Inference time optimizacion 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

Agradecimientos

Índice general

Índice general I

Índice de figuras IV

Índice de tablas VI

Resumen VIII

Abstract X

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 7

2. Entrenamiento del modelo mediante Google Colab 9

2.1. Modelo propuesto 10

2.2. Entorno Google Colab 13

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 **27**

4.1. Descripción del entorno y sus diferentes componentes 27

4.1.1. Google Storage 28

4.1.2. Google BigQuery 29

4.1.3. Google Pub/Sub 29

4.1.4. Google Compute Engine 30

4.1.5. Google Cloud Function 30

4.1.6. Container Registry 31

4.2. Explicación del flujo de datos 31

4.3. Codificación de los servidores web 32

4.3.1. Framework FastApi 35

4.3.2. Framework Flask 36

4.4. Encapsulación de entorno con Docker 37

5. Resultados experimentales **41**

5.1. Dataset y Hardware 41

5.2. Rendimiento en fase de entrenamiento 42

5.3. Rendimiento en fase de inferencias 45

5.4. Costes del proyecto 47

6. Conclusiones y trabajo futuro **50**

6.1. Conclusiones 50

6.2. Trabajo futuro 51

Bibliografía **52**

A. Introduction **53**

A.1. Motivation 53

A.2. Objectives 54
A.3. Organization of this memory 55

B. Conclusions and future work **58**
B.1. Conclusions 58
B.2. Lines of future work 59

Índice de figuras

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

2.1. Diferencia de núcleos para procesar cargas de trabajo en paralelo de forma eficiente entre CPU y GPU. 10
2.2. Función de activación Relu. 12
2.3. 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. 23

4.1. Arquitectura Cloud 28
4.2. Arquitectura Google Cloud 31
4.3. Petición HTTP al servidor 32
4.4. Arquitectura de la máquina virtual de OpenVINO. 37
4.5. Arquitectura de la máquina virtual de TensorFlow. 38

5.1. Resultados de entrenamiento del modelo usando una GPU TeslaK80 con un batch-size de 256 y 100 epochs (a) Rendimiento del modelo en acierto. (b) Rendimiento del modelo en pérdida 43
5.2. Resultados de entrenamiento del modelo usando una GPU TeslaK80 con un batch-size de 256 y 175 epochs (a) Rendimiento del modelo en acierto. (b) Rendimiento del modelo en pérdida 44

5.3. Resultados de entrenamiento del modelo usando una GPU TeslaK80 con un batch-size de 256 y 100 epochs (a) Rendimiento del modelo en acierto. (b) Rendimiento del modelo en pérdida 44

Índice de tablas

5.1. Comparativa de tiempo de inferencia con OpenVINO y TensorFlow. 45
5.2. Comparativa de tiempo de inferencia con OpenVINO y TensorFlow con distinto hardware. 46
5.3. Comparativa de tiempo de inferencia con OpenVINO y TensorFlow con distinto framework web. 47
5.4. Comparativa de tiempo de inferencia con OpenVINO y TensorFlow con distinto hardware y servidor web. 47

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 o prevenir y hacer un seguimiento de 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 nos abre la posibilidad de construir modelos capaces de reconocer zonas en las que ha ocurrido un desastre natural y poder actuar en consecuencia. Con la potencia de cálculo actual podemos conseguir que el procesamiento de los datos sea en tiempo real , por lo que se pueden tomar decisiones para poder minimizar daños, distribuir equipos de emergencia y optimizar en general todos los recursos que tenemos.

Estos últimos años el campo de la ciencia de datos ha sido capaz de crear modelos precisos en sectores como banca, industria, tecnología. Pero el círculo solo puede estar completo si podemos conseguir colocar estos modelos en un entorno en el que se puedan consumir de manera productiva y ser útiles fuera del escenario de desarrollo e investigación.

Mantener toda la infraestructura hardware y software para ejecutar nuestra aplicación es costoso, de la misma manera que contratar a las personas con los conocimientos necesarios para instalar y configurar todos estos componentes. El mundo ha evolucionado de manera que ya es no necesario seguir este comportamiento y podemos optar por proveedores que facilitan todos estos componentes así como el mantenimiento de los mismos, la nube.

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 OpenVINO a la par que se realiza la puesta a 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 hiperespectrales, OpenVINO, TensorFlow, Docker, Google Cloud

Abstract

The remote observation of the Earth has always been an object of interest for the human being. Over the years, the methods used for this purpose have evolved until, at present, the analysis of hyperspectral images constitutes a very active line of research, especially to monitor fires or prevent and monitoring natural disasters, chemical discharges or other types of environmental pollution.

Satellite images in a world where machine learning and data processing have advanced so much opens up the possibility of building models capable of recognizing areas where a natural disaster has occurred and be able to act accordingly.

With the current computing power we can make data processing in real time so we can made decision to minimize damage, distribute emergency teams and optimize all the resources we have. In recent years the field of data science has been able to create accurate models in sectors such as banking, industry, technology. But the circle can only be complete if we can manage to place these models in an environment where they can be consumed productively and be useful outside the development and research scenario.

Maintaining all the hardware and software infrastructure to run our application is expensive, in the same way that hiring people with the necessary knowledge to install and configure all these components. The world has evolved so that it is no longer necessary to follow this behavior and we can choose suppliers that facilitate all these components as well as their maintenance, the cloud.

In this final degree project, the optimization in inference times of a machine learning model used to detect natural disasters with OpenVINO is carried out at the same time that the deploy of the model in a Google cloud environment, so that our service supports miles of requests per minute.

Keywords

Hyperspectral images, OpenVINO, TensorFlow, Docker, Google Cloud

Capítulo 1

Introducción

1.1. Motivación y objetivos

El área del Deep Learning[**?**] 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[**?**] 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 llamadas concurrentes. Para llevar a cabo el entrenamiento del modelo se ha uti-

lizado como herramienta principal TensorFlow[?], un framework open source desarrollado por Google para la preparación de algoritmos de entrenamiento de redes neuronales. Este framework será totalmente codificado en el lenguaje de programación Python. Con la necesidad de que la aplicación sea robusta y flexible ante cambios se ha usado la tecnología de contendores Docker[1]. Empleando esta tecnología 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

La inteligencia artificial actualmente 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. En la actualidad, los algoritmos de Deep Learning son usados para todo tipo de problemas que abarcan multitud de sectores dentro de la industria, 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 interacciones que tienen los usuarios con sus dispostivos móviles, llamadas transacciones de dinero por internet, 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 policías, usando estas para detección de matrículas o para reconocer posibles delincuente. Médicos, que utilizan estos sistemas para mejorar la detección prematura de algunos tipos de cáncer. Industria, que se ayuda esta de estas soluciones para automatizar y clasificar procesos que antes suponían la supervisión o ejecución de una persona. 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 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 un hueco claro en el sector tecnológico a ser indispensables dentro de él.

Estudios como matemáticas, estadística y relacionados, se ven beneficiados ya que la capacidad de análisis y el perfil matemático que poseen son aptitudes muy valorables para realizar este tipo de tareas. Algunos lenguajes de programación menos usuales han visto impulsado su uso a raíz de este nuevo perfil de profesionales, debido a que su uso y curva de aprendizaje es mucho más sencillo que lenguajes más tradicionales como Java o C++ (ver Figura 1.1). También ha fomentado la creación de nuevas herramientas de desarrollo, como Jupyter, TensorFlow, Scikit-learn, PyTorch, todas ellas gratuitas y de código abierto.

1.2.1. Concepto Deep Learning

El Deep Learning lleva consigo como principal actuador 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 diaría de muchos trabajadores. Una de las claves de los algoritmos de Deep Learning es 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.

