Implementación paralela del Automatic Target Detection and Classification Algorithm haciendo uso de la ortogonalización de Gram Schmidt para el análisis de imágenes hiperespectrales

Andrés Ortiz Loaiza

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

Trabajo Fin de Grado en Ingeniería de Computadores

Madrid, mayo de 2019

Directores:

González Calvo, Carlos
Bernabé García, Sergio

Agradecimientos

Índice general

Índice de figuras

Índice de tablas

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 del en un entorno cloud de google, de manera 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 OpenVINO. Finalmente, el modelo se desplegará en un entorno cloud en el que pueda funcionar como servicio capaz de sopotar miles de llamadas concurrentes. 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 la velocidad de inferencia del mismo.
- 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.

1.2. 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 aprendizaje 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. Los gobiernos

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 la enseñanza de los mismos. 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 tecnologías que solían funcionar con un objeto de estudio se conviertan en una práctica diaria. 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 poder

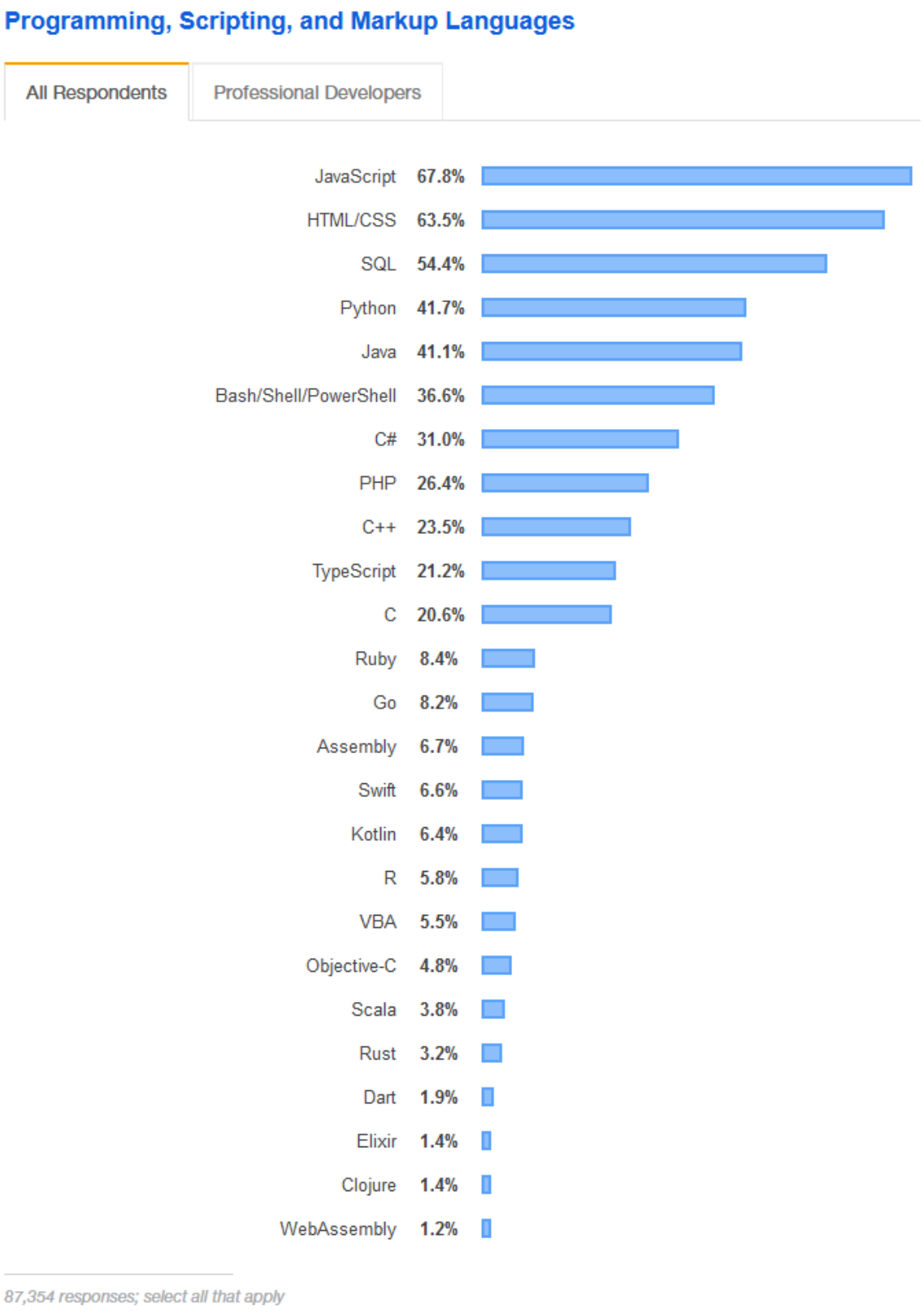

Figura 1.1: *Encuesta sobre lenguajes de programación usados en StackOverflow 2019.*

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 tales 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, 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.

