Optimización del tiempo de inferencia de un modelo de machine learning usando OpenVINO y despliegue del modelo en un entorno cloud

Inference time optimization on a machine learning model using OpenVINO and deployment of the model in a cloud environment

Andrés Ortiz Loaiza

GRADO EN INGENIERÍA DE COMPUTADORES
FACULTAD DE INFORMÁTICA - UNIVERSIDAD COMPLUTENSE DE MADRID

Trabajo Fin de Grado en Ingeniería de Computadores

Madrid, junio de 2020

Directores:

Bernabé García, Sergio
González Calvo, Carlos

Índice general

Índice general II

Índice de figuras V

Índice de tablas VII

Resumen VIII

Abstract IX

1. Introducción 1

1.1. Motivación y objetivos 1

1.2. Estado del arte 3

1.2.1. Concepto Deep Learning 4

1.2.2. Redes neuronales en el tratamiento de imágenes 6

1.3. Plan de trabajo 7

1.4. Organización de esta memoria 8

2. Entrenamiento del modelo mediante Google Colab 9

2.1. Modelo propuesto 10

2.2. Entorno Google Colab 15

3. Tecnología OpenVINO 17

3.1. Herramientas que lo componen 18

3.1.1. Optimizador de modelos de Deep Learning 18

3.1.2. Interfaz de inferencia de modelos de Deep Learning 19

3.2. Conversión del modelo a la plataforma OpenVINO 19

3.3. Inferencias. TensorFlow vs OpenVINO 21

4. Arquitectura Cloud propuesta **25**

4.1. Descripción del entorno y sus diferentes componentes 26

4.1.1. Google Storage 26

4.1.2. Google BigQuery 27

4.1.3. Google Pub/Sub 28

4.1.4. Google Compute Engine 28

4.1.5. Google Cloud Function 29

4.1.6. Container Registry 29

4.2. Explicación del flujo de datos 29

4.3. Codificación de los servidores web 30

4.3.1. Framework FastAPI 34

4.3.2. Framework Flask 35

4.4. Encapsulación de entorno con Docker 35

5. Resultados experimentales **39**

5.1. Dataset y Hardware 39

5.2. Rendimiento en fase de entrenamiento 40

5.2.1. Configuración de 100 Epochs y 256 Batch-size 41

5.2.2. Configuración de 175 Epochs y 256 Batch-size 41

5.2.3. Configuración de 200 Epochs y 256 Batch-size 42

5.3. Rendimiento en fase de inferencias 43

5.3.1. Rendimiento de acierto en la predicción. 44

5.3.2. Rendimiento en tiempo de inferencia. 44

5.3.3. Rendimiento en tiempo de inferencia con distinto hardware. 45

5.3.4. Rendimiento en tiempo de inferencia con distinto framework web. . . 46

5.3.5. Rendimiento en tiempo de inferencia con distinto framework web y hardware. 47
5.3.6. Número de peticiones por segundo soportadas por el servidor. 48
5.3.7. Rendimiento de inferencia en un entorno local. 48
5.4. Costes del proyecto 50

6. Conclusiones y trabajo futuro **52**
6.1. Conclusiones 52
6.2. Trabajo futuro 53

Bibliografía **57**

A. Introduction **58**
A.1. Motivation and objectives 58
A.2. State of the art 59
A.2.1. Deep Learning concept 61
A.2.2. Neural networks in image processing 61
A.3. Work plan 64
A.4. Organization of this project 64

B. Conclusions and future work **66**
B.1. Conclusions 66
B.2. Future work 67

Índice de figuras

1.1. Encuesta sobre lenguajes de programación usados en StackOverflow 2019. . . 5
1.2. Ejemplo de perceptrón. 6
1.3. Plan de trabajo. 7

2.1. Diferencia de núcleos para procesar cargas de trabajo en paralelo de forma eficiente entre CPU y GPU. 10
2.2. Ejemplo de imagen capturada por el satélite GeoEye-1. (a) Imagen de un terreno destruido. (b) Imagen de un terreno en buenas condiciones. 11
2.3. Función de activación Relu. 12
2.4. Topología de la red del modelo Deep Learning usado para la clasificación de daños. 14

3.1. Arquitectura de optimización de modelos con OpenVINO. 18
3.2. Arquitectura de TensorFlow serving. 22

4.1. Arquitectura Cloud. 26
4.2. Arquitectura Google Cloud. 30
4.3. Petición HTTP al servidor. 31
4.4. Arquitectura de la máquina virtual de OpenVINO. 36
4.5. Arquitectura de la máquina virtual de TensorFlow. 37

5.1. Resultados de entrenamiento del modelo usando una GPU Tesla K80 con un Batch-size de 256 y 100 Epochs. (a) Rendimiento del modelo en acierto. (b) Rendimiento del modelo en pérdida. 41

5.2. Resultados de entrenamiento del modelo usando una GPU Tesla K80 con un Batch-size de 256 y 175 Epochs. (a) Rendimiento del modelo en acierto. (b) Rendimiento del modelo en pérdida. 42
5.3. Resultados de entrenamiento del modelo usando una GPU Tesla K80 con un Batch-size de 256 y 200 Epochs. (a) Rendimiento del modelo en acierto. (b) Rendimiento del modelo en pérdida. 42
5.4. Comparativa de inferencias entre TensorFlow y OpenVINO en un entorno local. 49

A.1. Survey of programming languages at StackOverflow 2019. 62
A.2. Perceptron example. 63
A.3. Work plan. 64

Índice de tablas

5.1. Comparativa en el número de Epochs y tiempo de entrenamiento. 43
5.2. Acierto de inferencia en OpenVINO y TensorFlow. 44
5.3. Tiempo de inferencia en OpenVINO y TensorFlow. 44
5.4. Tiempo de inferencia en OpenVINO y TensorFlow con distinto hardware. . . 45
5.5. Comparativa de tiempo de inferencia en OpenVINO y TensorFlow con distinto framework web. 46
5.6. Comparativa de tiempo de inferencia en OpenVINO y TensorFlow con distinto hardware y servidor web. 47
5.7. Peticiones por segundo soportadas por el servidor para OpenVINO y TensorFlow. 48

Resumen

La observación remota de la Tierra ha sido siempre objeto de interés para el ser humano. A lo largo de los años, los métodos empleados con ese fin han ido evolucionando hasta que, en la actualidad, el análisis de imágenes multiespectrales constituye una línea de investigación muy activa, en especial para realizar la monitorización y el seguimiento de incendios, desastres naturales, vertidos químicos u otros tipos de contaminación ambiental.

Las imágenes satelitales en un mundo donde el machine learning y el procesamiento de datos ha avanzado tanto, abre la posibilidad de construir modelos capaces de reconocer, en tiempo real, zonas en las que ha ocurrido un desastre natural y poder actuar en consecuencia.

En este Trabajo de Fin de Grado se lleva a cabo la optimización en tiempos de inferencia de un modelo de machine learning usado para detectar desastres naturales con el kit de herramientas Intel OpenVINO. Además, se realiza la puesta en producción de la aplicación en un entorno cloud de Google, con el objetivo de que nuestro servicio soporte miles de peticiones por minuto.

Palabras clave

Imágenes multiespectrales, OpenVINO, Tensorflow, Docker, Google Cloud.

Abstract

Remote observation of the Earth has always been a point of interest to humans. Over the years, the methods used for this purpose have evolved until, at present, the analysis of multispectral images constitutes a very active line of research, in particular to carry out fire monitoring and follow-up, natural disasters, chemical spills or other types of environmental pollution.

Satellite imagery in a world where machine learning and data processing has advanced so far opens up the possibility of building real-time processing models witch's recognizes areas where a natural disaster has occurred, and being able to act accordingly.

This Final Degree Project carry out the optimization of a machine learning model used to detect natural disasters with the Intel OpenVINO toolkit. In addition, the application is deployed in a Google cloud environment, with the objective of support thousands requests per minute.

Keywords

Multispectral images, OpenVINO, TensorFlow, Docker, Google Cloud.

Capítulo 1

Introducción

1.1. Motivación y objetivos

El área del Deep Learning [1] ha avanzado exponencialmente en los últimos años. Esto ha permitido que a día de hoy se pueda contar con modelos predictivos capaces de procesar imágenes y clasificarlas según sus características primarias. La consecuencia principal de este proceso es la apertura de una ventana de oportunidad a la explotación de estos modelos en un entorno real, con el objetivo de que sean cruciales a la hora de detectar incendios, terremotos, así como todo tipo de desastres naturales.

El uso eficiente de estos modelos requiere una infraestructura capaz de soportar la fiabilidad necesaria en términos de robustez y velocidad. En estos casos, el procesamiento en tiempo real se vuelve algo indispensable para lograr optimizar recursos de emergencia, dirigir equipos a las zonas de desastre más afectadas y, en definitiva, prevenir los máximos riesgos posibles.

Los ejes que vertebran este proyecto se sitúan en torno a dos polos: primeramente, la aceleración del tiempo de entrenamiento de un modelo de Deep Learning usando una GPU [2] en el servicio de Google Colab, y en segundo lugar, la optimización del tiempo de inferencia del modelo mediante el kit de herramientas Intel OpenVINO. Finalmente, el modelo se desplegará en un entorno cloud en el que pueda funcionar como servicio capaz de soportar miles de eventos concurrentes. Para llevar a cabo el entrenamiento del modelo se ha

utilizado como herramienta principal TensorFlow [3], un framework open source desarrollado por Google para la preparación de algoritmos de entrenamiento de redes neuronales.

Con la necesidad de que la aplicación sea robusta y flexible ante cambios se ha usado la tecnología de contendores Docker[1]. Empleando esta herramienta se asegura que tanto las versiones del sistema operativo como de librerías externas sean compatibles entre sí, adicionalmente, se deja abierta la posibilidad de portar la aplicación a distintos entornos que aprovechen esta solución de contenedores. La consecución del objetivo general anteriormente mencionado se lleva a cabo en la presente memoria abordando una serie de objetivos específicos, los cuales se enumeran a continuación:

- Mejora en los tiempos de entrenamiento de un modelo de Deep Learning usando una GPU del servicio de Google Colab.
- Conversión de un modelo de TensorFlow a uno de OpenVINO para aumentar su velocidad de inferencia.
- Preparación de una arquitectura de Google cloud capaz de soportar tráfico concurrente en tiempos óptimos para el servicio.
- Codificación de una aplicación capaz de hacer uso de los distintos sistemas de inferencia de TensorFlow y OpenVINO.
- Codificación de una aplicación web apta para exponer todos los servicios en un entorno productivo.
- Encapsulación de los distintos entornos de producción haciendo uso de Docker.
- Despliegue de la aplicación y pruebas de carga.
- Obtención de resultados y realización de comparativas de rendimiento entre los distintos sistemas de inferencia, hardware y servidores web.

[1] https://www.docker.com/¸

1.2. Estado del arte

En la actualidad, la inteligencia artificial se compone de varias ramas tales como machine learning, natural language processing, entre otras. Una de ellas es el Deep Learning. Esta arquitectura de aprendizaje profundo persigue el estudio y clasificación de una variedad de problemas haciendo uso de sus propios algoritmos. Actualmente, los algoritmos de Deep Learning son usados para todo tipo de problemas que abarcan multitud de sectores dentro de la industria [4], los gobiernos y en definitiva, de la propia sociedad. La digitalización y expansión de internet provee de innumerables fuentes de datos capaces de ser procesadas y analizadas por este tipo de algoritmos, que son usadas para distintos fines.

El propio origen de los datos ha cambiado, ahora provienen de interacciones que tienen los usuarios con sus dispositivos móviles, llamadas, transacciones de dinero por internet [5], navegación de páginas web y, en el caso de este trabajo, imágenes de un satélite. El tratamiento de imágenes ha supuesto un avance en la sociedad del que ahora se aprovechan cuerpos de policía, usando estas herramientas para detección de matrículas o el reconocimiento de potenciales delincuentes[2]; médicos, que utilizan estos sistemas [6] para mejorar la detección prematura de algunos tipos de cáncer[3], o en el caso de la industria, que se ayuda de estas soluciones para automatizar y clasificar procesos que antes suponían la supervisión o ejecución de una persona. Del mismo modo, los países poseen sus sistemas personales de reconocimiento de imágenes para la clasificación de sus ciudadanos, sistemas de recomendación tanto para las empresas que buscan aumentar sus ventas como para bancos que buscan personas aptas para préstamos e incluso sirve como sesgo para evitar contenido indeseable en plataformas a través de la red.

En general, la cantidad masiva de datos ha creado una necesidad de explotación a través de los mismos, por lo que el Deep Learning se sitúa como una herramienta fiable para dar valor a todas las interacciones que están ocurriendo casi de manera permanente en cada

[2] https://www.nhtsa.gov/sites/nhtsa.dot.gov/files/documents/809811-lidarspeedmeasuringdevice.pdf

[3] https://www.nature.com/articles/srep46450

sistema tecnológico del planeta. Todo este estímulo lleva consigo la creación de miles de nuevos puestos de trabajo en el sector tecnológico, dedicados en exclusiva a la aplicación de algoritmos de aprendizaje automático, de igual manera que al aumento de su enseñanza. Esto ha abierto la posibilidad a profesionales que anteriormente no tenían una función claramente definida en este campo a situarse como prácticamente indispensables. Los beneficios son perceptibles en estudios como matemáticas, estadística y relacionados. En estos ámbitos, el perfil matemático y la capacidad de análisis son aptitudes muy valoradas para la realización de este tipo de tareas.

El desarrollo de este nuevo perfil de profesionales ha fomentado el empleo de lenguajes de programación menos usuales hasta el momento. Esto se debe a que tanto su uso como curva de aprendizaje es más asequible que el de otros lenguajes más tradicionales, caso de Java o C++ (ver Figura 1.1). Junto a ello, han surgido nuevas herramientas gratuitas y de código abierto [7], tales como Jupyter, TensorFlow, Scikit-learn o PyTorch, entre otras.

1.2.1. Concepto Deep Learning

El Deep Learning tiene como elemento definitorio el uso de algoritmos que basan su estructura en redes neuronales artificiales, imitando el comportamiento que tienen las del ser humano y su sistema nervioso central. La fuerza que ha proporcionado el surgimiento del Big Data ha conseguido que este tipo tecnologías se conviertan en la práctica diaria de muchos profesionales. Una de las claves de los algoritmos de Deep Learning está en la capacidad de aprendizaje que reside en ellos. Esto nos brinda la posibilidad de lidiar con problemas del mundo real, en el que las combinaciones de posibilidades y reconocimiento de patrones se quedan fuera de nuestros cálculos. Las redes neuronales representan la principal herramienta para clasificar imágenes. Esto es debido a que pueden extraer características fundamentales de cada píxel y poseen un alto porcentaje de acierto en la predicción.

Para poder materializar todos estos algoritmos de aprendizaje automático disponemos de servicios de grandes empresas como Google, Amazon e IBM, que implementan sus propias

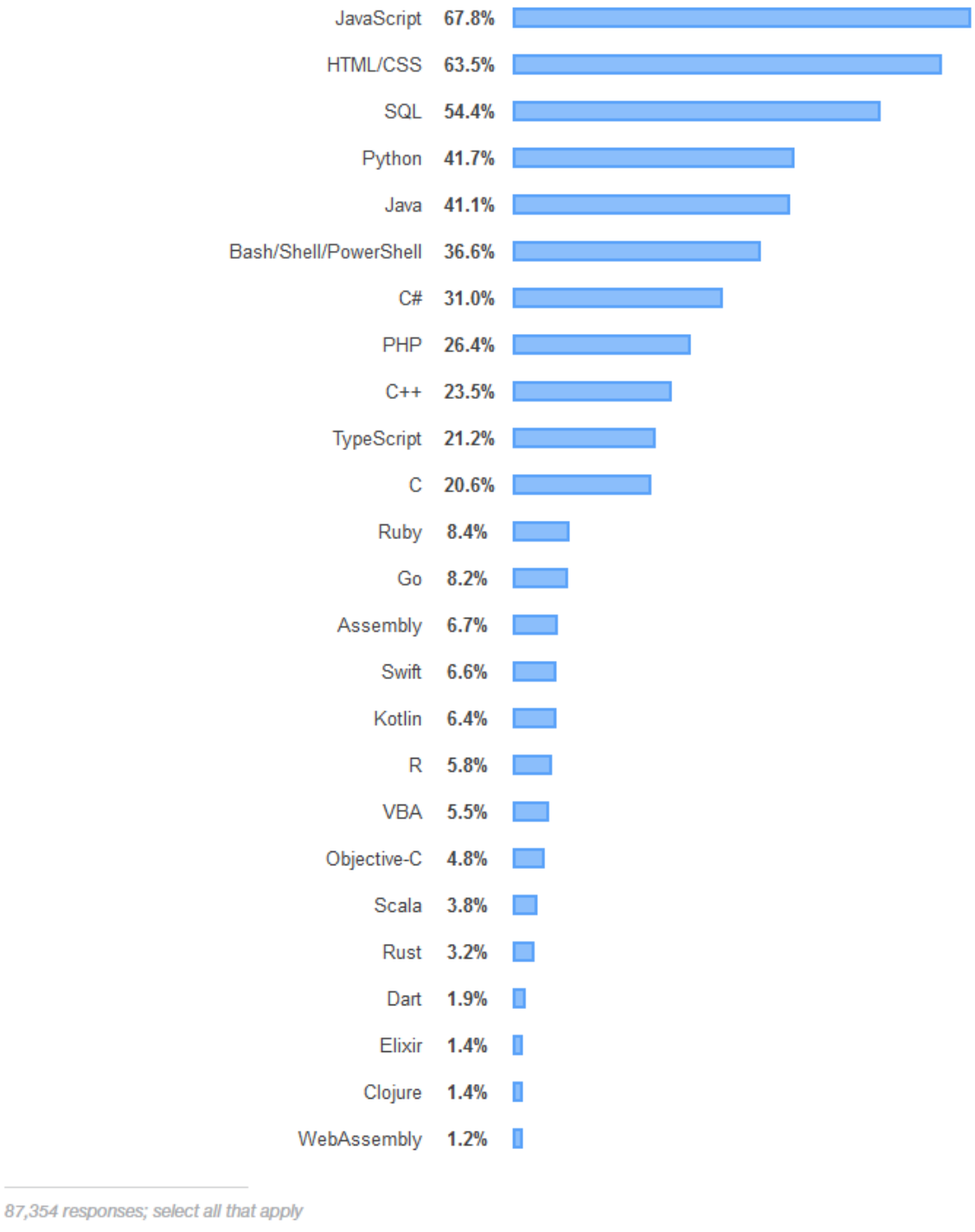


Figura 1.1: *Encuesta sobre lenguajes de programación usados en StackOverflow 2019.*

soluciones comerciales [8]. Pero también podemos optar por herramientas de código abierto como TensorFlow, una de las librerías más famosas de Deep Learning desarrollada por los ingenieros de Google, posteriormente liberada bajo licencia Apache. También disponemos de otras como PyTorch y Keras. Todas las mencionadas anteriormente fueron originalmente desarrolladas para el lenguaje de programación Python, el cual ha visto aumentado su porcentaje de uso debido a esta corriente de machine learning [9].