Para poder materializar todos estos algoritmos de aprendizaje automático disponemos de servicios de grandes empresas como Google, Amazon, IBM, los cuales implementan sus propias soluciones comerciales. 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 en primera instancia y 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

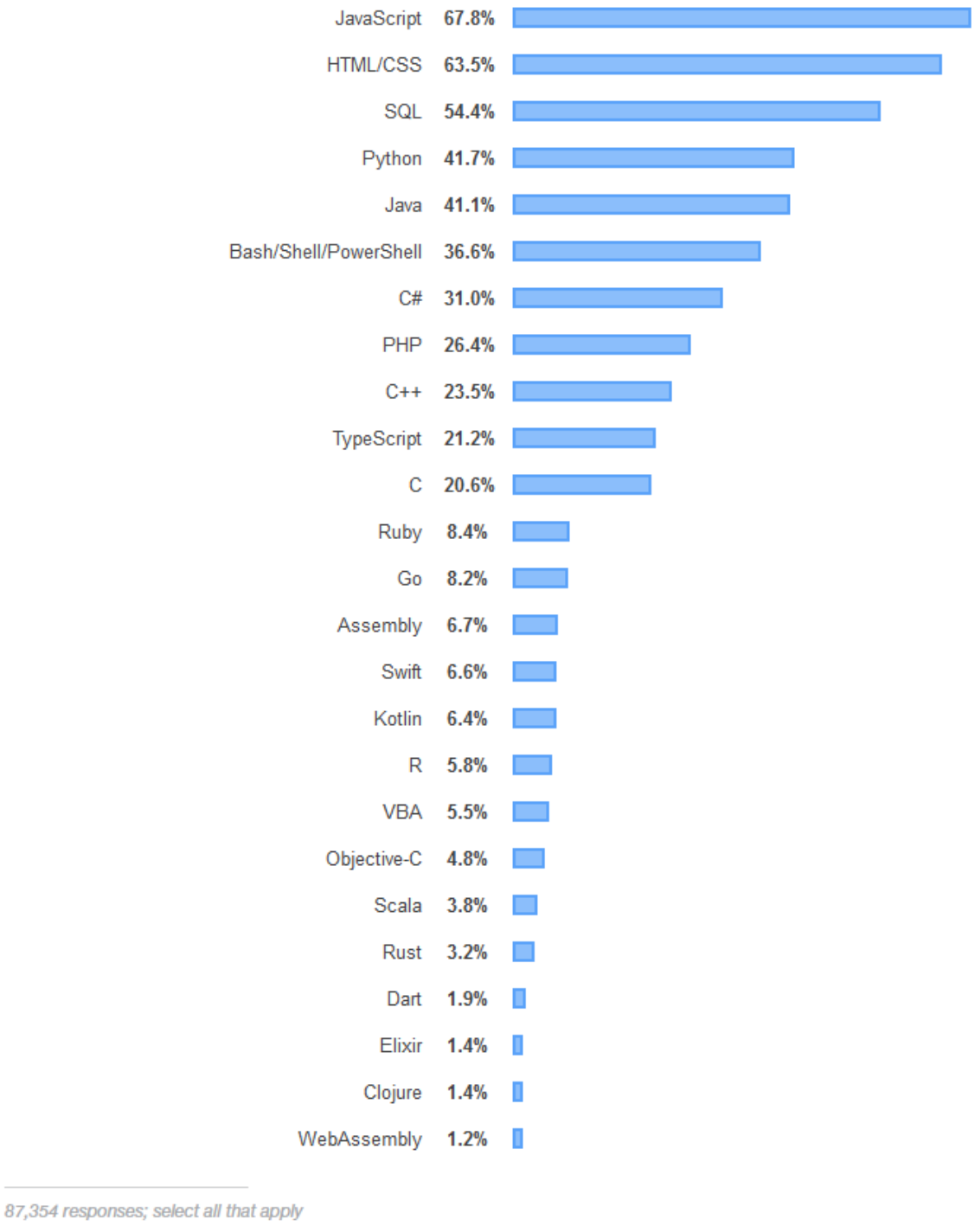


Figura 1.1: *Encuesta sobre lenguajes de programación usados en StackOverflow 2019.*

cual ha visto aumentado su porcentaje de uso debido a esta corriente de machine learning.

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 cual 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.

A partir de aquí 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.

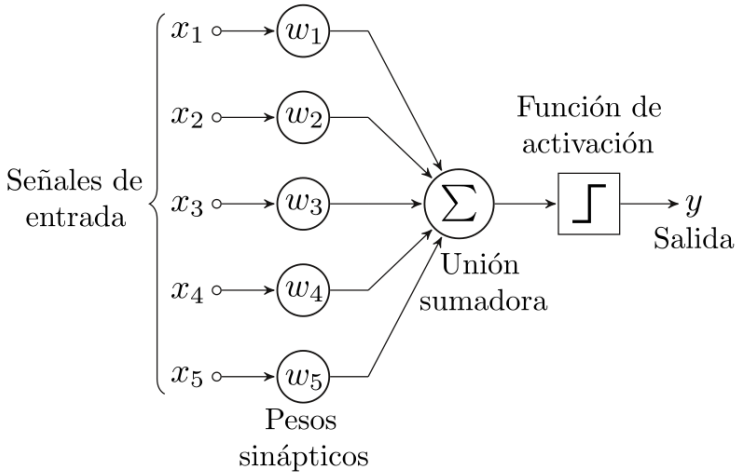


Figura 1.2: *Ejemplo de perceptrón.*

En este problema concreto nos centramos en clasificar imágenes multiespectrales con alta resolución espacial 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

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 del mismo 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[**?**], 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 mismo con distintos parámetros. Con ello, se puede reajustar constantemente el modelo 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 programada en el lenguaje de programación Python. Éste incluye una API 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[**?**] 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

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

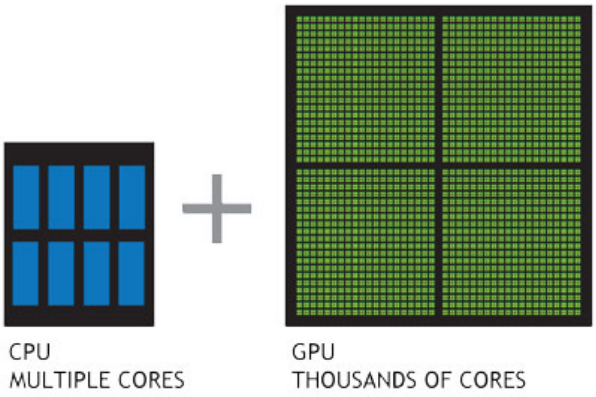


Figura 2.1: *Diferencia de núcleos para procesar cargas de trabajo en paralelo de forma eficiente entre CPU y GPU.*

número de núcleos o cores (ver Figura 2.1), gracias a su conectividad por PCI express y el ancho de banda que esta proporciona.

2.1. Modelo propuesto

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

Como framework principal para realizar el entrenamiento nos ayudaremos de TensorFlow, que incluye la librería de Deep Learning Keras, la cual 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. 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 partes de la misma. El térmido 2D se refiere al movimiento del filtro, el cuál es 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. Los filtros de mayor tamaño ayudarán al modelo a mejorar su aprendizaje.

- **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 dependiendo de sus valores y ayudar al modelo a tener en cuenta funciones no lineales, lo que 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 2.2, 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, 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, su objetivo es reducir la muestra de una representación de entrada mediante el acortamiento de sus dimensiones. En nuestro modelo aplicaremos una reducción a

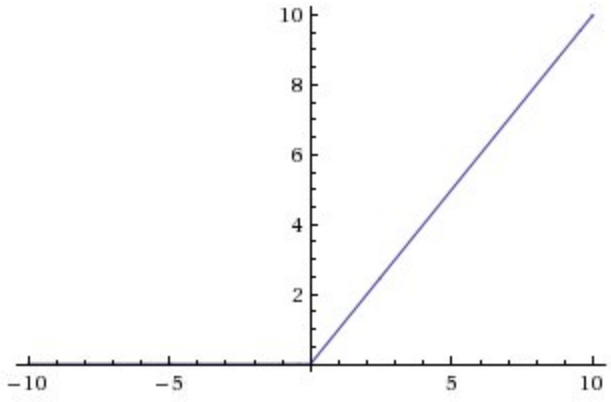


Figura 2.2: *Función de activación Relu.*

matrices de 2×2.

