Universidad Internacional de La Rioja (UNIR)

ESIT

Máster Universitario en Inteligencia Artificial

Clúster con hardware embebido, Kubernetes y paralelización de GPUs para *Edge* AI

Trabajo Fin de Máster

Presentado por: Crisóstomo Vals, Carlos y Garrido Viro, Rafael

Director/a: Cobos, Salvador

Ciudad: Madrid
Fecha: 22/07/2021

Índice de contenidos

Índice de tablas

Índice de figuras

Resumen

El presente trabajo de fin de Máster tiene como objetivo desarrollar una plataforma Edge de inteligencia artificial utilizando un clúster hardware de 2 placas nVidia Jetson Nano, sistemas operativos GNU/Linux Embebidos y tecnologías de Kubernetes y paralelización de GPUs para un caso de uso de visión computacional utilizando redes neuronales. Se realizarán diferentes comparativas de rendimiento del clúster respecto a los casos de uso de una única placa con GPU y otros equipos de referencia.

El caso de uso de visión computacional elegido es utilizar redes neuronales de tipo CNN como solución para la predicción temprana de casos de COVID-19, iniciativa planteada por la Comisión Europea en su publicación “AI-ROBOTICS vs COVID-19” (Alliance, n.d.) ,que tuvo mucha repercusión al inicio de la pandemia, utilizando imágenes de placas de tórax obtenidas con máquinas de Rayos X.

Este tipo de soluciones, al no depender de entidades externas, podrían ser utilizadas directamente en entornos hospitalarios sin más dependencias que la conexión con la propia máquina de Rayos X o un depósito de imágenes para interpretar y analizar las imágenes, lo que permitiría gestionar la privacidad de los datos de los pacientes y eliminar las dependencias con infraestructuras fuera del propio hospital o centro de salud.

Como detalles del proyecto planteamos el empleo y optimización de tecnologías consideradas de bajo coste en este campo como pueden ser las placas nVidia Jetson Nano, así como del soporte de software proporcionado por el fabricante de cara a optimización de IA para este tipo de hardware en su BSP, pero lo ampliamos incrementado la capacidad de procesamiento montando en un clúster de dos unidades, aunque con el objetivo que sea totalmente escalable para más unidades con mínimos cambios posibles. Para realizar esto contamos además con la plataforma de gestión de clústeres de contenedores Kubernetes adaptada para sistemas embebidos de este tipo, lo que permitirá una administración del sistema fácil y de escalabilidad totalmente configurable adaptada a las cargas de trabajo necesarias.

Concluyendo, el conjunto de tecnologías en Edge aplicadas en este trabajo servirá para facilitar y acelerar la detección de casos de COVID-19 in situ utilizando redes neuronales convolucionales, sin dependencias externas de ningún tipo, bajo coste y un alto rendimiento y escalabilidad.

Palabras Clave: Clúster, Kubernetes, Edge AI, CNN, COVID-19, Rayos X

Abstract

This Master's Dissertation aims to develop an artificial intelligence Edge platform using a hardware cluster of 2 nVidia Jetson Nano boards, GNU/Linux Embedded operating systems and Kubernetes and GPU parallelisation technologies for a computational vision use case using neural networks. Different performance comparisons of the cluster will be made with respect to the use cases of a single board with GPU and other reference equipment.

The computer vision use case chosen is to use CNN type neural networks as a solution for the early prediction of COVID-19 cases, an initiative proposed by the European Commission in its publication "AI-ROBOTICS vs COVID-19" (Alliance, n.d.)which had a great impact at the beginning of the pandemic, using chest X-ray machine images.

This type of solution, as it does not depend on external entities, could be used directly in hospital environments with no dependencies other than the connection to the X-ray machine itself or an image repository to interpret and analyse the images, which would allow the privacy of patient data to be managed and eliminate dependencies with infrastructures outside the hospital or health centre itself.

As details of the project, we propose the use and optimisation of technologies considered low-cost in this field, such as nVidia Jetson Nano boards, as well as the software support provided by the manufacturer for optimising AI for this type of hardware in its BSP, but we extend it by increasing the processing capacity by building a cluster of two units, although with the aim of making it fully scalable for more units with minimum changes. To achieve this we have the Kubernetes container cluster management platform adapted for embedded systems of this type, which will allow easy administration of the system and fully configurable scalability adapted to the necessary workloads.

In conclusion, the set of Edge technologies applied in this work will serve to facilitate and accelerate the detection of COVID-19 cases in situ using convolutional neural networks, without external dependencies of any kind, with low cost, high performance and scalability.

Keywords: Cluster, Kubernetes, Edge AI, CNN, COVID-19, X-Rays

Organización del trabajo en grupo

Partes comunes realizadas por los dos alumnos:

Capítulo 3. Objetivos y metodología

Capítulo 6. Discusión

Capítulo 7. Conclusiones y trabajo futuro

Partes realizadas por el alumno **Carlos Crisóstomo Vals**:

Capítulo 1. Introducción

1.4 *TensorFlow*

Objetivos perseguidos: Presentar de manera introductoria la plataforma TensorFlow utilizada en el proyecto como base para los desarrollos de Inteligencia Artificial empleados.

1.5 *Infraestructura en Edge*

Objetivos perseguidos: Presentar de manera introductoria la infraestructura en Edge a utilizar en el proyecto, sus características hardware, la plataforma Docker y el sistema de orquestación de contenedores Kubernetes.

Capítulo 2. Contexto y Estado del Arte

2.4 *Computación en Edge o Edge Computing*

Objetivos perseguidos: Estudiar y presentar el estado del arte actual respecto a las tecnologías de Edge Computing

2.5 *Uso de la inteligencia artificial en el Edge*

Objetivos perseguidos: Estudiar y presentar el estado del arte actual respecto a las tecnologías de inteligencia artificial aplicadas al Edge Computing

2.6 *TensorFlow*

Objetivos perseguidos: Estudiar y presentar el estado del arte actual respecto a TensorFlow, desde sus componentes másicos y arquitectura hasta su evolución a TensorFlow 2.0, la API Keras y el software de inferencia TensorFlow Serving.

2.7 *Docker*

Objetivos perseguidos: Estudiar y presentar el estado del arte actual respecto a la tecnología de contenedores Docker, incluyendo su infraestructura a nivel de Kernel Linux (LXC, CGroups, Namespaces), los conceptos de capas o Layers de un contenedor y el Copy-On-Write.

2.8 *Kubernetes y K3S*

Objetivos perseguidos: Estudiar y presentar el estado del arte actual respecto a la tecnología de orquestación de contenedores Kubernetes, tanto a nivel de componentes y arquitectura como su solución embebida K3S dispuesta por Rancher Labs y utilizada en el proyecto.

Capítulo 4. Descripción detallada del experimento

4.2 *Consideraciones para el diseño de la solución y los experimentos*

Objetivos perseguidos: Exponer ciertas consideraciones en cuanto al diseño de la solución y los experimentos a realizar, como son el uso de la placa Jetson, rendimiento teórico esperado y selección de la base de datos para los experimentos

4.4 *Entrenamiento y pruebas en portátil Macbook Pro*

Objetivos perseguidos: Describir las tareas realizadas en cuanto al entrenamiento y las pruebas realizadas en el equipo Macbook Pro.

4.5 *Montaje de infraestructura en clúster*

Objetivos perseguidos: Describir las tareas de montaje de la infraestructura del clúster realizada a nivel de sistema operativo, paquetes y runtimes utilizados, además de la configuración del propio clúster Kubernetes.

4.6 *Entrenamiento en clúster*

Objetivos perseguidos: Describir las tareas realizadas en el entrenamiento del clúster con las tecnologías aplicadas, desde la tarea inicial de ver el soporte de GPU en las placas, gestión de contenedores con GPU y entorno de desarrollo JupyterLab hasta el entrenamiento utilizando un nodo, dos nodos con configuración MultiWorkerMirroredStrategy y gestión completa con resolver de Kubernetes.

4.7 *Entrenamiento y pruebas en nVidia Xavier AGX*

Objetivos perseguidos: Describir las tareas realizadas en cuanto al entrenamiento y las pruebas realizadas en el equipo nVidia Xavier AGX.

4.8 *Inferencia en Macbook Pro*

Objetivos perseguidos: Describir las tareas realizadas para inferencia en el equipo Macbook Pro, desde la gestión y configuración de TensorFlow Serving en arquitectura x86_64 hasta la parte cliente de conexión a la API para obtener los resultados y previsión del modelo.

4.9 *Inferencia en clúster*

Objetivos perseguidos: Describir las tareas realizadas para inferencia en clúster, incluyendo la problemática de las imágenes de TensorFlow Serving para ARM64, la gestión y configuración de TensorFlow Serving conseguida, la parte cliente de conexión a la API para obtener los resultados y previsión del modelo utilizando contenedores y el uso completo del clúster con un balanceador de carga en Kubernetes.

Capítulo 5. Desarrollo y contribuciones del trabajo

5.2 *Entrenamiento y pruebas en portátil Macbook Pro*

Objetivos perseguidos: Datos y resultados relevantes del entrenamiento y las pruebas realizadas en el portátil Macbook Pro.

5.3 *Entrenamiento en clúster*

Objetivos perseguidos: Datos y resultados relevantes del entrenamiento y las pruebas realizadas en el clúster en todas sus casuísticas.

5.4 *Inferencia en nVidia Xavier AGX*

Objetivos perseguidos: Datos y resultados relevantes de la inferencia y las pruebas realizadas en la placa nVidia Xavier AGX.

5.5 *Inferencia en portátil Macbook Pro*

Objetivos perseguidos: Datos y resultados relevantes de la inferencia y las pruebas realizadas en el portátil Macbook Pro.

5.6 *Inferencia en clúster*

Objetivos perseguidos: Datos y resultados relevantes de la inferencia y las pruebas realizadas en el clúster en todas sus casuísticas.

Anexo 2. Repositorio de código fuente

Anexos A2.2, A2.2.3, A2.2.4, A2.3, A2.4, A2.5, A2.6, A2.7, A2.8, A2.9, A2.10, A2.11

Objetivos perseguidos: Registro de todo el código, scripts, ficheros YAML de definición de Kubernetes utilizados.

Partes realizadas por el alumno **Rafael Garrido Viro**:

Capítulo 1. Introducción

1.1 *Acerca del SARS-CoV-2*

Objetivos perseguidos: Presentar de manera introductoria el virus SARS-CoV-2, empleado como origen del caso de uso tomado como ejemplo para el desarrollo del proyecto.

1.2 *Diagnóstico del SARS-CoV-2 mediante Rayos-X*

Objetivos perseguidos: Presentar de manera introductoria el método de detección de la infección por el virus SARS-CoV-2 basadas en Rayos-X.

1.3 *Reconocimiento de imágenes – CNNs*

Objetivos perseguidos: Presentar de manera introductoria las tecnologías de reconocimiento de imágenes mediante redes neuronales artificiales, de tipo CNN.

1.6 *Nuestra prueba de concepto*

Objetivos perseguidos: Presentar inicialmente las posibles alternativas de procesamiento de algoritmos de reconocimiento de imágenes empleando diversas las opciones de Hardware y software que desarrollaremos ampliamente durante el proyecto.

Capítulo 2. Contexto y Estado del Arte

2.1 *Sobre la investigación del SARS-CoV-2*

Objetivos perseguidos: Estudiar y presentar el estado del arte actual respecto al origen, características y evolución del virus SARS-CoV-2 y sus tratamientos.

2.2 *Técnicas de clasificación de imágenes utilizando redes neuronales*

Objetivos perseguidos: Estudiar y presentar el estado del arte actual respecto a las tecnologías de reconocimiento de imágenes mediante redes neuronales artificiales, de tipo

CNN, y su aplicación al reconocimiento de infección por el virus SARS-CoV-2 basadas en imágenes de tórax tomadas con Rayos-X.

2.3 *Diagnóstico de COVID-19 mediante técnicas de análisis de imagen con redes neuronales*

Objetivos perseguidos: Estudiar y presentar el estado del arte actual respecto a los métodos de detección de la infección por el virus SARS-CoV-2 basadas en Rayos-X.

Capítulo 4. Descripción detallada del experimento

4.1 *Descripción general de las contribuciones*

Objetivos perseguidos: Exponer la generalidad de los objetivos marcados, las plataformas tanto Hardware como Software empleadas y el desarrollo de los supuestos planteados.

4.3 *Entrenamiento y pruebas en equipo sobremesa*

Objetivos perseguidos: Describir las tareas realizadas en cuanto la adecuación del dataset y algoritmo, así como el entrenamiento y las pruebas realizadas en el equipo de sobremesa.

Capítulo 5. Desarrollo y contribuciones del trabajo

5.1 *Entrenamiento y pruebas en equipo sobremesa*

Objetivos perseguidos: Datos y resultados relevantes de la selección de dataset y los entrenamientos y pruebas realizadas en el equipo de sobremesa.

Anexo 1. Repositorio de código fuente

Anexos A1.1, A1.2.1

Objetivos perseguidos: Registro del código y dataset utilizados.

Anexo. Artículo de investigación

Objetivos perseguidos: Adecuar el proyecto y la documentación desarrollada al formato de artículo científico.

---

1. Introducción

Una situación de pandemia como la actual pone aún más de manifiesto las limitaciones en los campos relacionados con la tecnología, que además cobra una importancia mucho mayor si cabe. Es necesario maximizar el rendimiento de los equipos hardware existentes a la vez que se busca una optimización de costes, lo que añadido a la falta de plataformas de cálculo potentes disponibles, hace que sea interesante el trabajar con tecnologías de clústering que permitan distribuir el trabajo y escalar según necesidades para reaprovechar al máximo posible los recursos sin una alta inversión económica.

Además de lo anterior, la tendencia a depender de sistemas basados en la nube (donde las tecnologías de clústering como Kubernetes son de uso bastante habitual) hace que para determinadas situaciones donde la conectividad, tiempos de respuesta o movimientos de datos personales sean un problema (por ejemplo, entornos hospitalarios) se barajen otras alternativas como los sistemas de computación en Edge.

Por otra parte, a la hora de buscar un caso de uso aplicado dentro de la Inteligencia Artificial para probar esta tecnología, consideramos que el COVID-19 debía estar presente en este trabajo, no sólo por ser el primer problema de salud mundial actual, sino además por formar parte de una de las iniciativas de la Comisión Europea más importantes en relación con la Inteligencia Artificial. Por esta razón empezamos a trabajar en las posibilidades relacionadas con el COVID-19, donde dimos con un caso aplicado lo suficientemente interesante y complejo para contemplar: El diagnóstico de COVID-19 mediante imágenes de placas de tórax obtenidas por Rayos X.

La elección del caso de uso y el contexto de la Comisión Europea sumado a disponer inicialmente de conjuntos de datos y algoritmos necesarios para plantearlo (sobre todo el *dataset* de imágenes de libre acceso de Joseph Paul Cohen, Paul Morrison y Lan Dao) creemos que justifica tanto la elección del caso de uso como de las tecnologías a utilizar.

Una vez planteado, pensábamos en el reto tecnológico que suponía de cara al ámbito de la Inteligencia Artificial y al hardware necesario el realizar un caso de uso como este. Fuimos pensando en las limitaciones de los sistemas actuales en un entorno hospitalario y la posibilidad de crear un sistema dedicado que solucionara este tipo de problemas además de plantear su viabilidad y escalabilidad según el problema a abordar, pero sin entrar en desarrollos a medida de clústering que no pudieran ser gestionados de manera eficiente, sino que fueran lo más estandarizados posibles.

Además de validad el diseño y creación de una solución así, sería importante tener una comparativa con otro tipo de soluciones para poder contrastar y comprobar su eficiencia. El conjunto de la tecnología planteada a nivel de arquitectura hardware (Sistemas embebidos, interconexión en clúster, tarjetas nVidia con soporte de GPU), arquitectura software (Clústering, Contenedores, Sistemas GNU/Linux Embebidos), el caso de uso (Inteligencia Artificial, Visión computacional, CNNs, COVID-19) y el producto en sí (Edge, sin dependencias de Cloud, pequeño y de bajo coste) creemos que es suficientemente novedoso y útil como para realizar esta prueba de concepto.

De cara a los resultados esperados, contamos con poder evaluar el rendimiento del sistema tanto en el entrenamiento como en la inferencia para el diagnóstico de resultados, realizando una comparativa de los resultados con otros equipos dedicados, y mostrando los problemas encontrados junto con las soluciones que hayamos podido encontrar para este reto.

1.1 Acerca del SARS-CoV-2

La actual pandemia mundial viene derivada de la enfermedad COVID-19, comúnmente abreviada como COVID, una enfermedad infecciosa provocada por el virus SARS-CoV-2, un coronavirus de tipo 2.

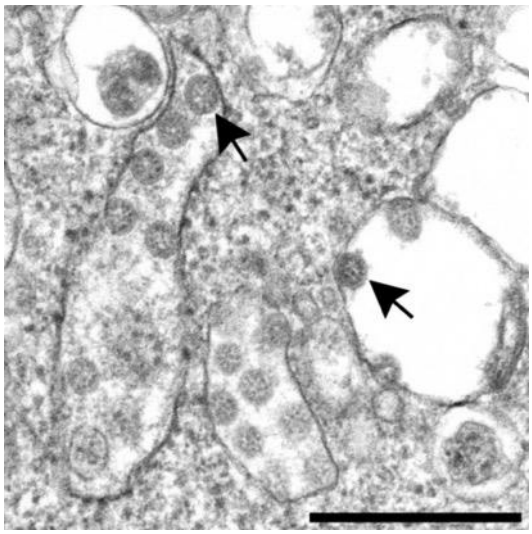


Figura 1. Imagen del Coronavirus SARS-CoV-2 (S. E. Miller & Goldsmith, 2020)

Aislamiento de coronavirus desarrollado en cultivo celular que muestra numerosas partículas virales esféricas (flechas) que se encuentran en las cisternas del retículo endoplásmico rugoso / área del complejo de Golgi de la célula. Note los puntos negros en el interior de las partículas, que son cortes transversales a través de la nucleocápside viral. Escala: 400 nm. (S. E. Miller & Goldsmith, 2020)

Estos coronavirus son una gran familia de virus ARN monocatenarios que pueden infectar tanto a humanos como animales, provocando enfermedades de tipo respiratorio, enfermedades gastrointestinales, hepáticas y neurológicas (Weiss y Leibowitz, 2013).

Los coronavirus se subdividen en cuatro clases: Alpha, Beta, Gamma y Delta (Yang y Leibowitz, 2015). Dentro de estos tipos conocíamos 6 que afectaban a los humanos:

- alpha - CoVs HCoVs-NL63
- alpha HCoVs-229E
- beta CoVs HCoVs-OC43
- beta HCoVs-HKU1
- Síndrome respiratorio agudo severo-CoV (SARS-CoV) (Drosten et al., 2020)

- Síndrome respiratorio de Oriente Medio-CoV (MERS-CoV) (Zaki et al., 2012)

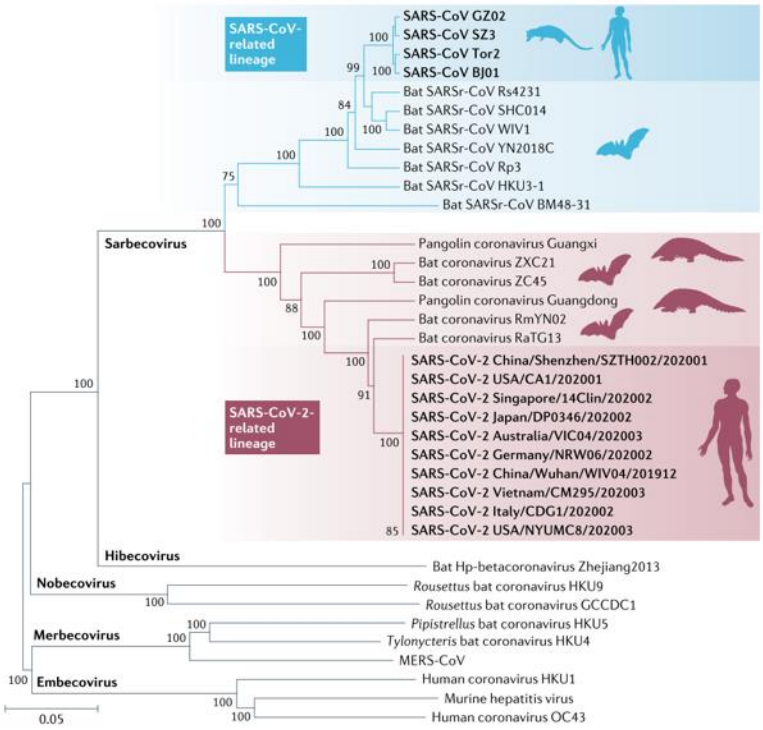


Figura 2. Phylogenetic tree of the full-length genome sequences of SARS-CoV-2, SARS-CoVs and other betacoronaviruses. (Hu et al., 2021)

Durante el año 2019 se detectó en el mercado de marisco de Wuhan en la provincia de Hubei (China) una nueva variante denominada SARS CoV-2, causante de la enfermedad COVID-19 que más tarde fue declarada como pandemia mundial.

1.2 Diagnóstico del SARS-CoV-2 mediante Rayos-X

Durante el año 2019 el patógeno SARS-CoV-2 fue el responsable de un grupo de casos de neumonía asociados con una enfermedad respiratoria grave conocida como Enfermedad por Coronavirus 2019 (COVID-19).

Al ser una enfermedad que ataca al sistema respiratorio, inicialmente se realizaron estudios para intentar realizar un diagnóstico de la enfermedad mediante radiografías de tórax (CXR), pero los resultados no fueron lo suficientemente satisfactorios. Posteriormente se realizaron estudios basados en tomografías de tórax (TAC), con imágenes de mucha mayor resolución y más detalladas, dando lugar a mejores resultados de cara a la detección de las anomalías

pulmonares en el estado temprano de la enfermedad, su progresión y su evaluación cuantitativa.

Uno de los estudios más relevantes al respecto fue el realizado por Borghesi & Maroldi, donde diseñaron un sistema de valoración de casos verificados y hospitalizados por SARS-CoV-2, subdividiendo el área pulmonar en la tomografía resultante en seis sub-áreas.

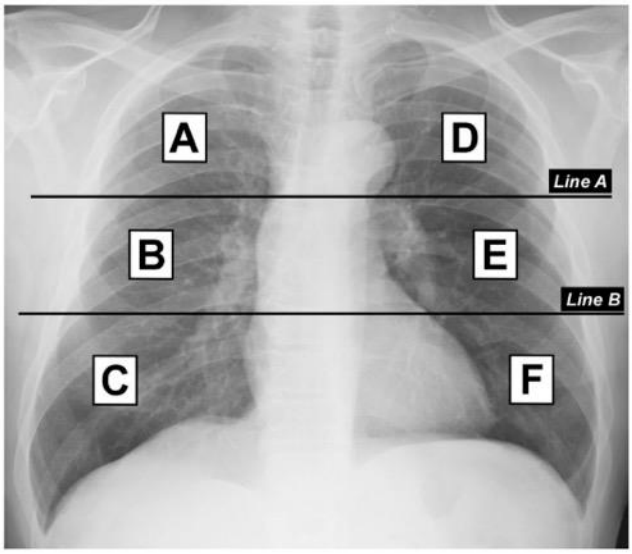


Figura 3. División de pulmones en seis zonas en la radiografía frontal de tórax. (Borghesi & Maroldi, 2020)

La línea A se traza al nivel de la pared inferior del arco aórtico. La línea B se traza al nivel de la pared inferior de la vena pulmonar inferior derecha. Zonas superiores A y D; Zonas medias B y E; Zonas inferiores C y F. (Borghesi & Maroldi, 2020)

Esta división permite realizar una clasificación más sencilla de las zonas afectadas por la enfermedad, centrando el estudio en los seis puntos distribuidos de la imagen.

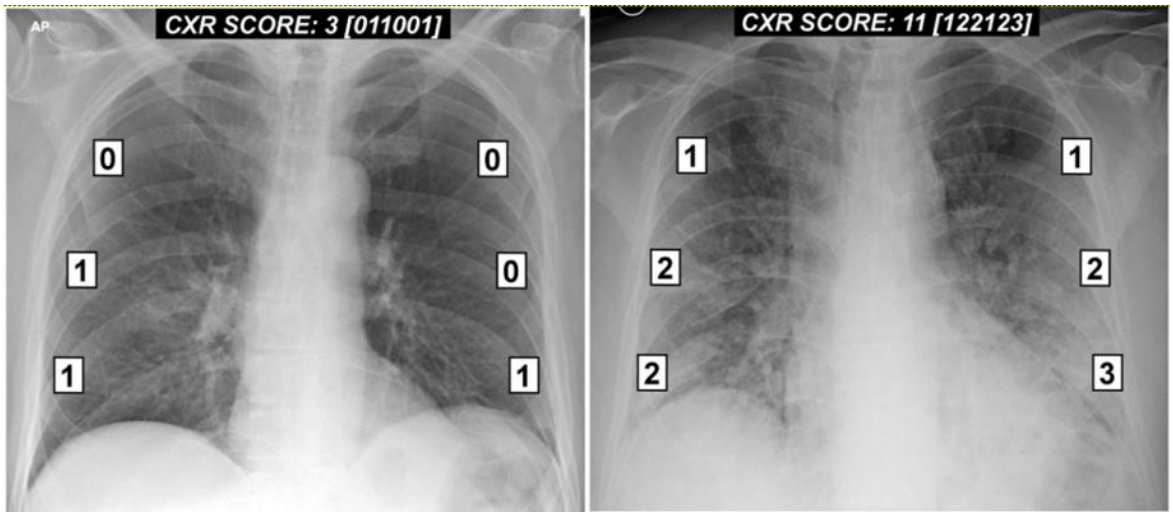


Figura 4. Ejemplos del sistema de puntuación de rayos X de tórax en dos pacientes con neumonía COVID-19 Intersticial. (Borghesi & Maroldi, 2020)

Dentro de la iniciativa de la Comisión Europea “AI-ROBOTIS vs COVID-19” una gran parte de la comunidad científica decide aplicar los conocimientos y resultados de estos estudios para plantear el análisis de radiografías mediante redes neuronales convolucionales (CNN) e intentar lograr un modelo clasificatorio para diagnóstico de COVID-19.

1.3 Reconocimiento de imágenes – CNNs

Las redes neuronales son sistemas que se emplean para modelar las relaciones existentes entre un conjunto de datos de entrada (también llamados datos de origen o estímulos) y unos datos de salida (también llamados resultados o productos).

Este tipo de sistemas utilizan una red interconectada de neuronas artificiales llamadas nodos, donde mediante un proceso de entrenamiento se van ajustando los valores de estas neuronas hasta poder aproximar los resultados deseados.

Son muy utilizados en el contexto de *Machine Learning* o Aprendizaje Automático ya que este tipo de sistemas son aproximadores universales, capaces de computar cualquier función matemática.

Las características principales de las redes neuronales son su arquitectura o topología (la manera en la que las neuronas están interconectadas entre sí), las funciones de activación que utilizan y el proceso de entrenamiento utilizado con la red.

Entre los objetivos de utilizar este tipo de sistemas estarían:

- Utilizar un sistema de red neuronal CNN-LSTM combinada para poder realizar el diagnóstico de COVID-19 en pacientes de manera automática y eficiente.
- Utilizar un conjunto de datos de radiografías de tórax en alta definición que permitan entrenar al sistema con casos positivos y negativos.
- Proporcionar un análisis experimental detallado con métricas de precisión, sensibilidad, especificidad y puntuación F1 además de una matriz de confusión y AUC utilizando la característica operativa del receptor (ROC) para con todo esto poder medir el rendimiento del sistema propuesto.

1.4 TensorFlow

Creada en 2015 por ingenieros de Google Brain, TensorFlow es una plataforma de código fuente abierto para realizar tareas de *Machine Learning* de manera distribuida a escala. Está disponible para múltiples plataformas tanto a nivel de *hardware* (CPUs, GPUs y aceleradores como TPUs) como de sistemas operativos (Linux, Windows, macOS y plataformas móviles como Android e iOS) y *software* (Python, C++, Java...)

1.5 Infraestructura en EDGE

1.5.1 Hardware EDGE

Para realizar nuestra prueba de concepto vamos a utilizar un *hardware* actual de bajo coste y muy utilizado para proyectos relacionados con robótica y computación en Edge por sus características: los kits de desarrollo Jetson Nano, del fabricante nVidia.

Figura 5. Imagen SBC nVidia Jetson Nano

Este hardware tiene características que son interesantes de cara a nuestra prueba de concepto, como son:

- Arquitectura de procesador ARM de 64 bits (ARM64)
- 4GB de RAM
- Soporte de GPU para aceleración gráfica y aceleración de tareas relacionadas con Inteligencia Artificial y redes neuronales (Deep Learning)
- Bajo consumo (5V/4A a máxima potencia 5V/2.5A en modo baja potencia)
- BSP de nVidia con sistema operativo GNU/Linux y las bibliotecas necesarias para desarrollar proyectos de IA (CUDA, TensorFlow…)

Para realizar nuestra prueba de concepto utilizaremos dos placas de este tipo en configuración de clúster utilizando Docker y Kubernetes.

1.5.2 Docker

Utilizaremos imágenes Docker para desplegar los contenedores en el clúster. Docker es una tecnología de código fuente abierto dependiente de la infraestructura del *Kernel* Linux que permite crear, desplegar y lanzar aplicaciones de manera aislada y donde cada contenedor tiene todas las dependencias necesarias encapsuladas e integradas de manera sencilla, lo que facilita su creación, despliegue y mantenimiento. Estas características las hacen ideales para poder ser usadas en el contexto de un clúster como el que queremos realizar en nuestra prueba de concepto para evitar problemas de dependencias, tanto para realizar entrenamiento distribuido entre nodos como inferencia.

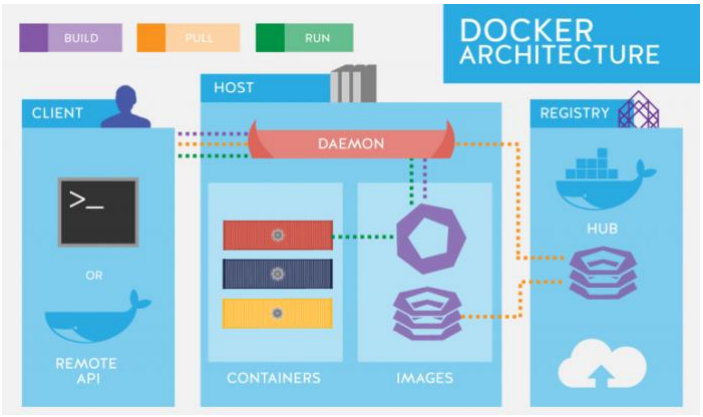


Figura 6. Imagen resumen arquitectura Docker (nVidia, n.d.)

1.5.3 Kubernetes

Kubernetes es un software de orquestación de contenedores desarrollado inicialmente como un proyecto interno de Google llamado Borg que se ha convertido en el estándar de facto de gestión de contenedores de forma distribuida. Este software permite la gestión de clústeres de contenedores para gestionar el ciclo de vida de éstos de manera sencilla, pudiendo ser utilizado tanto para realizar el entrenamiento del sistema como la inferencia del modelo.

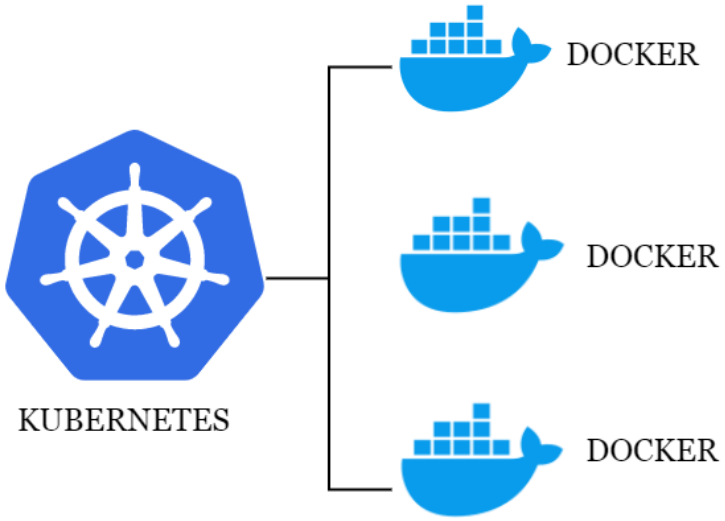


Figura 7. Esquema general de Kubernetes con Dockers

1.6 Nuestra prueba de concepto

Mediante el uso de las tecnologías y medios expuestos en los apartados anteriores podemos demostrar la viabilidad de realizar una prueba de concepto de un producto basado en un clúster escalable de tarjetas embebidas de este tipo, incrementando el rendimiento de los algoritmos de Inteligencia Artificial utilizados, y que a su vez permite respecto a soluciones más tradicionales reducir el consumo energético, menor coste, no tener dependencias externas y, ante un caso de uso como el elegido sobre detección de COVID-19, el poder utilizarse en entornos protegidos o limitados como centros de salud u hospitales.

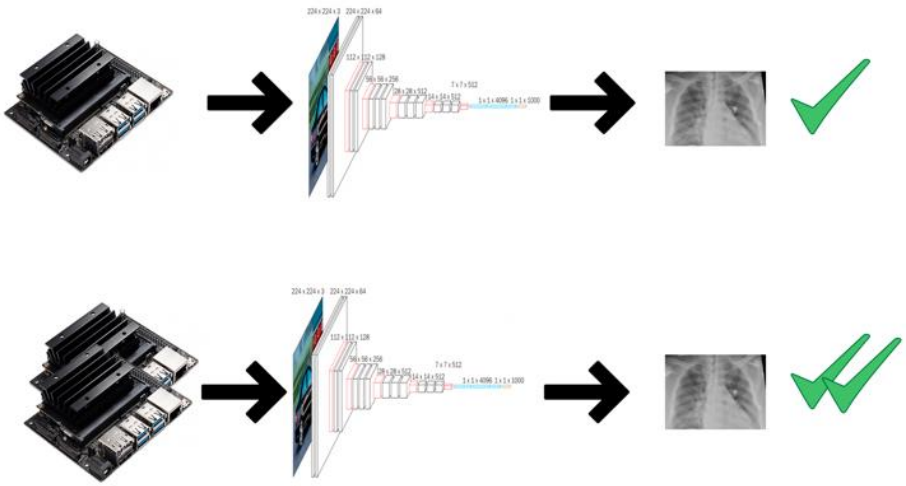


Figura 8. Diagrama de concepto del TFM

2. Contexto y estado del arte

En este capítulo se muestra una visión detallada de los casos, referencias, investigaciones publicadas y otros trabajos más relevantes a día de hoy relacionados con el proyecto realizado.

2.1 Sobre la investigación del SARS-CoV-2

El estado actual de los estudios centrados en la pandemia debida al coronavirus SARS-CoV-2 están englobados en las siguientes tres vertientes:

1. **Estudios generales** que describen de forma breve pero detallada la epidemiologia, los posibles anfitriones, la transmisión, manifestaciones clínicas, periodo de incubación, síntomas, diagnostico, y tratamientos y prevenciones, como se detallan en estudio de El brote de SARS-CoV-2: lo que sabemos (Wu et al., 2020)

   En este estudio se definen los siguientes apartados:

- Epidemiologia: Centra el origen de una inexplicable neumonía en el mercado de pescado de Wuhan causado por un joven y desconocido tipo de coronavirus, detallando la afección inicial en 31 provincias de China y con una valoración de los casos existente hasta el momento del estudio con la relación de infectados, sospechosos, muertos y recuperados.

- Posibles anfitriones: Los animales salvajes y los murciélagos son posibles anfitriones del virus, según indican los estudios relacionados con otros tipos de coronavirus.

- Transmisión: Confirma que la transmisión entre humanos es patente desde un huésped inicial que contrajo la enfermedad a partir de un animal. La sintomatología de afecciones respiratorias apunta a una transmisión aérea del virus, pero la aparición en las heces de pacientes del virus SARS-CoV-2 apuntan también a una posible transmisión fecal-oral, aunque aún no se ha comprobado que la ingesta de alimentos en contacto con el virus pueda producir su transmisión.

  También se ha confirmado un caso de transmisión neonatal entre madre e hijo, que s e encuentra en estudio.

  A día cinco de junio los casos totales de infectados ascienden a 173,483,489 (Worldometer, 2019)

- Manifestaciones clínicas: las diversas manifestaciones de la enfermedad no se detallan en el estudio, si bien refleja que la mayoría de los pacientes con SARS-CoV-2 tiene una gravedad entre normal y leve, y que la mortalidad es menor que en el SARS-CoV y MERS-CoV.
- Periodo de incubación: El estudio refleja un periodo de entre 5 y 8 días de incubación, con una media final de los casos estudiados de 5.2 días.
- Sintomatología: Presenta desde pacientes completamente asintomáticos hasta síntomas como:
    - Fiebre (98%).
    - Tos (76%).
    - Mialgia o fatiga (44%)
    - Esputo (28%)
    - Dolor de cabeza (8%)
    - Hemoptisis (5%)
    - Diarrea (3%)
    - Disnea (50%)
    - Linfocitopenia (63%)
    - Neumonía
        - Síndrome de dificultad respiratoria aguda (29%)
        - Lesión cardíaca aguda (12%)
        - Infecciones secundarias (10%);

    El 32% de los pacientes ingresó en la UCI y la mayoría de los afectados tenían entre 30 y 79 años (86,6%).
- Diagnóstico: La fiebre alta, tos seca y disnea son los principales indicadores de la infección por SARS-CoV2.

    Un examen físico puede no ser concluyente a la hora de detectar la infección.

Un estudio de radiografía de tórax es mucho más preciso a la hora de detectar la enfermedad.

Las pruebas específicas de laboratorio en última instancia ayudan a la confirmación de diagnóstico.

- Tratamientos y prevenciones: Actualmente, no existe ninguna vacuna 100% efectiva o tratamiento antiviral para humanos, si bien es cierto que siguen en estudio diferentes vacunas y sus correspondientes ensayos con resultados esperanzadores.

  Tanto en humanos como animales, se han tratado de hallar las mejores opciones de tratamiento farmacológico con el fin de aliviar o suprimir las disfunciones y enfermedades que el virus SARS-CoV-2 provoca en los diferentes casos. Actualmente están disponibles diferentes versiones de vacunas aprobadas por la OMS y se encuentran en desarrollo otras variables de vacunas. La base principal para el tratamiento clínico se centra en el tratamiento sintomático adaptado a cada paciente, incluyendo el soporte orgánico en UCI (cuidados intensivos) en casos graves de la enfermedad. Las medidas más extendidas como tratamiento en casos menos graves son:

  - Reposo en cama
  - Tratamientos de apoyo
    - Terapia antiviral (Arabi et al., 2018),
    - Aplicación de antibióticos
    - Terapia inmunomoduladora (Arabi et al., 2020)
    - Soporte de la función orgánica
    - Soporte respiratorio
    - Lavado bronco-alveolar (BAL)
    - Purificación y oxigenación de la sangre por membrana extracorpórea (ECMO) (Hong et al., 2020).

  La infección por el nuevo coronavirus es una nueva enfermedad transmisible con un brote emergente que afecta a todas las poblaciones (Burki, 2019). Esta infección ha sido clasificada por la OMS como enfermedad infecciosa de categoría B, pero ciertos países incluyendo China la han tratado como si fuese una enfermedad infecciosa de mayor categoría, es decir de tipo A.

Es de principal importancia disponer y/o redactar las normativas y manuales de actuación ante este tipo de enfermedades infecciosas de rápido despliegue, para controlar tanto el origen de la misma como poder acotar y restringir las fuentes de transmisión, en pro de proteger a la población en general, y en particular a aquellas poblaciones más susceptibles a la enfermedad.

La OMS, así como otras entidades y organismos relacionados con la salud pública, en una labor sin precedentes han centrado todos sus esfuerzos en la prevención de la transmisión del virus tanto de forma local como en la detección en los viajeros, y en actualizar las medidas para el control de infecciones. (Hui et al., 2020)

2. Estudios sobre la **composición del virus**, partes que lo conforman estructura química y detalles.

3. Por último, y quizás la más importante a día de hoy (aunque no para el desarrollo de este proyecto), se encuentran las **investigaciones sobre las vacunas** tanto a nivel de desarrollo y mejora de las mismas, como los estudios realizados sobre su efectividad y distribución.

En cuanto a la investigación sobre la creación y mejora de las vacunas, tenemos el estudio realizado por (Amanat & Krammer, 2020) con la siguiente clasificación en función de su producción:

Platform	Target	Existing, Licensed Human Vaccines Using the Same Platform	Advantages	Disadvantages
RNA vaccines	S protein	No	No infectious virus needs to be handled, vaccines are typically immunogenic, rapid production possible.	Safety issues with reactogenicity have been reported.
DNA vaccines	S protein	No	No infectious virus needs to be handled, easy scale up, low production costs, high heat stability, tested in humans for SARS-CoV-1, rapid production possible.	Vaccine needs specific delivery devices to reach good immunogenicity.
Recombinant protein vaccines	S protein	Yes for baculovirus (influenza, HPV) and yeast expression (HBV, HPV)	No infectious virus needs to be handled, adjuvants can be used to increase immunogenicity.	Global production capacity might be limited. Antigen and/or epitope integrity needs to be confirmed. Yields need to be high enough.
Viral vector-based vaccines	S protein	Yes for VSV (Ervebo), but not for other viral vectored vaccines	No infectious virus needs to be handled, excellent preclinical and clinical data for many emerging viruses, including MERS-CoV.	Vector immunity might negatively affect vaccine effectiveness (depending on the vector chosen).
Live attenuated vaccines	Whole virion	Yes	Straightforward process used for several licensed human vaccines, existing infrastructure can be used.	Creating infectious clones for attenuated coronavirus vaccine seeds takes time because of large genome size. Safety testing will need to be extensive.
Inactivated vaccines	Whole virion	Yes	Straightforward process used for several licensed human vaccines, existing infrastructure can be used, has been tested in humans for SARS-CoV-1, adjuvants can be used to increase immunogenicity.	Large amounts of infectious virus need to be handled (could be mitigated by using an attenuated seed virus). Antigen and/or epitope integrity needs to be confirmed.

Figura 9. Descripción general de las plataformas y tecnologías de producción de vacunas para la plataforma SARS-CoV-2 (Amanat & Krammer, 2020)

En cuanto a la distribución y efectividad de las misma, y ya más enfocada en las marcas comerciales de las mismas tenemos el estudio de (Creech et al., 2021) el cual se puede resumir la siguiente tabla:

Table. SARS-CoV-2 Vaccines

Vaccine	Manufacturer	Vaccine type	Antigen	Dose	Dosage	Storage conditions	Efficacy against severe COVID-19[a]	Overall efficacy	Current approvals
mRNA-1273	Moderna (US)	mRNA	Full-length spike (S) protein with proline substitutions	100 µg	2 Doses 28 d apart	−25° to −15 °C; 2-8 °C for 30 d; room temperature ≤12 h	100% 14 d After second dose (95% CI, not estimable to 1.00)	92.1% 14 d After 1 dose (95% CI, 68.8%-99.1%); 94.1% 14 d after second dose (95% CI, 89.3%-96.8%)	EUA: the US, EU, Canada, and UK
BNT162b2	Pfizer-BioNTech (US)	mRNA	Full-length S protein with proline substitutions	30 µg	2 Doses 21 d apart	−80° to −60 °C; 2-8 °C for 5 d; room temperature ≤2 h	88.9% After 1 dose (95% CI, 20.1%-99.7%)	52% After 1 dose (95% CI, 29.5%-68.4%); 94.6% 7 d after second dose (95% CI, 89.9%-97.3%)	EUA: the US, EU, Canada, and UK
Ad26.CoV2.S	Janssen/ Johnson & Johnson (US)	Viral vector	Recombinant, replication-incompetent human adenovirus serotype 26 vector encoding a full-length, stabilized SARS-CoV-2 S protein	5×10^{10} Viral particles	1 Dose	−20 °C; 2-8 °C for 3 mo	85% After 28 d; 100% after 49 d	72% in the US; 66% in Latin America; 57% in South Africa (at 28 d)	EUA: the US, EU, and Canada
ChAdOx1 (AZS1222)	AstraZeneca/ Oxford (UK)	Viral vector	Replication-deficient chimpanzee adenoviral vector with the SARS-CoV-2 S protein	5×10^{10} Viral particles (standard dose)	2 Doses 28 d apart (intervals >12 wk studied)	2-8 °C for 6 mo	100% 21 d After first dose	64.1% After 1 dose (95% CI, 50.5%-73.9%); 70.4% 14 d after second dose (95% CI, 54.8%-80.6%)	EUA: WHO/Covax, the UK, India, and Mexico
NVX-CoV2373	Novavax, Inc (US)	Protein subunit	Recombinant full-length, prefusion S protein	5 µg of protein and 50 µg of Matrix-M adjuvant	2 Doses	2-8 °C for 6 mo	Unknown	89.3% in the UK after 2 doses (95% CI, 75.2%-95.4%); 60% in South Africa (95% CI, 19.9%-80.1%)	EUA application planned
CVnCoV	CureVac/ GlaxoSmithKline (Germany)	mRNA	Prefusion stabilized full-length S protein of the SARS-CoV-2 virus	12 µg	2 Doses 28 d apart	2-8 °C for 3 mo; room temperature for 24 h	Unknown	Phase 3 trial ongoing	
Gam-COVID-Vac (Sputnik V)	Gamaleya National Research Center for Epidemiology and Microbiology (Russia)	Viral vector	Full-length SARS-CoV-2 glycoprotein S carried by adenoviral vectors	10^{11} Viral particles per dose for each recombinant adenovirus	2 Doses (first, rAd26; second, rAd5) 21 d apart	−18 °C (Liquid form); 2-8 °C (freeze dried) for up to 6 mo	100% 21 d After first dose (95% CI, 94.4%-100%)	87.6% 14 d After first dose (95% CI, 81.1%-91.8%); 91.1% 7 d after second dose (95% CI, 83.8%-95.1%)	EUA: Russia, Belarus, Argentina, Serbia, UAE, Algeria, Palestine, and Egypt
CoronaVac	Sinovac Biotech (China)	Inactivated virus	Inactivated CN02 strain of SARS-CoV-2 created from Vero cells	3 µg With aluminum hydroxide adjuvant	2 Doses 14 d apart	2-8 °C; Lifespan unknown	Unknown	Phase 3 data not published; reported efficacy 14 d after dose 2: 50.38% (mild) and 78% (mild to severe) in Brazil, 65% in Indonesia, and 91.25% in Turkey	EUA: China, Brazil, Columbia, Bolivia, Brazil, Chile, Uruguay, Turkey, Indonesia, and Azerbaijan
BBIBP-CorV	Sinopharm 1/2 (China)	Inactivated virus	Inactivated HB02 strain of SARS-CoV-2 created from Vero cells	4 µg With aluminum hydroxide adjuvant	2 Doses 21 d apart	2-8 °C; Lifespan unknown	Unknown	Phase 3 data not published; unpublished reports of 79% and 86% efficacy	EUA: China, UAE, Bahrain, Serbia, Peru, and Zimbabwe

Abbreviations: EUA, Emergency Use Authorization; UAE, United Arab Emirates; WHO, World Health Organization.

