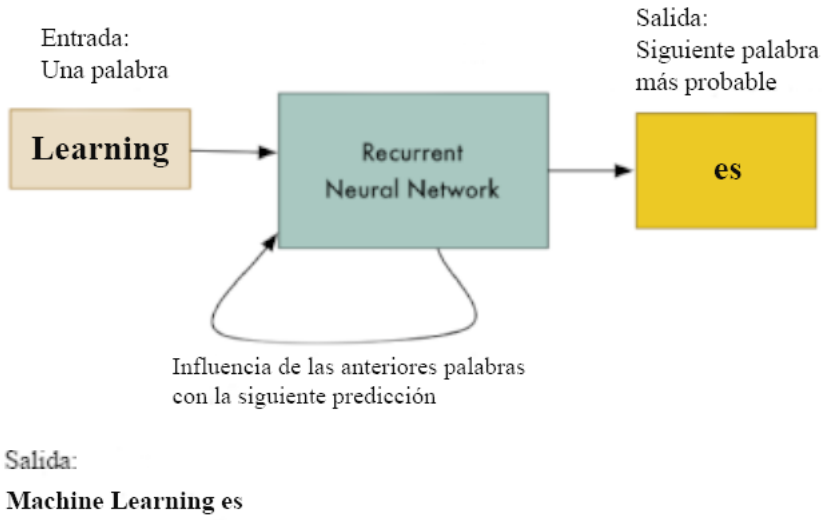


Figura B.6: Ejemplo del funcionamiento de *Recurrent Neural Network.*

- **Radial Basis Function Neural Network**: Considera la distancia de cualquier punto con respecto al centro. Posee dos capas y se usa en sistemas de almacenado de energía, permitiendo realizar esta tarea en el menor tiempo posible [31].

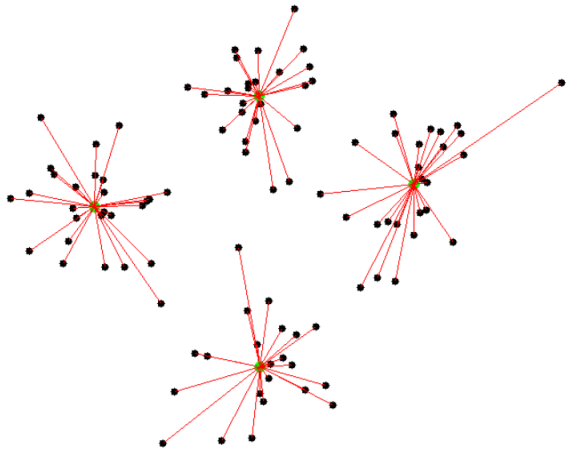


Figura B.7: Ejemplo del funcionamiento de *Radial Basis Function Neural Network.*

- **Modular Neural Network**: Poseen diferentes redes que funcionan independientemente realizando pequeñas tareas. Estas no interactúan entre ellas, es decir, trabajan de manera

individual hasta conseguir su objetivo. Aunque se trata de un sistema complejo, la demanda computacional no es tan alta, debido a que las redes funcionan en solitario [31].

- **Sequence-To-Sequence Models**: Está formada por dos *Recurrent Neural Networks*. En el que una de ellas funciona como codificador (entrada) y otra como descodificador (salida) (ver la Figura B.8). Este modelo se aplica cuando la entrada de datos no tiene la misma longitud que la salida. Se utiliza en sistemas de traducción automática y de pregunta-respuesta [31].