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 un criterio para seleccionar un sub-grupo a partir de un grupo de componentes más grande. 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.

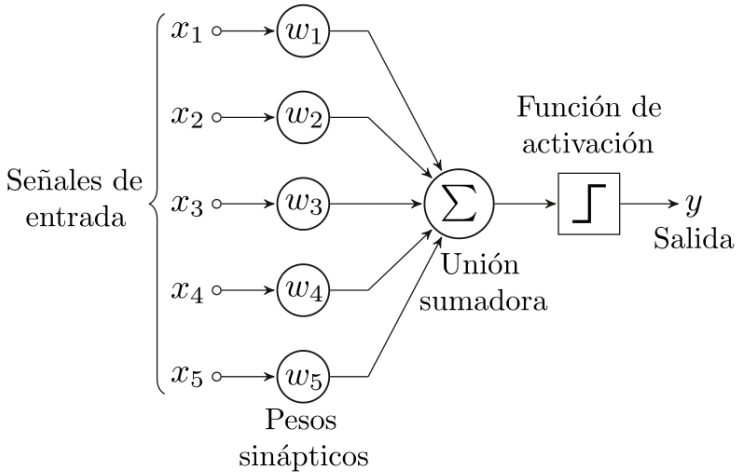

Figura 1.2: *Ejemplo de perceptrón.*

A partir de aquí cada neurona aplica una función de cálculo a partir de los pesos mediante la cual acaba propagando el resultado a las capas posteriores, y finalmente a la capa de salida mediante la que podremos obtener nuestra clasificación final. En este problema concreto, nos centramos en clasificar imágenes multiespectrales, que consisten en la captura de datos de imágenes dentro de rangos de longitud de onda específicos a través del espectro electromagnético. Nuestro conjunto de imágenes multiespectrales pertenece a una zona parcialmente destruida por un desastre natural. El objetivo de nuestro algoritmo 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.
- **Tecnología OpenVINO**: Se define el propósito de la herramienta de Intel OpenVINO así como la transformación de un modelo de TensorFlow para que sea compatible con dicha herramienta.
- **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

Uno de los parámetros más importantes dentro del entrenamiento de modelos de deep learning es la velocidad. La velocidad de entrenamiento es de vital importancia ya que los modelos pueden requerir entradas de tamaño masivo, por lo que la capacidad de cómputo va a ser clave para acelerar este proceso y poder enfocarnos plenamente a la mejora del rendimiento de nuestro modelo, sin tener como cuello de botella la espera que nos pueda producir volver a entrenar el mismo con distintos parámetros. De este modo podemos reajustar constantemente nuestro modelo para encontrar el punto óptimo de manera ágil. En este trabajo vamos a entrenar nuestro modelo haciendo uso del framework de código abierto de TensorFlow, el cual 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 y los propios procedimientos que realizan las redes neuronales para hacer sus cálculos son operaciones matriciales. Nuestro objetivo será aprovechar al máximo el rendimiento que una GPU nos puede dar en este tipo de operaciones gracias a su arquitectura de paralelización, ya que es la idónea para este tipo de trabajo. La ventaja que nos da frente a la CPU es la capacidad de cómputo con un mayor número de núcleos o cores (ver Figura 2.1), gracias a su conectividad por PCI express y el ancho de banda que esta proporciona.