[a] Efficacy against severe disease, which includes COVID-19–related hospitalization, varies by age and by time after vaccination.

Figura 10. Estudio de vacunas por nombre para SARS-CoV-2 (Creech et al., 2021)

2.2 Técnicas de clasificación de imágenes utilizando redes neuronales convolucionales

El uso de redes neuronales para realizar tareas de clasificación de imágenes es uno de los campos que más ha crecido en los últimos años, sobre todo siendo muy relevantes en campos de la medicina actual como detección de enfermedades, además de tener una gran repercusión mediática en el año actual en el que estamos desarrollando este proyecto.

Para nuestro caso nos basamos en el uso de arquitecturas de redes neuronales convolucionales para clasificación de imágenes de ImageNet (Krizhevsky et al., 2017), cuya red sigue el esquema representado en la siguiente figura:

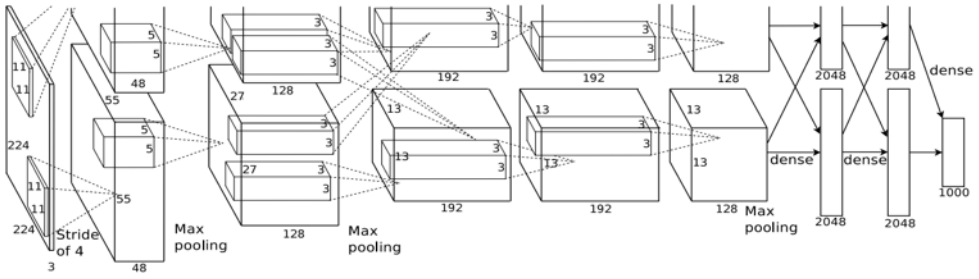


Figura 11. Arquitectura CNN utilizada sobre el dataset ImageNet (Krizhevsky et al., 2017)

El dataset ImageNet (Deng et al., 2009) es uno de los más utilizados actualmente para tareas de visión computacional basadas en redes neuronales. Consta de un conjunto de imágenes agrupadas de manera jerárquica con la misma estructura que la base de datos léxica WordNet (G. Miller et al., 1991) (se ha construido con la misma base que ésta) lo que permite ofrecer un enorme conjunto de 3,2 millones de imágenes clasificadas y etiquetadas que son muy utilizadas como datos de entrenamiento para tareas de reconocimiento y agrupación de objetos similares junto a clasificación de objetos.

La arquitectura tiene como características principales el uso de dos GPUs para acelerar las operaciones matriciales de convolución 2D en la arquitectura: Una GPU ejecuta la sección superior de la arquitectura y la segunda GPU ejecuta la inferior, pero la comunicación entre éstas se da únicamente en ciertas capas de la red.

La red neuronal completa tiene como características el tener 150.528 dimensiones de entrada a la red, mientras que el número de neuronas de las siguientes capas es el siguiente:

- 253
- 440-186
- 624-64
- 896-64
- 264-4096-4096-1000

En cuanto al uso de las capas convolucionales, la segunda capa convolucional tiene como entrada la respuesta normalizada y agrupada de la primera capa convolucional y la filtra con 256 núcleos de tamaño 5 × 5 × 48.

La tercera, cuarta y quinta capas convolucionales están conectadas entre sí sin ninguna intervención en cuanto a capas de agrupación o normalización. La tercera capa convolucional tiene 384 granos de tamaño 3 × 3 × 256 conectados a las salidas (normalizadas, agrupadas)

de la segunda capa convolucional. La cuarta capa convolucional tiene 384 granos de tamaño 3 × 3 × 192, y la quinta capa convolucional tiene 256 cuadros de tamaño 3 × 3 × 192. Las capas completamente conectadas tienen 4096 neuronas cada una.

En base a esta arquitectura de Krizhevsky et al., 2017 se han ido desarrollado nuevas arquitecturas de redes neuronales optimizadas para casos de usos más concretos (y que aún a día de hoy continúan evolucionando).

Debido a las restricciones en nuestro proyecto finalmente optamos por una arquitectura convolucional denominada VGG16, entrenada previamente con el mismo conjunto de datos de ImageNet y que está disponible en el framework Keras para poder utilizarse de manera directa o bien para realizar tareas de reentrenamiento con otro subconjunto de datos mediante técnicas de Transfer Learning.

Uno de los estudios de partida para el algoritmo empleado en nuestro proyecto es el trabajo realizado por (Sitaula & Hossain, 2021) que parte de la estructura de capas original propuesta en VGG, mostrada en la siguiente tabla:

Input size	Output size	Layer	Stride	Kernel
224× 224 × 3	224 × 224 × 64	conv1-64	1	3 × 3
224× 224 × 64	224 × 224 × 64	conv1-64	1	3 × 3
224× 224 × 64	112 × 112 × 64	maxpool	2	2 × 2
112×112 × 64	112 × 112 × 128	conv2-128	1	3 × 3
112× 112 × 128	112 × 112 × 128	conv2-128	1	3 × 3
112× 112 × 128	56 × 56 × 128	maxpool	2	2 × 2
56× 56 × 128	56 × 56 × 256	conv3-256	1	3 × 3
56× 56 × 256	56 × 56 × 256	conv3-256	1	3 × 3
56× 56 × 256	56 × 56 × 256	conv3-256	1	3 × 3
56× 56 × 256	28 × 28 × 256	maxpool	2	2 × 2
28× 28 × 256	28 × 28 × 512	conv4-512	1	3 × 3
28× 28 × 512	28 × 28 × 512	conv4-512	1	3 × 3
28× 28 × 512	28 × 28 × 512	conv4-512	1	3 × 3
28× 28 × 512	14 × 14 × 512	maxpool	2	2 × 2
14× 14 × 512	14 × 14 × 512	conv5-512	1	3 × 3
14× 14 × 512	14 × 14 × 512	conv5-512	1	3 × 3
14× 14 × 512	14 × 14 × 512	conv5-512	1	3 × 3
14× 14 × 512	7 × 7× 512	maxpool	2	2 × 2
1× 1 × 25088	1 × 1 × 4096	fc	–	1× 1
1× 1 × 4096	1 × 1 × 4096	fc	–	1 × 1
1× 1 × 4096	1 × 1 × 1000	fc	–	1 × 1

Figura 12. Parámetros originales del modelo VGG-16 (Krizhevsky et al., 2017)

En este estudio se realizan diferentes modificaciones y adaptaciones de las capas de la arquitectura de la red junto con diferentes experimentos de análisis de imagen para clasificar afecciones pulmonares, obteniendo como resultado mejoras en los métodos utilizados anteriormente para este tipo de tareas.

Si bien durante este último año se han visto grandes avances en este campo de la medicina gracias a este tipo de técnicas, la evolución y optimización de este tipo de redes neuronales han llevado a estudios muy relevantes también en otros campos médicos como el reconocimiento de emociones (Cheah et al., 2021), la clasificación y detección de anomalías en los huesos (El-Saadawy et al., 2021) e incluso la detección del cáncer de mama (Jahangeer & Rajkumar, 2021).

2.3 Diagnóstico de COVID-19 mediante técnicas de análisis de imagen con redes neuronales

Desde los inicios de la pandemia de COVID-19, según han ido evolucionando las técnicas de reconocimiento de imágenes anteriores, han ido aplicándose para entrenar nuevos modelos de redes neuronales para detección de casos de COVID-19 (como por ejemplo utilizando imágenes de Rayos-X en alta resolución).

De hecho, si nos centramos en los primeros estudios de referencia desde el comienzo de la pandemia, como puede ser el caso del estudio "*Enfoque automatizado basado en el aprendizaje de transferencia profunda para la detección de la infección por COVID-19 en la radiografía de tórax"* (Narayan Das et al., 2020) podemos ver una primera aproximación a la detección de neumonías causadas por el virus, siguiendo el esquema de detección siguiente:

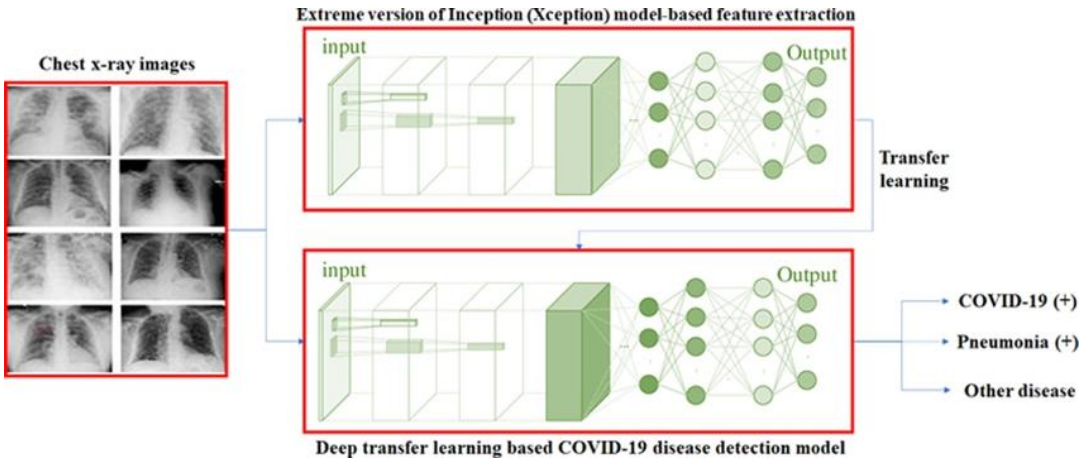


Figura 13. Esquema de detección (Narayan Das et al., 2020)

En estudios más recientes como es el caso de "*Detección automática de la enfermedad por coronavirus (COVID-19) mediante imágenes de Rayos-X y redes neuronales convolucionales profundas*" (Narin et al., n.d.) se han realizado pruebas con diferentes arquitecturas de redes convolucionales :

- ResNet50
- ResNet101
- ResNet152
- InceptionV3
- Inception-ResNetV2

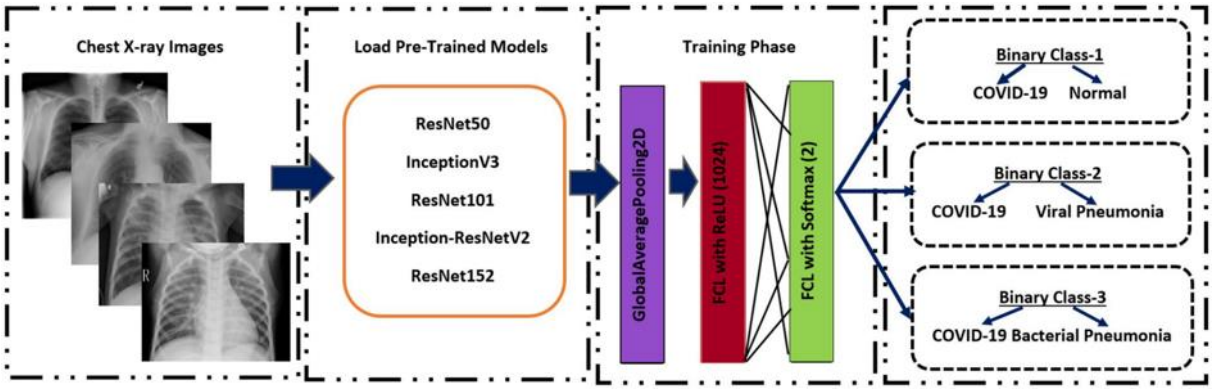


Figura 14. Esquema de modelos pre - entrenados para la predicción de pacientes con: neumonía normal (sana), COVID-19, bacteriana y Viral (Narin et al., n.d.)

En este estudio se emplean tres clasificaciones binarias diferentes con cuatro clases ("COVID-19", "normal" (saludable), "neumonía viral" y "neumonía bacteriana") mediante el uso de una validación cruzada de cinco iteraciones.

Los resultados del estudio determinan que el modelo de aprendizaje residual ResNet50 (He et al., n.d.) es el modelo que proporciona mejor rendimiento. En nuestro caso al tener una limitación de hardware tan estricta, optamos como comentamos anteriormente por el modelo de arquitectura VGG16 para nuestro trabajo.

El esquema de los resultados del estudio puede verse en el siguiente cuadro, donde aparecen los estudios previos con las arquitecturas de CNNs correspondientes, las clases y el factor de precisión conseguido:

Previous study	Data type	Methods/classifier	Number of classes	Accuracy (%)
Narayan Das et al. [9]	X-ray	Xception	3	97.40
Singh et. al. [52]	X-ray	MADE-based CNN	2	94.65 ∓ 2.1
Afshar et al. [30]	X-ray	Capsule Networks	4	95.7
Ucar and Korkmaz [24]	X-ray	Bayes-SqueezeNet	3	98.26
Khan et al. [21]	X-ray	CoroNet	4	89.60
Sahinbas and Catak [26]	X-ray	VGG16, VGG19, ResNet DenseNet and InceptionV3	2	80
Medhi et al. [27]	X-ray	Deep CNN	2	93
Zhang et al. [16]	X-ray	CAAD	2	95.18
Apostopolus et al. [25]	X-ray	VGG-19	3	93.48
Narin et al. [31]	X-ray	InceptionV3, ResNet50, Inception-ResNetV2	2	98
This study	X-ray	InceptionV3, ResNet50, ResNet101 ResNet152, Inception-ResNetV2	2 (COVID-19/Normal)	96.1
This study	X-ray	InceptionV3, ResNet50, ResNet101 ResNet152, Inception-ResNetV2	2 (COVID-19/Viral Pne.)	99.5
This study	X-ray	InceptionV3, ResNet50, ResNet101 ResNet152, Inception-ResNetV2	2 (COVID-19/Bacterial Pne.)	99.7

Figura 15. Comparación de rendimiento sobre métodos de diagnóstico de COVID-19 utilizando imágenes de rayos X de tórax (Narin et al., n.d.)

2.4 La computación en el borde o Edge computing

Desde hace algunos años los paradigmas de la computación han evolucionado de estar centrados en modelos más clásicos tipo cliente-servidor a modelos orientados al *Cloud* (o computación en la nube a través de Internet), donde los recursos y desarrollos se centralizan en infraestructuras de pago por uso.

Según el NIST (Mell & Grance, 2011) la computación en la nube se define como "*un modelo que permite un acceso a la red versátil, cómodo y a la carta a un conjunto compartido de recursos informáticos configurables (por ejemplo, redes, servidores, almacenamiento, aplicaciones y servicios) que pueden ser rápidamente aprovisionados y liberados con un mínimo esfuerzo de gestión o interacción del proveedor de servicios*."

Este modelo centralizado ha sido ampliamente adoptado por las empresas debido sobre todo a una menor inversión fija en infraestructuras, aumentando la posibilidad de realizar proyectos de desarrollo que de otro modo no serían viables en coste y tiempo.

Debido a esta centralización en servicios a través de Internet, se crearon nuevos desarrollos que se interconectaban directamente con el *Cloud* para la adquisición de datos y su tratamiento posterior, lo que dio lugar al concepto de *Internet-of-Things* (IoT) o Internet de las cosas, cuya definición es "*un conjunto de dispositivos informáticos (es decir, cosas)*

interconectados a través de Internet y destinados a ofrecer servicios dirigidos a todo tipo de aplicaciones, al tiempo que se cumplen los requisitos de seguridad." (De Donno et al., 2019)

La proliferación de la Internet de las cosas (IoT) y el éxito de los servicios en la nube enriquecidos han impulsado el horizonte de un nuevo paradigma informático, la computación en el borde o *Edge*, que exige el procesamiento de los datos directamente en el propio borde de la red (Shi et al., 2016).

Las necesidades que han propiciado el auge del *Edge Computing* vienen dadas por las limitaciones del proceso de datos en *Cloud*, donde determinadas características como el ancho de banda, el tiempo de procesamiento o las posibles restricciones de cara al tratamiento y envío de datos en cuanto a seguridad y privacidad son fundamentales.

Una posible clasificación de requisitos para casos de uso de *Edge Computing* fue presentada por (Khan et al., 2019) donde se habla de siete puntos o requisitos para poder utilizarlo. De estos siete puntos consideramos que algunos son demasiado específicos de cara a la definición de *Edge Computing* (por ejemplo, la necesidad de un mecanismo de facturación dinámica o soporte en tiempo real) mientras que otros como la gestión de recursos, la necesidad de arquitectura escalable, la redundancia y la seguridad si son factores críticos a contemplar para este tipo de arquitecturas.

2.5 Uso de la inteligencia artificial en Edge

En los últimos años la necesidad de más capacidad de cómputo en Edge ha dado lugar a una mejora sustancial en la capacidad de los sistemas embebidos y el hardware, donde la Inteligencia Artificial, y en especial el Deep Learning, ha sido uno de los campos estrella a considerar en esta evolución.

En cuanto al hardware Edge dedicado a este campo podemos ver la clasificación realizada en la obra "*Edge AI: Convergence of Edge Computing and Artificial Intelligence - Xiaofei Wang, Yiwen Han, Victor C. M. Leung, Dusit Niyato, Xueqiang Yan, Xu Chen - Google Libros*, n.d." en la siguiente tabla :

Tabla 1. Edge computing AI Hardware

	Owner	Production	Feature
Integrated commodities	Microsoft	Data Box Edge	*Competitive in data pre-processing and data transmission*
	Intel	Movidius Neural Compute Stick	*Prototype on any platform with plug-and-play simplicity*
	NVIDIA	Jetson	*Easy-to-use platforms that runs in as little as 5 W*
	Huawei	Atlas Series	*An all-scenario AI infrastructure solution that bridges "device, edge, and cloud"*
AI hardware for edge computing	Qualcomm	Snapdragon 8 Series	*Powerful adaptability to major DL frameworks*
	HiSilicon	Kirin 600/900 Series	*Independent NPU for DL computation*
	HiSilicon	Ascend Series	*Full coverage—from the ultimate low energy consumption scenario to high computing power scenario*
	MediaTek	Helio P60	*Simultaneous use of GPU and NPU to accelerate neural network computing*
	NVIDIA	Turing GPUs	*Powerful capabilities and compatibility but with high energy consumption*
	Google	TPU	*Stable in terms of performance and power consumption*
	Intel	Xeon D-2100	*Optimized for power- and space-constrained cloud–edge solutions*
	Samsung	Exynos 9820	*Mobile NPU for accelerating AI tasks*
Edge computing frameworks	Huawei	KubeEdge	*Native support for edge–cloud collaboration*
	Baidu	OpenEdge	*Computing framework shielding and application production simplification*
	Microsoft	Azure IoT Edge	*Remotely edge management with zero-touch device provisioning*
	Linux Foundation	EdgeX	*IoT edge across the industrial and enterprise use cases*
	Linux Foundation	Akraino Edge Stack	*Integrated distributed cloud–edge platform*
	NVIDIA	NVIDIA EGX	*Real-time perception, understanding, and processing at the edge*
	Amazon	AWS IoT Greengrass	*Tolerance to edge devices even with intermittent connectivity*
	Google	Google Cloud IoT	*Compatible with Google AI products, such as TensorFlow Lite and Edge TPU*

(Edge AI: Convergence of Edge Computing and Artificial Intelligence - Xiaofei Wang, Yiwen Han, Victor C. M. Leung, Dusit Niyato, Xueqiang Yan, Xu Chen - Google Libros, n.d.)

Uno de los sistemas hardware referenciados es la serie de equipos Jetson de NVIDIA, que es un sistema integrado diseñado por NVIDIA para una nueva generación de máquinas autónomas (*Edge AI: Convergence of Edge Computing and Artificial Intelligence - Xiaofei Wang, Yiwen Han, Victor C. M. Leung, Dusit Niyato, Xueqiang Yan, Xu Chen - Google Libros*, n.d.). La serie de productos hardware de la familia Jetson es una plataforma de específica para inteligencia artificial y se utiliza principalmente para máquinas autónomas, sensores de alta definición y análisis de vídeo.

La mayoría de los casos aplicados de inteligencia artificial en Edge se basan en modelos que son entrenados y desplegados en *Cloud* mientras que los equipos Edge envían la información y esperan el resultado de la inferencia. El envío de la información (privacidad) y la latencia resultante son un problema en este aspecto, lo que ha dado pie a una evolución hardware para que los equipos Edge realicen la propia inferencia sin dependencias del *Cloud* e incluso el entrenamiento del propio modelo.

Para realizar las tareas de inferencia en Edge se realizan diferentes técnicas entre las que están el uso de métodos genéricos de optimización de modelos como la poda de parámetros, factorización de rango bajo, filtros de convolución compactos o destilación de conocimientos, la optimización específica según las características del modelo para equipos Edge, donde se pueden utilizar diferentes técnicas para reducir el espacio de búsqueda a nivel de entrada, reducir la complejidad de la estructura del modelo, el propio tratamiento o la elección del framework, o la compartición de recursos de computación entre diferentes equipos Edge.

En cuanto a las tareas de entrenamiento en *Edge*, la mejor aproximación actualmente es realizar entrenamiento distribuido entre los diferentes equipos *Edge*. En caso que el número de equipos sea muy grande, se opta por una solución tipo *Federated Learning* (Bonawitz et al., 2019; *Google AI Blog: Federated Learning: Collaborative Machine Learning without Centralized Training Data*, n.d.) donde se centralizan los datos de entrenamiento en una máquina o en un centro de datos.

Para los casos de *Edge* en local donde la privacidad y la conexión es rápida se pueden utilizar técnicas de clústering para distribuir tanto el entrenamiento como la inferencia entre los nodos.

2.6 Tecnología de contenedores. Del *Cloud* al *Edge*.

Los paradigmas han cambiado en los últimos años en cuanto a la gestión de la información y el uso de infraestructuras *Cloud*. En paralelo a esto se han ido desarrollando otras tecnologías

a nivel software que han permitido mayor facilidad de gestión de cara a la utilización de estas nuevas infraestructuras.

Una de estas tecnologías que se ha considerado de las más influyentes son los contenedores, y en especial la herramienta de código fuente abierto Docker (Merkel, n.d.)

Anteriormente las máquinas virtuales eran el núcleo de las arquitecturas tipo *Cloud*, donde los servicios corrían en sistemas operativos aislados con sus componentes software en un único equipo físico. La tecnología de contenedores superaba ampliamente a las máquinas virtuales debido sobre todo a un menor coste computacional, mayor rapidez de despliegue y menor coste en recursos con respecto a éstas. (Ismail et al., 2016)

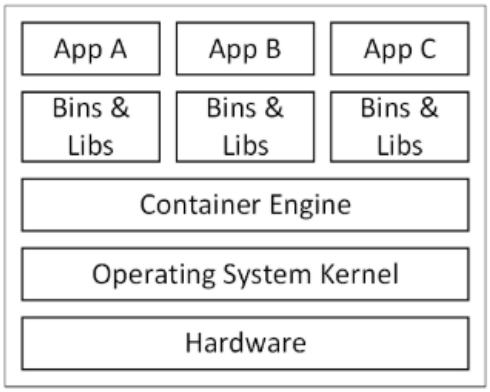


Figura 16. Arquitectura de contenedores. (Ismail et al., 2016)

En paralelo con la evolución del *Cloud* hacia el Edge, se valoraron este tipo de tecnologías también para este caso, pasando de modelos centralizados a distribuidos, llegando a la conclusión de que el uso de este tipo de tecnologías es viable para el uso en Edge por su bajo consumo de recursos (Ismail et al., 2016)

De cara a los casos de uso centrados en inteligencia artificial en Edge, la tecnología de contenedores Docker también ha sido validada como una solución muy conveniente (Xu et al., 2017) ya que su flexibilidad, aislamiento y bajo consumo de recursos son ideales bajo este escenario, siendo equiparables en rendimiento a utilizar las herramientas sin esta tecnología.

2.7. Orquestación de contenedores. Kubernetes y K3S.

Una de las características más interesantes de los contenedores es que pueden ser utilizados de manera sencilla para crear arquitecturas de clústeres, de forma que se crean soluciones escalables y tolerantes a fallos independientemente de que se ejecuten en un solo host o en varios. A partir de este concepto se crearon los orquestadores.

Citando a Google (*Google Cloud Platform Blog: Containers, VMs, Kubernetes and VMware*, n.d.) en la presentación de su solución de código fuente abierto, Kubernetes : "*Mientras que la ejecución de contenedores individuales es suficiente para algunos casos de uso, el verdadero poder de los contenedores proviene de la implementación de sistemas distribuidos, y para hacer esto se necesita una red. Sin embargo, no se necesita cualquier red. Los contenedores ofrecen a los usuarios finales una abstracción que convierte a cada contenedor en una unidad de computación autónoma. Tradicionalmente, un lugar donde esto se ha roto es la red, donde los contenedores están expuestos en la red a través de la dirección de la máquina anfitriona compartida. En Kubernetes, hemos adoptado un enfoque alternativo: que cada grupo de contenedores (llamado Pod) merece su propia y única dirección IP que es accesible desde cualquier otro Pod en el clúster, ya sea que estén ubicados en la misma máquina física o no*".

La solución de Kubernetes es, a día de hoy, la solución más ampliamente adoptada del mercado para estas tareas. En la 2021 Kubernetes Adption Surver realizada por la empresa Portworx (Storage, n.d.) arroja unos datos interesantes: Un 68% de los profesionales del sector IT afirman que utilizan más Kubernetes a raíz de la pandemia de COVID-19 y un 90% de los mismos espera que Kubernetes juegue un papel más importante en la gestión de la infraestructura de IT de sus organizaciones.

Si nos centramos en la adopción en Edge, existen varios proyectos en desarrollo actualmente para adaptar las soluciones de Kubernetes y orquestación a las características y recursos más limitados propios del hardware Edge: KubeEdge (Wang et al., 2020), K3S (Pääkkönen et al., 2021), MicroK8S (Böhm & Wirtz, 2020). Fundamentalmente estas aproximaciones se diferencian en la dependencia que puedan tener en infraestructuras *Cloud*, la manera de desplegar y configurar el propio clúster de Kubernetes o la gestión de los propios binarios, dando lugar a las llamadas distribuciones de Kubernetes ligeras o particulares para *Edge* (Böhm & Wirtz, 2020).

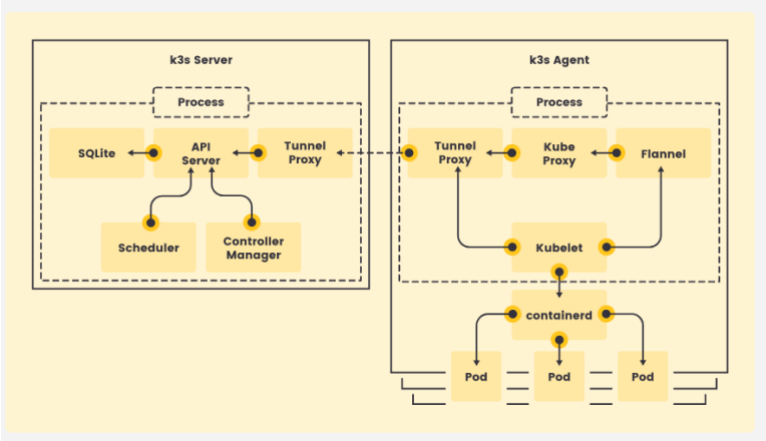


Figura 17. Diagrama de funcionamiento de K3S (Sendgrid et al., 2020)

2.6 TensorFlow

TensorFlow es una plataforma de código fuente abierto para realizar tareas de Machine Learning de manera distribuida a escala. Fue creada en el año 2015 por ingenieros de Google Brain como evolución de su anterior proyecto propietario interno llamado DistBelief. Esta plataforma puede ejecutarse en un conjunto heterogéneo tanto de hardware (CPUs, GPUs y aceleradores como TPUs) como de sistemas operativos (Linux, Windows, macOS y plataformas móviles como Android e iOS) y software (Python, C++, Java...)

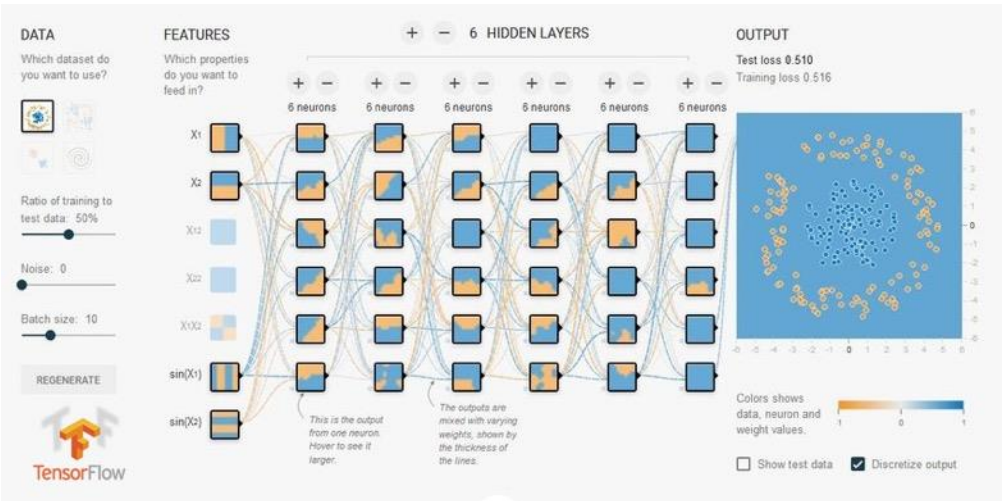


Figura 18. Arquitectura de una red neuronal CNN empleando TensorFlow (Kallam et al., 2018)

TensorFlow utiliza un único gráfico de flujo de datos para representar todos los cálculos y el estado de un algoritmo de aprendizaje automático, incluyendo las operaciones matemáticas individuales, los parámetros, sus reglas de actualización y el pre-procesamiento de la entrada.

2.6.1 Componentes básicos

Tensores (*Tensors*): Son la unidad fundamental de trabajo de TensorFlow. Un tensor es una matriz n-dimensional que representa cualquier tipo de datos a utilizar en el modelo, tanto de entrada como de salida. Pueden ser creados por los datos iniciales del programa o bien como resultado de otras operaciones entre tensores.

Gráficas (Graphs): Una gráfica en TensorFlow define el flujo de operaciones (o unidades de computación) a realizar en el modelo utilizando los tensores (o unidades de datos).

Estas gráficas son estructuras de datos muy flexibles que permiten guardar su estado y poder restaurarlo en un futuro, además de poder distribuir la carga de trabajo de manera distribuida entre diferente hardware.

2.6.2 Arquitectura de TensorFlow

La arquitectura de TensorFlow se puede dividir fundamentalmente en cuatro partes:

- Pre-procesamiento de información
- Creación del modelo
- Entrenamiento del modelo
- Estimación del modelo

Se pueden ver estas partes en la imagen siguiente:

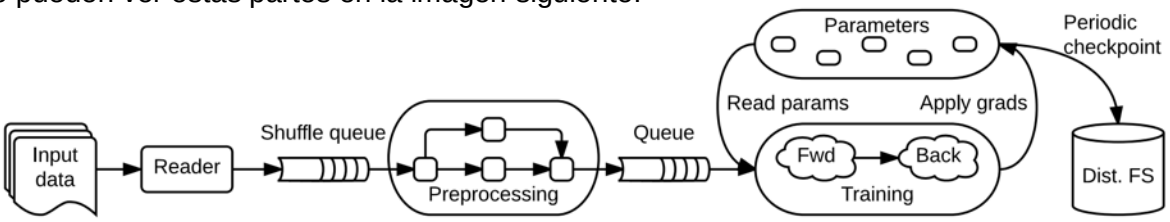


Figura 19. Diagrama de flujo de un entrenamiento con gráficos de entrada en TensorFlow con estado de pre-procesamiento, entrenamiento y puntos de control.

2.6.3 Evolución a TensorFlow 2.0

En sus versiones iniciales TensorFlow utilizaba el concepto de gráfica estática para realizar los cálculos: Primero se creaba la gráfica con las operaciones a realizar entre los tensores y posteriormente se ejecutaba esta gráfica. El 30 de septiembre de 2019 se liberó la versión 2.0

de TensorFlow, que entre otros cambios significativos pone cambia por defecto el uso del modo de ejecución por gráficas al modo de ejecución Eager (o "ansioso"). Este modo permite evaluar las operaciones y los valores de los tensores de manera inmediata, lo que se traduce en mayor facilidad a la hora de codificar y depurar a costa de una penalización de rendimiento general (en torno a un 20% según la documentación oficial). (*TensorFlow*, n.d.)

Además de estos cambios, la versión 2.0 integró la API Keras como API de alto nivel por defecto.

2.6.4 KERAS

Keras es una API de alto nivel que se utiliza encima de diferentes plataformas para realizar tareas comunes de Machine Learning de manera más sencilla, de forma que sea fácil de utilizar, modular y extensible. Fue desarrollada por el ingeniero de Google Francois Chollet (MAIL) el año 2015 y pasó a ser la API de alto nivel oficial de TensorFlow el año 2017 con la versión 1.4, siendo el estándar de uso a partir de la versión 2.0.

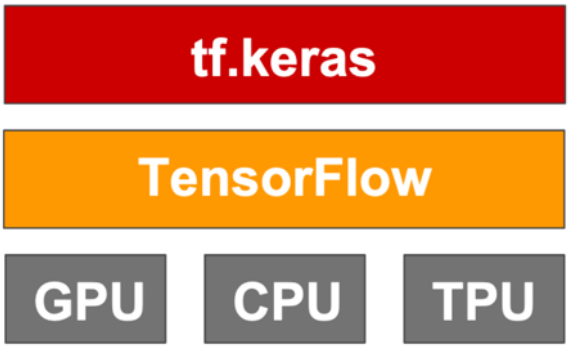


Figura 20. Esquema de sistema TensorFlow con API Keras (Chollet, 2018)

2.6.5 TensorFlow Serving

Según su propia web: "*TensorFlow Serving es un sistema de publicación flexible y de alto rendimiento para modelos de aprendizaje automático, diseñado para entornos de producción. TensorFlow Serving facilita la implementación de nuevos algoritmos y experimentos, al tiempo que mantiene la misma arquitectura de servidor y API. TensorFlow Serving proporciona una integración lista para usar con los modelos de TensorFlow, pero se puede ampliar fácilmente para ofrecer otros tipos de modelos.*"(Mart\'\in~Abadi et al., 2015a).

Una vez entrenado el modelo y exportado desde TensorFlow, la idea es utilizar la infraestructura de TensorFlow Serving para servir el modelo generado y poder realizar el trabajo de inferencia mediante su API.

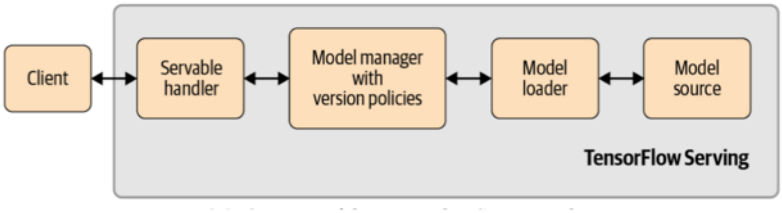


Figura 21. Arquitectura de TensorFlow Serving (Hannes Hapke & Nelson, 2020)

2.7 Docker

En los últimos años se han popularizado los despliegues de aplicaciones dentro de los llamados contenedores de software, donde el sistema operativo permite aislar componentes de software a nivel de usuario de manera que pueda correr cada componente con un espacio y recursos asignados, independientemente del resto.

Esta tecnología aporta beneficios tales como la estandarización, productividad, eficiencia, compatibilidad, facilidad de mantenimiento, simplicidad, despliegue rápido, fácil escalabilidad, plataformas *multi-cloud*, aislamiento y seguridad entre otras.

La solución más popular del mercado para la gestión de contenedores es el proyecto Docker de código fuente abierto. Docker fue creado en el año 2013 como un proyecto Open Source por la empresa dotCloud, que posteriormente cambió su nombre a Docker, Inc. Esta solución permitía gestionar contenedores de software en sistemas GNU/Linux utilizando diferentes componentes a nivel de *Kernel*, mejorando las implementaciones de contenedores anteriores.

2.7.1 LXC y CGroups

LXC (*LinuX Containers*) es una tecnología de gestión de contenedores nativa al Kernel Linux que permite ejecutar procesos de manera aislada. Su principal funcionalidad está basada en el soporte de grupos de control (*cgroups*), que permiten definir conjuntos de reglas sobre los procesos que pueden correr en un sistema, desde limitación de consumo de CPU hasta memoria o incluso operaciones de entrada/salida.

Esta tecnología es fundamental para poder limitar la asignación de recursos de procesos en soluciones de contenedores.

2.7.2 Namespaces

Otra tecnología fundamental para la utilización de contenedores son los espacios de nombres (namespaces). Estos espacios están soportados a nivel de kernel Linux para permitir aislar una serie de recursos para que sean únicamente visibles por un grupo de procesos en concreto.

Esta tecnología permite aislar y gestionar varios componentes para diferentes procesos, como pueden ser procesos visibles, puntos de montaje accesibles, IPCs, tráfico de red y usuarios del sistema.

Junto con la anterior, es fundamental para poder aislar determinados recursos en los contenedores y restringirlos con respecto al resto.

2.7.3 *Layers* y *Copy-On-Write*

La tercera tecnología utilizada en contenedores son las capas. Cada contenedor está formado por una capa base y una o varias capas que se van "apilando", como sistemas de archivos independientes y que forman la propia imagen, más una capa modificable para la ejecución.

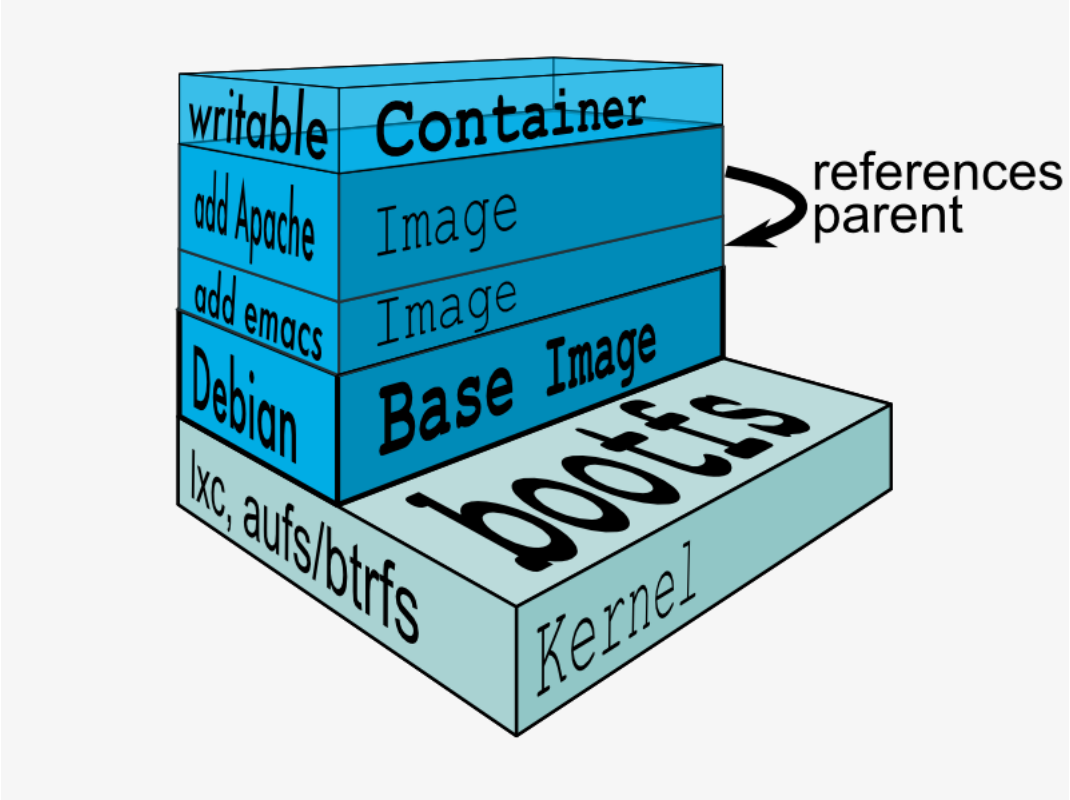


Figura 22. Esquema de imágenes Docker y contenedores(Ning-An, 2017)

Este concepto de capas permite poder reutilizarlas en diferentes contenedores permitiendo un mejor control de versiones y optimización del almacenamiento, además de poder implementar una solución de *Copy-On-Write*: En el caso que un fichero exista en alguna capa se accede directamente a él, y si es modificado, se guarda una copia del fichero en la capa de escritura del contenedor para acceder de manera más eficiente, registrando todos los cambios respecto a la imagen original.

2.8 Kubernetes y K3S

Kubernetes es un proyecto de código fuente abierto liberado por Google basado en su proyecto interno *Borg*. Se trata de un framework que permite gestionar clústeres de contenedores de manera sencilla centrado en escalabilidad y APIs normalizadas (Authors, 2021).

Kubernetes se ha convertido en el estándar actual de gestión de grandes volúmenes de contenedores y es el núcleo de los desarrollos actuales de software basados en computación en la nube.

2.8.1 Componentes de Kubernetes

La plataforma Kubernetes divide sus componentes en dos partes diferenciadas: El plano de control y el plano de ejecución. El plano de control se localiza en el nodo Master del clúster y es el que gestiona todo el propio clúster, mientras que el plano de ejecución se localiza en los nodos Secundarios de ejecución del clúster.

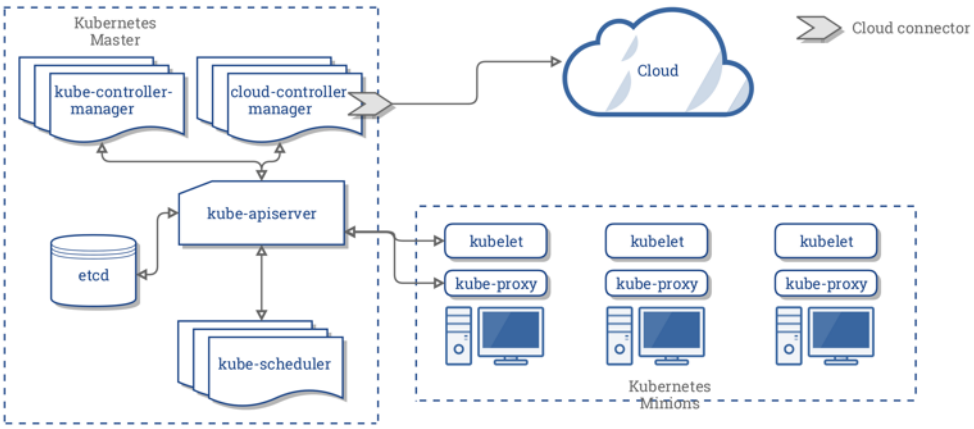


Figura 23. Kubernetes - Architecture cloud controller (Kubernetes.io, 2020)

2.8.1.1 Plano de control

El plano de control contempla todas las partes necesarias para gestionar y controlar el clúster. Contiene 5 componentes fundamentales para esta tarea:

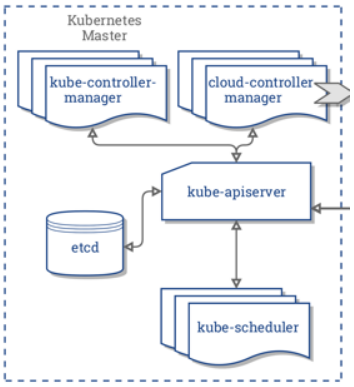


Figura 24. Kubernetes – Control plane (Kubernetes.io, 2020)

- **KUBE-APISERVER**

  El Kube-APIServer es el componente que expone la API de Kubernetes al exterior. Este componente permite controlar el estado de todo el clúster mediante esta API, recibiendo las peticiones y actualizando el estado del clúster en la base de datos etcd.

- **ETCD**

  Base de datos persistente de tipo par clave: valor que guarda la configuración y el estado interno del clúster. Es actualizada por el resto de componentes del Plano de Control.

- **KUBE-SCHEDULER**

  Planificador que despliega los Pods que no estén asignados a ningún nodo a un nodo donde puedan ejecutarse según una serie de requisitos como pueden ser recursos, políticas, hardware, afinidad.