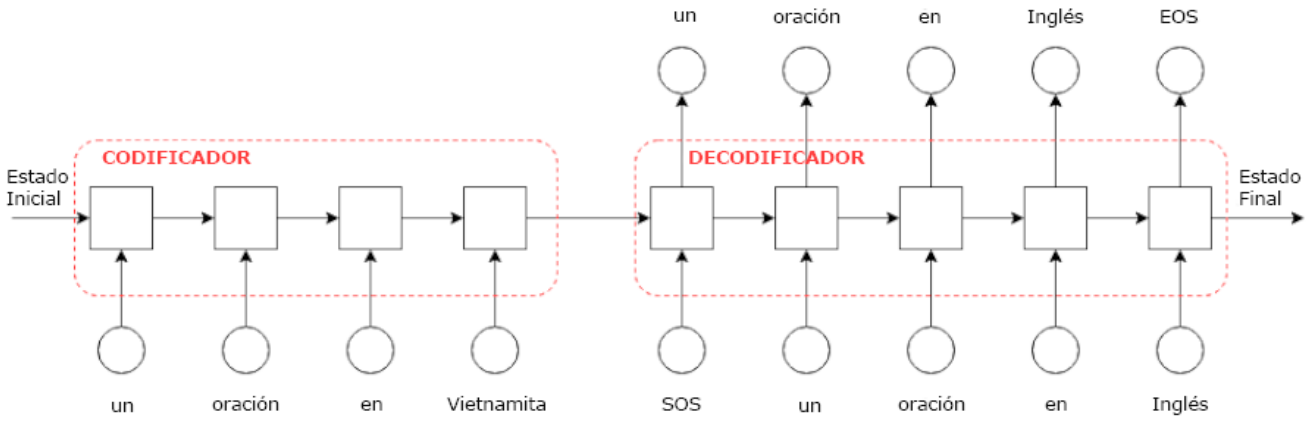


Figura B.8: Ejemplo del funcionamiento del modelo *Sequence to Sequence*.

B.3. Tipos de Aplicaciones

Dependiendo de la clase de información que se quiera obtener del modelo, existen diferentes tipos de aplicaciones a las que se aplica una NN. Dentro de cada una hay una infinidad de categorías a su vez, pero en términos más generales y siguiendo la principal convención que se utiliza en problemas de aprendizaje automático, se podría realizar la siguiente distinción:

- **Regresión:** Modelan la relación entre distintas variables proporcionando una predicción con valor continuo, es decir, que siguen una distribución lineal. Un ejemplo de este tipo podría ser la predicción de la temperatura en base a un histórico. Algunos de los algoritmos más usados son: Regresión Lineal (ML), Regresión Lineal Bayesiana (ML), etc.

- **Clasificación:** Se usan los datos para realizar una predicción sobre un conjunto de valores o categorías discretos y previamente conocidos. Representa el tipo de problema que se aborda

en este proyecto, pero en este caso, con el uso de imágenes como datos.

En este tipo de problemas resulta muy útil el uso de la matriz de confusión para poder analizar los resultados predichos con los valores reales.

- **Procesamiento de Datos:** Tiene el objetivo de filtrar, agrupar, separar o comprimir los datos para obtener una distribución, formato o simplemente unas características concretas.

En el ámbito de este proyecto, las categorías utilizadas no se ciñen a esta descripción ya que se trata de problemas relacionados concretamente con imágenes.

Apéndice C

Inferencia utilizando vídeo

El proyecto tiene como objeto de análisis la red neuronal con el conjunto de datos proporcionado para la clasificación binaria de zonas dañadas. Sin embargo, al indagar en este campo y revisando la información proporcionada por las predicciones obtenidas (*insights*), se abrió adicionalmente una nueva línea de desarrollo para su implementación con vídeo.

En realidad, el funcionamiento es ciertamente parecido al realizado con imágenes, ya que un vídeo esta compuesto por un conjunto de fotogramas (*frames*). Por lo tanto, el proceso consiste en obtener el fotograma, preprocesarlo para el modelo e inferir esa imagen. Mientras tanto, se realiza el preprocesamiento de la imagen siguiente, haciendo que una vez se obtenga la predicción de la primera imagen, la segunda se encuentra en la fase de inferencia. Esta consecución de pasos podría recordar a la inferencia asíncrona por su dinámica y plantea la posibilidad de generar lotes (*batches*) para procesar pequeños bloques del vídeo. Aunque, en este contexto, su uso con más de un elemento por cada uno no tiene mucho sentido, debido a que se busca obtener el resultado del *frame* lo antes posible.

El propósito de este experimento adicional consiste en la familiarización de la inferencia con vídeo, tanto con el *toolkit* como de manera conceptual, y la comprobación de la certeza de las predicciones en base a las imágenes mostradas, debido a que la red empleada había sido entrenada inicialmente con las imágenes del dataset. Y por consiguiente, observar la veracidad de este con un conjunto de datos totalmente "desconocido". Sin embargo, esto plantea un problema en cuanto a que el vídeo utilizado debe tratar sobre lo mismo y la vista debe ser muy parecida (cenital), además de la calidad, marcas de agua y diversas topografías que se pueden encontrar en este.

Afortunadamente, fue posible encontrar uno de similares características. Este proviene del huracán Michael ocurrido en la zona de México en el año 2018[1].

Con todo ello, se consiguió efectuar la implementación con vídeo, obteniendo resultados relativamente prometedores en ámbito general. Principalmente, porque resulta muy complicado medir la pertenencia del *frame* a una categoría u a otra. Por lo que, visualmente las predicciones resultan bastante fidedignas con respecto a la imagen mostrada. A modo de ejemplo, las Figuras C.1a y C.1b muestran los resultados extraídos.