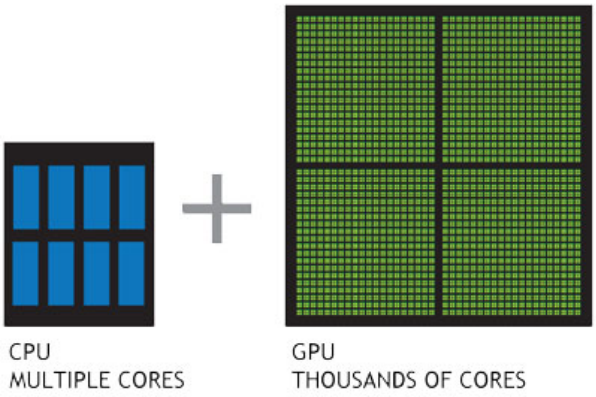

Figura 2.1: *Arquitectura de paralelización de una GPU.*

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 (describir dataset, procedencia). Una imagen multiespectral es la que captura datos de imágenes dentro de rangos de longitud de onda específicos a través del espectro electromagnético 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 gracias a que sus objetos y funciones están construidos de una manera intuitiva. La distribución a efectuar sobre el conjunto de datos en entrenamiento y test es del 70 y 30 por ciento respectivamente. Para la construcción de este algoritmo haremos uso de las siguientes capas, sobre las que TensorFlow nos da una API para tener control total sobre su configuración.

- **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 requisitos una imagen en tres dimensiones, 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 3x3 ya que nuestras imágenes son de 128x128 píxeles para ayudar a nuestro modelo a mejorar su aprendizaje con filtros de mayor tamaño.

- **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 no lineales entre entradas y salidas. 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. Mientras que otras funciones de activación como la función Sigmoide o Tanh modifican todos los valores de entrada, la función Relu mantendrá los valores de peso 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 la reducción de sus dimensiones. En nuestro modelo aplicaremos una reducción a matrices de 2x2.

- **Dropout**: Es una capa cuyo cometido es ignorar ciertas neuronas de forma aleatoria para no incluirlas en el entrenamiento, de modo 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**: Es 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 tasas de aprendizaje individuales para cada parámetro.

En la Figura 2.2 podemos observar el conjunto de capas utilizado para crear la topología de la red de nuestro modelo.

Además, realizaremos algunas optimizaciones a nivel de hardware para acelerar el proceso de forma general.

- **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 (Accelerated Linear Algebra) optimiza el grafo de nuestro modelo de manera específica haciendo uso de la GPU.
- **Valores altos del parámetro de entrenamiento BatchSize**: Gracias a la capacidad de cómputo de nuestra tarjeta gráfica podemos permitirnos el uso de valores altos en este parámetro de entrenamiento.

El modelo ha sido entrenado con los siguientes hiperparámetros para conseguir el máximo nivel de precición en el mínimo tiempo posible.

- **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.

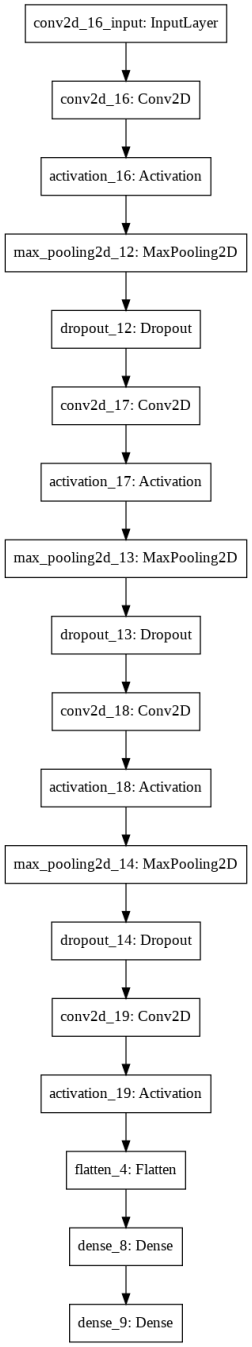

Figura 2.2: *Topología de la red del modelo de redes neuronales.*

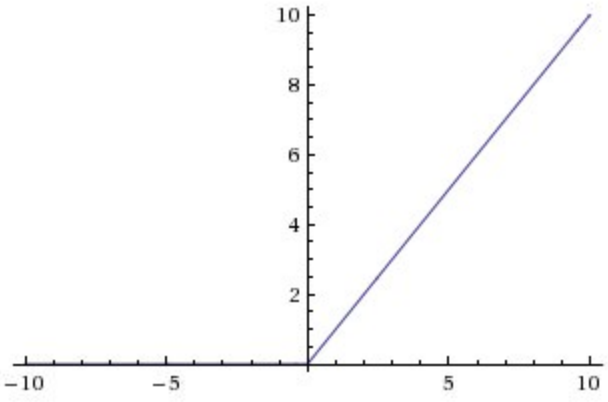

Figura 2.3: *Función de activación Relu.*

- **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.
- **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.