1.2.2. Redes neuronales en el tratamiento de imágenes

La unidad básica de procesamiento de las redes neuronales es el perceptrón (ver Figura 1.2), a partir del que se desarrolla un algoritmo capaz de generar criterios de selección de subconjuntos de neuronas. Este conjunto de neuronas pasará a formar parte de las distintas capas que componen por completo la red neuronal. Cada neurona recibe una entrada, ya sea de una fuente externa o de otra neurona.

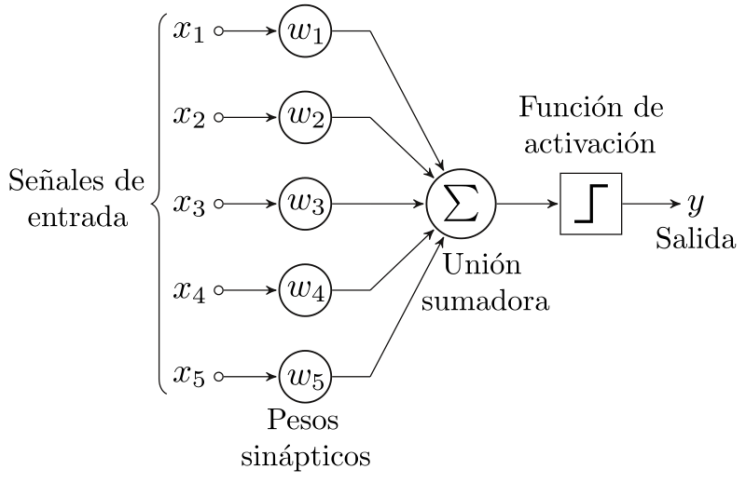


Figura 1.2: *Ejemplo de perceptrón.*

Cada neurona aplica una función de cálculo a partir de la cual se generan los pesos correspondientes de cada neurona. Estos pesos representan el nivel de interacción de las neuronas y deberán de ser ajustados de manera que se ciñan lo más posible a los datos que conocemos. Los pesos de entrada de una capa tienen origen en una capa anterior y sus

salidas forman parte de la entrada de una capa posterior. La propagación se produce hasta llegar a la última capa de la red, que será la capa de salida de la que obtengamos el resultado de nuestra clasificación.

En este problema concreto nos centramos en clasificar imágenes multiespectrales con alta resolución espacial [10] haciendo uso de las bandas espectrales RGB (Red, Green and Blue). Nuestro conjunto de imágenes pertenece a una zona parcialmente destruida por un desastre natural en Haití, ocurrido en el año 2010. Estas imágenes fueron adquiridas por el satélite de observación terrestre de alta resolución GeoEye-1, lanzado en septiembre de 2008. Por lo tanto, el fin de nuestro modelo de Deep Learning es tener la capacidad de clasificar dichas imágenes dependiendo si la zona está dañada o, por el contrario, está en buenas condiciones.

1.3. Plan de trabajo

En la Figura 1.3 podemos observar el plan de consecución de objetivos seguidos en este trabajo. La métrica de esfuerzo que aparece en ella hace referencia al número de jornadas invertidas en cada tarea, con una suma total de 10 semanas. Las actividades están ordenadas de manera secuencial en su ejecución y presentan una dependencia entre sí, por lo que nunca se ha pasado a otra tarea sin completar la anterior.

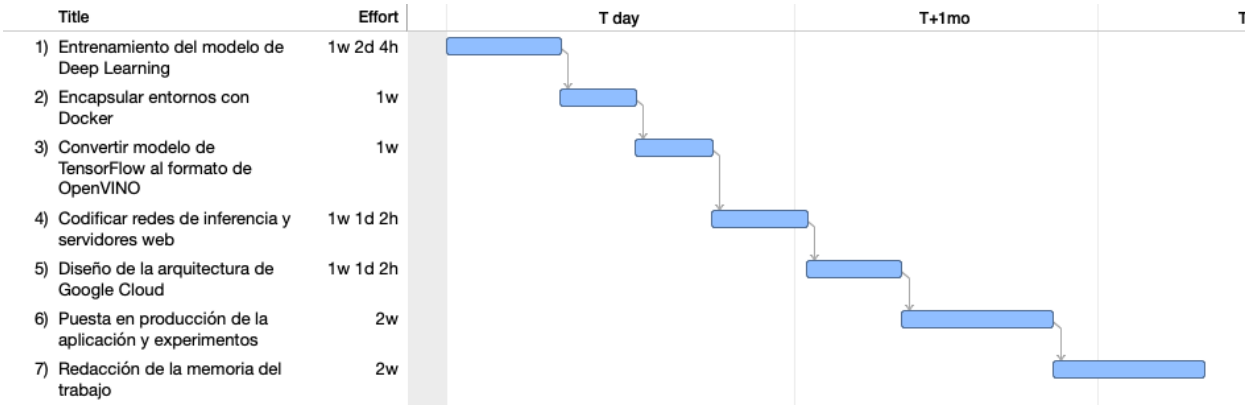


Figura 1.3: *Plan de trabajo.*

1.4. Organización de esta memoria

Teniendo presentes los anteriores objetivos concretos, se procede a describir la organización del resto de esta memoria, estructurada en una serie de capítulos cuyos contenidos se describen a continuación:

- **Entrenamiento del modelo mediante Google Colab**: Se define el proceso de entrenamiento y aumento de la velocidad usando la plataforma Google Colab y su hardware asociado.
- **Tecnología OpenVINO**: Se define el propósito del kit de herramientas de Intel OpenVINO así como la transformación de un modelo de TensorFlow para que sea compatible con dicha solución.
- **Arquitectura Cloud propuesta**: Se presenta la arquitectura de Google Cloud diseñada para soportar toda la infraestructura de la aplicación y se explica la puesta en producción del servicio.
- **Resultados experimentales**: Se preparan los distintos frameworks web que van a ser puestos a prueba haciendo uso del lenguaje de programación Python, mostrando el rendimiento obtenido en las fases de entrenamiento y de inferencias. Además, se presentará el cálculo aproximado de los costes del proyecto.
- **Conclusiones y trabajo futuro**: Se presentan las conclusiones obtenidas mediante las pruebas de carga y también algunas posibles líneas de trabajo futuro que se pueden desempeñar en relación al presente trabajo.

Capítulo 2

Entrenamiento del modelo mediante Google Colab

Dentro del entrenamiento de modelos Deep Learning [11] la velocidad es uno de los parámetros fundamentales. Los modelos pueden requerir entradas de tamaño masivo en las que la capacidad de cómputo se torne clave para acelerar el proceso; esto permite enfocarse plenamente en la mejora de rendimiento del modelo planteado. El objetivo es evitar la posible espera que pueda producir volver a entrenar el modelo con distintos parámetros. Con ello, se puede reajustar constantemente para encontrar el punto óptimo de manera ágil.

En este trabajo se va a entrenar un modelo de Deep Learning haciendo uso del framework de código abierto de TensorFlow[1], que está programado en el lenguaje de programación Python. Éste incluye una API (Interfaz de Programación de Aplicaciones) de Deep Learning llamada Keras, que será la que utilicemos.

El tipo de operaciones que requiere nuestra aplicación en la parte del tratamiento de imágenes, así como los procedimientos que realizan las redes neuronales [12] para hacer sus cálculos son, en muchas ocasiones, operaciones matriciales. Nuestro objetivo será aprovechar al máximo el rendimiento que una GPU (Unidad de Procesamiento Gráfico) puede aportar en este tipo de operaciones, principalmente por su arquitectura de paralelización, idónea para este tipo de trabajo. La ventaja que aporta frente a la CPU (Unidad de Procesamiento Central) es la capacidad de cómputo con un mayor número de núcleos o cores (ver

[1] https://www.tensorflow.org/

Figura 2.1), gracias a su conectividad por PCI express y el ancho de banda que proporciona.

Cabe mencionar que Google Colab, al ser una plataforma gratuita, no asegura la disponibilidad de sus componentes. Esto conlleva que tanto el hardware requerido como su memoria disponible pueda variar según la demanda del sistema. En este trabajo se han mantenido los mismos componentes hardware en Google Colab con la máxima memoria disponible.

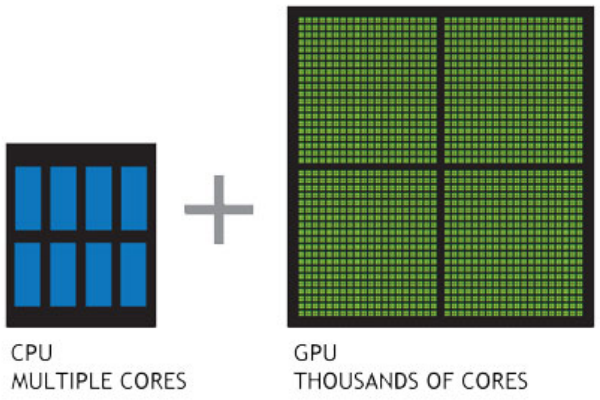


Figura 2.1: *Diferencia de núcleos para procesar cargas de trabajo en paralelo de forma eficiente entre CPU y GPU.*

2.1. Modelo propuesto

En esta parte del trabajo se pretende conseguir la máxima velocidad de entrenamiento posible manteniendo unos niveles de acierto elevados en la predicción.

Nuestro modelo [13] tiene como cometido primordial poder clasificar distintas imágenes según el estado del terreno que aparece en la fotografía, siendo las opciones: terreno dañado y terreno en buenas condiciones. Para ello, disponemos de un dataset de 268 imágenes multiespectrales, adquirido por el satélite de observación terrestre de alta resolución GeoEye-1 durante el terremoto ocurrido en Haití en 2010. En este tipo de imágenes se capturan datos dentro de rangos de longitud de onda específicos a través del espectro electromagnético visible. En la Figura 2.2 podemos observer un ejemplo del tipo de imágenes que se usarán en este trabajo.

Como framework principal para realizar el entrenamiento nos ayudaremos de TensorFlow, que incluye la librería de Deep Learning Keras. Esta librería simplifica mucho la implementación de este tipo de algoritmos de aprendizaje automático, debido a que sus objetos y funciones están programados de una manera intuitiva.

La distribución a efectuar sobre el conjunto de datos en entrenamiento y test es del 70 % y 30 % respectivamente.

(a) (b)

Figura 2.2: *Ejemplo de imagen capturada por el satélite GeoEye-1. (a) Imagen de un terreno destruido. (b) Imagen de un terreno en buenas condiciones.*

Para la construcción de este modelo haremos uso de las siguientes capas, sobre las que TensorFlow nos da una API para tener control total sobre su configuración. Son descritas a continuación:

- **Conv2D**: capa convolucional cuyo principal objetivo es extraer características de la imagen de entrada o/y sus partes. El término 2D se refiere al movimiento del filtro, un parámetro de entrada de este tipo de capas. El filtro atraviesa la imagen en dos dimensiones. Tiene como parámetros de entrada una imagen en tres dimensiones y el número de filtros que vamos a aplicar sobre la imagen.

  Aplicaremos sobre esta capa una configuración de 64 filtros y un tamaño de kernel de 3×3 ya que nuestras imágenes son de 128×128 píxeles.

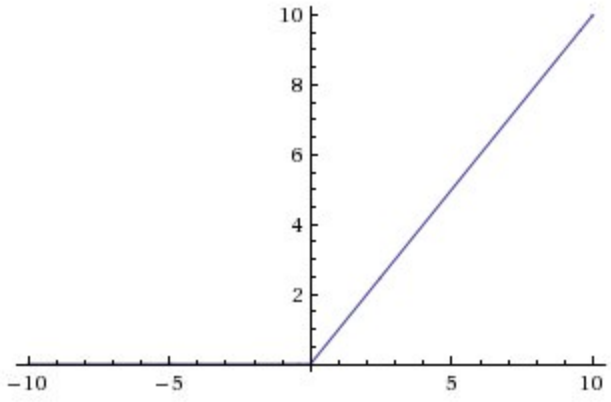


Figura 2.3: *Función de activación Relu.*

- **Activación Relu (Recitified Linear Unit)**: en redes neuronales, una función de activación es la responsable de transformar la entrada. Sus principales funciones son detectar posibles correlaciones entre dos variables distintas y ayudar al modelo a tener en cuenta funciones no lineales. Ello significa que la red neuronal es capaz de realizar microajustes para capturar relaciones entre entradas y salidas que no sigan una línea recta en el plano cartesiano.

  Como podemos observar en la Figura 2.3, la función de activación Relu se comporta devolviendo un 0 para valores de entrada negativos y, en caso contrario, devolviendo el propio valor de entrada. Esta función de activación conserva los valores que contienen algún patrón en la imagen y los transfiere a la siguiente capa, mientras que los pesos negativos no son importantes y son establecidos con el valor 0. Otras funciones de activación como la función Sigmoide o Tanh modifican todos los valores de entrada, mientras que la función Relu mantendrá los valores de peso positivo para las capas posteriores.

- **MaxPooling2D**: es una capa que sigue un proceso de discretización basado en muestras, y su objetivo es reducir la muestra de una representación de entrada mediante el acortamiento de sus dimensiones. En nuestro modelo aplicaremos una reducción a matrices de 2×2.

- **Dropout**: El cometido de esta capa es ignorar ciertas neuronas de forma aleatoria para no incluirlas en el entrenamiento. Las neuronas restantes serán las encargadas de representar las predicciones de la red. De esta manera también reducimos la complejidad de nuestra red y la posibilidad de sobreentrenamiento.

- **Flatten**: capa de aplanamiento usada para reducir a uno el número de dimensiones de nuestra matriz de entrada.

- **Dense**: una de las capas más utilizadas en la API de Keras, es la manera de efectuar multiplicaciones matriciales.

- **Optimizador Adam**: es un algoritmo de optimización diseñado especialmente para redes neuronales. Aprovecha el poder de los métodos de tasas de aprendizaje adaptativo para encontrar nivel de aprendizaje individuales para cada parámetro. Este optimizador posee un hiperparámetro llamado *learning rate* que regula la rapidez con la que el modelo avanza hacia el valor óptimo de sus pesos. Un *learning rate* bajo implicaría que los pesos evolucionan lentamente durante el entrenamiento, con lo cual puede tardar mucho en llegar al valor óptimo, mientras que con un learning rate alto avanzamos más rápido pero podemos sobrepasar este punto óptimo. El valor de este hiperparámetro en este trabajo se mantiene a 0.0008.

Para un mejor entendimiento, en la Figura 2.4 podemos observar el conjunto de capas utilizado para crear la topología de la red de nuestro modelo. Además, algunas optimizaciones a nivel de hardware han sido realizadas para acelerar el proceso de forma general. Estas modificaciones son las siguientes:

- **Uso de variables de 16 bits en vez de 32 bits**: una de las posibilidades que nos brinda el uso de una GPU es reducir a la mitad el uso en memoria de las variables del proceso. Usaremos esto siempre y cuando no afecte a la calidad de la predicción.

- **Uso del compilador XLA**: el compilador XLA[2] (Accelerated Linear Algebra) op-

[2] https://www.tensorflow.org/xla

timiza el grafo de nuestro modelo de manera específica haciendo uso de la GPU. Normalmente cuando se ejecuta un programa de TensorFlow cada operación tiene una implementación de kernel de GPU previamente compilada a la que el ejecutor envía datos. Con el compilador XLA conseguimos fusionar en una sola ejecución todas estas operaciones, consiguiendo así reducir el ancho de banda usado en memoria.