- **KUBE-CONTROL-MANAGER**

  Gestor de controladores de Kubernetes. Los controladores son procesos únicos binarios que permiten gestionar entre otras cosas el estado de los pods (Controlador de Nodos), el nivel de replicación del clúster (Controlador de replicación), los endpoints de acceso a los servicios (Controlador de Endpoints) y las cuentas de servicio y acceso a la API (Controladores de tokens).

- **CLOUD-CONTROL-MANAGER**

  Gestor de controladores en *Cloud* de Kubernetes. Sirve para gestionar accesos a proveedores externos de *Cloud*. En caso que se utilice el clúster en un entorno *Cloud*, Kubernetes proporciona una derivación de los controladores del Kube-Control-Manager para gestionarlos directamente mediante plugins en distintos proveedores de *Cloud* con una API normalizada.

2.8.1.2 Plano de ejecución

Los componentes del plano de ejecución son los que se ejecutan en los propios Nodos del clúster. Son 3 componentes:

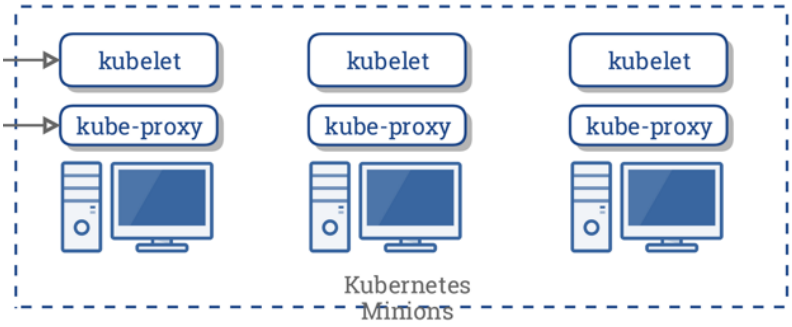


Figura 25. Kubernetes plano ejecución (Kubernetes.io, 2020)

- **KUBELET**

  Los Kubelets son los agentes que se ejecutan en cada nodo del clúster y su función es comprobar que los contenedores están corriendo en un Pod de manera correcta. Utiliza un formato de especificaciones de los Pod para comprobar que los contenedores que estén descritos en este formato estén funcionando y corriendo adecuadamente.

- **KUBE-PROXY**

  Kube-Proxy permite abstraer las conexiones de red y definir servicios en Kubernetes con la configuración de red adecuada y gestionando el reenvío de conexiones.

- **RUNTIME DE CONTENEDORES**

  El *runtime* de los contenedores es el software que se encarga de la propia ejecución de los contenedores. Habitualmente se utilizan los *runtimes* Docker (del que hemos hablado en anteriores puntos del documento) y containerd.

2.8.1 K3S

K3S es una distribución de Kubernetes creada por *Rancher Labs* en el año 2019 como proyecto de código fuente abierto y que actualmente es un proyecto mantenido por la CNCF (*Cloud Native Computing Foundation*) desde el año 2020. Entre sus particularidades se encuentra:

- Distribución empaquetada como un único binario “k3s”
- Utilización de base de datos sqlite3 como almacenamiento para la base de datos etcd
- Contempla seguridad desde el inicio

Esta distribución permite gestionar todo un clúster completo de Kubernetes desde un único binario de manera encapsulada. Es muy útil para entornos embebidos ya que simplifica mucho la gestión y el despliegue de Kubernetes al estar todo incluido en el mismo paquete. (Sendgrid et al., 2020)

3. Objetivos y metodología de trabajo

Basándonos en el estado del arte, referencias, investigaciones y trabajos mencionados en los anteriores capítulos, pasamos a plantear los objetivos marcados para el proyecto y la metodología necesaria para alcanzarlos.

3.1 Objetivo General

Tomamos como objetivo del proyecto el diseñar una solución de computación en el borde (Edge) de bajo coste en una arquitectura en clúster que permita realizar tareas de inteligencia artificial con utilización de GPUs en paralelo para mejora de rendimiento y sin tener dependencias externas de conectividad, de tal forma que sea un producto completo.

Esta solución debe suponer una mejora en cuanto a rendimiento respecto a soluciones atómicas de bajo coste y en cuanto a precio respecto a otras más complejas, pero además debe poder ser escalable según las necesidades del proyecto. Esta escalabilidad deberá ser fácilmente gestionable utilizando infraestructuras de contenedores y soluciones estándar de mercado para gestionarlos como Kubernetes.

Este enfoque de computación en el borde (o Edge Computing) tiene ventajas claras sobre las soluciones de uso más actual basadas en computación en la nube (o Cloud Computing), como por ejemplo:

- Poder utilizarse en situaciones en las que las conexiones externas no están permitidas debido a la criticidad de los datos utilizados o a la legislación vigente en cuanto a la privacidad y seguridad.
- Menor latencia a la hora de analizar y mostrar los resultados en la fase de inferencia.
- Procesamiento local de los datos en el equipo sin envíos, recepciones o tratamientos externos.

En cuanto a la parte algorítmica de inteligencia artificial, planteamos el procesamiento local en el propio clúster tanto para entrenamiento como para la inferencia, pero de manera distribuida entre los nodos para cumplir la escalabilidad necesaria pudiendo ampliarse según

sea necesario con más nodos dependiendo de las necesidades del proyecto evitando las limitaciones tradicionales de un único hardware disponible.

En cuanto a la parte de gestión del propio clúster, contamos con utilizar soluciones estándar de mercado aplicadas tradicionalmente en entornos en la nube (Cloud) como contenedores y Kubernetes para realizar la gestión distribuida de los algoritmos entre los nodos, lo que además podría dar lugar a futuras opciones de implementación y configuración tan interesantes como sistemas híbridos *Cloud/Edge* dinámicos.

En cuanto al hardware a utilizar, contamos con los SoCs (*System-On-Chip*) de la familia Jetson de nVidia, que son las placas más utilizadas en el contexto de inteligencia artificial, robótica y computación en el borde (*Edge*) debido a su bajo coste, arquitectura ARM de 64 bits, BSP (*Board Support Package*) con GNU/Linux y soporte de GPUs con bibliotecas CUDA.

Para validar todos estos puntos anteriores hemos elegido un caso de uso de inteligencia artificial aplicado como es la ejecución de algoritmos de clasificación de imágenes basados en redes neuronales CNN para la detección de COVID-19 en función a imágenes de Rayos-X de alta definición. Dada su complejidad y requisitos, consideramos que este caso cumple con los objetivos marcados en los puntos anteriores y permite valorar la viabilidad de la solución planteada.

3.2 Objetivos específicos

- Demostrar la viabilidad de uso de una infraestructura de clúster con las características software necesarias:
    - Soporte de sistema GNU/Linux completo a nivel de dependencias en arquitecturas ARM de 64 bits.
    - Soporte de bibliotecas estándar de mercado como TensorFlow o Keras para la parte algorítmica.
    - Soporte de infraestructura de contenedores para desarrollo y despliegue (Docker, conteiner) nativos en el clúster.
    - Soporte de infraestructura de clúster de contenedores y orquestación con soluciones de mercado como Kubernetes.

    - Aceleración de los cálculos algorítmicos utilizando GPUs en cada nodo utilizando bibliotecas estándar.
    - Posibilidad de procesamiento en paralelo de tareas de IA, tanto en CPU como en GPUs entre los nodos utilizando las infraestructuras anteriores.

- Validar la solución diseñada con el caso de uso, cumpliendo los objetivos del proyecto con las métricas adecuadas para tal fin.
- Valorar diferentes iteraciones y modificaciones en los hiperparámetros utilizados y contrastar las métricas resultantes con otros sistemas como:
    - nVidia Jetson Nano (Una única placa)
    - Equipo de sobremesa PC AMD Ryzen 1800X 16GB con nVidia RTX 2070
    - nVidia Jetson Xavier AGX
    - MacBook Pro

3.3 Metodología del trabajo

Tomamos como punto de partida los trabajos mencionados sobre detección de COVID-19, tanto los algoritmos planteados como los conjuntos de datos de imágenes de Rayos-X de tórax disponibles que se plantearon para el reto de la Comisión Europea en su iniciativa "AI-ROBOTICS vs COVID-19" (Alliance, n.d.).

Planteamos una aproximación ágil con sucesivas iteraciones, realizando una selección de algoritmos y base de datos de imágenes a utilizar para posteriormente adaptarlos a las requerimientos y características de la plataforma, más limitada que los utilizados en los sistemas de ejemplo de los trabajos.

Con esto tenido en cuenta realizamos un planteamiento inicial basado en el siguiente cuadro, con las fases y los artefactos generados en cada iteración.

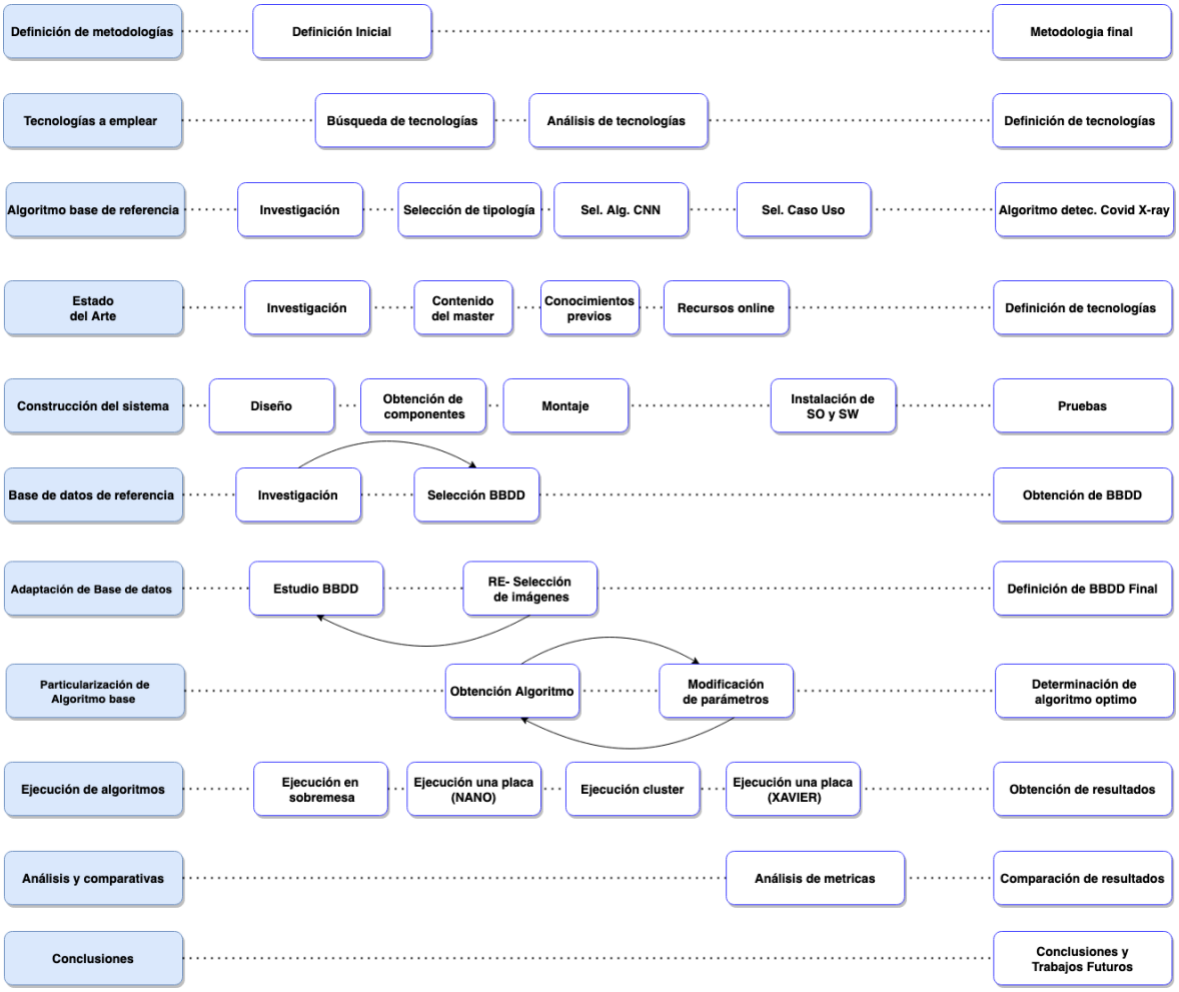


Figura 26. Diagrama de metodología

3.3.1 Definición de metodologías

Al comienzo del proyecto definimos las metodologías a seguir basándonos en los objetivos iniciales que nos habíamos propuesto. Tras el planteamiento inicial tuvimos una puesta en común con un pequeño "*brainstorming*" donde pusimos sobre la mesa una gran diversidad de posibles proyectos. Debido a la experiencia y formación previas de cada uno de nosotros, optamos plantear un proyecto propio, sin referencias existentes. Este tipo de proyectos son mucho más arriesgados, pero a su vez mucho más interesantes desde el punto de vista de investigación y estado del arte actuales.

Una vez aprobada la posibilidad de su elaboración nos planteamos cual sería la mejor forma de llevar a buen término el desarrollo de los retos planteados.

Utilizando nuestra experiencia previa en desarrollo de productos para ámbitos empresariales, industriales y "*StartUps*" además de metodologías ágiles y gestión de proyectos, pudimos definir unas necesidades iniciales del proyecto que fueran lo suficientemente concretas como para ir asegurando su éxito, aunque hayan sido actualizadas conforme se ha ido avanzando en el proyecto de manera iterativa.

3.3.2 Tecnologías a emplear

En esta fase inicialmente realizamos una búsqueda inicial de tecnologías disponibles tanto a nivel hardware como software que pudiéramos utilizar. Una vez realizada la búsqueda, procedimos a analizar los resultados para determinar las características mínimas que debía cumplir el producto y seleccionar cuales serían las que se emplearían en éste, tanto a nivel hardware como software.

- **Hardware**

De cara al hardware estudiado inicial se descartó el uso de SBCs (*Single Board Computer*) ARM populares como la familia de Raspberry Pi y similares que, aunque con un coste que lo justificaría como una opción posible, debido a su escasa orientación hacia algoritmos de inteligencia artificial y su falta de rendimiento en cuanto a paralelización utilizando GPUs (el rendimiento del VideoCore de Raspberry Pi para procesos con TensorFlow es muy limitado y otras alternativas de SBCs con las mismas características también tienen similares resultados), fueron descartadas.

Otra opción viable, aunque más costosa fue la utilización en paralelo de múltiples tarjetas gráficas en una estación de trabajo. Esta solución suele ser habitual en entornos de cálculos complejos e inteligencia artificial, pero debida al excesivo coste y la falta de equipos disponibles para pruebas, también fue descartada.

Finalmente debido a la disponibilidad de tarjetas nVidia Jetson Nano de bajo coste por parte de los integrantes del equipo (anteriormente utilizadas en proyectos de robótica con visión artificial) se optó por profundizar en el soporte de este hardware para los requisitos necesarios de escalabilidad y clústering, dando el visto bueno a los requisitos y seleccionando finalmente esta solución para el proyecto.

NOTA: Durante el desarrollo del proyecto surgió la oportunidad de disponer de manera temporal de una placa de la misma familia de las seleccionadas para el proyecto pero con un incremento notable en capacidades, denominada nVidia Jetson Xavier NX. Esto nos permitió realizar algunas pruebas con la misma para incorporarlas a la comparativa final.

- **Software**

En cuanto a las tecnologías software empleadas en el proyecto, fueron claramente definidas al inicio del proyecto y no se ha producido ninguna alteración durante su desarrollo.

- Sistema Operativo

Inicialmente partimos del BSP (*Board Support Package*) del fabricante nVidia con su JetPack 4.5, una solución que incluye un sistema operativo basado en Ubuntu Linux, con Kernel con soporte de la infraestructura necesaria para utilizar contenedores Docker.

- Infraestructura de contenedores y Kubernetes

Los paquetes relacionados con Docker, containerd y la distribución de Kubernetes para sistemas embebidos k3s estaban disponibles para la arquitectura de este equipo y fueros desplegados e instalados sin mayores problemas.

- Desarrollo

El lenguaje de programación Python en su versión 3 junto con las dependencias y bibliotecas necesarias para los trabajos de inteligencia artificial (Scikit-Learn, OpenCV, TensorFlow, Keras, JupyterLab, etc...) estaban disponibles para esta arquitectura de manera externa como paquetes en los repositorios de Python o bien directamente incluidas en el sistema operativo.

- Aceleración GPU

Las bibliotecas necesarias para dar soporte a la aceleración de GPU en los equipos estaban disponibles directamente en el BSP de nVidia o bien en las páginas de descarga de la plataforma Jetson (por ejemplo, la revisión de TensorFlow con soporte de GPU de las placas Jetson).

NOTA: Como se verá más adelante, las pruebas iniciales de software y de desarrollo fueron ejecutadas inicialmente sobre el sistema operativo de los equipos, pero en cuanto el soporte de contenedores estuvo funcionando, migramos la manera de desarrollar para que fuera todo en contenedores y no dependiera de la infraestructura del propio sistema operativo.

3.3.3 Algoritmo base de referencia

Una vez definidas las tecnologías a emplear, pasamos a investigar y analizar los diferentes algoritmos que podríamos utilizar. Debido a los requisitos necesitábamos un algoritmo ya existente que fuera lo suficientemente potente en cuanto a calidad de resultados, exigente a nivel hardware y con cierta relevancia actual para poder llevar al extremo la tecnología que habíamos seleccionado.

Al ser el proyecto de tipo piloto experimental debíamos de centrarnos en utilizar el algoritmo elegido y realizar las comparativas oportunas con otros equipos más que en el refinamiento y optimización de resultados de éste.

Debido al estado actual de pandemia de COVID-19, buscamos como opciones algoritmos que cumplieran estas capacidades y estuvieran relacionados con la pandemia. La Unión Europea, dentro de la comisión *European AI Alliance* planteó un reto denominado “AI-ROBOTICS vs COVID-19”, donde se expusieron varias tecnologías, experimentos y soluciones. Después de investigar sobre los algoritmos y casos de uso planteados, vimos que uno de los más extendidos y que se adaptaba a nuestras necesidades fue el uso de inteligencia artificial en algoritmos de detección de pacientes positivos con COVID-19 mediante técnicas de clasificación de imágenes de radiografías de tórax.

Los estudios elegidos utilizaban arquitecturas de redes neuronales convolucionales (CNNs), que nos permitían una gran flexibilidad a la hora de modificar desde los propios hiperparámetros de la red como el conjunto de datos a utilizar (base de datos de imágenes) para poder adecuarlos a nuestras necesidades y limitaciones.

3.3.4 Estado del Arte

Es habitual en todo proyecto de investigación (y más aún si hablamos de un piloto experimental como este caso) revisar el llamado "Estado del Arte", donde se localizan los trabajos publicados a nivel mundial sobre los temas y tecnologías que se van a emplear.

Esto es necesario tanto para no caer en tentativas de plagio o duplicidad de información como para poder tomar como referencia los últimos avances disponibles en lo que se va a tratar durante el proyecto.

Debido por un lado a la cantidad de tecnologías referidas en este trabajo y por otro a la actualidad de la pandemia de COVID-19 relacionada con el coronavirus SaRS-COV-2, vemos necesario hacer referencia a los siguientes puntos dentro del estado del arte:

- Estudios más relevantes relacionados con el SaRS-COV-2 y la enfermedad COVID-19, para poder contextualizar el caso de uso elegido para el proyecto y la selección del algoritmo adecuado. Nos hemos centrado en los últimos estudios relacionados con el virus, desde los aspectos mas básicos y generales hasta los últimos avances.

- Técnicas de diagnóstico de COVID-19 con máquinas de Rayos-X, ya que el algoritmo seleccionado está incluido en este tipo de técnicas, es necesario entender tanto el método original de detección de enfermedades que afectan al sistema respiratorio como el diagnóstico de los expertos con los resultados de este tipo de pruebas. En este caso además contemplamos los avances relacionados en cuanto a los algoritmos de visión computacional con inteligencia artificial para la clasificación de imágenes.

- Técnicas de reconocimiento de imágenes con redes neuronales convolucionales (CNNs). Una vez vimos la relevancia de este tipo de arquitecturas a la hora de discriminar patologías respiratorias a partir de radiografías de tórax, estudiamos el origen de estas técnicas y sus avances, tanto a nivel algorítmico como a nivel de datasets o conjuntos de datos disponibles.

- Computación en el borde o *Edge Computing*. Comprobamos el estado actual de este tipo de paradigma, qué tipo de soluciones existen, características que cumplen y los últimos avances relacionados con el campo junto con sus beneficios asociados.

- Inteligencia artificial aplicada en *Edge*. Al igual que el caso anterior pero centrándonos en las soluciones existentes del sector de la inteligencia artificial, donde vimos las ventajas y avances de este tipo de computación aplicada a la IA así como los diversos sistemas hardware y software soportados a día de hoy.

- Tecnología de contenedores, partiendo del *Cloud* al *Edge*. Este es quizás uno de los puntos más relevantes en desarrollo e implementación de software a día de hoy. Si bien hace escasos años todos los paradigmas de programación y servicios estaban claramente orientados a la infraestructura Cloud, se ha visto como tendencia un desplazamiento de estas operaciones hacia dispositivos cada vez más interconectados (*IoT*) que para evitar saturar las redes de comunicaciones y los servidores en Cloud empiezan a gestionar cada vez más información de manera autónoma, lo que da pie al

paradigma de computación en *Edge* y el uso de estas tecnologías de aislamiento en contenedores en los propios equipos.

- Soluciones de orquestación de contenedores. Nos centramos en ver las soluciones actuales de orquestación para gestionar contenedores como Kubernetes y las distribuciones específicas para equipos pequeños embebidos como K3S para poder utilizarlas en el proyecto. Detallamos la utilización de estas tecnologías, qué componentes utilizan y sus evoluciones más recientes.
- TensorFlow, siendo de las plataformas de Machine Learning de código fuente abierto más extendidas en la actualidad. Detallamos sus componentes, arquitectura y evolución, junto con los requisitos para nuestro caso de uso particular.
- Keras, la API de alto nivel desarrollada para poder trabajar con redes neuronales bajo TensorFlow de manera más rápida y sencilla. Actualmente estándar de uso de este tipo de redes en TensorFlow.

3.3.5 Construcción del sistema

Una vez determinados los componentes hardware y ver los accesorios de que disponíamos, procedimos a determinar los componentes necesarios para el montaje del clúster : carcasa, fuentes de alimentación, refrigeración adicional, configuración de los jumpers de las placas para modo de alto consumo, conexiones y cableado de red...

Al terminar los requisitos hardware, se instala el sistema operativo de referencia en ambas placas al igual que el resto de paquetes de software necesarios para la interacción entre ellas.

3.3.6 Base de datos de referencia (Datasets)

Inicialmente se realiza una búsqueda sobre los conjuntos de datos de referencia utilizados en las publicaciones y documentos científicos previamente estudiados.

Una vez vista la disponibilidad de las bases de datos (si son de libre acceso o no), seleccionamos una de ellas y procedimos a la selección de imágenes en dos categorías: casos positivos y casos negativos.

Esta selección de casos se va refinando durante todo el proyecto tanto en número de imágenes a utilizar como en la división de los casos positivos/negativos en los conjuntos de entrenamiento, validación y testeo, para poder obtener la mejor solución con el mismo algoritmo inicial.

3.3.7 Adecuación de Base de datos

Debido a las restricciones hardware de nuestro caso de uso, realizamos un estudio visual de las imágenes tratadas basándonos en la resolución original, el detalle y la claridad de las mismas. En este caso determinamos que el conjunto de datos originalmente seleccionado debía ser modificado, repitiendo el proceso hasta obtener una base de datos de imágenes con la calidad suficiente acorde a nuestros requisitos.

3.3.8 Adecuación de Algoritmo base

Una vez seleccionada la arquitectura de redes neuronales CNN para el reconocimiento de imágenes, realizamos pruebas en el equipo sobremesa empleado como referencia para sacar información relativa a los parámetros empleados, tiempos de procesamiento, carga de procesador, carga de GPU, memoria...

Estos datos nos adelantan la imposibilidad de poder utilizar este algoritmo en su estado inicial con los parámetros por defecto, por lo que procedemos en diversas iteraciones a modificar los parámetros para obtener el mejor caso de ejecución tanto a nivel de resultados obtenidos como de ajuste de proceso del hardware elegido para el proyecto.

3.3.9 Ejecución de algoritmos

Después de haber obtenido tanto el algoritmo más optimizado como la base de datos de imágenes necesarias, procedemos a ejecutar el algoritmo junto con el conjunto de datos en las diversas plataformas disponibles con el objetivo de conseguir unos resultados a comparar en el entrenamiento de la red.

Si bien es cierto que debido a las características hardware de la placa Jetson y a los resultados iniciales obtenidos en cuanto a la carga de trabajo en los casos de placa individual y ejecución en clúster, llegamos a determinar que en el caso del clúster el rendimiento y uso de recursos puede ser incrementado un poco más, por lo que realizaremos un posterior ajuste para incrementar el rendimiento y poder compararlo con el resto de equipos (excepto el caso de una única placa, que ya no puede ser comparado a igualdad de condiciones).

3.3.10 Análisis y comparativas

Una vez realizados los trabajos de entrenamiento e inferencia en las diferentes plataformas utilizando los algoritmos y las bases de datos optimizadas, disponemos de todas las métricas para medir rendimiento, resultados intermedios y finales entre los diferentes equipos.

3.3.11 Conclusiones

Debido a los trabajos y métricas obtenidas en el análisis comparativo, podemos obtener conclusiones para cada uno de los ámbitos comparados, incluyendo la ampliación de capacidad del clúster y la modificación de hiperparámetros del algoritmo, lo que refleja y confirma uno de los objetivos del proyecto.

En este apartado expondremos las confirmaciones y resultados a las preguntas y retos inicialmente planteados junto con otras conclusiones y aprendizajes obtenidos durante el desarrollo del proyecto.

Igualmente plantearemos alternativas y trabajos futuros que podrían desarrollarse a partir de este proyecto debido a la gran cantidad de variantes de arquitecturas, software, hardware etc... que están en continua evolución y mejora en este campo.

Por último, reflejamos las posibles oportunidades de negocio y los incrementos relativos de coste/resultado de cada uno de los sistemas empleados en la comparativa.

4. Descripción detallada del experimento

4.1. Descripción general de las contribuciones

Este experimento trata sobre la capacidad de incrementar la productividad de los algoritmos de inteligencia artificial empleando una infraestructura de hardware dedicada (Edge) en clúster junto con las tecnologías de clústering, contenedores y orquestación.

Debido a las premisas de apartados anteriores para nuestro caso de uso nos centramos en el análisis de rendimiento y comparativa para las siguientes plataformas:

- Workstation - PC de sobremesa
- Macbook pro Retina
- 1x nVidia Jetson Nano
- Clúster “Edge AI” (2x nVidia Jetson Nano)
- 1x nVidia Jetson Xavier NX

Las características tanto de software (Sistema Operativo) como hardware (CPU, Memoria RAM, capacidades de GPU y consumo promedio) para cada uno de los equipos utilizados está disponible en la siguiente tabla.

Tabla 2. Descripción de los sistemas empleados en la comparativa

	CPU	RAM	GPU				Consumo	S.O.
			Modelo	GFLOPs	NVIDIA CUDA® Cores	NVIDIA Tensor Cores		
Workstation - PC de sobremesa	AMD Ryzen 1800X	16 GB DDR4 3200	RTX 2070 8GB Arquitectura NVIDIA Turing™	6497	2304	288	GPU 185 W Global max 1200W	Windows 10
Apple MacBook Pro Retina	Core I7 4850HQ	16 GB DDR3 1600	GeForce GT 750M Mac Edition de NVIDIA Kepler	711.2	348		250W	MacOS X Big Sur 11.5.22

1x nVidia Jetson Nano	Procesador ARM® Cortex®-A57 MPCore de cuatro núcleos	LPDDR4 de 4 GB y 64 bits	Arquitectura NVIDIA Maxwell™	472	128	-	5-10 W	Linux base
Cluster Edge AI (2 X nVidia Jetson Nano)	2 X (ARM® Cortex®-A57)	8 GB LPDDR4 64 bit	Arquitectura NVIDIA Maxwell™	944 (aprox.)	256 (aprox.)	-	10-20 W	Linux base
1x nVidia Jetson Xavier NX	CPU NVIDIA Carmel ARM®v8.2 de 64 bits de 6 núcleos 6 MB L2 + 4 MB L3	LPDDR4x de 8 GB y 128 bits a 51,2 GB/s	GPU NVIDIA Volta	21000	384	48	10-15 W	Linux base

Presentamos los diferentes experimentos realizados de forma detallada incluyendo en cada caso:

- Diseño de la solución y características tanto Hardware como Software.
- Detalle de los algoritmos ejecutados, parámetros de ajuste y datos relevantes de bases de datos (*dataset*)
- Ajustes y modificaciones realizadas en caso de ser necesarias, tanto a nivel del algoritmo utilizado, como de bases de datos de imágenes (*dataset*) y su justificación.
- Entorno detallado a nivel de arquitectura en Edge.
- Comparaciones y métricas empleadas.
- Innovaciones presentadas y características relevantes.

4.2. Consideraciones para el diseño de la solución y de los experimentos

La solución propuesta de arquitectura en clúster con hardware embebido, contenedores, Kubernetes y paralelización de GPUs ha sido implementada sobre la base de dos placas nVidia Jetson Nano, por lo que inicialmente podríamos suponer un incremento teórico del doble de rendimiento (x2) en todos los aspectos con respecto a la ejecución en una única placa.

Si bien este rendimiento teórico en la realidad es inalcanzable debido a múltiples factores (carga del sistema en la gestión del clúster, comunicaciones entre los nodos de las placas, orquestación, etc...) expondremos los parámetros y resultados obtenidos llevando al límite la capacidad de dicho clúster y la de una placa de ejecución en solitario, para obtener unos resultados con una comparativa lo más equilibrada posible.

De esta forma la ejecución del mismo algoritmo y el uso de la misma base de datos ajustada para el mayor rendimiento en una única placa será considerado holgado para su ejecución en el resto de plataformas de la comparativa, por lo que se modificará y adaptará para exprimir las características del clúster y poder realizar la comparativa de métricas con valores máximos con el resto de plataformas disponibles.

4.3 Entrenamiento y pruebas en equipo sobremesa

Este fue el primer equipo empleado para la valoración y ejecución tanto de algoritmos como de conjuntos de datos, ya que es una estación de trabajo habitual con menos restricciones en cuanto a características técnicas y que permitió realizar valoraciones de manera rápida.

Es necesario tener en cuenta que el otro equipo utilizado, la placa nVidia Xavier NX, no estuvo disponible durante todo el proyecto y únicamente de manera puntual para la comparativa.

- Pruebas iniciales y problemas con el conjunto de datos

Como se detalla en la tabla comparativa de equipamiento hardware, disponemos de un equipo sobremesa de tipo *Workstation* destinado a un uso generalista, aunque orientado al sector "*Gaming*", lo cual implica unas mayores capacidades en cuanto a recursos que son aplicables directamente a mejorar los campos de algoritmos de inteligencia artificial y la capacidad de memoria disponible.

Figura 27. Equipo sobremesa

Además de los datos de la tabla, como datos relevantes hay que contemplar que el equipo cuenta con un almacenamiento de tipo SSD M2 con velocidades de lectura/escritura de 3500/2700 MB/sec.

En cuanto al sistema operativo del equipo es un Windows 10 Profesional, con la plataforma Anaconda para inteligencia artificial instalada para desarrollo, lo que incluye Python 3, las bibliotecas necesarias, la versión 6.1.4 de Jupyter Notebook como entorno de desarrollo y la versión 2.4.1 de TensorFlow con soporte de CPU.

En lo relativo a la base de datos utilizada en este caso, inicialmente optamos por la detallada en el proyecto "*COVID-19 Image Data Collection: Prespective Predictions Are The Future*" (Cohen et al., 2020).

Pero tras realizar varias ejecuciones del algoritmo tanto en esta plataforma como en el entrenamiento en contenedores (1 nodo), y ver los resultados vimos que debíamos realizar un mejor ajuste de la misma, realizando por lo tanto un análisis de la calidad de los datos contenidos en dicha base de datos.

Tras realizar varias ejecuciones del algoritmo tanto en esta plataforma como posteriormente en el entrenamiento en contenedores en un único nodo (1x Jetson Nano) vimos que debíamos realizar un mejor ajuste del conjunto de datos y de las características de las imágenes.

- Análisis de la base de datos de imágenes

En primera instancia como podemos observar en la imagen a continuación, no todas las imágenes de la data set siguen la misma tipología de radiografía. Algunas de ellas están tomadas de manera lateral al individuo, e incluso de forma cenital:

Figura 28. Fragmento de imagen de data set inicial.(Cohen et al., 2020)

Realizamos por lo tanto una primera iteración para eliminar a simple vista las imágenes que no fueran tomadas de forma frontal (la mayoría) y que se observe al menos la caja torácica completa.

Una vez realizada esta primera criba volvimos a ejecutar el algoritmo pero los resultados, aunque un poco mejores, todavía distaban de ser los esperados.

Debido a esto volvemos a realizar un análisis más profundo del *dataset*, obteniendo en este caso los detalles de resolución y tamaño de las imágenes. Y como ya suponíamos nos encontramos con una relación de calidades y tamaños excesivamente heterogénea.

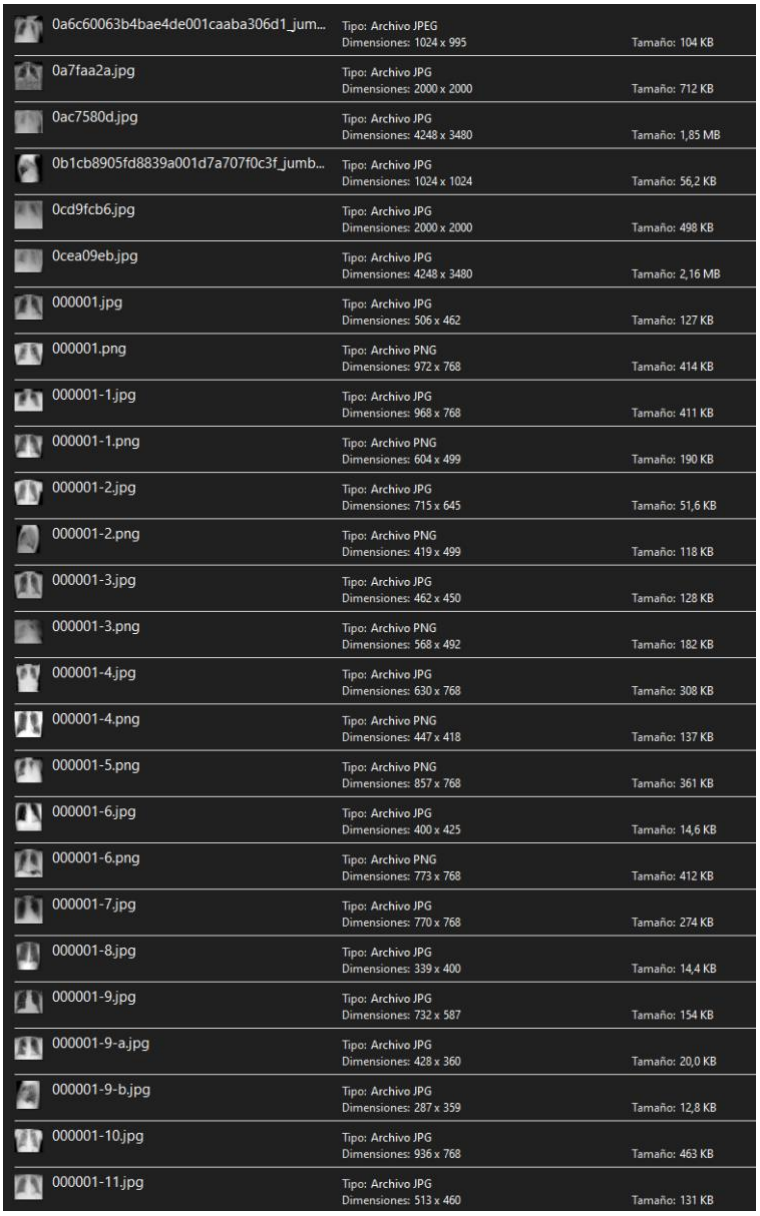


Figura 29. Fragmento de imagen con los detalles de resolución y tamaño de data set inicial. (Cohen et al., 2020)

Una vez obtenido los resultados de esta segunda iteración de análisis, decidimos realizar una nueva búsqueda de data sets disponibles que cumpliesen con los requisitos de homogeneidad tanto a nivel de detalle, tamaño y resolución, como de ser radiografías de tórax completo tomadas de forma frontal.

En esta nueva búsqueda de *datasets* relacionadas con el proyecto, nos encontramos con varios conjuntos de imágenes más recientes y de mejor calidad que el planteado inicialmente.

Tras realizar de nuevo un análisis preliminar de los resultados de la nueva búsqueda, optamos por el empleado en Deep-COVID: "*Predicting COVID-19 From Chest X-Ray Images Using Deep Transfer Learning"* (Minaee et al., 2021), en la cual se referencian los siguientes valores generales para dicho *dataset*:

Tabla 3. Esquema de elementos del dataset utilizado

Split	COVID-19	Non-COVID
Training Set	84 (420 *after augmentation*)	2000
Test Set	100	3000

- Carga de dataset y etiquetado

Como primer paso realizamos la carga y normalización de las imágenes en cuanto a tamaño (224x224 píxeles, definido a raíz del modelo de red convolucional seleccionado en el siguiente paso) y canales de color (RGB) utilizando la biblioteca OpenCV, además de asignación de etiquetas (*labels*) según el directorio donde están localizadas para discriminar los resultados, tal y como se muestra en el fragmento de código siguiente :

Carga de imagenes y creación de labels para entrenamiento

```
# grab the list of images in our dataset directory, then initialize
# the list of data (i.e., images) and class images

# Carga de imagenes del dataset
print("[INFO] Cargando imágenes...")
imagePaths = list(paths.list_images("dataset"))
data = []
labels = []

# loop over the image paths
for imagePath in imagePaths:
    # extract the class label from the filename
    label = imagePath.split(os.path.sep)[-2]

    # load the image, swap color channels, and resize it to be a fixed
    # 224x224 pixels while ignoring aspect ratio
    image = cv2.imread(imagePath)
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    image = cv2.resize(image, (224, 224))

    # update the data and labels lists, respectively
    data.append(image)
    labels.append(label)

# convert the data and labels to NumPy arrays while scaling the pixel
# intensities to the range [0, 1]
data = np.array(data) / 255.0
labels = np.array(labels)
```

Figura 30. Fragmento de algoritmo base - Normalización de imágenes y etiquetado

Una vez realizada la normalización y etiquetado de las imágenes, dividimos el conjunto de datos en conjuntos de entrenamiento y conjuntos de test.

- Parametrización del modelo de red convolucional

El modelo elegido de arquitectura de red neuronal convolucional es el VGG16. Este modelo está soportado por la biblioteca Keras, está disponible en todas las plataformas, cubre los requisitos del hardware elegido y es capaz de proveer una precisión suficiente como para obtener unos resultados válidos en la detección.

```
1 from tensorflow.keras.applications import VGG16
2
3 baseModel = VGG16(weights="imagenet", include_top=False,
4     input_tensor=Input(shape=(224, 224, 3)))
```

Figura 31. Fragmento de algoritmo base – Selección de modelo VGG16

Como se puede observar en la ilustración anterior, los parámetros del modelo incluye la selección de pesos ya preparada con el dataset ImageNet (comentado también en el estado del arte y los puntos anteriores). El valor por defecto de las imágenes de entrada es de 224x224 pixeles con 3 niveles de color (RGB), de ahí la decisión del paso anterior de redimensionar las imágenes a esta resolución concreta.

Además, el parámetro "`include_top=False`" es añadido para indicarle a Keras que las últimas capas del modelo (las más específicas) no las incluya.

- Transfer Learning y definición de cabecera

El motivo de no incluir las últimas capas a la hora de cargar el modelo sirve para poder realizar el proceso llamado Transfer Learning. Este proceso sirve para poder reentrenar un modelo de red neuronal complejo, como en este caso el VGG16 para la detección de los objetos del dataset ImageNet, en detectar nuevos casos con nuevas características, aprovechando lo aprendido en las capas más "profundas" del modelo en su entrenamiento inicial.

La forma tradicional de realizar esta operación es bloquear o congelar el modelo completo excepto sus últimas capas (donde se decide la clasificación final) y reentrenar estas últimas con los nuevos objetivos a clasificar.

La biblioteca Keras permite realizar esta tarea directamente con los parámetros de `include_top` y la particularización del modelo con inputs (parte congelada del modelo anterior, definida como `trainable=False`) y outputs (nueva cabecera con nuevas características para entrenar).

Creación de nueva cabecera de red neuronal y congelamiento del resto

```
# construct the head of the model that will be placed on top of the
# the base model
headModel = baseModel.output
headModel = AveragePooling2D(pool_size=(4, 4))(headModel)
headModel = Flatten(name="flatten")(headModel)
headModel = Dense(64, activation="relu")(headModel)
headModel = Dropout(0.5)(headModel)
headModel = Dense(2, activation="softmax")(headModel)

# place the head FC model on top of the base model (this will become
# the actual model we will train)
model = Model(inputs=baseModel.input, outputs=headModel)

# loop over all layers in the base model and freeze them so they will
# *not* be updated during the first training process
for layer in baseModel.layers:
    layer.trainable = False
```

Figura 32. Fragmento del algoritmo base – Cabecera para reentrenamiento

Los parámetros utilizados para definir la nueva cabecera a entrenar vienen dados por la siguiente tabla:

Tabla 4. Arquitectura de la nueva cabecera para reentrenamiento

PARAMETRO	DESCRIPCIÓN
baseModel.output	Definición de modelo base para salida (output).
AverageAverage Pooling 2D (pool_size=(4,4)	Al eliminar los primeros 3 niveles de la red neuronal, debemos especificar la salida del tensor 4D del último bloque convolucional. Esta agrupación promedio se aplicará a la salida y debe ser un tensor bidimensional, con valores 4,4 en este caso.
Flatten	Compacta el vector resultado de las convoluciones anteriores.
Dense (64, activation="relu")	Capa densa con función de activación Relu de 64 unidades.
Dropout(0.5)	Capa Dropout con una frecuencia de 0.5 para prevenir overfitting cambiando valores de entrada por ceros.
Dense (2, activation="softmax")	Capa densa con función de activación SoftMax para clasificar los resultados (positivo/negativo).

- Re-entrenamiento del modelo con nueva cabecera

Una vez definida la nueva cabecera, procedemos al reentrenamiento. Los parámetros iniciales utilizados son :

- Epochs para entrenamiento : 25
- Optimizador Adam con ratio de aprendizaje $1 * 10^{-3}$ y valor de decay $\frac{1*10^{-3}}{25}$
- Batch Size : Inicialmente probamos con valores empezando en 8, pero debido a las sucesivas pruebas y limitaciones en el clúster (ver capítulo 4.2.3) el valor de batch size tuvo que reducirse a 1.
- Función de pérdida para clasificación : `binary_crossentropy`
- Métricas relevantes para el entrenamiento : `accuracy` (precisión)

Transfer Learning - Entrenamiento de nueva cabecera

```
# Parámetros de training
INIT_LR = 1e-3
EPOCHS = 25
#BS = 8
#BS = 4
BS = 1

# Optimizador
opt = Adam(lr=INIT_LR, decay=INIT_LR / EPOCHS)
model.compile(loss="binary_crossentropy", optimizer=opt,
	metrics=["accuracy"])

# Entrenamiento
print("[INFO] training head...")
#H = model.fit_generator(
#	trainAug.flow(trainX, trainY, batch_size=BS),
#	steps_per_epoch=len(trainX) // BS,
#	validation_data=(testX, testY),
#	validation_steps=len(testX) // BS,
#	epochs=EPOCHS,
#    verbose=1)

H = model.fit(
    trainX, trainY, batch_size=BS,
	steps_per_epoch=len(trainX) // BS,
	validation_data=(testX, testY),
	validation_steps=len(testX) // BS,
	epochs=EPOCHS,
    verbose=1)
```

Figura 33. Fragmento del algoritmo base – Re-entrenamiento del modelo

El código completo del algoritmo base utilizado está disponible en el Anexo correspondiente.

Lanzamos el algoritmo con diferentes subconjuntos de imágenes para entrenamiento y tests para valorar la carga del sistema.

- 25 imágenes positivas contra 75 negativas
- 100 imágenes positivas contra 100 negativas
- 184 imágenes positivas contra 184 negativas
- 184 imágenes positivas contra 368 negativas

La elección inicial del número de imágenes vino condicionada por el entrenamiento en las placas Jetson, como puede verse en el siguiente punto 4.2.3 en la sección correspondiente.

Finalmente optamos por el dataset de 184 imágenes positivas contra 184 imágenes negativas para poder cubrir todos los casos.

4.4 Entrenamiento y pruebas en portátil Macbook Pro

Empleando el algoritmo seleccionado con el datase de 184 imágenes positivas y negativas, pasamos a ejecutarlo en un equipo portátil , equipado con Sistema operativo MacOS X en su versión Big sur 11.5.2, con la versión de TensorFlow 2.5.0 para MacOS X con tensorflow-metal 0.1.2 (este con soporte GPU Metal).