2.2. Entorno Google Colab

La plataforma de Google Colab 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 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 tarjeta gráfica Tesla K80. 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 y 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, permitiendo así poder trabajar de manera fluida realizando otro tipo de cometidos en nuestra máquina o simplemente ejecutando un proceso para el cual no tenemos suficiente potencia disponible.

Capítulo 3

Tecnología Openvino

Openvino es un conjunto de herramientas multi plataforma desarrollada por Intel que facilita la transición entre entre los entornos de entrenamiento y producción de nuestro modelo de aprendizaje profundo. Openvino a pesar de estar desarrolalda por una empresa comercial como Intel peternece al conjunto de aplicaciones de código abierto, de modo que podemos visualizar su código fuente, repotar fallos en incluso realizar aportaciones. Podemos visualizar el diseño y el código de la aplicación en su repositorio oficial de Github en el siguiente link https://github.com/openvinotoolkit/openvino. 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, son estos archivos los que procesa su propia red de inferencia multi plataforma, la cual está preparada para procesar estos archivos y poder realizar los trabajos de manera concurrente, aprovechando así, toda la potencia de los procesadores o gpu actuales. En la siguiente figura podemos observar el flujo de trabajo que se ha seguido en este trabajo haciendo uso de la herramienta Openvino 3.1 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

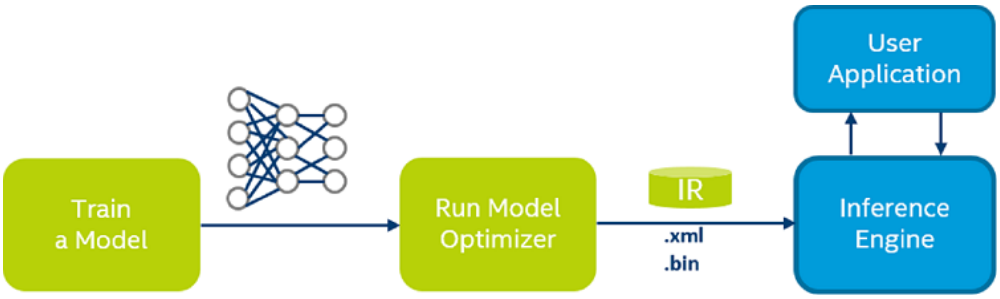

Figura 3.1: *Arquitectura de optimización de modelos con Openvino*

3.1. Herramientas que lo componen

Esta tecnología desarrollada por intel se centra principalmente en la optimización de modelos de redes neuronales convolucionales para potenciar su velocidad de inferencia mediante las distintas herramientas que lo componen. La herramienta soporta distintos hardwares como FPGA, Intel movidius, procesamiento por GPU y también distintos sistemas operativos como 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 multi plataforma para realizar la inferencia de manera rápida.

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 realiza una conversión de cada capa interna de la red a una nueva capa. Esta nueva capa, ya del formato de Openvino conserva los pesos de la red anterior, sin embargo, la definición de la misma está preparada

para la que la aplicación de inferencia de Openvino pueda leerla correctamente. Openvino proporciona de manera genérica distintos scripts para realizar esta conversión. Se incluyen distintos scripts para los diferentes frameworks de deep learning más actuales, los mismos están codificados en python y podemos modificar su código fuente, aunque en principio no es necesario porque ya vienen preparados para funcionar.

3.1.2. Interfaz de infernecia de modelos de deep learning

Con nuestro modelo y su topología convertida a un formato válido de Openvino tenemos todo lo necesario para poder realizar clasificaciones con la interfaz de inferencia de Openvino. La optimización de inferencia se produce en este punto, donde cada capa de nuestro modelo original es procesada por la aplicación de inferencia en un lenguaje de bajo nivel, el cual está optimizado para realizar operaciones vectoriales bajo total control del programador. Este lenguaje es C++, aunque el programa puede ser utilizada también en Python gracias al uso de su API, la cual traduce las peticiones realizadas en Python al core de la interfaz, el cual es C++ puro. Todas las capas usadas en este proyecto son compatibles de manera directa con las predefinidas por la interfaz de inferencia. Openvino nos da la posibilidad de añadir capas personalizadas, con el inconveniente de que estas capas deben de 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 modelo 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 grafico subyacente de Tensorflow. Separa los graficos que se guardan para prediccion (servicio), capacitacion y evaluacion. Si el modelo no se compilo antes, solo el grafico de inferencia se exporta.
- La configuracion de arquitectura del modelo, si esta disponible