- **Valores altos del parámetro de entrenamiento Batch Size**: este parámetro define el número de muestras con las que se va a ir entrenando la red hasta completar el número de registros totales. Gracias a la capacidad de cómputo de nuestra GPU podemos permitirnos el uso de valores altos en este parámetro de entrenamiento. El uso de valores pequeños puede reducir el uso de memoria y la velocidad de entrenamiento. En nuestra red el número total de registros es tan pequeño que nos permitiremos usar un valor cercano al total de imágenes que disponemos.

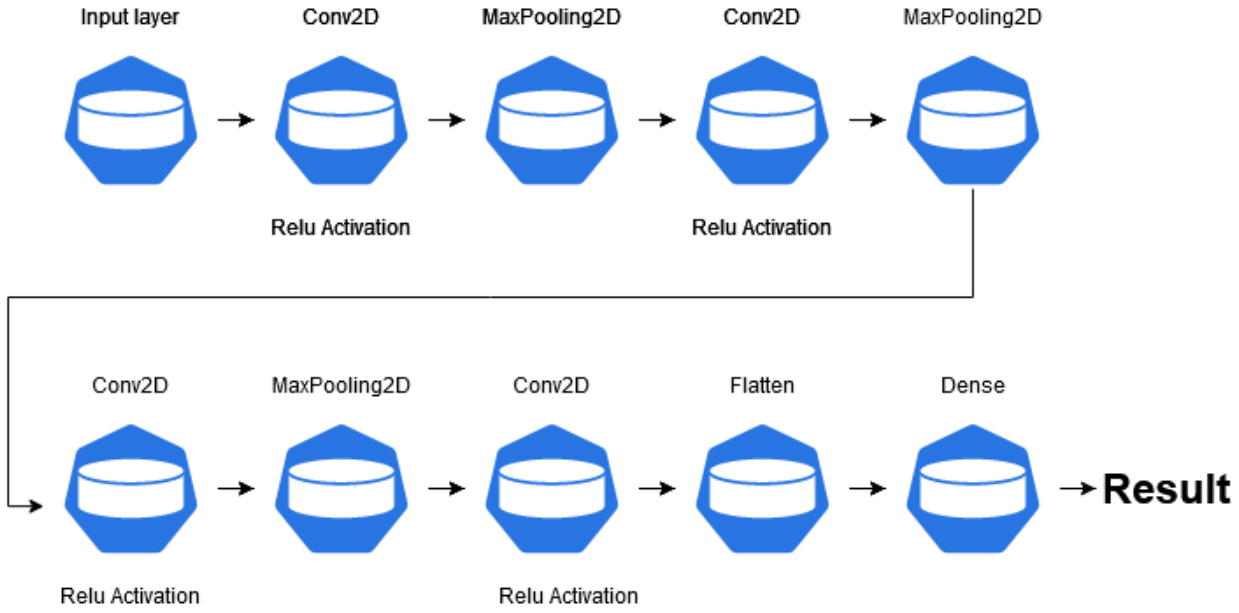


Figura 2.4: *Topología de la red del modelo Deep Learning usado para la clasificación de daños.*

2.2. Entorno Google Colab

La plataforma de Google Colab[3] es un servicio gratuito de Google, con el que podemos ejecutar e instalar librerías del lenguaje de programación Python [14].

Una de las grandes ventajas de trabajar con este entorno es que no necesitamos configuración ninguna, se ejecuta de forma íntegra en el navegador sin necesidad de instalar nada previamente. Esto supone que toda la carga computacional reside en la herramienta de Google, lo que permite trabajar de manera fluida realizando otro tipo de actividades en nuestra máquina, o simplemente ejecutar un proceso para el que no tenemos suficiente potencia disponible.

Estas características permiten a Google Colab convertirse en un entorno muy válido para personas que están dando sus primeros pasos en este área de la inteligencia artificial, pero haciendo uso de unas herramientas profesionales. En este trabajo haremos uso de la GPU Tesla K80[4]. Las características principales de nuestra principal unidad de cómputo son las siguientes:

- 4992 núcleos de NVIDIA CUDA con diseño de dos GPU.
- Hasta 2.91 teraflops de rendimiento en operaciones de precisión doble con NVIDIA GPU Boost.
- 24 GB de memoria GDDR5.
- 480 GB/s de ancho de banda de memoria agregado.
- Hasta 8.73 teraflops de rendimiento en operaciones de precisión simple con NVIDIA GPU Boost.

El uso de este tipo de herramientas en esta plataforma es extrapolable a otras nubes sin las restricciones en cuanto al número de unidades de procesamiento que necesitamos, la

[3] https://colab.research.Google.com/notebooks/intro.ipynb
[4] https://www.nvidia.com/es-es/data-center/tesla-k80/

interoperabilidad de sus elementos con otros componentes externos, tales como servidores o respositorios de código, así como la configuración explícita de cada uno de los entornos de ejecución.

Capítulo 3

Tecnología OpenVINO

OpenVINO es un conjunto de herramientas multiplataforma desarrolladas por Intel, que facilita la transición entre los entornos de entrenamiento y producción de nuestro modelo de aprendizaje profundo. A pesar de estar desarrollada por una empresa comercial como Intel, pertenece al conjunto de aplicaciones de código abierto, de modo que se puede visualizar su código fuente, reportar fallos e incluso realizar aportaciones.

El cometido principal de esta aplicación es la optimización del tiempo de inferencia de un modelo de Deep Learning previamente entrenado. Para ello, OpenVINO dispone de su propio formato de definición de modelos. Estos archivos son los que procesa su propia red de inferencia multiplataforma, ya que se encuentra preparada para poder trabajar de manera concurrente, aprovechando así toda la potencia de los procesadores o GPU actuales [2].

En la siguiente Figura 3.1 se puede observar el flujo de trabajo que se ha seguido en este trabajo, haciendo uso de la herramienta OpenVINO. En primer lugar, optimizaremos la topología de nuestro modelo y obtendremos una nueva representación del mismo, pero esta vez, con el formato necesario para ser procesado por la red de inferencia de alto rendimiento de OpenVINO. Finalmente, nuestra red de inferencia será la que realice el trabajo de clasificación en el entorno de producción de nuestra aplicación.

Podemos visualizar el código de la aplicación en su repositorio oficial de GitHub[1].

[1] https://github.com/openvinotoolkit/openvino

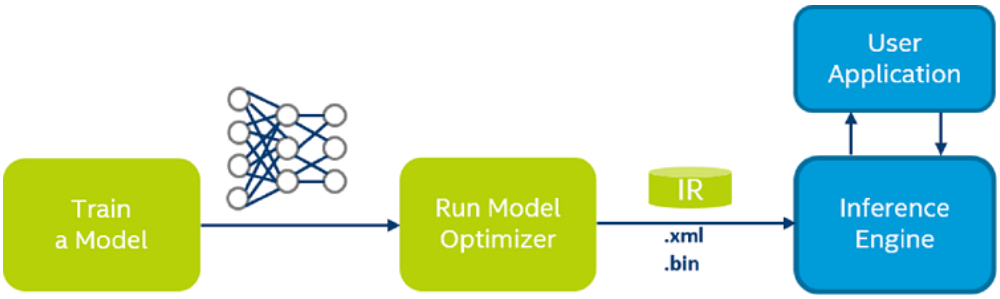


Figura 3.1: *Arquitectura de optimización de modelos con OpenVINO.*

3.1. Herramientas que lo componen

Esta tecnología desarrollada por Intel tiene por objetivo principal la optimización de modelos de redes neuronales convolucionales para potenciar su velocidad de inferencia. OpenVINO es capaz de soportar distintos hardwares (FPGA, Intel Movidius, procesamiento por GPU) y también varios sistemas operativos (Mac Os, Linux o Windows).

Las características principales de esta aplicación se resumen en dos puntos:

- **Optimizador de modelos de Deep Learning**: Aplicación de interfaz de línea de comando, la cual usa como base modelos de frameworks populares como Caffe, TensorFlow, MXNet, Kaldi y ONNX para convertirlos a un modelo optimizado de OpenVINO.

- **Interfaz de inferencia de modelos de Deep Learning**: API de alto rendimiento multiplataforma para realizar la inferencia de manera rápida y eficiente sobre dispositivos de Intel.

3.1.1. Optimizador de modelos de Deep Learning

Para poder realizar la optimización de nuestro modelo de Deep Learning previamente entrenado, se necesita el binario que contiene la topología de la red del modelo. Una vez el optimizador de OpenVINO procesa nuestro modelo procede a realizar una conversión de cada capa interna de la red a una nueva capa. Esta nueva capa, ya convertida al formato de

OpenVINO, conserva los pesos de la red anterior, sin embargo, está preparada para que la aplicación de inferencia de OpenVINO pueda leerla correctamente.

Esta herramienta proporciona de manera genérica distintos scripts para realizar esta conversión. Se incluyen diferentes ficheros de código fuente para los frameworks de Deep Learning más actuales, codificados en Python y totalmente modificables, aunque en principio no es necesario porque ya vienen preparados para funcionar.

3.1.2. Interfaz de inferencia de modelos de Deep Learning

Con nuestro modelo y su topología convertida a un formato válido de OpenVINO, ya tenemos todo lo necesario para poder realizar clasificaciones con su interfaz de inferencia. La optimización de inferencia se produce en este punto, donde cada capa de nuestro modelo original es procesada por la aplicación en un lenguaje de bajo nivel, preparado para realizar operaciones vectoriales bajo total control del programador. El lenguaje empleado para la codificación de la aplicación es C++ pese a que el programa pueda ser utilizado también en Python. Esto se debe al uso de su API, que traduce las peticiones realizadas en Python al core de la interfaz, que es C++ puro.

Todas las capas usadas en este proyecto son compatibles de manera directa con las predefinidas por la interfaz de inferencia, incluidas en la versión de OpenVINO 2020.1.023. OpenVINO nos da la posibilidad de añadir capas personalizadas, con el inconveniente de que deben ser programadas por el usuario de manera explícita en C++ para hacerlas compatibles con el resto de la aplicación, y, por supuesto, mantener estas nuevas capas a lo largo de las distintas actualizaciones y posibles cambios que pueda sufrir la aplicación.

3.2. Conversión del modelo a la plataforma OpenVINO

Para realizar la conversión, en primer lugar es necesaria la exportación del original de TensorFlow a un formato compatible con la red de optimización de modelos de OpenVINO.

La serialización por defecto de un modelo de TensorFlow puede incluir de manera inde-

pendiente:

- Un punto de control TensorFlow que contiene los pesos del modelo.
- Un prototipo 'SavedModel' que contiene la topología de la red del modelo de TensorFlow.

Los métodos exactos para la serialización del modelo varían según la versión de TensorFlow, en este caso 1.15.2. La metodología de conversión exacta utilizada para este proyecto se puede encontrar en el repositorio de código fuente en GitHub[2]. Este modelo de OpenVINO va a servir tanto de punto de partida para su optimización como para su uso directo en el servicio personalizado de TensorFlow, con el objetivo de desplegar modelos en producción.

Una vez exportado el modelo de Deep Learning al formato estándar de TensorFlow, tendremos a nuestra disposición los ficheros necesarios para proceder a su transformación al formato de OpenVINO. Para realizar esta operación se hace uso de la herramienta de optimización de modelos, en concreto, con el script específico de TensorFlow, cuyo nombre es `mo_tf.py`. Este código fuente es ejecutado en la línea de comandos del sistema operativo correspondiente con los siguientes parámetros:

Listing 3.1: *Comando de terminal para convertir un modelo TensorFlow a uno de OpenVINO.*

```
1 mo_tf.py --input_model model.pb --input_model_is_text -b 1
```

El comando usado epecifica con el flag `-input_model_is_text` que nuestro fichero no está codificado en binario, por lo que es texto plano. Esta opción es totalmente configurable y depende del proceso de exportación. Se ha encontrado útil la opción de exportación a texto plano ya que de esta manera se puede observar la arquitectura de la red y los pesos pertenecientes a cada capa.

Configuramos también el *flag* -b, esta opción determina el valor de reemplazo cuando se reciben valores negativos. Seleccionamos como valor uno porque en las entradas de nues-

[2] https://github.com/A-Ortiz-L/multispectral-imaging-cnn-final-degree-work/blob/master/src/entity/keras_model.py

tra red neuronal pueden propagarse valores negativos, los cuales no son válidos para su procesamiento en OpenVINO.

3.3. Inferencias. TensorFlow vs OpenVINO

Como se ha mencionado anteriormente, Tensorflow posee su propio sistema de inferencia preparado para su uso productivo en un entorno real. Este sistema se llama TensorFlow serving, que incorpora un servidor codificado mediante el patrón diseño API REST[3], de modo que las peticiones de inferencia se realizan al servidor por medio del protocolo de transmisión de datos HTTP. La aplicación de TensorFlow está diseñada para el escalado tanto en el número de modelos para los que puede recibir inferencias y sus versiones, como para la escalabilidad en capacidad de cálculo. El escalado de cálculo está preparado para funcionar en una arquitectura clúster, en este caso de contenedores como es Kubernetes[4], por lo que el servidor puede ser encapsulado en su totalidad en un contenedor de Docker. Para este proyecto se ha trabajado con una versión encapsulada de Docker, pero no con la extensibilidad del escalado con Kubernetes [15]. La unidad de cálculo principal del sistema de inferencia también se puede configurar, lo que permite el uso tanto de CPU como de GPU. Por esa razón, en caso de que se decida emplear esta tecnología junto con una GPU, es necesario configurar de manera explicita el entorno.

En la Figura 3.2 podemos observar el diseño de la arquitectura de la aplicación. Una de las ventajas frente a usar el sistema de inferencias clásico de TensorFlow es que la aplicación está preparada y optimizada para recibir tanto peticiones en streaming como en batch. Adicionalmente, reducimos el tamaño de las librerías a instalar al mínimo necesario para realizar las inferencias y configurar el servidor. Esta solución es más ligera y portable, evitando así instalar todo el sistema de construcción de algoritmos y demás artefactos que incorpora la librería de TensorFlow para programar redes neuronales.

La inicialización del servidor de inferencia y los métodos necesarios para realizar la

[3] https://stackoverflow.blog/2020/03/02/best-practices-for-rest-api-design/
[4] https://kubernetes.io/es/docs/concepts/overview/what-is-kubernetes/

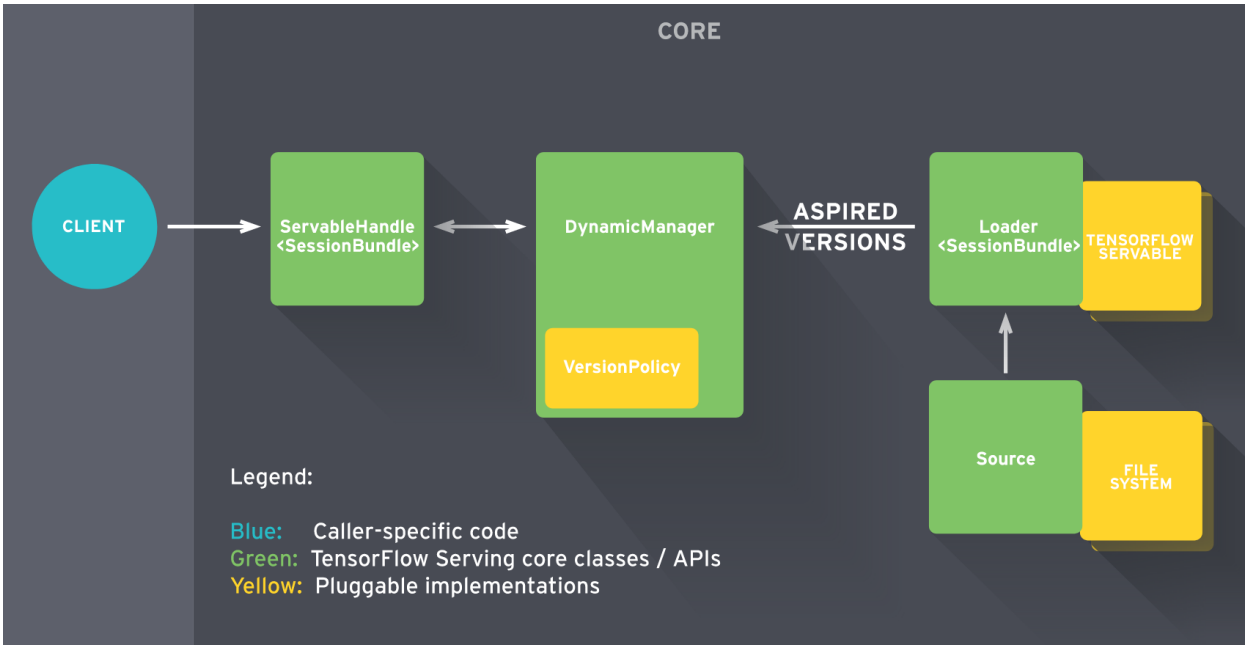


Figura 3.2: *Arquitectura de TensorFlow serving.*

petición aparecen en el Código 3.2. En su inicialización arrancamos el servidor REST, que funcionará de forma transparente para el usuario de nuestra aplicación, ya que se ejecuta en la red local de nuestro contenedor Docker.