- **Dropout**: es una capa cuyo cometido es ignorar ciertas neuronas de forma aleatoria para no incluirlas en el entrenamiento, por lo que 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, este aprovecha el poder de los métodos de tasas de aprendizaje adaptativo para encontrar nivel de aprendizaje individuales para cada parámetro. Este optimizador posee un hyperparámetro llamado *learning rate* que regula la rapidez con la que el modelo avanza hacia el valor optimo de sus pesos, un *learning rate* bajo implicaría que los pesos evolucionan lentamente durante el entrenamiento, con lo cual puede tardarse mucho en llegar al valor optimo, mientras que con un learning rate alto avanzamos mas rapido pero podemos sobrepasar este punto óptimo. El valor de este

hyperparámetro en este trabajo se mantiene a 0.0008.

Para un mejor entendimiento, en la Figura 2.3 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) optimiza 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 el ancho de banda usado en memoria.

- **Valores altos del parámetro de entrenamiento Batch Size**: gracias a la capacidad de cómputo de nuestra GPU podemos permitirnos el uso de valores altos en este parámetro de entrenamiento.

2.2. Entorno Google Colab

La plataforma de Google Colab[3] es un servicio gratuito de Google, mediante el cual podemos ejecutar e instalar librerías del lenguaje de programación Python[**?**]. 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 y podemos compartir nuestro trabajo con otras personas. Estas características permiten

[2] https://www.tensorflow.org/xla
[3] https://colab.research.Google.com/notebooks/intro.ipynb

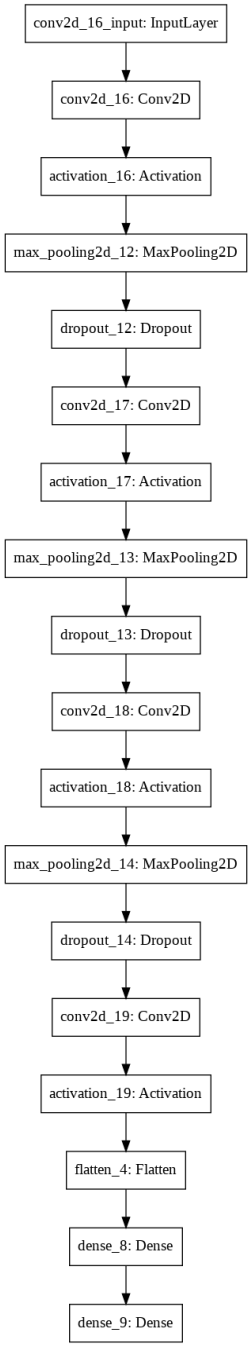


Figura 2.3: *Topología de la red del modelo Deep Learning usado para la clasificación de daños.*

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 primarias 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 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. Una de las principales ventajas que tiene poder usar un entorno como Google Colab es que el servicio se ejecuta de manera íntegra online, de modo que toda la carga computacional reside en la herramienta de Google y no en nuestro computador. Esto permite trabajar de manera fluida realizando otro tipo de cometidos en nuestra máquina, o simplemente ejecutar un proceso para el que no tenemos suficiente potencia disponible.

[4] https://www.nvidia.com/es-es/data-center/tesla-k80/

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, peternece al conjunto de aplicaciones de código abierto, de modo que se puede visualizar su código fuente, repotar fallos e incluso realizar aportaciones. Podemos visualizar el diseño y el código de la aplicación en su repositorio oficial de GitHub[1]. 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 con los mismos de manera concurrente, aprovechando así toda la potencia de los procesadores o GPU[**?**] actuales.

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 a uno preparado 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.

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

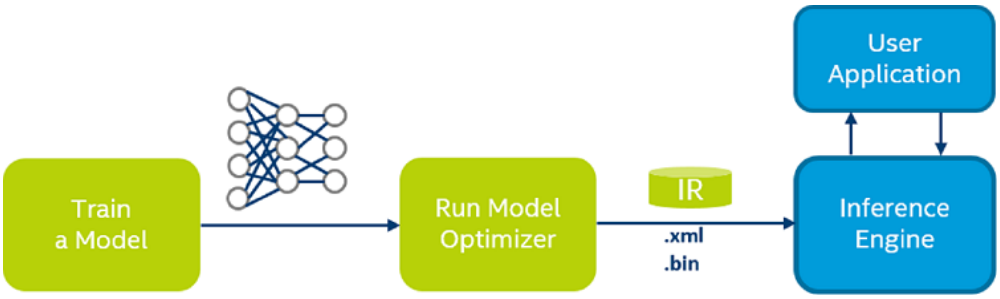


Figura 3.1: *Arquitectura de optimización de modelos con OpenVINO.*

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, mediante las distintas herramientas que lo componen. 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 recibe como argumento 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. La herramienta de OpenVINO 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 a los que se puede modifcar su código, 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, optimizado 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. Entre otras cosas, 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 del modelo, 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 independiente:

- Un punto de control TensorFlow que contiene los pesos del modelo.
- Un prototipo 'SavedModel' que contiene el gráfico subyacente de TensorFlow. Este prototipo separa los gráficos que se guardan para predicción (servicio), capacitación y evaluación.
- La configuración de la arquitectura del modelo, si está disponible.

Los métodos exactos para la serialización del modelo varían según la versión de TensorFlow, en el caso presente versión 1.15.2. En cualquier caso ,la metodología de conversión exacta utilizada para este proyecto se puede encontrar en el repositorio de código fuente de este trabajo 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 para 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 realizar la transformación de estos 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 código binario, por lo que es texto plano. Esta opción es totalmente

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

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 tamaño del batch que queremos especificar para la conversión, seleccionamos 1 porque en las entradas de nuestra 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, el cual incorpora un servidor codificado mediante el patrón diseño api rest, 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 un clúster de contenedores como es Kubernetes[3], por lo que el servidor puede ser encapsulado en su totalidad en un contenedor de Docker.

En este trabajo se ha trabajado con una versión encapsulada de docker, pero no con la extensibilidad del escalado con Kubernetes[**?**]. La unidad de cálculo principal del sistema de inferncia también es configurable, perimitiendo así el uso tanto de CPU como de GPU. Si se usa la opción de contenedores docker configurando una GPU es necesario configurar de manera explícita este 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

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

para realizar las inferencias y configurar el servidor, por lo que la 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 y los métodos necesarios para realizar la petición de inferencia se codifican de la siguiente manera 3.2.

Listing 3.2: *Clase 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]:
        start = time.time()
        image = self.shape_image(file_route)
        predict = self.network_request(image)
        return predict, (time.time() - start)

    def network_request(self, image) -> bool:
        headers = {"content-type": "application/json"}
        data = json.dumps({"signature_name": "serving_default", "instances": image})
        res = requests.post(self.model_uri, data=data,
                            headers=headers)
        predictions = json.loads(res.text)['predictions']
        predict = True if predictions[0][0] >= 0.5 else False
        return predict