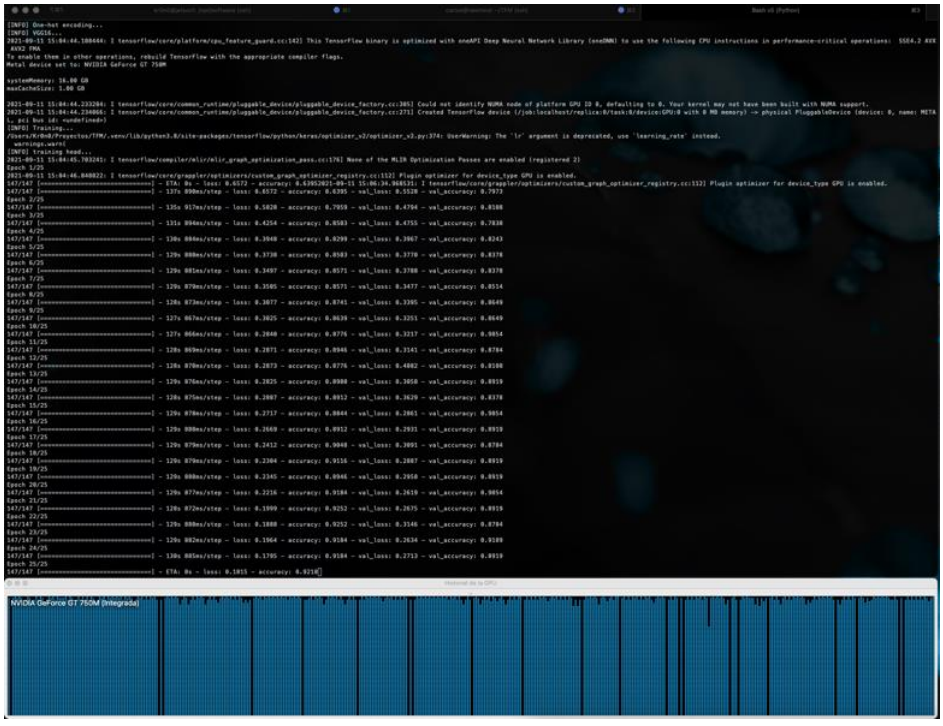


Figura 34. Entrenamiento del modelo usando tensorflow-metal en Macbook Pro

4.5 Montaje de infraestructura en clúster

HARDWARE

Debido al uso en modo clúster de las placas se ha optado por utilizar dos fuentes de alimentación de 5V/4A para poder utilizarse en modo de máximo consumo. Además de esto, se han utilizado dos ventiladores de 5V alimentados directamente desde la placa como sistema de refrigeración adicional de la marca iBest y un armazón de clúster de la marca GeekPi, que añade el soporte de un ventilador frontal de 5V añadido para refrigeración. En cuanto a la configuración de red, las dos placas están conectadas a un Switch GigaLAN con cables UTP categoría 6 para la máxima velocidad disponible en comunicaciones, y en cuanto al almacenamiento se ha optado por dos tarjetas mSD Sandisk U10 de 64GB.

Figura 35. Hardware adicional

Además de esta configuración conectamos en ambas placas el jumper J48 para poder habilitar el modo de máximo rendimiento según el manual de usuario de Jetson Nano (nVidia, 2020)

Con esta infraestructura garantizamos una buena refrigeración y comunicación a máxima potencia de utilización en las placas, con suficiente espacio para realizar todas las pruebas necesarias.

SISTEMA OPERATIVO

En cuanto a nivel software se ha utilizado como base el sistema operativo GNU/Linux proporcionado por el fabricante para estas placas, Linux 4 Tegra 32.5.0, incluido en el JetPack 4.5 oficial de nVidia, con una serie de particularidades:

- Aumento de memoria Swap a 16GB (utilizando el script de JetsonHacks disponible en el repositorio (Repository, 2019))
- Habilitación del perfil de máximo rendimiento (*Best performance - MAXN*) con la herramienta *nvpmodel* de nVidia.
- Configuración de red habilitada por DHCP con la siguiente configuración en lado servidor:

Tabla 5. Configuración de red del clúster

NOMBRE	DIRECCIÓN IP / MASCARA	DIRECCIÓN MAC
jetson1.local	192.168.0.81 / 255.255.255.0	00:04:4b:e4:1a:2f
jetson2.local	192.168.0.82 / 255.255.255.0	00:04:4b:e4:10:2f

Para realizar toda esta configuración de manera automática nos apoyaremos en varios scripts disponibles en los Anexos del presente documento y un servicio específico para configurar las placas al arranque utilizando systemd.

PAQUETES Y RUNTIMES

Tal y como hemos comentado es necesario el soporte de paquetes para la gestión de contenedores y el propio clúster de Kubernetes.

Para la gestión de contenedores con soporte de GPU en la propia imagen vienen preinstalados los paquetes necesarios instalados mediante la herramienta estándar apt (docker.io, containerd, nVidia-docker2).

Para la gestión del clúster es necesario descargar los binarios de k3s y kubectl de las webs correspondientes y desplegarlos en la ruta `/usr/local/bin` con permisos de ejecución.

- K3S: (K3S, 2021)
- Kubectl: (k8s, 2021)

CONFIGURACIÓN DEL CLUSTER K3S

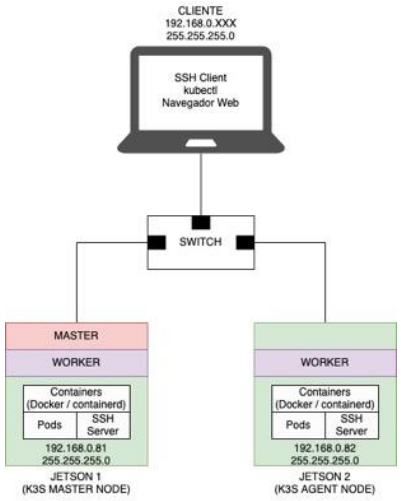


Figura 36. Arquitectura de comunicaciones del clúster k3s

La configuración del clúster implica crear un nodo maestro (*master*) para la gestión del clúster y dos nodos trabajadores (*worker)* para el despliegue de los Pods con sus contenedores. Para este caso usaremos una configuración donde Jetson1 será el nodo *master* y a su vez estará configurado como *worker* para poder realizar además las tareas correspondientes de entrenamiento e inferencia.

La configuración se realizará con una clave común, asignando las IPs de cada uno de los nodos y especificando en el *worker* la dirección del master para registrarse en el clúster.

Para la gestión desde el ordenador cliente se puede utilizar el fichero generado por k3s (*k3s.yaml*) que permite conectarse al nodo maestro y gestionar toda la configuración del clúster, o bien utilizar la misma herramienta desde el nodo principal.

En el Anexo están disponibles tanto los ficheros de configuración como los scripts de inicio y la configuración cliente para recrear y utilizar esta configuración.

```
NAME      STATUS   ROLES           AGE    VERSION
jetson1   Ready    master,worker   148d   v1.19.7+k3s1
jetson2   Ready    worker          118d   v1.19.7+k3s1
```

Figura 37. Captura del estado del clúster K3S con los dos nodos en ejecución

Una vez realizada la configuración base de los sistemas procederemos al siguiente paso, comprobar el soporte inicial de TensorFlow en contenedores Docker con GPU en los nodos.

4.6 Entrenamiento en clúster

Para realizar el entrenamiento en clúster se realizarán varias fases para validar las posibilidades, empezando desde el soporte inicial de contenedores en los nodos, posteriormente el entrenamiento del modelo en contenedores en un único nodo para pasar posteriormente al entrenamiento en clúster con un nodo y posteriormente en contenedores con los dos nodos de manera distribuida. Para cada caso veremos los resultados, limitaciones encontradas, *workarounds* aplicados y comparativas.

4.6.1 Soporte inicial en contenedores (2 nodos)

Para el soporte inicial es necesario realizar las pruebas según el cuadro siguiente :

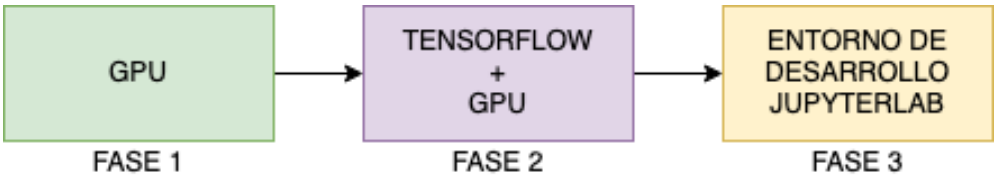


Figura 38. Fases del soporte inicial de contenedores.

En una primera fase se debe conseguir acceso completo a la GPU desde los contenedores y el clúster. En la segunda fase una vez validado el soporte, pasamos a validar el uso de TensorFlow desde los contenedores para poder ejecutar el resto de tareas de la prueba de concepto, y para terminar un entorno de desarrollo que sea fácilmente accesible donde poder desarrollar los futuros pasos de entrenamiento e inferencia, como JupyterLab.

Para cada uno de estos pasos es necesario crear la imagen del contenedor utilizando la herramienta Docker y varios Dockerfiles, pero hay que tener en cuenta que el clúster Kubernetes utiliza el *runtime* containerd para ejecutar los contenedores, por lo que será necesario validar la ejecución del contenedor con containerd antes de probar en el clúster.

Contenedor con soporte GPU – Fase 1

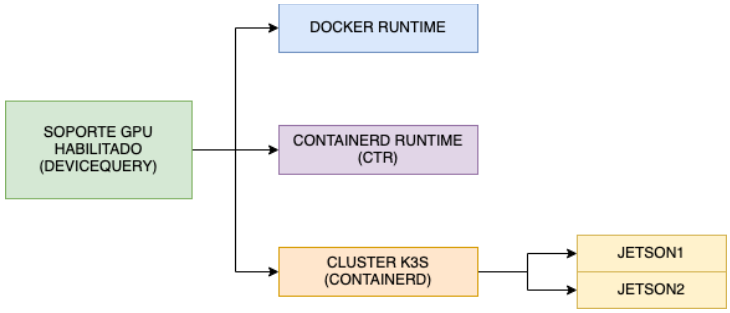


Figura 39. Fases de soporte de GPU para contenedores.

Para el primer paso, el proveedor nVidia provee de un software demo llamado deviceQuery incluido en el sistema operativo que permite consultar el acceso a la GPU integrada en los nodos, dando una descripción detallada del hardware disponible.

Para realizar esta prueba será necesario crear una imagen Docker con esta herramienta que esté basada en la imagen Docker de Linux para Tegra acorde al sistema operativo (l4t-base:r32.5.0) y disponible desde el repositorio de *nVidia Container Registry* (nvcr.io), y ejecutarla con el soporte de *runtime* Docker de nVidia. (Ver Anexo I)

Una vez creados los contenedores comprobamos que la ejecución de la herramienta y el acceso a GPU son satisfactorios, con `Result = PASS` con la imagen generada.

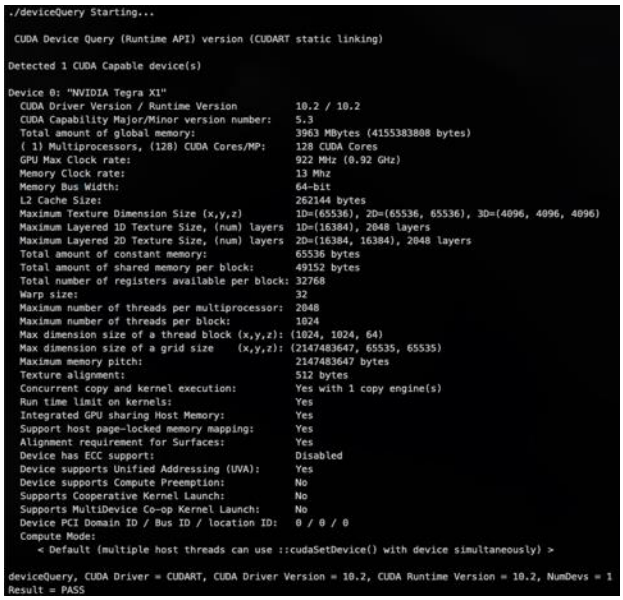


Figura 40. Captura de resultados de deviceQuery con detección de GPU positiva

Para realizar el paso de ejecución en el clúster es necesario primero cambiar la configuración del *runtime* de ejecución de k3s para que utilice el *runtime* de nVidia en containerd (y así el acceso a GPU) para todos los nodos del clúster, además de realizar una prueba de ejecución de este mismo contenedor con containerd y su herramienta ctr de manera aislada. De esta forma primero comprobamos que el soporte con containerd funciona y posteriormente que el soporte en el clúster también funciona (al utilizar el mismo *runtime*).

Para el caso del containerd, utilizando la herramienta ctr se ve que la ejecución funciona correctamente en los logs de la ejecución del contenedor (ver Anexos) de manera individual, por lo que podemos pasar a la ejecución en clúster.

Para la ejecución en clúster se ha creado un Pod para despliegue con esta imagen generada y probaremos a desplegarlo en los dos nodos, jetson1 y jetson2, utilizando la directiva nodeName. La ejecución se podrá realizar desde el cliente conectado al clúster con la herramienta kubectl, lo que muestra el resultado correcto de detección de GPU en los dos nodos.

Una vez realizado este proceso ya hemos validado la fase primera con el soporte de GPU en contenedores tanto de manera individual como en el clúster Kubernetes.

Contenedor con soporte TensorFlow – Fase 2

Una vez comprobado el soporte de GPU en los contenedores, pasamos a comprobar el soporte de la biblioteca TensorFlow y las herramientas de Python necesarias para los siguientes pasos.

Para esto nos basaremos en la misma imagen utilizada para el caso anterior proporcionada por nVidia, donde instalaremos TensorFlow y las dependencias asociadas.

Para esta parte ya no es necesario realizar la validación de manera individual en containerd, por lo que el flujo de trabajo pasa a ser el siguiente:

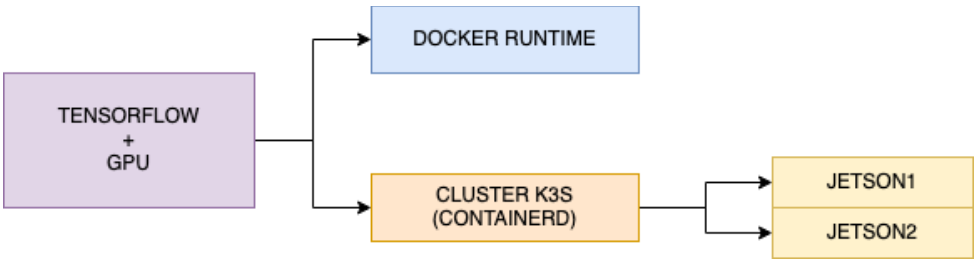


Figura 41. Esquema de fase de soporte TensorFlow con GPU.

Es importante tener en cuenta que TensorFlow en el caso de las placas Jetson es proporcionado directamente por nVidia para tener el soporte por GPU, por lo que debemos descargarlo directamente desde el propio repositorio oficial que proporciona nVidia y solventar las posibles dependencias acorde a esta versión.

En este caso para el programa a ejecutar en el contenedor utilizaremos la biblioteca `device_lib` y `list_local_devices ()` de TensorFlow de forma análoga a la fase anterior y

poder ver el listado de dispositivos con soporte GPU que son detectados y pueden ser utilizados por TensorFlow

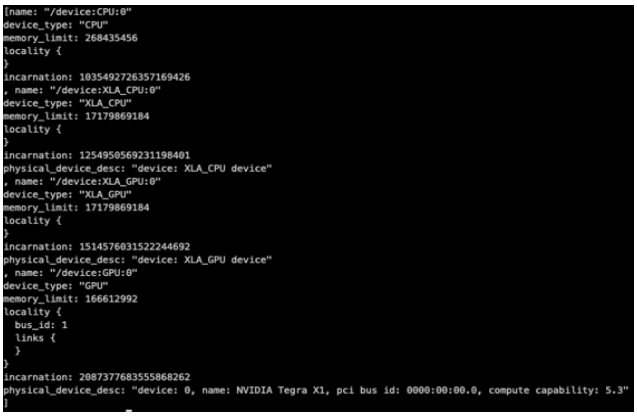


Figura 42. Captura de soporte de GPU bajo TensorFlow.

Una vez comprobado en la imagen Docker local, pasamos a realizar el mismo trabajo en el clúster. Para esto creamos un Pod para desplegar en el clúster que únicamente ejecute un comando `sleep` y dentro ejecutamos directamente el código Python desde el propio contenedor, dando el mismo resultado que en la ejecución anterior y por lo tanto validando esta fase.

Entorno de trabajo – Fase 3

Una vez validados los pasos anteriores, procederemos a crear un entorno de desarrollo donde poder realizar las tareas de entrenamiento e inferencia, utilizando JupyterLab y el acceso web que proporciona para poder trabajar y gestionar las dependencias.

Seguiremos un flujo de trabajo similar a la fase anterior, creando primero la imagen Docker y desplegándolo en un nodo para después desplegarla en el clúster como un *deployment* para poder utilizarlo en los pasos siguientes (Aunque en este caso al ser un entorno puro de desarrollo no sería estrictamente necesario).

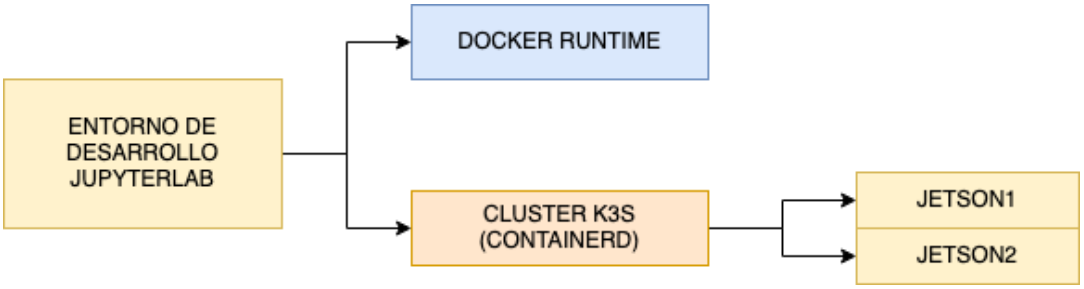


Figura 43. Esquema de fase de creación de entorno de trabajo.

Hay que tener en cuenta que JupyterLab tiene un acceso al servidor web mediante un token autogenerado, por lo que también es necesario comprobar el token generado en los logs del clúster para utilizar la clave y poder acceder al entorno de desarrollo.

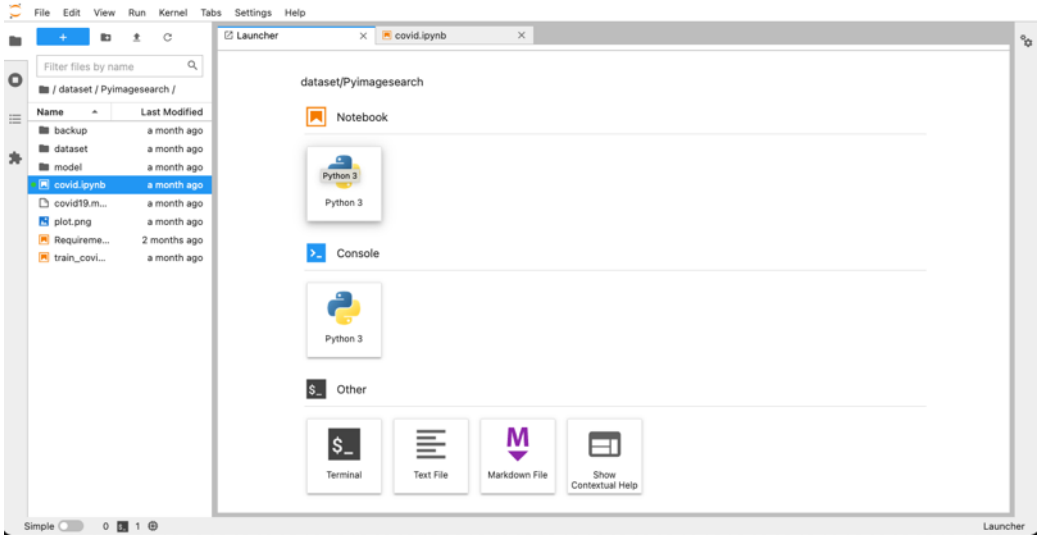


Figura 44. Captura de entorno de desarrollo JupyterLab desplegado en clúster.

Una vez accedemos al entorno, ya podemos empezar a trabajar en los siguientes puntos: Entrenamiento e Inferencia.

4.6.2 Entrenamiento en contenedores (1 nodo)

Para llevar a cabo este caso de uso, se implementa un contenedor que ejecute la versión adecuada de Python únicamente en una de las placas nVidia Jetson Nano.

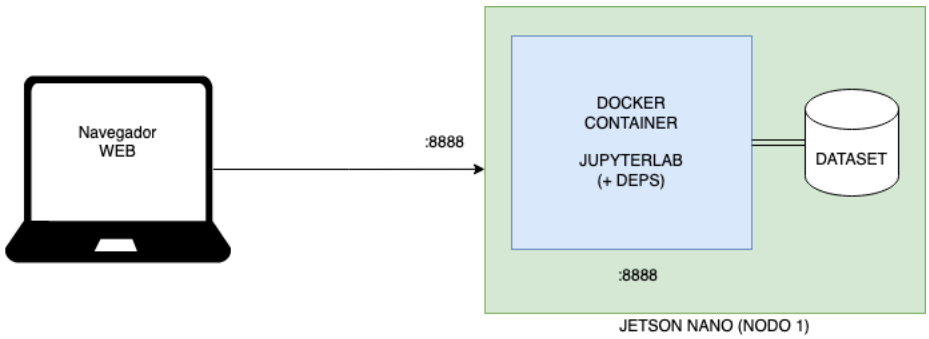


Figura 45. Esquema de entrenamiento en contenedores con un nodo.

El primer punto a contemplar es realizar el entrenamiento de la red neuronal en una única placa. Para esto utilizamos inicialmente el entorno de desarrollo del paso anterior con JupyterLab para importar y particularizar el código utilizado en el ordenador de sobremesa, para después exportarlo en un contenedor dedicado con las dependencias asociadas.

En este punto empezamos a ver las propias limitaciones del entorno embebido con respecto al sistema sobremesa, sobre todo de cara al uso de la memoria RAM del equipo.

En nuestro caso utilizamos 16GB de RAM en Swap (archivo de disco) configuradas al inicio del plataformado de las placas, pero es importante tener en cuenta que la memoria Swap puede ser utilizada por la CPU, pero NO puede ser utilizada por la GPU, por lo que la limitación en cuanto al tamaño de la memoria a utilizar es relevante en nuestro caso.

Al realizar la carga del modelo VGG16 inicialmente para posteriormente realizar el tratamiento de las imágenes, el consumo de memoria se disparaba y a la hora de congelar el modelo para reentrenar la cabecera con los datos nuevos el proceso de TensorFlow moría con el mensaje: "`ResourceExhaustedError (see above for traceback): OOM when allocating tensor with shape[X][X]`".

Para solucionar este problema realizamos varios cambios en el código:

- Bajar el batch size a 1, como una de las soluciones planteadas en los foros de nVidia ante situaciones de este tipo.
- Redimensionar el número de imágenes a tratar para cada uno de los casos de uso (entrenamiento y test).
- Realizar primero el tratamiento de las imágenes y posteriormente la carga de VGG16 para el reentrenamiento de la cabecera.

En cuanto al conjunto de imágenes a utilizar optamos al final por el dataset de 184 imágenes positivas contra 184 negativas, ya que suponía un nivel alto de carga en la placa pero funcional, no pudiendo llegarse al caso estudiado inmediatamente posterior de 552 imágenes (184 imágenes positivas contra 368 negativas) del equipo sobremesa.

Una vez realizadas estas pruebas vimos que el entrenamiento finalmente era posible realizarlo en una placa con estos parámetros.

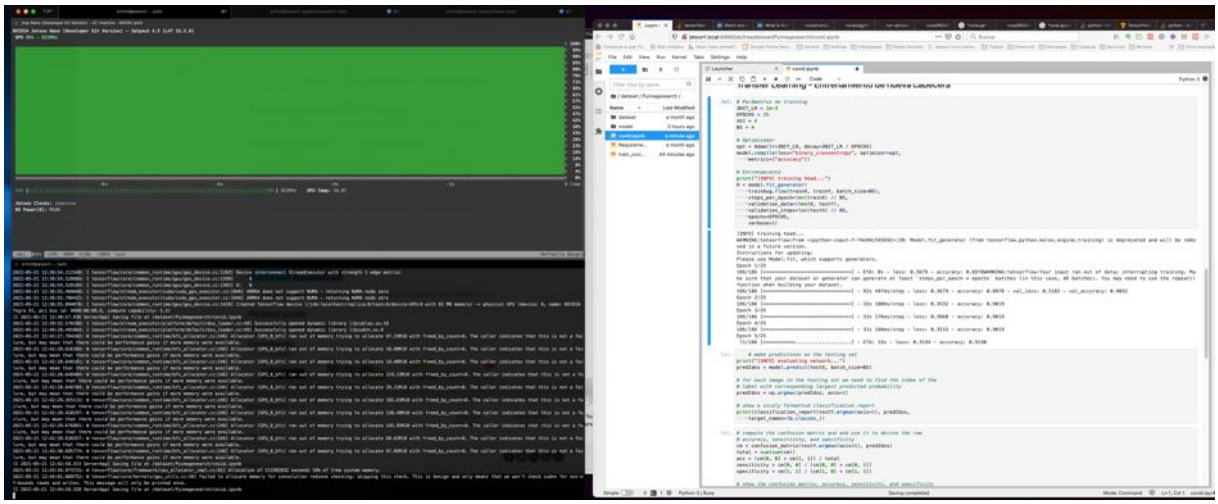


Figura 46. Capturas de entrenamiento empleando el entorno de desarrollo en una única placa/nodo.

(Podemos ver el consumo de GPU en la ventana localizada arriba a la izquierda con la herramienta jtop de nVidia)

Como en los casos anteriores, tanto los scripts de creación de imágenes de contenedores, ficheros de configuración, scripts y código utilizado están disponibles en los Anexos en las secciones correspondientes.

4.6.3 Entrenamiento en clúster Kubernetes (1 nodo)

Para llevar a cabo este caso de uso, se implementa el orquestador K3s gestionando un contenedor que ejecute la versión adecuada de Python en una única placa nVidia Jetson Nano.

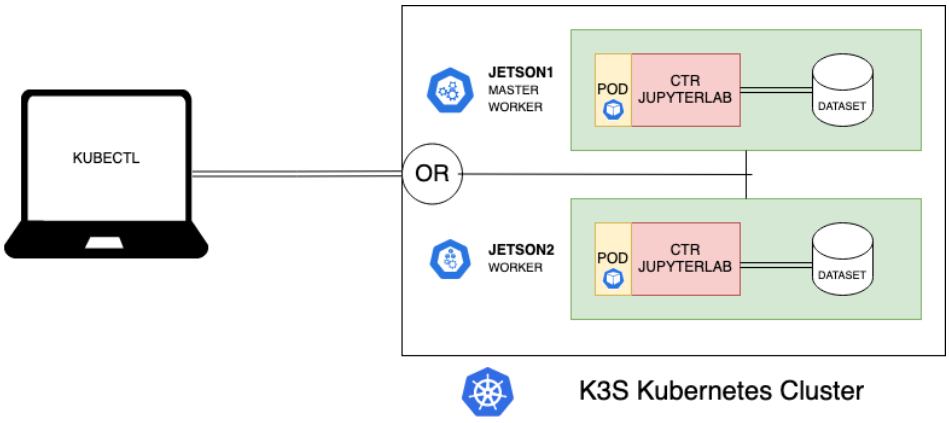


Figura 47. Esquema de entrenamiento en contenedores con un nodo (Deployment).

Una vez realizado el entrenamiento en un único nodo utilizando el entorno de desarrollo, quisimos probar el entrenamiento realizándolo en un único nodo pero con la infraestructura de Kubernetes mediante el uso del mismo contenedor con el entorno de desarrollo, pero esta vez primero como un Pod aislado y posteriormente realizando un deployment para poder acceder tanto en el nodo 1 como en el nodo 2, dependiendo de donde se ejecutara.

Las limitaciones en este caso venían dadas por el propio acceso al dataset, ya que en los casos anteriores al ejecutarlos montábamos explícitamente el dataset en el contenedor para que se pudiera acceder a las imágenes, mientras que en clúster la solución en este caso fue replicar manualmente el dataset en las dos placas (ya que uno de los objetivos era ver si podía desplegarse tanto en uno como en otro nodo y realizar el entrenamiento) y montarlo explícitamente en las mismas rutas en la configuración del deployment de Kubernetes.

Como en los casos anteriores, tanto los scripts de creación de imágenes de contenedores, ficheros de configuración, scripts y código utilizado están disponibles en los Anexos en las secciones correspondientes.

4.6.4 Entrenamiento en contenedores (2 nodos)

Para llevar a cabo este caso de uso, se implementan los contenedores necesarios para ejecutar la versión adecuada de Python en ambas placas nVidia Jetson Nano, lanzando la ejecución del algoritmo de manera distribuida utilizando estrategias distribuidas de TensorFlow.

Para este tipo de tareas consultamos la documentación de TensorFlow que tenemos disponible para las placas Jetson relativa al uso de estrategias distribuidas en su rama experimental (`tf.distribute.experimental`). Dentro de estas estrategias, la opción de MultiWorkerMirroredStrategy es la que nos sirve para nuestro caso de uso, ya que implementa entrenamiento distribuido entre diferentes nodos (llamados *workers*, con la misma nomenclatura que la utilizada en Kubernetes), donde cada uno de estos nodos puede tener disponible una o varias GPUs, siempre que sean del mismo tipo (clúster heterogéneo)

Esta estrategia replica todas las variables y los cálculos a realizar en cada uno de los nodos de trabajo, pero utiliza una implementación de algoritmo de distribución de trabajo estilo "*Allreduce*", donde cada nodo de trabajo comparte sus datos con el resto de nodos y aplica

operaciones de reducción en los tensores, de tal modo que cada GPU ejecutará una réplica del modelo y las variables estarán sincronizadas por cada Batch completo de cálculo.

Las comunicaciones entre los nodos se realizan mediante gRPC gestionado por el propio TensorFlow. Para esto es necesario definir una variable de entorno en los dos nodos llamada TF_CONFIG que contiene la información de la configuración del clúster en formato JSON :

```
{"clúster":
 {
  "worker": [
     "jetson1.local:12345",
     "jetson2.local:12345"
     ],
  "task": {
     "index": 0,
     "type": "worker"
    }
 }
}
```

El campo worker es el que define el número de nodos que van a utilizarse para entrenamiento (los worker deseables). El campo index es el que indica el número de nodo que forma parte del clúster, siendo en nuestra ejecución el valor 0 para el primer nodo (jetson1.local) y el valor 1 para el siguiente nodo (jetson2.local). Los puertos utilizados son los puertos 12345 en cada uno de los nodos.

Los datos del dataset han sido copiados a los dos nodos con la herramienta rsync para que estuvieran sincronizados a la hora de utilizarlos, aunque posteriormente en casos de uso posteriores veremos que el uso de sistemas de ficheros como NFS podría haberse utilizado desde este mismo caso en adelante.

Los cambios más importantes respecto al código del algoritmo son :

- Definición de la estrategia distribuida para TensorFlow (`tf.distribute.experimental.MultiWorkerMirroredStrategy()`)
- Uso del `strategy.scope()` desde el momento de definir el modelo base de la red neuronal hasta el entrenamiento con `model.fit()`
- Uso de la variable de entorno TF_CONFIG, que define la configuración del clúster, nodos, puertos disponibles para la conexión entre ellos

También se realizaron cambios a la hora de ejecutar los contenedores para poder pasar las variable TF_CONFIG dependiendo del nodo que fuera y añadir a la red de contenedores los hosts con las direcciones IP correspondientes, ya que de otro modo no sabrían comunicarse entre ellas.

Al ejecutarse el programa en los dos nodos, en el momento de realizar el entrenamiento hasta que no están todos los nodos definidos como *workers* en la configuración levantados y comunicando, el proceso se paraliza. Cuando todos los nodos están ya preparados, es cuando se distribuye la carga y comienza el entrenamiento utilizando todos los worker definidos en la lista.

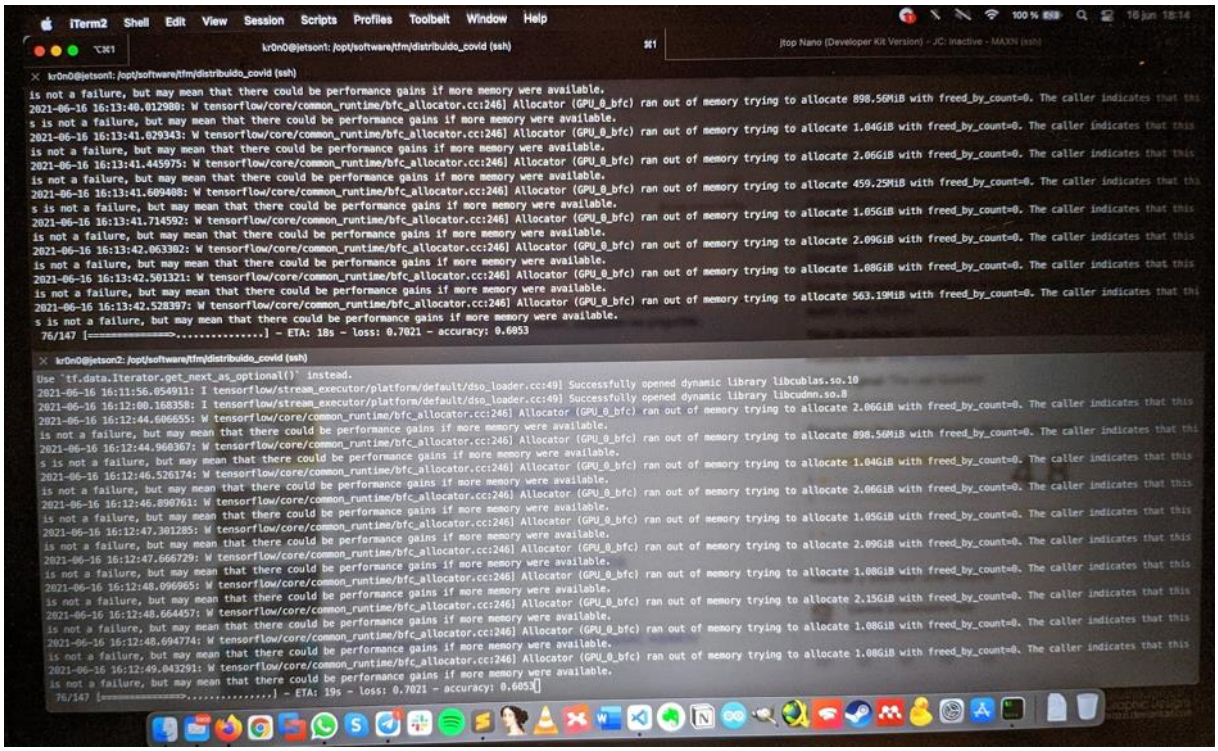


Figura 48. Captura de entrenamiento en clúster con 2 nodos utilizando MultiWorkerMirroredStrategy

En la imagen anterior podemos ver el proceso de entrenamiento con dos terminales en ejecución en cada uno de los nodos con el algoritmo corriendo, donde se ve sincronizado el entrenamiento entre los nodos en el mismo epoch.

Dado que este proceso es realizado mediante aceleración de GPU de cada una de las placas, utilizamos la herramienta `jtop` para poder monitorizar el consumo de GPU de los nodos durante el entrenamiento.

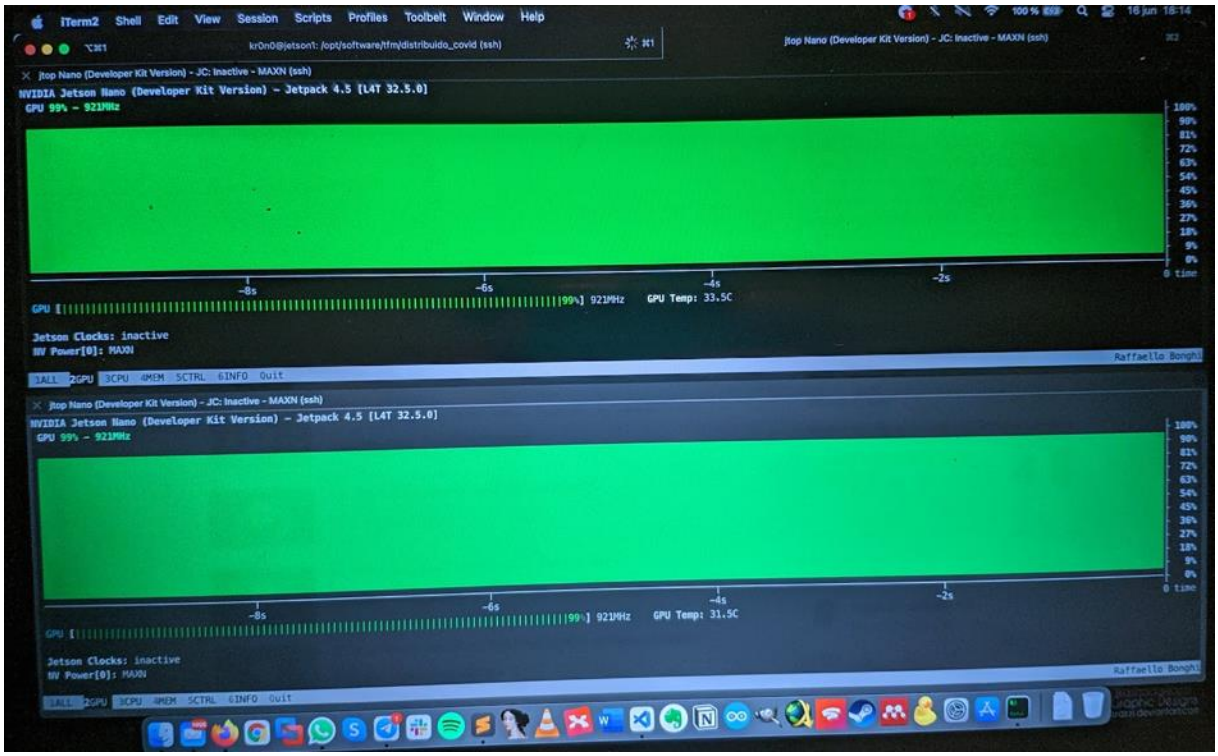


Figura 49. Captura de consumo de GPU en los dos nodos con entrenamiento utilizando MultiWorkerMirroredStrategy

Como en los casos anteriores, tanto los scripts de creación de imágenes de contenedores, ficheros de configuración, scripts y código utilizado están disponibles en los Anexos en las secciones correspondientes.

4.6.5 Entrenamiento en clúster Kubernetes (2 nodos)

Este caso de entrenamiento es el más complejo y el que realmente es el objetivo de todo el proyecto. Para esta tarea tuvimos que plantear inicialmente la manera de definir los trabajos en el clúster, ya que teníamos varias incógnitas que no sabíamos resolver aún:

- No sabíamos que tipo de estructura dentro de las permitidas por Kubernetes sería la más adecuada (Pod, Servicios, Deployments, Jobs…)
- No sabíamos como configurar el almacenamiento del dataset de tal forma que estuviera disponible para todos los nodos del clúster, sin incluirlo en las propias imágenes como en el caso anterior.
- No sabíamos como definir afinidades para evitar la ejecución del algoritmo en el mismo nodo.

- No sabíamos como poder configurar el índice asociado a cada nodo de manera automática sin la variable TF_CONFIG, la cual debíamos omitir y profundizar más en las estructuras de clústering que ofrece TensorFlow.

Invertimos bastante tiempo en esta parte para solventar todos los problemas (incluso realizando modificaciones al código de TensorFlow, o utilizar ciertos módulos de Kubernetes que estaban soportados desde hace una semana atrás.) Realmente este fue el reto más importante del proyecto. Pasamos a explicar cada parte y una breve explicación del contexto y la solución encontrada:

ESTRUCTURA A UTILIZAR

En cuanto al tipo de estructura a utilizar para el clúster, optamos por utilizar un objeto tipo Job. En el contexto de Kubernetes, este tipo de objetos sirve para lanzar uno o varios Pods definidos hasta que un número concreto de ellos termina de manera satisfactoria. Contamos con que para nuestro caso queríamos un número de intentos satisfactorio igual al número de nodos utilizados (2 en nuestro caso) por lo que lo aplicamos al número de ejecuciones en paralelo permitidas (2) y al número de tareas completadas de manera satisfactoria (2) con los campos "`completions`" y "`parallelism`" en la definición del Job. (Para más información ver Apéndice correspondiente).

ALMACENAMIENTO DISTRIBUIDO

En cuanto al almacenamiento del dataset, aquí no podíamos mapear los volúmenes como en el caso de los contenedores, así que profundizamos en el uso de las posibilidades de almacenamiento compartido en Kubernetes.

Este campo en Kubernetes es un mundo en sí mismo, con multitud de soluciones posibles y proyectos de gran envergadura. Inicialmente probamos una solución de código fuente abierto, muy activa en la comunidad, llamada LongHorn (Longhorn Authors & The Linux Foundation, 2021) y (Spiteri, 2021).

Realizamos pruebas con ella, pero el *footprint* y el consumo de recursos eran excesivos para nuestro caso de uso, así que pasamos a una solución más sencilla y clásica en el mundo de sistemas como utilizar un servidor NFS para montar los puntos de acceso en red en el clúster.

Este tipo de soluciones se implementan en Kubernetes utilizando dos objetos llamados `PertistentVolume` y `PersistentVolumeClaim`: El primero (PV) sirve para registrar un volumen persistente de datos con unos parámetros entre los que están el tamaño, modo de acceso, IP y ruta. El segundo (PVC) sirve para reclamar en un modo de acceso concreto un tamaño concreto para otro objeto del clúster.

Montamos un servidor NFS en el nodo inicial (jetson1.local) y exportamos la carpeta con el dataset, para poder crear el PV y el PVC de tipo `storage-nfs` y poder acceder al dataset desde cualquier nodo que forme parte del clúster.

AFINIDADES EN EL CLUSTER

Realizando sucesivas pruebas vimos que en la configuración de un Job era posible que las dos ejecuciones del algoritmo ocurrieran en un mismo nodo. Estuvimos documentándonos sobre como solventar esto, hallando una solución en las afinidades de Kubernetes (o más bien, en las anti-afinidades).

Es posible definir un campo de afinidad en ciertos objetos de Kubernetes donde se puede especificar donde sería deseable que se ejecutaran ciertos Pods basados en algunas características de los nodos. Esta lógica también es aplicable a la manera inversa, lo que se ha venido a llamar anti-afinidad.

Para nuestro caso definimos una anti-afinidad para los Pods que se tuviera en cuenta únicamente a la hora de planificar la ejecución de éstos en el Job, en el que si una etiqueta concreta (“app”, definida a la hora de crear un Pod nuevo en el Job, con valor “`tensorflow`”) existía ya en el equipo donde se fuera a desplegar, fuera rechazada y tuviera que ejecutarse en otro nodo. De esta forma conseguimos que si un Pod ya estuviera en ejecución en un nodo, el siguiente fuera a otro nodo donde no hubiera ninguno en ejecución.

INDICE DE LOS PODS EN EJECUCIÓN

En el caso anterior la gestión del clúster había sido mediante la variable de entorno TF_CONFIG donde definíamos cada nodo, puertos, índice de cada uno, etc.

En este caso tuvimos que profundizar en la gestión de Kubernetes desde TensorFlow, para poder montar la misma infraestructura de gRPC obtenida en el caso anterior pero de forma dinámica según los valores del Clúster.

En la documentación de Kubernetes vimos que la variable TF_CONFIG servía para configurar de manera sencilla una estructura llamada `ClusterSpec`. Vimos además que existía una serie de estructuras entre las que estaba `KubernetesClusterResolver`, el cual si se le pasaba un namespace de Kubernetes y unas etiquetas en concreto, devolvía las direcciones IP de los nodos en una estructura `ClusterSpec`. El problema es según su propia documentación (Mart\'\in~Abadi et al., 2015b), que el tipo de tarea o el identificador de tarea no pueden definirse, por lo que hay que buscar otra manera de poder obtenerlos.

El tipo de tarea era sencillo poder meterla en código como *workers* ya que cada uno de ellos sería un *worker*, pero el índice debería ser obtenido mediante algún parámetro del propio clúster (y la solución oficial de TensorFlow no lo soportaba).

Para solucionar este problema tuvimos que buscar la manera de poder obtener el índice de los Pods ejecutados en el Job y acceder desde el propio código a este valor para configurar el clúster de manera directa.

Dos semanas antes se había liberado una nueva versión de Kubernetes (la v1.21) donde se habían añadido unos nuevos tipos de Jobs llamados "`IndexedJobs`" (Culquicondor, 2021). Este tipo de Jobs permitían crear un índice de trabajos (desde 0 hasta los definidos como por completar -1) en ejecución, lo que permitía también el poder acceder dinámicamente a este campo si se tuviera acceso al clúster de Kubernetes desde los propios Pods de ejecución.

Pudimos ver que la revisión de k3s que estábamos utilizando había sido actualizada el día anterior a la realización de estas pruebas e incluía la revisión de Kubernetes v1.21 con estos cambios, añadiendo un parámetro en la creación del clúster para darle soporte y utilizando la variable JOB_COMPLETION_INDEX para poder sacar el índice necesario para la configuración del clúster.