Listing 3.2: *Código Python para la red de inferencia de TensorFlow.*

```
class TensorflowNetwork:
    def __init__(self):
        self.model_uri = 'http://localhost:8501/v1/models/model:predict'
        self.init_tensorflow_serve()

    @staticmethod
    def shape_image(file_route):
        img_array = cv2.imread(file_route, cv2.IMREAD_GRAYSCALE)
        new_array = cv2.resize(img_array, (128, 128))
        img = new_array.reshape(-1, 128, 128, 1) / 255.0
        img = np.float32(img).tolist()
        return img

    def process_image(self, file_route) -> Tuple[bool, float]:
        image = self.shape_image(file_route)
        start = time.time()
        predict = self.network_request(image)
        return predict, (time.time() - start)
```

```
19
20      def network_request(self, image) -> bool:
21          headers = {"content-type": "application/json"}
22          data = json.dumps({"signature_name": "serving_default", "instances": image})
23          res = requests.post(self.model_uri, data=data,
24                              headers=headers)
25          predictions = json.loads(res.text)['predictions']
26          predict = True if predictions[0][0] >= 0.5 else False
27          return predict
28
29      @staticmethod
30      def init_tensorflow_serve():
31          os.system('tensorflow_model_server '
32                    '--rest_api_port=8501 --model_name=model '
33                    '--model_base_path=/app/model &')
```

Por otro lado, OpenVINO también incorpora en su conjunto de herramientas un sistema de inferencia optimizado. Al igual que el sistema de inferencia de TensorFlow, OpenVINO también puede configurar tanto un procesador como una tarjeta gráfica para realizar sus cálculos. La aplicación de inferencia de OpenVINO usada en este trabajo se codifica de manera íntegra haciendo uso del lenguaje de programación Python. Podemos observar esta implementación en el Código 3.3.

Listing 3.3: *Código Python para la red de inferencia de OpenVINO.*

```
1       class OpenVinoNetwork:
2       def __init__(self):
3           self.plugin = IEPlugin(device='CPU')
4           self.net = IENetwork(model=f'{pickle_dir}model.xml',
5                                weights=f'{pickle_dir}model.bin')
6           self.exec_net = self.plugin.load(network=self.net)
7
8           self.input_blob = next(iter(self.net.inputs))
9           self.out_blob = next(iter(self.net.outputs))
10          self.net.batch_size = 1
11          self.image_shape = 128
12
13      def process_image(self, image_path) -> Tuple[bool, float]:
14          image = self.shape_image(image_path)
15          start = time.time()
16          res = self.network_request(image)
17          return res, time.time() - start
18
```

```
19      def shape_image(self, file_route):
20          image = cv2.imread(file_route, cv2.IMREAD_GRAYSCALE)
21          image = cv2.resize(image, (self.image_shape, self.image_shape))
22          image = image.reshape(self.image_shape, self.image_shape) / 255.0
23          return image
24
25      def network_request(self, image) -> bool:
26          res = self.exec_net.infer(inputs={self.input_blob: image})
27          res = res[self.out_blob]
28          res = False if res < 0.5 else True
29          return res
```

En su inicialización se lee el fichero ya optimizado del modelo de Deep Learning, también se inicia la clase perteneciente a la API de inferencia de OpenVINO que se encarga de realizar las inferencias. Se configuran métodos específicos para transformar la imagen a las dimensiones correspondientes y para hacer la petición a la red. La potencia de este modelo reside en la optimización que realiza la red en un lenguaje de bajo nivel, centrándose así en reducir los tiempos de inferencia.

Ambas soluciones implementan tecnologías de contenedores mantenidos de manera oficial por los fabricantes[5,6], por lo que el uso de Docker es la mejor opción para transportar nuestras redes de inferencia a cualquier sistema o dispositivo hardware. Como punto distintivo, OpenVINO implementa soluciones de fábrica para FPGA e Intel Movidius. Las opciones de portabilidad son más amplias en esta tecnología, aunque no se descarta el uso de TensorFlow en estas plataformas, ya que su código fuente es accesible para todo el mundo y puede ser modificado. De manera adicional y como paso natural, las dos herramientas se han preparado para ser incluidas en un clúster de contenedores[7,8].

[5] https://hub.docker.com/u/openvino
[6] https://hub.docker.com/r/tensorflow/tensorflow/
[7] https://github.com/openvinotoolkit/model_server
[8] https://www.tensorflow.org/tfx/serving/serving_kubernetes

Capítulo 4

Arquitectura Cloud propuesta

La implementación y puesta en marcha de la aplicación se materializa en un entorno cloud, concretamente Google Cloud Platform.

Los servicios que proporciona son de especial utilidad debido, principalmente, a que los mismos son mantenidos y actualizados por la propia plataforma. Ello tiene como consecuencia que no haya que dedicar un gran esfuerzo en su configuración, más allá del primer uso o cuando se quiera hacer cualquier modificación en un parámetro determinado.

Entre las numerosas ventajas que otorga, también se puede citar su capacidad de escalar nuestras aplicaciones casi de manera infinita, según la demanda de la solución que hemos desarrollado.

A diferencia del resto de herramientas mencionadas hasta el momento, los servicios de Google sí son de pago. En muchos casos existe una modalidad gratuita; modalidad cuyos recursos en varias ocasiones quedan limitados en su capacidad de trabajo o en el tiempo que se pueden usar.

En la Figura 4.1 podemos observar el diseño de la arquitectura cloud a la que dotaremos de identidad con las soluciones que ofrece Google.

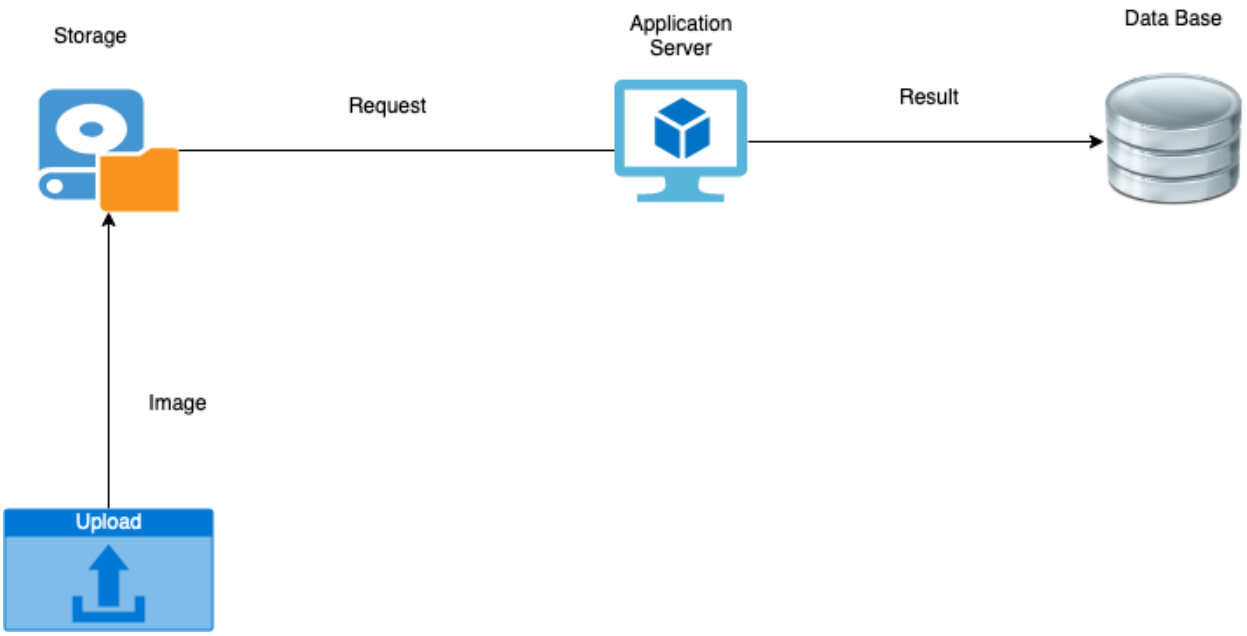


Figura 4.1: *Arquitectura Cloud.*

4.1. Descripción del entorno y sus diferentes componentes

Google Cloud posee multitud de herramientas específicas para cada situación. La mayoría cuenta con un gran nivel de interoperabilidad con el resto de herramientas.

La configuración de las herramientas es posible tanto por línea de comandos, gracias a su SDK[1], por las múltiples APIs para los distintos lenguajes de programación, así como por la interfaz gráfica de la web.

4.1.1. Google Storage

Es el sistema de almacenamiento de Google [16]. Su objetivo dentro de la aplicación es poder ser utilizado como sistema de copia de seguridad de todas las imágenes que se vayan procesando, además de disparador para procesar dicha imagen en el pipeline del proceso principal y poder clasificarla. Esto se debe a que el servicio incorpora eventos automáticos cada vez que un archivo se ha descargado, subido o modificado.

[1] https://www.google.com/search?client=firefox-b-d&q=SDK+google

Esta solución es la principal entrada de datos de la aplicación, por lo que nadie sin una clave de autenticación específica del proyecto puede subir archivos.

Las características principales que se han encontrado en el uso de esta aplicación para el trabajo son:

- Escalabilidad prácticamente infinita en el volumen de almacenamiento de los archivos.
- Posibilidad de configurar distintas o múltiples ubicaciones para almacenar los datos, de modo que se pueden tener réplicas del historial en distintas partes del mundo de manera simultánea. La replicación de los nodos a través de las distintas ubicaciones y su consistencia es una ventaja totalmente gestionada por Google.
- Opción de carga en paralelo, la cual sido utilizada para las pruebas de la aplicación.
- Encriptación de los datos y restricción de los accesos a los archivos de forma individual o colectiva.
- Interoperabilidad con el resto de servicios cloud.

4.1.2. Google BigQuery

Es una base de datos columnar y distribuida mantenida por Google, de modo que no hay que configurar su funcionamiento interno.

En la aplicación de este trabajo, BigQuery funciona como herramienta de análisis y exploración de los resultados obtenidos en las distintas pruebas de carga. Todos los dataset de esta base de datos han sido configurados en Europa para disminuir la latencia lo máximo posible. Las razones principales de su uso son:

- Capacidad de análisis del orden de Petabytes en cuestión de segundos debido a los múltiples nodos que ejecutan las cargas de trabajo de manera distribuida.
- Posibilidad de almacenar los distintos conjuntos de datos en ubicaciones distintas, reduciendo así la latencia dependiendo del sitio donde se ejecuten los trabajos.

- Soporte para la ingesta de datos en tiempo real.
- Acceso a distintas API en varios lenguajes de programación.
- Uso de SQL [17] estándar como lenguaje de consulta.

4.1.3. Google Pub/Sub

Es un sistema de colas de mensajería diseñado para eventos,basado en el patrón de diseño productor/consumidor.

En el caso de esta arquitectura cloud, servirá de hilo conductor para las distintas partes de la aplicación cada vez que se produzca un evento como el de una nueva carga de imagen o la petición al servidor para realizar la clasificación. Algunas de sus ventajas son las siguientes:

- Configuración de distintas colas de mensajes, separando así de manera lógica los eventos de la aplicación.
- Consumo de eventos en tiempo real, con la mínima latencia posible.
- Ingesta asíncrona de los mensajes.
- Encriptación del contenido de los mensajes.

4.1.4. Google Compute Engine

Es el servicio principal de Google para proporcionar máquinas virtuales totalmente configurables, tanto su capacidad de cálculo seleccionando el tipo de procesador, GPU y memoria RAM que se necesite, como el entorno en el que se va a ejecutar la aplicación, ya sea Docker [18] o las distintas distribuciones de sistemas operativos.

Este servidor usará como aplicación principal una imagen de Docker almacenada en el registro de contenedores. El sistema operativo que soporta el servicio de Docker es la distribución de Linux Debian 9. Carece de interfaz gráfica, por lo que todas las interacciones se harán mediante conexión SSH.

Google Compute Engine será el principal ejecutor de la aplicación, procesando y clasificando las nuevas imágenes que se carguen en el sistema de almacenamiento y volcando el resultado a la base de datos.

4.1.5. Google Cloud Function

Este sistema nos permite ejecutar una pequeña porción de código en el mínimo tiempo posible en una máquina virtual activada por eventos.

Estas máquinas escalan bajo demanda y no tienen que encenderse o apagarse cada vez que reciben una petición, por lo que siempre están disponibles y no hay apenas latencia.

En esta aplicación servirán de puente entre el volcado de una imagen al sistema de almacenamiento y el procesamiento de la petición al servidor con la información y metadatos correspondientes, con el objetivo de realizar la clasificación de la imagen.

4.1.6. Container Registry

Dado que todo el desarrollo de la aplicación y su versionado ha sido efectuado mediante Docker, se ha usado un repositorio de imágenes para su almacenamiento y tagueado. En este repositorio se almacenan las distintas imágenes de Docker tanto para OpenVINO como para TensorFlow, ya que se han separado para optimizar el tamaño de la imagen origen que tiene cada una de las aplicaciones. Además, así se pueden evitar posibles colisiones entre las dependencias de cada sistema de inferencia.

4.2. Explicación del flujo de datos

En la siguiente Figura 4.2 podemos observar una imagen completa de la arquitectura mencionada anteriormente, así como el flujo de trabajo que seguiría una imagen desde que es volcada en el sistema de almacenamiento hasta que su clasificación es almacenada en la base de datos para su respectivo análisis.

Cuando una imagen es cargada en el sistema de almacenamiento 4.1.1 dispara auto-

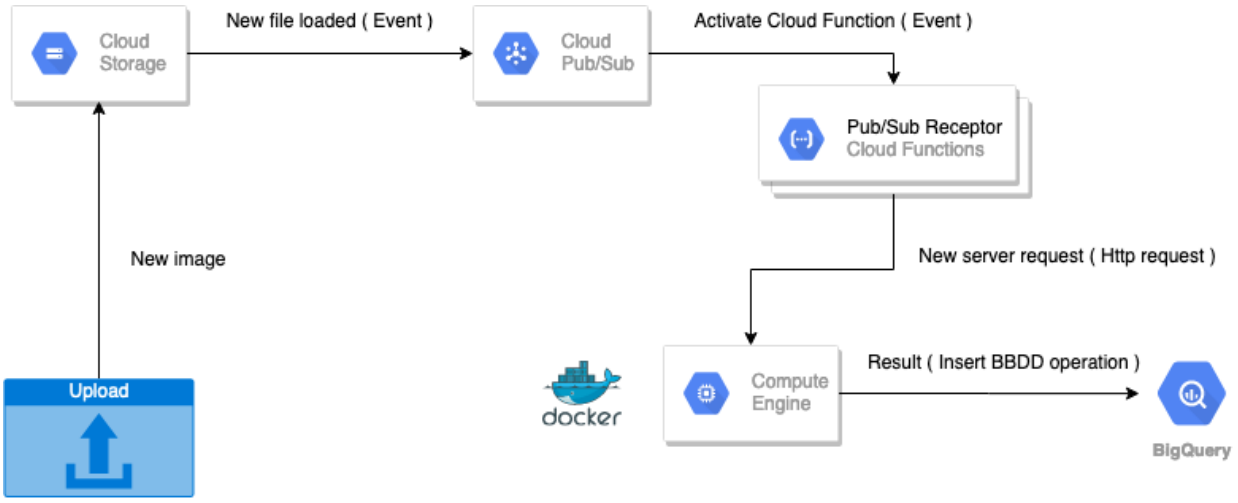


Figura 4.2: *Arquitectura Google Cloud.*

máticamente un evento que se introduce en la cola de mensajería de la aplicación 4.1.3. El servicio de Cloud Functions 4.1.5 consume todos estos mensajes de manera inmediata porque está configurado para activarse cada vez que un nuevo evento de este tipo ocurre. Por cada evento consumido una máquina virtual que se activa de manera instantánea será la encargada de formatear y enviar la petición al servidor. El servidor 4.1.4, con toda la información acerca de la imagen, procederá a usar la red de inferencia conveniente para finalmente insertar los resultados en la base de datos 4.1.2.

4.3. Codificación de los servidores web

Con el objetivo de poder procesar todas las peticiones y recoger todos los metadatos necesarios para su posterior análisis, se necesita codificar un servidor web que realice todo este trabajo. Para ello, se ha elegido el lenguaje de programación Python, debido a la creciente comunidad actual en el mundo del desarrollo software, lo que hace que el nivel de información sobre este lenguaje sea significativamente más alto que otros casos.

Se ha empleado un paradigma de programación orientado a objetos, por la organización que proporcionan con aplicaciones de un tamaño considerable, además de la dotación de identidad que facilita la distinción de los componentes.

Debido a la multitud de framework web actuales, se han elegido varios para su puesta a prueba en la aplicación.

En la Figura 4.3 podemos observar que las peticiones al servidor se han configurado para llegar al *endpoint* '*image*', que es donde se procesan las imágenes. Todas las peticiones al servidor se realizan de manera interna, por lo que el tráfico HTTP está cerrado al exterior, evitando así posibles ataques de denegación de servicio y la explotación de la aplicación sin consentimiento del propietario.