    @staticmethod
    def init_tensorflow_serve():
        os.system('tensorflow_model_server '
                  '--rest_api_port=8501 --model_name=model '
                  '--model_base_path=/app/model &')
```

En la inicialización de la clase 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 servidor principal.

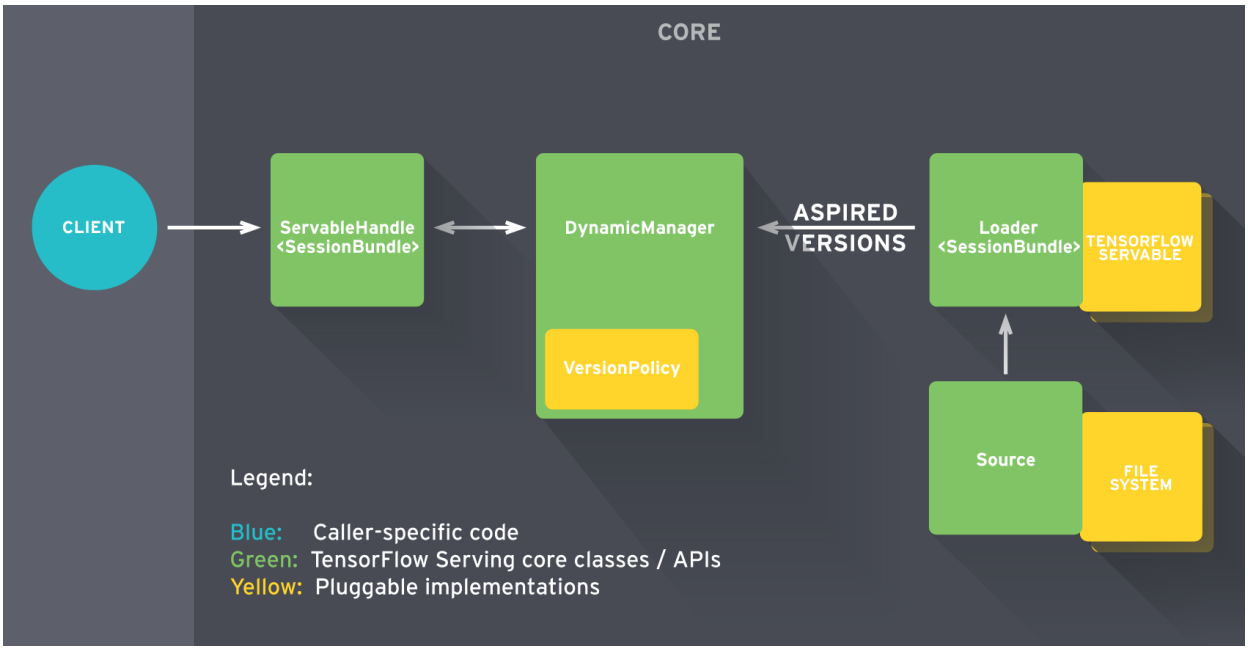


Figura 3.2: *Arquitectura de TensorFlow serving.*

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, por lo que el diseño software de este servicio y el procesamiento de las imágenes se codifica de la siguiente manera 3.3 en el repositorio del trabajo.

Listing 3.3: *Clase 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))
```

```
        self.out_blob = next(iter(self.net.outputs))
        self.net.batch_size = 1
        self.image_shape = 128

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

    def shape_image(self, file_route):
        image = cv2.imread(file_route, cv2.IMREAD_GRAYSCALE)
        image = cv2.resize(image, (self.image_shape, self.image_shape))
        image = image.reshape(self.image_shape, self.image_shape) / 255.0
        return image

    def network_request(self, image) -> bool:
        res = self.exec_net.infer(inputs={self.input_blob: image})
        res = res[self.out_blob]
        res = False if res < 0.5 else True
        return res
```

En la inicialización de la clase 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 inferencia. Se configuran métodos específicos para transformar la imagen a las dimensiones correspondientes y para hacer la petición a la red de inferencia. 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[4][5], por lo que el uso de Docker es la mejor opción para transportar nuestras redes de inferencia a cuualquier sistema o dispositivo hardware. De manera adicional y como paso natural, las dos herramientas se han diseñado de manera que puedan ser incluidas en un clúster de contenedores[6][7]. Como punto distintivo, OpenVINO implementa de soluciones

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

de fábrica para FPGA e Intel Movidius, como consecuencia, 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.

Capítulo 4

Arquitectura Cloud propuesta

La implementación y puesta en marcha de la aplicación se materaliza 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 muchos casos 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.

4.1. Descripción del entorno y sus diferentes componentes

Google Cloud posee multitud de herramientas específicas para cada situación, en este trabajo nos hemos centrado en las siguientes.

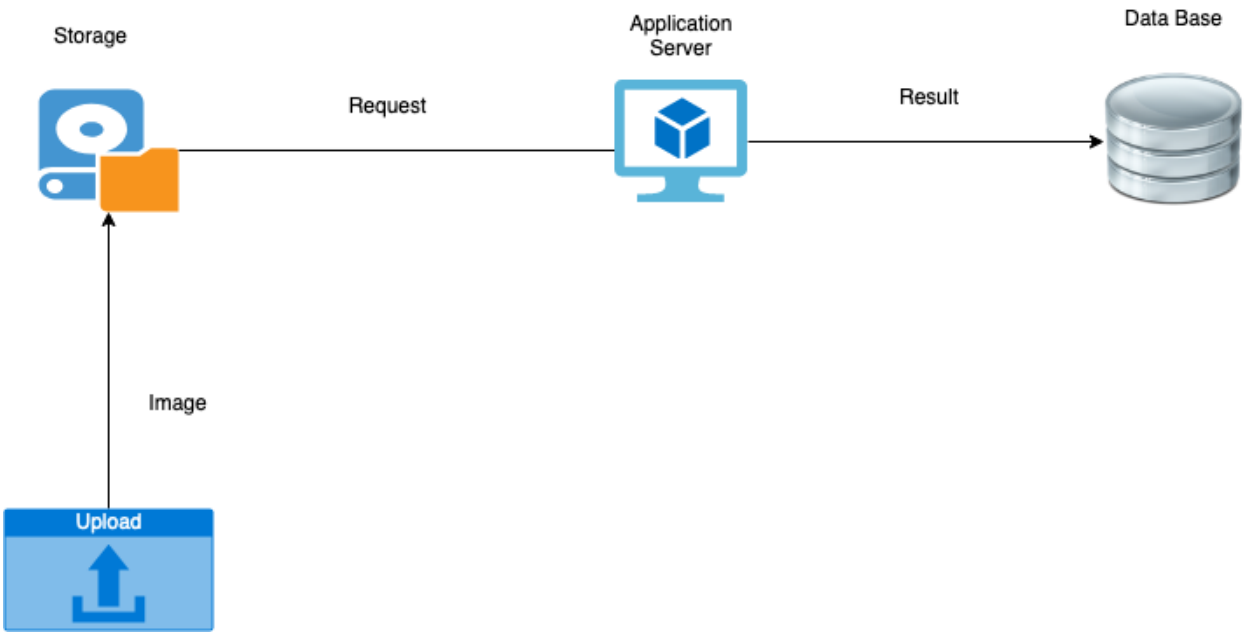


Figura 4.1: *Arquitectura Cloud*

4.1.1. Google Storage

Es el sistema de almacenamiento de Google[**?**]. Su objetivo dentro de la aplicación es poder ser utilizada como sistema de copia de seguridad de todas las imágenes que se vayan procesando, así como también 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. 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 ubicaciones o múltiples 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 ha sido utilizada para los benchmark 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 conjuntos de datos 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[**?**] estándar como lenguaje de consulta.

4.1.3. Google Pub/Sub

Es un sistema de colas de mensajería diseñada para eventos, está 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. Sus características se pueden enumerar en:

- Se pueden configurar distintas colas de mensajes, separando así de manera lógica los eventos de la aplicación.
- Este tipo de eventos está pensado para ser consumido en tiempo real, con la mínima latencia posible.
- Soporte para la ingesta de datos en tiempo real.

4.1.4. Google Compute Engine

Es el servicio principal de Google para proporcionar máquinas virtuales totalmente configurables, tanto en su capacidad de cálculo seleccionando el tipo de procesador, GPU y memoria RAM que se necesite, como en el entorno en el que se va a ejecutar la aplicación, ya sea docker[**?**] o las distintas distribuciones de sistemas operativos. Este servidor usará como aplicación principal una imagen de docker almacenada en el registro de contenedores. 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. Se configura este servidor con una versión Debian 9[**?**] como sistema operativo.

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.

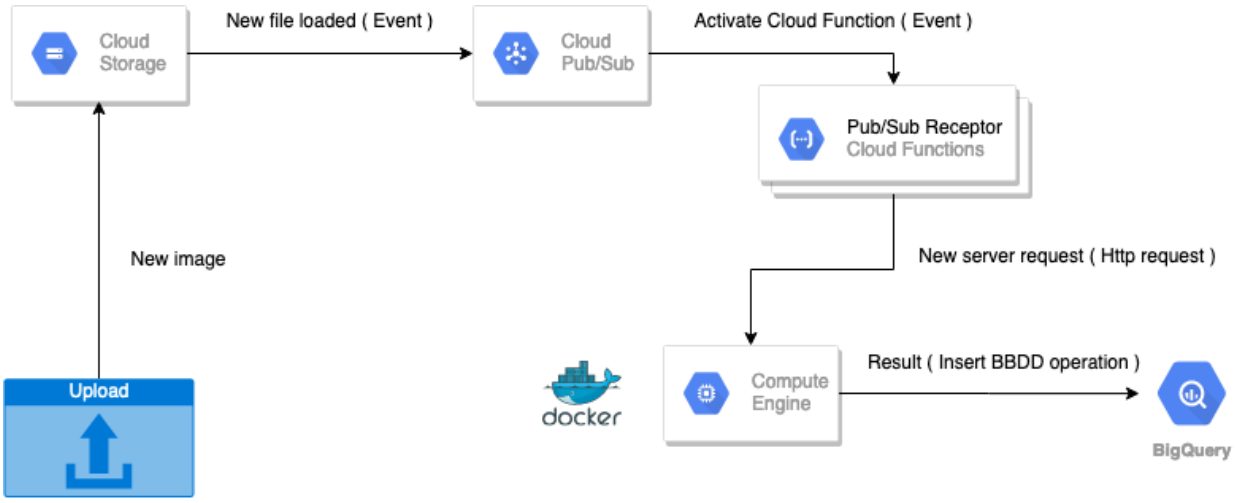


Figura 4.2: *Arquitectura Google Cloud*

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 lógicamente para optimizar el tamaño de la imagen origen que tiene cada una de las aplicaciones.