Los métodos exactos para la serializaciñon del modelo varían según la versión de tensorflow, en cualquier caso la metodologíoa de conversión exacta utilizada para este proyecto se puede encontrar en el repositorio de código fuente de este trabajo en Github : https://github.com/A-Ortiz-L/multispectral-imaging-cnn-final-degree-work

3.3. Inferencias. Tensorflow vs OpenVINO

(Los resultados no habría que ponerlos en teoría aquí, sino llevarlos al apartado

Capítulo 4

Implementación

4.1. Descripción del entorno y sus diferentes componentes

ok

4.2. Codificación de los servidores web

ok

4.2.1. Framework FastApi

ok

4.2.2. Framework Flask

oki

4.3. Encapsulación de entorno con Docker

(Explicación de las distintas imágenes de Docker de la aplicación y cómo se han construido)

Capítulo 5

Resultados experimentales

5.1. Dataset usado

ok

5.2. Rendimiento en fase de entrenamiento

ok

5.3. Ren dimiento en fase de inferencias

ok

5.4. Costes del proyecto

Capítulo 6

Conclusiones y trabajo futuro

6.1. Conclusiones

En este Trabajo de Fin de Grado se ha llevado a cabo el diseño y la implementación del algoritmo ATDCA-GS, que utiliza la ortogonalización de Gram Schmidt con el objetivo de optimizar y mejorar el rendimiento de operaciones complejas como es el caso del cálculo de la inversa de una matriz. Se han empleado los lenguajes de programación VHDL y OpenCL y se han evaluado los resultados de su ejecución en FPGA para posteriormente realizar una comparación de rendimiento entre ambas alternativas.

Como parte del diseño, se ha realizado una adaptación del algoritmo adecuándolo al flujo habitual de un diseño específico hardware, minimizando lo máximo posible la cantidad de recursos a emplear y paralelizando las operaciones llevadas a cabo durante la ejecución del algoritmo.

Para realizar una comparativa en cuanto al rendimiento en tiempo de las dos implementaciones, se ha teniendo en cuenta la aceleración de una respecto a la otra haciendo uso de imágenes reales y sintéticas. Además, se ha verificado que, excepto la implementación en OpenCL para imágenes de gran tamaño, el procesamiento en ambas alternativas no supera el tiempo límite (máximo) y por tanto se puede realizar el análisis en tiempo real, cumpliendo uno de los objetivos principales de este trabajo.

Las pruebas de rendimiento en cuanto a recursos empleados en cada implementación

han desvelado que el porcentaje de recursos utilizados aumenta linealmente con el número de bandas. También revela que para un número elevado de las mismas (256) el recurso de mayor uso apenas alcanza un 86 % de utilización en VHDL y un 48 % en OpenCL, por lo que se concluye que el rendimiento es adecuado.

6.2. Trabajo futuro

En primer lugar, sería conveniente mejorar las optimizaciones de la implementación en OpenCL para que permita un análisis en tiempo real al igual que las demás alternativas. Además, dado que la tendencia del tamaño de las imágenes es continuar creciendo cada vez más, la opción de trabajo futuro que parece más evidente es la de conseguir procesar otras imágenes reales de un tamaño aún mayor.

Una posible vía de trabajo futuro para este trabajo sería desarrollar el algoritmo convirtiendo la aritmética de punto flotante a aritmética entera. De esta manera se conseguiría un mejor rendimiento ya que los cálculos serían aún más sencillos y, por lo tanto, el número de recursos necesarios disminuiría a la vez que aumentaría la frencuencia de reloj.

Otro posible camino de continuación podría ser la modificación de la plataforma de test para utilizar el bus PCIe 3x8. De esta manera se reducirían las penalizaciones debidas a la E/S.

Por último, se podría optar por analizar si los kernels de la implementación en OpenCL pueden seguir un modelo de programación paralela a nivel de tarea, de modo que la tarea se refiera a la ejecución de un kernel con un work-group que contenga un work-item y, así, el compilador intente acelerar el único work-item para conseguir un rendimiento mejor.

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.