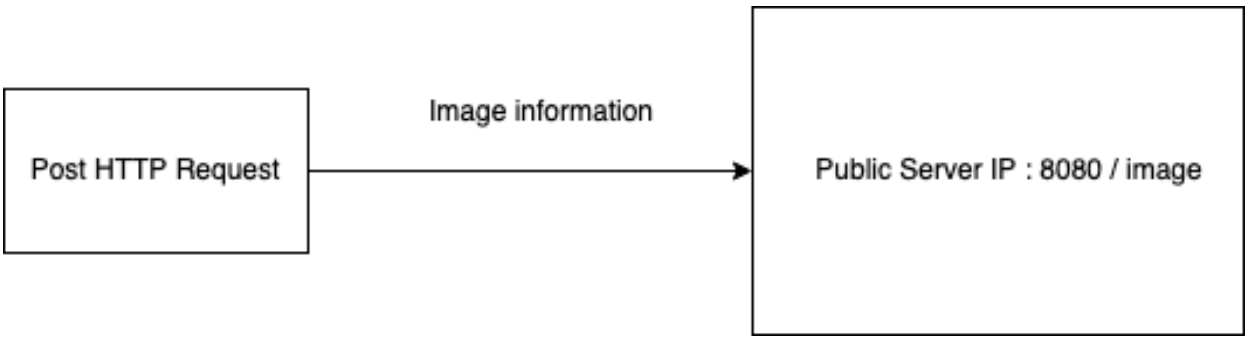


Figura 4.3: *Petición HTTP al servidor.*

Para el procesamiento generalizado de imágenes y el sistema de recogida de metadatos se han codificado dos clases principales. El primer Código 4.1 es el encargado de la inicialización de los servicios de Storage 4.1.1 y BigQuery 4.1.2, para los cuales se han codificado métodos para descarga y carga de información.

Listing 4.1: *Código Python para la API de la aplicación.*

```
1   class Api:
2   def __init__(self, net, sys: SystemTrack):
3       self.__storage = GoogleStorage()
4       self.__big_query = GoogleBigQuery()
5       self.__net = net
6       self.sys_track = sys
7
8   def cloud_storage_request(self, item: dict):
9       start = time.time()
10      image_name = item['name']
11      size = item['size']
12      file_type = item['contentType']
13      time_created = item['timeCreated']
```

```
14          image_path = f'{data_dir}{image_name}'
15          self.__storage.download_blob(bucket, image_name, image_path)
16          prediction, inference_time = self.__net.process_image(image_path)
17          total_time = time.time() - start
18          row = [
19              (
20                  image_name, size, file_type,
21                  time_created, prediction, inference_time,
22                  total_time, self.sys_track.physical_cores,
23                  self.sys_track.total_cores, self.sys_track.system, self.sys_track.processor,
24                  self.sys_track.system_memory,
25                  self.sys_track.system_memory_available,
26                  self.sys_track.so_version, self.sys_track.so_release, self.sys_track.inference_engine,
27                  self.sys_track.web_engine, self.sys_track.processor_unit,
28                  self.sys_track.docker, self.sys_track.cloud
29              )
30          ]
31          self.__big_query.insert_row(row)
32          os.remove(image_path)
```

Se presenta un método principal para procesar las peticiones al servidor, que modela toda la información que va a ser incluida en la base datos. Esta información consta de los siguientes campos:

- Nombre de la imagen.
- Tipo de archivo, que especifica el formato del fichero.
- Fecha exacta de creación del archivo en el sistema de almacenamiento.
- Clasificación de la imagen, para saber si el terreno de esta está dañado o no.
- Tiempo de inferencia en la red. En este caso dependerá de si estamos usando OpenVINO o TensorFlow serving para realizar esta tarea.
- Tiempo total de ejecución desde que se que llega una petición al servidor hasta que se procesa.
- Número de núcleos físicos del procesador.

- Número de núcleos virtuales del procesador.
- Sistema operativo.
- Versión del sistema operativo.
- Memoria RAM del sistema.
- Sistema de inferencia, en este caso puede ser OpenVINO o TensorFlow.
- Framework web utilizado: Flask o FastAPI en esta aplicación.
- Campo booleano para determinar si se está usando Docker para encapsular la aplicación. Para realizar las pruebas siempre se ha usado Docker.
- Campo booleano para saber si la aplicación se está ejecutando en un entorno local o en cloud. En este caso siempre se ejecuta la aplicación en un entorno cloud.

El segundo Código 4.2 es el encargado de generar y recabar toda esta información para que la clase anterior 4.1 pueda procesarla. Se ha hecho uso de las librerías de psutil[2] y platform[3] para conseguir la información necesaria del sistema. Contamos de manera adicional con un método de conversión de unidades para normalizar los datos a un estándar preestablecido.

Este objeto de tipo SystemTrack es el que será enviado por argumento a la clase API, que procesará todos estos datos.

Listing 4.2: *Clase Python para generar información sobre el sistema.*

```
1
2 class SystemTrack:
3     def __init__(self, docker: bool, inference_engine: str,
4             web_engine: str, cloud: bool, processor_unit: str):
5         self.sys_information = platform.uname()
6         self.sys_memory = psutil.virtual_memory()
7         self.physical_cores = psutil.cpu_count(logical=False)
```

[2] https://pypi.org/project/psutil/
[3] https://docs.python.org/3/library/platform.html

```
        self.total_cores = psutil.cpu_count(logical=True)
        self.system = self.sys_information.system
        self.processor = self.sys_information.processor
        self.system_memory = self.__get_size(self.sys_memory.total)
        self.system_memory_available = self.__get_size(self.sys_memory.available)
        self.so_version = self.sys_information.version
        self.so_release = self.sys_information.release

        self.docker = docker
        self.inference_engine = inference_engine
        self.web_engine = web_engine
        self.cloud = cloud
        self.processor_unit = processor_unit

    @staticmethod
    def __get_size(num_bytes, suffix="B"):
        factor = 1024
        for unit in ['', 'K', 'M', 'G', 'T', 'P']:
            if num_bytes < factor:
                return f'{num_bytes:.2f}{unit}{suffix}'
            num_bytes /= factor
```

4.3.1. Framework FastAPI

FastAPI[4] es un framework web de alto rendimiento preparado para su puesta en producción.

Las características principales de este framework son las siguientes :

- Desarrollo rápido debido a la arquitectura de componentes del framework.
- Documentación detallada para cada componente.
- Incorpora un sistema de tipado de objetos para su fácil identificación a la hora de codificar.
- Estándar OpenApi[5] para la especificación del formato de las peticiones HTTP entrantes y las respuestas del servidor.

[4] https://fastapi.tiangolo.com/
[5] https://github.com/OAI/OpenAPI-Specification

Para soportar este framework se usará Uvicorn[6] como servidor ASGI[7] (Asynchronous Server Gateway Interface), lo que significa que el servidor puede procesar tanto peticiones asíncronas como síncronas. Ademas, nos proporcionará las herramientas necesarias para paralelizar la carga de las peticiones entre los distintos nodos e hilos de la aplicación. Adicionalmente, se dispondrán de opciones de configuración de certificados SSL, logs y puerto por el que funciona la aplicación dentro del host. Este servidor soporta actualmente el protocolo de transmisión de datos HTTP/1.1 y está preparado para recibir peticiones asíncronas. Este software es de código abierto y podemos encontrar su código fuente en su repositorio oficial.

4.3.2. Framework Flask

Flask[8] es el framework ligero por excelencia de Python. Está desarrollado de manera sencilla y sin componentes adicionales, diferenciándose así de otros frameworks pesados como Django[9], que incluyen muchas dependencias y acaban ocupando mucho espacio en disco y en memoria cuando son ejecutados.

A diferencia de FastAPI, Flask [19] se centra en la simplicidad de sus elementos y dota a sus usuarios se una serie de interfaces y decoradores simples para su codificación.

Como servidor web se usará Gunicorn[10] que sigue el estándar WSGI[11] (Web Server Gateway Interface), una convención simple para gestionar llamadas síncronas al servidor. Con este servidor podremos seleccionar la paralelización y distribución de carga del procesador.

4.4. Encapsulación de entorno con Docker

En las Figuras 4.4 y 4.5 podemos observar la arquitectura de máquina virtual para OpenVINO y TensorFlow respectivamente. Las dos soluciones se desplegarán en las máquinas virtuales de Google.

[6] https://www.uvicorn.org/
[7] https://asgi.readthedocs.io/en/latest/
[8] https://flask.palletsprojects.com/en/1.1.x/
[9] https://www.djangoproject.com/
[10] https://gunicorn.org/
[11] https://wsgi.readthedocs.io/en/latest/learn.html

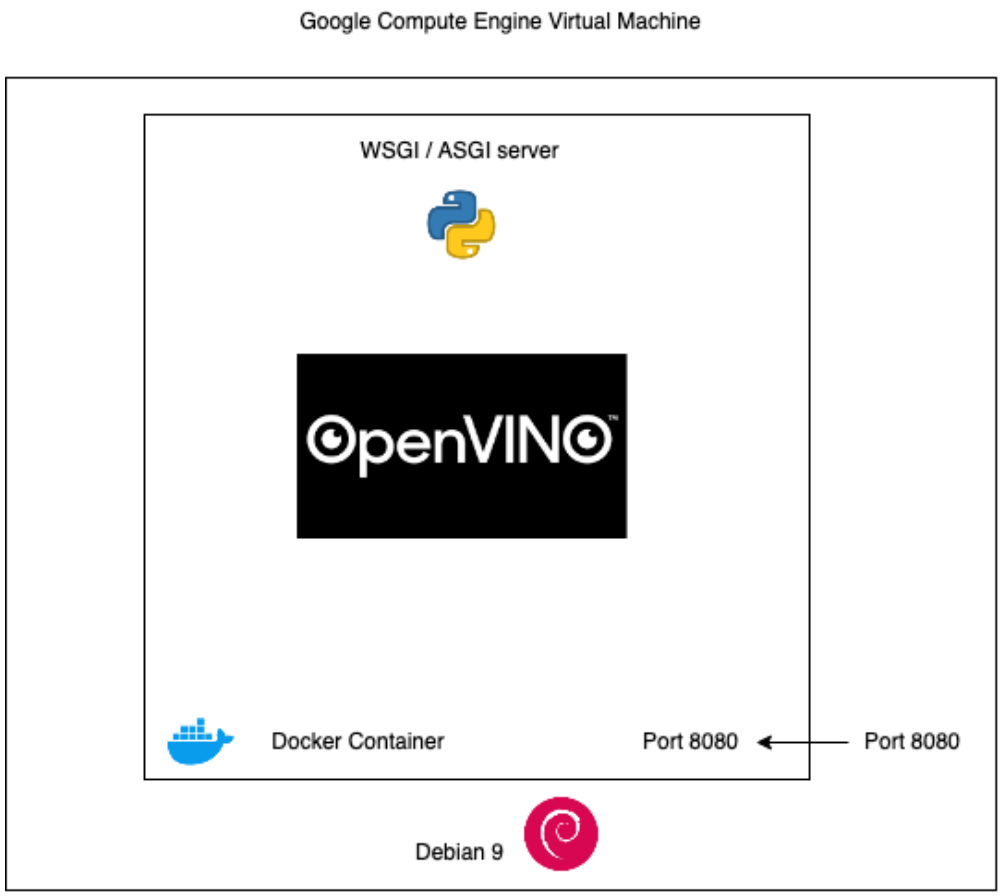


Figura 4.4: *Arquitectura de la máquina virtual de OpenVINO.*

Se han encapsulado todas las aplicaciones usando la tecnología de contenedores Docker[12] buscando la máxima portabilidad entre entornos y así no cerrar la posibilidad de traslado a otra plataforma.

Los contenedores de Docker se conectarán con el host mediante el puerto 8080, permitiendo así el tráfico de información. Dentro del contenedor se levantarán los correspondientes servidores WSGI o ASGI, dependiendo de si estamos usando Flask o FastAPI respectivamente.

Es necesario clarificar que nunca habrá dos servidores activos al mismo tiempo, por lo que se dispondrán de distintas versiones de las aplicaciones en el registro de contenedores, preparadas para usar cada servidor de manera independiente.

[12] https://www.docker.com/

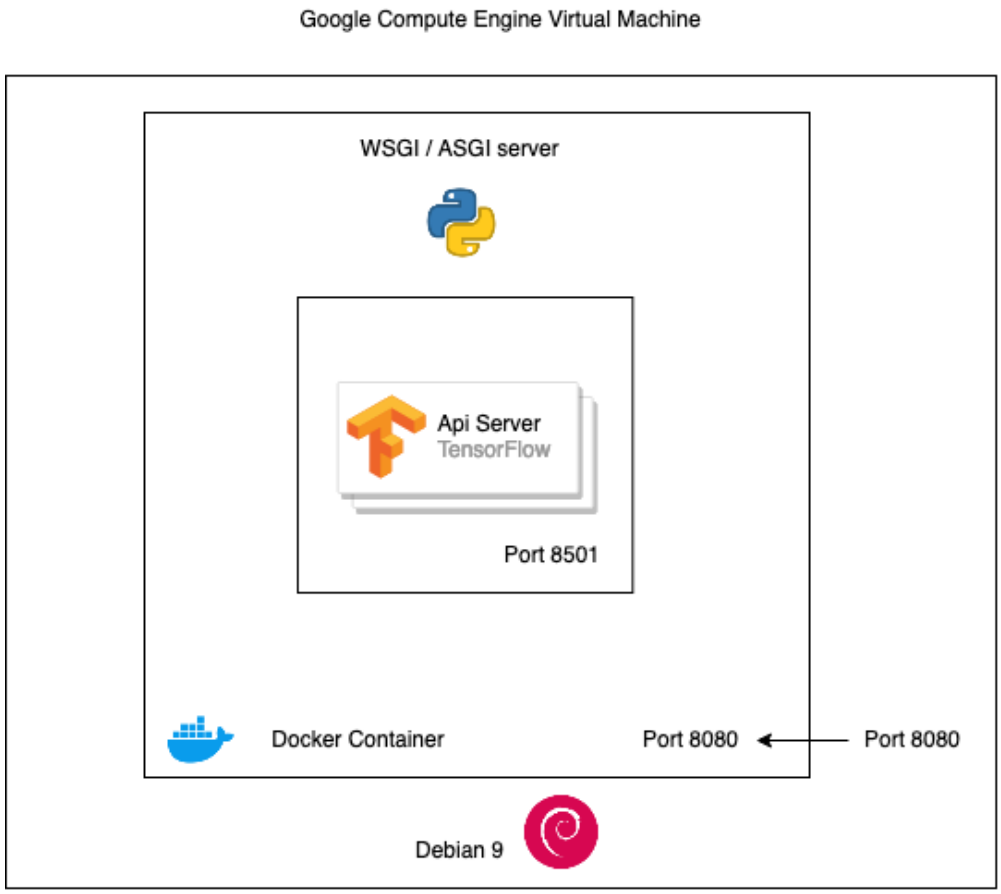


Figura 4.5: *Arquitectura de la máquina virtual de TensorFlow.*

Dentro de estos contenedores, y dependiendo de la aplicación a usar, se puede encontrar directamente la aplicación de inferencia de OpenVINO, la cual funcionará de manera directa, o, por el contrario, si usamos TensorFlow tendremos otro servidor REST API que procesará las peticiones dentro del contenedor por el puerto 8501. De igual modo, en los servidores nunca existirá la posibilidad de tener instalados OpenVINO y TensorFlow al mismo tiempo, por lo que habrá que independizar su uso en función de los requisitos y necesidades del usuario.

Se han versionado las siguientes imágenes de Docker para la aplicación :

- Imagen para la aplicación de entrenamiento del modelo de Deep Learning, que ha sido utilizada para realizar pruebas de concepto en un entorno de desarrollo local

antes de usar su funcionalidad interna en Google Colab. Se ha encontrado útil su uso debido a que el proceso de desarrollo se ha llevado a cabo en distintos entornos y sistemas operativos como MacOS, Windows o Linux dependiendo de las necesidades del programador.

- Imagen para la red de inferencia de OpenVINO, que contiene las librerías necesarias para su ejecución y puesta en producción.
- Imagen para la red de inferencia de TensorFlow, configurando el servidor interno que proporciona las inferencias al contenedor, y este a la base de datos.

Para la construcción de estas imágenes se han codificado de manera explícita cada una de ellas y constan en el repositorio de este trabajo[13]. En su configuración se especifican los siguientes puntos:

- Sistema operativo base o imagen de la que hereda.
- Puertos necesarios para ejecutar la aplicación, que son expuestos al exterior.
- Paquetes y actualizaciones del sistema operativo necesarios para funcionar.
- Librerías de Python.
- Variables de entorno del sistema operativo.
- Código y ficheros que se van a incluir en la imagen.

Como herramienta adicional y enlazada al uso de Docker se ha utilizado Docker Compose[14] para las pruebas de funcionamiento y testeo de todas estas imágenes. Con esta aplicación podemos describir exactamente cómo se va a ejecutar cada una de las imágenes de Docker y cuáles son los comandos que queremos que se ejecuten de manera automática al iniciarse el contenedor.