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. Una imagen que es cargada en el sistema de almacenamiento 4.1.1 dispara automáticamente 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 se

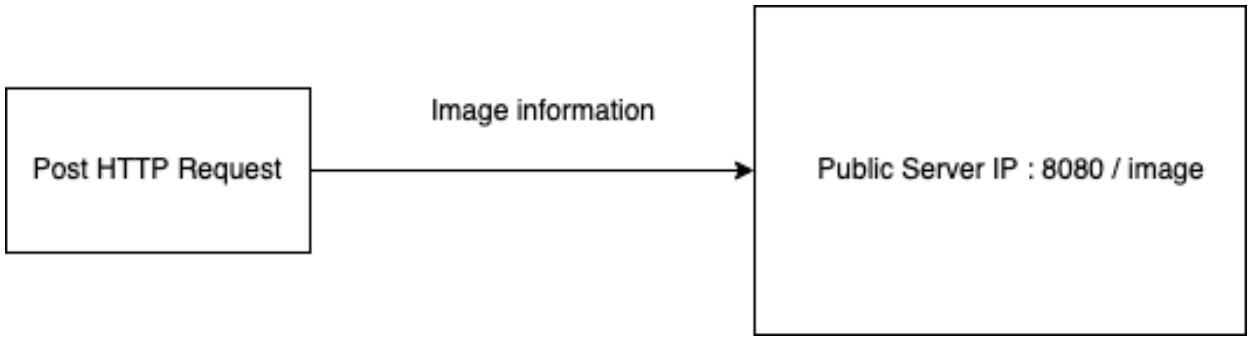


Figura 4.3: *Petición HTTP al servidor*

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 de la informática y el desarrollo software, lo que hace que el nivel de información sobre desarrollo con 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 a los distintos 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.

Para el procesamiento generalizado de imágenes y el sistema de recogida de metadatos

se han codificado dos clases principales. La primera 4.1 es la encargada 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. 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 apliación.
- 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.

Listing 4.1: *Clase Python para la API de la aplicación. label*

```
class Api:
def __init__(self, net, sys: SystemTrack):
    self.__storage = GoogleStorage()
    self.__big_query = GoogleBigQuery()
    self.__net = net
    self.sys_track = sys

def cloud_storage_request(self, item: dict):
    start = time.time()
    image_name = item['name']
    size = item['size']
    file_type = item['contentType']
    time_created = item['timeCreated']
    image_path = f'{data_dir}{image_name}'
    self.__storage.download_blob(bucket, image_name, image_path)
    prediction, inference_time = self.__net.process_image(image_path)
    total_time = time.time() - start
    row = [
        (
            image_name, size, file_type,
            time_created, prediction, inference_time,
            total_time, self.sys_track.physical_cores,
            self.sys_track.total_cores, self.sys_track.system, self.sys_track.processor,
            self.sys_track.system_memory,
            self.sys_track.system_memory_available,
            self.sys_track.so_version, self.sys_track.so_release, self.sys_track.inference_engine,
            self.sys_track.web_engine, self.sys_track.processor_unit,
            self.sys_track.docker, self.sys_track.cloud
        )
    ]
    self.__big_query.insert_row(row)
    os.remove(image_path)
```

La segunda clase 4.2 es la encargada 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[1] y platform[2] para conseguir la información necesaria del sistema. Contamos de manera adicional con un