Una vez conseguido el índice, para conseguir las direcciones IP de los nodos y los puertos a utilizar utilizando la etiqueta "`job-name`" con el valor "`training-cluster`" para cada uno de los nodos y ya conseguimos hacerlo funcionar completamente.

En la siguiente imagen puede verse una captura de pantalla del proceso de entrenamiento en clúster, donde cada nodo está situado en cada eje horizontal, con el entrenamiento a la izquierda y el consumo de GPU a la derecha.

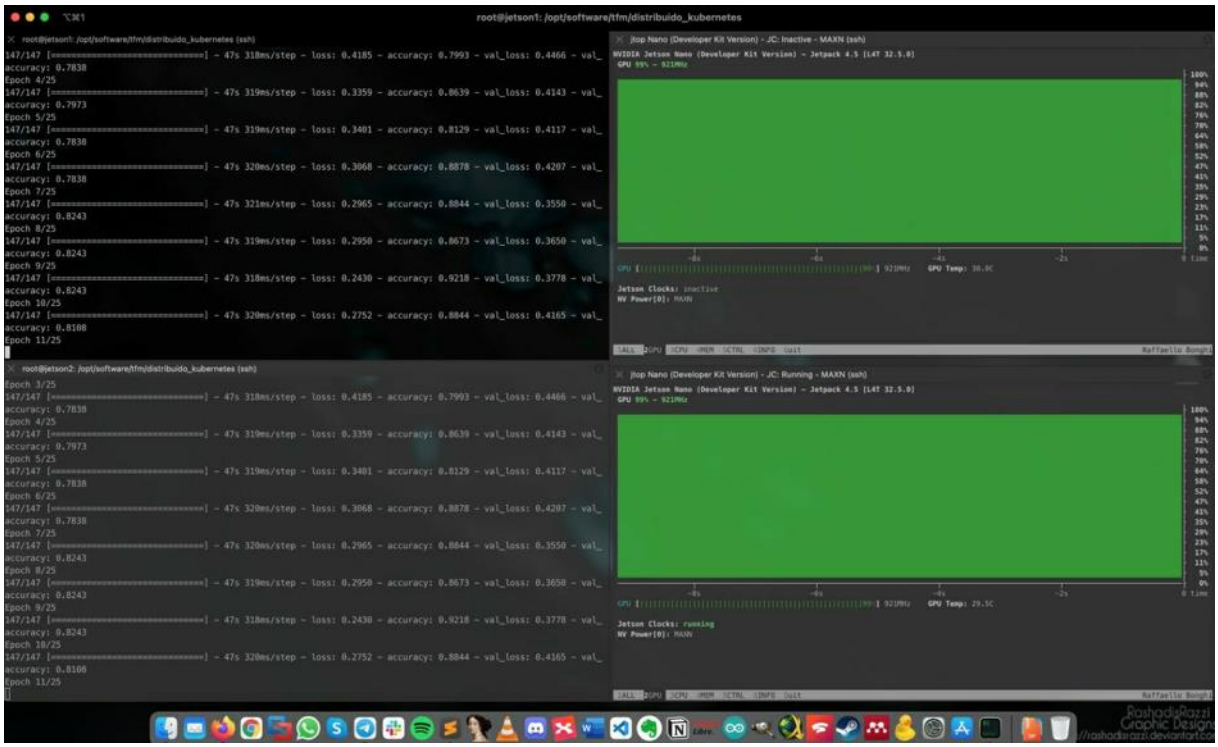


Figura 50. Captura de entrenamiento distribuido en clúster Kubernetes (Entrenamiento y consumo de GPU)

4.7 Entrenamiento y pruebas en nVidia Xavier AGX

Durante las pruebas de entrenamiento del clúster tuvimos acceso a una tarjeta de desarrollo de gama alta nVidia Xavier AGX, utilizada en proyectos de robótica avanzada como conducción autónoma.

Figura 51. Foto de placa nVidia Xavier AGX

Dicha tarjeta comparte sistema operativo con las ya utilizadas nVidia Jetson Nano, por lo que fue sencillo incluirla en las pruebas con la configuración ya creada de Entrenamiento en contenedores con un único nodo y las imágenes Docker correspondientes, como puede comprobarse por la siguiente imagen:

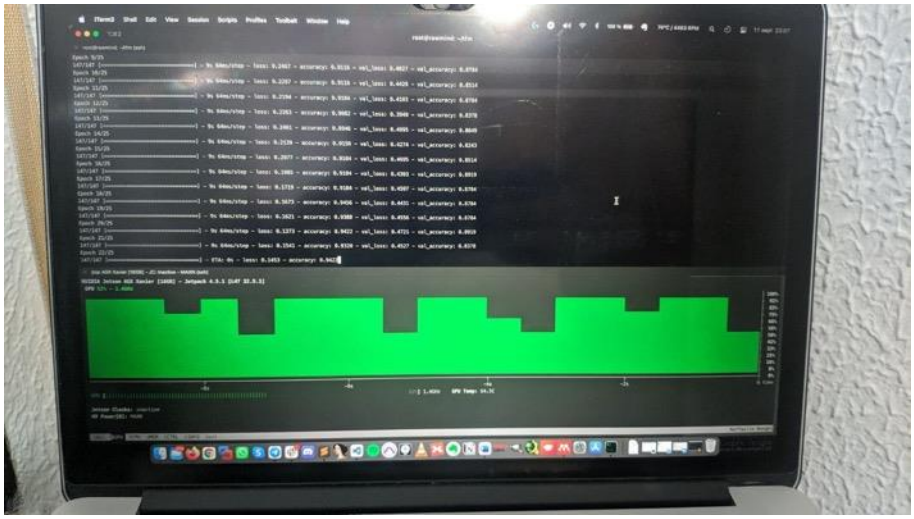


Figura 52. Captura de entrenamiento en nVidia Xavier AGX

4.8 Inferencia en portátil Macbook Pro

Para el caso de la inferencia, utilizaremos el software TensorFlow Serving, ya que es el más utilizado para ofrecer servicios de modelos de TensorFlow en Producción en forma de API.

Este software permite utilizar TensorFlow para exportar un modelo de API que permite enviar un elemento para consultar al modelo en formato JSON y que éste nos devuelva automáticamente el valor predicho en el mismo formato JSON, de manera muy sencilla, realizando una consulta de tipo HTTP a la API del servidor.

Las únicas diferencias apreciables entre los equipos en este caso vienen más dadas como veremos por la arquitectura del procesador y la manera de gestionar las peticiones, por lo que las pruebas y comparativas entre equipos son más limitadas.

4.8.1 Servidor

En caso de arquitecturas Intel de 64 bits (x86_64) como la del equipo de pruebas Macbook Pro, está disponible una imagen Docker con todo el software preparado para utilizar.

La imagen utilizada en este caso es la oficial de TensorFlow (`tensorflow/serving`), y necesita varios parámetros para poder utilizarse:

- Modelo exportado en formato PB

Este modelo es el exportado en el proceso de entrenamiento en la ruta "covid19". Debe tener esta estructura dentro del contenedor, donde 1 es el número de versión del modelo :

```
/models/covid19/1/assets

/models/covid19/1/variables

/models/covid19/1/saved_model.pb

/models/covid19/1/keras_metadata.pb
```

- Nombre del modelo

El nombre del modelo a utilizar. En nuestro caso "covid19", que coincidirá con la ruta dentro de la carpeta /`models`/.

Una vez lanzada la imagen, veremos que TensorFlow Serving levanta un servidor a la escucha en el puerto 8501, esperando peticiones de tipo HTTP.

4.8.2 Cliente

Para realizar las peticiones al servidor, cogeremos dos imágenes del conjunto de pruebas (una positiva y otra negativa) para enviar al servidor y ver la predicción del modelo.

Las imágenes utilizadas para las pruebas son las siguientes :

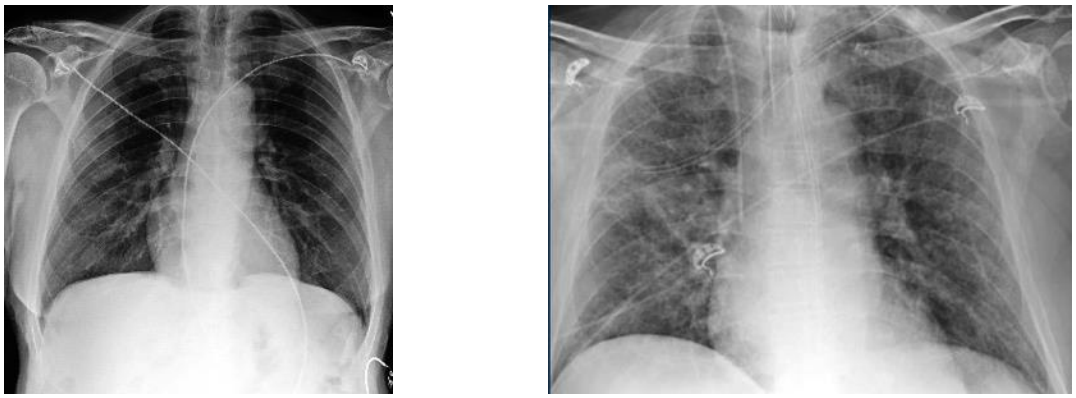


Figura 53. Imágenes positivas y negativas utilizadas para las pruebas

Para esto utilizaremos un algoritmo que redimensione las imágenes al tamaño adecuado, las serialice en formato JSON y las envíe a la API.

El resultado devuelto por la API será el porcentaje de la predicción por cada clase.

Figura 54. Captura de inferencia realizada en equipo MacBook Pro con TensorFlow Serving utilizando dos imágenes

Es importante tener en cuenta que la imagen de TensorFlow Serving utilizada no tiene soporte de GPU, ya que no hay disponibles imágenes que soporten TensorFlow Metal con el plugin de Apple, únicamente los de nVidia, así que la ejecución del modelo se está realizando sin aceleración de GPU, únicamente con CPU.

Como en los casos anteriores, tanto los scripts de creación de imágenes de contenedores, ficheros de configuración, scripts y código utilizado están disponibles en los Anexos en las secciones correspondientes.

4.9 Inferencia en clúster

4.9.1 Inferencia en contenedores (1 nodo)

En el caso anterior en el equipo sobremesa vimos que utilizar la imagen de TensorFlow Serving que provee TensorFlow era la opción más rápida. Lamentablemente, no existe una imagen oficial para la arquitectura de nuestro clúster (ARM64).

Viendo estas limitaciones, el Ingeniero de Software Erik Maciejewski creó un proyecto para recompilar TensorFlow Serving para diferentes arquitectura ARM en su repositorio (Maciejewski, 2021).

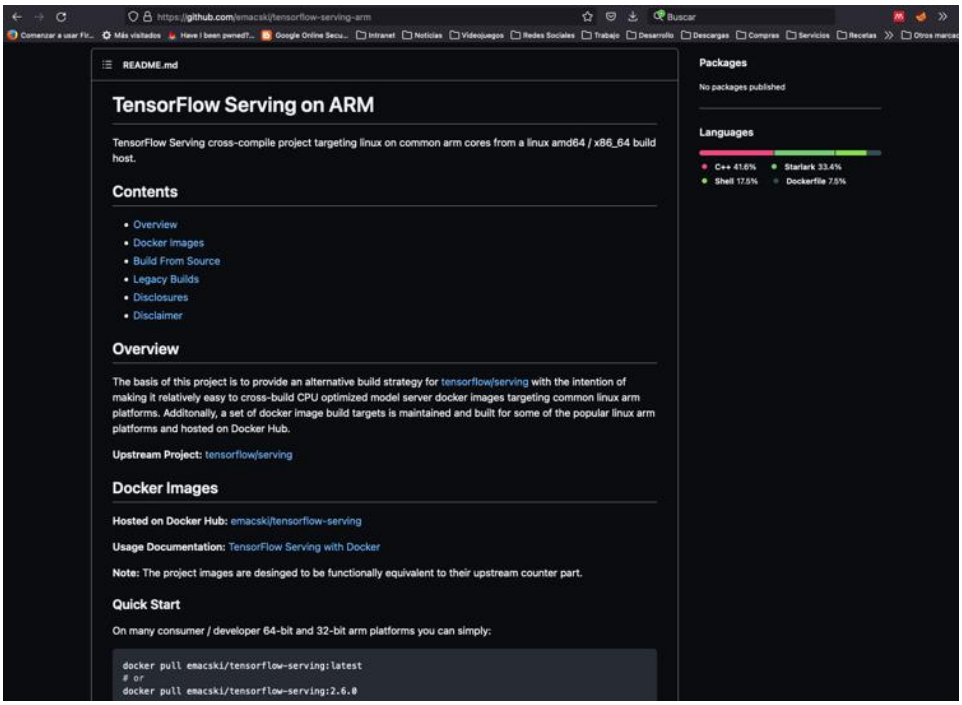


Figura 55. Repositorio GitHub de emacski

Utilizando estas imágenes, hemos podido realizar el trabajo de inferencia en un nodo con arquitectura ARM64 de la misma forma que en el caso anterior.

Para la parte cliente decidimos utilizar una imagen Docker preparada para hacer el procedimiento más sencillo y replicable para todas las arquitecturas.

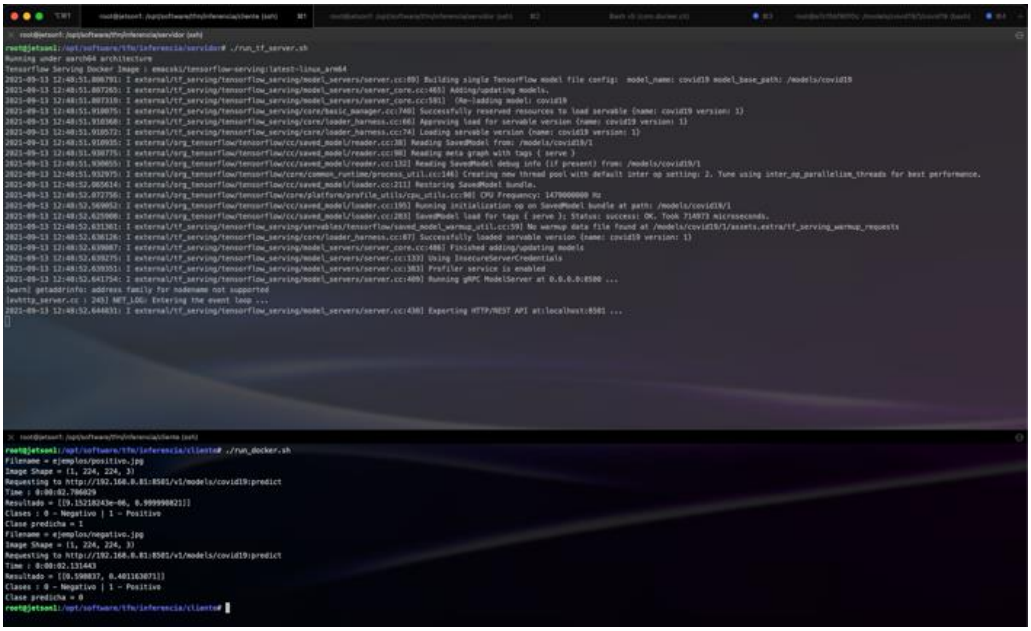


Figura 56. Captura de inferencia en clúster con contenedores y TensorFlow Serving utilizando dos imágenes

Como en los casos anteriores, tanto los scripts de creación de imágenes de contenedores, ficheros de configuración, scripts y código utilizado están disponibles en los Anexos en las secciones correspondientes.

4.9.2 Inferencia en clúster Kubernetes (2 nodos)

Para realizar la inferencia en clúster Kubernetes procedemos a hacer un caso parecido al del entrenamiento en clúster, por lo que necesitaremos :

- Un PV y un PVC para el modelo exportado a compartir entre los nodos.
- Un objeto de tipo Deployment, para desplegar el contenedor como si fuera una aplicación, con dos réplicas deseadas (una por nodo) y la misma anti-afinidad que el caso del entrenamiento, para evitar que haya dos réplicas corriendo en el mismo nodo.
- Exportar el Deployment para tener acceso al exterior mediante un balanceador de carga. Esto se hace creando un servicio de tipo LoadBalancer, de esta manera se puede

acceder desde el exterior al clúster y el balanceador de carga irá mandando las peticiones a los diferentes nodos.

En la siguiente imagen se puede ver el proceso completo de monitorización del clúster y la ejecución de la predicción del modelo desde el equipo MacBook Pro atacando al clúster :

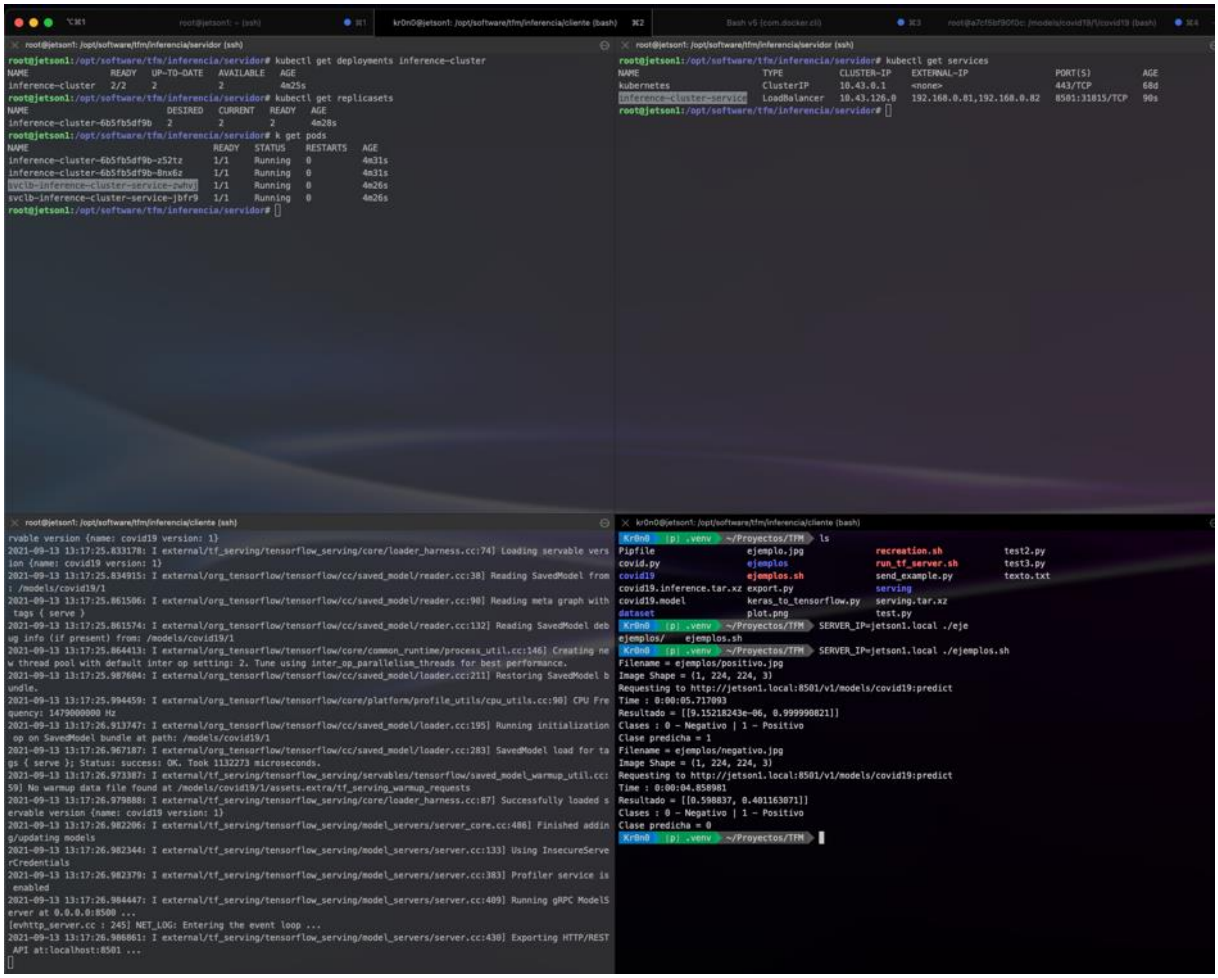


Figura 57. Captura de inferencia con TensorFlow Serving utilizando Kubernetes

- Parte superior izquierda: Registro de logs del Deployment y los ReplicaSets definidos.
- Parte superior derecha: Registro de logs del Service creado y las direcciones IPs externas utilizadas.
- Parte inferior izquierda: Registro de logs de TensorFlow Serving en el Pod correspondiente.
- Parte inferior derecha: Ejecución de los dos ejemplos desde el equipo MacBook Pro.

5. Desarrollo y contribuciones del trabajo

Como objetivo principal se emplea un algoritmo de detección de SARS-CoV-2 basado en la tomografía computarizada de tórax de pacientes con y sin la enfermedad, para demostrar el incremento de rendimiento y la reducción de tiempo en la ejecución de dicho algoritmo, en las plataformas planteadas en este TFM.

Los resultados del entrenamiento y las métricas acordes se han tenido en cuenta mediante los resultados generados por la salida de la ejecución del algoritmo cada Epoch de entrenamiento:

Figura 58. Captura de ejemplo de salida del entrenamiento para Epoch 25

Para cada salida obtenemos la siguiente información :

- Número de Epoch actual / Número de Epoch totales
- Número de Step actual / Número de Step totales
- Tiempo de cálculo de step actual
- Función de pérdida (*loss*) actual en el conjunto de entrenamiento
- Función de exactitud (*accuracy*) actual en el conjunto de entrenamiento
- Función de pérdida (*loss*) actual en el conjunto de validación/test
- Función de exactitud (*accuracy*) actual en el conjunto de validación/test

A la hora de evaluar la red neuronal en cuanto a la clasificación del modelo entrenado utilizamos las siguientes métricas basadas en el reporte de clasificación generado por la biblioteca *scikit-learn* (paquete *metrics*) junto con sus valores promedios ponderados por cada una de las dos clases:

- Precisión (*Precision*): Porcentaje de los clasificados en una clase que lo son realmente.
- Exhaustividad (*Recall*) : Cantidad de los clasificados en una clase que se han identificado correctamente.
- Valor-F (*F1-Score*): Combinación de las dos medidas anteriores en una única métrica
- Soporte (*Support*): Cantidad de ocurrencias para cada clase dadas como ciertas.

Además de estas métricas utilizamos la matriz de confusión, donde se identifica el detalle de los valores verdaderos y falsos clasificados para cada tipo tanto en los reales como en los predichos, según el siguiente esquema:

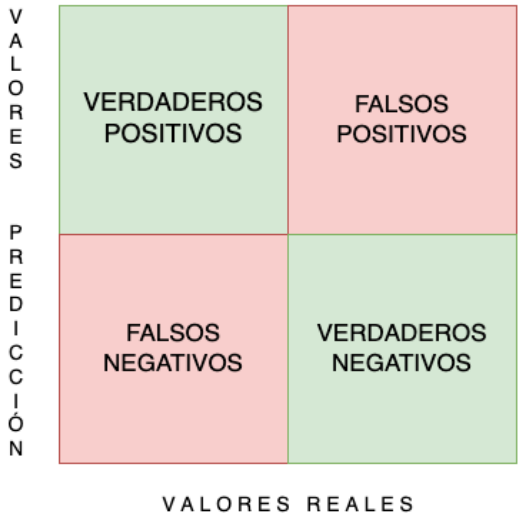


Figura 59. Esquema de matriz de confusión

Como métricas finales tenemos el cálculo de sensibilidad y especificidad, donde la sensibilidad define la probabilidad de clasificar de manera correcta un caso positivo y la especificidad define la probabilidad de clasificar de manera correcta un caso negativo.

5.1 Entrenamiento y pruebas en equipo sobremesa

Teniendo en cuenta las consideraciones previas vistas en el apartado anterior, pasamos a detallar los resultados obtenidos para cada una de las iteraciones realizadas en función de la cantidad de imágenes del dataset.

5.1.1 Dataset 25/75

Dataset formado por 25 imágenes positivas contra 75 imágenes negativas y un Batch Size igual a 1.

```
[INFO] evaluating network...
              precision    recall  f1-score   support

    negativo       0.91      0.86      0.89        37
    positivo       0.87      0.92      0.89        37

    accuracy                           0.89        74
   macro avg       0.89      0.89      0.89        74
weighted avg       0.89      0.89      0.89        74

[[32  5]
 [ 3 34]]
acc: 0.8919
sensitivity: 0.8649
specificity: 0.9189
```

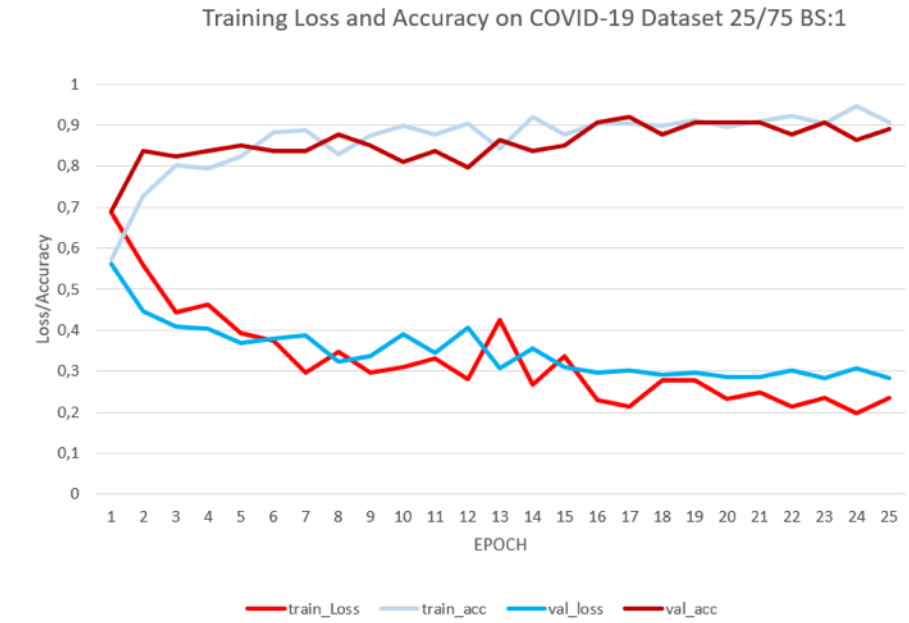


Figura 60. Valores de loss y accuracy para el dataset 25/75 con BatchSize=1

Este caso fue el primero a utilizar en las pruebas, donde el número de imágenes negativas está desbalanceado para darle más importancia a la detección de este tipo de casos.

De cara al entrenamiento del modelo, podemos ver unos valores de exactitud finales en torno al 90% y una función de pérdida en torno al 25%.

En cuanto a la evaluación del modelo, en la matriz de confusión podemos ver 5 casos de falsos positivos y 3 casos de falsos negativos evaluados por el modelo.

Finalmente tenemos unas métricas de sensibilidad en torno al 86% y una especificidad de casi un 92% como resultados finales de la evaluación.

5.1.2 Dataset 100/100

Dataset formado por 100 imágenes positivas contra 100 imágenes negativas y un Batch Size igual a 1.

```
[INFO] evaluating network...
              precision    recall  f1-score   support

    negativo       0.82      0.70      0.76        20
    positivo       0.74      0.85      0.79        20
                                                  [[14  6]
                                                   [ 3 17]]
    accuracy                           0.78        40  acc: 0.7750
   macro avg       0.78      0.77      0.77        40  sensitivity: 0.7000
weighted avg       0.78      0.78      0.77        40  specificity: 0.8500
```

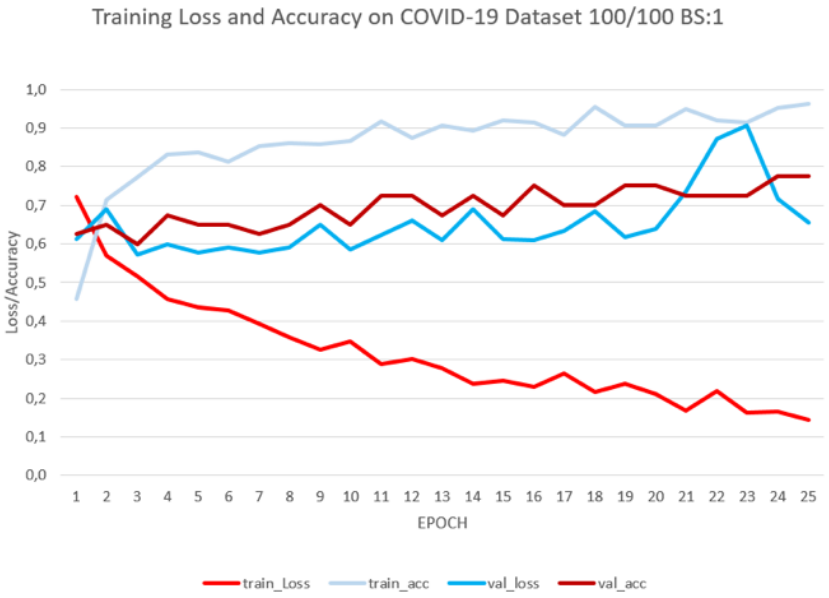


Figura 61. Valores de loss y accuracy para el dataset 100/100 con BatchSize=1

De cara al entrenamiento del modelo, podemos ver unos valores de exactitud finales en torno al 77% y una función de pérdida en torno al 70% para los datos de validación. Este es un valor realmente alto y que demuestra que el dataset no está bien balanceado.

Para la evaluación del modelo, en la matriz de confusión podemos ver 6 casos de falsos positivos y 3 casos de falsos negativos de un total de 40 muestras.

Finalmente tenemos unas métricas de sensibilidad en torno al 70% y una especificidad de un 85% como resultados finales de la evaluación.

Debido a los malos resultados pasamos a incrementar el número de elementos en el dataset a 184.

5.1.3 Dataset 184/184

Dataset formado por 184 imágenes positivas contra 184 imágenes negativas y un Batch Size igual a 1.

```
[INFO] evaluating network...
              precision    recall  f1-score   support

    negativo       0.89      0.92      0.91        37
    positivo       0.92      0.89      0.90        37

    accuracy                           0.91        74
   macro avg       0.91      0.91      0.91        74
weighted avg       0.91      0.91      0.91        74

[[34  3]
 [ 4 33]]
acc: 0.9054
sensitivity: 0.9189
specificity: 0.8919
```

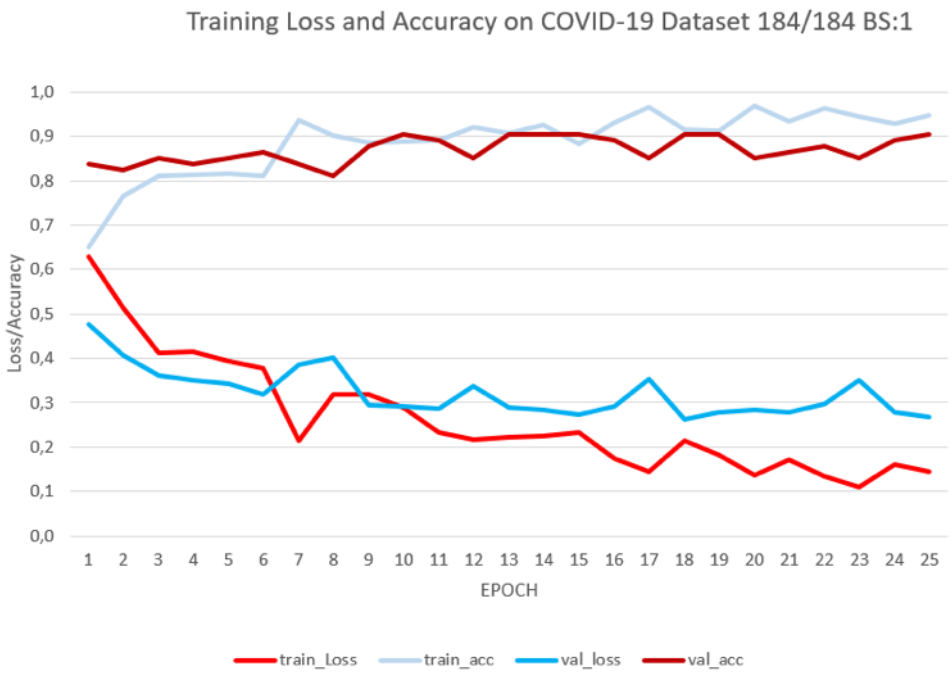


Figura 62. Valores de loss y accuracy para el dataset 184/184 con BatchSize=1

Podemos ver en el entrenamiento del modelo unos resultados mucho mejores que en el caso anterior, con unos valores de exactitud finales en torno al 90% y una función de pérdida en torno al 10% para los datos de validación.

La matriz de confusión resultante de la evaluación del modelo nos da 3 casos de falsos positivos y 4 casos de falsos negativos de un total de 74 muestras.

La métrica de sensibilidad en este caso está en torno al 91% y la especificidad en un 89% como resultados finales de la evaluación, lo cual es una mejora sustancial.

5.1.4 Dataset 184/368

Dataset formado por 184 imágenes positivas contra 368 imágenes negativas y un Batch Size igual a 1.

```
[INFO] evaluating network...
              precision    recall  f1-score   support

    negativo       0.95      0.93      0.94        74
    positivo       0.87      0.89      0.88        37

    accuracy                           0.92       111
   macro avg       0.91      0.91      0.91       111
weighted avg       0.92      0.92      0.92       111

[[69  5]
 [ 4 33]]
acc: 0.9189
sensitivity: 0.9324
specificity: 0.8919
```

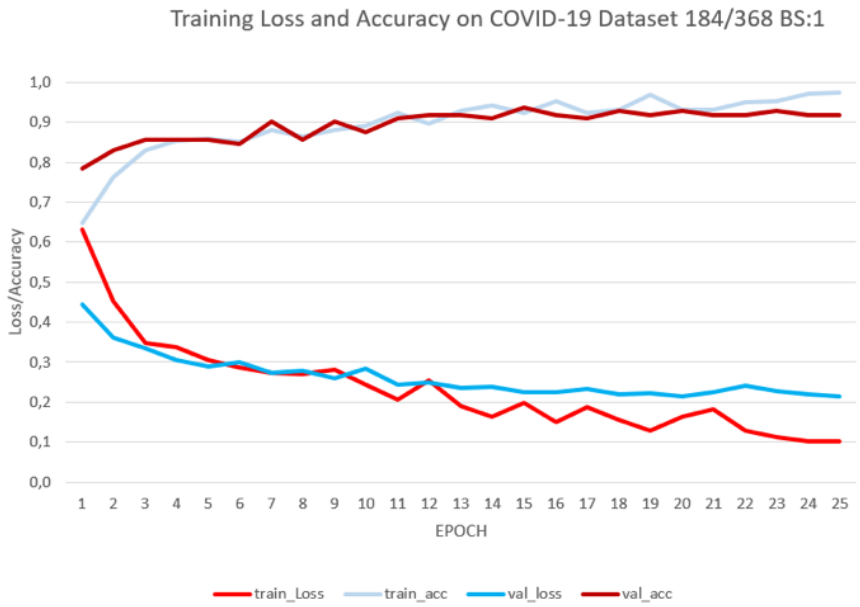


Figura 63. Valores de loss y accuracy para el dataset 184/368 con BatchSize=1

Los resultados del entrenamiento son muy cercanos al caso anterior, donde podemos ver unos valores de exactitud finales en torno al 91% y una función de pérdida en torno al 10% para los datos de validación.

La evaluación del modelo también es similar al caso anterior, con una matriz de confusión con 5 casos de falsos positivos y 4 casos de falsos negativos de un total de 111 muestras.

En cuanto a la métrica de sensibilidad está en torno al 93%, la especificidad está en torno a un 89%, teniendo mejores resultados que el modelo anterior.

5.1.5 Dataset 184/184 con BS=2

Tras realizar las pruebas en las diferentes casuísticas planteadas, observamos la capacidad de aumentar el valor del parámetro Batch Size, de 1 a 2, por lo que de nuevo volvimos a realizar la ejecución del algoritmo en el equipo de sobremesa, modificando dicho parámetro:

```
[INFO] evaluating network...
              precision    recall  f1-score   support

    negativo       0.89      0.89      0.89        37
    positivo       0.89      0.89      0.89        37

    accuracy                           0.89        74
   macro avg       0.89      0.89      0.89        74
weighted avg       0.89      0.89      0.89        74

[[33  4]
 [ 4 33]]
acc: 0.8919
sensitivity: 0.8919
specificity: 0.8919
```

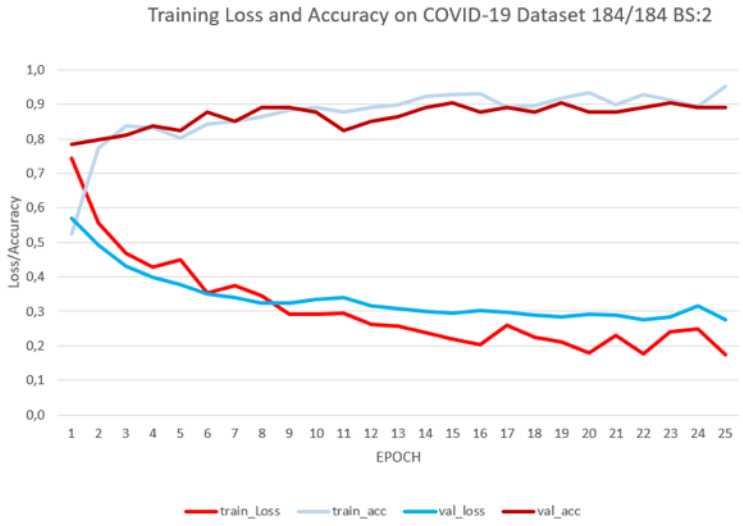


Figura 64. Valores de loss y accuracy para el dataset 184/184 con BatchSize=2

Esta configuración, debido al uso de menos recursos que la anterior y pudiendo utilizar el parámetro Batch Size en modo clúster, fue la seleccionada para utilizar en las comparativas con el resto de equipos como veremos más adelante

Para el entrenamiento del modelo podemos ver unos valores de exactitud finales en torno al 90% y una función de pérdida en torno al 17% para los datos de validación.

A la hora de evaluarlo podemos ver la matriz de confusión con 4 casos de falsos positivos y 4 casos de falsos negativos en un total de 74 muestras.

Finalmente tenemos unas métricas de sensibilidad en torno al 89% y una especificidad de un 89% como resultados finales de la evaluación.

.

5.2 Entrenamiento y pruebas en portátil MacBook Pro

Estos son los resultados obtenidos en la fase de entrenamiento en el equipo portátil MacBook Pro utilizado:

	precision	recall	f1-score	support	
negativo	0.94	0.86	0.90	37	
positivo	0.88	0.95	0.91	37	[[32 5]
					[2 35]]
accuracy			0.91	74	acc: 0.9054
macro avg	0.91	0.91	0.91	74	sensitivity: 0.8649
weighted avg	0.91	0.91	0.91	74	specificity: 0.9459

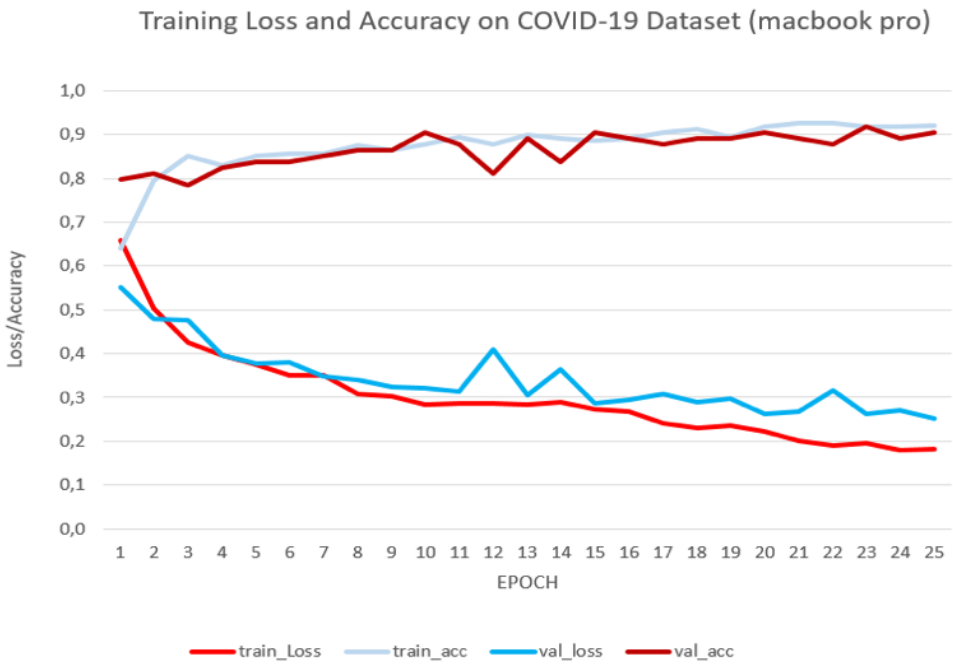


Figura 65. Valores de loss y accuracy para el dataset 184/184 con BatchSize=2 en Macbook Pro

Para este caso de entrenamiento tenemos unos valores de exactitud finales en torno al 90% y una función de pérdida en torno al 19% para los datos de validación.

De cara a la evaluación del modelo, en la matriz de confusión podemos ver 5 casos de falsos positivos y 2 casos de falsos negativos en un total de 74 muestras.

Como últimos resultados tenemos unas métricas de sensibilidad en torno al 86% y una especificidad de un 94% para la evaluación.

Estos resultados son bastante similares a los obtenidos en el equipo sobremesa del caso anterior.

5.3 Entrenamiento en clúster

En estas secciones se irán mostrando los resultados de cada uno de los casos expuestos en el capítulo anterior, desde el uso más básico de una única placa con entorno de desarrollo JupyterLab hasta las dos configuraciones de clúster, la nativa de TensorFlow gRPC con MultiWorkerMirroredStrategy y la final con la infraestructura de Kubernetes completa.

5.3.1 Entrenamiento en contenedores (1 nodo)

Resultados del entrenamiento utilizando una placa desde el entorno de desarrollo JupyterLab preparado en el equipo usando aceleración por GPU.

```
[INFO] evaluating network...
              precision    recall  f1-score   support

    negativo       0.84      0.86      0.85        37
    positivo       0.86      0.84      0.85        37
                                                      [[32  5]
                                                       [ 6 31]]
    accuracy                           0.85        74
   macro avg       0.85      0.85      0.85        74
weighted avg       0.85      0.85      0.85        74
                                                      acc: 0.8514
                                                      sensitivity: 0.8649
                                                      specificity: 0.8378
```

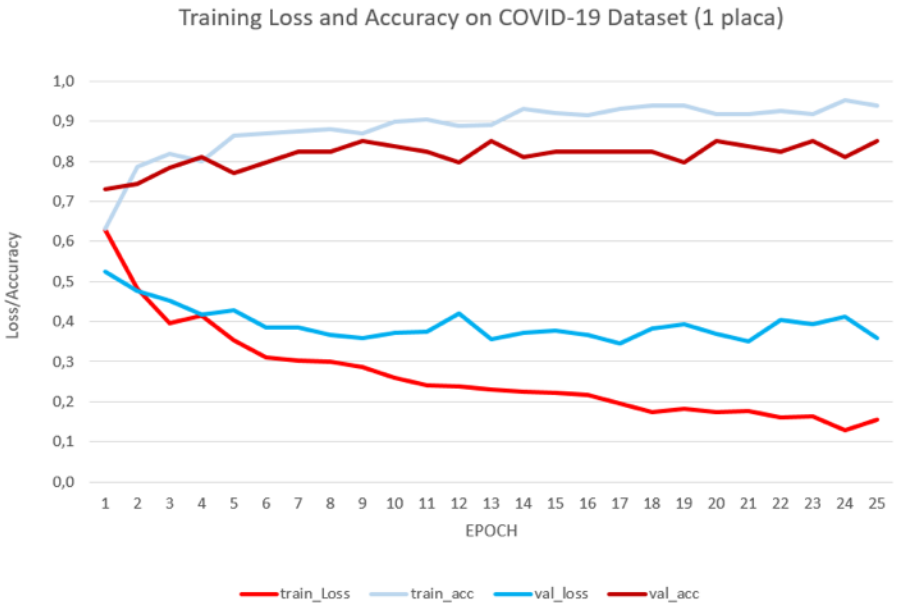


Figura 66. Valores de loss y accuracy para el dataset 184/184 con BatchSize=2 en una única placa nVidia Jetson

Para este caso de ejecución en una placa los resultados son algo peores que en los casos anteriores, sobre todo en cuanto a exactitud y especificidad., donde los valores de exactitud finales están entorno al 85% con una función de pérdida en torno al 15% para los datos de validación.

Observando la evaluación del modelo y su matriz de confusión podemos ver 5 casos de falsos positivos y 6 casos de falsos negativos en un total de 74 muestras.

Finalmente tenemos unas métricas de sensibilidad en torno al 86% y una especificidad de un 83% como resultados finales de la evaluación.