[13] https://github.com/A-Ortiz-L/multispectral-imaging-cnn-final-degree-work/tree/master/docker

[14] https://docs.docker.com/compose/

Capítulo 5

Resultados experimentales

5.1. Dataset y Hardware

Para el entrenamiento del modelo de *Deep Learning* se ha usado un conjunto de datos de 268 imágenes RGB. En cuanto a la muestra presentada, para el estudio de las métricas de inferencia y pruebas de carga se han realizado 64000 peticiones al servidor, con imágenes cargadas en el sistema de almacenamiento que han sido procesadas en el entorno productivo de la aplicación. El tamaño de la muestra es mucho mayor que el de las imágenes de origen. Esto se debe a que se han cargado de manera reiterada muchas de ellas con el objetivo de generar numerosas peticiones concurrentes en el sistema. La división del conjunto de datos es la siguiente :

- 8000 muestras con un procesador de 2 núcleos físicos, 4 hilos, 4 GB de RAM usando Flask y TensorFlow.
- 8000 muestras con un procesador de 2 núcleos físicos, 4 hilos, 4 GB de RAM usando FastAPI y TensorFlow.
- 8000 muestras con un procesador de 4 núcleos físicos, 8 hilos, 8 GB de RAM usando FastAPI y TensorFlow.
- 8000 muestras con un procesador de 4 núcleos físicos, 8 hilos, 8 GB de RAM usando Flask y TensorFlow.

- 8000 muestras con un procesador de 2 núcleos físicos, 4 hilos, 4 GB de RAM usando Flask y OpenVINO.
- 8000 muestras con un procesador de 2 núcleos físicos, 4 hilos, 4 GB de RAM usando FastAPI y OpenVINO.
- 8000 muestras con un procesador de 4 núcleos físicos, 8 hilos, 8 GB de RAM usando Flask y OpenVINO.
- 8000 muestras con un procesador de 4 núcleos físicos, 8 hilos, 8 GB de RAM usando FastAPI y OpenVINO.

Estas imágenes han sido testeadas por los distintos entornos productivos, sistemas de inferencia y frameworks web. La carga de las imágenes al sistema de almacenamiento se ha realizado de manera paralela, gracias al soporte multi-threading de Google Storage. El equipo que ha realizado la carga tiene como hardware principal los siguientes componentes:

- Procesador AMD Ryzen 5–3600 @ 4.2 GHz (6 núcleos físicos, 12 hilos)
- 16 GB de memoria RAM DDR4 @ 3200 MHz.
- Conexión a internet de fibra óptica simétrica de 600 MB.

La comparativa de resultados y el correspondiente análisis ha sido realizado en la base de datos distribuida BigQuery, haciendo uso de SQL estándar.

5.2. Rendimiento en fase de entrenamiento

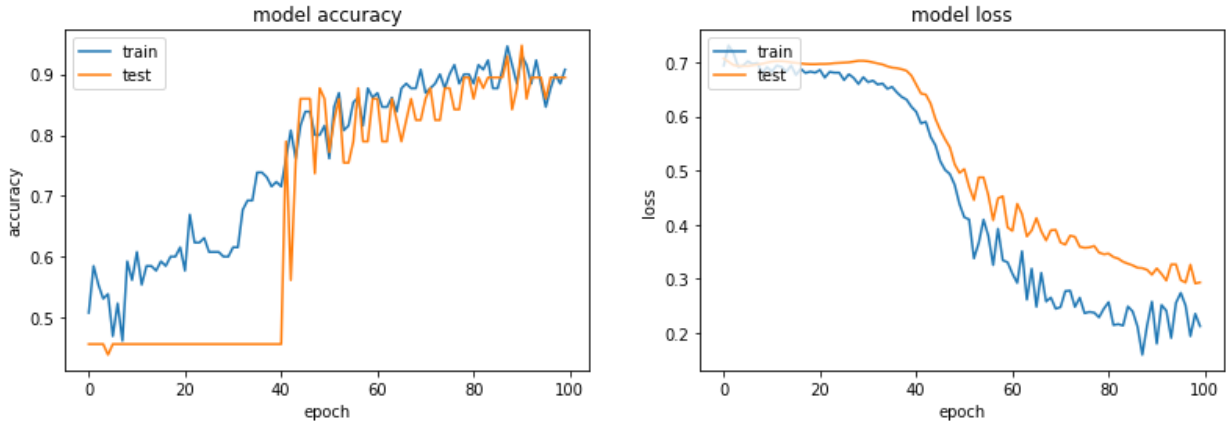
Para obtener los resultados de este experimento se ha usado el hardware disponible en la plataforma de Google Colab, usando como comparativa:

- Entrenamiento usando un procesador Intel(R) Xeon(R) CPU @ 2.30GHz.
- Entrenamiento con una GPU Tesla K80.

El tiempo total de entrenamiento utilizando la CPU ha sido de 11 minutos. El nivel de acierto de clasificación en ambas redes supera el 85 % en el conjunto de datos de prueba. El resultado más fiable y rápido utilizando la Tesla K80 ha sido una configuración en la red neuronal de 175 Epochs y 256 de Batch-size.

5.2.1. Configuración de 100 Epochs y 256 Batch-size

Con un tiempo total de entrenamiento de 16.79 segundos y una precisión del 85 % sobre el conjunto de datos de entrenamiento. En las Figuras 5.1 (a) y 5.1 (b) se encuentran los resultados de entrenamiento en términos de precisión y pérdida, respectivamente.

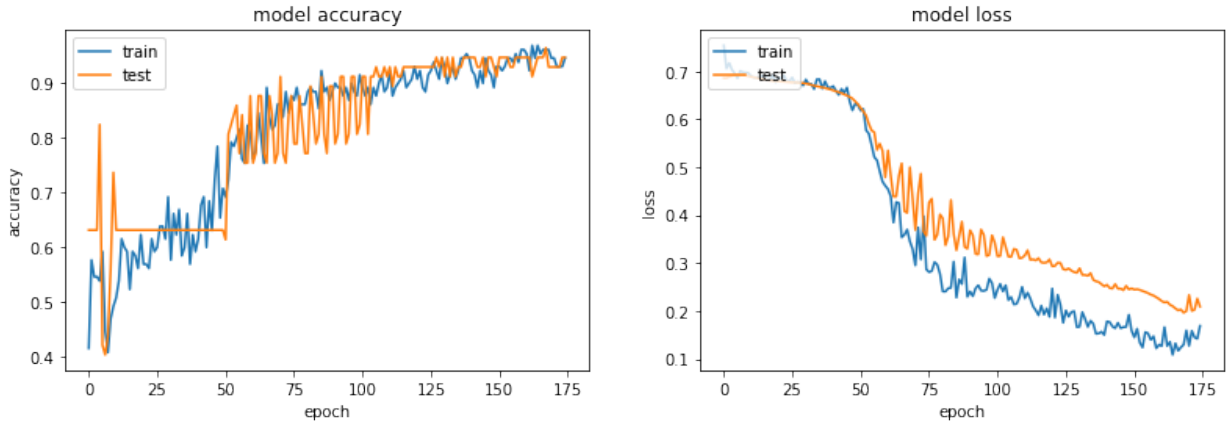


(a) (b)

Figura 5.1: *Resultados de entrenamiento del modelo usando una GPU Tesla K80 con un Batch-size de 256 y 100 Epochs. (a) Rendimiento del modelo en acierto. (b) Rendimiento del modelo en pérdida.*

5.2.2. Configuración de 175 Epochs y 256 Batch-size

Con un tiempo total de entrenamiento de 25.86 segundos y una precisión del 93 % sobre el conjunto de datos de entrenamiento. En las Figuras 5.2 (a) y 5.2 (b) se encuentran los resultados de entrenamiento en términos de precisión y pérdida.

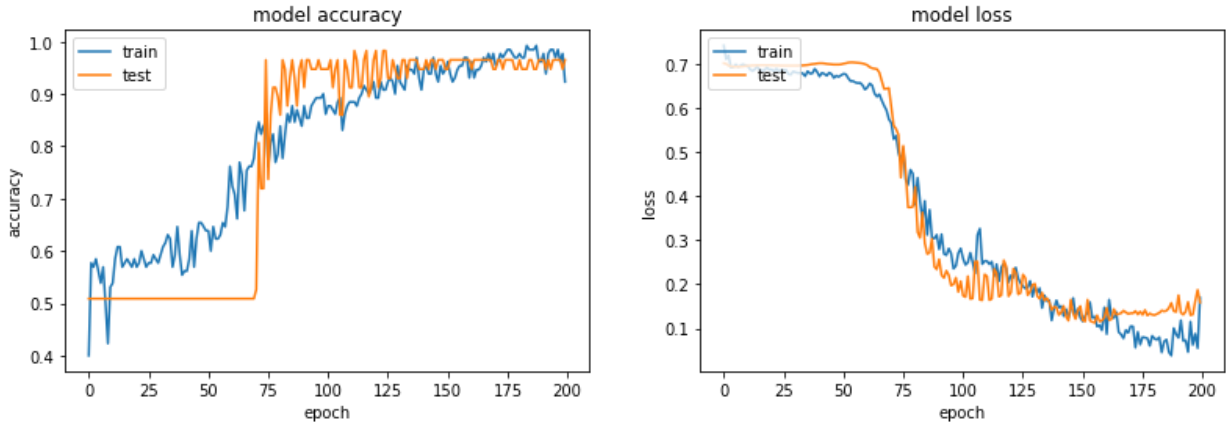


(a) (b)

Figura 5.2: *Resultados de entrenamiento del modelo usando una GPU Tesla K80 con un Batch-size de 256 y 175 Epochs. (a) Rendimiento del modelo en acierto. (b) Rendimiento del modelo en pérdida.*

5.2.3. Configuración de 200 Epochs y 256 Batch-size

Con un tiempo total de entrenamiento de 29.94 segundos y una precisión del 87 % sobre el conjunto de datos de entrenamiento. Ver Figuras 5.3 (a) y 5.3 (b) para los resultados en precisión y pérdida.



(a) (b)

Figura 5.3: *Resultados de entrenamiento del modelo usando una GPU Tesla K80 con un Batch-size de 256 y 200 Epochs. (a) Rendimiento del modelo en acierto. (b) Rendimiento del modelo en pérdida.*

Podemos observar en la Tabla 5.1 que la configuración con 100 Epoch es la que mejor resultado da en cuanto a velocidad, sin embargo, no es la más fiable en cuanto a pérdida del modelo, pudiendo reducir aún más el número de fallos si se aumentan las iteraciones. Las configuraciones restantes mantienen un nivel alto de predicción. La variante de 200 Epoch al final de su entrenamiento comienza a denotar una tendencia ascendente en el número fallos. Escogeremos la solución de 175 Epoch al ser más rápida que la de 200 Epoch y más fiable en el número de fallos que la de 100.

epoch	accuracy	time	loss
100	85 %	16.79 s	2.1 %
175	93 %	25.86 s	1.5 %
200	87 %	29.94 s	1.9 %

Tabla 5.1: *Comparativa en el número de Epochs y tiempo de entrenamiento.*

5.3. Rendimiento en fase de inferencias

Se han configurado los distintos servidores web para que empleen todos los núcleos del procesador de manera concurrente, de modo que la capacidad de procesamiento de peticiones sea la máxima posible.

Los resultados se presentan haciendo uso de una muestra de 64000 peticiones de inferencia en el entorno de producción de la aplicación, 32000 para la red de TensorFlow y 32000 para la red de OpenVINO. La unidad de cálculo principal es el procesador, siendo su modelo un Intel Xeon (Cascada Lake) con una frecuencia de 2.8 GHz de base y un turbo hasta 3.4 GHz.

Se han probado distintas configuraciones de este procesador, tanto en su versión de 2 núcleos físicos (4 virtuales), como en la de 4 núcleos físicos (8 virtuales). La memoria RAM utilizada varía de 4 GB con el procesador de 2 núcleos físicos a 8 GB en la versión de 4 núcleos físicos.

La memoria disponible en todas todas las configuraciones es inferior a la contratada, ya

que el total incluye el almacenamiento del sistema operativo y librerías esenciales para su funcionamiento, así como el servicio de Docker.

5.3.1. Rendimiento de acierto en la predicción.

En la Tabla 5.2 podemos observar que la tasa de acierto para ambas redes mantiene un porcentaje de acierto elevado que supera el 97 %. A lo largo de las 32000 inferencias tomadas para cada red se encuentra una diferencia notable en el número de predicciones erróneas. OpenVINO tiene una tasa de error 1.7 veces superior a TensorFlow. Esto es debido a que en OpenVINO se está usando un método de optimización basado en la reducción del tamaño en memoria que ocupan sus variables. En OpenVINO la red de inferencia opera de manera predeterminada con variables en punto flotante de 32 bits, que con el cambio son de 16 bits. La mejora que nos da esta optimización en la velocidad de inferencia de OpenVINO causa una disminución del potencial de acierto de la red.

inference_engine	samples	right_guess	error	percentage_guess
OpenVINO	32000	31141	859	97 %
TensorFlow	32000	31521	479	98 %

Tabla 5.2: *Acierto de inferencia en OpenVINO y TensorFlow.*

5.3.2. Rendimiento en tiempo de inferencia.

En la Tabla 5.3 se puede observar como la media de inferencia a lo largo de las 64000 muestras tomadas es de 106 ms para TensorFlow y 3 ms para OpenVINO, lo que supone una velocidad de inferencia 35 veces superior de OpenVINO frente a TensorFlow.

inference_engine	samples	avg_inference_ms	avg_ms_total
TensorFlow	32000	106 ms	381 ms
OpenVINO	32000	3 ms	108 ms

Tabla 5.3: *Tiempo de inferencia en OpenVINO y TensorFlow.*

El tiempo total de inferencia, que incluye la latencia de la mensajería entre los distintos componentes y el servidor es de 381 ms para TensorFlow y 108 ms para OpenVINO. La solución de OpenVINO no incluye servidores adicionales para su funcionamiento como si hace TensorFlow, por lo que no tiene latencia adicional en este punto.

5.3.3. Rendimiento en tiempo de inferencia con distinto hardware.

Podemos visualizar en la Tabla 5.4 que el tiempo total de inferencia que incluye la latencia del servidor web, los pipelines de procesamiento del entorno y la propia inferencia, es inferior al segundo en ambos casos.

Las distintas configuraciones hardware también revelan que el consumo de memoria del sistema de inferencia de TensorFlow afecta al rendimiento de la aplicación, mejorando mucho su rendimiento con un hardware más potente. OpenVINO, por el contrario, mantiene resultados similares con un hardware de bajo coste, con unos resultados de inferencia de 3 ms con el procesador de 2 núcleos y 2 ms con el de 4 núcleos.

inference_engine	samples	physical_core	system_memory	avg_inference_ms	avg_total_ms
OpenVINO	16000	2	3.46 GB	3 ms	87 ms
OpenVINO	16000	4	7.00 GB	2 ms	128 ms
TensorFlow	16000	2	3.46 GB	146 ms	521 ms
TensorFlow	16000	4	7.00 GB	67 ms	241 ms

Tabla 5.4: *Tiempo de inferencia en OpenVINO y TensorFlow con distinto hardware.*

Se contempla que el tiempo medio total de inferencia aumenta pese a mejorar el hardware en OpenVINO. Con el procesador de 4 núcleos tenemos un número total de 8 hilos en el servidor web, el coste de paralelizar las llamadas entre los distintos nodos eleva el coste total de la petición HTTP.

TensorFlow mejora 2 veces la velocidad de inferencia pasando de un procesador de 2 núcleos y 4 GB de RAM a uno de 4 núcleos y 8 GB de RAM teniendo un tiempo de 146 ms con el primero y 67 ms con el segundo, lo que denota que el consumo de su servicio de inferencia requiere de un hardware más potente. Del mismo modo mejora 2 veces su tiempo

de inferencia total con la mejora del hardware pasando de 521 ms a 241 ms.

5.3.4. Rendimiento en tiempo de inferencia con distinto framework web.

Es necesario recordar que cada framework web está configurado con un tipo de servidor distinto que pone en marcha su funcionamiento.

Flask se configura junto con el servidor WSGI[1] Gunicorn[2], mientras que FastAPI se configura con el servidor ASGI[3] Uvicorn[4]. El rendimiento de estos servidores influirá en el tiempo total de la llamada HTTP.

En la Tabla 5.5 podemos contemplar que OpenVINO se mantiene estable con ambos framework web, con un ligero aumento de la latencia haciendo uso de FastAPI.

TensorFlow se comporta mucho mejor con Flask, mejorando en 2 veces su tiempo de inferencia en la red pasando de 148 ms con FastAPI a 65 ms con Flask y 2 veces en el tiempo total de ejecución teniendo un rendimiento de 517 ms con FastAPI y 246 ms con Flask.

inference_engine	samples	web_engine	avg_inference_ms	avg_total_ms
OpenVINO	16000	Flask	2 ms	103 ms
OpenVINO	16000	FastAPI	3 ms	112 ms
TensorFlow	16000	Flask	65 ms	246 ms
TensorFlow	16000	FastAPI	148 ms	517 ms

Tabla 5.5: *Comparativa de tiempo de inferencia en OpenVINO y TensorFlow con distinto framework web.*

Flask es un servidor web con los mínimos componentes para funcionar, pero configurado de la manera correcta puede ser el framework idóneo para realizar una tarea específica. En este caso, la complejidad del servicio de TensorFlow convive mejor con un framework web sin demasiados componentes adicionales.