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

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, la cual 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)
8          self.total_cores = psutil.cpu_count(logical=True)
9          self.system = self.sys_information.system
10         self.processor = self.sys_information.processor
11         self.system_memory = self.__get_size(self.sys_memory.total)
12         self.system_memory_available = self.__get_size(self.sys_memory.available)
13         self.so_version = self.sys_information.version
14         self.so_release = self.sys_information.release
15
16         self.docker = docker
17         self.inference_engine = inference_engine
18         self.web_engine = web_engine
19         self.cloud = cloud
20         self.processor_unit = processor_unit
21
22     @staticmethod
23     def __get_size(num_bytes, suffix="B"):
24         factor = 1024
25         for unit in ['', 'K', 'M', 'G', 'T', 'P']:
26             if num_bytes < factor:
27                 return f'{num_bytes:.2f}{unit}{suffix}'
28             num_bytes /= factor
```

4.3.1. Framework FastApi

FastApi[3] 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.

[3]https://fastapi.tiangolo.com/

- Documentación detallada para componente.
- Incorpora un sistema de tipado de objetos para su fácil identificación.
- Estándar OpenApi[4] para la especificación del formato de las peticiones entrantes y las respuestas del servidor.

Para soportar este framework se usará Uvicorn[5] como servidor ASGI[6] (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. El software es de código abierto y podemos encontrar su código fuente en su repositorio oficial.

4.3.2. Framework Flask

Flask[7] es el framework ligero por excelencia de Python, está desarrollado de manera sencilla y ligera, diferenciándose así de otros frameworks pesados como Djando, que incluyen muchas dependencias y acaban ocupando mucho espacio en disco y en memoria.

A diferencia de fastapi, Flask[**?**] 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[8] que sigue el estándar WSGI[9] (Web Server Gateway Interface), que es una convención simple para gestionar llamadas síncronas al servidor. Con este servidor podremos seleccionar la paralelización y distribución de carga del procesador.

[4] https://github.com/OAI/OpenAPI-Specification
[5] https://www.uvicorn.org/
[6] https://asgi.readthedocs.io/en/latest/
[7] https://flask.palletsprojects.com/en/1.1.x/
[8] https://gunicorn.org/
[9] https://en.wikipedia.org/wiki/Web_Server_Gateway_Interface

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, ambos se desplegarán en las máquinas virtuales de Google.

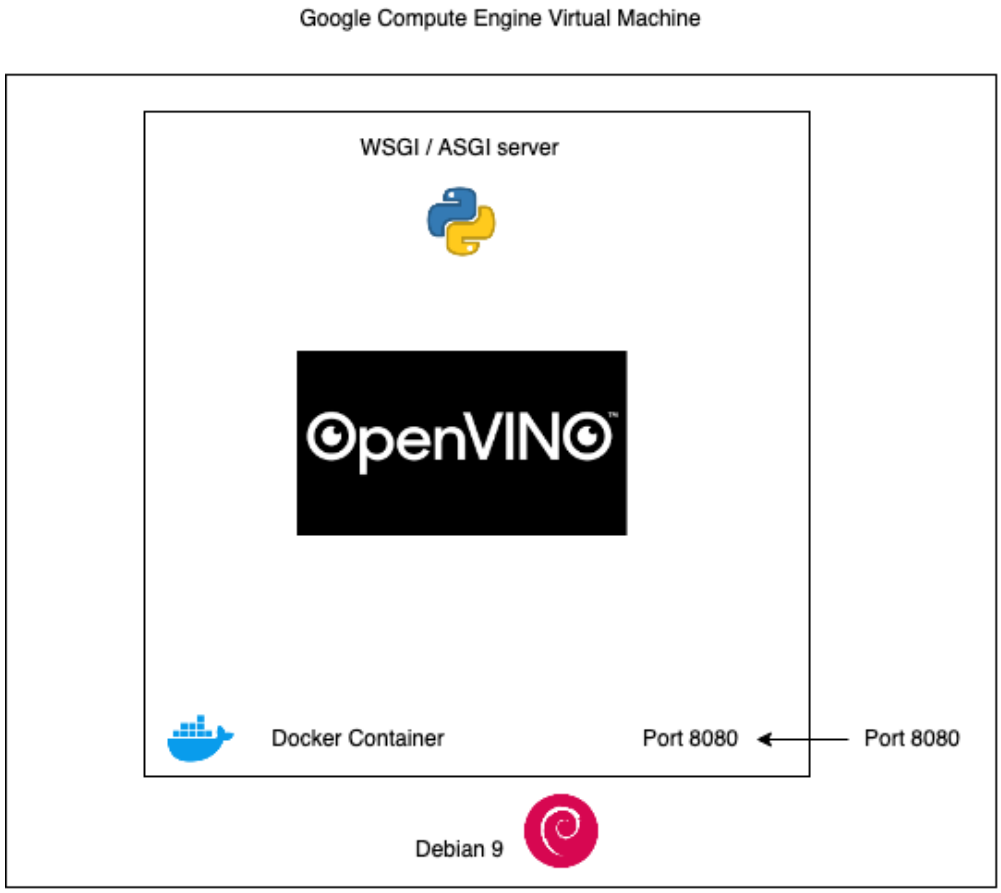


Figura 4.4: *Arquitectura de la máquina virtual de OpenVINO.*

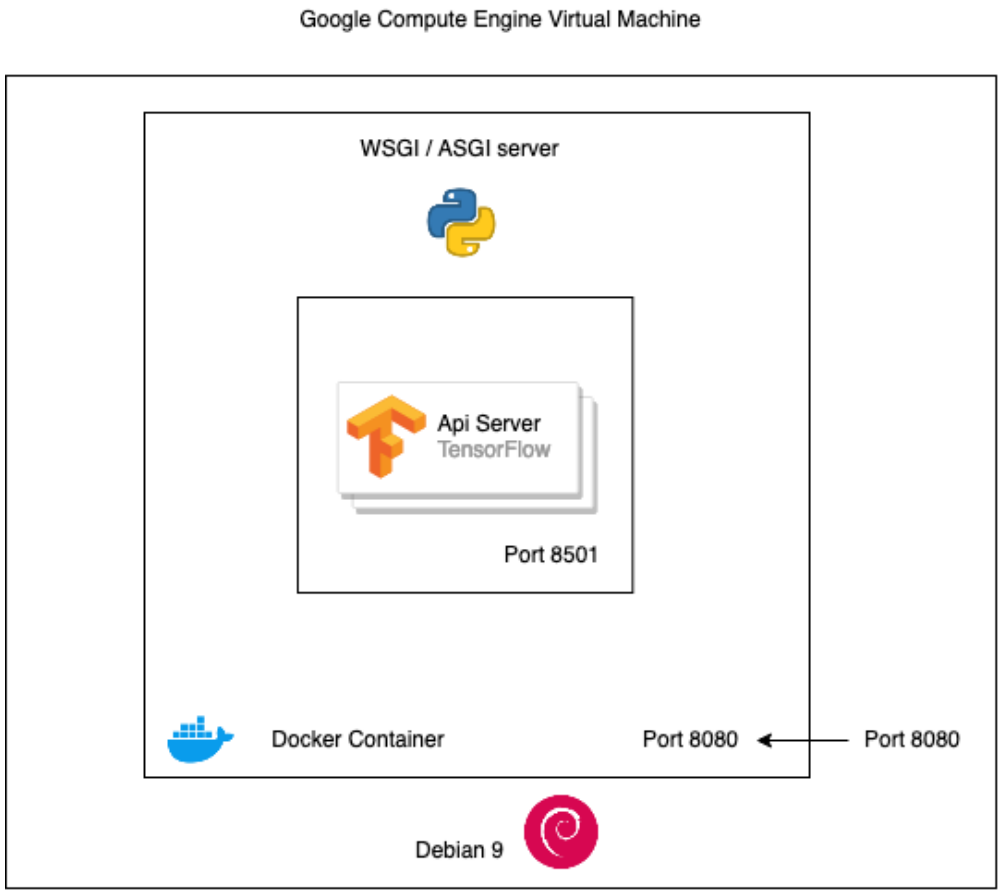


Figura 4.5: *Arquitectura de la máquina virtual de TensorFlow.*

Buscando la máxima portabilidad entre entornos, y así no cerrar la posibilidad de traslado de estas a otra plataforma, se han encapsulado todas las aplicaciones usando la tecnología de contenedores docker[10]. Este contenedor de docker se conectará con el host mediante el puerto 8080, permitiendo así el tráfico de información. Dentro del mismo 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. 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

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

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 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 acabo 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[11]. En su configuración se especifican los siguientes puntos:

- Sistema operativo base o imagen de la que hereda el contenedor.

- Puertos necesarios para ejecutar la aplicación, que son expuestos al exterior.

- Paquetes y actualizaciones del sistema operativo necesarios para funcionar.

- Librerías de Python.

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

- 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[12] 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 los comandos que queremos que se ejecuten de manera automática al iniciarse el contenedor.

[12] 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 subido 16.000 imágenes 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 :

- 2000 muestras con un procesador de 2 núcleos físicos, 4 hilos usando flask y TensorFlow.
- 2000 muestras con un procesador de 2 núcleos físicos, 4 hilos usando fastapi y TensorFlow.
- 2000 muestras con un procesador de 4 núcleos físicos, 8 hilos usando fastapi y TensorFlow.
- 2000 muestras con un procesador de 4 núcleos físicos, 8 hilos usando flask y TensorFlow.
- 2000 muestras con un procesador de 2 núcleos físicos, 4 hilos usando flask y OpenVINO.

- 2000 muestras con un procesador de 2 núcleos físicos, 4 hilos usando fastapi y OpenVINO.
- 2000 muestras con un procesador de 4 núcleos físicos, 8 hilos usando flask y OpenVINO.
- 2000 muestras con un procesador de 4 núcleos físicos, 8 hilos 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.

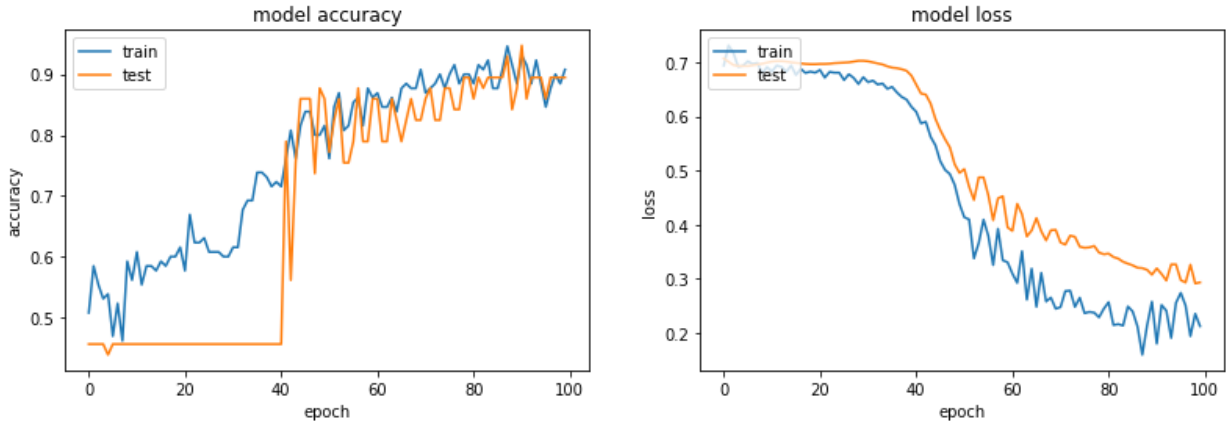
La comparativa de resultados y el correspondiente análisis ha sido realizado en la base de datos distribuida BigQuery, haciendo uso de SQL estándar.

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



(a) (b)

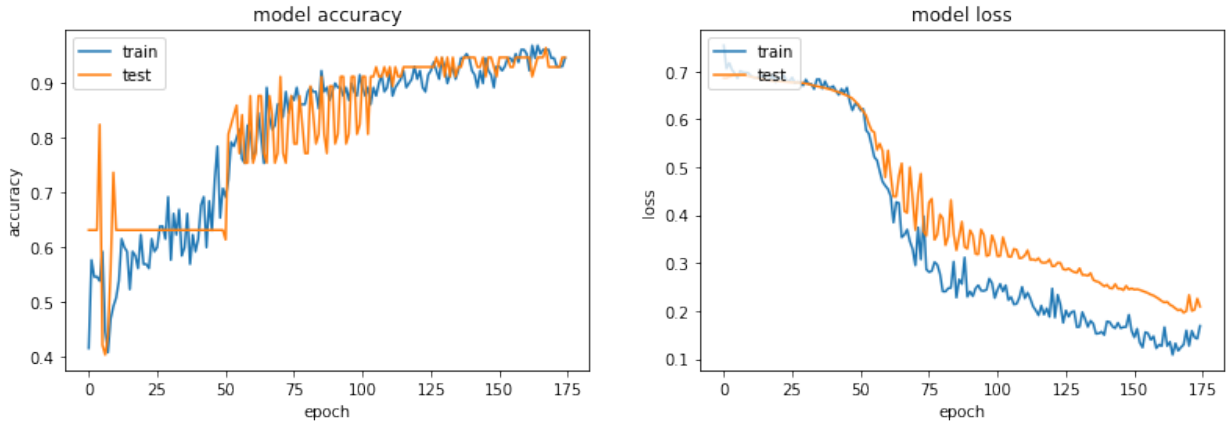
Figura 5.1: *Resultados de entrenamiento del modelo usando una GPU TeslaK80 con un batch-size de 256 y 100 epochs (a) Rendimiento del modelo en acierto. (b) Rendimiento del modelo en pérdida*

red neuronal de 175 epochs y 256 de tamaño de batch. El modelo ha sido entrenado con los siguientes hiperparámetros para conseguir el máximo nivel de precisión en el mínimo tiempo posible:

- **100 Epochs, 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 5.2 se encuentran los resultados de entrenamiento en términos de precisión y pérdida, respectivamente.

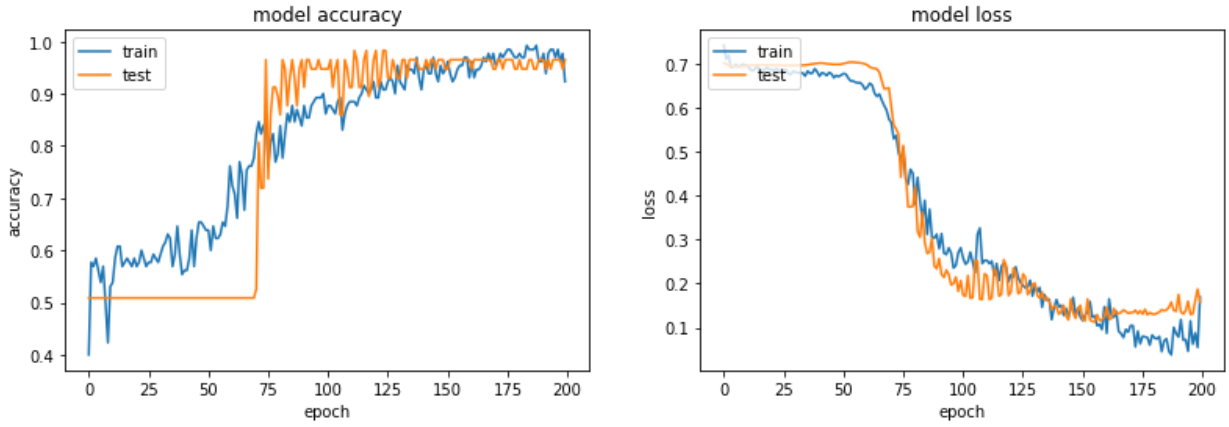
- **175 Epochs, 256 Batch-size**: con un tiempo total de entrenamiento de 25.86 segundos y una precisión del 93 % sobre el conjunto de datos de entrenamiento. En las Figura 5.2 se encuentran los resultados de entrenamiento en términos de precisión y pérdida.

- **200 Epochs, 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 Figura 5.2 para los resultados en precisión y pérdida.



(a) (b)

Figura 5.2: *Resultados de entrenamiento del modelo usando una GPU TeslaK80 con un batch-size de 256 y 175 epochs (a) Rendimiento del modelo en acierto. (b) Rendimiento del modelo en pérdida*



(a) (b)

Figura 5.3: *Resultados de entrenamiento del modelo usando una GPU TeslaK80 con un batch-size de 256 y 100 epochs (a) Rendimiento del modelo en acierto. (b) Rendimiento del modelo en pérdida*

5.3. Rendimiento en fase de inferencias