5.3.2 Entrenamiento en clúster con MultiWorkerMirroredStrategy

Resultados del entrenamiento en clúster utilizando la estrategia MultiWorkerMirroredStrategy de TensorFlow y las comunicaciones gRPC:

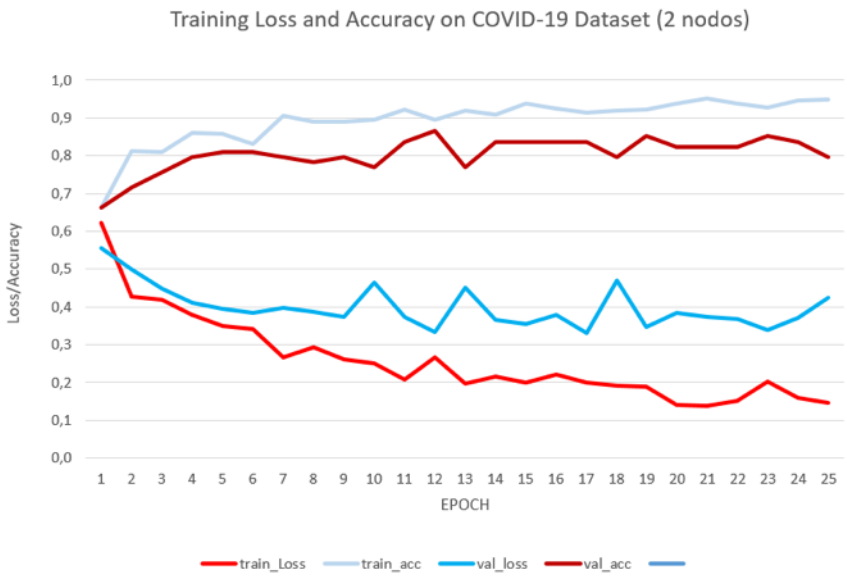


Figura 67. Valores de loss y accuracy para el dataset 184/184 con BatchSize=2 en clúster con MultiWorkerMirroredStrategy

En este caso para el modo de ejecución en paralelo no está soportado el realizar una evaluación en paralelo, por lo que no disponemos de métricas en este caso para evaluarlo con respecto al resto de equipos.

Para los valores de entrenamiento si podemos ver unos valores de exactitud finales en torno al 80% y una función de pérdida en torno al 15% para los datos de validación.

5.3.3 Entrenamiento en clúster Kubernetes

Los resultados de entrenamiento utilizando Kubernetes para dos placas con un Job de Kubernetes son los siguientes:

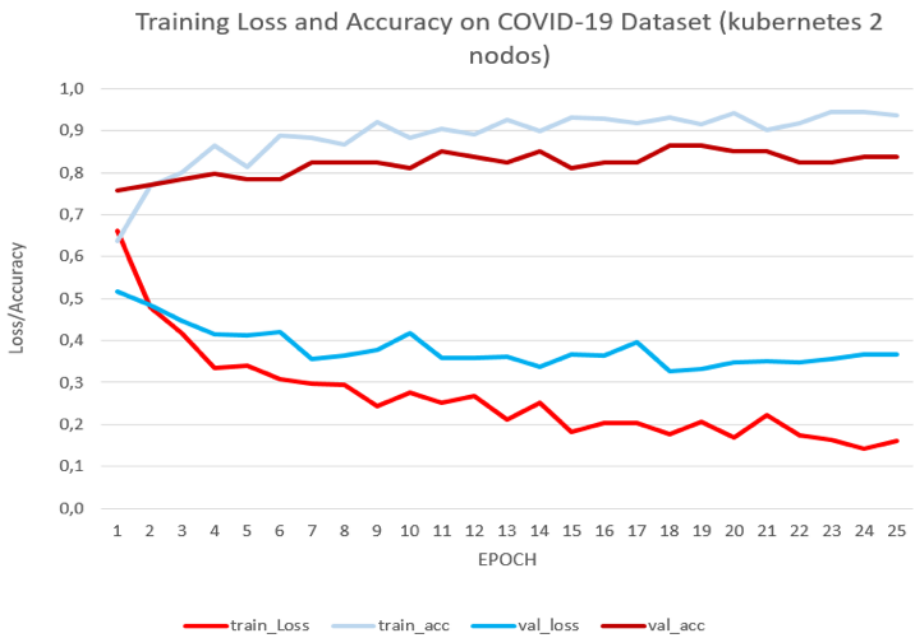


Figura 68. Valores de loss y accuracy para el dataset 184/184 con BatchSize=2 en clúster Kubernetes

También como el caso anterior en el modo de ejecución en paralelo no está soportado el realizar una evaluación en paralelo del modelo, por lo que no disponemos de métricas en este caso para evaluarlo con respecto al resto de casos.

De cara a la parte de entrenamiento del modelo en paralelo, podemos ver unos valores de exactitud finales en torno al 85% y una función de pérdida en torno al 15% para los datos de validación presentados.

5.4 Entrenamiento y pruebas en nVidia Xavier AGX

Los resultados siguientes son los obtenidos en el entrenamiento realizado en la placa nVidia Xavier AGX con una configuración de contenedores en un nodo :

	precision	recall	f1-score	support
negativo	0.83	0.92	0.87	37
positivo	0.91	0.81	0.86	37
accuracy			0.86	74
macro avg	0.87	0.86	0.86	74
weighted avg	0.87	0.86	0.86	74

```
[[34  3]
 [ 7 30]]
acc: 0.8649
sensitivity: 0.9189
specificity: 0.8108
```

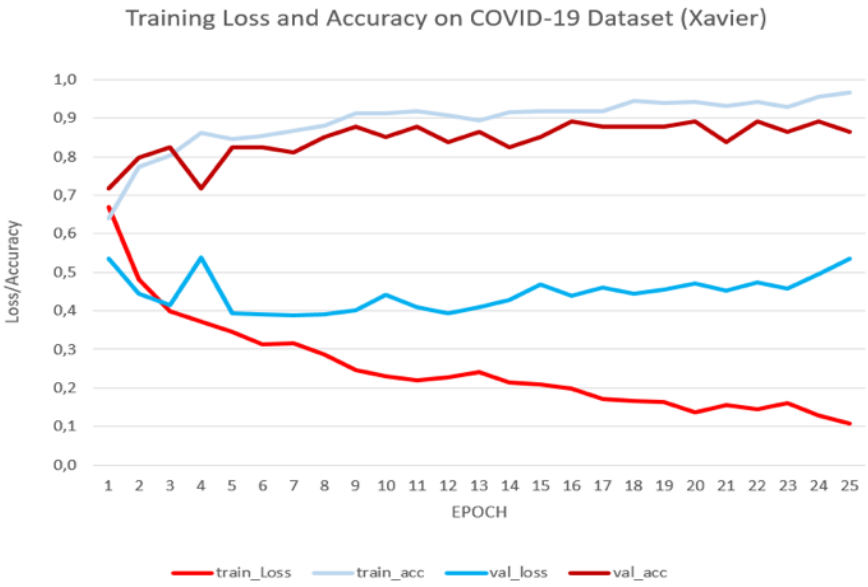


Figura 69. Valores de loss y accuracy para el dataset 184/184 con BatchSize=2 en nVidia Xavier AGX

Este caso es el hardware más potente de los que utilizamos para realizar las pruebas. Al realizar el entrenamiento del modelo podemos ver unos valores de exactitud finales en torno al 86% y una función de pérdida en torno al 10% para los datos de validación.

En cuanto a la evaluación del modelo podemos ver en la matriz de confusión 3 casos de falsos positivos y 7 casos de falsos negativos de 74 muestras disponibles en total.

A la hora de valorar las métricas restantes, podemos ver una sensibilidad en torno al 91% y una especificidad de un 81% como resultados finales de la evaluación.

5.5 Inferencia en portátil MacBook Pro

Para valorar los resultados de la inferencia medimos el tiempo entre la solicitud desde el lado cliente (*request*) hasta que recibimos la respuesta del lado servidor (*response*).

Los resultados en este caso son efectuados en el equipo MacBook Pro utilizando la imagen de contenedor existente de TensorFlow, como se ha desarrollado en el apartado 4.7 del presente documento.

Tabla 6. Resultados de Inferencia – Macbook Pro

	EJEMPLO POSITIVO		**EJEMPLO NEGATIVO**	
TIEMPO	0:00:00.696479		0:00:00.617084	
ESTIMACIÓN DE CLASE	9.15218243e-06	0.999990821	0.598836124	0.401163876
CLASE PREDICHA	1 (OK - POSITIVO)		0 (OK - NEGATIVO)	

Tanto los valores de estimación como los tiempos son muy buenos ya que no tarda ni un segundo en realizar la petición y devolver los resultados con la predicción estimada.

5.6 Inferencia en clúster

Los resultados en este caso son los realizados utilizando una tarjeta nVidia Jetson Nano y el clúster completo con soporte de Kubernetes, como se ha desarrollado en apartado 4.8 del presente documento.

5.6.1 - Inferencia en contenedores (1 nodo)

Inferencia realizada en una única tarjeta nVidia Jetson Nano con la imagen de contenedor.

Tabla 7. Resultados de Inferencia – 1x Jetson Nano

	EJEMPLO POSITIVO		**EJEMPLO NEGATIVO**	
TIEMPO	0:00:02.786029		0:00:02.131443	
ESTIMACIÓN DE CLASE	9.15218243e-06	0.999990821	0.598837	0.401163071
CLASE PREDICHA	1 (OK - POSITIVO)		0 (OK - NEGATIVO)	

5.6.2 - Inferencia en clúster Kubernetes (2 nodos)

Realizado con la infraestructura de Kubernetes en los dos nodos: Deployment, Service y dos contenedores.

Tabla 8. Resultados de Inferencia – Clúster Kubernetes con LoadBalancer

	EJEMPLO POSITIVO		**EJEMPLO NEGATIVO**	
TIEMPO	0:00:05.717093		0:00:04.858981	
ESTIMACIÓN DE CLASE	9.15218243e-06	0.999990821	0.598837	0.401163071
CLASE PREDICHA	1 (OK - POSITIVO)		0 (OK - NEGATIVO)	

Los resultados de tiempo son mucho mayores que en los casos anteriores. Los motivos los desarrollaremos en el siguiente capítulo.

6. Discusión

En los siguientes puntos vamos a analizar los resultados obtenidos en los procesos anteriores y realizar comparativas para el número de elementos de dataset utilizados, los resultados del entrenamiento en los diferentes equipos y las métricas de tiempo en cuanto a entrenamiento e inferencia.

6.1 Número de elementos en el dataset

La comparativa siguiente se basa en diferentes resultados de entrenamiento con diferentes conjuntos de datos, definidos por el número de valores positivos, número de valores negativos y el Batch Size correspondiente. Todas estas comparativas han sido realizadas en el equipo sobremesa inicial AMD Ryzen 1800X hasta dar con la mejor combinación para el hardware elegido.

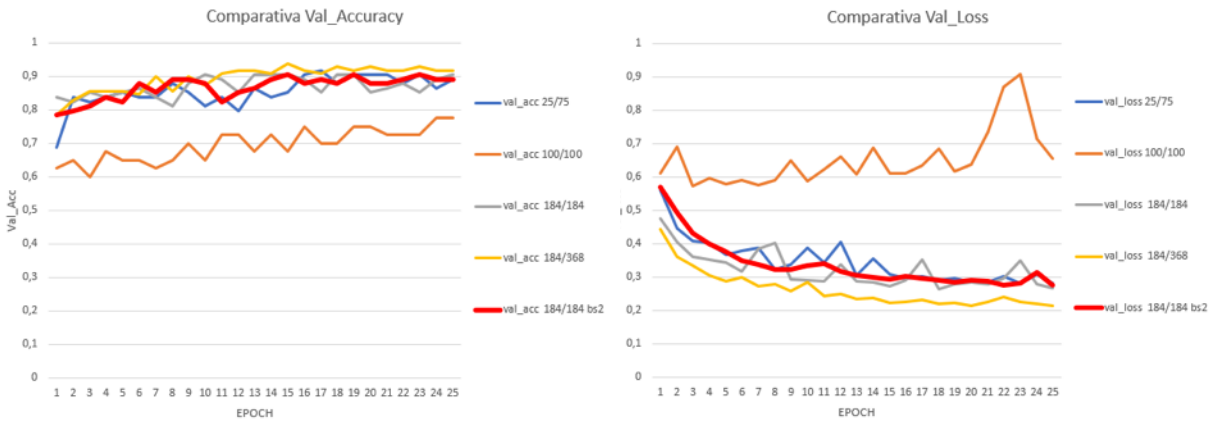


Figura 70. Comparativa de exactitud y pérdida entre los diferentes datasets

Podemos ver que los niveles de exactitud finales son muy similares en todos los casos salvo en el dataset de 100/100. Entre los 184/184, 184/384 y 184/184 con BS=2 los resultados son muy parecidos, mejorando levemente en cuanto al valor de la función pérdida en el caso del 184/384, pero como hemos comentado anteriormente elegimos el conjunto 184/184 con BS=2 debido a una menor carga de recursos para los equipos embebidos.

6.2 Rendimiento de los modelos

En cuanto al resultado de los diferentes entrenamientos en los equipos realizados con el dataset 184/184 BS=2, tenemos los siguientes resultados:

Tabla 9. Resultados promedio de las métricas de rendimiento de los diferentes modelos

EQUIPO	Accuracy	Loss	Sensibilidad	Especificidad
AMD Ryzen 1800X 16GB RAM	90	17	89%	89%
Macbook Pro 16GB RAM + nVidia 750M	90	19	86%	94%
nVidia Jetson Nano	85	15	86%	83%
2x nVidia Jetson Nano (gRPC)	80	15	N/A	N/A
2x nVidia Jetson Nano (Kubernetes)	85	15	N/A	N/A
nVidia Jetson Xavier AGX	86	10	91%	81%

Tabla 10. Leyenda de equipos y etiqueta para las gráficas

EQUIPO	ETIQUETA
AMD Ryzen 1800X con Dataset 184/184 y Batch Size 1	184/184
AMD Ryzen 1800X con Dataset 184/184 y Batch Size 2	184/184 bs2
MacBook Pro 2013	Macbook pro
nVidia Jetson Nano 1 nodo	1 nodo
2x nVidia Jetson Nano (clúster gRPC)	Ini Cont (2 nodos)
2x nVidia Jetson Nano (clúster Kubernetes)	Kub (2 nodos)
nVidia Xavier AGX	Xavier

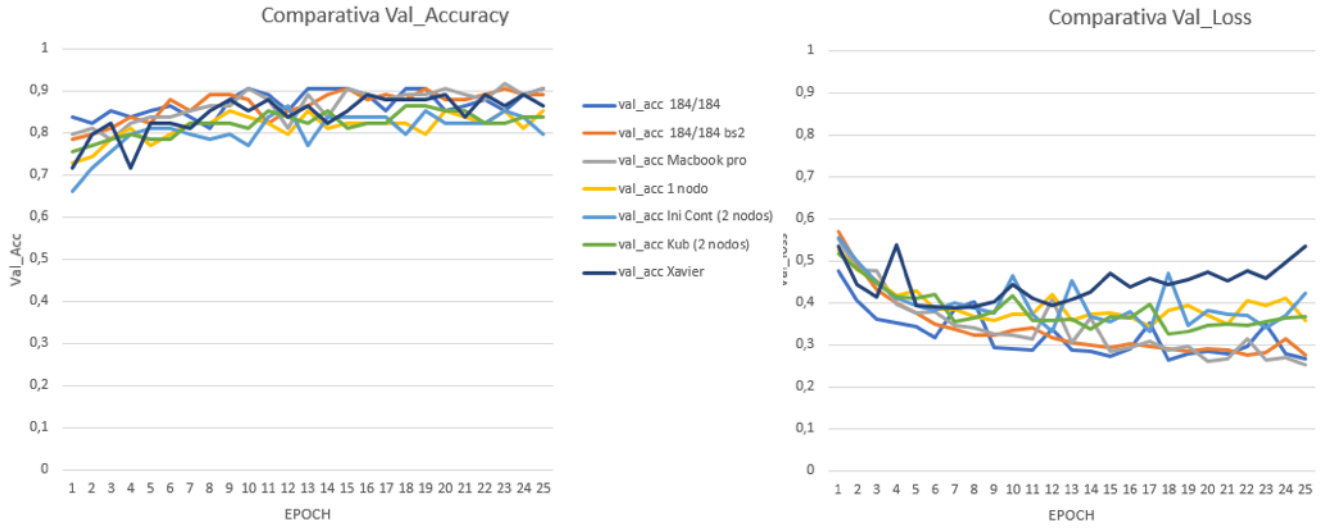


Figura 71. Comparativa de exactitud y pérdida entre los diferentes equipos

En cuanto al rendimiento del modelo no podemos tener una conclusión clara en cuanto a que haya un modelo mejor que otro, aunque podemos considerar que todos los resultados están dentro de unos valores aceptables. Este resultado puede ser debido sobre todo a las diferentes versiones de TensorFlow utilizadas y las optimizaciones de las diferentes arquitecturas a la hora de realizar los entrenamientos.

Aunque está fuera del alcance de este TFM, con el objetivo de estudiar las posibles diferencias entre los modelos y poder sacar una conclusión más acertada se plantea un punto nuevo como futura línea de investigación contemplado en el Capítulo 7.

6.3 Tiempo de entrenamiento

En cuanto al tiempo de entrenamiento, tenemos los siguientes resultados en la comparativa:

Tabla 11. Resultados promedio de entrenamiento – Segundos por Epoch

EQUIPO	GPU	RESULTADOS
Macbook Pro 16GB RAM + nVidia 750M	Si	128 sec/epoch
nVidia Jetson Nano	Si	88 sec/epoch
2x nVidia Jetson Nano (gRPC)	Si	47 sec/epoch
2x nVidia Jetson Nano (Kubernetes)	Si	47 sec/epoch
AMD Ryzen 1800X 16GB RAM	No	40 sec/epoch
nVidia Jetson Xavier AGX	Si	9 sec/epoch

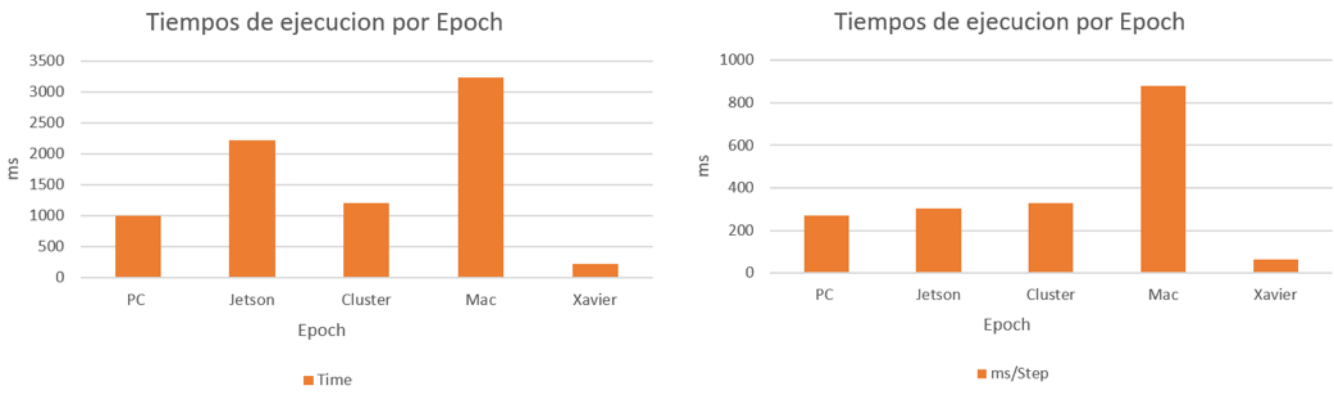


Figura 72. Tiempos de ejecución por Epoch (ms/epoch y ms/step)

En cuanto al rendimiento en el entrenamiento, vemos que los resultados son relevantes en cuanto a la mejora de utilización de dos placas Jetson con respecto a una, ya que indican una mejora en cuanto a los tiempos muy cercanos al rendimiento teórico del clúster, siendo casi la mitad de tiempo (88 segundos contra 47) de entrenamiento por Epoch. Si a esto le sumamos que en el caso del clúster está corriendo además toda la infraestructura de Kubernetes en los dos nodos más la latencia de comunicaciones, no penaliza apenas el entrenamiento con respecto al caso de clúster sin Kubernetes, teniendo los mismos resultados de tiempo por Epoch.

Es interesante ver que la arquitectura de la GPU prima mucho en cuanto al rendimiento en este tipo de tareas, teniendo un equipo con una tarjeta gráfica de 2013 (MacBook Pro con GeForce GT 750M, arquitectura Kepler) que tiene peor rendimiento que una tarjeta Jetson Nano (arquitectura Maxwell). También se ve el salto evidente con la nueva arquitectura Volta de la Xavier AGX, que tiene el mejor rendimiento de todas.

Como último caso a tener en cuenta, en el caso de un procesador AMD Ryzen 1800X es capaz de equipararse por puro cálculo de CPU a una tarjeta de las prestaciones del clúster.

6.4 Tiempo de inferencia

Tabla 12. Resultados promedio de inferencia – Segundos por request

EQUIPO	ARQUITECTURA	PROMEDIO REQUEST
Macbook Pro 16GB RAM + nVidia 750M	Intel 64 Bits	0,656781 sec
nVidia Jetson Nano	ARM 64 Bits	2,458736 sec
2x nVidia Jetson Nano (Clúster)	ARM 64 Bits	5,288037 sec

En cuanto a los resultados de inferencia, lo importante en este caso es medir el tiempo promedio desde que se hace la solicitud a la API, enviando la imagen para evaluar, hasta que ésta devuelve el resultado con los porcentajes de estimación por clase.

Al ser un proceso puramente de CPU en esta infraestructura (no existen versiones de TensorFlow Serving para GPU ni en Mac OS X ni en arquitectura ARM64), los resultados son esperables en cuanto a la comparativa entre el MacBook Pro y una única tarjeta nVidia Jetson (0,6 vs 2,4 segundos), ya que la CPU del MacBook tiene mucho más rendimiento que la de la tarjeta.

En cuanto a la comparativa de una tarjeta y el clúster, el incremento de tiempo es notable en cuanto a la respuesta (más del doble, 5 segundos contra 2) pero hay que tener en cuenta precisamente que la infraestructura en Clúster tiene un balanceador de carga entre las diferentes tarjetas, por lo que la arquitectura es más compleja en cuanto a los pasos que tiene que realizar en las comunicaciones (el stack de Kubernetes) pero a su vez es más escalable ante avalanchas de peticiones. Este tipo de pruebas serán consideradas para líneas futuras de investigación y reflejadas en el siguiente apartado.

7. Conclusiones y trabajo futuro

La principal premisa de esta prueba de concepto era diseñar una solución embebida de bajo coste que permitiera realizar tareas de inteligencia artificial con aceleración gráfica, de forma escalable y sin dependencias externas de conectividad.

Esta solución debía tener mejor rendimiento que las soluciones embebidas de bajo coste existentes (como placas SBC con GPUs) y, a su vez, menor coste que las soluciones más complejas (como estaciones de trabajo completas o servidores dedicados). Además de esto, debía tratarse de una solución 100% escalable según el tipo de proyecto, añadiendo nuevas placas dependiendo de las necesidades y gestionando el clúster de manera sencilla para, así, aumentar el rendimiento sin incrementar excesivamente el coste ni la complejidad de su gestión.

Para valorar esta solución, elegimos como caso práctico el diagnóstico de COVID-19 con imágenes de Rayos X mediante redes neuronales convolucionales de forma que, mientras desarrollábamos la prueba de concepto, pudiéramos generar métricas y resultados para comparar con otros equipos. Este caso de uso cumple, además, con un enfoque necesario de computación en Edge ya que los datos utilizados son de carácter sensible (imágenes de pacientes) y no utiliza otras conexiones externas, excepto la posibilidad de conectarse directamente a la máquina de Rayos X que genera las imágenes.

Finalmente, realizamos el desarrollo del experimento y comprobamos que cumplimos los objetivos propuestos, pudiendo realizar, bajo la solución propuesta, las tareas tanto de entrenamiento como de inferencia, aunque con ciertas deficiencias, sobre todo relacionadas con la arquitectura ARM64 utilizada: falta de soporte de algunos componentes software o la capacidad hardware disponible en los equipos.

Como resultados a recalcar en las comparativas vemos una mejora en los tiempos para entrenamiento de casi el triple respecto al equipo MacBook Pro, el doble respecto a una placa única, y similares respecto al equipo sobremesa con procesador Ryzen, aunque los tiempos están aún muy por encima de los conseguidos en la placa Xavier AGX. En cuanto a la inferencia, los tiempos medidos son de casi el doble a la hora de realizar un diagnóstico de imagen comparado con otros equipos, aunque es esperable debido a la naturaleza del clúster a la hora de balancear las cargas entre los nodos.

De cara a contemplar posibles mejoras o futuras líneas de investigación, podemos centrarlas, por un lado, respecto al caso de uso de diagnóstico de COVID-19 y, por otro, respecto a la infraestructura tecnológica:

Para la mejora del caso de uso podrían utilizarse otros datasets más grandes o de mayor calidad para mejorar los resultados (por ejemplo con TACs de ultra alta resolución). También sería interesante realizar otras pruebas con otros modelos de redes convolucionales más avanzados que vayan surgiendo para poder comparar resultados, siempre que el hardware tenga capacidad para ello.

Para la mejora de la infraestructura tecnológica se podría intentar adaptar diferentes herramientas que no están disponibles actualmente (o tienen funcionalidad limitada) en la arquitectura ARM64 con soporte de GPU como KubeFlow, TensorFlow Serving o LongHorn y comparar su rendimiento con las utilizadas en este trabajo. También sería interesante un estudio sobre peticiones masivas en el clúster Kubernetes para valorar tiempos de respuesta y estabilidad con el balanceador de carga respecto a los otros casos estudiados.

Bibliografía

Alliance, C. europea A. (n.d.). *Comisión europea AI Alliance*. https://ec.europa.eu/futurium/en/ai-robotics-vs-covid19/join-ai-robotics-vs-covid-19-initiative-european-ai-alliance

Amanat, F., & Krammer, F. (2020). SARS-CoV-2 Vaccines: Status Report. *Immunity*, *52*(4), 583–589. https://doi.org/10.1016/j.immuni.2020.03.007

Arabi, Y. M., Alothman, A., Balkhy, H. H., Al-Dawood, A., AlJohani, S., Al Harbi, S., Kojan, S., Al Jeraisy, M., Deeb, A. M., Assiri, A. M., Al-Hameed, F., AlSaedi, A., Mandourah, Y., Almekhlafi, G. A., Sherbeeni, N. M., Elzein, F. E., Memon, J., Taha, Y., Almotairi, A., … Taif, J. S. (2018). Treatment of Middle East Respiratory Syndrome with a combination of lopinavir-ritonavir and interferon-β1b (MIRACLE trial): Study protocol for a randomized controlled trial. *Trials*, *19*(1), 1–13. https://doi.org/10.1186/s13063-017-2427-0

Arabi, Y. M., Murthy, S., & Webb, S. (2020). COVID-19: a novel coronavirus and a novel challenge for critical care. *Intensive Care Medicine*, *46*(5), 833–836. https://doi.org/10.1007/S00134-020-05955-1

Authors, T. K. (2021). *Kubernetes*. The Linux Foundation. https://kubernetes.io/

Böhm, S., & Wirtz, G. (2020). *Profiling Lightweight Container Platforms: MicroK8s and K3s in Comparison to Kubernetes*. https://kubernetes.io/

Bonawitz, K., Eichner, H., Grieskamp, W., Huba, D., Ingerman, A., Ivanov, V., Kiddon, C., Konečný, J., Mazzocchi, S., McMahan, H. B., Van Overveldt, T., Petrou, D., Ramage, D., & Roselander, J. (2019). *Towards Federated Learning at Scale: System Design*. http://arxiv.org/abs/1902.01046

Borghesi, A., & Maroldi, R. (2020). COVID-19 outbreak in Italy: experimental chest X-ray scoring system for quantifying and monitoring disease progression. *Radiologia Medica*, *125*(5), 509–513. https://doi.org/10.1007/s11547-020-01200-3

Cheah, K. H., Nisar, H., Yap, V. V., Lee, C. Y., & Sinha, G. R. (2021). Optimizing residual networks and vgg for classification of eeg signals: Identifying ideal channels for emotion recognition. *Journal of Healthcare Engineering*, *2021*. https://doi.org/10.1155/2021/5599615

Chollet, F. (2018). 16-Keras. *Deep Learning with Python*, 97–111. http://link.springer.com/10.1007/978-1-4842-2766-4_7

Cohen, J. P., Morrison, P., Dao, L., Roth, K., Duong, T. Q., & Ghassemi, M. (2020). *COVID-19 Image Data Collection: Prospective Predictions Are the Future*. http://arxiv.org/abs/2006.11988

Creech, C. B., Walker, S. C., & Samuels, R. J. (2021). SARS-CoV-2 Vaccines. *JAMA - Journal of the American Medical Association*, *325*(13), 1318–1320. https://doi.org/10.1001/jama.2021.3199

Culquicondor, A. (2021). *Introducing Indexed Jobs*. https://kubernetes.io/blog/2021/04/19/introducing-indexed-jobs/

De Donno, M., Tange, K., & Dragoni, N. (2019). Foundations and Evolution of Modern Computing Paradigms: Cloud, IoT, Edge, and Fog. *IEEE Access*, *7*, 150936–150948. https://doi.org/10.1109/ACCESS.2019.2947652

Deng, J., Dong, W., Socher, R., Li, L., Li, K., & Fei-Fei, L. (2009). ImageNet: A large-scale hierarchical image database. *2009 IEEE Conference on Computer Vision and Pattern Recognition*, 248–255. https://doi.org/10.1109/CVPR.2009.5206848

Edge AI: Convergence of Edge Computing and Artificial Intelligence - Xiaofei Wang, Yiwen Han, Victor C. M. Leung, Dusit Niyato, Xueqiang Yan, Xu Chen - Google Libros. (n.d.). Retrieved June 7, 2021, from https://books.google.es/books?hl=es&lr=&id=MGn6DwAAQBAJ&oi=fnd&pg=PR7&dq=edge+ai+convergence+of+edge&ots=TnIA5Hzo5b&sig=tYsk4wSn8z-KN6F80WVH2gdQ7p0#v=onepage&q=edge ai convergence of edge&f=false

El-Saadawy, H., Tantawi, M., Shedeed, H. A., & Tolba, M. F. (2021). A Hybrid Two-Stage GNG–Modified VGG Method for Bone X-Rays Classification and Abnormality Detection. *IEEE Access*, *9*, 76649–76661. https://doi.org/10.1109/ACCESS.2021.3081915

Google AI Blog: Federated Learning: Collaborative Machine Learning without Centralized Training Data. (n.d.). Retrieved June 8, 2021, from https://ai.googleblog.com/2017/04/federated-learning-collaborative.html

Google Cloud Platform Blog: Containers, VMs, Kubernetes and VMware. (n.d.). Retrieved June 8, 2021, from https://cloudplatform.googleblog.com/2014/08/containers-vms-kubernetes-and-vmware.html

Hannes Hapke, H., & Nelson, C. (2020). *Building Machine Learning Pipelines* (O'Reilly (Ed.)). O'Reilly. https://www.oreilly.com/library/view/building-machine-learning/9781492053187/

He, K., Zhang, X., Ren, S., & Sun, J. (n.d.). *Deep Residual Learning for Image Recognition*. http://image-net.org/challenges/LSVRC/2015/

Hong, X., Xiong, J., Feng, Z., & Shi, Y. (2020). Extracorporeal membrane oxygenation (ECMO): does it have a role in the treatment of severe COVID-19? *International Journal of Infectious Diseases*, *94*, 78–80. https://doi.org/10.1016/j.ijid.2020.03.058

Hu, B., Guo, H., Zhou, P., & Shi, Z. L. (2021). Characteristics of SARS-CoV-2 and COVID-19. *Nature Reviews Microbiology*, *19*(3), 141–154. https://doi.org/10.1038/s41579-020-00459-7

Hui, D. S., I Azhar, E., Madani, T. A., Ntoumi, F., Kock, R., Dar, O., Ippolito, G., Mchugh, T. D., Memish, Z. A., Drosten, C., Zumla, A., & Petersen, E. (2020). The continuing 2019-nCoV epidemic threat of novel coronaviruses to global health — The latest 2019 novel coronavirus outbreak in Wuhan, China. *International Journal of Infectious Diseases*, *91*, 264–266. https://doi.org/10.1016/j.ijid.2020.01.009

Ismail, B. I., Mostajeran Goortani, E., Ab Karim, M. B., Ming Tat, W., Setapa, S., Luke, J. Y., & Hong Hoe, O. (2016). Evaluation of Docker as Edge computing platform. *ICOS 2015 - 2015 IEEE Conference on Open Systems*, 130–135. https://doi.org/10.1109/ICOS.2015.7377291

Jahangeer, G. S. B., & Rajkumar, T. D. (2021). Early detection of breast cancer using hybrid of series network and VGG-16. *Multimedia Tools and Applications*, *80*(5), 7853–7886. https://doi.org/10.1007/s11042-020-09914-2

K3S. (2021). *K3S GitHub Repository - v1.19.7 Release for ARM64*.

k8s. (2021). *Kubernetes kubectl client for ARM64*.

Kallam, S., Basha, M., Rajput, D., Patan, R., Balamurugan, B., & Basha, S. (2018). *Evaluating the Performance of Deep Learning Techniques on Classification Using Tensor Flow Application*. https://doi.org/10.1109/ICACCE.2018.8441674

Khan, W. Z., Ahmed, E., Hakak, S., Yaqoob, I., & Ahmed, A. (2019). Edge computing: A survey. *Future Generation Computer Systems*, *97*, 219–235. https://doi.org/10.1016/j.future.2019.02.050

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2017). ImageNet Classification with Deep Convolutional Neural Networks. *COMMUNICATIONS OF THE ACM*, *60*(6). https://doi.org/10.1145/3065386

Kubernetes.io. (2020). *Conceptos subyacentes del Cloud Controller Manager*.

Longhorn Authors, & The Linux Foundation. (2021). *Longhorn.io*. https://longhorn.io/

Maciejewski, E. (2021). *TensorFlow Serving on ARM*. https://github.com/emacski/tensorflow-serving-arm

Mart\'\in~Abadi, Ashish~Agarwal, Paul~Barham, Eugene~Brevdo, Zhifeng~Chen, Craig~Citro, Greg~S.~Corrado, Andy~Davis, Jeffrey~Dean, Matthieu~Devin, Sanjay~Ghemawat, Ian~Goodfellow, Andrew~Harp, Geoffrey~Irving, Michael~Isard, Jia, Y., Rafal~Jozefowicz, Lukasz~Kaiser, Manjunath~Kudlur, … Xiaoqiang~Zheng. (2015a). *{TensorFlow}: Architecture*. https://www.tensorflow.org/tfx/serving/architecture/

Mart\'\in~Abadi, Ashish~Agarwal, Paul~Barham, Eugene~Brevdo, Zhifeng~Chen, Craig~Citro, Greg~S.~Corrado, Andy~Davis, Jeffrey~Dean, Matthieu~Devin, Sanjay~Ghemawat, Ian~Goodfellow, Andrew~Harp, Geoffrey~Irving, Michael~Isard, Jia, Y., Rafal~Jozefowicz, Lukasz~Kaiser, Manjunath~Kudlur, … Xiaoqiang~Zheng. (2015b). *{TensorFlow}: Kubernetes Cluster Resolver*. https://www.tensorflow.org/versions/r2.3/api_docs/python/tf/distribute/cluster_resolver/KubernetesClusterResolver

Mell, P. M., & Grance, T. (2011). *The NIST definition of cloud computing*. https://doi.org/10.6028/NIST.SP.800-145

Merkel, D. (n.d.). *Docker: Lightweight Linux Containers for Consistent Development and Deployment*. Retrieved June 3, 2021, from http://www.docker.io

Miller, G., Beckwith, R., Fellbaum, C., Gross, D., & Miller, K. (1991). *Introduction to WordNet: An On-line Lexical Database*. 3*. https://doi.org/10.1093/ijl/3.4.235

Miller, S. E., & Goldsmith, C. S. (2020). Caution in Identifying Coronaviruses by Electron Microscopy. *Journal of the American Society of Nephrology*, *31*(9), 2223 LP – 2224. https://doi.org/10.1681/ASN.2020050755

Minaee, S., Kafieh, R., Sonka, M., Yazdani, S., & Soufi, G. J. (2021). Deep-COVID: Predicting COVID-19 From Chest X-Ray Images Using Deep Transfer Learning. In *Medical Image Analysis*. www.elsevier.com/locate/media

Narayan Das, N., Kumar, N., Kaur, M., Kumar, V., & Singh, D. (2020). Automated Deep Transfer Learning-Based Approach for Detection of COVID-19 Infection in Chest X-rays. *IRBM*. https://doi.org/10.1016/j.irbm.2020.07.001

Narin, A., Kaya, C., & Pamuk, Z. (n.d.). Automatic detection of coronavirus disease (COVID-19) using X-ray images and deep convolutional neural networks. *Pattern Analysis and Applications*, *1*, 3. https://doi.org/10.1007/s10044-021-00984-y

Ning-An, C. (2017). *VMDK to Docker Image*.

nVidia. (n.d.). *Hello Docker*. https://thingsolver.com/hello-docker/

nVidia. (2020). *Jetson Nano User Guide*. https://developer.nVidia.com/embedded/dlc/Jetson_N.

Pääkkönen, P., Pakkala, D., Kiljander, J., & Sarala, R. (2021). Architecture for enabling edge inference via model transfer from cloud domain in a kubernetes environment. *Future Internet*, *13*(1), 1–24. https://doi.org/10.3390/fi13010005

Repository, J. G. (2019). *resizeSwapMemory*.

Sendgrid, I., LTD, V. S. (PTY), & Schulze, W. (2020). *K3S*. https://k3s.io/

Shi, W., Cao, J., Zhang, Q., Li, Y., & Xu, L. (2016). Edge Computing: Vision and Challenges. *IEEE INTERNET OF THINGS JOURNAL*, *3*(5). https://doi.org/10.1109/JIOT.2016.2579198

Sitaula, C., & Hossain, M. B. (2021). *Attention-based VGG-16 model for COVID-19 chest X-ray image classification*. *19*, 2850–2863.

Spiteri, A. (2021). *Longhorn – Distributed Block Storage for Kubernetes Install and Configure*. https://anthonyspiteri.net/longhorn-storage-kubernetes-install-configure/

Storage, P. (n.d.). *2021 Kubernetes Adoption Survey | Pure Storage*.

TensorFlow. (n.d.). Retrieved July 15, 2021, from https://www.tensorflow.org/guide/eager

Wang, S., Hu, Y., & Wu, J. (2020). *KubeEdge.AI: AI Platform for Edge Devices*. http://arxiv.org/abs/2007.09227

Worldometer. (2019). *World of Meter CoVID*. https://www.worldometers.info/coronavirus/

Wu, D., Wu, T., Liu, Q., & Yang, Z. (2020). The SARS-CoV-2 outbreak: What we know. *International Journal of Infectious Diseases*, *94*, 44–48. https://doi.org/10.1016/j.ijid.2020.03.004

Xu, P., Shi, S., & Chu, X. (2017). Performance Evaluation of Deep Learning Tools in Docker Containers. *Proceedings - 2017 3rd International Conference on Big Data Computing and*

Communications, BigCom 2017, 395–403. http://arxiv.org/abs/1711.03386

Anexo I. Repositorio de código fuente

Tanto el código fuente como el dataset están disponibles bajo petición en el repositorio de GitHub localizado en la dirección https://github.com/Kr0n0/TFM-UNIR

Adjuntamos en este Anexo las partes más representativas para una mejor comprensión del trabajo realizado, realzando las partes más significativas en negrita.