[1] https://linuxgazette.net/115/orr.html
[2] https://gunicorn.org/
[3] https://channels.readthedocs.io/en/latest/asgi.html
[4] https://www.uvicorn.org/

Las mejoras que proporciona FastAPI como un sistema de logging detallado de las ejecuciones y algunas características adicionales para el desarrollador[5], pueden ocasionar cierto aumento de la latencia en los tiempos. Aun así, cuando la fiabilidad es uno de los requisitos y objetivos principales, estas mejoras pueden valer la pena a la hora de escalar nuestra aplicación.

5.3.5. Rendimiento en tiempo de inferencia con distinto framework web y hardware.

En general, FastAPI requiere de un hardware más potente para sacar su máximo rendimiento, mientras que con un *framework* minimalista como Flask podemos optar por reducir costes en hardware sin penalizar demasiado el rendimiento.

En la Tabla 5.6 podemos ver el rendimiento según hardware, servidor web y sistema de inferencia utilizado. El sistema de inferencia más rápido en todas las casuísticas es OpenVINO, siendo su configuración con 4 núcleos, 8 GB y Flask la más veloz, con 2 ms de inferencia y 126 ms de procesamiento total de la petición. Esta opción también permite la máxima paralelización de llamadas posibles, al tener consigo los 8 hilos del procesador de 4 núcleos.

engine	samples	cores	memory	framework	avg_inference_ms	avg_total_ms
OpenVINO	8000	2	3.46 GB	Flask	3 ms	81 ms
OpenVINO	8000	2	3.46 GB	FastAPI	4 ms	94 ms
OpenVINO	8000	4	7.00 GB	Flask	2 ms	126 ms
OpenVINO	8000	4	7.00 GB	FastAPI	2 ms	130 ms
TensorFlow	8000	2	3.46 GB	Flask	64 ms	221 ms
TensorFlow	8000	2	3.46 GB	FastAPI	229 ms	821 ms
TensorFlow	8000	4	7.00 GB	Flask	67 ms	270 ms
TensorFlow	8000	4	7.00 GB	FastAPI	67 ms	212 ms

Tabla 5.6: *Comparativa de tiempo de inferencia en OpenVINO y TensorFlow con distinto hardware y servidor web.*

[5] https://fastapi.tiangolo.com/features/

5.3.6. Número de peticiones por segundo soportadas por el servidor.

Se ha tenido en cuenta el rendimiento del framework web, pero no su parte más importante, la paralelización. Es indispensable saber el número de peticiones que podemos soportar por segundo.

Para este experimento (ver Tabla 5.7) se han tomado nuevas muestras para cada sistema de inferencia. La configuración hardware es la más potente posible para ambos casos. Se ha decidido usar el framework de FastAPI ya que su sistema de logging permitirá al desarrollador saber si todas las peticiones enviadas al servidor llegan de manera correcta.

inference_engine	samples	seconds	avg_image_s	requests_received
OpenVINO	8000	119 s	67 img/s	8000
TensorFlow	8000	125 s	64 img/s	8000

Tabla 5.7: *Peticiones por segundo soportadas por el servidor para OpenVINO y TensorFlow.*

El número total de peticiones recibidas por el servidor, en ambos casos, es un 100 % del total de enviadas, por lo que la aplicación soporta con creces este volumen de trabajo.

TensorFlow pese a tener un sistema de inferencia claramente inferior al de OpenVINO es capaz de compensar esta carencia debido a la concurrencia que ofrece el servidor, consiguiendo un rendimiento similar en el tiempo total del procesamiento de todas las imágenes.

Hay que señalar que las dos redes están recibiendo las mismas peticiones por minuto, por lo que puede que alguna no se esté aprovechando del todo y sea capaz de realizar más cálculos por unidad de tiempo.

5.3.7. Rendimiento de inferencia en un entorno local.

Toda la infraestructura que soporta la aplicación acumula latencia en cada conexión que hace entre sus componentes. Transportar todo este procesamiento a un entorno local, en el que las imágenes ya estén listas para procesar, minimiza el tiempo total de inferencia. En este punto ponemos a prueba los sistemas de inferencia sin herramientas que puedan

generar ruido. Para esta prueba hemos conservado el hardware más potente que usábamos en los servidores web: un procesador con 4 núcleos físicos y 8 GB de memoria RAM.

Como podemos observar en la Figura 5.4, se reafirman los resultados obtenidos en las pruebas de rendimiento de inferencia en el pipeline del servidor. OpenVINO es mucho más rápido calculando los resultados de las predicciones. Es en este entorno donde podemos ver el verdadero potencial de usar un lenguaje de bajo nivel. Esto nos permite gestionar operaciones en memoria, manipular el borrado de variables que ya no se necesitan, elegir el número de BYTES que ocupan nuestras estructuras de datos. El poder de decisión que nos brindan estas ventajas se convierten en rendimiento cuando ponemos a prueba nuestra aplicación. En este entorno conseguimos una tasa de 541 img/s para OpenVINO y 39 img/s para TensorFlow. Sin la gestión de concurrencia que proporciona el servidor web la diferencia es notable. TensorFlow no ofrece una rápida inferencia unitaria, por lo que su capacidad de procesamiento no es la mejor en este entorno.

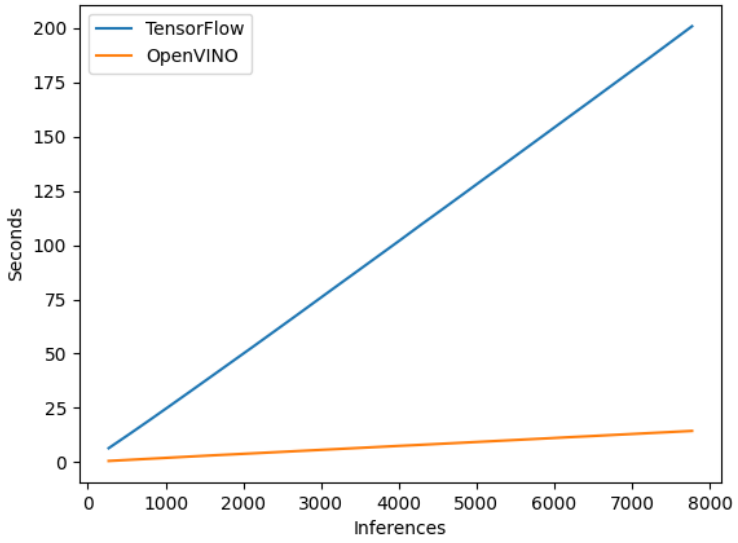


Figura 5.4: *Comparativa de inferencias entre TensorFlow y OpenVINO en un entorno local.*

Todos los resultados y registros se encuentran en el repositorio oficial del trabajo en

GitHub[6].

5.4. Costes del proyecto

A continuación se presentan los costes del proyecto de toda la plataforma de producción. La suma total durante todo el desarrollo del proyecto del entorno productivo es de 10.20 dólares. Estos costes han sido recogidos haciendo uso de la calculadora de precios de Google[7].

- Máquina virtual 4 núcleos y 4 GB de memoria RAM. Un total de 6.80 dólares por un uso de 24 horas, que fue el tiempo utilizado para realizar pruebas de concepto y cargas en este trabajo.
- Máquina virtual 2 núcleos y 4 GB de memoria RAM. Un total de 3.40 dólares por un uso de 24 horas, que fue el tiempo real consumido para este servicio.
- BigQuery, con un coste de 0.00 dólares para un almacenamiento de 1 GB de tablas en la base de datos, 1 GB de procesamiento en tiempo real y 1 GB de trabajos SQL al realizar los análisis de resultados.
- Pub/Sub con un coste total de 0.00 dólares, ya que su uso entraba dentro del rango gratuito del servicio.
- Cloud Functions, con un coste total de 0.00 dólares, haciendo uso de la modalidad gratuita, que permite hasta 2 millones de llamadas al mes.

Según la calculadora de salarios de Stack Overflow[8], el salario bruto de un ingeniero *Data Scientist junior* en Madrid, que maneja herramientas como Docker, Google Cloud Platform, Python, TensorFlow y SQL, puede rondar los 28000 y 42000 euros brutos al año.

[6] https://github.com/A-Ortiz-L/multispectral-imaging-cnn-final-degree-work/tree/master/result/snapshot

[7] https://cloud.Google.com/products/calculator?hl=es

[8] https://stackoverflow.blog/2019/10/16/coding-salaries-in-2019-updating-the-stack-overflow-salary-calculator/

El total de semanas consumidas en la consecución de este proyecto es de diez. Suponiendo un salario bruto de 28000 euros anuales, un desarrollador tendría que estar contratado durante diez semanas a jornada completa para la elaboración del proyecto, lo que resultaría en un total de 5832 euros.

Capítulo 6

Conclusiones y trabajo futuro

6.1. Conclusiones

Los efectos de la globalizacion de internet son cada día más perceptibles en nuestras vidas. El aumento de la población que interactúa por medio de aplicaciones tiene como consecuencia la generación de una cantidad infinita de datos, que contienen toda la información de los usuarios que las emplean. Desde páginas web, aplicaciones móviles hasta sistemas que interactúan de manera automática, sin necesidad de que los usuarios tengan que usarlos o configurarlos. Hasta hace unos pocos años, estos ámbitos apenas escapaban de la esfera académica e investigadora, mientras que en los tiempos que corren se han convertido en una necesidad real de un mundo cada vez más interconectado a escala global.

A medida que crecen los datos en la red, aumenta la necesidad de su análisis y explotación. Por ello, plataformas como Google Cloud, Amazon Web Services y similares llevan a cabo este proceso, brindando toda la potencia autogestionada de cómputo a los usuarios. Diariamente nacen nuevas tecnologías de análisis específicas para estos datos. Este es el caso de TensorFlow u OpenVINO, propiedad de empresas comerciales como Google o Intel. La primera de ellas fue convertida a formato de código abierto, mientras que la segunda se situó como gratuita desde su lanzamiento; ello da debida cuenta de la creciente comunidad que hace uso de estas herramientas. Este tipo de recursos gratuitos, como Google Colab, otorgan a la comunidad de soluciones que solían ser de pago y accesible solo para cierto público

dentro del sector tecnológico. Y, al mismo tiempo, es esa misma comunidad la que reporta errores, propone actualizaciones o codifica directamente nuevas soluciones a incorporar.

En este trabajo se ha conseguido entrenar un modelo de *Deep Learning* con un acierto del 93 % sobre el conjunto de datos de entrenamiento. Asimismo, dicho modelo ha sido optimizado para mejorar su inferencia haciendo uso de la herramienta de Intel OpenVINO. El modelo se ha productivizado en el entorno de Google Cloud mediante una arquitectura que configura y orquesta el pipeline del proceso completo. Todas las aplicaciones desplegadas están encapsuladas dentro de un contenedor Docker. El entorno de producción está preparado, por una parte, para desplegar la red de inferencia de OpenVINO y TensorFlow y, finalmente, para ingestar sus clasificaciones en tiempo real.

El tiempo de inferencia para ambas redes en el entorno productivo es inferior al segundo, siendo 67 ms en TensorFlow y 2 ms en OpenVINO. El tiempo total por petición HTTP es de 212 ms para TensorFlow y 126 ms para OpenVINO, siendo las opciones de configuración para estos resultados las que más procesamiento concurrente soportan. La tasa de acierto de clasificación en el entorno productivo es del 97 %.

Es la competitividad entre todas estas herramientas lo que impulsa una mejora constante en todas ellas. TensorFlow nació como un framework enfocado en la generación de modelos de Deep Learning, pero no con el objetivo principal de la aceleración de la inferencia de los modelos, como si hace OpenVINO. La simbiosis entre ambos crea una combinación completa que cubre las necesidades tanto de creación, como del uso de los modelos de aprendizaje profundo.

6.2. Trabajo futuro

La plataforma se ha configurado de manera que su capacidad de procesamiento de peticiones sea escalable, esto es, porque el uso de la tecnología de contenedores Docker está preparada para su integración en la solución de clúster Kubernetes[1]. Este trabajo ha usado

[1] https://kubernetes.io/es/docs/concepts/overview/what-is-Kubernetes/

Google Cloud como plataforma de despliegue de la aplicación, pero explorar otras soluciones como AWS o Azure daría una perspectiva más general de qué entorno está más preparado para construir y utilizar modelos de Deep Learning.

La visualización de resultados en este trabajo ha sido realizada mediante los trabajos SQL[2] en una base de datos distribuida, pero sería ideal poder contemplar los datos de una manera más intuitiva. Docker nos permite portar fácilmente esta solución de contenedores a otros sistemas con distinto hardware, y, en consecuencia, puede funcionar de manera local sin tener una plataforma web o cloud que la soporte. La clasificación podría producirse dentro del propio elemento que genera las imágenes, lo que eliminaría el tiempo de latencia de otros elementos adicionales.

Existe la posibilidad de introducir un nuevo proceso automático para reentrenar el modelo, siempre y cuando las nuevas muestras no afecten a la configuración actual. Los datos de este trabajo son imágenes RGB de una parte del planeta, por lo que tienen latitud y longitud exactas. La técnica de visionado Heatmap[3] es la idónea para ver en tiempo real las zonas del terreno más afectadas por el desastre natural y poder redirigir los equipos de emergencia de manera rápida. La configuración de los servidores ingesta las llamadas de manera interna, por lo que no está abierta a peticiones públicas de manera directa. Si se decidiera abrir al público la clasificación de imágenes, habría que configurar en los servidores todos los certificados necesarios para funcionar con el protocolo seguro HTTPS. Además, ciertas medidas de seguridad contra factores como la denegación de servicio o intento de conexión no deseados a los servidores por los puertos abiertos de la aplicación, en nuestro caso 8080 y 22, que usamos para conectarnos por SSH[4].

En nuestro problema un vehículo aéreo podría portar el hardware de clasificación y sobrevolar el terreno dañado para clasificar las imágenes que va recogiendo. Este sistema sería similar a opciones que ya se pueden encontrar en el mercado, como las AWS DeepLens[5]

[2] https://cloud.google.com/bigquery/docs/jobs-overview
[3] https://en.wikipedia.org/wiki/Heat_map
[4] https://www.ssh.com/ssh/
[5] https://aws.amazon.com/es/deeplens/

de Amazon.

Más allá de lógicas comerciales, sería deseable que este tipo de soluciones se extendiese en un formato de código abierto. Este formato permitiría la mejora del proyecto a través de la comunidad, que sería libre de hacer aportaciones.

Bibliografía

[1] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[2] David A. Yuen, Long Wang, Xuebin Chi, Lennart Johnsson, Wei Ge, and Yaolin Shi. *GPU Solutions to Multi-scale Problems in Science and Engineering*. Lecture Notes in Earth System Sciences. Springer Berlin Heidelberg, 2013.

[3] Giancarlo Zaccone and Md. Rezaul Karim. *Deep Learning with TensorFlow*. Packt Publishing, 2018.

[4] Vishnu Nath and Stephen E. Levinson. *Autonomous robotics and deep learning*. Springer, 2014.

[5] Arvind Narayanan, Joseph Bonneau, Edward Felten, Andrew Miller, and Steven Goldfeder. *Bitcoin and cryptocurrency technologies*. Princeton University Press, 2016.

[6] Le Lu, Yefeng Zheng, Gustavo Carneiro, and Lin Yang. *Deep learning and convolutional neural networks for medical image computing*. Springer, 2017.

[7] Steven Weber. *The success of open source*. Harvard University Press, 2004.

[8] Derrick Rountree and Ileana Castrillo. *The basics of cloud computing*. Syngress, 2014.

[9] Luis P. Coelho and Willi Richert. *Building machine learning systems with Python*. Packt Publishing, 2015.

[10] Thomas B. Moeslund. *Introduction to video and image processing*. Springer, 2012.

[11] Umberto Michelucci. *Advanced applied Deep Learning*. Apress, 2019.

[12] Puyin Liu and Hong-Xing Li. *Fuzzy neural network theory and application*. World Scientific, 2014.

[13] 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.

[14] Steven F. Lott. *Mastering Object-oriented Python*. Packt Publishing, 2014.

[15] Deepak Vohra. *Kubernetes microservices with Docker*. Apress, 2016.

[16] Krish Krishnan. *Data warehousing in the age of big data*. Elsevier, 2013.

[17] Joe Celko. *Joe Celko's SQL for smarties: advanced SQL programming*. Morgan Kaufmann, 2005.

[18] Deepak Vohra. *Pro Docker*. Apress, 2016.

[19] Kunal Relan. *Building REST APIs with Flask: create Python web services with MySQL*. Apress, 2019.

[20] Charlie Catlett, Wolfgang Gentzsch, Gerhard R. Joubert Lucio Grandinetti, and Jose L. Vazquez-Poletti. *Cloud Computing and Big Data*. IOS Press, 2013.

Apéndice A

Introduction

A.1. Motivation and objectives

The Deep Learning [1] area has advanced exponentially in the last years. This has allowed to have predictive models capable of processing images and classifying them according to their primary features. The main consequence of this process is the opening of a window of opportunity for the exploitation of these models in a real environment, with the objective of making them crucial when detecting fires, earthquakes, as well as all kinds of natural disasters.

The efficient use of these models requires an infrastructure capable of supporting the necessary reliability in terms of availability and speed. In these cases, real-time processing becomes essential to optimize emergency resources, manage teams to the most affected disaster areas and prevent the maximum possible risks.