En cuanto a los tiempos de inferencia, la diferencia es notable entre OpenVINO y TensorFlow. Los resultados se presentan haciendo uso de una muestra de 16.000 ejecuciones de inferencia en el entorno de producción de la aplicación, tanto para los entornos con OpenVINO, como para TensorFlow. 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 en su primera versión junto con el procesador de 2 núcleos físicos a 8 GB en la versión de 4 núcleos físicos.

En la Tabla 5.1 se puede observar como la media de inferencia a lo largo de las 16.000 muestras tomadas es de 200 ms para TensorFlow y 4 ms para OpenVINO lo que supone una velocidad de inferencia 50 veces superior de OpenVINO frente a TensorFlow. También podemos visualizar en la Tabla 5.2 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. Y se reduce aún más en OpenVINO debido a la rapidez de su red de inferencia y la simplicidad con la que se implementa. Al contrario que TensorFlow, no necesita ningún otro tipo de servicio adicional más que la API de alto rendimiento, que funciona como una librería almacenada de manera local en el sistema operativo.

inference_engine	avg_inference_seconds	avg_seconds_total
TensorFlow	0.215	0.502
OpenVINO	0.004	0.098

Tabla 5.1: *Comparativa de tiempo de inferencia con OpenVINO y TensorFlow.*

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

inference_engine	physical_core	system_memory	avg_inference_seconds	avg_seconds_total
TensorFlow	4	7.00GB	0.066	0.188
OpenVINO	4	7.00GB	0.003	0.100
OpenVINO	2	3.46GB	0.005	0.097
TensorFlow	2	3.46GB	0.364	0.816

Tabla 5.2: *Comparativa de tiempo de inferencia con OpenVINO y TensorFlow con distinto hardware.*

resultados similares con un hardware de bajo coste. TensorFlow mejora en 5 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 360 ms con el primero y 66 ms con el segundo, lo que denota que el consumo de su servicio de inferencia requiere de un hardware más caro. Del mismo modo mejora 8 veces su tiempo de inferencia total con la mejora del hardware pasando de 800 ms a 97 ms.

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. En la Tabla 5.3 podemos contemplar que OpenVINO se mantiene estable con ambos framework, con un ligero aumento de la latencia haciendo uso de fastapi. TensorFlow se comporta mucho mejor con flask, mejorando en 4 veces su tiempo de inferencia en la red, pasando de 348 ms con fastapi a 83 ms con flask y 3.7 veces en el tiempo total de ejecución teniendo un rendimiento de 700 ms con fastapi a 200 ms con flask. 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. Los componentes que proporciona fastapi como un sistema de logging detallado de las ejecuciones y algunas características adicionales para el desarrollador, 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.

En general, fastapi requiere de un hardware más potente para sacar su máximo rendi-

inference_engine	web_engine	avg_inference_seconds	avg_seconds_total
OpenVINO	Flask	0.004	0.094
OpenVINO	FastApi	0.005	0.103
TensorFlow	Flask	0.083	0.212
TensorFlow	FastApi	0.348	0.792

Tabla 5.3: *Comparativa de tiempo de inferencia con OpenVINO y TensorFlow con distinto framework web.*

miento, mientras que con una framework minimalista como flask podemos optar por reducir costes en hardware sin penalizar demasiado el rendimiento. En la Tabla5.4 podemos ver el rendimiento según hardware, servidor web y sistema de inferencia utilizado.

engine	cores	framework	memory	avg_inference_seconds	avg_seconds_total
OpenVINO	2	3.46GB	FastApi	0.006	0.104
OpenVINO	4	7.00GB	FastApi	0.004	0.102
TensorFlow	2	3.46GB	FastApi	0.608	1.349
TensorFlow	4	7.00GB	FastApi	0.087	0.235
OpenVINO	2	3.46GB	Flask	0.004	0.090
OpenVINO	4	7.00GB	Flask	0.003	0.098
TensorFlow	2	3.46GB	Flask	0.120	0.284
TensorFlow	4	7.00GB	Flask	0.045	0.141

Tabla 5.4: *Comparativa de tiempo de inferencia con OpenVINO y TensorFlow con distinto hardware y servidor web.*

El número total de aciertos de clasificación asciende a 1566 registros de un total de 16000, teniendo así un porcentaje de acierto general del 97 %. Con un recuento de 108 fallos en la red TensorFlow y 226 en OpenVINO. Todos los resultados y registros se encuentran en el repositorio oficial del trabajo en GitHub[1].

5.4. Costes del proyecto

A continuación se presentan los costes del proyecto de toda la plataforma de producción. Estos costes han sido recogidos haciendo uso de la calculadora de precios de Google[2].

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

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

- Máquina virtual 4 vCPUs, 3,6 GB 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 4 vCPUs, 3,6 GB 3.40 dólares por un uso de 24 horas, que fue el tiempo real consumido para este servicio.
- BigQuery, con un coste de 0 dólares para un almacenamiento de 1 GB de tablas en la base de datos, 1GB de procesamiento en tiempo real y 1GB de trabajos SQL al realizar los análisis de resultados.
- Pub/Sub con un coste total de 0 dólares, ya que su uso entraba dentro del rango gratuito del servicio.
- Cloud Functions, con un coste total de 0 dólares, haciendo uso de la modalidad gratuita, que permite hasta 2 millones de llamadas al mes.

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.

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 de principal de la puesta en producción 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 cluster Kubernetes[1]. Este trabajo ha usado 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 en una base de datos, 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.

En nuestro problema un vehículo aéreo podría portar el propio 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[2]

[1] https://kubernetes.io/es/docs/concepts/overview/what-is-Kubernetes/
[2] https://aws.amazon.com/es/deeplens/

de Amazon. 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].

[3] https://en.wikipedia.org/wiki/Heat_map
[4] https://es.wikipedia.org/wiki/Secure_Shell

Apéndice A

Introduction

A.1. Motivation

A hyperspectral image is a high spectral resolution image obtained through sensors capable of obtaining hundreds or even thousands of images on the same terrestrial area but corresponding to different wavelength channels. The set of spectral bands is not strictly limited to the visible spectrum but also covers the infrared and the ultraviolet.

At present, the use of hyperspectral images is increasing considerably due to the launching of new satellites and the interest in remote observation of the Earth, which has utility in areas as diverse as defense, precision agriculture, geology (detection of mineral deposits) , valuation of environmental impacts or even artificial vision.

During the last years there have been many advances with regard to sensor technology, which has revolutionized the collection, handling and analysis of the data collected. This evolution has managed to go from having a few tens of bands to having hundreds and the tendency is for the number to continue increasing. Institutions such as National Aeronautics and Space Administration (NASA) or the European Space Agency (ESA) are continuously obtaining a large amount of data that needs to be processed. As a result, new challenges have arisen in the processing of data.