A1.1 Conjunto de datos utilizado (*Dataset*)

Figura 73. Conjunto de imágenes de casos positivos de infección empleados

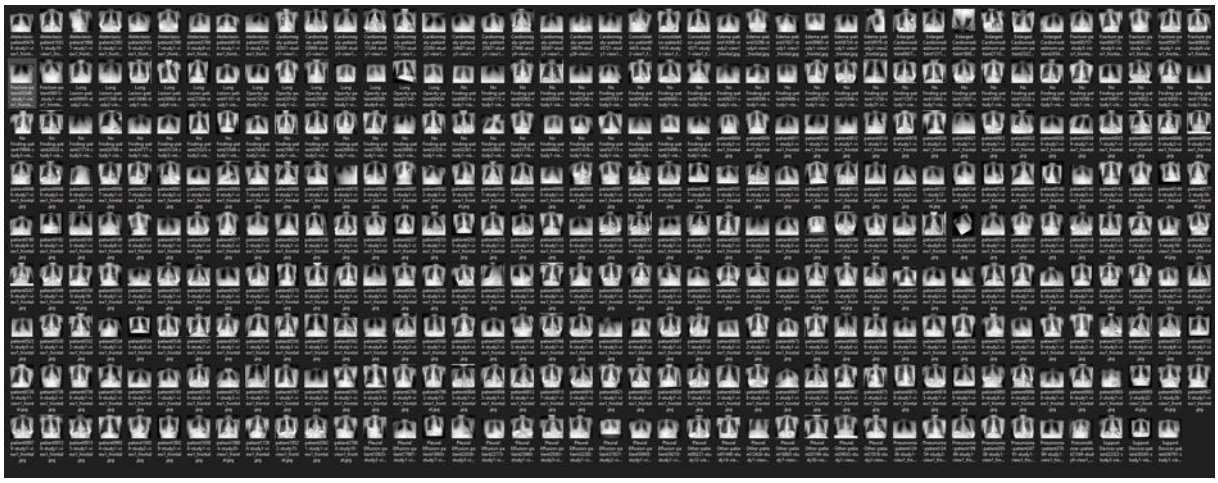


Figura 74. Conjunto de imágenes de casos negativos de infección empleados

A1.2 Algoritmos de entrenamiento

A1.2.1 Desarrollo inicial - Cuaderno Jupyter (covid.ipynb)

Detecting COVID-19 in X-ray images with Keras, TensorFlow, and Deep Learning (dataset deepcovid)

In [1]:

```
# SCIKIT-LEARN
# ============
from sklearn.preprocessing import LabelBinarizer
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix

# TENSORFLOW
# ==========
# Modelo Base - EfficentNetB0
from tensorflow.keras.applications import EfficientNetB0

# Layers a utilizar
from tensorflow.keras.layers import AveragePooling2D
from tensorflow.keras.layers import Dropout
from tensorflow.keras.layers import Flatten
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Input

# Modelos, optimizadores y utilidades
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.preprocessing.image import ImageDataGenerator

# RESTO DE DEPENDENCIAS
# =====================
from imutils import paths
import matplotlib.pyplot as plt
import numpy as np
import argparse
import cv2
import os
```

Carga de imagenes y creación de labels para entrenamiento

In [2]:
```
# grab the list of images in our dataset directory, then initialize
# the list of data (i.e., images) and class images

# Carga de imagenes del dataset
print("[INFO] Cargando imágenes...")
imagePaths = list(paths.list_images("dataset"))
data = []
labels = []

# loop over the image paths
for imagePath in imagePaths:
    # extract the class label from the filename
    label = imagePath.split(os.path.sep)[-2]

    # load the image, swap color channels, and resize it to be a fixed
    # 224x224 pixels while ignoring aspect ratio
    image = cv2.imread(imagePath)
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    image = cv2.resize(image, (224, 224))

    # update the data and labels lists, respectively
    data.append(image)
    labels.append(label)

# convert the data and labels to NumPy arrays while scaling the pixel
# intensities to the range [0, 1]
data = np.array(data) / 255.0
labels = np.array(labels)
```

```
[INFO] Cargando imágenes...
```

In [3]:
```
# perform one-hot encoding on the labels
lb = LabelBinarizer()

labels = lb.fit_transform(labels)
labels = to_categorical(labels)

# partition the data into training and testing splits using 80% of
# the data for training and the remaining 20% for testing
(trainX, testX, trainY, testY) = train_test_split(data, labels,
    test_size=0.20, stratify=labels, random_state=42)

# initialize the training data augmentation object
# trainAug = ImageDataGenerator(    rotation_range=15,fill_mode="nearest")
```

Carga de modelo de ImageNet - EfficentNetB0

In [4]:
```
#baseModel = EfficientNetB0(weights='imagenet', include_top=False,input_tens

```

In [5]:
```
from tensorflow.keras.applications import VGG16

baseModel = VGG16(weights="imagenet", include_top=False,
    input_tensor=Input(shape=(224, 224, 3)))
```

Creación de nueva cabecera de red neuronal y congelamiento del resto

In [6]:
```
# construct the head of the model that will be placed on top of the
# the base model
headModel = baseModel.output
headModel = AveragePooling2D(pool_size=(4, 4))(headModel)
headModel = Flatten(name="flatten")(headModel)
headModel = Dense(64, activation="relu")(headModel)
headModel = Dropout(0.5)(headModel)
headModel = Dense(2, activation="softmax")(headModel)

# place the head FC model on top of the base model (this will become
# the actual model we will train)
model = Model(inputs=baseModel.input, outputs=headModel)

# loop over all layers in the base model and freeze them so they will
# *not* be updated during the first training process
for layer in baseModel.layers:
    layer.trainable = False
```

Transfer Learning - Entrenamiento de nueva cabecera

In [7]:

```
# Parámetros de training
INIT_LR = 1e-3
EPOCHS = 25
#BS = 8
#BS = 4
BS = 1

# Optimizador
opt = Adam(lr=INIT_LR, decay=INIT_LR / EPOCHS)
model.compile(loss="binary_crossentropy", optimizer=opt,
    metrics=["accuracy"])

# Entrenamiento
print("[INFO] training head...")
#H = model.fit_generator(
#    trainAug.flow(trainX, trainY, batch_size=BS),
#    steps_per_epoch=len(trainX) // BS,
#    validation_data=(testX, testY),
#    validation_steps=len(testX) // BS,
#    epochs=EPOCHS,
#    verbose=1)

H = model.fit(
    trainX, trainY, batch_size=BS,
    steps_per_epoch=len(trainX) // BS,
    validation_data=(testX, testY),
    validation_steps=len(testX) // BS,
    epochs=EPOCHS,
    verbose=1)
```

```
curacy: 0.9076 - val_loss: 0.2671 - val_accuracy: 0.8784
Epoch 20/25
294/294 [==============================] - 43s 146ms/step - loss: 0.2116 - ac
curacy: 0.8950 - val_loss: 0.2718 - val_accuracy: 0.9054
Epoch 21/25
294/294 [==============================] - 43s 146ms/step - loss: 0.2217 - ac
curacy: 0.9226 - val_loss: 0.2684 - val_accuracy: 0.9054
Epoch 22/25
294/294 [==============================] - 43s 146ms/step - loss: 0.1674 - ac
curacy: 0.9354 - val_loss: 0.3019 - val_accuracy: 0.8649
Epoch 23/25
294/294 [==============================] - 43s 146ms/step - loss: 0.1573 - ac
curacy: 0.9456 - val_loss: 0.3005 - val_accuracy: 0.8649
Epoch 24/25
294/294 [==============================] - 43s 146ms/step - loss: 0.1524 - ac
curacy: 0.9378 - val_loss: 0.2656 - val_accuracy: 0.9054
Epoch 25/25
294/294 [==============================] - 43s 146ms/step - loss: 0.1701 - ac
curacy: 0.9435 - val_loss: 0.2709 - val_accuracy: 0.9054
```

In [8]:
```
    # make predictions on the testing set
print("[INFO] evaluating network...")
predIdxs = model.predict(testX, batch_size=BS)

# for each image in the testing set we need to find the index of the
# label with corresponding largest predicted probability
predIdxs = np.argmax(predIdxs, axis=1)

# show a nicely formatted classification report
print(classification_report(testY.argmax(axis=1), predIdxs,
    target_names=lb.classes_))
```

```
[INFO] evaluating network...
              precision    recall  f1-score   support

    negativo       0.88      0.95      0.91        37
    positivo       0.94      0.86      0.90        37

    accuracy                           0.91        74
   macro avg       0.91      0.91      0.91        74
weighted avg       0.91      0.91      0.91        74
```

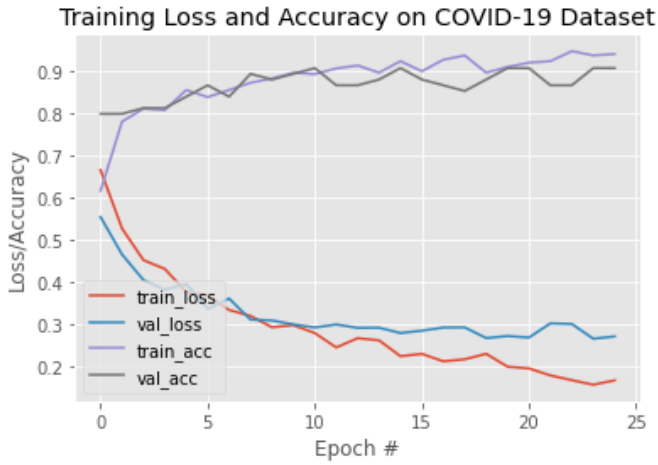
In [9]:
```
# compute the confusion matrix and and use it to derive the raw
# accuracy, sensitivity, and specificity
cm = confusion_matrix(testY.argmax(axis=1), predIdxs)
total = sum(sum(cm))
acc = (cm[0, 0] + cm[1, 1]) / total
sensitivity = cm[0, 0] / (cm[0, 0] + cm[0, 1])
specificity = cm[1, 1] / (cm[1, 0] + cm[1, 1])

# show the confusion matrix, accuracy, sensitivity, and specificity
print(cm)
print("acc: {:.4f}".format(acc))
print("sensitivity: {:.4f}".format(sensitivity))
print("specificity: {:.4f}".format(specificity))
```

```
[[35  2]
 [ 5 32]]
acc: 0.9054
sensitivity: 0.9459
specificity: 0.8649
```

In [10]:
```
# plot the training loss and accuracy
N = EPOCHS
plt.style.use("ggplot")
plt.figure()
plt.plot(np.arange(0, N), H.history["loss"], label="train_loss")
plt.plot(np.arange(0, N), H.history["val_loss"], label="val_loss")
plt.plot(np.arange(0, N), H.history["accuracy"], label="train_acc")
plt.plot(np.arange(0, N), H.history["val_accuracy"], label="val_acc")
plt.title("Training Loss and Accuracy on COVID-19 Dataset")
plt.xlabel("Epoch #")
plt.ylabel("Loss/Accuracy")
plt.legend(loc="lower left")
plt.savefig("plot.png")

```



In [11]:
```
# serialize the model to disk
print("[INFO] saving COVID-19 detector model...")
model.save("covid19.model", save_format="h5")
```

```
[INFO] saving COVID-19 detector model...
```

A1.2.2 Entrenamiento individual en contenedor (covid.py)

```
#!/usr/bin/env python
# coding: utf-8

# Detecting COVID-19 in X-ray images with Keras, TensorFlow, and Deep Learning
# MODO : ONE NODE – STAND-ALONE

# SCIKIT-LEARN
# ============
from sklearn.preprocessing import LabelBinarizer
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix

# TENSORFLOW
# ==========
# Modelo Base - EfficentNetB0
#from tensorflow.keras.applications import EfficientNetB0

# Layers a utilizar
import tensorflow as tf
from tensorflow.keras.layers import AveragePooling2D
from tensorflow.keras.layers import Dropout
from tensorflow.keras.layers import Flatten
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Input

# Modelos, optimizadores y utilidades
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.preprocessing.image import ImageDataGenerator

# RESTO DE DEPENDENCIAS
# =====================
from imutils import paths
import matplotlib.pyplot as plt
import numpy as np
import argparse
import cv2
import os
import json

## Carga de imagenes y creación de labels para entrenamiento

# Carga de imagenes del dataset
print("[INFO] Cargando imagenes...")
imagePaths = list(paths.list_images("dataset"))
data = []
labels = []
```

```
# loop over the image paths
for imagePath in imagePaths:
 # extract the class label from the filename
 label = imagePath.split(os.path.sep)[-2]

 # load the image, swap color channels, and resize it to be a fixed
 # 224x224 pixels while ignoring aspect ratio
 image = cv2.imread(imagePath)
 image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
 image = cv2.resize(image, (224, 224))

 # update the data and labels lists, respectively
 data.append(image)
 labels.append(label)

# convert the data and labels to NumPy arrays while scaling the pixel
# intensities to the range [0, 1]
data = np.array(data) / 255.0
labels = np.array(labels)

print("[INFO] One-hot encoding...")
# perform one-hot encoding on the labels
lb = LabelBinarizer()
labels = lb.fit_transform(labels)
labels = to_categorical(labels)

# partition the data into training and testing splits using 80% of
# the data for training and the remaining 20% for testing
(trainX, testX, trainY, testY) = train_test_split(data, labels,
 test_size=0.20, stratify=labels, random_state=42)

print("[INFO] VGG16...")
from tensorflow.keras.applications import VGG16

baseModel = VGG16(weights="imagenet", include_top=False,
input_tensor=Input(shape=(224, 224, 3)))

# construct the head of the model that will be placed on top of the
# the base model
headModel = baseModel.output
headModel = AveragePooling2D(pool_size=(4, 4))(headModel)
headModel = Flatten(name="flatten")(headModel)
headModel = Dense(64, activation="relu")(headModel)
headModel = Dropout(0.5)(headModel)
headModel = Dense(2, activation="softmax")(headModel)

# place the head FC model on top of the base model (this will become
# the actual model we will train)
model = Model(inputs=baseModel.input, outputs=headModel)
```

```
# loop over all layers in the base model and freeze them so they will
# *not* be updated during the first training process
for layer in baseModel.layers:
 layer.trainable = False

# ## Transfer Learning - Entrenamiento de nueva cabecera
print("[INFO] Training...")
# Parámetros de training
INIT_LR = 1e-3
EPOCHS = 25
#BS = 8
#BS = 4
BS = 2

# Optimizador
opt = Adam(lr=INIT_LR, decay=INIT_LR / EPOCHS)
model.compile(loss="binary_crossentropy", optimizer=opt,
        metrics=["accuracy"])

# Entrenamiento
print("[INFO] training head...")

# Para trainAug - Data augmentation

#H = model.fit_generator(
# trainAug.flow(trainX, trainY, batch_size=BS),
# steps_per_epoch=len(trainX) // BS,
# validation_data=(testX, testY),
# validation_steps=len(testX) // BS,
# epochs=EPOCHS,
#    verbose=1)

H = model.fit(
        trainX, trainY, batch_size=BS,
        steps_per_epoch=len(trainX) // BS,
        validation_data=(testX, testY),
        validation_steps=len(testX) // BS,
        epochs=EPOCHS,
        verbose=1)

# make predictions on the testing set
print("[INFO] evaluating network...")
predIdxs = model.predict(testX, batch_size=BS)

# for each image in the testing set we need to find the index of the
# label with corresponding largest predicted probability
predIdxs = np.argmax(predIdxs, axis=1)

# show a nicely formatted classification report
print(classification_report(testY.argmax(axis=1), predIdxs,
target_names=lb.classes_))
```

```
# compute the confusion matrix and and use it to derive the raw
# accuracy, sensitivity, and specificity
cm = confusion_matrix(testY.argmax(axis=1), predIdxs)
total = sum(sum(cm))
acc = (cm[0, 0] + cm[1, 1]) / total
sensitivity = cm[0, 0] / (cm[0, 0] + cm[0, 1])
specificity = cm[1, 1] / (cm[1, 0] + cm[1, 1])

# show the confusion matrix, accuracy, sensitivity, and specificity
print(cm)
print("acc: {:.4f}".format(acc))
print("sensitivity: {:.4f}".format(sensitivity))
print("specificity: {:.4f}".format(specificity))

# plot the training loss and accuracy
N = EPOCHS
plt.style.use("ggplot")
plt.figure()
plt.plot(np.arange(0, N), H.history["loss"], label="train_loss")
plt.plot(np.arange(0, N), H.history["val_loss"], label="val_loss")
plt.plot(np.arange(0, N), H.history["accuracy"], label="train_acc")
plt.plot(np.arange(0, N), H.history["val_accuracy"], label="val_acc")
plt.title("Training Loss and Accuracy on COVID-19 Dataset")
plt.xlabel("Epoch #")
plt.ylabel("Loss/Accuracy")
plt.legend(loc="lower left")
plt.savefig("plot.png")

# serialize the model to disk in h5 and pb formats
print("[INFO] Saving COVID-19 detector model...")
model.save("covid19.h5", save_format="h5")
model.save("covid19")
```

A1.2.3 Entrenamiento en clúster (MultiWorkerMirroredStrategy)

```
#!/usr/bin/env python
# coding: utf-8

# # Detecting COVID-19 in X-ray images with Keras, TensorFlow, and Deep Learning
# MODO : CLUSTER - MULTIWORKERMIRROREDSTRATEGY

# SCIKIT-LEARN
# ============
from sklearn.preprocessing import LabelBinarizer
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
```

```
# TENSORFLOW
# ==========
# Modelo Base - EfficentNetB0
#from tensorflow.keras.applications import EfficientNetB0

# Layers a utilizar
import tensorflow as tf
from tensorflow.keras.layers import AveragePooling2D
from tensorflow.keras.layers import Dropout
from tensorflow.keras.layers import Flatten
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Input

# Modelos, optimizadores y utilidades
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.preprocessing.image import ImageDataGenerator

# RESTO DE DEPENDENCIAS
# =====================
from imutils import paths
import matplotlib.pyplot as plt
import numpy as np
import argparse
import cv2
import os
import json

# Definición de estrategia distribuida para TensorFlow
strategy = tf.distribute.experimental.MultiWorkerMirroredStrategy()

# ## Carga de imagenes y creación de labels para entrenamiento

# Carga de imagenes del dataset
print("[INFO] Cargando imagenes...")
imagePaths = list(paths.list_images("dataset"))
data = []
labels = []

# loop over the image paths
for imagePath in imagePaths:
 # extract the class label from the filename
 label = imagePath.split(os.path.sep)[-2]

 # load the image, swap color channels, and resize it to be a fixed
 # 224x224 pixels while ignoring aspect ratio
 image = cv2.imread(imagePath)
 image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
 image = cv2.resize(image, (224, 224))
```

```
  # update the data and labels lists, respectively
  data.append(image)
  labels.append(label)

# convert the data and labels to NumPy arrays while scaling the pixel
# intensities to the range [0, 1]
data = np.array(data) / 255.0
labels = np.array(labels)

print("[INFO] One-hot encoding...")
# perform one-hot encoding on the labels
lb = LabelBinarizer()
labels = lb.fit_transform(labels)
labels = to_categorical(labels)

# partition the data into training and testing splits using 80% of
# the data for training and the remaining 20% for testing
(trainX, testX, trainY, testY) = train_test_split(data, labels,
  test_size=0.20, stratify=labels, random_state=42)

print("[INFO] VGG16...")
from tensorflow.keras.applications import VGG16

with strategy.scope():
  baseModel = VGG16(weights="imagenet", include_top=False,
    input_tensor=Input(shape=(224, 224, 3)))

# construct the head of the model that will be placed on top of the
# the base model
  headModel = baseModel.output
  headModel = AveragePooling2D(pool_size=(4, 4))(headModel)
  headModel = Flatten(name="flatten")(headModel)
  headModel = Dense(64, activation="relu")(headModel)
  headModel = Dropout(0.5)(headModel)
  headModel = Dense(2, activation="softmax")(headModel)

# place the head FC model on top of the base model (this will become
# the actual model we will train)
  model = Model(inputs=baseModel.input, outputs=headModel)

# loop over all layers in the base model and freeze them so they will
# *not* be updated during the first training process
  for layer in baseModel.layers:
    layer.trainable = False


# Distributed training
  per_worker_batch_size = 1
  tf_config = json.loads(os.environ['TF_CONFIG'])
  num_workers = len(tf_config['cluster']['worker'])
```

```
 print("======")
 print("Number of workers : ", num_workers)
 print(tf_config)
 print("======")

 global_batch_size = per_worker_batch_size * num_workers

# ## Transfer Learning - Entrenamiento de nueva cabecera
 print("[INFO] Training...")
 # Parámetros de training
 INIT_LR = 1e-3
 EPOCHS = 25
 #BS = 8
 #BS = 4
 #BS = 1
 BS = 2

# Optimizador
 opt = Adam(lr=INIT_LR, decay=INIT_LR / EPOCHS)
 global_batch_size = per_worker_batch_size * num_workers
 model.compile(loss="binary_crossentropy", optimizer=opt,
 metrics=["accuracy"])

 # Entrenamiento
 print("[INFO] training head...")

# Para trainAug - Data augmentation

#H = model.fit_generator(
# trainAug.flow(trainX, trainY, batch_size=BS),
# steps_per_epoch=len(trainX) // BS,
# validation_data=(testX, testY),
# validation_steps=len(testX) // BS,
# epochs=EPOCHS,
#    verbose=1)

 H = model.fit(
 trainX, trainY, batch_size=BS,
 steps_per_epoch=len(trainX) // BS,
 validation_data=(testX, testY),
 validation_steps=len(testX) // BS,
 epochs=EPOCHS,
  verbose=1)

# make predictions on the testing set
 print("[INFO] evaluating network...")
 predIdxs = model.predict(testX, batch_size=BS)

# for each image in the testing set we need to find the index of the
# label with corresponding largest predicted probability
 predIdxs = np.argmax(predIdxs, axis=1)
```

```
# show a nicely formatted classification report
 print(classification_report(testY.argmax(axis=1), predIdxs,
 target_names=lb.classes_))

# compute the confusion matrix and and use it to derive the raw
# accuracy, sensitivity, and specificity
cm = confusion_matrix(testY.argmax(axis=1), predIdxs)
total = sum(sum(cm))
acc = (cm[0, 0] + cm[1, 1]) / total
sensitivity = cm[0, 0] / (cm[0, 0] + cm[0, 1])
specificity = cm[1, 1] / (cm[1, 0] + cm[1, 1])

# show the confusion matrix, accuracy, sensitivity, and specificity
print(cm)
print("acc: {:.4f}".format(acc))
print("sensitivity: {:.4f}".format(sensitivity))
print("specificity: {:.4f}".format(specificity))

# plot the training loss and accuracy
N = EPOCHS
plt.style.use("ggplot")
plt.figure()
plt.plot(np.arange(0, N), H.history["loss"], label="train_loss")
plt.plot(np.arange(0, N), H.history["val_loss"], label="val_loss")
plt.plot(np.arange(0, N), H.history["accuracy"], label="train_acc")
plt.plot(np.arange(0, N), H.history["val_accuracy"], label="val_acc")
plt.title("Training Loss and Accuracy on COVID-19 Dataset")
plt.xlabel("Epoch #")
plt.ylabel("Loss/Accuracy")
plt.legend(loc="lower left")
plt.savefig("plot.png")

# serialize the model to disk in h5 and pb formats
print("[INFO] Saving COVID-19 detector model...")
model.save("covid19.h5", save_format="h5")
model.save("covid19")
```

A1.2.4 Entrenamiento en clúster (KubernetesClusterResolver)

```
#!/usr/bin/env python
# coding: utf-8

# # Detecting COVID-19 in X-ray images with Keras, TensorFlow, and Deep Learning
# MODO : CLUSTER - KUBERNETES

# SCIKIT-LEARN
# ============
from sklearn.preprocessing import LabelBinarizer
```

```
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix

# TensorFlow
import tensorflow as tf
from tensorflow.keras.layers import AveragePooling2D
from tensorflow.keras.layers import Dropout
from tensorflow.keras.layers import Flatten
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Input

# Modelos, optimizadores y utilidades
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.preprocessing.image import ImageDataGenerator

# Kubernetes
from kubernetes import client, config

# RESTO DE DEPENDENCIAS
# =====================
from imutils import paths
import matplotlib.pyplot as plt
import numpy as np
import argparse
import cv2
import os
import json

cluster_mode = os.getenv("CLUSTER_MODE")

print("= Getting cluster setup...")
if cluster_mode == "KUBERNETES":
  print ("Kubernetes cluster mode")
  task_id = (int(os.environ['JOB_COMPLETION_INDEX']))
  print ("Job completion index : {}".format(task_id))
  cluster_resolver = tf.distribute.cluster_resolver.KubernetesClusterResolver(
     {"worker": ["job-name=training-cluster"]})
  cluster_resolver.task_type = "worker"
  cluster_resolver.task_id = int(os.environ['JOB_COMPLETION_INDEX'])
  strategy = tf.distribute.experimental.MultiWorkerMirroredStrategy(
    cluster_resolver=cluster_resolver)
  print ("Kubernetes cluster mode")
if cluster_mode == "DIRECT":
  strategy = tf.distribute.experimental.MultiWorkerMirroredStrategy()
  print ("Direct cluster mode")

# Carga de imagenes del dataset
print("[INFO] Cargando imagenes...")
```

```
imagePaths = list(paths.list_images("dataset"))
data = []
labels = []

# loop over the image paths
for imagePath in imagePaths:
 # extract the class label from the filename
 label = imagePath.split(os.path.sep)[-2]

 # load the image, swap color channels, and resize it to be a fixed
 # 224x224 pixels while ignoring aspect ratio
 image = cv2.imread(imagePath)
 image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
 image = cv2.resize(image, (224, 224))
# image = cv2.resize(image, (299, 299))

 # update the data and labels lists, respectively
 data.append(image)
 labels.append(label)

# convert the data and labels to NumPy arrays while scaling the pixel
# intensities to the range [0, 1]
data = np.array(data) / 255.0
labels = np.array(labels)

print("[INFO] One-hot encoding...")
# perform one-hot encoding on the labels
lb = LabelBinarizer()
labels = lb.fit_transform(labels)
labels = to_categorical(labels)

# partition the data into training and testing splits using 80% of
# the data for training and the remaining 20% for testing
print("[INFO] Train-test-split...")
(trainX, testX, trainY, testY) = train_test_split(data, labels,
 test_size=0.20, stratify=labels, random_state=42)

print("[INFO] VGG16...")
from tensorflow.keras.applications import VGG16
#from tensorflow.keras.applications import Xception

with strategy.scope():
 print("[INFO] Cargando modelo base VGG16...")
 baseModel = VGG16(weights="imagenet", include_top=False,
  input_tensor=Input(shape=(224, 224, 3)))

#  print("[INFO] Cargando modelo base Xception...")
#  basemodel = Xception(include_top=False, weights="imagenet", input_tensor=Input(shape=(299, 299, 3)))

 print("[INFO] Imagenet cargado")
# construct the head of the model that will be placed on top of the
```

```
# the base model
 print("[INFO] Construyendo nueva cabecera...")
 headModel = baseModel.output
 headModel = AveragePooling2D(pool_size=(4, 4))(headModel)
 headModel = Flatten(name="flatten")(headModel)
 headModel = Dense(64, activation="relu")(headModel)
 headModel = Dropout(0.5)(headModel)
 headModel = Dense(2, activation="softmax")(headModel)

# place the head FC model on top of the base model (this will become
# the actual model we will train)
 model = Model(inputs=baseModel.input, outputs=headModel)

# loop over all layers in the base model and freeze them so they will
# *not* be updated during the first training process
 for layer in baseModel.layers:
   layer.trainable = False

# ## Transfer Learning - Entrenamiento de nueva cabecera
 print("[INFO] Training...")
 # Parámetros de training
 INIT_LR = 1e-3
 EPOCHS = 25
 BS = 2

# Optimizador
 opt = Adam(lr=INIT_LR, decay=INIT_LR / EPOCHS)
#  global_batch_size = per_worker_batch_size * num_workers
 model.compile(loss="binary_crossentropy", optimizer=opt,
 metrics=["accuracy"])

# Entrenamiento
print("[INFO] training head...")
H = model.fit(
 trainX, trainY, batch_size=BS,
 steps_per_epoch=len(trainX) // BS,
 validation_data=(testX, testY),
 validation_steps=len(testX) // BS,
 epochs=EPOCHS,
   verbose=1)

# make predictions on the testing set
print("[INFO] evaluating network...")
predIdxs = model.predict(testX, batch_size=BS)

# for each image in the testing set we need to find the index of the
# label with corresponding largest predicted probability
predIdxs = np.argmax(predIdxs, axis=1)

# show a nicely formatted classification report
print(classification_report(testY.argmax(axis=1), predIdxs,
```

```
 target_names=lb.classes_))

# compute the confusion matrix and and use it to derive the raw
# accuracy, sensitivity, and specificity
cm = confusion_matrix(testY.argmax(axis=1), predIdxs)
total = sum(sum(cm))
acc = (cm[0, 0] + cm[1, 1]) / total
sensitivity = cm[0, 0] / (cm[0, 0] + cm[0, 1])
specificity = cm[1, 1] / (cm[1, 0] + cm[1, 1])

# show the confusion matrix, accuracy, sensitivity, and specificity
print(cm)
print("acc: {:.4f}".format(acc))
print("sensitivity: {:.4f}".format(sensitivity))
print("specificity: {:.4f}".format(specificity))

# plot the training loss and accuracy
N = EPOCHS
plt.style.use("ggplot")
plt.figure()
plt.plot(np.arange(0, N), H.history["loss"], label="train_loss")
plt.plot(np.arange(0, N), H.history["val_loss"], label="val_loss")
plt.plot(np.arange(0, N), H.history["accuracy"], label="train_acc")
plt.plot(np.arange(0, N), H.history["val_accuracy"], label="val_acc")
plt.title("Training Loss and Accuracy on COVID-19 Dataset")
plt.xlabel("Epoch #")
plt.ylabel("Loss/Accuracy")
plt.legend(loc="lower left")
plt.savefig("plot.png")

# serialize the model to disk in h5 and pb formats
print("[INFO] Saving COVID-19 detector model...")
model.save("covid19.h5", save_format="h5")
model.save("covid19")
```

A1.3 Algoritmos de inferencia

A1.3.1 Código de request (send_example.py)

```
#!/usr/bin/env python
# coding: utf-8

import cv2
import numpy as np
import sys
import requests
import json
```

```
import os

server_ip = os.environ['SERVER_IP']

image_filename = sys.argv[1]
print("Filename = {}".format(image_filename))

# Image conversion
image_data = []
image = cv2.imread(image_filename)
image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
image = cv2.resize(image, (224, 224))
image_data.append(image)
image_data = np.array(image_data) / 255.0

print("Image Shape = {}".format(image_data.shape))
data = json.dumps({
      'instances': image_data.tolist()
   })
headers = {"content-type": "application/json"}


server_url = 'http://'+ server_ip + ':8501/v1/models/covid19:predict'
print("Requesting to {}".format(server_url))
json_response = requests.post(server_url, data=data, headers=headers)
print("Time : {}".format(json_response.elapsed))
predictions = json.loads(json_response.text)['predictions']
print("Resultado = {}".format(predictions))
print("Clases : 0 - Negativo | 1 - Positivo")
for pred in predictions:
   print("Clase predicha = {}".format(np.argmax(pred)))
```

A1.4 Configuración del sistema

A1.4.1 Script de inicio (startup.sh)

```
#!/bin/bash

# nVidia Jetson System Settings
/root/Scripts/system/setMaxPower.sh

# Network - Disable IPv6
/root/Scripts/network/disable_ipv6.sh

# Kubernetes - Cluster K3S
nohup /root/Scripts/k3s/lanzar_k3s.sh > /var/log/k3s.log 2>&1 &
```

A1.4.2 Gestión del clúster Kubernetes / K3S

- **Lanzar clúster K3S (lanzar_k3s.sh)**

```
#!/bin/bash

MASTER="jetson1"
AGENT="jetson2"

if [ ${HOSTNAME} = ${MASTER} ]
then
  echo "Starting K3S Master (jetson1)..."
  k3s server -c /etc/rancher/k3s/master/config.yaml --kube-apiserver-arg feature-gates=IndexedJob=true --kube-controller-manager-arg feature-gates=IndexedJob=true
elif [ ${HOSTNAME} == ${AGENT} ]
then
  echo "Starting K3S Agent (jetson2)..."
  k3s agent -c /etc/rancher/k3s/agent/config.yaml
fi
```

- **Parar clúster K3s (k3s-killall.sh)**

```
#!/bin/sh
[ $(id -u) -eq 0 ] || exec sudo $0 $@
for bin in /var/lib/rancher/k3s/data/**/bin/; do
    [ -d $bin ] && export PATH=$PATH:$bin:$bin/aux
done
set -x
for service in /etc/systemd/system/k3s*.service; do
    [ -s $service ] && systemctl stop $(basename $service)
done
for service in /etc/init.d/k3s*; do
    [ -x $service ] && $service stop
done
pschildren() {
    ps -e -o ppid= -o pid= | \
    sed -e 's/^\s*//g; s/\s\s*/\t/g;' | \
    grep -w "^$1" | \
    cut -f2
}
pstree() {
    for pid in $@; do
        echo $pid
        for child in $(pschildren $pid); do
            pstree $child
        done
    done
}
killtree() {
    kill -9 $(
```

```
    { set +x; } 2>/dev/null;
    pstree $@;
    set -x;
  ) 2>/dev/null
}
getshims() {
  ps -e -o pid= -o args= | sed -e 's/^ *//; s/\s\s*/\t/;' | grep -w 'k3s/data/[^/]*/bin/containerd-shim' | cut -f1
}
killtree $({ set +x; } 2>/dev/null; getshims; set -x)
do_unmount_and_remove() {
  awk -v path="$1" '$2 ~ ("^" path) { print $2 }' /proc/self/mounts | sort -r | xargs -r -t -n 1 sh -c 'umount "$0" && rm -rf
"$0"'
}
do_unmount_and_remove '/run/k3s'
do_unmount_and_remove '/var/lib/rancher/k3s'
do_unmount_and_remove '/var/lib/kubelet/pods'
do_unmount_and_remove '/run/netns/cni-'
# Remove CNI namespaces
ip netns show 2>/dev/null | grep cni- | xargs -r -t -n 1 ip netns delete
# Delete network interface(s) that match 'master cni0'
ip link show 2>/dev/null | grep 'master cni0' | while read ignore iface ignore; do
  iface=${iface%%@*}
  [ -z "$iface" ] || ip link delete $iface
done
ip link delete cni0
ip link delete flannel.1
rm -rf /var/lib/cni/
iptables-save | grep -v KUBE- | grep -v CNI- | iptables-restore
```

- **Estado clúster K3S (estado_cluster.sh)**

```
#!/bin/bash
export KUBECONFIG=/etc/rancher/k3s/k3s.yaml

kubectl get nodes
kubectl get pods --all-namespaces
```

- **Resinstalación de clúster K3S (reinstall.sh)**

```
#!/bin/bash

/root/Scripts/k3s/k3s-killall.sh
echo "Killing k3s process..."
pkill -9 k3sa
echo "Erasing /var/lib/rancher and /var/lib/kubelet..."
rm -rf /var/lib/rancher
rm -rf /var/lib/kubelet
```

```
rm -rf rm /var/log/k3s.log
echo "Configuring template for nVidia..."
mkdir -p /var/lib/rancher/k3s/agent/etc/containerd/
cp config.toml.tmpl /var/lib/rancher/k3s/agent/etc/containerd/
```

- **Template de configuración para runtime nVidia (config.toml.tmpl)**

```
[plugins.opt]
  path = "/var/lib/rancher/k3s/agent/containerd"

[plugins.cri]
  stream_server_address = "127.0.0.1"
  stream_server_port = "10010"
  enable_selinux = false
  sandbox_image = "docker.io/rancher/pause:3.1"

[plugins.cri.containerd]
  snapshotter = "overlayfs"

[plugins.cri.cni]
  bin_dir = "/var/lib/rancher/k3s/data/ad8f0f93ebb9db5c507884fcdec249d73dd348293dac194e01462c57815cca46/bin"
  conf_dir = "/var/lib/rancher/k3s/agent/etc/cni/net.d"

[plugins.cri.containerd.runtimes.runc]
  runtime_type = "io.containerd.runtime.v1.linux"

[plugins.linux]
  runtime = "nVidia-container-runtime"
```

A1.4.3 Deshabilitar IPv6 (disable_ipv6.sh)

```
#!/bin/bash
sudo sysctl -w net.ipv6.conf.all.disable_ipv6=1
sudo sysctl -w net.ipv6.conf.default.disable_ipv6=1
sudo sysctl -w net.ipv6.conf.lo.disable_ipv6=1
sudo sysctl -w net.ipv6.conf.all.forwarding=1
```

A1.4.4 Gestión de memoria Swap (setSwapMemorySize.sh)

```
#!/bin/bash
# Copyright (c) 2019 Jetsonhacks
# MIT License
# Set the zram swap file size on Jetson Nano
# Default is 2GB
```

```
function usage
{
   echo " usage: ./setSwapFileSize [ [-g #gigabytes ] | [ -m #megabytes ] | [ -h ]"
   echo "   -g #gigabytes - #gigabytes total to use for swap area"
   echo "   -m #megabytes - #megabytes total to use for swap area"
   echo "   -h | --help  This message"
}

# Iterate through command line inputs
if [ "$#" -gt 2 ] || [ "$#" == 0 ]  ; then
  usage
  exit 1
fi
MEGABYTES=""
while [ "$1" != "" ]; do
   case $1 in
      -g | -G )
      MEGABYTES=1000
                        ;;
      -m | -M )
                        MEGABYTES=1
                        ;;

       ( *[!0-9]* | *[0-9] )
                        MEGABYTES=$((MEGABYTES*$1))
                        if [ $MEGABYTES == 0 ] ; then
                          echo "Please specify a number > 0"
                          usage
                          exit 1
      fi
      ;;

      -h | --help )         usage
                        exit
                        ;;

      * )                echo "test"
      usage
                        exit 1
                        ;;
   esac
  shift
done
NRDEVICES=$(grep -c ^processor /proc/cpuinfo | sed 's/^0$/1/')
REQUESTED_BYTES=$((( "$MEGABYTES" / "${NRDEVICES}" )*1024*1024))
# The configuration file is here:
CONFIG_FILE=/etc/systemd/nvzramconfig.sh

if [ -f $CONFIG_FILE ] ; then
```

```
  # we replace the memory request with the desired outcome
  sudo sed -i '/^mem=/c\mem='"$REQUESTED_BYTES"'' $CONFIG_FILE
  echo "Please reboot for changes to take effect."
else
  echo "The swap configuration file does not exist."
  echo "Unable to configure swap memory"
fi
```

A1.4.5 Modo de alto rendimiento (setMaxPower.sh)

```
#!/bin/bash

sudo nvpmodel -m 0
sleep 1
sudo nvpmodel -q
```

A1.4.6 Ruta de exportación de dataset/models via NFS (/etc/exports)

```
# /etc/exports: the access control list for filesystems which may be exported
#   to NFS clients.  See exports(5).
#
/opt/software/data/Pyimagesearch/dataset  *(rw,sync,no_subtree_check,no_root_squash)
/opt/software/tfm/inferencia/servidor
*(rw,sync,no_subtree_check,no_root_squash)
```

A1.5 Configuración de imagen deviceQuery

A1.5.1 Soporte de contenedores

- **Dockerfile (Dockerfile.deviceQuery)**

```
FROM nvcr.io/nVidia/l4t-base:r32.5.0

RUN apt-get update && apt-get install -y --no-install-recommends make g++
COPY ./samples /tmp/samples

WORKDIR /tmp/samples/1_Utilities/deviceQuery
RUN make clean && make

CMD ["./deviceQuery"]
```

- **Script de creación de imagen Docker (build_docker.sh)**

```
#!/bin/bash
NAME=tfm/jetson_devicequery:r32.5.0
docker build -t ${NAME} . -f Dockerfile.deviceQuery
```

- **Script de ejecución usando Docker (run_docker.sh)**

```
#!/bin/bash
NAME=tfm/jetson_devicequery:r32.5.0
docker run --rm --runtime nVidia ${NAME}
```

- **Script de ejecución usando containerd (run_ctr.sh)**

```
#!/bin/bash
IMAGEN=tfm/jetson_devicequery:r32.5.0
export KUBECONFIG=/etc/rancher/k3s/k3s.yaml

ctr i pull docker.io/${IMAGEN}
ctr run --rm --gpus 0 --tty docker.io/${IMAGEN} deviceQuery
```

A1.5.2 Soporte de Kubernetes

- **Fichero de configuración del Pod (pod_deviceQuery.yaml)**

```
apiVersion: v1
kind: Pod
metadata:
  name: devicequery
spec:
  containers:
    - name: nVidia
      image: tfm/jetson_devicequery:r32.5.0
      command: [ "./deviceQuery" ]
```

- **Fichero de configuración del Pod alternativo para ejecución en Jetson2 (pod_deviceQuery_jetson2.yaml)**

```
apiVersion: v1
kind: Pod
metadata:
  name: devicequery
spec:
  nodeName: jetson2
  containers:
    - name: nVidia
      image: tfm/jetson_devicequery:r32.5.0
      command: [ "./deviceQuery" ]
```

- **Script de lanzamiento del Pod (run_pod.sh)**

```
#!/bin/bash
export KUBECONFIG=/etc/rancher/k3s/k3s.yaml

kubectl apply -f pod_deviceQuery.yaml
```

- **Script de parada y borrado del Pod (stop_pod.sh)**

```
#!/bin/bash
export KUBECONFIG=/etc/rancher/k3s/k3s.yaml

kubectl delete -f pod_deviceQuery.yaml
```

- **Script de muestra de logs de ejecución del Pod (logs_pod.sh)**

```
#!/bin/bash
export KUBECONFIG=/etc/rancher/k3s/k3s.yaml

echo "= Describe"
kubectl describe pod devicequery
echo "= LOGs devicequery"
kubectl logs devicequery
```

A1.6 Configuración de imagen TensorFlow base

A1.6.1 Soporte de contenedores

- **Dockerfile (Dockerfile.tf)**

```
FROM nvcr.io/nVidia/l4t-base:r32.5.0

RUN apt-get update -y
RUN apt-get install python3-pip -y
RUN pip3 install -U pip
RUN DEBIAN_FRONTEND=noninteractive apt-get install libhdf5-serial-dev hdf5-tools libhdf5-dev zlib1g-dev zip libjpeg8-dev liblapack-dev libblas-dev gfortran -y
RUN DEBIAN_FRONTEND=noninteractive apt-get install python3 python-dev python3-dev build-essential libssl-dev libffi-dev libxml2-dev libxslt1-dev zlib1g-dev -yq
RUN pip install -U Cython
RUN pip install -U testresources setuptools==49.6.0
RUN pip install numpy==1.16.1 h5py==2.10.0
RUN pip install future==0.18.2 mock==3.0.5 keras_preprocessing==1.1.1 keras_applications==1.0.8 gast==0.2.2 futures protobuf pybind11
```

```
RUN pip3 install -U grpcio absl-py py-cpuinfo psutil portpicker gast astor termcolor wrapt google-pasta
RUN pip3 install --pre --extra-index-url https://developer.download.nVidia.com/compute/redist/jp/v45 tensorflow
```

- **Script de creación de imagen Docker (build_docker.sh)**

```
#!/bin/bash
NAME=tfm/l4t-tensorflow:r32.5.0-tf2.3.1-py3
docker build -t ${NAME} . -f Dockerfile.tf
```

- **Script de ejecución usando Docker (run_docker.sh)**

```
#!/bin/bash
NAME=tfm/l4t-tensorflow:r32.5.0-tf2.3.1-py3
docker run -ti --rm --runtime nVidia ${NAME} /bin/bash
```

A1.6.2 Soporte de Kubernetes

- **Fichero de configuración del Pod (pod_tf.yaml)**

```
apiVersion: v1
kind: Pod
metadata:
  name: tf
spec:
  containers:
    - name: nVidia
      image: tfm/l4t-tensorflow:r32.5.0-tf2.3.1-py3
      command: [ "sleep" ]
      args: [ "1d" ]
```

- **Script de lanzamiento del Pod (run_pod.sh)**

```
#!/bin/bash
export KUBECONFIG=/etc/rancher/k3s/k3s.yaml

kubectl apply -f pod_tf.yaml
```

- **Script de parada y borrado del Pod (stop_pod.sh)**

```
#!/bin/bash
export KUBECONFIG=/etc/rancher/k3s/k3s.yaml

kubectl delete -f pod_tf.yaml
```

- **Script de muestra de logs de ejecución del Pod (logs_pod.sh)**

```
#!/bin/bash
export KUBECONFIG=/etc/rancher/k3s/k3s.yaml

echo "= Describe"
kubectl describe pod tf
echo "= LOGs tf"
kubectl logs tf
```

- **Script de entrada a la Shell del Pod (enter_pod.sh)**

```
#!/bin/bash
export KUBECONFIG=/etc/rancher/k3s/k3s.yaml

kubectl exec -it tf -- /bin/bash
```

A1.7 Configuración de entorno de desarrollo JupyterLab

A1.7.1 Soporte de contenedores

- **Dockerfile (Dockerfile.jupyterlab)**

```
FROM tfm/l4t-tensorflow:r32.5.0-tf2.3.1-py3

ENV DEBIAN_FRONTEND=noninteractive

RUN apt-get update ; apt-get install -y curl
RUN curl -fsSL https://deb.nodesource.com/setup_12.x | bash -
RUN apt-get install -y nodejs
RUN pip3 install jupyter jupyterlab
RUN jupyter labextension install @jupyter-widgets/jupyterlab-manager
RUN pip3 install --upgrade pip
RUN pip3 install imutils matplotlib
RUN pip3 install scikit-learn
RUN pip3 install opencv-python==4.4.0.46
#RUN apt install -y git
#RUN sed -i -e 's/# en_US.UTF-8 UTF-8/en_US.UTF-8 UTF-8/' /etc/locale.gen && \
#    dpkg-reconfigure --frontend=noninteractive locales && \
#    update-locale LANG=en_US.UTF-8
#ENV LANG en_US.UTF-8
#RUN pip install -U git+https://github.com/qubvel/efficientnet

EXPOSE 8888
ENTRYPOINT ["jupyter", "lab", "--ip=0.0.0.0", "--allow-root"]
```

- **Script de creación de imagen Docker (build_docker.sh)**

```
#!/bin/bash
NAME=tfm/l4t-tensorflow-jupyterlab:r32.5.0-tf2.3.1-py3
docker build -t ${NAME} . -f Dockerfile.jupyterlab
```

- **Script de ejecución usando Docker (run_docker.sh)**

```
#!/bin/bash
NAME=tfm/l4t-tensorflow:r32.5.0-tf2.3.1-py3
docker run -ti --rm --runtime nVidia -p 8888:8888 -v /opt/software/data:/dataset tfm/l4t-tensorflow-jupyterlab:r32.5.0-tf2.3.1-py3
```

- **Script de muestra de token login aleatorio de JupyterLab (token_login.sh)**

```
#!/bin/bash
docker logs `docker ps | grep tfm | cut -d ' ' -f 1`
```

A1.7.2 Soporte de Kubernetes

- **Fichero de configuración del Pod (pod_jupyterlab.yaml)**

```
apiVersion: v1
kind: Pod
metadata:
  name: tf-jupyterlab
spec:
  containers:
  - name: jupyterlab
    image: tfm/l4t-tensorflow-jupyterlab:r32.5.0-tf2.3.1-py3
    ports:
    - containerPort: 8888
```

- **Script de lanzamiento del Pod (run_pod.sh)**

```
#!/bin/bash
export KUBECONFIG=/etc/rancher/k3s/k3s.yaml

kubectl apply -f pod_jupyterlab.yaml
```

- **Fichero de configuración del Deployment (deployment_jupyterlab.yaml)**

```
apiVersion: apps/v1
kind: Deployment
metadata:
 name: jupyterlab
spec:
 replicas: 1
 selector:
  matchLabels:
   app: jupyterlab
 template:
  metadata:
   labels:
    app: jupyterlab
  spec:
   containers:
   - name: jupyterlab
    image: tfm/l4t-tensorflow-jupyterlab:r32.5.0-tf2.3.1-py3
    imagePullPolicy: IfNotPresent
    volumeMounts:
    - name: dataset
      mountPath: /dataset
    ports:
    - containerPort: 8888
   volumes:
   - name: dataset
    hostPath:
     path: /opt/software/data
```

- **Script de lanzamiento del Desployment (run_deployment.sh)**

```
#!/bin/bash -e
export KUBECONFIG=/etc/rancher/k3s/k3s.yaml
export DEPLOYMENTFILE=deployment_jupyterlab.yaml

# Deployment
kubectl apply -f ${DEPLOYMENTFILE}
kubectl expose deployments/jupyterlab

# Status
sleep 10
kubectl get svc jupyterlab
kubectl get ep jupyterlab
sleep 5

# Logs de jupyterlab con IP
PODNAME=`kubectl get pods | grep jupyterlab | cut -d' ' -f 1`
kubectl logs ${PODNAME}
```

- **Script de parada y borrado del Deployment (stop_deployment.sh)**

```
#!/bin/bash
export KUBECONFIG=/etc/rancher/k3s/k3s.yaml

kubectl delete deployments.apps jupyterlab
kubectl delete service jupyterlab
sleep 3
```

A1.8 Configuración de imagen con entrenamiento individual

A1.8.1 Soporte de contenedores

- **Dockerfile (Dockerfile.individual)**

```
FROM tfm/l4t-tensorflow:r32.5.0-tf2.3.1-py3
RUN pip3 install --upgrade pip
RUN pip3 install imutils matplotlib
RUN pip3 install scikit-learn
RUN pip3 install opencv-python==4.4.0.46
COPY covid.py dataset /
ENTRYPOINT ["python3", "/covid.py"]
#ENTRYPOINT ["/bin/bash"]
```

- **Script de creación de imagen Docker (build_docker.sh)**

```
#!/bin/bash
NAME=tfm/training_individual:latest
docker build -t ${NAME} . -f Dockerfile.individual
```

- **Script de ejecución usando Docker (run_docker.sh)**

```
#!/bin/bash
NAME=tfm/training_individual:latest
docker run -ti --rm --runtime nVidia -v /opt/software/data:/dataset ${NAME}
```

A1.8.2 Soporte de Kubernetes

- **Fichero de configuración del Pod (training_pod.yaml)**

```
apiVersion: v1
kind: Pod
metadata:
  name: training_individual
spec:
```

```
 containers:
   - name: nVidia
     image: tfm/training_individual:latest
     command: [ "python3" ]
     args: [ "/covid.py" ]
```

- **Script de lanzamiento del Pod (run_pod.sh)**

```
#!/bin/bash
export KUBECONFIG=/etc/rancher/k3s/k3s.yaml

kubectl apply -f training_pod.yaml
```

- **Script de parada y borrado del Pod (stop_pod.sh)**

```
#!/bin/bash
export KUBECONFIG=/etc/rancher/k3s/k3s.yaml

kubectl delete -f training_pod.yaml
```

- **Script de muestra de logs de ejecución del Pod (logs_pod.sh)**

```
#!/bin/bash
export KUBECONFIG=/etc/rancher/k3s/k3s.yaml

echo "= Describe"
kubectl describe pod training_individual
echo "= LOGs training_individual"
kubectl logs training_individual
```

- **Script de entrada a la Shell del Pod (enter_pod.sh)**

```
#!/bin/bash
export KUBECONFIG=/etc/rancher/k3s/k3s.yaml

kubectl exec -it training_individual -- /bin/bash
```

A1.9 Configuración de imagen con entrenamiento en clúster (MultiWorkerMirroredStrategy)

A1.9.1 Soporte de contenedores

- **Dockerfile (Dockerfile.multiworker)**

```
FROM tfm/l4t-tensorflow:r32.5.0-tf2.3.1-py3
```

```
RUN pip3 install --upgrade pip
RUN pip3 install imutils matplotlib
RUN pip3 install scikit-learn
RUN pip3 install opencv-python==4.4.0.46
COPY covid.py /
ENTRYPOINT ["python3", "/covid.py"]
#ENTRYPOINT ["/bin/bash"]
```