The points that resumes this project are: firstly, the acceleration of the training time of a Deep Learning model using a GPU [2] in the Google Colab service, and secondly, optimization of model inference time using the Intel OpenVINO toolkit. Finally, the model will be deployed in a cloud environment where it can operate as a service capable of supporting thousands of concurrent calls. TensorFlow [3], an open source framework developed by Google for the preparation of neural network training algorithms, has been used as the main tool to carry out the model training.

With the requirement for the application to be maintainable and flexible to possible changes, Docker[1] container technology has been used. Using this tool ensures that the versions of the operating system and external libraries are compatible with each other, additionally, the possibility of porting the application to different environments that take advantage of this container solution is left open. The achievement of general goals is carried out by following a series of specific objectives, which are listed below:

- Improvement in the training times of a model of Deep Learning using a GPU from the Google Colab service.
- Converting a TensorFlow model to an OpenVINO model to increase its inference speed.
- Preparation of a Google cloud architecture capable of supporting concurrent traffic in optimal times for the service.
- Coding of an application capable of making use of the different inference systems of TensorFlow and OpenVINO.
- Coding of a web application suitable to expose all services in a production environment.
- Encapsulation of the different environments using Docker.
- Application deployment and load tests.
- Obtaining results and performance comparing between the different inference systems, hardware and web servers.

A.2. State of the art

Currently, artificial intelligence is made up of several branches such as machine learning, natural language processing, among others. One of them is Deep Learning. The deep learning architecture pursues the study and classification of a variety of problems making use of their

[1] https://www.docker.com/¸

own algorithms. Currently, Deep Learning algorithms are used for all kinds of problems that cover a multitude of sectors within the industry [4], governments and, ultimately, society itself. The digitization and expansion of the internet provides countless data sources capable of being processed and analyzed by this kind of algorithms, which are used for different purposes.

The source of the data has changed, now they come from interactions that users have with their mobile devices, calls, internet money transactions [5], navigation of web pages and, in the case of this project, images from a satellite. Image processing has been an advance in society that police forces are now taking advantage of, using these tools to detect license plates or to recognize potential criminals[2]; doctors, who use these systems [6] to improve the early detection of some types of cancer[3], or in the case of industry, which is helped by these solutions to automate and classify processes that previously involved the supervision or execution of a person. In the same way, countries have their personal image recognition systems for classifying their citizens, recommendation systems both for companies seeking to increase their sales and for banks seeking loan-eligible individuals, and even to avoid undesirable content through the web.

In general, the massive amount of data has created a necessity for exploitation through them, the Deep Learning is positioned as a reliable tool to give value to all the interactions that are occurring almost permanently in every technological system on the planet. All this stimulus leads to the creation of thousands of new jobs in the technology sector, dedicated exclusively to the application of machine learning algorithms, as well as to the increase in their teaching. This has opened the possibility for professionals who previously did not have a clearly defined role in this field to position themselves as practically indispensable. The benefits are noticeable in studies like math, statistics, and related. In these areas, mathematical profile and analytical skills are highly valued for carrying out this type of

[2] https://www.nhtsa.gov/sites/nhtsa.dot.gov/files/documents/809811-lidarspeedmeasuringdevice.pdf

[3] https://www.nature.com/articles/srep46450

task.

The development of this new profile of professionals has encouraged the use of less common programming languages. This is because its use and learning curve is more affordable than other more traditional languages, in the case of Java or C++ (see Figure A.1). Along with this, new free and open source [7] tools have emerged, such as Jupyter, TensorFlow, Scikit-learn or PyTorch, among others.

A.2.1. Deep Learning concept

The Deep Learning has its defining element the use of algorithms that base its structure on artificial neural networks, imitating the behavior of human beings and their central nervous system. The strength provided by the emergence of Big Data has made this type of technology become the daily practice of many professionals. One of the keys to the Deep Learning algorithms is in the learning capacity that resides in them. This gives us the ability to deal with real world problems, where combinations of possibilities and pattern recognition are left out of our calculations. Neural networks represent the main tool for classifying images. This is because they can extract fundamental characteristics from each pixel and have a high percentage of accuracy in the prediction.

In order to materialize all these machine learning algorithms we have services from large companies such as Google, Amazon, IBM, which implement their own business solutions [20]. But we can also choose open source tools like TensorFlow, one of the most famous Deep Learning libraries developed by Google engineers, later released under Apache license. We also have others like PyTorch and Keras. All of the them were originally developed for the Python programming language, which has seen its usage percentage increase due to this machine learning [9] wave.

A.2.2. Neural networks in image processing

The basic processing unit for neural networks is the perceptron (see Figure A.2), which develops an algorithm capable of generating selection criteria for subsets of neurons. This

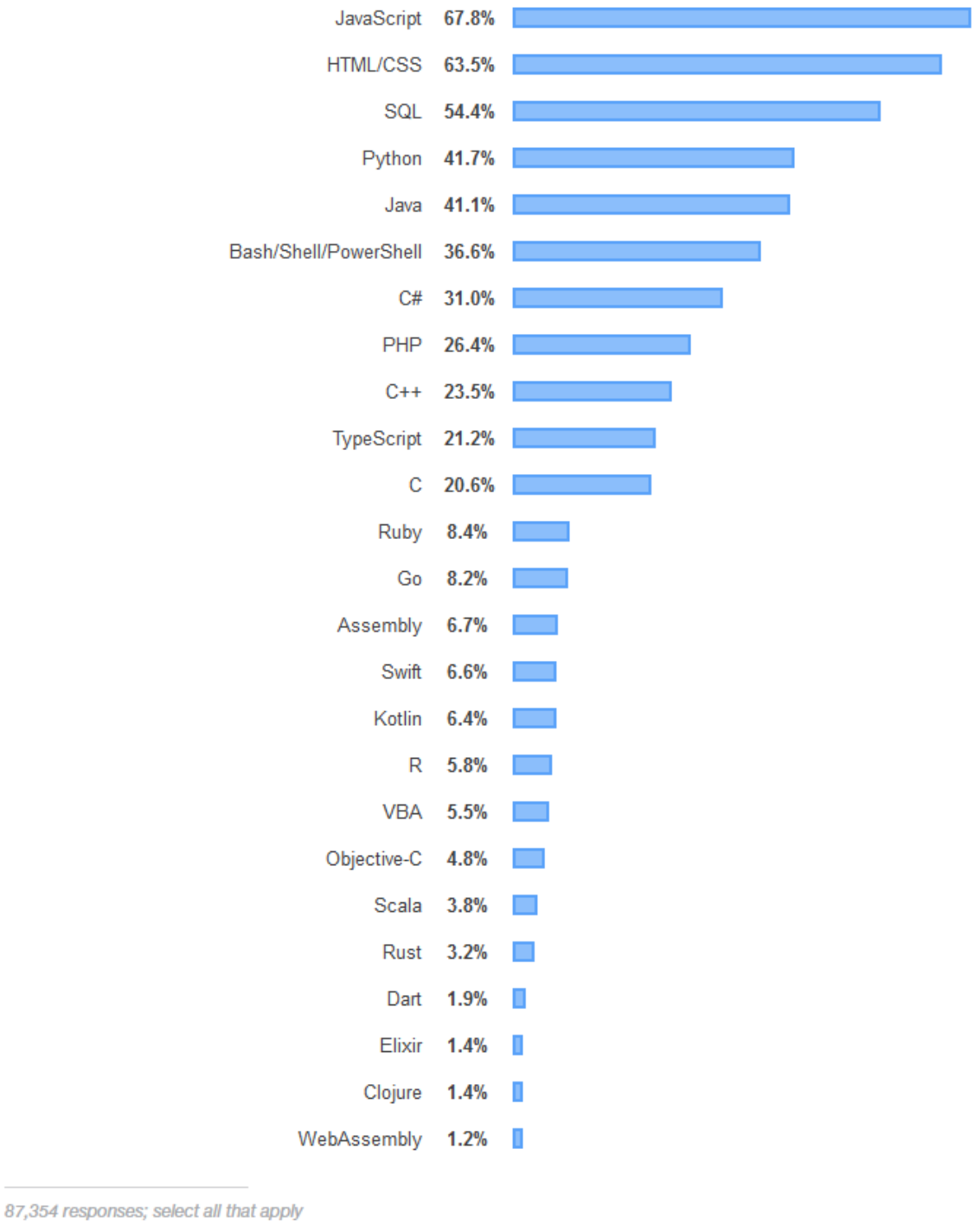


Figura A.1: *Survey of programming languages at StackOverflow 2019.*

set of neurons will become part of the different layers that completely compose the neural network. Each neuron receives an input, either from an external source or from another neuron.

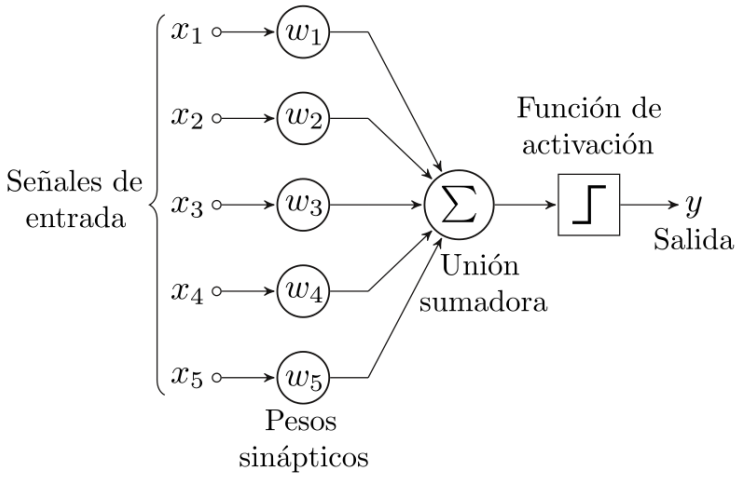


Figura A.2: *Perceptron example.*

Each neuron applies a calculation function which generates the corresponding weights of each neuron. These weights represent the level of interaction of the neurons and should be adjusted so that they are as close as possible to the data we know. The input weights of a layer originate from a previous layer, and its outputs are part of the input of a subsequent layer. Propagation occurs until we reach the last layer of the network, which will be the output layer from which we obtain the result of our classification.

In this specific problem, we focus on classifying multispectral images [10] with high spatial resolution using the RGB spectral bands (Red, Green and Blue). Our set of images belongs to an area partially destroyed by a natural disaster in Haiti, which occurred in 2010. These images were acquired by the GeoEye-1 high-resolution Earth observation satellite, launched in September 2008. Therefore, the purpose of our Deep Learning model is to have the ability to classify these images depending on whether the area is damaged or in good condition.

A.3. Work plan

In Figure A.3 we can see the plan for achieving the objectives followed in this work. The effort metric that appears in it refers to the number of days invested in each task, with a total sum of 10 weeks. The activities are sequentially ordered in their execution and have a dependency on each other, so it has never been moved on to another task without completing the previous one.

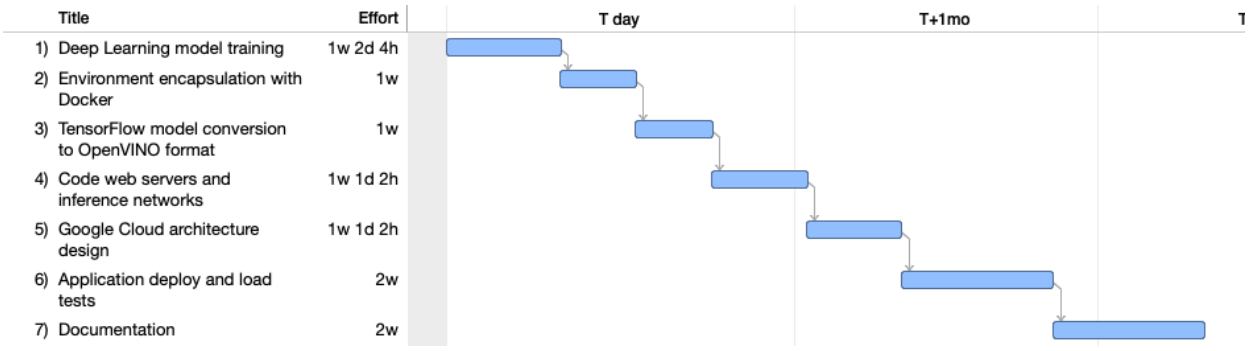


Figura A.3: *Work plan.*

A.4. Organization of this project

Bearing in mind the above specific objectives, we proceed to describe the organization of the rest of this project, structured in a series of chapters whose contents are described below:

- **Training the model using Google Colab**: The training and speed increase process using the Google Colab platform and its associated hardware.
- **OpenVINO technology**: The purpose of the Intel OpenVINO toolkit is defined, as well as the transformation of a TensorFlow model to be compatible with said solution.
- **Proposed Cloud Architecture**: Explanation of the Google Cloud architecture designed to support the entire application infrastructure.

- **Experimental results**: The different web frameworks to be tested using the Python programming language are prepared, showing the performance obtained in the training and inferences phases. In addition, the approximate calculation of the project costs will be presented.
- **Conclusions and future work**: The conclusions obtained through load tests and also some possible lines of future work.

Apéndice B

Conclusions and future work

B.1. Conclusions

The effects of the globalization of the internet are increasingly noticeable in our lives. The increase in the population that interacts through applications results in the generation of an infinite amount of data, which contains all the information of the users who use them. From web pages, mobile applications to systems that interact automatically, without the need for users to use or configure them. Until a few years ago, these fields barely escaped the academic and research sphere, while in these times they have become a real need for an increasingly interconnected world on a global scale.

As data grows on the network, the need for its analysis and exploitation increases. For this reason, platforms such as Google Cloud, Amazon Web Services carry out this process, providing all the self-managed computing power to users. Daily new analysis technologies specific to these data are born. This is the case of TensorFlow or OpenVINO, owned by commercial companies such as Google or Intel. The first of them was converted to open source format, while the second was placed as free since its launch; this gives due account of the growing community that makes use of these tools. These types of free resources, such as Google Colab, provide the community with solutions that used to be paid and accessible only to a certain audience within the technology sector. And, at the same time, it is the community which reports errors, proposes updates or directly codifies new solutions

to incorporate.

In this work, it has been possible to train a *Deep Learning* model with a 93 % accuracy on the training dataset. Furthermore, said model has been optimized to improve its inference using the Intel OpenVINO tool. The model has been deployed in the Google Cloud environment using an architecture that configures and orchestrates the entire process pipeline. All deployed applications are encapsulated within a Docker container. The production environment is prepared, on the one hand, to deploy the OpenVINO and TensorFlow inference network and, finally, to ingest their rankings in real time.

The inference time for both networks in the production environment is less than the second, being 67 ms in TensorFlow and 2 ms in OpenVINO. The total time per HTTP request is 212 ms for TensorFlow and 126 ms for OpenVINO, with the configuration options for these results supporting the most concurrent processing. The classification success rate in the productive environment is 97 %.

It is the competitiveness among all these tools that drives constant improvement in all of them. TensorFlow was born as a framework focused on the generation of Deep Learning models, but not with the main objective of accelerating model inference, as OpenVINO does. The symbiosis between them creates a complete combination that covers the needs of both creation and the use of deep learning models.

B.2. Future work

The platform has been configured so that its request processing capacity is scalable, that is, because the use of Docker container technology is prepared for integration into the Kubernetes[1]. This work has used Google Cloud as an application deployment platform, but exploring other solutions such as AWS or Azure would give a more general perspective of which environment is better prepared to build and use Deep Learning models.

The visualization of results in this work has been done through SQL jobs[2] in a distributed

[1] https://clustersolution.kubernetes.io/es/docs/concepts/overview/what-is-Kubernetes/
[2] https://cloud.google.com/bigquery/docs/jobs-overview

database, but it would be ideal to be able to look at the data in a more intuitive way. Docker allows us to easily port this container solution to other systems with different hardware, and consequently, it can work locally without having a web or cloud platform that supports it. The classification could occur within the element that generates the images, which would eliminate the latency time of other additional elements.

There is the possibility of introducing a new automatic process to train the model, as long as the new samples do not affect the current configuration. The data in this work are RGB images of a part of the planet, so they have exact latitude and longitude. The Heatmap[3] visualization technique is ideal for seeing in real time the areas of the terrain most affected by the natural disaster and being able to redirect the emergency teams quickly. The configuration of the servers ingests the calls internally, so it is not open to public requests directly. If it were decided to open the classification of images to the public, all the necessary certificates would have to be configured on the servers to work with the secure HTTPS protocol. In addition, certain security measures against factors such as denial of service or attempted unwanted connection to servers through the application's open ports, in our case 8080 and 22, which we use to connect via SSH[4].

In our problem, an aerial vehicle could carry the classification hardware and fly over the damaged terrain to classify the images it is collecting. This system would be similar to options that can already be found in the market, such as the AWS DeepLens[5] from Amazon.

Beyond business logic, it would be desirable for such solutions to be extended in an open source format. This format would allow the improvement of the project through the community, which would be free to make contributions.

[3] https://en.wikipedia.org/wiki/Heat_map
[4] https://www.ssh.com/ssh/
[5] https://aws.amazon.com/es/deeplens/