(a) Zona Dañada. **(b)** Zona No Dañada.

Figura C.1: Fotogramas obtenidos en la inferencia con vídeo.

Esta prueba supone un primer acercamiento a las inferencias utilizando un conjunto de fotogramas como dato de entrada. Por lo que, esta disposición puede suponer una próxima linea de investigación en base a este proyecto, tal y como se detalla en los posibles trabajos futuros especificados en el Capítulo 6.

[1] `https://www.nytimes.com/video/us/100000006156817/mexico-beach-hurricane-michael.html`

Apéndice D

Conclusions and Future Work

At the beginning of the project, it established a series of milestones to solve being the establishment of a methodology for the portability of neural networks to OpenVINO, and therefore the analysis of the enhancement offered by it.

After finalizing this process and applying an analysis of the whole project, we can extract the following deductions:

- The methodology planned allows us to make the portability of any neural network to OpenVINO and getting results on this toolkit. However, to get the best performance depends largely on the hardware available and the optimizations provided by Intel for this device.

- The enhancement provided by the toolkit is **very significant** in any case. Within these, it can establish that the **MULTI** configuration is the one which reaches the best performance, and, in another way, with the stick NCS2 the times registered are so far below the other ones. Although, it has to be highlighted the power provided by both devices is not at all the same, being points in favor of the last one, the accessibility and the cost.

  Related to the type of inference, it can say that the configuration ASYNC offers a better performance with the available data, independently of the configuration of the devices established. However, it has to be highlighted that the difference is much bulkier than the NCS2 case.

- The results obtained in the training phrase endorse that the performance got is **superior** to the one registered in the paper. In terms of accuracy and loss, all the cases fulfill or enhance

the data extracted on this. The only factor that it has not been able to get clear is the training time with GPU, which it has not been the expected result. Nevertheless, with the optimizations made, the metrics got to identify an **enhancement in every aspect**, forming the best training configuration got.

- The approach of the scenarios shows that it can be **excessive** the number of iterations defined in the analysis of the training performance. As you can see in the table correspondent to the first scenario, the accuracy values are on top with both training and testing data. Hence, one of the possible valuations to make is that if decreasing the number of iterations, the results can reach a successful performance, generating that the execution time considerably decreases.

- The graphic analysis of the model shows some lack of shelter on this one that could not be seen by the peak values of the accuracy or minimum value of the loss. This inquire of the resultant model provides some insights about its possible bad behavior, such as the overfitting or a strange learning trajectory. Thus, it is indispensable to make a graphic analysis of the obtained metrics, after reviewing that the peak values are right.

These ideas planned correspond to data and valuations made during the whole project, keeping in mind the data and procedures done on this. Once unraveled the obtained conclusions based on the milestones proposed, it can be boarded a series of future lines of this, or at least, some context where the offered content can be useful. The reasons behind the non-implemented features can be several: lack of time, are not part of the framework planned for the project, did not contemplate, etc. Some of these ideas are:

- Test the network performance on a low-powered hardware (such as Raspberry Pi) using the toolkit for its next implementation in a real scenario. Which one can be an IoT configuration with a bunch of routers that supply of data to the model or, on the other hand, following the Edge Computing paradigm.

- Measure the network performance using various NC2 sticks in parallel, extracting the level of

improvement with the best scenario proposed in this project and the time difference regarding only one device of this type.

- Comparative between the Python and C/C++ API, due to the first of both has certain limitations and does not offer good support for multithreading and multiprocess designs. Thus, the C/C++ API will offer a better performance allowing this kind of implementations. Furthermore, the toolkit is entirely written in this programming language.

- Comparative between the Python and C/C++ API, due to the first of both has certain limitations and does not offer good support for multithreading and multiprocess designs. Thus, the C/C++ API will offer a better performance allowing this kind of implementations. Furthermore, the toolkit is entirely written in this programming language.

- Implementation of this topology with streaming data and/or the use of video as input, returning to train the model and watching its adequation to this new scenario.

Hence, these are some of the possible lines of investigation which can be extracted from this project. However, it has to highlight that these technologies are still in development and relatively new. Thus, it may arise different alternatives to those exposed. In such a way that this project may result in a manner of contextualizing this kind of tool or, on the other hand, constitute a base for other projects. But as it happens in the field of artificial intelligence, the paradigms and technologies are in a continuous change.