- **Script de creación de imagen Docker (build_docker.sh)**

```
#!/bin/bash
NAME=tfm/training_multiworker:latest
docker build -t ${NAME} . -f Dockerfile.multiworker
```

- **Script de ejecución usando Docker multi-nodo (run_docker_multinode.sh)**

```
#!/bin/bash
SOURCE_PATH=`pwd`/src
if [ $HOSTNAME == "jetson1" ]
then
 #Index 0
 echo "Running on jetson1.local..."
 docker run -ti --rm --runtime nVidia -p 12345:12345 \
 --add-host jetson1.local:192.168.0.81 --add-host jetson2.local:192.168.0.82 \
 -v /opt/software/data/Pyimagesearch/dataset:/dataset \
      -e TF_CONFIG='{"cluster": {"worker": ["jetson1.local:12345", "jetson2.local:12345"]}, "task": {"index": 0, "type":
"worker"}}' \
      training_multiworker:latest
else
 #Index 1
 echo "Running on jetson2.local..."
 docker run -ti --rm --runtime nVidia -p 12345:12345 \
 --add-host jetson1.local:192.168.0.81 --add-host jetson2.local:192.168.0.82 \
 -v /opt/software/data/Pyimagesearch/dataset:/dataset \
      -e TF_CONFIG='{"cluster": {"worker": ["jetson1.local:12345", "jetson2.local:12345"]}, "task": {"index": 1, "type":
"worker"}}' \
      training_multiworker:latest
fi
```

A1.10 Configuración de imagen con entrenamiento en clúster Kubernetes (KubernetesClusterResolver)

A1.10.1 Soporte de contenedores

- **Dockerfile (Dockerfile.kubernetes)**

```
FROM tfm/l4t-tensorflow:r32.5.0-tf2.3.1-py3
RUN pip3 install --upgrade pip
RUN pip3 install imutils matplotlib
RUN pip3 install scikit-learn
RUN pip3 install opencv-python==4.4.0.46
RUN pip3 install kubernetes
RUN apt install -y vim
COPY k3s.yaml covid.py /
ENV KUBECONFIG /k3s.yaml
ENV PYTHONUNBUFFERED 1
ENV PYTHONIOENCODING UTF-8
ENTRYPOINT ["python3", "/covid.py"]
#ENTRYPOINT ["/bin/bash"]
```

- **Script de creación de imagen Docker (build_docker.sh)**

```
#!/bin/bash
NAME=tfm/training_cluster:latest
docker build -t ${NAME} . -f Dockerfile.kubernetes
```

- **Script de ejecución usando Docker en modo clúster directo (run_docker_cluster.sh)**

```
#!/bin/bash
SOURCE_PATH=`pwd`/src
if [ $HOSTNAME == "jetson1" ]
then
 #Index 0
 echo "Running on jetson1.local..."
 docker run -ti --rm --runtime nVidia -p 8470:8470 \
 --add-host jetson1.local:192.168.0.81 --add-host jetson2.local:192.168.0.82 \
 -v /opt/software/data/Pyimagesearch/dataset:/dataset \
 -e TF_CONFIG='{"cluster": {"worker": ["jetson1.local:8470", "jetson2.local:8470"]}, "task": {"index": 0, "type": "worker"}}' \
 -e CLUSTER_MODE="DIRECT" \
     tfm/training_cluster:latest:latest
else
 #Index 1
 echo "Running on jetson2.local..."
 docker run -ti --rm --runtime nVidia -p 8470:8470 \
 --add-host jetson1.local:192.168.0.81 --add-host jetson2.local:192.168.0.82 \
 -v /opt/software/data/Pyimagesearch/dataset:/dataset \
 -e TF_CONFIG='{"cluster": {"worker": ["jetson1.local:8470", "jetson2.local:8470"]}, "task": {"index": 1, "type": "worker"}}' \
 -e CLUSTER_MODE="DIRECT" \
     tfm/training_cluster:latest
fi
```

A1.10.2 Soporte de Kubernetes

- **Fichero de configuración del PersistentVolume (PV) y PersistentVolumeClaim (PVC) via NFS (nfs-dataset-pv-pvc.yaml)**

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: kube-nfs-dataset-pv
spec:
  storageClassName: storage-nfs
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteMany
  nfs:
    server: jetson1.local
    path: "/opt/software/data/Pyimagesearch/dataset"
---
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: kube-nfs-dataset-pvc
spec:
  storageClassName: storage-nfs
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 10Gi
```

- **Script de lanzamiento de PV y PVC (lanzar_nfs.sh)**

```
#!/bin/bash

kubectl create -f nfs-dataset-pv-pvc.yaml
```

- **Script de parada y borrado de PV y PVC (parar_nfs.sh)**

```
#!/bin/bash

kubectl delete -f nfs-dataset-pv-pvc.yaml
```

- **Fichero de configuración del Job (training_job.yaml)**

```
apiVersion: batch/v1
kind: Job
```

```
metadata:
 name: 'training-cluster'
spec:
 completions: 2
 parallelism: 2
 completionMode: Indexed
 template:
  metadata:
   labels:
    app: tensorflow
  spec:
   restartPolicy: Never
   hostNetwork: true
   containers:
   - name: 'training-cluster'
    image: 'tfm/training_cluster:latest'
    #imagePullPolicy: Never
    ports:
    - containerPort: 8470
     hostPort: 8470
    env:
    - name: CLUSTER_MODE
     value: "KUBERNETES"
    volumeMounts:
    - name: kube-nfs_dataset-pvc
     mountPath: /dataset
   imagePullSecrets:
   - name: myregistrykey
   volumes:
   - name: kube-nfs_dataset-pvc
    persistentVolumeClaim:
     claimName: kube-nfs_dataset-pvc
   nodeSelector:
    training: enabled
   affinity:
    podAntiAffinity:
     requiredDuringSchedulingIgnoredDuringExecution:
     - labelSelector:
       matchExpressions:
       - key: "app"
        operator: In
        values:
        - tensorflow
      topologyKey: "kubernetes.io/hostname"
```

- **Script de lanzamiento del Job (lanzar_job.sh)**

```
#!/bin/bash
KUBECONFIG=/etc/rancher/k3s/k3s.yaml
```

```
kubectl create -f training_job.yaml
```

- **Script de muestra de logs de ejecución del Job (logs_job.sh)**

```
#!/bin/sh
kubectl describe job training-cluster
```

- **Script de parada y borrado del Job (parar_job.sh)**

```
#!/bin/bash
KUBECONFIG=/etc/rancher/k3s/k3s.yaml

kubectl delete -f training_job.yaml
```

A1.11 Configuración de imagen para inferencia (TensorFlow Serving)

A1.11.1 Soporte de contenedores

- **Script de lanzamiento de servidor (run_tf_server.sh)**

```
#!/bin/bash

MODEL_NAME=covid19
MODEL_DIR=`pwd`/${MODEL_NAME}

if [[ "${HOSTTYPE}" == "aarch64" ]]
then
  IMAGE=emacski/tensorflow-serving:latest-linux_arm64
else
  IMAGE=tensorflow/serving
fi

echo "Running under ${HOSTTYPE} architecture"
echo "TensorFlow Serving Docker Image : ${IMAGE}"

docker run -it --rm -p 8501:8501 \
   -v "${MODEL_DIR}:/models/${MODEL_NAME}" \
   -e MODEL_NAME=${MODEL_NAME} \
   ${IMAGE}
```

- **Dockerfile para cliente (Dockerfile.client)**

```
FROM tfm/l4t-tensorflow:r32.5.0-tf2.3.1-py3

RUN pip3 install --upgrade pip
RUN pip3 install imutils matplotlib
RUN pip3 install scikit-learn
RUN pip3 install opencv-python==4.4.0.46
RUN pip3 install nVidia-pyindex
COPY ejemplos /ejemplos
COPY ejemplos.sh /
COPY send_example.py /
ENTRYPOINT ["/ejemplos.sh"]
```

- **Script de creación de imagen Docker (build_docker.sh)**

```
#!/bin/bash
```

```
NAME=tfm/inference_client:latest
docker build -t ${NAME} . -f Dockerfile.individual
```

- **Script de ejecución usando Docker (run_docker.sh)**

```
#!/bin/bash
NAME=tfm/inference_client:latest
docker run -ti --rm --runtime nVidia --net=host -e SERVER_IP=192.168.0.81 ${NAME}
```

- **Script de lanzamiento de ejemplos (ejemplos.sh)**

```
#!/bin/bash
python3 ./send_example.py ejemplos/positivo.jpg
python3 ./send_example.py ejemplos/negativo.jpg
```

A1.11.2 Soporte de Kubernetes

- **Fichero de configuración del PersistentVolume (PV) y PersistentVolumeClaim (PVC) via NFS (nfs-model-pv-pvc.yaml)**

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: kube-nfs-model-pv
spec:
  storageClassName: storage-nfs
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteMany
  nfs:
    server: jetson1.local
    path: "/opt/software/tfm/inferencia/servidor"
---
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: kube-nfs-model-pvc
spec:
  storageClassName: storage-nfs
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 10Gi
```

- **Script de lanzamiento de PV y PVC (lanzar_nfs.sh)**

```
#!/bin/bash

kubectl create -f nfs-model-pv-pvc.yaml
```

- **Script de parada y borrado de PV y PVC (parar_nfs.sh)**

```
#!/bin/bash

kubectl delete -f nfs-model-pv-pvc.yaml
```

- **Fichero de configuración del Deployment para x86_64 (inference_deployment_x86_64.yaml)**

```
apiVersion: apps/v1
kind: Deployment
metadata:
 labels:
  app.kubernetes.io/name: inference-cluster-deployment
 name: 'inference-cluster'
spec:
 replicas: 2
 selector:
  matchLabels:
   app.kubernetes.io/name: inference-cluster-deployment
 template:
  metadata:
   labels:
    app: tensorflow-serving
    app.kubernetes.io/name: inference-cluster-deployment
  spec:
   containers:
   - name: 'inference-server'
    image: 'emacski/tensorflow-serving:latest-linux_arm64'
    ports:
    - containerPort: 8501
    env:
    - name: MODEL_NAME
      value: "covid19"
    volumeMounts:
    - name: kube-nfs-model-pvc
      mountPath: /models/
   imagePullSecrets:
   - name: myregistrykey
   volumes:
   - name: kube-nfs-model-pvc
    persistentVolumeClaim:
     claimName: kube-nfs-model-pvc
```

```
    nodeSelector:
      training: enabled
    affinity:
      podAntiAffinity:
        requiredDuringSchedulingIgnoredDuringExecution:
        - labelSelector:
            matchExpressions:
            - key: "app"
              operator: In
              values:
              - tensorflow-serving
          topologyKey: "kubernetes.io/hostname"
```

- **Fichero de configuración del Deployment para ARM64 (inference_deployment_arm64.yaml)**

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app.kubernetes.io/name: inference-cluster-deployment
  name: 'inference-cluster'
spec:
  replicas: 2
  selector:
    matchLabels:
      app.kubernetes.io/name: inference-cluster-deployment
  template:
    metadata:
      labels:
        app: tensorflow-serving
        app.kubernetes.io/name: inference-cluster-deployment
    spec:
      containers:
      - name: 'inference-server'
        image: 'emacski/tensorflow-serving:latest-linux_arm64'
        ports:
        - containerPort: 8501
        env:
        - name: MODEL_NAME
          value: "covid19"
        volumeMounts:
        - name: kube-nfs-model-pvc
          mountPath: /models/
      imagePullSecrets:
      - name: myregistrykey
      volumes:
      - name: kube-nfs-model-pvc
        persistentVolumeClaim:
```

```
      claimName: kube-nfs-model-pvc
  nodeSelector:
    training: enabled
  affinity:
    podAntiAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
      - labelSelector:
          matchExpressions:
          - key: "app"
            operator: In
            values:
            - tensorflow-serving
        topologyKey: "kubernetes.io/hostname"
```

- **Script de lanzamiento del Deployment (lanzar_deployment.sh)**

```
#!/bin/bash

if [[ "${HOSTTYPE}" == "aarch64" ]]
then
  YAML=inference_deployment_arm64.yaml
else
  YAML=inference_deployment_x86_64.yaml
fi

kubectl create -f ${YAML}
sleep 5
kubectl expose deployment inference-cluster --type=LoadBalancer --name=inference-cluster-service
```

- **Script de muestra de logs de ejecución del Deployment (logs_deployment.sh)**

```
#!/bin/bash

KUBECONFIG=/etc/rancher/k3s/k3s.yaml

echo "[=] Deployment"
kubectl get deployments inference-cluster
echo "[=] ReplicaSets"
kubectl get replicasets
kubectl describe replicasets
echo "[=] Services"
kubectl get services
  kubectl describe services inference-cluster-service
```

- **Script de parada y borrado del Deployment (parar_deployment.sh)**

```
#!/bin/bash

kubectl delete services inference-cluster-service

if [[ "${HOSTTYPE}" == "aarch64" ]]
then
    YAML=inference_deployment_arm64.yaml
else
    YAML=inference_deployment_x86_64.yaml
fi
kubectl delete -f ${YAML}
```

Anexo. Artículo de investigación

Clúster con hardware embebido, Kubernetes y paralelización de GPUs para Edge AI

Carlos Crisóstomo Vals y Rafael Garrido Viro

Universidad Internacional de la Rioja, Logroño (España)

Fecha 21/09/2021

Resumen

La acuciante necesidad de incrementar la capacidad de cálculo y la rapidez de obtención de resultados, nos llevan a desarrollar una plataforma Edge de inteligencia artificial utilizando un clúster hardware de 2 placas nVidia Jetson Nano, sistemas operativos GNU/Linux Embebidos y tecnologías de Kubernetes y paralelización de GPUs para un caso de uso de visión computacional utilizando redes neuronales.

Para demostrar la viabilidad y mejoras aportadas por este sistema, compararemos la ejecución de uno de los algoritmos solución para la iniciativa planteada por la Comisión Europea en su publicación "AI-ROBOTICS vs COVID-19" [1] relacionada con el de uso de visión computacional y redes neuronales de tipo CNN como solución para la predicción temprana de casos de COVID-19.

Con ello se facilitará y acelerará la detección de casos de COVID-19 mediante redes neuronales convolucionales, sin dependencias externas de ningún tipo, bajo coste y un alto rendimiento y escalabilidad.

Palabras Clave

Cluster, Kubernetes, Edge AI, CNN, COVID-19, Rayos X.

I. Introducción

La situación de pandemia en la que aun nos encontramos alienta al ser humano a buscar soluciones tecnológicas para maximizar el rendimiento de los equipos y optimizar recursos, tiempo y costes.

Uno de los grandes problemas planteados es la detección precoz de infectados por SARs-CoV-2, tal y como se expone en la iniciativa de la Comisión Europea "AI-ROBOTIS vs COVID-19[1]". Para lo cual el uso de algoritmos de IA basados en redes neuronales CNN brinda una oportunidad sin igual de acelerar este proceso.

Cabe reseñar la problemática de gestión de datos de pacientes en la nube con total dependencia de la conectividad, tiempos de respuesta y la complejidad de movimiento de dichos datos.

Todo ello unido a tecnologías como clústering, Docker, Kubernetes, computación en Edge y su implementación en placas de bajo coste y orientadas a este tipo de algoritmia. Nos lleva a plantear una solución de bajo coste y alto rendimiento caracterizado a nivel de arquitectura hardware (Sistemas embebidos, interconexión en clúster, tarjetas nVidia con soporte de GPU), arquitectura software (Clústering, Contenedores, Sistemas GNU/Linux Embebidos), el caso de uso (Inteligencia Artificial, Visión computacional, CNNs, COVID-19) y el producto en sí (Edge, sin dependencias de Cloud, pequeño y de bajo coste)

En pro de evaluar los resultados se realizarán comparativas con diferentes equipos.

II. Estado del Arte

En lo referente a las investigaciones de SARS-CoV-2 es relevante mencionar los estudios de Wu, D and Wu, T "*The* SARS-CoV-2 *outbreak: What we know*."[2] En los que se detalla la epidemiologia, transmisión, manifestación, periodo de incubación, sintomatología, tratamientos y demás detalles de la enfermedad. Por otra parte, tenemos estudios sobre la composición del virus y otros quizá hoy día más relevantes como los de Amanat & Krammer "Descripción general de las plataformas y tecnologías de producción de vacunas para la plataforma SARS-CoV-2" [3] y Estudio de vacunas por nombre para SARS-CoV-2 de Creech et al [4]

De cara a las técnicas de clasificación de imágenes, inevitablemente hemos de comentar la clasificación de imágenes de ImageNet de Krizhevsky et al [5] y el dataset de Deng et al. [6], así como los estudios de Sitaula & Hossain "Attention-based VGG-16 model for COVID-19 chest X-ray image classification",[7].

Unificando los dos puntos anteriores tenemos los siguientes estudios y avances sobre la detección de COVID-19 mediante el análisis de imágenes de rayos-x. Uno los primeros estudios de referencia son "Enfoque automatizado basado en el aprendizaje de transferencia profunda para la detección de la infección por COVID-19 en la radiografía de tórax" (Narayan Das et al., 2020) [8], donde podemos ver una primera aproximación a la detección de neumonías causadas por el virus. más recientemente tenemos la "Detección automática de la enfermedad por coronavirus (COVID-19) mediante imágenes de Rayos-X y redes neuronales convolucionales profundas" (Narin et al., n.d.) [9].

La computación en el borde o Edge computing. Según el NIST (Mell & Grance, 2011) [10] la computación en la nube se define como "un modelo que permite un acceso a la red versátil, cómodo

y a la carta a un conjunto compartido de recursos informáticos configurables (por ejemplo, redes, servidores, almacenamiento, aplicaciones y servicios) que pueden ser rápidamente aprovisionados y liberados con un mínimo esfuerzo de gestión o interacción del proveedor de servicios."

La proliferación de la Internet de las cosas (IoT) y el éxito de los servicios en la nube impulsan un nuevo paradigma informático, que exige el procesamiento de los datos directamente en el propio borde de la red (Shi et al., 2016) [11].

Inteligencia artificial en Edge, en lo referente al hardware necesario cabe mencionar la clasificación del libro "*Edge AI: Convergence of Edge Computing and Artificial Intelligence*" [12]. Centrándonos en el objetivo del proyecto, las tareas de entrenamiento en Edge, la mejor aproximación actualmente es realizar entrenamiento distribuido entre los diferentes equipos Edge. En caso que el número de equipos sea muy grande, se opta por una solución tipo Federated Learning (Bonawitz et al.) [13] donde se centralizan los datos de entrenamiento en una máquina o en un centro de datos.

TensorFlow, plataforma de código fuente abierto para realizar tareas de Machine Learning de manera distribuida a escala. Fue creada en el año 2015 por ingenieros de Google Brain como evolución de su anterior proyecto propietario interno llamado DistBelief. Esta plataforma puede ejecutarse en un conjunto heterogéneo tanto de hardware (CPUs, GPUs y aceleradores como TPUs) como de sistemas operativos (Linux, Windows, macOS y plataformas móviles como Android e iOS) y software (Python, C++, Java...)

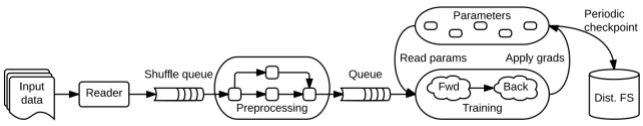


Fig. 1. Diagrama de flujo de un entrenamiento con gráficos de entrada en TensorFlow con estado de pre-procesamiento, entrenamiento y puntos de control.

Docker, la popularización de despliegues de aplicaciones dentro de los llamados contenedores de software, donde el sistema operativo permite aislar componentes de software a nivel de usuario de manera que pueda correr cada componente con un espacio y recursos asignados, independientemente del resto, como podemos observar en la imagen a continuación:

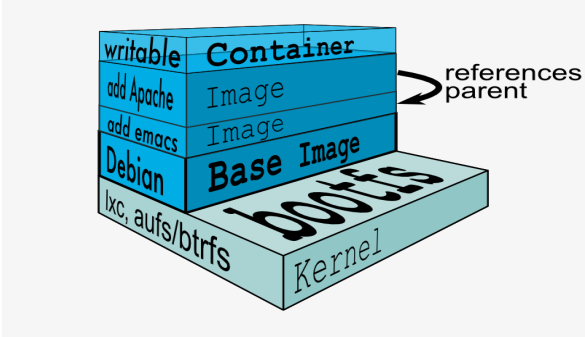


Fig. 2. Esquema de imágenes Docker y contenedores (Ning-An, 2017) [14]

Kubernetes divide sus componentes en dos partes diferenciadas: El plano de control y el plano de ejecución. El plano de control se localiza en el nodo Master del clúster y es el que gestiona todo el propio clúster, mientras que el plano de ejecución se localiza en los nodos Secundarios de ejecución del clúster.

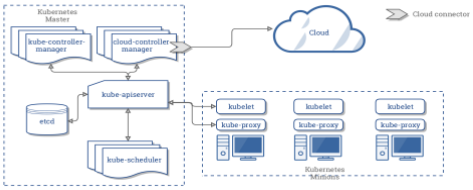


Fig. 3. Kubernetes - Architecture cloud controller (Kubernetes.io, 2020) [15]

En lo referente a KS3 es una distribución de Kubernetes creada por Rancher Labs en el año 2019 como proyecto de código fuente abierto y que actualmente es un proyecto mantenido por la CNCF (Cloud Native Computing Foundation) desde el año 2020. Entre sus particularidades se encuentra:

- Distribución empaquetada como un único binario "k3s"
- Utilización de base de datos sqlite3 como almacenamiento para la base de datos etcd
- Contempla seguridad desde el inicio

Esta distribución permite gestionar todo un clúster completo de Kubernetes desde un único binario de manera encapsulada. Es muy útil para entornos embebidos ya que simplifica mucho la gestión y el despliegue de Kubernetes al estar todo incluido en el mismo paquete. (Sendgrid et al., 2020) [16]

III. Objetivos y Metodología

El objetivo es diseñar una solución de computación en el borde (Edge) de bajo coste en una arquitectura en clúster que permita realizar tareas de inteligencia artificial con utilización de GPUs en paralelo y sin tener dependencias externas. Lo mejorara el rendimiento y la escalabilidad, además de incluir las ventajas de *Edge computing* (menor latencia, procesamiento local de datos, uso de conexiones externas innecesario ya sea por privacidad o requisitos de seguridad)

Como objetivos específicos tenemos:

Demostrar la viabilidad de uso de una infraestructura de clúster con las características software necesarias:

- o Soporte de sistema GNU/Linux completo a nivel de dependencias en arquitecturas ARM de 64 bits.
- o Soporte de bibliotecas estándar de mercado como TensorFlow o Keras para la parte algorítmica.
- o Soporte de infraestructura de contenedores para desarrollo y despliegue (Docker, conteiner) nativos en el clúster.
- o Soporte de infraestructura de clúster de contenedores y orquestación con soluciones de mercado como Kubernetes.
- o Aceleración de los cálculos algorítmicos utilizando GPUs en cada nodo utilizando bibliotecas estándar.
- o Posibilidad de procesamiento en paralelo de tareas de IA, tanto en CPU como en GPUs entre los nodos utilizando las infraestructuras anteriores.

Validar la solución diseñada con el caso de uso planteado.

Valorar diferentes iteraciones y modificaciones en los hiperparámetros utilizados, contrastando las métricas resultantes con el resto de sistemas de la comparativa:

- o nVidia Jetson Nano (Una única placa)
- o Equipo de sobremesa

- o nVidia Jetson Xavier AGX
- o MacBook Pro

En cuanto a la metodología utilizada, partimos de uno de las propuestas solución a la iniciativa de la Comisión Europea "AI-ROBOTIS vs COVID-19" [1], realizando diversas iteraciones y adecuaciones sobre los algoritmos y bases de datos empleadas para obtener un conjunto común de algoritmo – dataset adecuado a los equipos seleccionados para la comparativa.

Dichos equipos se resumen en un equipo de sobremesa, un portátil Macbook pro, 2 placas nVidia Jetson Nano, y una placa nVidia Jetson Xavier NX. Cada uno con su particular S.O. Cabe reseñar que con las placas nVidia Jetson Nano, configuraremos el clúster objeto de este proyecto.

Entrando en detalle de la algoritmia, se realizarán diversas iteraciones, ajustando los valores de los hiperparámetros, con el fin de obtener la versión mas potente y de mejor resultado capaz de ejecutarse en el equipo con menores capacidades, una placa nVidia Jetson Nano. De igual forma, el dataset se ajustará a los mismos parámetros, para poder llevar a cabo una comparativa con las condiciones óptimas

Una vez obtenido el algoritmo y dataset expuesto, se realizarán las ejecuciones pertinentes en los diversos equipos y configuraciones realizando y exponiendo los ajustes pertinentes en caso de ser necesarios.

Tras las ejecuciones llevaremos a cabo al comparativa de cada caso y obtendremos las conclusiones que validaran la propuesta de este proyecto.

IV. Contribución

Este proyecto versa sobre la capacidad de incrementar la productividad de los algoritmos de inteligencia artificial empleando una infraestructura de hardware dedicada (Edge) en clúster junto con las tecnologías de clústering, contenedores y orquestación.

Para ello emplearemos las siguientes plataformas:

- Workstation - PC de sobremesa
- Macbook pro Retina
- 1x nVidia Jetson Nano
- Clúster "Edge AI" (2x nVidia Jetson Nano)
- 1x nVidia Jetson Xavier NX

La contribución respecto al diseño de la solución se basa en la arquitectura en clúster con hardware embebido, contenedores, Kubernetes y paralelización de GPUs implementada sobre la base de dos placas nVidia Jetson Nano, con un incremento teórico del doble de rendimiento (x2).

Inicialmente realizamos el entrenamiento en el equipo de sobremesa y en una de las placas Jetson Nano para verificar y fijar tanto el dataset de inicio, como los hiperparámetros definidos en el algoritmo. Inicialmente partimos del dataset de imágenes de libre acceso de Joseph Paul Cohen, Paul Morrison y Lan Dao [17] pero tras unas primeras iteraciones, observamos la heterogeneidad de las imágenes y tras varias búsquedas optamos por el dataset proporcionado en el estudio Deep-COVID: "Predicting COVID-19 From Chest X-Ray Images Using Deep Transfer Learning" [18].

Ya fijados tanto el dataset y las imágenes a emplear (184/184), como el algoritmo y sus hiperparámetros. Pasamos a construir el clúster tanto a nivel hardware como software.

En el caso de hardware empleamos fuentes de alimentación para las placas, un sistema de refrigeración adicional y un switch GigaLAN con cables Cat 6, y tarjetas mSD U10 de 64 GB.

En cuanto al S.O. empleamos el sistema operativo GNU/Linux proporcionado por el fabricante para estas placas, Linux 4 Tegra 32.5.0, incluido en el JetPack 4.5 oficial de nVidia, con una serie de particularidades:

- Aumento de memoria Swap a 16GB
- Habilitación del perfil de máximo rendimiento (Best performance - MAXN) con la herramienta nvpmodel de nVidia.
- Configuración de red habilitada por DHCP.

Para la gestión de contendores con soporte de GPU disponemos de (docker.io, containerd, nVidia-docker2). Para la gestión del clúster emplearemos k3s y kubectl desplegados en la ruta /usr/local/bin con permisos de ejecución.

- K3S: (K3S, 2021)
- Kubectl: (k8s, 2021)

Tal y como podemos observar en la imagen

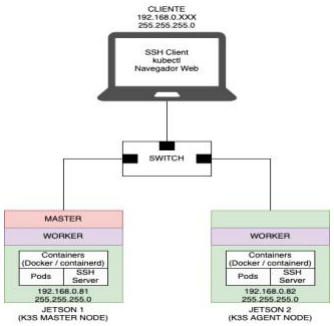


Fig. 4. Arquitectura de comunicaciones del clúster k3s.

V. Evaluación y Resultados

A continuación, detallaremos los resultados obtenidos en las diferentes iteraciones y evaluaremos los resultados obtenidos en el entrenamiento en los diferentes equipos y las métricas de tiempo en cuanto a entrenamiento e inferencia

Evaluación 1 Número de elementos en el dataset

Adecuación del Data set en cuanto al número de elementos. La comparativa siguiente se basa en diferentes resultados de entrenamiento con diferentes conjuntos de datos, definidos por el número de valores positivos, número de valores negativos y el Batch Size correspondiente. Todas estas comparativas han sido realizadas en el equipo sobremesa hasta dar con la mejor combinación para el hardware elegido.

Fig. 5. Comparativa de exactitud y pérdida entre los diferentes datasets.

En la Figura 5 observamos que los niveles de exactitud finales son muy similares en todos los casos salvo en el dataset de 100/100 imágenes positivas/negativas. Entre los 184/184, 184/384 y 184/184 con BS=2 los resultados son muy parecidos, mejorando levemente en cuanto al valor de la función pérdida en el caso del 184/384, pero como hemos comentado anteriormente

elegimos el conjunto 184/184 con BS=2 debido a una menor carga de recursos para los equipos embebidos.

Evaluación 2 Rendimiento de los modelos

En cuanto al resultado de los diferentes entrenamientos en los equipos realizados con el dataset 184/184 BS=2, tenemos los siguientes resultados en la Tabla 1:

TABLA I

RESULTADOS PROMEDIO DE LAS MÉTRICAS DE RENDIMIENTO DE LOS DIFERENTES MODELOS

EQUIPO	Accuracy	Loss	Sensibilidad	Especificidad
AMD Ryzen	90	17	0,89	0,89
Macbook Pro	90	19	0,86	0,94
nVidia Jetson Nano	85	15	0,86	0,83
2x Jetson Nano (gRPC)	80	15	N/A	N/A
2x Jetson Nano (Kubernetes)	85	15	N/A	N/A
Jetson Xavier AGX	86	10	0,91	0,81

TABLA II

LEYENDA DE EQUIPOS Y ETIQUETA PARA LAS GRÁFICAS

EQUIPO	ETIQUETA
AMD Ryzen 1800X con Dataset 184/184 y Batch Size 1	184/184
AMD Ryzen 1800X con Dataset 184/184 y Batch Size 2	184/184 bs2
MacBook Pro 2013	Macbook pro
nVidia Jetson Nano 1 nodo	1 nodo
2x nVidia Jetson Nano (clúster gRPC)	Ini Cont (2 nodos)
2x nVidia Jetson Nano (clúster Kubernetes)	Kub (2 nodos)
nVidia Xavier AGX	Xavier

Respecto al modelo no obtenemos una conclusión clara en cuanto a que haya un modelo mejor que otro, aunque todos los resultados son aceptables. Esto puede ser debido a las diferentes versiones de TensorFlow y a las optimizaciones de las diferentes arquitecturas.

Aunque está fuera del alcance de este TFM, con el objetivo de estudiar las posibles diferencias entre los modelos y poder sacar una conclusión más acertada se plantea un punto nuevo como futura línea de investigación contemplado en las conclusiones.

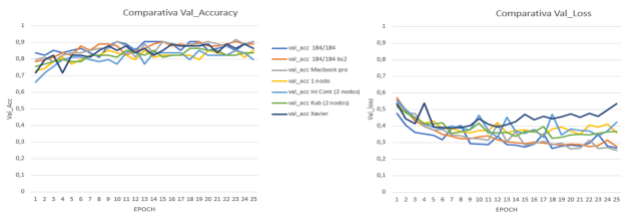


Fig. 1. Comparativa de exactitud y pérdida entre los diferentes equipos.

Evaluación 3 Tiempo de entrenamiento

En función del tiempo de entrenamiento obtenemos los siguientes resultados, según la tabla III:

TABLA III

RESULTADOS PROMEDIO DE ENTRENAMIENTO – SEGUNDOS POR EPOCH

EQUIPO	GPU	RESULTADOS
Macbook Pro 16GB RAM + nVidia 750M	Si	128 sec/epoch
nVidia Jetson Nano	Si	88 sec/epoch
2x nVidia Jetson Nano (gRPC)	Si	47 sec/epoch
2x nVidia Jetson Nano (Kubernetes)	Si	47 sec/epoch
AMD Ryzen 1800X 16GB RAM	No	40 sec/epoch
nVidia Jetson Xavier AGX	Si	9 sec/epoch

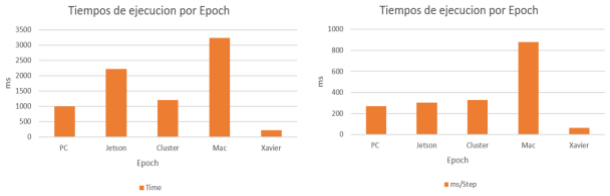


Fig. 6. Tiempos de ejecución por Epoch (ms/epoch y ms/step).

Observamos en la figura 6, una mejora de rendimiento en el entrenamiento en cuanto a emplear dos placas Jetson con respecto a una indicando una mejora en cuanto a los tiempos muy cercanos al rendimiento teórico del clúster, siendo casi la mitad de tiempo (88 segundos contra 47) de entrenamiento por Epoch. De cara al clúster teniendo en cuenta toda la infraestructura de Kubernetes en los dos nodos más la latencia de comunicaciones, vemos que no penaliza apenas el entrenamiento con respecto al caso de clúster sin Kubernetes

Es interesante ver que la arquitectura de la GPU prima mucho en cuanto al rendimiento en este tipo de tareas, teniendo un equipo con una tarjeta gráfica de 2013 (MacBook Pro con GeForce GT 750M, arquitectura Kepler) que tiene peor rendimiento que una tarjeta Jetson Nano (arquitectura Maxwell). También se ve el salto evidente con la nueva arquitectura Volta de la Xavier AGX, que tiene el mejor rendimiento de todas.

Como último caso a tener en cuenta, en el caso de un procesador AMD Ryzen 1800X es capaz de equipararse por puro cálculo de CPU a una tarjeta de las prestaciones del clúster.

Evaluación 4 Tiempo de inferencia

Para este caso lo más relevante es medir el tiempo promedio desde que se hace la solicitud a la API, enviando la imagen para evaluar, hasta que ésta devuelve el resultado con los porcentajes de estimación por clase.

Al ser un proceso puramente de CPU en esta infraestructura (no existen versiones de TensorFlow Serving para GPU ni en Mac OS X ni en arquitectura ARM64), los resultados son esperables en cuanto a la comparativa entre el MacBook Pro y una única tarjeta nVidia Jetson (0,6 vs 2,4 segundos), ya que la CPU del MacBook tiene mucho más rendimiento que la de la tarjeta

TABLA IV

RESULTADOS PROMEDIO DE INFERENCIA – SEGUNDOS POR REQUEST

EQUIPO	ARQUITECTURA	PROMEDIO REQUEST
Macbook Pro 16GB RAM + nVidia 750M	Intel 64 Bits	0,656781 sec
nVidia Jetson Nano	ARM 64 Bits	2,458736 sec
2x nVidia Jetson Nano (Clúster)	ARM 64 Bits	5,288037 sec

En cuanto a la comparativa de una tarjeta y el clúster, el incremento de tiempo es notable en cuanto a la respuesta (más del doble) debido a que la infraestructura en Clúster tiene un balanceador de carga entre las diferentes tarjetas, por lo que la arquitectura es más compleja en cuanto a los pasos que tiene que realizar en las comunicaciones, pero a su vez es más escalable ante avalanchas de peticiones.

VI. DISCUSIÓN

En base a los resultados obtenidos, observamos de cara al número de elementos en el dataset que los niveles de exactitud finales son muy similares en todos los casos salvo en el dataset de 100/100 imágenes positivas/negativas. Para el caso con BS=2 se mejora levemente el valor de la función pérdida.

Respecto al rendimiento entre los diversos modelos estudiados no podemos definir un modelo que resalte en cuanto a resultados.

Basándonos en el tiempo de entrenamiento de cada modelo, podemos confirmar la mejora de rendimiento del clúster respecto a la versión de una única placa. Si bien es relevante que la implementación del clúster con Kubernetes apenas penaliza los tiempos de ejecución. También podemos observar como la evolución de las arquitecturas de GPU favorece una mejora notable en los rendimientos.

Por último, centrándonos en la inferencia observamos un incremento notable en cuanto a la respuesta del Macbook, una placa y 2 placas, lo cual en el caso del clúster es debido al balanceador de carga entre las diferentes tarjetas, y a la complejidad de la arquitectura, pero hay que tener en cuenta su escalabilidad ante avalanchas de peticiones.

VII. CONCLUSIONES

Puesto que el proyecto se basa en crear como prueba de concepto una solución embebida de bajo coste que permitiera realizar tareas de inteligencia artificial con aceleración gráfica, de forma escalable y sin dependencias externas de conectividad.

Debíamos cumplir con una mejora de rendimiento lo suficientemente sustancial que la proporcionada por equipos con soluciones embebidas de bajo coste ya existentes en el mercado, tales como placas SBC con GPU.

También teníamos que proporcionar una solución de coste inferior a las soluciones o sistemas complejos empleados habitualmente para el tipo de tareas de Inteligencia Artificial expuestas en este proyecto, y más concretamente en el área del reconocimiento de imágenes mediante redes neuronales de tipo CNN, y en el ámbito médico-sanitario. Ya que los equipos e instrumentales empleados en dicho ámbito suelen ser de un coste en ocasiones prohibitivo, tales como estaciones de trabajo y o calculo, con varios nodos de proceso grafico dedicado.

Otro objetivo a cumplir, es la escalabilidad, sin la cual este proyecto no tendría mucho sentido, ya que se ha demostrado que mediante el incremento de placas en el clúster podemos incrementar la potencia de calculo y atender a futuras necesidades y problemas más complejos. Dicha escalabilidad también debía cumplir unos requisitos de sencillez e incremento de rendimiento lo suficientemente relevantes como para justificar, de cara a futuro, la inversión en el incremento de dicho clúster, sin que esta fuese excesivamente cara, ni compleja a la hora de gestionarse.

Puesto que necesitamos de un caso lo suficientemente complejo y actual, que mejor que buscar soluciones aplicadas de inteligencia Artificial, a solucionar alguno de los aspectos de la pandemia en la que este envuelto todo el planeta en mayor o menor medida. De forma que práctico el diagnóstico de COVID-19 con imágenes de Rayos X mediante redes neuronales convolucionales, era el caso práctico perfecto para realizar la demostración de viabilidad de nuestra propuesta.

Con este supuesto tenemos, por una parte, la suficiente información (datasets, imágenes y estudios previos) y algoritmia (debida en gran parte a la iniciativa de la Comisión Europea "*AI robotics vs covid 19*" [1]) disponible como para poder obtener datos de otros estudios recientes, como para obtener métricas y resultados en los diversos equipos de la comparativa. Y por otra parte la justificación necesaria como para plantearse el uso de la computación en Edge, debido al uso de datos con un alto nivel de criticidad, sensibilidad y seguridad, ya que son imágenes medicas de pacientes particulares. De esta forma podemos carecer de las necesidades de conectividad externa a internet u otras redes sensibles. Utilizando las conexiones de forma exclusiva entre las propias placas que conforman el clúster, o incluso incluir una conectividad limitada a la máquina que genere las imágenes de rayos -x llegado el caso.

Finalmente, tras realizar el desarrollo del experimento podemos comprobar las premisas planteadas al inicio del mismo, de forma que tanto en las tareas de entrenamiento, como en las tareas de inferencia se han cumplido los objetivos. Si bien hay que reseñar ciertas deficiencias o carencias relacionadas principalmente con la arquitectura ARM64, la falta de soporte de algunos componentes software o la capacidad hardware disponible en algunos equipos, destacando la de las placas Jetson Nano del proyecto.

Destacamos que en las comparativas se observa una mejora sustancial, incluso superior a la esperada, en cuanto a los tiempos para el entrenamiento de casi el triple respecto al equipo MacBook Pro, el doble cuando empleamos a una placa única, y similares respecto al equipo sobremesa con procesador Ryzen, aunque los tiempos están aún muy por encima de los conseguidos en la placa Xavier AGX. En lo que respecta al caso de cálculos de inferencia, los tiempos medidos son de casi el doble a la hora de realizar un diagnóstico de imagen comparado con otros equipos, aunque es esperable debido a la naturaleza del clúster a la hora de balancear las cargas entre los nodos.

Como posibles mejoras o futuras líneas de investigación, nos centramos en dos aspectos. El primero de ellos relacionado con el caso de uso de diagnóstico de COVID-19 y, el segundo con a la infraestructura tecnológica:

Para la mejora en cuanto al diagnóstico de COVID-19se podrían utilizar otros datasets mayores tanto en cantidad como en calidad de las imágenes de cara a mejorar los resultados (bien con TACs de ultra alta resolución y/o con una muestra superior de pacientes e imágenes en diferentes estadios de evolución de la enfermedad). Otra faceta a estudiar, sería la posible comparación y actualización de modelos de redes convolucionales más complejos y avanzados que surjan, siempre y cuando las capacidades hardware lo permitan.

En cuanto la mejorar la infraestructura tecnológica, podríamos intentar adaptar las diferentes herramientas empleadas tanto en el caso base del equipo de sobremesa, como en el Macbook pro, y que no están disponibles (o disponen de una funcionalidad reducida) en la arquitectura ARM64 con soporte de GPU (placas Jetson Nano y Xavier) tales como KubeFlow, TensorFlow Serving o LongHorn y comparar su rendimiento con las utilizadas en este trabajo.

Para finalizar consideramos interesante un estudio de disponibilidad y carga de cara a solicitudes masivas en el clúster Kubernetes para poder valorar los tiempos de respuesta y la estabilidad con el balanceador de carga respecto a los otros casos estudiados.

REFERENCIAS

[1] European Commission AI Alliance. Join AI robotics vs covid 19 initiatives. https://ec.europa.eu/futurium/en/ai-robotics-vs-covid19/join-ai-robotics-vs-covid-19-initiative-european-ai-alliance

[2] Wu, D., Wu, T., Liu, Q., & Yang, Z. (2020). The SARS-CoV-2 outbreak: What we know. *International Journal of Infectious Diseases*, *94*, 44–48. https://doi.org/10.1016/j.ijid.2020.03.004

[3] Amanat, F., & Krammer, F. (2020). SARS-CoV-2 Vaccines: Status Report. *Immunity*, *52*(4), 583–589. https://doi.org/10.1016/j.immuni.2020.03.007

[4] Creech, C. B., Walker, S. C., & Samuels, R. J. (2021). SARS-CoV-2 Vaccines. *JAMA - Journal of the American Medical Association*, *325*(13), 1318–1320. https://doi.org/10.1001/jama.2021.3199

[5] Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2017). ImageNet Classification with Deep Convolutional Neural Networks. COMMUNICATIONS OF THE ACM, 60(6). https://doi.org/10.1145/3065386

[6] Deng, J., Dong, W., Socher, R., Li, L., Li, K., & Fei-Fei, L. (2009). ImageNet: A large-scale hierarchical image database. 2009 IEEE Conference on Computer Vision and Pattern Recognition, 248–255. https://doi.org/10.1109/CVPR.2009.5206848

[7] Sitaula, C., & Hossain, M. B. (2021). Attention-based VGG-16 model for COVID-19 chest X-ray image classification. 19, 2850–2863.

[8] Narayan Das, N., Kumar, N., Kaur, M., Kumar, V., & Singh, D. (2020). Automated Deep Transfer Learning-Based Approach for Detection of COVID-19 Infection in Chest X-rays. IRBM. https://doi.org/10.1016/j.irbm.2020.07.001

[9] Narin, A., Kaya, C., & Pamuk, Z. (n.d.). Automatic detection of coronavirus disease (COVID-19) using X-ray images and deep convolutional neural networks. Pattern Analysis and Applications, 1, 3. https://doi.org/10.1007/s10044-021-00984-y

[10] Mell, P. M., & Grance, T. (2011). *The NIST definition of cloud computing*. https://doi.org/10.6028/NIST.SP.800-145

[11] Shi, W., Cao, J., Zhang, Q., Li, Y., & Xu, L. (2016). Edge Computing: Vision and Challenges. IEEE INTERNET OF THINGS JOURNAL, 3(5). https://doi.org/10.1109/JIOT.2016.2579198

[12] "*Edge AI: Convergence of Edge Computing and Artificial Intelligence* - Xiaofei Wang, Yiwen Han, Victor C. M. Leung, Dusit Niyato, Xueqiang Yan, Xu Chen - Google Libros, n.d.

[13] Bonawitz, K., Eichner, H., Grieskamp, W., Huba, D., Ingerman, A., Ivanov, V., Kiddon, C., Konečný, J., Mazzocchi, S., McMahan, H. B., Van Overveldt, T., Petrou, D., Ramage, D., & Roselander, J. (2019). Towards Federated Learning at Scale: System Design. http://arxiv.org/abs/1902.01046

[14] Ning-An, C. (2017). VMDK to Docker Image.

[15] Kubernetes.io. (2020). Conceptos subyacentes del Cloud Controller Manager.

[16] Sendgrid, I., LTD, V. S. (PTY), & Schulze, W. (2020). K3S. https://k3s.io/

[17] Cohen, J. P., Morrison, P., Dao, L., Roth, K., Duong, T. Q., & Ghassemi, M. (2020). *COVID-19 Image Data Collection: Prospective Predictions Are the Future*. http://arxiv.org/abs/2006.11988

[18] Minaee, S., Kafieh, R., Sonka, M., Yazdani, S., & Soufi, G. J. (2021). Deep-COVID: Predicting COVID-19 From Chest X-Ray Images Using Deep Transfer Learning. In Medical Image Analysis. www.elsevier.com/locate/media