If we add to the increase in the amount of information collected, many current and future remote observation applications require real-time processing capabilities (in the same time

or less than the satellite takes to capture the data) or close to this real-time, it is essential to use parallel architectures for the efficient [**?**] and fast processing of this type of images.

The main problem in the processing of hyperspectral images lies in the spectral mixture, that is to say, the existence of mixed pixels in which several different materials coexist at the subpixel level. This type of pixels are the most common in hyperspectral images and for their analysis it is necessary to use complex algorithms with a high computational cost, which makes the execution of the demixing algorithms slow and requires acceleration or parallelization.

To address this type of tasks, parallel computing has been widely used through multi-core processors, GPUs (Graphics Processing Units) or dedicated hardware such as FPGAs (Field-Programmable Gate Arrays). Of all the alternatives, the latter present an efficient option in terms of performance, offering reduced times, in addition to a lesser use of resources, being the few alternatives that can be adapted in a sensor to perform on-board processing in space missions such as Mars Pathfinder or Mars Surveyor [**?**].

On the one hand, VHDL or Verilog are the native ways to program this type of devices, at a low level and more optimal. On the other hand, there is an alternative in OpenCL that allows a high level programming, faster and allowing its execution in a variety of architectures but less optimal at the level of hardware resources than in FPGAs devices.

A.2. Objectives

The general objective of this work is the parallel implementation on FPGA of the Automatic Target Detection and Classification Algorithm [**? ?**] making use of the Gram Schmidt Orthogonalization and the programming languages VHDL and OpenCL. This will allow a very interesting comparison between a native language for said platform (VHDL) and another paradigm of parallel programming at a high level (OpenCL) that can be ported to other platforms such as multi-core processors, GPUs or other accelerators.

The achievement of the general objective is carried out in the present memory by ad-

dressing a series of specific objectives, which are listed below:

- Design of individual modules in VHDL that serve to perform all the operations that are needed for the implementation of the ATDCA-GS algorithm.
- Elaboration of a state machine and implementation of the algorithm using the individual modules.
- Analysis and optimization of a previous parallel implementation in OpenCL of the algorithm.
- Obtaining results and performance comparisons between both programming languages.

A.3. Organization of this memory

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

- **Hyperspectral analysis**: the hyperspectral image concept and the linear mixing model are defined; some hyperspectral sensors (AVIRIS and EO-1 Hyperion) and some spectral libraries (USGS and ASTER) are mentioned; and finally, the need for parallelization and platforms that can be used to address the problem of performance improvement is presented.
- **FPGAs technologies**: FPGA technologies are defined in a short way.
- **Implementation**: the algorithm ATDCA-GS in series is defined and the parallelization and optimization that has been carried out in both VHDL and OpenCL languages is explained.

- **Results**: the results obtained after the implementation and execution of the algorithm in FPGAs devices are presented.
- **Conclusions and future work**: the main conclusions of the aspects addressed in the work that have been reached and also some possible lines of future work that can be performed in relation to this work are presented.

Apéndice B

Conclusions and future work

B.1. Conclusions

In this end-of-degree project, the design and implementation of the ATDCA-GS algorithm has been carried out, using Gram Schmidt orthogonalization in order to optimizing and improving the performance of complex operations such as the calculation of the inverse of a matrix. The programming languages VHDL and OpenCL have been used and the results of their execution in FPGA have been evaluated to subsequently make a performance comparison between both alternatives.

As part of the design, an adaptation of the algorithm to the usual flow of a specific hardware design has been carried out, minimizing as much as possible the amount of resources to be used and parallelizing the operations carried out during the execution of the algorithm.

To make a comparison in terms of the performance in time of the two implementations, it has been compared the acceleration of one with respect to the other making use of real and synthetic images. In addition, it has been verified that, except for the implementation in OpenCL for large images, the processing in both alternatives does not exceed the time limit (maximum) and therefore the real-time analysis can be performed, fulfilling one of the main objectives of this proyect.

The performance tests in terms of resources used in each implementation have revealed that the percentage of resources used increases linearly with the number of bands. It also

revealed that for a large number of them (256), the resource with the highest use hardly reaches 86 % of use in VHDL and 48 % in OpenCL, so it is concluded that the performance is adequate.

B.2. Lines of future work

In the first place, it would be convenient to improve the implementation optimizations in OpenCL so that it allows a real-time analysis as well as the other alternatives. In addition, since the tendency of the size of the images is to continue growing more and more, the future work option that seems more evident is to be able to process other real images of an even larger size.

A possible future work path for this work would be to develop the algorithm by converting the floating-point arithmetic to whole arithmetic. In this way a better performance would be obtained since the calculations would be even simpler and, therefore, the number of necessary resources would decrease while increasing the clock frequency.

Another possible way of continuation could be the modification of the test platform to use the PCIe 3x8 bus. In this way penalties due to I / O would be reduced.

Finally, another way would be to choose whether the implementation kernels in OpenCL can follow a parallel programming model at task level, so that the task refers to the execution of a kernel with a work-group that contains a work-item and, thus, the compiler tries to accelerate the only work-item to get a better performance.